Localization for Autonomous Driving

Prepared by OEI

30 Jan 2025

Contents

Course Overview

This course explores the fundamental concepts and practical implementation of precise localization systems for autonomous vehicles. By the end of this course, students will understand the theoretical foundations of localization and gain hands-on experience implementing core algorithms.

Learning Objectives

After completing this course, students will be able to:

- Explain the importance of precise localization in autonomous driving
- Understand the challenges and limitations of current localization systems
- Implement and tune Extended Kalman Filters for sensor fusion
- Develop Particle Filter-based localization systems
- Evaluate and compare different localization approaches

Module 0: Sensors and Filtering Fundamentals for Autonomous Driving

Introduction to Autonomous Vehicle Sensing

Before diving into filtering algorithms, we need to understand the sensors that provide our raw data and why their measurements need to be filtered. Autonomous vehicles rely on a diverse sensor suite because each sensor type has unique strengths and limitations. Understanding these characteristics is crucial for developing effective localization systems.

Camera Systems

Cameras are fundamental sensors in autonomous driving, providing rich visual information about the environment.

Key Characteristics:

- Resolution: Typically 1-8 megapixels
- Frame rate: Usually 30-60 fps
- Field of view: 60-120 degrees (can be wider with special lenses - fisheye lens for example)
- Operating wavelength: Visible light (400-700nm)

Strengths:

- High spatial resolution
- Rich semantic information
- Color information
- Relatively inexpensive
- Passive sensor (no energy emission)

Limitations:

- Highly dependent on lighting conditions
- Performance degrades in adverse weather
- No direct depth measurement
- Requires significant processing for 3D interpretation

Example of Camera Limitations:

Imagine driving at dusk. Your cameras might see a dark patch on the road ahead. Is it:

- A shadow from a tree?
- A pothole?
- A puddle of water?
- A patch of new asphalt?

This ambiguity illustrates why cameras alone aren't sufficient for autonomous driving.

Lidar (Light Detection and Ranging)

Lidar systems actively scan the environment by emitting laser pulses and measuring their return time to create precise 3D point clouds.

Key Characteristics:

- Point density: 100,000 - 2,000,000 points per second
- Range accuracy: ±2-3 cm
- Maximum range: 100-200 meters
- Rotation rate: 5-20 Hz
- Vertical resolution: 16-128 channels

Strengths:

- Precise 3D measurements
- Works in various lighting conditions
- Excellent spatial resolution
- Direct geometric measurements

Limitations:

- Performance degrades in adverse weather
- High cost
- Moving mechanical parts
- Large data volumes
- Limited range in adverse conditions

Example of Lidar Limitations:

Consider driving through heavy rain. Each raindrop reflects the laser pulse, creating false returns. A single scan might show hundreds of phantom obstacles that need to be filtered out.

Radar (Radio Detection and Ranging)

Radar systems emit radio waves and measure their reflections to detect objects and their velocities.

Key Characteristics:

- Frequency bands: 24 GHz, 77 GHz
- Range: up to 200+ meters
- Range accuracy: ±0.1-1.0 meters
- Velocity accuracy: ±0.1-1.0 m/s

Strengths:

- Works in all weather conditions
- Direct velocity measurements
- Long range capability
- Low cost compared to lidar

Limitations:

- Lower spatial resolution than lidar
- Limited angular resolution
- Complex signal processing required
- Multiple reflection issues

Example of Radar Limitations:

When approaching a metal guardrail on a curved road, radar might show multiple reflections, making it appear as if there are several obstacles at different distances. This "multipath" effect needs to be filtered out.

Inertial Measurement Unit (IMU)

IMUs measure acceleration and angular velocity using accelerometers and gyroscopes.

Key Characteristics:

- Update rate: 100-1000 Hz
- Accelerometer accuracy: ±0.01-1.0 m/s^2
- Gyroscope accuracy: ±0.01-1.0 deg/s
- Bias stability: 0.1-10 deg/hour

Strengths:

- Very high update rate
- Independent of external conditions
- Provides direct motion measurements
- No external infrastructure needed

Limitations:

- Drift over time (integration error)
- Bias instability
- Temperature sensitivity
- Requires calibration

Example of IMU Limitations:

Let's say you're tracking position using only IMU data. Even with a high-quality IMU, double-integrating acceleration to get position leads to position errors growing cubically with time. After just 60 seconds, you might be off by 100 meters or more.

