Principles of Robot Autonomy: Homework 5

[Chetan Reddy] [Narayanaswamy]

12/03/2024

Other students worked with: None
Time spent on homework: 7 hours

Problem 1:

Part (i)

The state vector under consideration is x_t^m which represents the global 2D positions of the landmarks. Since, these are stationary objects, the value of x_t^m remains constant with time t. This implies that **A is an 8x8 identity matrix**. Therefore:

$$x_{t+1}^m = x_t^m$$
$$x_{t+1}^m = Ax_t^m$$

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

Part (ii)

The measurement vector in this case is directly given by the state vector x_t^m. Therefore **C is also an 8x8 identity matrix**.

$$z_t = x_t^m$$
$$z_t = Cx_t^m$$

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

Part (iii)

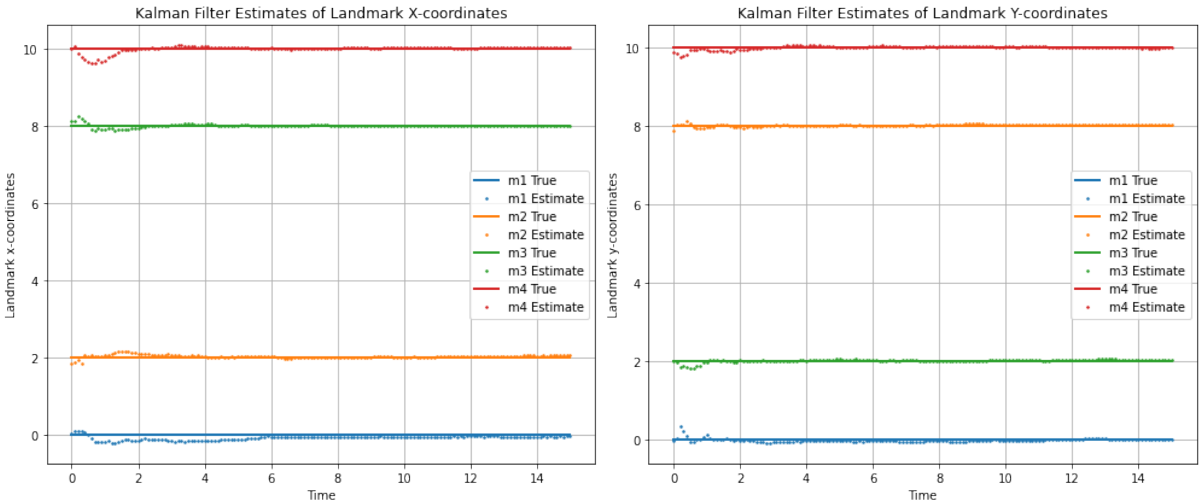


Figure 1: Kalman Filter Estimates of the Landmarks over time

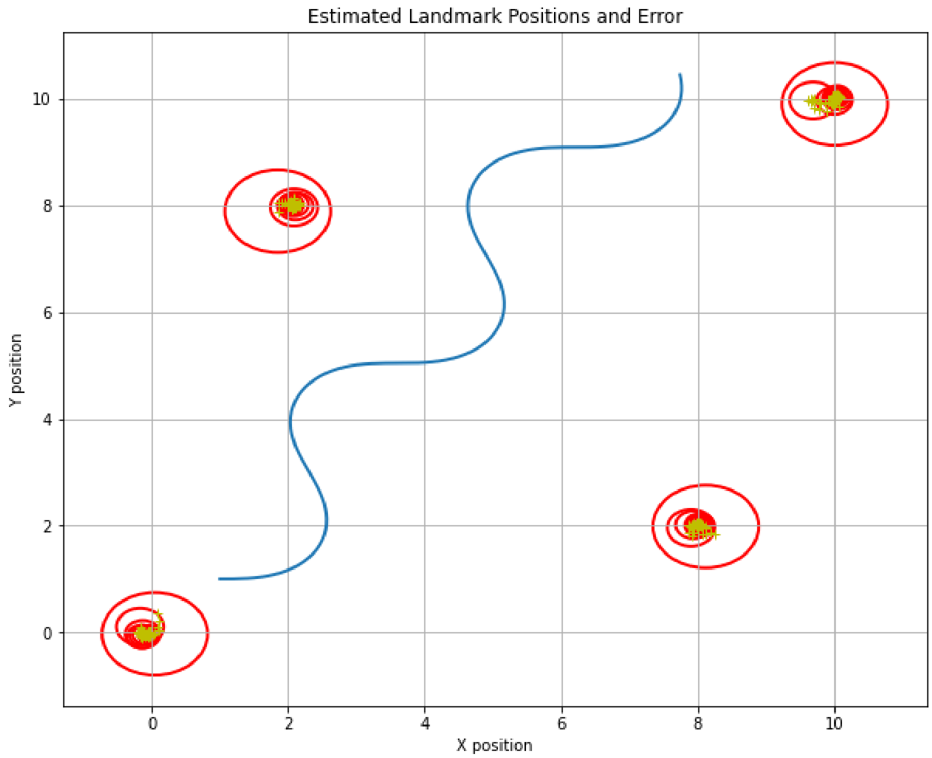


Figure 2: Estimated Landmarks positions with Errors given by the Ellipsoids

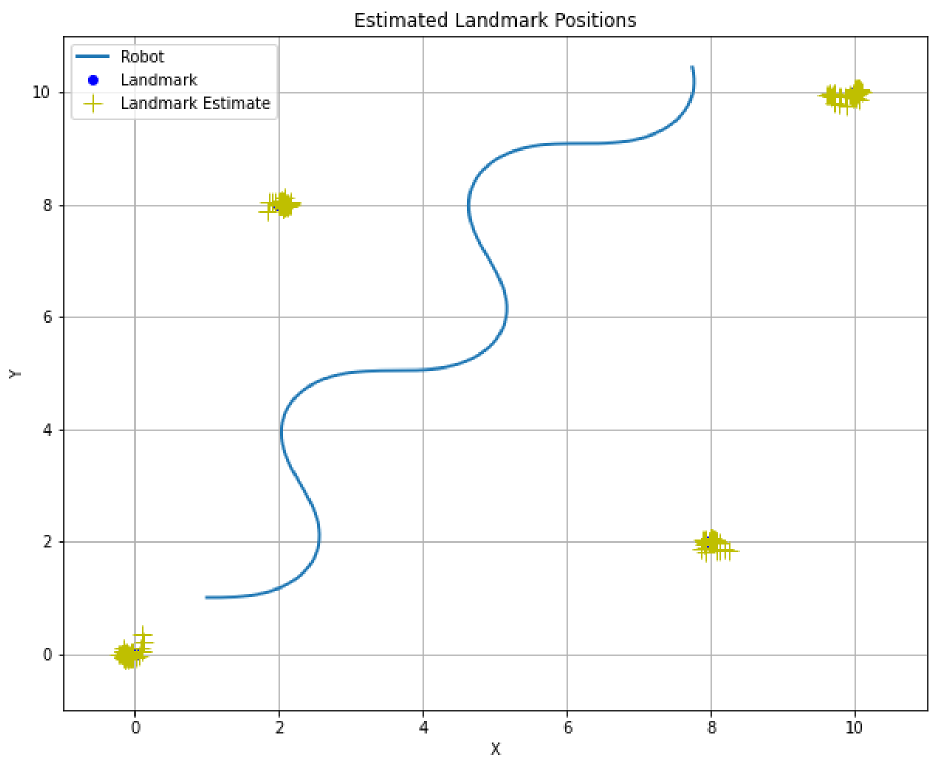


Figure 3: Estimated Landmarks positions

Part (iv)

a) The process noise Q is assumed to be zero. This means that the landmarks are **completely stationary** during the entire time.

b) For Kalman Filters in general, we assume that the process is linear and that the noise follows a Gaussian Distribution.

Problem 2:

Part (i)

The state dynamics is given by:

$$\begin{bmatrix} x^r_{t+1} \\ y^r_{t+1} \\ \theta^r_{t+1} \end{bmatrix} = \begin{bmatrix} x^r_t + V_t \cos(\theta^r_t)\Delta t \\ y^r_t + V_t \sin(\theta^r_t)\Delta t \\ \theta^r_t + \omega_t \Delta t \end{bmatrix} + \begin{bmatrix} w^x_t \\ w^y_t \\ w^\theta_t \end{bmatrix}.$$

