Principles of Robot Autonomy: Homework 5

Wei-Lin Pai
11/30/24

Other students worked with:
Time spent on homework: 6 hrs

Problem 1: Kalman Filter for 2D Lanfmark Localization

i

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

ii

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

iii

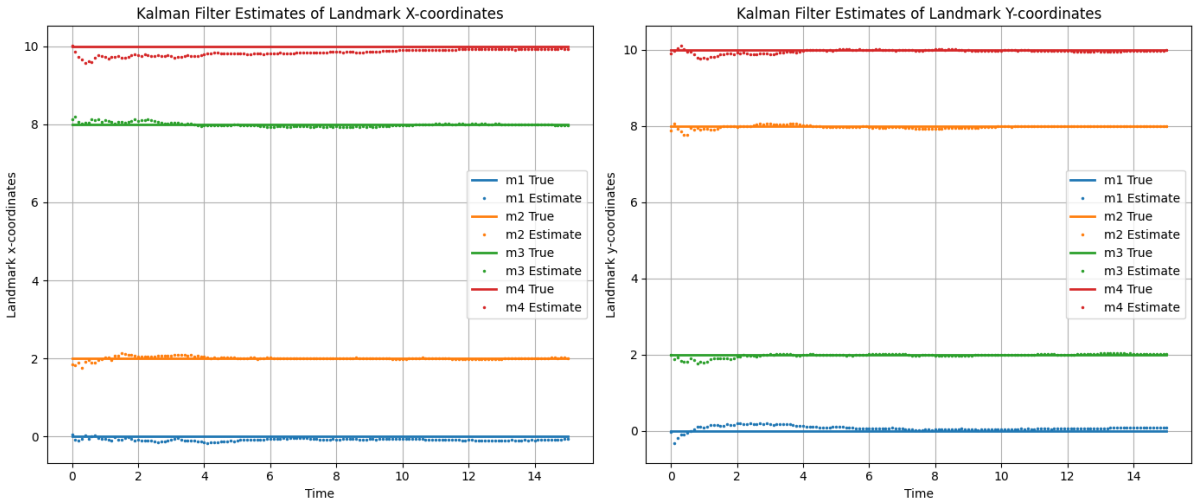

Figure 1: Estimated x y

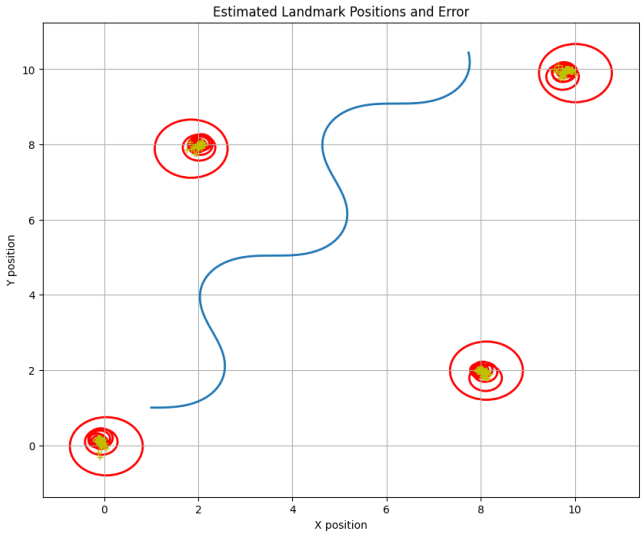

Figure 2: Estimated Landmark Position and Error

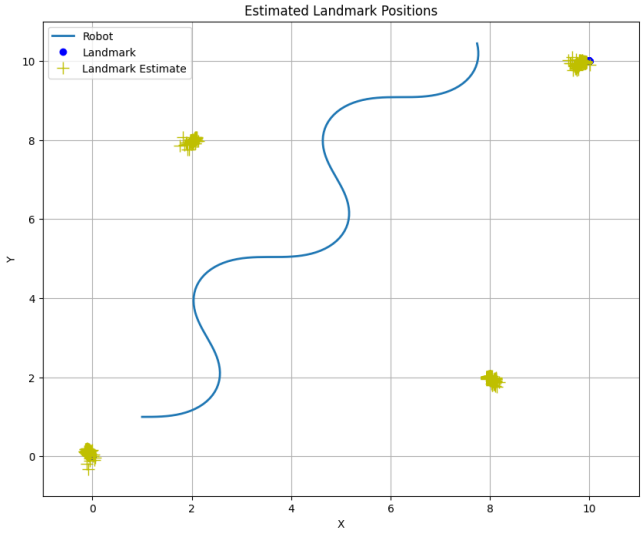

Figure 3: Estimated Landmark Position

iv

(a) The value of process noise is 0. Assume that the landmarks are stationary.
(b) In general, the KF will assume that the object is moving or the robot pose is not fully known.

Problem 2: Extended Kalman Filter for Robot Localization

i

$$G = \begin{bmatrix} \frac{\partial x_{r,t+1}}{\partial x_{r,t}} & \frac{\partial x_{r,t+1}}{\partial y_{r,t}} & \frac{\partial x_{r,t+1}}{\partial \theta_{r,t}} \\ \frac{\partial y_{r,t+1}}{\partial