Wheel Odometry

Wheel odometry measures vehicle motion through wheel rotation sensors.

Key Characteristics:

- Update rate: 50-100 Hz
- Resolution: 0.1-1.0 degrees of wheel rotation
- Accuracy: ±1-5% of distance traveled

Strengths:

- High update rate
- Direct velocity measurement
- Works in all weather conditions
- Low cost

Limitations:

- Wheel slip errors
- Requires accurate wheel radius
- Accumulates error over distance
- Cannot detect lateral slip

Example of Odometry Limitations:

When driving on a slippery road, wheels might spin without the vehicle moving, or the vehicle might slide sideways while the wheels roll normally. Either situation creates significant odometry errors.

Global Navigation Satellite System (GNSS)

GNSS provides absolute position information through satellite signals.

Key Characteristics:

- Update rate: 1-20 Hz
- Standard GPS accuracy: ±5-10 meters
- RTK GPS accuracy: ±1-2 centimeters
- Velocity accuracy: ±0.1-0.2 m/s

Strengths:

- Provides absolute position
- Global coverage

- No error accumulation
- Available in all weather

Limitations:

- Signal blockage in urban canyons
- Multipath effects
- Requires clear sky view
- Variable accuracy

Example of GNSS Limitations:

When driving through a city with tall buildings, GNSS signals reflect off buildings before reaching your receiver. These multipath effects can cause position errors of 50 meters or more.

Why Do We Need Filtering?

Let's examine three concrete scenarios that demonstrate why filtering is essential:

Scenario 1: Highway Driving

Imagine you're driving on a highway at 100 km/h. You have:

- GNSS updates at 1 Hz with $\pm$5m accuracy
- IMU measurements at 100 Hz with drift
- Wheel odometry at 50 Hz

Problem: Between GNSS updates (1 second), your vehicle travels 27.8 meters. You need to know your position during this interval for lane-keeping and safe following distance.

Solution: An Extended Kalman Filter can:

- Use IMU and odometry for high-rate position updates
- Correct drift using periodic GNSS measurements
- Maintain accurate position estimates between GNSS updates
- Provide uncertainty estimates for safety planning

Scenario 2: Parking Garage

You're navigating in an underground parking garage where:

- No GNSS signal is available
- Lidar sees concrete pillars and walls
- Wheel odometry is available
- IMU measurements continue

Problem: Without GNSS, position error accumulates. How do you maintain accurate positioning?

Solution: A Particle Filter can:

- Use building pillars as landmarks
- Match lidar scans to a known map
- Maintain multiple position hypotheses
- Gradually eliminate incorrect possibilities
- Handle the non-Gaussian uncertainty of indoor positioning

Scenario 3: Urban Canyon

You're driving in a dense urban environment where:

- GNSS signals reflect off buildings
- Some satellites are blocked
- Multiple possible GNSS solutions exist
- Visual landmarks are visible

Problem: GNSS reports multiple possible positions due to multipath, some off by 50+ meters.

Solution: An Extended Kalman Filter or Particle Filter can:

- Maintain consistent trajectory estimates
- Reject physically impossible GNSS jumps
- Use visual landmarks for correction
- Weight measurements based on their reliability
- Provide robust position estimates despite poor GNSS

Module 1: Introduction to Precise Localization

What is Precise Localization?

Precise localization refers to the process of determining an autonomous vehicle's exact position and orientation (pose) in a global reference frame with high accuracy. This typically means achieving centimeter-level accuracy in position and sub-degree accuracy in orientation.

Unlike basic GPS navigation used in consumer applications, autonomous vehicles require significantly higher precision because they need to:

- Stay within lane boundaries (typically 3-4 meters wide)
- Maintain safe distances from other vehicles and obstacles
- Make precise maneuvers for parking and navigation
- Align sensor data with high-definition maps

Why is Precise Localization Mandatory?

Precise localization forms the foundation of autonomous driving for several critical reasons:

1. Safety: Autonomous vehicles must know their exact position to maintain safe distances from obstacles and other vehicles. Even small positioning errors can lead to dangerous situations.

2. Decision Making: Path planning and decision-making algorithms rely on accurate positioning to determine appropriate actions. For example, deciding when to change lanes or make turns requires precise knowledge of the vehicle's position relative to road features.

