Q3. Nonlinear Filters

In [ ]:
```
import q3_simulator_class as sim
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import normalize
import time
```

(a) EKF

The EKF uses the Jacobian matrices of the state transition and measurement models to linearize around the current estimate.

Extended Kalman Filter (EKF):

- Nonlinear Update: Uses a nonlinear measurement model
- Linearization of Measurement Model: The EKF linearizes the measurement model using the Jacobian matrix evaluated at the predicted state.
- Kalman Gain and State Update: Similar to KF, but using the Jacobian for updates.
- Covariance Update: Updated similarly to KF.

Some parameters

In [ ]:
```
# Define update parameters
n_steps = 100
dt = robot.dt # [s]

# Define base station localtion, pj
p_1 = robot.base_station_locs[0] # [m]
p_2 = robot.base_station_locs[1] # [m]

# Define noise matrix
Q = robot.q_mat # process noise
R = robot.r_mat # measuremnt noise

# Define control input
s_t = 1 # speed input [m/s]
phi_t = lambda t: np.sin(t) # rotation rate input [rad/s] (as a function of t)

# Define initial state estimation
mu_0 = np.array([0, 0, 0]) # mean, [m,m,rad]
sigma_0 = 0.1 * np.eye(3) # cov

# A (function), Jacobian of state transition matrix (f(xt,ut))
A_func = lambda mu, u: np.array([[1, 0, -dt * u[0] * np.sin(mu[2])],
                                 [0, 1,  dt * u[0] * np.cos(mu[2])],
                                 [0, 0, 1]])
# C_func = lambda mu: np.block([[(mu[:2] - p_1) / np.sqrt(np.sum((mu[:2] - p_1) **
#                             [(mu[:2] - p_2) / np.sqrt(np.sum((mu[:2] - p_2) ** 2)),
# C (function), Jacobian of the measurement model (Yt)
C_func = lambda mu: np.block([[(mu[:2] - p_1) / np.linalg.norm(mu[:2] - p_1), 0],
                             [(mu[:2] - p_2) / np.linalg.norm(mu[:2] - p_2), 0]])
```

Initialize State

In [ ]:
```
state = mu_0 # assume initial state = initial state estimation
robot = sim.MobileRobotSimulator()
pos_hist, meas_hist, u_hist = robot.simulate(state, n_steps)
```

EKF function + Updated State

In [ ]:
```
def EKF(mu_0, sigma_0, u_arr, y_arr, A, C, Q, R):
    mu_update = [mu_0]  # Updated state mean
    cov_update = [sigma_0]  # Updated state covariance
    mu_pred = []  # Predicted state mean
    cov_pred = []  # Predicted state covariance
    A_list = []
    C_list = []

    for t, (u, y) in enumerate(zip(u_arr, y_arr)):
        # Current state estimate for Jacobian calculation
        mu_cur = mu_update[-1]

        # Compute Jacobians for the current estimate
        A_t = A(mu_cur, u)
        C_t = C(mu_cur)

        # PREDICT
        mu_bar_next, _ = robot.noiseless_dynamics_step(mu_cur, t = t)  # nonlinear
        sigma_bar_next = A_t @ cov_update[-1] @ A_t.T + Q  # Predict the next covar

        # UPDATE
        K_t_numerator = sigma_bar_next @ C_t.T
        K_t_denominator = C_t @ sigma_bar_next @ C_t.T + R
        K_t = K_t_numerator @ np.linalg.inv(K_t_denominator)  # Kalman gain

        # Observation prediction
        expected_y = robot.noiseless_measurement_step(mu_cur)  # non-linear measure
        # print(mu_bar_next.shape, K_t.shape, y.shape, expected_y.shape)
        mu_next = mu_bar_next + K_t @ (y - expected_y)  # Updated state mean

        sigma_next = (np.eye(len(mu_0)) - K_t @ C_t) @ sigma_bar_next  # Updated st

        # Collect results for output
        mu_update.append(mu_next) # store the updated
        cov_update.append(sigma_next)
        mu_pred.append(mu_bar_next) # Store the predicted
        cov_pred.append(sigma_bar_next)
        A_list.append(A_t) # store the Jacobians
        C_list.append(C_t)

    return np.array(mu_update[1:]), np.array(cov_update[1:]), np.array(mu_pred), np

mu_update, cov_update, mu_pred, cov_pred, A_arr, C_arr = EKF(mu_0, sigma_0, u_hist,
```