x_{r,t}} & \frac{\partial y_{r,t+1}}{\partial y_{r,t}} & \frac{\partial y_{r,t+1}}{\partial \theta_{r,t}} \\ \frac{\partial \theta_{r,t+1}}{\partial x_{r,t}} & \frac{\partial \theta_{r,t+1}}{\partial y_{r,t}} & \frac{\partial \theta_{r,t+1}}{\partial \theta_{r,t}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -V_t \sin(\theta_{r,t})t \\ 0 & 1 & V_t \cos(\theta_{r,t})t \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

ii

$$H_{m1} = \begin{bmatrix} \frac{\partial z_{m,t}}{\partial x_{r,t}} & \frac{\partial z_{m,t}}{\partial y_{r,t}} & \frac{\partial z_{m,t}}{\partial \theta_{r,t}} \end{bmatrix} = \begin{bmatrix} -\cos(\theta_{r,t}) & -\sin(\theta_{r,t}) & -\sin(\theta_{r,t})(x_{m1,t} - x_{r,t}) + \cos(\theta_{r,t})(y_{m1,t} - y_{r,t}) \\ \sin(\theta_{r,t}) & -\cos(\theta_{r,t}) & -\cos(\theta_{r,t})(x_{m1,t} - x_{r,t}) - \sin(\theta_{r,t})(y_{m1,t} - y_{r,t}) \end{bmatrix} \tag{4}$$

for each landmark.
The resulting Jacobian will be a 8 X 3 matrix.