3. Map Alignment: Modern autonomous vehicles use high-definition maps containing detailed information about lane markings, traffic signs, and road geometry. These maps are only useful if the vehicle can precisely align its position with the map data.

4. Regulatory Requirements: Emerging regulations for autonomous vehicles are likely to specify minimum positioning accuracy requirements for safety certification.

Current Challenges in Localization

Several factors make precise localization a continuing challenge:

1. Environmental Factors:
   - Urban canyons blocking or reflecting GNSS signals
   - Weather conditions affecting sensor performance
   - Dynamic environments with moving objects
   - Seasonal changes affecting visual and lidar-based features
2. Sensor Limitations:
   - GNSS accuracy limitations and multipath effects
   - IMU drift and bias
   - Camera limitations in poor lighting conditions
   - Cost constraints limiting sensor quality
3. Computational Challenges:
   - Real-time processing requirements
   - Resource constraints on embedded systems
   - Data fusion complexity
   - State estimation in non-linear systems

Module 2: Fundamental Concepts in State Estimation: Filtering Theory and application in Autonomous Driving

Part 1: Kalman Filtering Fundamentals

1.1 Introduction to Kalman Filtering

The Kalman filter is a recursive state estimator that provides an optimal solution for linear systems with Gaussian noise. For autonomous vehicle localization, we typically use the Extended Kalman Filter (EKF) to handle non-linear vehicle dynamics and measurement models.

Key concepts:

1. State Prediction: Using vehicle dynamics to predict next state
2. Measurement Update: Correcting predictions using sensor measurements
3. Covariance Propagation: Tracking uncertainty in estimates
4. Innovation: Difference between predicted and actual measurements

To understand Kalman Filtering deeply, let's break down its key components and build up to the complete algorithm.

The Core Idea

At its heart, the Kalman filter tries to answer a fundamental question: "Given noisy measurements and an imperfect understanding of how our system moves, what's our best guess about the true state of the system?"

Think of it like trying to track a car on a highway:

- You have a physics model that tells you how cars generally move (system model)
- You have GPS readings that give you noisy position measurements (measurement model)
- You want to combine both pieces of information optimally

1.2 State Space Representation

The first step in understanding Kalman filtering is the state space representation. For a vehicle, our state vector x might include:

$$x = [position\_x, position\_y, heading, velocity, angular\_velocity]^\top$$

The system evolution is described by:

$$x(k+1) = F(k)x(k) + B(k)u(k) + w(k)$$

where:

- $F(k)$ is the state transition matrix
- $B(k)$ is the control input matrix
- $u(k)$ is the control input
- $w(k)$ is process noise ~ N(0, Q)

Measurements are described by:

$$z(k) = H(k)x(k) + v(k)$$

where:

- $H(k)$ is the measurement matrix
- $v(k)$ is measurement noise ~ N(0, R)

1.3 The Gaussian Connection

The Kalman filter's magic comes from its use of Gaussian distributions. When we multiply two Gaussians, we get another Gaussian. This property allows us to:

1. Represent uncertainty using covariance matrices
2. Combine different sources of information optimally
3. Maintain computational efficiency

The state estimate is represented by:

- Mean ($\hat{x}$): our best guess of the true state
- Covariance (P): our uncertainty about that guess

1.4 The Kalman Filter Algorithm

The algorithm consists of two main steps:

Prediction Step (Time Update)

$$\hat{x}(k|k-1) = F(k)\hat{x}(k-1|k-1) + B(k)u(k)$$
$$P(k|k-1) = F(k)P(k-1|k-1)F(k)^\top + Q(k)$$

Intuitive interpretation:

- We project our previous estimate forward using our motion model
- Uncertainty grows during prediction (addition of Q)
- The further we predict, the more uncertain we become

Update Step (Measurement Update)

$$\begin{aligned} Innovation &: \tilde{y}(k) = z(k) - H(k)\hat{x}(k|k-1) \\ Innovationcovariance &: S(k) = H(k)P(k|k-1)H(k)^\top + R(k) \\ Kalmangain &: K(k) = P(k|k-1)H(k)^\top S(k)^{-1} \\ \\ Stateupdate &: \hat{x}(k|k) = \hat{x}(k|k-1) + K(k)\tilde{y}(k) \\ Covarianceupdate &: P(k|k) = (I - K(k)H(k))P(k|k-1) \end{aligned}$$