Check observable

In [ ]:
```
def check_observability_from_step(A_arr, C_arr):
    # Initialize parameters
    is_observable_from = None
    last_known_observable = False

    # Number of steps to check
    total_steps = len(A_arr)
```

```
    for k in range(1, total_steps + 1):  # Start from 1 to avoid zero-step scenario
        rank_O_k, state_dimension, is_observable = calculate_observability(A_arr, C

        # Output results only if state changes
        if is_observable and not last_known_observable:
            print(f"System becomes observable from step {k} onwards (0-index)")
            is_observable_from = k
            last_known_observable = True
        elif not is_observable and last_known_observable:
            print(f"System loses observability at step {k} (0-index)")
            last_known_observable = False

        # print every step's detail (comment out)
        # print(f"Step {k}: Rank = {rank_O_k}, Dimension = {state_dimension}, Obser

    return is_observable_from

# Call the function
observable_from_step = check_observability_from_step(A_arr, C_arr)
if observable_from_step is not None:
    print(f"The system is consistently observable from step {observable_from_step}
else:
    print("The system is never fully observable within the given steps.")
```

```
System becomes observable from step 2 onwards (0-index)
The system is consistently observable from step 2 onwards. (0-index)
```

Plot EKF

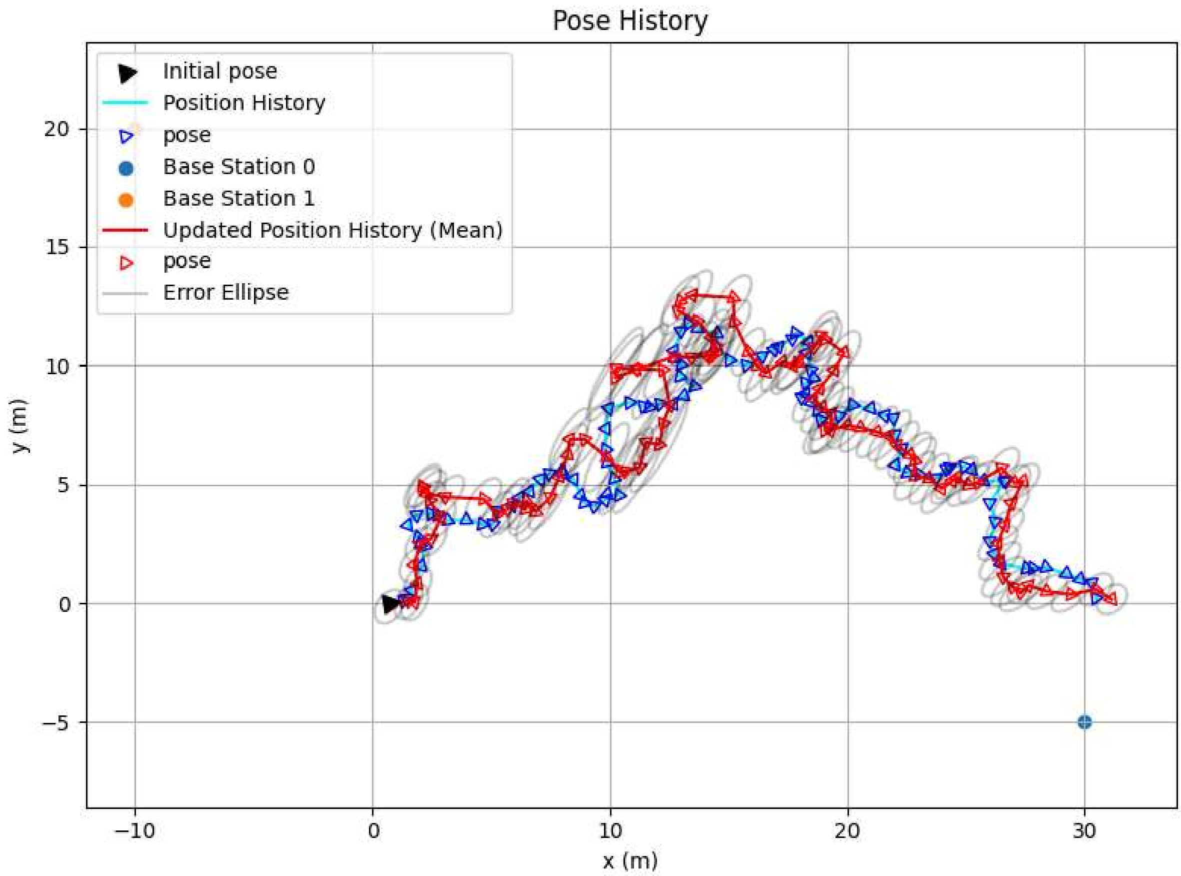
In [ ]:

```
# plot error ellipse from Pset2
import scipy
from scipy.stats import chi2

def error_ellipse(ax, mu, sig, p=0.95):
    # Ellipse
    t = np.linspace(0, 2*np.pi, 100)
    circ = np.array([np.cos(t), np.sin(t)]).T
    # calculate stretch
    r = chi2.ppf(p, df=2)
    eta = scipy.linalg.sqrtm(sig) * np.sqrt(r)
    ellipse = circ @ eta + mu

    # Draw
    ax.plot(ellipse[:,0], ellipse[:,1], color='k', alpha=0.2, label="Error Ellipse'
```