which is compactly represented by the following:

$$\mathbf{x^r_t} = \mathbf{g}(\mathbf{u_t}, \mathbf{x^r_{t-1}}) + \mathbf{w_t}$$

To find the jacobian, we take the partial derivatives of each element in $\mathbf{g}$ wrt each of the states and store it in a matrix. Therefore **G is a 3x3 matrix** as follows:

$$\mathbf{G}(\mathbf{x^r_t}) = \begin{bmatrix} 1 & 0 & -V_t \sin(\theta^r_{t-1})\Delta t \\ 0 & 1 & V_t \cos(\theta^r_{t-1})\Delta t \\ 0 & 0 & 1 \end{bmatrix}.$$

Part (ii)

The measurement equation for landmark i is given by:

$$\mathbf{z}^{mi}_t = \begin{bmatrix} \hat{x}^{mi}_t \\ \hat{y}^{mi}_t \end{bmatrix} = \begin{bmatrix} \cos(\theta^r_t) & \sin(\theta^r_t) \\ -\sin(\theta^r_t) & \cos(\theta^r_t) \end{bmatrix} \left(\begin{bmatrix} x^{mi}_t \\ y^{mi}_t \end{bmatrix} - \begin{bmatrix} x^r_t \\ y^r_t \end{bmatrix} \right) + \mathbf{v}^{mi}_t,$$

$$\mathbf{z}^{mi}_t = \begin{bmatrix} \hat{x}^{mi}_t \\ \hat{y}^{mi}_t \end{bmatrix} = \begin{bmatrix} \cos(\theta^r_t)(x^{mi}_t - x^r_t) + \sin(\theta^r_t)(y^{mi}_t - y^r_t) \\ -\sin(\theta^r_t)(x^{mi}_t - x^r_t) + \cos(\theta^r_t)(y^{mi}_t - y^r_t) \end{bmatrix} + \begin{bmatrix} v^{mi,x}_t \\ v^{mi,y}_t \end{bmatrix}.$$

$$\mathbf{H_i} = \begin{bmatrix} -\cos(\theta^r_t) & -\sin(\theta^r_t) & (x^{mi}_t - x^r_t)\sin(\theta^r_t) + (y^{mi}_t - y^r_t)\cos(\theta^r_t) \\ \sin(\theta^r_t) & -\cos(\theta^r_t) & -(x^{mi}_t - x^r_t)\cos(\theta^r_t) + (y^{mi}_t - y^r_t)\sin(\theta^r_t) \end{bmatrix}$$

$$\mathbf{H}(\mathbf{x^r_t}) = \begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ H_4 \end{bmatrix}$$