Intuitive interpretation:

- Innovation ($\tilde{y}$) is the difference between what we measured and what we expected to measure
- Kalman gain (K) determines how much we trust the measurement versus our prediction
- High measurement noise (R) leads to small K (trust predictions more)
- High prediction uncertainty (P) leads to large K (trust measurements more)

1.5 Extended Kalman Filter (EKF)

For autonomous vehicles, we need to handle nonlinear systems. The EKF extends the basic Kalman filter by linearizing around the current estimate.

Nonlinear System Model

$$\begin{aligned} x(k+1) &= f(x(k), u(k)) + w(k) \\ z(k) &= h(x(k)) + v(k) \end{aligned}$$

Linearization

We compute Jacobian matrices:

$$F(k) = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}(k|k)}$$
$$H(k) = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}(k|k-1)}$$

For our vehicle model, the Jacobians look like:

$$F = \begin{bmatrix} 1 & 0 & -v*dt*sin(\theta) & dt*cos(\theta) & 0 \\ 0 & 1 & v*dt*cos(\theta) & dt*sin(\theta) & 0 \\ 0 & 0 & 1 & 0 & dt \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

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

EKF Algorithm

The Prediction step of the algorithm:

$$\begin{aligned} \hat{x}(k|k-1) &= f(\hat{x}(k-1|k-1), u(k)) \\ P(k|k-1) &= F(k)P(k-1|k-1)F(k)^\top + Q(k) \end{aligned}$$

The Update step of the algorithm:

$$\begin{aligned} \tilde{y} &= z(k) - h(\hat{x}(k|k-1)) \\ S(k) &= H(k)P(k|k-1)H(k)^\top + R(k) \\ K(k) &= P(k|k-1)H(k)^\top S(k)^{-1} \\ \\ \hat{x}(k|k) &= \hat{x}(k|k-1) + K(k)\tilde{y}(k) \\ P(k|k) &= (I - K(k)H(k))P(k|k-1) \end{aligned}$$

Part 2: Particle Filtering

2.1 Introduction to Particle Filtering

Particle filters take a fundamentally different approach from Kalman filters. Instead of maintaining a single Gaussian estimate, they approximate the full probability distribution using a set of weighted samples (particles).

Key Advantages

1. Can represent any probability distribution
2. Handles severe nonlinearity naturally
3. Can maintain multiple hypotheses
4. No linearization required

Particle Filters are particularly useful for:

- Handling non-Gaussian noise
- Representing multi-modal distributions
- Incorporating non-linear constraints
- Dealing with kidnapped robot problems

2.2 Particle Filter Components

State Representation

Each particle represents a possible state:

$particle = [x, y, \theta, v, \omega, weight]$

The complete filter maintains N such particles, where N might be 1000-10000 depending on the application.

2.3 Core Algorithm

1. Initialization

```python
def initialize_particles(N):
    particles = []
    for i in range(N):
        # Sample initial state from prior distribution
        state = sample_from_prior()
        weight = 1.0/N
        particles.append([state, weight])
    return particles
```

Intuitive interpretation:

- Start with particles spread across likely initial states
- Equal weights represent uniform prior belief
- More particles in areas we think are more likely

3. Update Step

```python
def update_weights(particles, measurement):
    total_weight = 0
    for particle in particles:
        # Calculate measurement likelihood
        expected_measurement = measurement_model(particle.state)
        likelihood = gaussian_probability(measurement -
    expected_measurement, R)

        # Update weight
        particle.weight *= likelihood
        total_weight += particle.weight

    # Normalize weights
    for particle in particles:
        particle.weight /= total_weight
```

Intuitive interpretation:

- Particles that better match measurements get higher weights
- Weights represent our belief in each particle
- Normalization ensures weights sum to 1

4. Resampling

```python
def resample(particles):
    N = len(particles)
    # Systematic resampling
    cumsum_weights = cumulative_sum(particles.weights)
    new_particles = []

```

```
    u = random(0, 1/N)
    j = 0
    for i in range(N):
        while u > cumsum_weights[j]:
            j += 1
        new_particles.append(copy(particles[j]))
        new_particles[-1].weight = 1/N
        u += 1/N

    return new_particles
```