In [ ]:

```
# PLOT result (traj)

# Plot true pose
fig_ekf = robot.plot_pose_history(pos_hist, show_plot=False)

# Plot updated traj
ax = fig_ekf.axes[0]
ax.plot(mu_update[:,0], mu_update[:,1], color='r', label="Updated Position History

# Plot the updated pose
labeled = False
for px, py, theta in mu_update:
    plt.scatter(px, py,
                edgecolors='red', facecolors='none',
                s=50, zorder=2,
                marker=(3, 0, -90 + np.rad2deg(theta)),
                label="pose" if not labeled else None)
```

```
    labeled = True

# Plot Error Ellipse
for t in range(mu_update.shape[0]):
    mean = mu_update[t,:2] # mean for pos
    cov = cov_update[t,:2, :2] # cov for pos
    error_ellipse(ax, mean, cov)


handles, legends = ax.get_legend_handles_labels()
plt.legend(handles[:8], legends[:8])
plt.show()
```



(b) UKF

```
In [ ]: def UKF(robot=robot, mu0=mu_0, sig0=sigma_0, Q=Q, R=R, yrun_hist=meas_hist, n_steps
    # Initialize arrays to store predicted and updated means and covariances
    mu_pred_UKF = np.zeros((n_steps + 1, mu0.shape[0]))  # (101, 3)
    sig_pred_UKF = np.zeros((n_steps + 1, sig0.shape[0], sig0.shape[0]))  # (303, 3
    mu_update_UKF = np.zeros_like(mu_pred_UKF)
    sig_update_UKF = np.zeros_like(sig_pred_UKF)

    # Set initial predicted and updated states
    mu_pred_UKF[0, :] = mu0
    sig_pred_UKF[0, ...] = sig0
    mu_update_UKF[0, :] = mu0
    sig_update_UKF[0, ...] = sig0

    # Unscented Transform function
    def unscented_transform(mean, cov, lam=2):
        n = cov.shape[0]
        sqrt_cov = np.sqrt(lam + n) * sqrtm(cov)
        sigma_points = np.tile(mean, (2 * n + 1, 1))
        weights = lam / (lam + n) * np.ones((2 * n + 1, 1))
```