H is a 8x3 matrix as given below (Note that $x = x^r_t, y = y^r_t, \theta = \theta^r_t$)

$$\mathbf{H} = \begin{bmatrix} -\cos(\theta) & -\sin(\theta) & (x^{m1}_t - x)\sin(\theta) + (y_1 - y)\cos(\theta) \\ \sin(\theta) & -\cos(\theta) & -(x^{m1}_t - x)\cos(\theta) + (y_1 - y)\sin(\theta) \\ -\cos(\theta) & -\sin(\theta) & (x^{m2}_t - x)\sin(\theta) + (y_2 - y)\cos(\theta) \\ \sin(\theta) & -\cos(\theta) & -(x^{m2}_t - x)\cos(\theta) + (y_2 - y)\sin(\theta) \\ -\cos(\theta) & -\sin(\theta) & (x^{m3}_t - x)\sin(\theta) + (y_3 - y)\cos(\theta) \\ \sin(\theta) & -\cos(\theta) & -(x^{m3}_t - x)\cos(\theta) + (y_3 - y)\sin(\theta) \\ -\cos(\theta) & -\sin(\theta) & (x^{m4}_t - x)\sin(\theta) + (y_4 - y)\cos(\theta) \\ \sin(\theta) & -\cos(\theta) & -(x^{m4}_t - x)\cos(\theta) + (y_4 - y)\sin(\theta) \end{bmatrix}$$

Part (iii)

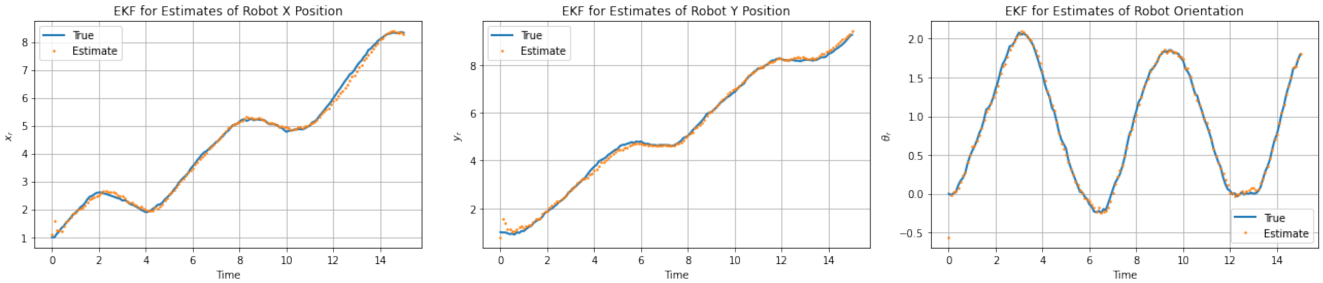
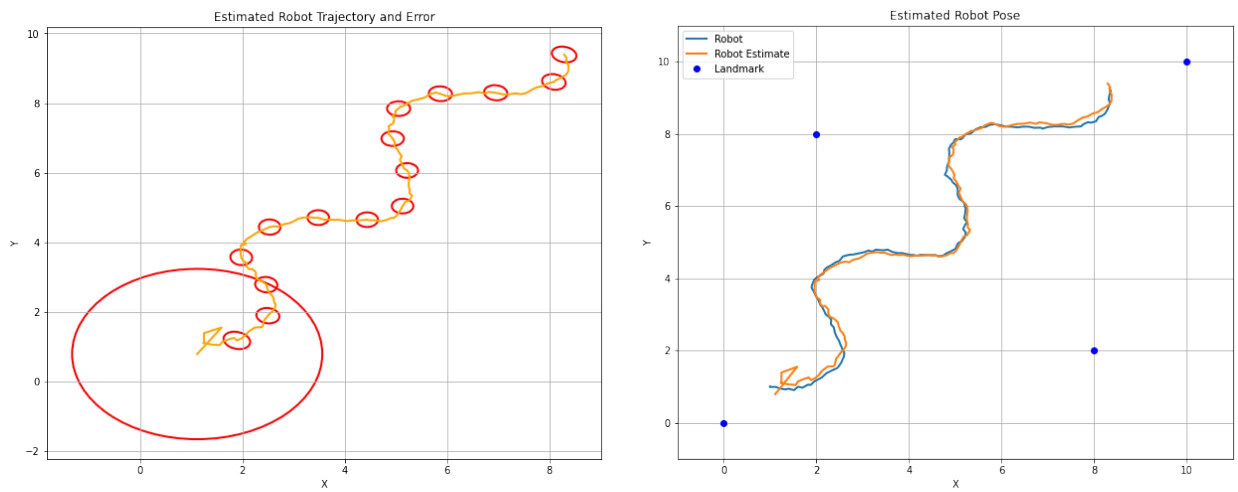


Figure 4: EKF Estimates for x_t^r over t



(a) (b)

Figure 5: Robot Trajectory with EKF

Part (iv)

EKF relies on a linear approximation at a particular point. This can fail in multiple scenarios:

- If the turtlebot is not close to the linearising point. This can happen if it takes sharp turns or is given high control inputs.
- Secondly, EKF fails if either the process or measurements noises are not gaussian or if the initial belief is not gaussian.

One strategy to handle this is to use an Unscented Kalman Filter where we do not linearise about a single point like EKF. Instead, we carefully select multiple points and fit a gaussian after transforming them.