Intuitive interpretation:

- Particles with high weights are likely to be replicated
- Particles with low weights likely disappear
- Maintains focus on promising regions of state space

2.4 Important Considerations

Number of Particles

- More particles = better approximation but more computation
- Too few particles can lead to particle deprivation (no particles close to the correct state)
- Adaptive particle numbers can balance accuracy and speed

Resampling Strategy

- Resampling too often can reduce diversity

- Common approach: resample when effective number of particles drops below threshold

```
N_eff = 1 / sum(weights^2)
if N_eff < N/2:
  resample()
```

Proposal Distribution

The basic particle filter uses the motion model as proposal distribution, but we can do better:

- Use latest measurement to guide proposal
- Incorporate local optimization
- Use mixture proposals for robustness

Part 3: Practical Considerations

3.1 Choosing Between EKF and PF

EKF Advantages

- Computationally efficient

- Optimal for nearly linear systems
- Clear uncertainty representation
- Well-suited for sensor fusion

PF Advantages

- Handles arbitrary distributions
- No linearization required
- Can recover from kidnapped robot problem
- Better for highly nonlinear systems

3.2 Implementation Tips

For EKF:

1. Careful tuning of Q and R matrices is crucial
2. Watch for numerical stability in covariance updates
3. Consider square root filtering for better conditioning
4. Validate Jacobian matrices numerically

For PF:

1. Start with more particles than you think you need
2. Monitor effective sample size
3. Consider parallel implementation
4. Use efficient data structures for particles

3.3 Common Failure Modes

EKF Failures:

- Linearization errors in highly nonlinear regions
- Overconfident estimates leading to divergence
- Numerical instability in covariance updates
- Wrong noise parameters leading to filter inconsistency

PF Failures:

- Particle deprivation
- Sample impoverishment after resampling
- High computational cost with many particles
- Difficulty handling very precise measurements

3.4 Hybrid Approaches

Modern systems often combine multiple filtering approaches:

1. UKF for better nonlinear handling than EKF
2. Rao-Blackwellized particle filters
3. Multiple model approaches
4. Adaptive filtering techniques

Module 3: Extended Kalman Filter Implementation

Let's implement an EKF for fusing GNSS and IMU data. We'll break this down into steps:

```
classdef VehicleEKF
    properties
        % State vector [x, y, theta, v, omega]'
        state
        % State covariance matrix
        P
        % Process noise covariance
        Q
        % Measurement noise covariance
        R
        dt % Time step
    end

    methods
        function obj = VehicleEKF()
            % Initialize EKF parameters
            obj.state = zeros(5,1);
            obj.P = eye(5);
            obj.Q = diag([0.1, 0.1, 0.01, 0.1, 0.01]);
            obj.R = diag([1, 1, 0.1]); % GPS(x,y) and compass measurements
            obj.dt = 0.1; % 10Hz update rate
        end

        function [state_pred, P_pred] = predict(obj, acc, gyro)
            % Predict step using IMU measurements
            % acc: linear acceleration
            % gyro: angular velocity

            % Current state
            x = obj.state(1); y = obj.state(2);
            theta = obj.state(3);
            v = obj.state(4); omega = obj.state(5);

            % Predict next state
            state_pred = obj.state;
            state_pred(1) = x + v*cos(theta)*obj.dt;
            state_pred(2) = y + v*sin(theta)*obj.dt;
            state_pred(3) = theta + omega*obj.dt;
            state_pred(4) = v + acc*obj.dt;
            state_pred(5) = omega + gyro;
```

```
            % Compute Jacobian
            F = eye(5);
            F(1,3) = -v*sin(theta)*obj.dt;
            F(1,4) = cos(theta)*obj.dt;
            F(2,3) = v*cos(theta)*obj.dt;
            F(2,4) = sin(theta)*obj.dt;
            F(3,5) = obj.dt;

            % Predict covariance
            P_pred = F*obj.P*F' + obj.Q;
        end

        function [state_updated, P_updated] = update(obj, gps_x, gps_y,
    compass)
            % Update step using GPS and compass measurements

            % Measurement model
            z = [gps_x; gps_y; compass];
            h = [obj.state(1); obj.state(2); obj.state(3)];

            % Measurement Jacobian
            H = zeros(3,5);
            H(1:3,1:3) = eye(3);

            % Innovation
            y = z - h;
            % Wrap angle difference to [-pi, pi]
            y(3) = atan2(sin(y(3)), cos(y(3)));

            % Innovation covariance
            S = H*obj.P*H' + obj.R;

            % Kalman gain
            K = obj.P*H'/S;

            % Update state and covariance
            state_updated = obj.state + K*y;
            P_updated = (eye(5) - K*H)*obj.P;
        end
    end
end
```