iii

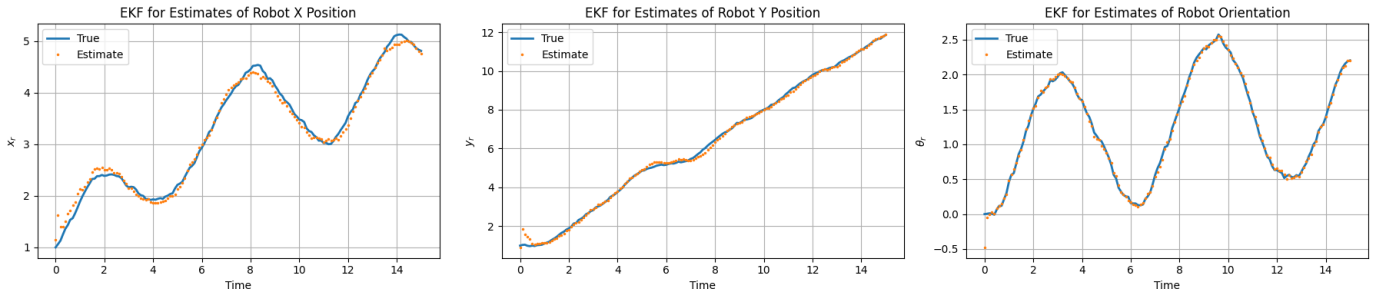

Figure 4: Estimated x y

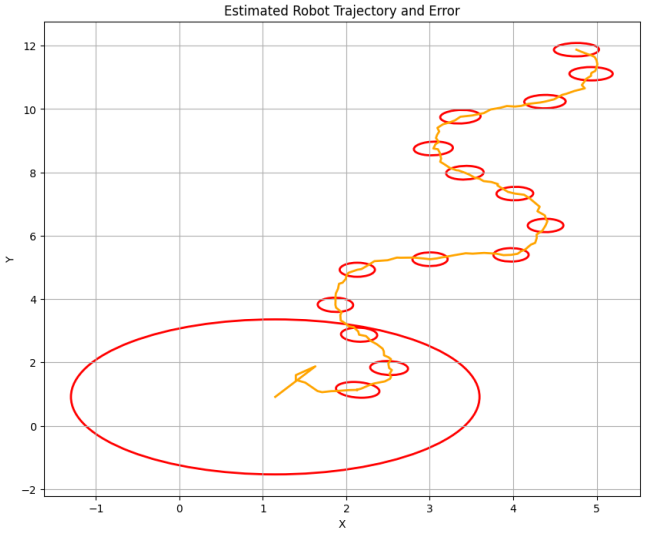

Figure 5: Estimated Robot Trajectory and Error

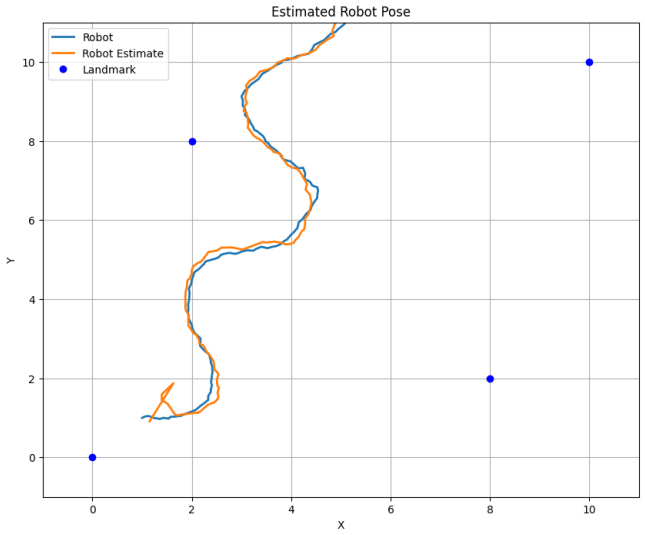

Figure 6: Estimated Robot Pose

iv

The first order-linear approxmiation of EKF may fail if the robot has rapid changes in direction or velocity, where linearization become inaccurate. For example, sharp turns or sudden accelerations.
For more generalized filter, we can choose particle filter which represent beliefs as particles, thus can handle highly nonlinear or multi-modal scenario.