Problem 3:

Part (i)

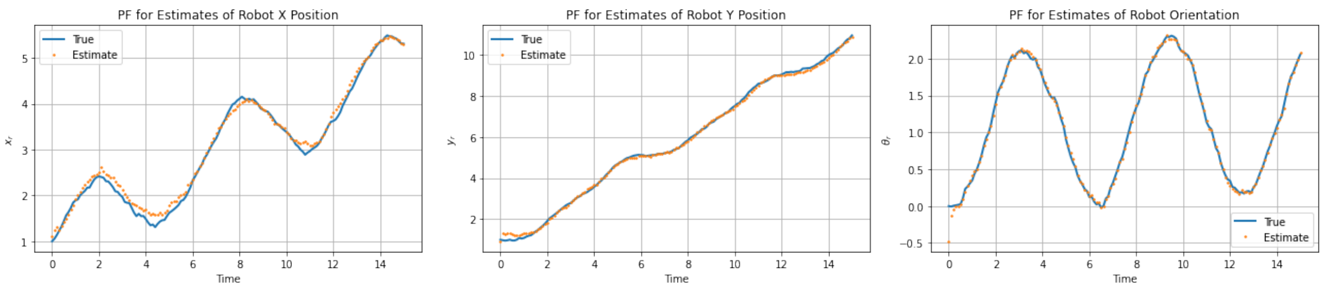
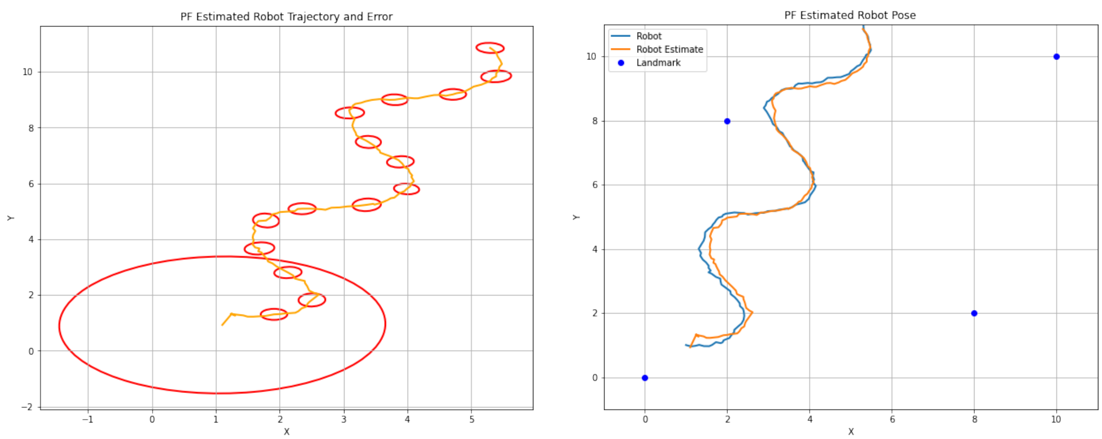


Figure 6: PF Estimates for x_t^r over t



(a) (b)

Figure 7: Robot Trajectory with PF

Part (ii)

Both the plots look very similar but few minor differences between Particle Filter (PF) and Extended Kalman Filter (EKF) from the plots include (however, these can be because of random noise)

- The particle filter is better in the initial few time steps in its estimation
- The ellipsoids are more stretched in PF compared to EKF

Part (iii)

a) FALSE, particle filter does not impose any condition on the type of measurement or dynamic model

b) FALSE, Since the particles are samples at every iteration, they are stochastic

c) TRUE, Unimodal distributions can be fitted with a Gaussian and using Kalman filters is more efficient.

Problem 4:

Part (i)

G is a 11x11 matrix given by:

$$\mathbf{G_r} = \begin{bmatrix} 1 & 0 & -V_t \sin(\theta^r_{t-1})\Delta t \\ 0 & 1 & V_t \cos(\theta^r_{t-1})\Delta t \\ 0 & 0 & 1 \end{bmatrix}.$$

$$\mathbf{G} = \begin{bmatrix} G_r & O \\ & \\ O & I_{8x8} \end{bmatrix}.$$

Part (ii)

$$\mathbf{H}(\theta) = \begin{bmatrix} \cos(\theta^r_t) & \sin(\theta^r_t) \\ -\sin(\theta^r_t) & \cos(\theta^r_t) \end{bmatrix}$$

$$\mathbf{H_i} = \begin{bmatrix} -\cos(\theta^r_t) & -\sin(\theta^r_t) & (x^{mi}_t - x^r_t)\sin(\theta^r_t) + (y^{mi}_t - y^r_t)\cos(\theta^r_t) \\ \sin(\theta^r_t) & -\cos(\theta^r_t) & -(x^{mi}_t - x^r_t)\cos(\theta^r_t) + (y^{mi}_t - y^r_t)\sin(\theta^r_t) \end{bmatrix}$$

H is a 8x11 matrix given by

$$\mathbf{H} = \begin{bmatrix} H_1 & H(\theta) & O & O & O \\ H_2 & O & H(\theta) & O & O \\ H_3 & O & O & H(\theta) & O \\ H_4 & O & O & O & H(\theta) \end{bmatrix}$$

Part (iii)