Module 4: Particle Filter Implementation

Now let's implement a particle filter for the same localization problem:

```
classdef VehicleParticleFilter
    properties
        num_particles
        particles % Each particle: [x, y, theta, v, omega, weight]
        motion_noise
        measurement_noise
        dt
    end

    methods
        function obj = VehicleParticleFilter(n_particles)
            obj.num_particles = n_particles;
            obj.particles = zeros(n_particles, 6);
            % Initialize particles randomly
            obj.particles(:,1:2) = randn(n_particles,2)*10; % Position
            obj.particles(:,3) = randn(n_particles,1)*0.1; % Heading
            obj.particles(:,4:5) = zeros(n_particles,2); % Velocity
            obj.particles(:,6) = 1/n_particles; % Weights

            obj.motion_noise = [0.1, 0.1, 0.01, 0.1, 0.01];
            obj.measurement_noise = [1, 1, 0.1];
            obj.dt = 0.1;
        end

        function particles = predict(obj, acc, gyro)
            % Predict step using IMU measurements
            for i = 1:obj.num_particles
                % Extract state
                x = obj.particles(i,1); y = obj.particles(i,2);
                theta = obj.particles(i,3);
                v = obj.particles(i,4); omega = obj.particles(i,5);

                % Add control inputs with noise
                v_noisy = v + acc*obj.dt + randn*obj.motion_noise(4);
                omega_noisy = omega + gyro + randn*obj.motion_noise(5);

                % Update particle state
                obj.particles(i,1) = x + v_noisy*cos(theta)*obj.dt + randn*
    obj.motion_noise(1);
                obj.particles(i,2) = y + v_noisy*sin(theta)*obj.dt + randn*
    obj.motion_noise(2);
                obj.particles(i,3) = theta + omega_noisy*obj.dt + randn*obj
```

```
    .motion_noise(3);
                obj.particles(i,4) = v_noisy;
                obj.particles(i,5) = omega_noisy;
            end
        end

        function particles = update(obj, gps_x, gps_y, compass)
            % Update weights based on measurements
            for i = 1:obj.num_particles
                % Compute measurement likelihood
                pos_error = norm([obj.particles(i,1) - gps_x;
                                obj.particles(i,2) - gps_y]);
                angle_error = abs(atan2(sin(obj.particles(i,3) - compass),
    ...
                                cos(obj.particles(i,3) - compass)));

                % Update weight using Gaussian likelihood
                likelihood = exp(-0.5*(pos_error^2/obj.measurement_noise(1)
    ^2 + ...
                                angle_error^2/obj.measurement_noise(3)^2));
                obj.particles(i,6) = obj.particles(i,6) * likelihood;
            end

            % Normalize weights
            obj.particles(:,6) = obj.particles(:,6) / sum(obj.particles
    (:,6));

            % Resample if effective number of particles is too low
            Neff = 1/sum(obj.particles(:,6).^2);
            if Neff < obj.num_particles/2
                obj.resample();
            end
        end

        function resample(obj)
            % Systematic resampling
            cumsum_weights = cumsum(obj.particles(:,6));
            new_particles = zeros(size(obj.particles));

            % Generate systematic samples
            u = (rand + (0:obj.num_particles-1))/obj.num_particles;
            j = 1;
            for i = 1:obj.num_particles
                while u(i) > cumsum_weights(j)
                    j = j + 1;
                end
                new_particles(i,:) = obj.particles(j,:);
                new_particles(i,6) = 1/obj.num_particles;
            end

            obj.particles = new_particles;
        end
    end
end
```

Module 5: Visualization and Analysis Tools

These visualization tools provide several benefits for students:

1. Real-time visualization of filter performance
2. Visual comparison between true trajectory and estimates
3. Particle distribution visualization for understanding filter behavior
4. Error ellipse visualization for EKF uncertainty
5. Quantitative performance analysis tools