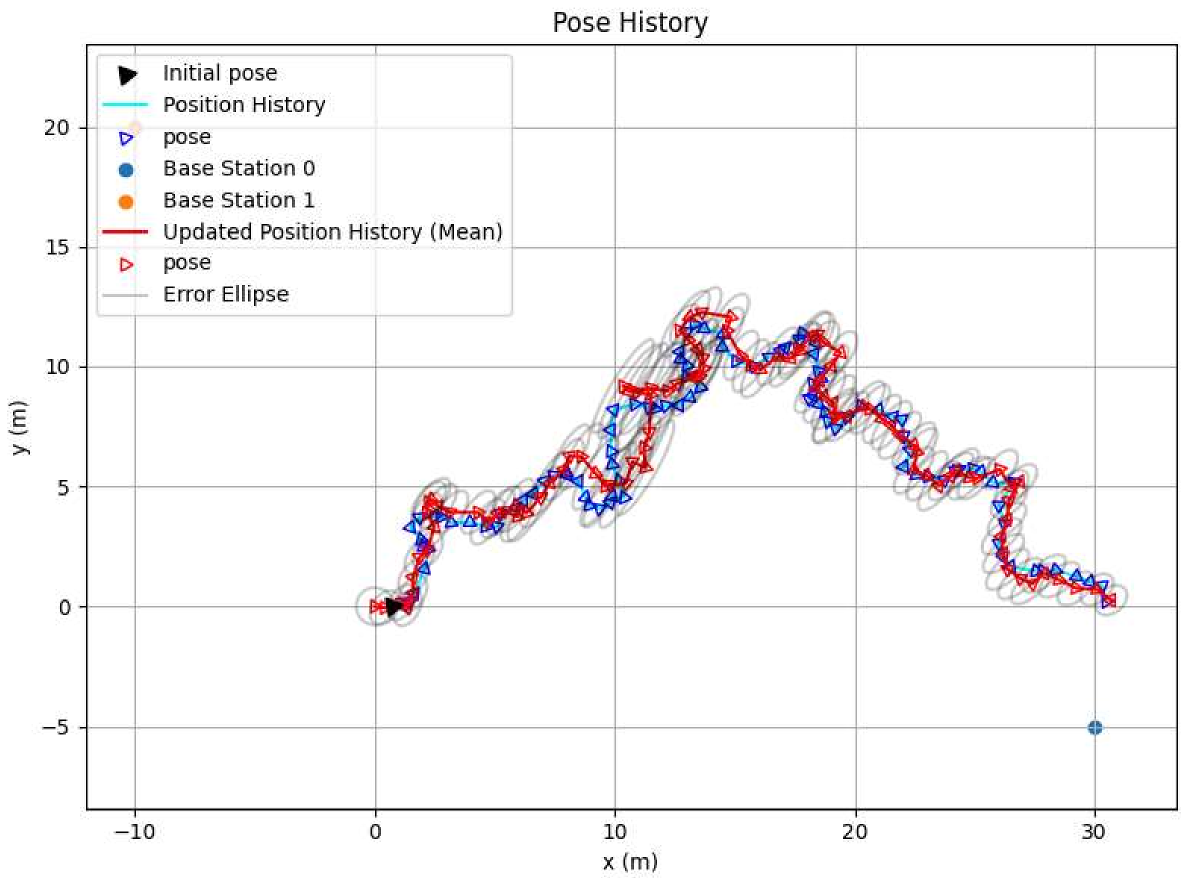
```
        for i in range(1, n + 1):
            sigma_points[i] += sqrt_cov[:, i - 1]
            weights[i] = 1 / (2 * (lam + n))
            sigma_points[i + n] -= sqrt_cov[:, i - 1]
            weights[i + n] = 1 / (2 * (lam + n))

        return sigma_points, weights

    # Function to invert the Unscented Transform
    def unscented_transform_inverse(sigma_points, weights):
        mean = np.sum(weights * sigma_points, axis=0)
        cov = (sigma_points - mean).T @ (weights * (sigma_points - mean))
        return mean, cov

    # Unscented Kalman Filter
    for t in range(1, n_steps + 1):
        # Prediction step
        sigma_points_pred, weights_pred = unscented_transform(mu_update_UKF[t - 1,
        sigma_points_pred_ = np.copy(sigma_points_pred)

        for i in range(sigma_points_pred_.shape[0]):
            sigma_points_pred_[i], _ = robot.noiseless_dynamics_step(sigma_points_p

        mu_pred, sig_pred = unscented_transform_inverse(sigma_points_pred_, weights
        sig_pred += Q  # Adding process noise

        # Update step
        sigma_points_update, weights_update = unscented_transform(mu_pred, sig_pred
        y_pred = np.zeros((sigma_points_update.shape[0], 2))

        for i in range(y_pred.shape[0]):
            y_pred[i] = robot.noiseless_measurement_step(sigma_points_update[i])

        y_exp, sig_y = unscented_transform_inverse(y_pred, weights_update)
        sig_y += R  # Adding measurement noise
        sig_xy = (sigma_points_update - mu_pred).T @ (weights_update * (y_pred - y_

        kalman_gain = np.transpose(np.linalg.solve(sig_y.T, sig_xy.T))
        mu_update = mu_pred + kalman_gain @ (yrun_hist[t - 1] - y_exp)
        sig_update = sig_pred - kalman_gain @ sig_xy.T

        # Record predictions and updates
        mu_pred_UKF[t, :] = mu_pred
        sig_pred_UKF[t, ...] = sig_pred
        mu_update_UKF[t, :] = mu_update
        sig_update_UKF[t, ...] = sig_update

    return mu_pred_UKF, sig_pred_UKF, mu_update_UKF, sig_update_UKF

mu_pred_UKF, sig_pred_UKF, mu_update_UKF, sig_update_UKF = UKF()
```