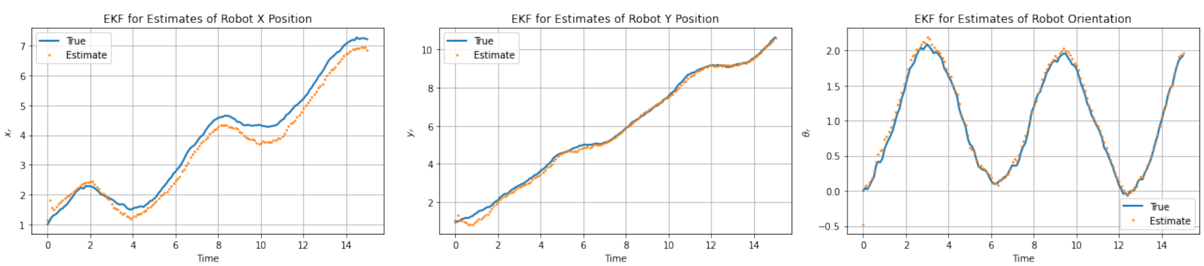


Figure 8: Caption

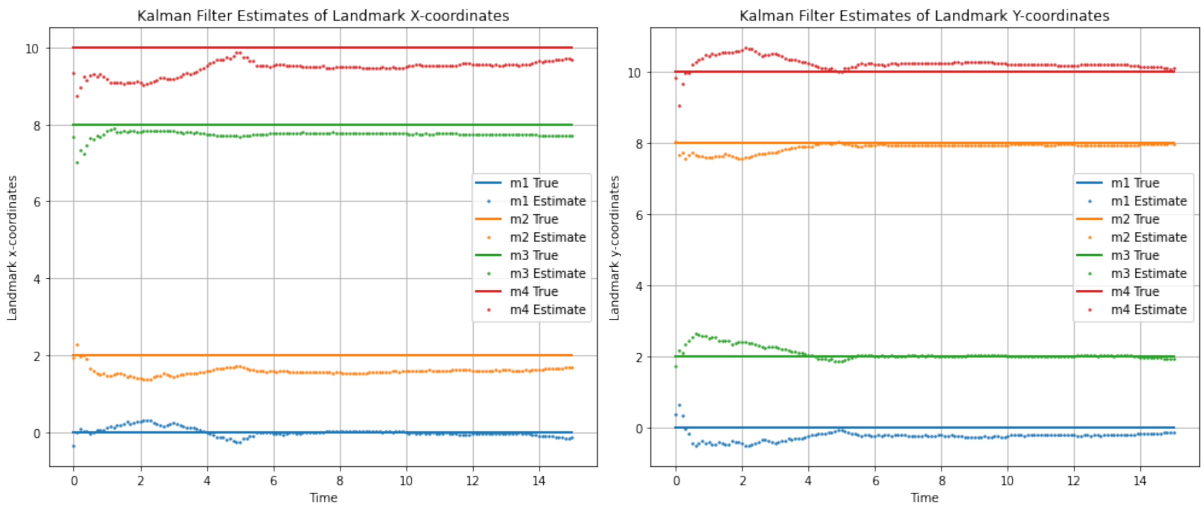


Figure 9: Caption

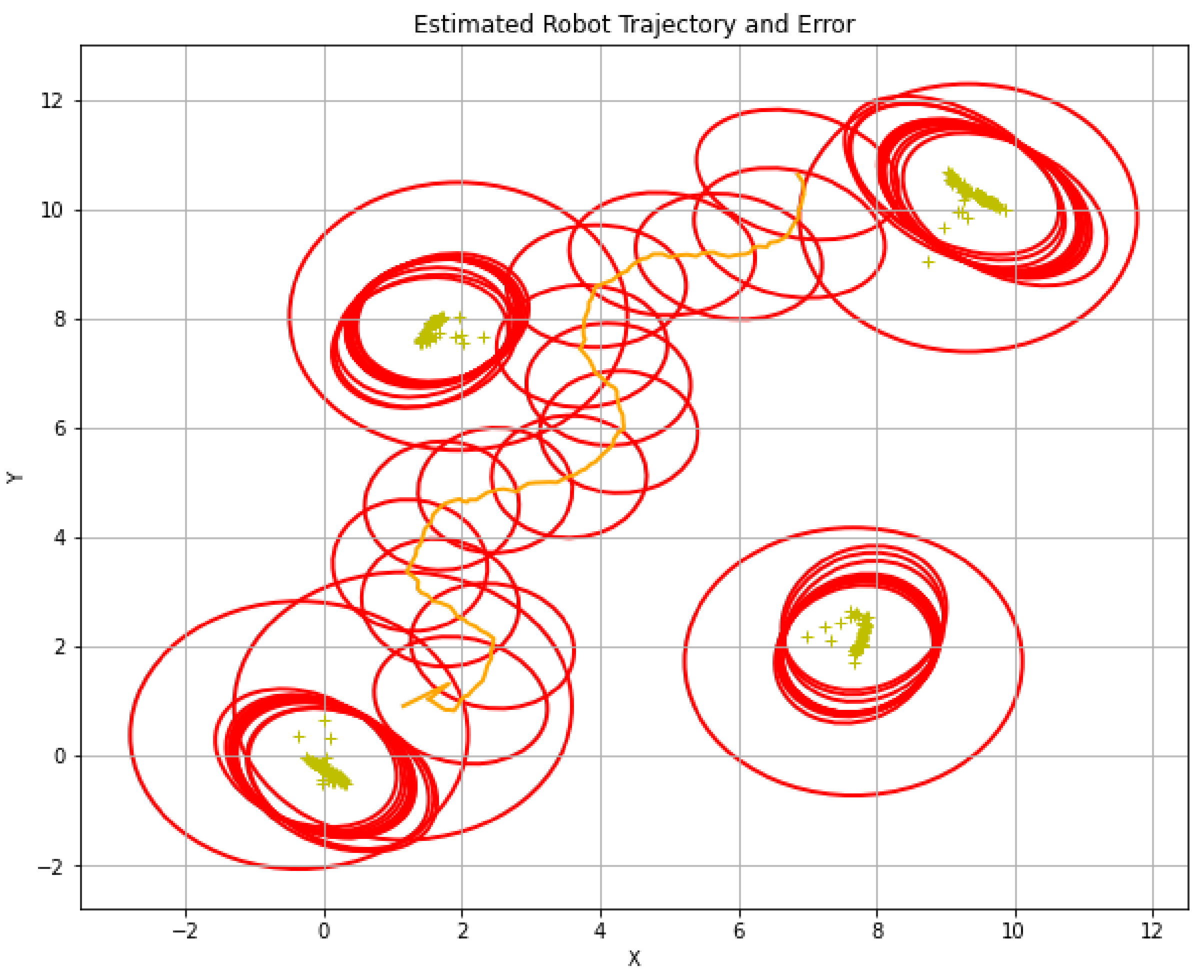


Figure 10: Caption

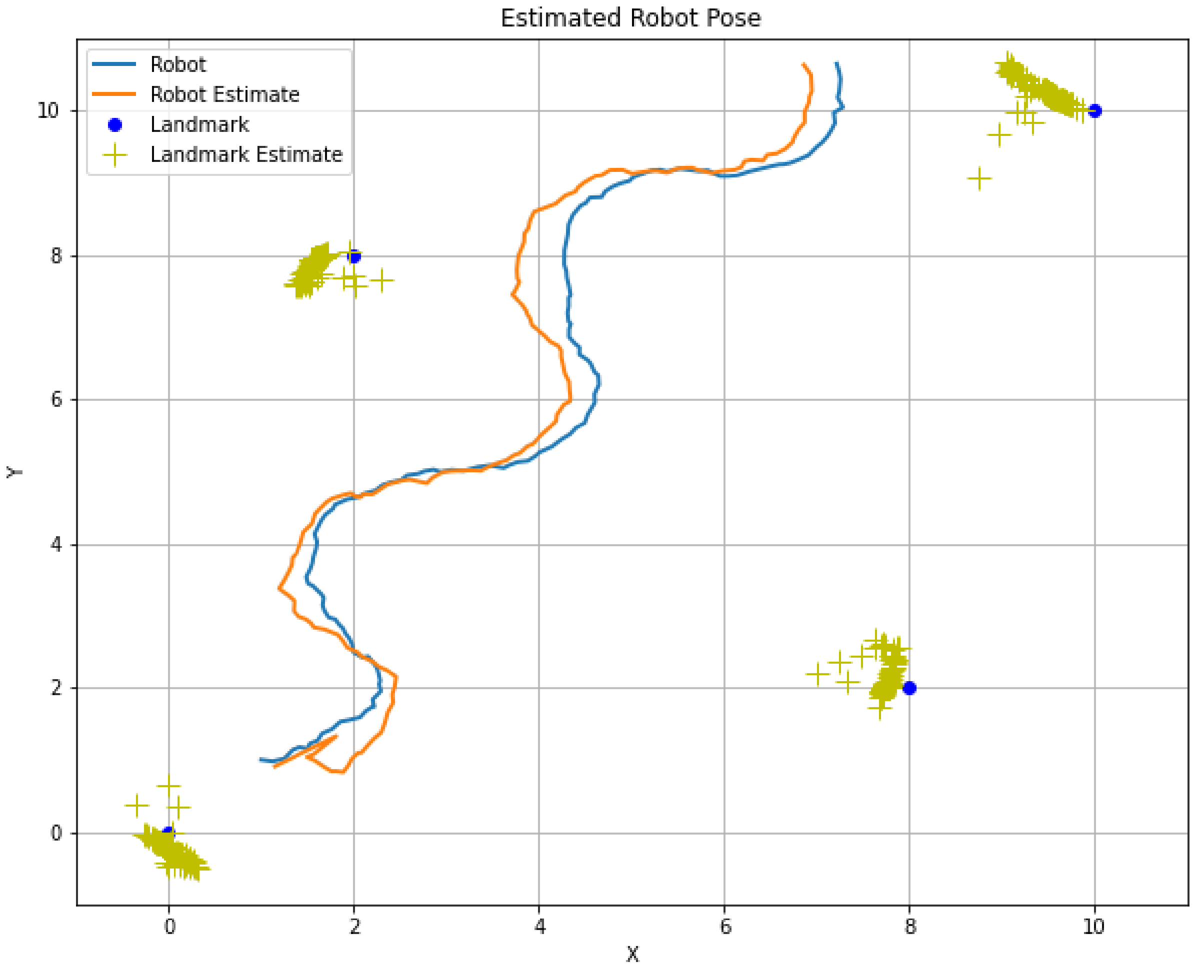


Figure 11: Caption

Part (iv)

The uncertainty here is clearly higher because, we are estimating both the robot pose and the landmark location simultaneously which are unknown. In problem 1, the objective was to find only the landmark location and the noise was also zero. In problem 2, the landmark location was already known.

Problem 1

```
####################### Code starts here #######################
# NOTE: What are the state transition and observation matrices?
A = np.eye(8)
C = np.eye(8)
####################### Code ends here #######################

for i in range(1, len(time)):
    ### Simulation.

    # True landmark dynamics
    x[:, i] = A @ x[:, i-1]

    # True received measurement
    v_noise = np.random.multivariate_normal(np.zeros((8,)), R)
    y = C @ x[:, i] + v_noise

    ### Estimation.

    ####################### Code starts here #######################
    # NOTE: Implement Kalman Filtering Predict and Update steps.
    # Write resulting means to mu_kf.
    # Write resulting covariances to cov_kf.

    # Prediction
    mu_bar_t = A@mu_kf[:,i-1:i] # Shape of 8,1
    sigma_bar_t = A@cov_kf[i-1]@A.T + Q

    # Correction Step
    Kt = sigma_bar_t@C.T@np.linalg.inv(C@sigma_bar_t@C.T + R)

    mu_t = mu_bar_t + Kt@(y.reshape(8,-1)-C@mu_bar_t)
    sigma_t = (np.eye(8) - Kt@C)@sigma_bar_t

    # Saving the Values
    mu_kf[:,i:i+1] = mu_t
    cov_kf[i] = sigma_t;

    ####################### Code ends here #######################
```