The visualizers can help students:

- Debug their implementations
- Understand the effects of different parameter settings
- Compare filter performance in different scenarios
- Develop intuition about filter behavior

5.1 EKF Visualization

```
classdef EKFVisualizer
    properties
        figure_handle
        trajectory_plot
        estimate_plot
        uncertainty_plot
        particles_plot
    end

    methods
        function obj = EKFVisualizer()
            % Create main figure
            obj.figure_handle = figure('Name', 'EKF Localization');
            hold on;
            grid on;

            % Initialize plot handles
            obj.trajectory_plot = plot(NaN, NaN, 'k-', 'LineWidth', 2, 'DisplayName', 'True Path');
            obj.estimate_plot = plot(NaN, NaN, 'r--', 'LineWidth', 2, 'DisplayName', 'EKF Estimate');
            obj.uncertainty_plot = plot(NaN, NaN, 'r:', 'LineWidth', 1);

            xlabel('X Position (m)');
```

```
            ylabel('Y Position (m)');
            title('EKF Localization Visualization');
            legend('show');
            axis equal;
        end

        function update(obj, true_pose, ekf_state, ekf_cov)
            % Update true trajectory
            x_true = get(obj.trajectory_plot, 'XData');
            y_true = get(obj.trajectory_plot, 'YData');
            set(obj.trajectory_plot, 'XData', [x_true true_pose(1)], ...
                                     'YData', [y_true true_pose(2)]);

            % Update EKF estimate
            x_est = get(obj.estimate_plot, 'XData');
            y_est = get(obj.estimate_plot, 'YData');
            set(obj.estimate_plot, 'XData', [x_est ekf_state(1)], ...
                                   'YData', [y_est ekf_state(2)]);

            % Draw uncertainty ellipse
            [X, Y] = obj.get_error_ellipse(ekf_state(1:2), ekf_cov(1:2,1:2)
    );
            set(obj.uncertainty_plot, 'XData', X, 'YData', Y);

            % Update view
            axis equal;
            drawnow;
        end

        function [X, Y] = get_error_ellipse(obj, mean, cov)
            % Generate points for 95% confidence ellipse
            theta = linspace(0, 2*pi, 100);
            chi2 = chi2inv(0.95, 2);
            [eigvec, eigval] = eig(cov);

            % Scale eigenvalues for chi-square distribution
            xy = [cos(theta); sin(theta)];
            xy = sqrt(chi2) * sqrt(eigval) * xy;

            % Rotate and translate ellipse
            xy = eigvec * xy;
            X = xy(1,:) + mean(1);
            Y = xy(2,:) + mean(2);
        end
    end
end
```

5.2 Particle Filter Visualization

```
classdef ParticleFilterVisualizer
    properties
        figure_handle
        trajectory_plot
        particles_scatter
        estimate_plot
        current_pose_plot
```

```
    end

    methods
        function obj = ParticleFilterVisualizer()
            % Create main figure
            obj.figure_handle = figure('Name', 'Particle Filter Localization');
            hold on;
            grid on;

            % Initialize plot handles
            obj.trajectory_plot = plot(NaN, NaN, 'k-', 'LineWidth', 2, 'DisplayName', 'True Path');
            obj.particles_scatter = scatter([], [], 20, 'b.', 'DisplayName', 'Particles');
            obj.estimate_plot = plot(NaN, NaN, 'r--', 'LineWidth', 2, 'DisplayName', 'PF Estimate');
            obj.current_pose_plot = quiver(NaN, NaN, NaN, NaN, 'g', 'LineWidth', 2, ...
                                          'MaxHeadSize', 0.5, 'DisplayName', 'Current Pose');

            xlabel('X Position (m)');
            ylabel('Y Position (m)');
            title('Particle Filter Localization Visualization');
            legend('show');
            axis equal;
        end

        function update(obj, true_pose, particles)
            % Update true trajectory
            x_true = get(obj.trajectory_plot, 'XData');
            y_true = get(obj.trajectory_plot, 'YData');
            set(obj.trajectory_plot, 'XData', [x_true true_pose(1)], ...
                                    'YData', [y_true true_pose(2)]);

            % Update particles
            set(obj.particles_scatter, 'XData', particles(:,1), ...
                                      'YData', particles(:,2));

            % Calculate and update weighted mean estimate
            weights = particles(:,6);
            est_x = sum(particles(:,1) .* weights);
            est_y = sum(particles(:,2) .* weights);
            est_theta = atan2(sum(sin(particles(:,3)) .* weights), ...
                                sum(cos(particles(:,3)) .* weights));

            % Update estimate trajectory
            x_est = get(obj.estimate_plot, 'XData');
            y_est = get(obj.estimate_plot, 'YData');
            set(obj.estimate_plot, 'XData', [x_est est_x], ...
                                  'YData', [y_est est_y]);

            % Update current pose arrow
            arrow_length = 1.0; % meters
            set(obj.current_pose_plot, 'XData', est_x, ...
                                      'YData', est_y, ...
```

```
                                                'UData', arrow_length * cos(est_theta)
    , ...
                                                'VData', arrow_length * sin(est_theta)
    );

            % Update view
            axis equal;
            drawnow;
        end
    end
end
```