Problem 3: Particle Filter

i

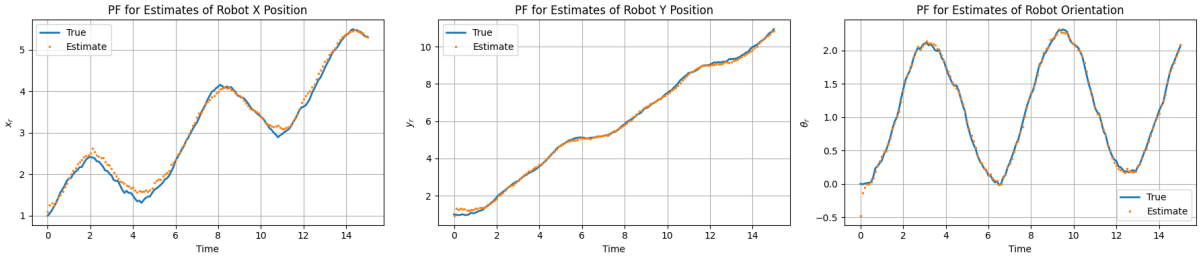

Figure 7: Estimated x y

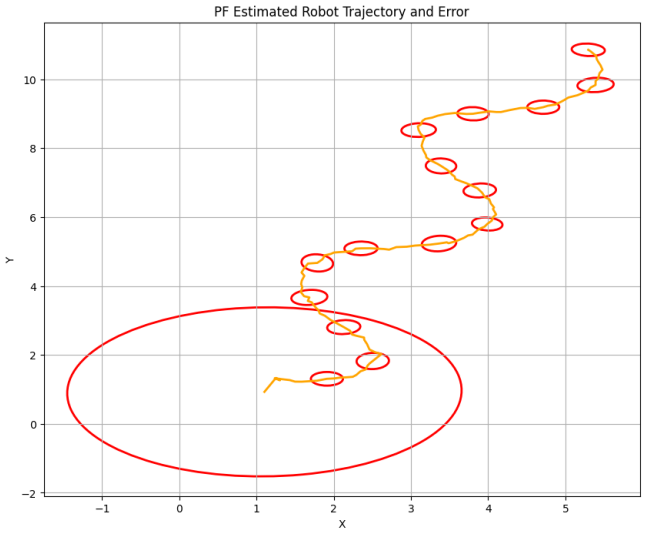

Figure 8: Estimated Robot Trajectory and Error

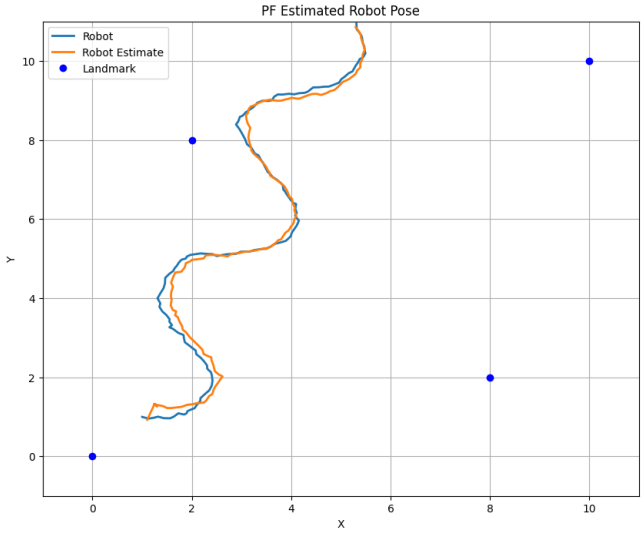

Figure 9: Estimated Robot Pose

ii

According to picture 6 and 9, PF estimates appear smoother and more consistent compared to the EKF result. The differences in the plots are that the PF estimates are likely to be smoother, more consistent, and closer to the true robot pose compared to the EKF estimates when it exhibit more sudden or nonlinear behavior.

iii

(a) True. Particle filter are non-parametric methods.
(b) False. Particle filter are inherently stochastic, it use random sampling.
(c) True. For unimodal problem, Kalman filter are more computationally efficient.

Problem 4: EKF Slam

i

$$G(x_t) = \begin{bmatrix} G_{robot} & 0_{3x8} \\ 0_{8x3} & I_{8x8} \end{bmatrix}, where\ G_{robot} = \begin{bmatrix} 1 & 0 & -V_t \sin(\theta_{r,t})t \\ 0 & 1 & V_t \cos(\theta_{r,t})t \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

ii

$$H_{m1} = \begin{bmatrix} \frac{\partial h_1^x}{\partial x_r} & \frac{\partial h_1^x}{\partial m_1} & \frac{\partial h_1^x}{\partial m_2} & \frac{\partial h_1^x}{\partial m_3} & \frac{\partial h_1^x}{\partial m_4} \\ \frac{\partial h_1^y}{\partial x_r} & \frac{\partial h_1^y}{\partial m_1} & \frac{\partial h_1^y}{\partial m_2} & \frac{\partial h_1^y}{\partial m_3} & \frac{\partial h_1^y}{\partial m_4} \end{bmatrix} \tag{6}$$