Problem 2

```
def getG(
    xr: np.ndarray,
    v: float,
    omega: float,
):
    """Computes the Jacobian of the dynamics model with respect
    to the robot state and input commands.

    Args:
        xr (np.ndarray): Robot state.
        v (float): Linear velocity command.
        omega (float): Angular velocity command.
```

```python

    Returns:
        G (np.ndarray): Jacobian of the dynamics model.
    """
    ####################### Code starts here #######################
    theta = xr[2]
    G = np.eye(3)
    G[0,2] = -v*np.sin(theta)*dt
    G[1,2] = v*np.cos(theta)*dt
    ####################### Code ends here #######################
    return G


def getH(
    xr: np.ndarray,
    xm: np.ndarray,
) -> np.ndarray:
    """Computes the Jacobian of the measurement model with respect
    to the robot state and the landmark positions.

    Args:
        xr (np.ndarray): Robot state.
        xm (np.ndarray): Landmark positions.

    Returns:
        H (np.ndarray): Jacobian of the measurement model.
    """
    ####################### Code starts here #######################
    H = np.zeros((8,3))

    def get_2rows(xmi,xr):
        """
        xmi: (2,1) array for (xmi,ymi)
        xr: x,y,theta
        """
        x,y,theta = xr.flatten()
        xmi,ymi = xmi.flatten() # Flattening to make it 2, even if it is (2,1)

        rows = np.array([[-np.cos(theta),-np.sin(theta), -np.sin(theta)*(xmi-x)+np.cos(theta)*(ymi-y)],
                         [np.sin(theta), -np.cos(theta), -np.cos(theta)*(xmi-x)-np.sin(theta)*(ymi-y)]])

        return rows


    for i in range(4):
        xmi = xm[2*i:2*i+2]
        H[2*i:2*i+2] = get_2rows(xmi,xr)

    ####################### Code ends here #######################
    return H
```

```python
for i in range(1, len(time)):
    ### Simulation.

    # True robot commands
```

```python
    v = 1
    omega = np.sin(time[i])

    # True robot dynamics
    w_noise = np.random.multivariate_normal(np.zeros((3,)), Q)
    x[:, i] = dynamics_model(x[:, i-1], v, omega) + w_noise

    # True received measurement
    v_noise = np.random.multivariate_normal(np.zeros((8,)), R)
    y = measurement_model(x[:, i], xm) + v_noise

    ### Estimation.

    ###################### Code starts here ######################
    # NOTE: Implement Extended Kalman Filter Predict and Update steps.
    # Write resulting means to mu_ekf.
    # Write resulting covariances to cov_ekf.

    # EKF Prediction
    # Hint: Find current G (Jacobian of the state dynamics model)

    # EKF Update
    # Hint: Find current H (Jacobian of the measurement model)



    # Prediction
    mu_prev = mu_ekf[:,i-1]
    G = getG(mu_prev,v,omega)

    mu_bar_t = dynamics_model(mu_prev,v,omega) # Shape of 8,1
    assert mu_bar_t.shape == (3,1) or mu_bar_t.shape == (3,)
    sigma_bar_t = G@cov_ekf[i-1]@G.T + Q
    assert sigma_bar_t.shape == (3,3)

    # Correction Step
    H = getH(mu_bar_t,xm)
    assert H.shape == (8,3)


    Kt = sigma_bar_t@H.T@np.linalg.inv(H@sigma_bar_t@H.T + R)
    assert Kt.shape == (3,8)

    z_measurement_model = measurement_model(mu_bar_t,xm)
    # print(z_measurement_model.shape)

    mu_t = mu_bar_t + Kt@(y-z_measurement_model)
    # print(mu_t.shape)
    sigma_t = (np.eye(3) - Kt@H)@sigma_bar_t

    # # Saving the Values
    mu_ekf[:,i] = mu_t
    cov_ekf[i] = sigma_t;


    ###################### Code ends here ######################
```

Problem 3

```python
# unpack dimensions
T = len(time)
n = mu0.shape[0]

# containers for belief
mu_pf = np.zeros((n, T))
cov_pf = np.zeros((n, n, T))

# containers for particles
particles = np.zeros((n, num_particles))
updated_particles = np.zeros((n, num_particles))

# sample particles
particles = (mu0.reshape(n, 1) + scipy.linalg.sqrtm(sigma0)
                @ np.random.normal(size=(n, num_particles)))

# allocate weight vectors
weights = np.ones(num_particles) / num_particles
updated_weights = np.ones(num_particles)

# precompute meas. noise covariance inverse
R_inv = np.linalg.inv(R)

for i in range(0, len(time) - 1):
    ### Simulation.
    # True robot commands
    v = 1
    omega = np.sin(time[i])

    # True robot dynamics
    w_noise = np.random.multivariate_normal(np.zeros((3,)), Q)
    x[:, i+1] = dynamics_model(x[:, i], v, omega) + w_noise

    # True received measurement
    v_noise = np.random.multivariate_normal(np.zeros((8,)), R)
    y = measurement_model(x[:, i+1], xm) + v_noise

    ### Estimation.
    # sample particle noises
    W_particles = (np.linalg.cholesky(Q)
                    @ np.random.normal(size=(n, num_particles)))
    # print(W_particles.shape)

    ####################### Code starts here #######################
    # TODO: Implement Particle Filter's Predict and Update steps.
    # 1) Store the current belief's mean and covariance to 'mu_pf' and 'cov_pf'. Hint: Use functions 'np.sum()' and 'np.cov()'.
    # 2) Update each particle with its weight. Hint: Use functions 'dynamics_model()', 'measurement_model()' and 'gaussian_pdf()'.
    # Do not forget to add the process noise to each particle, i.e., 'W_particles'.
    # 3) Update and normalize the weights.
    # 4) Resample particles according to the updated particle weights. Hint: Use function 'np.random.choice()'.
    # 5) Reset weights to uniform.