Plot UKF

In [ ]:

```
# PLOT result (traj)

# Plot true pose
fig_ukf = robot.plot_pose_history(pos_hist, show_plot=False)

# Plot updated traj
ax = fig_ukf.axes[0]
ax.plot(mu_update_UKF[:,0], mu_update_UKF[:,1], color='r', label="Updated Position
```

```
# Plot the updated pose
labeled = False
for px, py, theta in mu_update_UKF:
    plt.scatter(px, py,
                edgecolors='red', facecolors='none',
                s=50, zorder=2,
                marker=(3, 0, -90 + np.rad2deg(theta)),
                label="pose" if not labeled else None)
    labeled = True

# Plot Error Ellipse
for t in range(mu_update_UKF.shape[0]):
    mean = mu_update_UKF[t,:2] # mean for pos
    cov = sig_update_UKF[t,:2, :2] # cov for pos
    error_ellipse(ax, mean, cov)


handles, legends = ax.get_legend_handles_labels()
plt.legend(handles[:8], legends[:8])
plt.show()
```



(c) PF

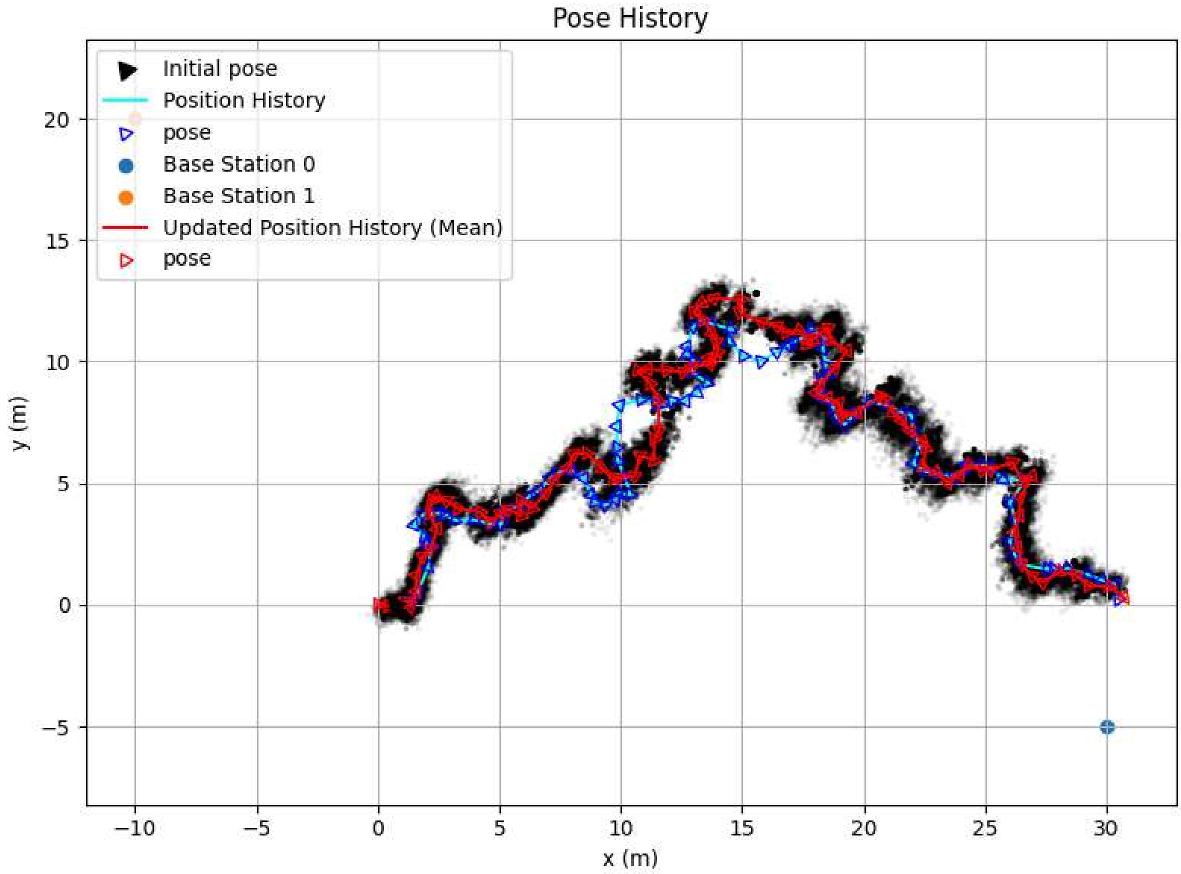
In [ ]:
```
def PF(n_steps, initial_state_mean, initial_state_cov, num_particles=1000, robot=rc
    """
    Runs a Particle Filter for state estimation.

    Returns:
    -------
    particle_history : np.ndarray
        History of particle states throughout the filtering process.
    weights : np.ndarray
        Final particle weights after all filtering steps.
    """
```

```
    np.random.seed(42)
    particle_history = np.zeros((n_steps + 1, num_particles, initial_state_mean.sha
    particles = np.random.multivariate_normal(initial_state_mean, initial_state_cov
    weights = np.ones(num_particles) / num_particles
    particle_history[0] = particles

    # Measurement likelihood function
    measurement_likelihood = lambda y, x: np.exp(-0.5 * (y - robot.noiseless_measur

    for t in range(1, n_steps + 1):
        predicted_particles = np.zeros_like(particles)
        predicted_weights = np.zeros_like(weights)