Where:

$$\frac{\partial h_1^x}{\partial x_r} = \begin{bmatrix} -\cos(\theta_r) & -\sin(\theta_r) & -\sin(\theta_r)(x_m - x_r) + \cos(\theta_r)(y_m - y_r) \end{bmatrix} \tag{7}$$

$$\frac{\partial h_1^y}{\partial x_r} = \begin{bmatrix} \sin(\theta_r) & -\cos(\theta_r) & -\cos(\theta_r)(x_m - x_r) - \sin(\theta_r)(y_m - y_r) \end{bmatrix} \tag{8}$$

$$\frac{\partial h_1^x}{\partial m_1} = \begin{bmatrix} \cos(\theta_r) & \sin(\theta_r) \end{bmatrix} \tag{9}$$

$$\frac{\partial h_1^y}{\partial m_1} = \begin{bmatrix} -\sin(\theta_r) & \cos(\theta_r) \end{bmatrix} \tag{10}$$

$$\frac{\partial h_1^{x,y}}{\partial m_{2,3,4}} = \begin{bmatrix} 0 & 0 \end{bmatrix} \tag{11}$$

The resulting H will be a 8 X 11 matrix, where each landmarks H_{mi} is the 2 X 11 Jacobian for the i-th landmark.

iii

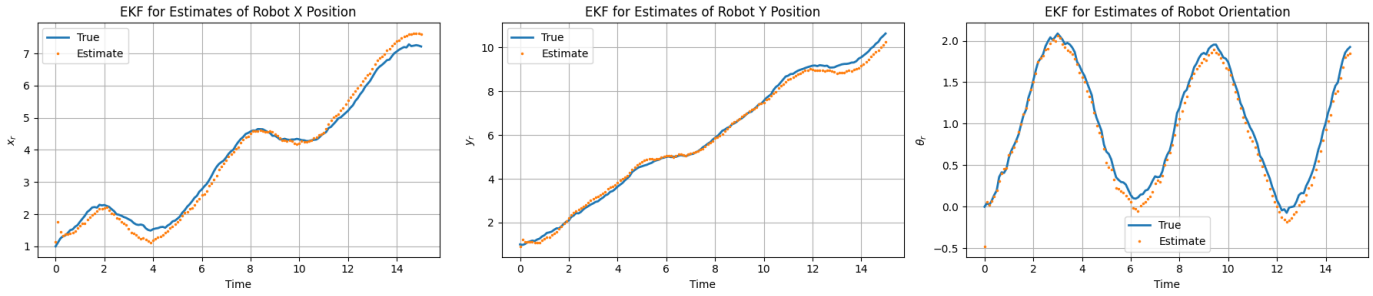

Figure 10: EKF Estimate for Robot Position and Orientation

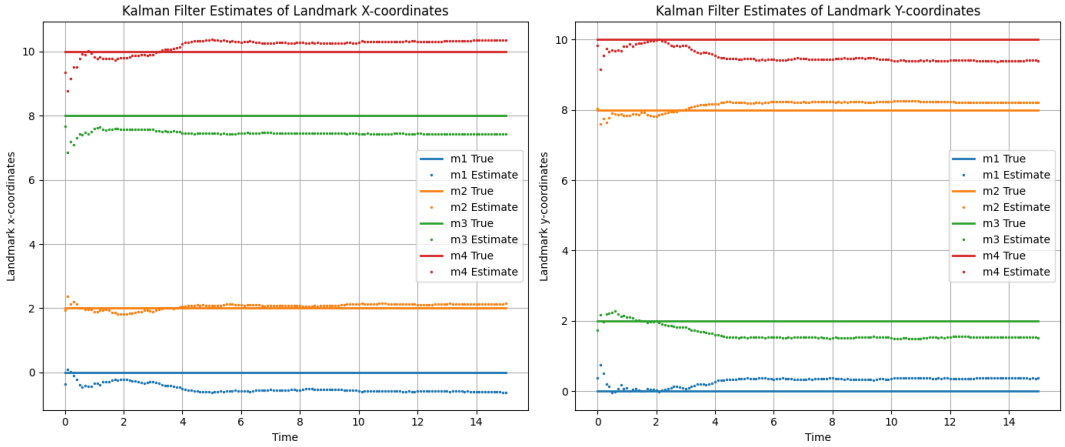

Figure 11: Kalman Filter Estimates of Landmark X,Y

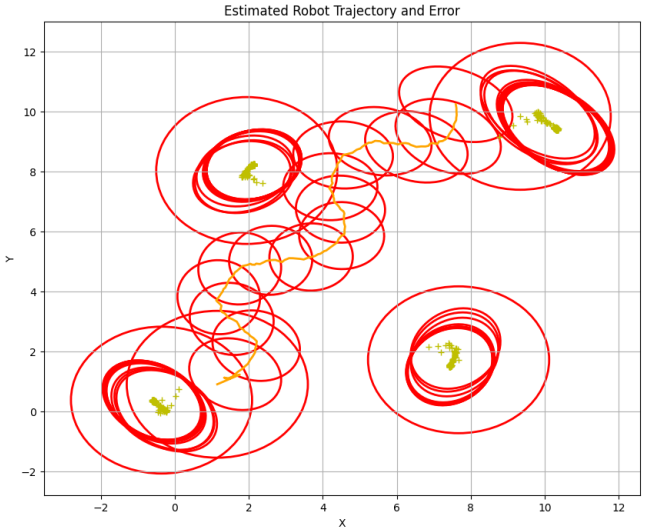

Figure 12: Estimated Robot Trajectory and Error

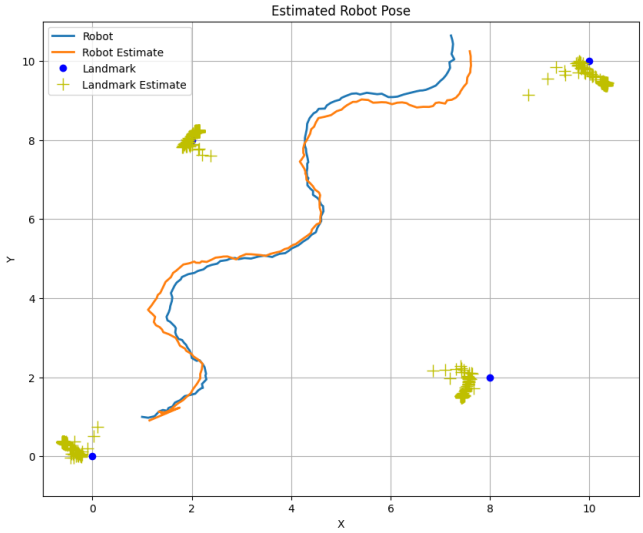

Figure 13: Estimated Robot Pose

iv

In problem 1, the robot pose was known. In problem 2 landmark position were known. However, in this problem we are solving SLAM which simultaneously estimate both robot pose and landmark positions. As we know, robot pose and landmark position are now interdependent, so error in estimating the robot pose will affects landmarks position estimate. Conversly, the uncertainty in landmark positions impacts robot pose estimation. Thus, the estimation error might accumulate and cause higher uncertainty.

Appendix A: Code Submission

p1 kalman filtering.ipynb

```
######################## Code starts here ########################
# NOTE: What are the state transition and observation matrices?
A = np.identity(8, dtype = float)
C = np.identity(8, dtype = float)
######################## 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.
    mu_kf_pred = A @ mu_kf[:, i-1]
    cov_kf_pred = A @ cov_kf[i-1] @ A.T + Q
    K_t = cov_kf_pred @ C.T @ np.linalg.inv(C @ cov_kf_pred @ C.T + R)
    mu_kf[:, i] = mu_kf_pred + K_t @ (y - C @ mu_kf_pred)
    cov_kf[i] = (np.identity(8) - K_t @ C) @ cov_kf_pred
    ######################## Code ends here ########################
```