```

```
    # Prediction Step
    mu_pf_i = np.mean(particles,axis=1) # (3,)
    cov_pf_i = np.cov(particles) # (3,3)

    mu_pf[:,i] = mu_pf_i
    cov_pf[:,:,i] = cov_pf_i


    weights_i = np.zeros(num_particles)
    X_bar_i = np.zeros((n, num_particles))
    for m in range(num_particles):
        xtm = dynamics_model(particles[:,m],v,omega) + W_particles[:,m]

        X_bar_i[:,m] = xtm
        wtm = gaussian_pdf(y,measurement_model(xtm,xm),R_inv) # p(zt|xt)
        weights_i[m] = wtm

    weights_i = weights_i/weights_i.sum()

    selected_indices = np.random.choice(X_bar_i.shape[1], size=num_particles, p=
    weights_i)
    particles = X_bar_i[:,selected_indices]
    ####################### Code ends here #######################

# store final belief
mu_pf[:, -1] = np.mean(particles, axis=1)
cov_pf[:, :, -1] = np.cov(particles)
```

Problem 4

```
def getG(
    x: np.ndarray,
    v: float,
    omega: float,
):
    """Computes the Jacobian of the dynamics model with respect
    to the robot state and input commands.

    Args:
        x (np.ndarray): Robot + Landmark concatenated state.
        v (float): Linear velocity command.
        omega (float): Angular velocity command.

    Returns:
        G (np.ndarray): Jacobian of the dynamics model.
    """
    ####################### Code starts here #######################
    xr = x[:3]
    xm = x[3:]

    theta = xr[2]
    Gr = np.eye(3)
    Gr[0,2] = -v*np.sin(theta)*dt
    Gr[1,2] = v*np.cos(theta)*dt

    G = np.eye(x.shape[0])
```

```python
    G[:3,:3] = Gr


    ###################### Code ends here ######################
    return G


def getH(
    x: np.ndarray,
) -> np.ndarray:
    """Computes the Jacobian of the measurement model with respect
    to the robot state and the landmark positions.

    Args:
        x (np.ndarray): Robot + Landmark concatenated state.
    Returns:
        H (np.ndarray): Jacobian of the measurement model.
    """
    ###################### Code starts here ######################
    num_landmarks = 4
    num_states = 3

    xr = x[:num_states]
    xm = x[num_states:]

    H = np.zeros((8,11))

    def get_2rows(xmi,xr):
        """
        xmi: (2,1) array for (xmi,ymi)
        xr: x,y,theta
        """
        x,y,theta = xr.flatten()
        xmi,ymi = xmi.flatten() # Flattening to make it 2, even if it is (2,1)

        rows = np.array([[-np.cos(theta),-np.sin(theta), -np.sin(theta)*(xmi-x)+np.cos(
    theta)*(ymi-y)],
                         [np.sin(theta), -np.cos(theta), -np.cos(theta)*(xmi-x)-np.sin(
    theta)*(ymi-y)]])

        return rows

    theta = xr[2]
    H_theta = np.array([[np.cos(theta),np.sin(theta)],
                        [-np.sin(theta),np.cos(theta)]])

    for i in range(num_landmarks):
        xmi = xm[2*i:2*i+2]
        H[2*i:2*i+2,:num_states] = get_2rows(xmi,xr)

        H[2*i:2*i+2,2*i+num_states:2*i+num_states+2] = H_theta

    ###################### Code ends here ######################
    return H
```

```python
for i in range(1, len(time)):
    ### Simulation.
```

```python

    # True robot commands
    v = 1
    omega = np.sin(time[i])

    # True robot dynamics
    w_noise = np.random.multivariate_normal(np.zeros((11,)), Q)
    x[:, i] = dynamics_model(x[:, i-1], v, omega) + w_noise

    # True received measurement
    v_noise = np.random.multivariate_normal(np.zeros((8,)), R)
    y = measurement_model(x[:, i]) + v_noise

    ### Estimation.

    ###################### Code starts here ######################
    # NOTE: Implement Extended Kalman Filter Predict and Update steps.
    # Write resulting means to mu_ekf.
    # Write resulting covariances to cov_ekf.

    # EKF Prediction
    # Hint: Find current G (Jacobian of the state dynamics model)

    # EKF Update
    # Hint: Find current H (Jacobian of the measurement model)
    # Prediction
    mu_prev = mu_ekf[:,i-1]
    G = getG(mu_prev,v,omega)

    mu_bar_t = dynamics_model(mu_prev,v,omega) # Shape of 8,1
    assert mu_bar_t.shape == (11,1) or mu_bar_t.shape == (11,)
    sigma_bar_t = G@cov_ekf[i-1]@G.T + Q
    assert sigma_bar_t.shape == (11,11)

    # Correction Step
    H = getH(mu_bar_t)
    assert H.shape == (8,11)


    Kt = sigma_bar_t@H.T@np.linalg.inv(H@sigma_bar_t@H.T + R)
    assert Kt.shape == (11,8)

    z_measurement_model = measurement_model(mu_bar_t)
    # print(z_measurement_model.shape)

    mu_t = mu_bar_t + Kt@(y-z_measurement_model)
    # print(mu_t.shape)
    sigma_t = (np.eye(11) - Kt@H)@sigma_bar_t

    # # Saving the Values
    mu_ekf[:,i] = mu_t
    cov_ekf[i] = sigma_t;

    ###################### Code ends here ######################
```