        for i in range(num_particles):
            predicted_particles[i], _ = robot.noisy_dynamics_step(particles[i], t=t
            predicted_weights[i] = measurement_likelihood(measurement_history[t-1],

        # Normalize weights
        weights = predicted_weights / np.sum(predicted_weights)

        # Resampling step
        resample_indices = np.random.choice(num_particles, size=num_particles, p=we
        particles = predicted_particles[resample_indices]
        particle_history[t] = particles

        # Reset weights after resampling
        weights = np.ones(num_particles) / num_particles

    return particle_history, weights
p_hist, _ = PF(n_steps, mu_0, sigma_0)
```

Plot PF

In [ ]:

```
# PF
# Plot true pose
fig_PF = robot.plot_pose_history(pos_hist, show_plot=False)
ax = fig_PF.axes[0]

# Plot updated traj
mu_update_PF = np.mean(p_hist, axis=1)
for t in range(1, mu_update.shape[0]):
    plt.scatter(p_hist[t, :, 0], p_hist[t, :, 1], color='k', alpha=0.05, s=3) # par
ax.plot(mu_update_PF[:,0], mu_update_PF[:,1], color='r', label="Updated Position Hi

# Plot the updated pose
labeled = False
for px, py, theta in mu_update_PF:
    plt.scatter(px, py,
                edgecolors='red', facecolors='none',
                s=50, zorder=2,
                marker=(3, 0, -90 + np.rad2deg(theta)),
                label="pose" if not labeled else None)
    labeled = True

handles, legends = ax.get_legend_handles_labels()
plt.legend(handles[:8], legends[:8])
plt.show()
```



(d) Plot

Plots for EKF, UKF, and PF are shown in the previous section

(e) Computation Time

The computation time for PF >> UKF ~= EKF. From the notebook output, we can see the time took for both UKF and EKF are nearly zero, which indicating the difference be in ~ ms. But for the PF, we can see it took 11.7s second for the computation which is much larger than other two filters.