p2 ekf localization.ipynb

```
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.

    Returns:
        G (np.ndarray): Jacobian of the dynamics model.
    """
    ######################## Code starts here ########################
```

```
    G = np.eye(3)
    G[0,2] = -dt*v*np.sin(xr[2])
    G[1,2] = dt*v*np.cos(xr[2])
    ######################## 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))
    for i in range(4):
      H[2*i,:] = np.array([-np.cos(xr[2]), -np.sin(xr[2]),
          -np.sin(xr[2])*(xm[2*i]-xr[0])+np.cos(xr[2])*(xm[2*i+1]-xr[1])])
      H[2*i+1,:] = np.array([np.sin(xr[2]), -np.cos(xr[2]),
          -np.cos(xr[2])*(xm[2*i]-xr[0])-np.sin(xr[2])*(xm[2*i+1]-xr[1])])
    ######################## Code ends here ########################

for i in range(1, len(time)):
    ### 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] = 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)
    mu_ekf_pred = dynamics_model(mu_ekf[:, i-1], v, omega)
    G = getG(mu_ekf_pred, v, omega)
    # EKF Update
    # Hint: Find current H (Jacobian of the measurement model)
    H = getH(mu_ekf_pred, xm)

    cov_ekf_pred = G @ cov_ekf[i-1] @ G.T + Q
    K_t = cov_ekf_pred @ H.T @ np.linalg.inv(H @ cov_ekf_pred @ H.T + R)
    mu_ekf[:, i] = mu_ekf_pred + K_t @ (y - measurement_model(mu_ekf_pred, xm))
    cov_ekf[i] = (np.eye(3) - K_t @ H) @ cov_ekf_pred
    ######################## Code ends here ########################
```

p3 pf localization.ipynb

```
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)))

     ######################## 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.
    mu_pf[:,i] = np.mean(particles, axis=1)
    cov_pf[:,:,i] = np.cov(particles,aweights=weights)

    for j in range(num_particles):
        particles[:,j] = dynamics_model(particles[:,j], v, omega) + W_particles[:,j]
        measurement_model_particles = measurement_model(particles[:,j], xm)
        weights[j] = gaussian_pdf(y, measurement_model_particles, R_inv)
```

```
    updated_particles = np.random.choice(num_particles, num_particles,
        p=weights/np.sum(weights))
    particles = particles[:,updated_particles]
    weights = np.ones(num_particles) / num_particles
    ######################## Code ends here ########################

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

p4 ekf slam.ipynb

```
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 ########################
    G = np.eye(11)
    G[0,2] = -dt*v*np.sin(x[2])
    G[1,2] = dt*v*np.cos(x[2])
    ######################## 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 ########################
    H = np.zeros((8,11))
    for i in range(4):
      H[2*i,:3] = np.array([-np.cos(x[2]), -np.sin(x[2]),
          -np.sin(x[2])*(xm[2*i]-x[0])+np.cos(x[2])*(xm[2*i+1]-x[1])])
```

```
      H[2*i+1,:3] = np.array([np.sin(x[2]), -np.cos(x[2]),
          -np.cos(x[2])*(xm[2*i]-x[0])-np.sin(x[2])*(xm[2*i+1]-x[1])])
      H[2*i,2*i+3:2*i+5] = np.array([np.cos(x[2]), np.sin(x[2])])
      H[2*i+1,2*i+3:2*i+5] = np.array([-np.sin(x[2]), np.cos(x[2])])
    ######################## Code ends here ########################
    return H
for i in range(1, len(time)):
    ### Simulation.

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

    cov_ekf_pred = G @ cov_ekf[i-1] @ G.T + Q
    K_t = cov_ekf_pred @ H.T @ np.linalg.inv(H @ cov_ekf_pred @ H.T + R)
    mu_ekf[:, i] = mu_ekf_pred + K_t @ (y - measurement_model(mu_ekf_pred))
    cov_ekf[i] = (np.eye(11) - K_t @ H) @ cov_ekf_pred
    ######################## Code ends here ########################
```