5.3 Example Usage and Simulation

Here's how to use these visualizers in a complete simulation:

```
% Simulation parameters
sim_time = 60; % seconds
dt = 0.1;      % time step
steps = sim_time/dt;

% Initialize true vehicle state [x, y, theta, v, omega]
true_state = zeros(5, 1);

% Initialize filters
ekf = VehicleEKF();
pf = VehicleParticleFilter(1000);

% Initialize visualizers
ekf_vis = EKFVisualizer();
pf_vis = ParticleFilterVisualizer();

% Simulation loop
for i = 1:steps
    % Generate true motion (circular trajectory example)
    R = 10; % radius
    omega = 0.1; % angular velocity
    v = R * omega; % linear velocity

    % True motion
    true_state(4) = v;
    true_state(5) = omega;
    true_state(1) = true_state(1) - v*sin(true_state(3))*dt;
    true_state(2) = true_state(2) + v*cos(true_state(3))*dt;
    true_state(3) = true_state(3) + omega*dt;

    % Generate noisy sensor measurements
    acc = v*omega + randn*0.1;
    gyro = omega + randn*0.01;
    gps_x = true_state(1) + randn*1.0;
    gps_y = true_state(2) + randn*1.0;
    compass = true_state(3) + randn*0.1;

    % Update EKF
    [state_pred, P_pred] = ekf.predict(acc, gyro);
    [state_updated, P_updated] = ekf.update(gps_x, gps_y, compass);
```

```
    ekf.state = state_updated;
    ekf.P = P_updated;

    % Update Particle Filter
    pf.predict(acc, gyro);
    pf.update(gps_x, gps_y, compass);

    % Update visualizations
    ekf_vis.update(true_state(1:3), ekf.state, ekf.P);
    pf_vis.update(true_state(1:3), pf.particles);

    % Small pause to control simulation speed
    pause(0.01);
end
```

5.4 Performance Analysis Tools

```
classdef LocalizationAnalyzer
    methods (Static)
        function [rmse_ekf, rmse_pf] = calculate_rmse(true_traj, ekf_traj, pf_traj)
            % Calculate Root Mean Square Error for both filters
            ekf_errors = sqrt(sum((true_traj - ekf_traj).^2, 2));
            pf_errors = sqrt(sum((true_traj - pf_traj).^2, 2));

            rmse_ekf = sqrt(mean(ekf_errors.^2));
            rmse_pf = sqrt(mean(pf_errors.^2));
        end

        function plot_error_comparison(time, true_traj, ekf_traj, pf_traj)
            figure('Name', 'Localization Error Comparison');

            % Position errors
            ekf_pos_error = sqrt(sum((true_traj(:,1:2) - ekf_traj(:,1:2)).^2, 2));
            pf_pos_error = sqrt(sum((true_traj(:,1:2) - pf_traj(:,1:2)).^2, 2));

            % Heading errors
            ekf_heading_error = abs(angdiff(true_traj(:,3), ekf_traj(:,3)));
            pf_heading_error = abs(angdiff(true_traj(:,3), pf_traj(:,3)));

            % Plot position errors
            subplot(2,1,1);
            plot(time, ekf_pos_error, 'r-', 'DisplayName', 'EKF');
            hold on;
            plot(time, pf_pos_error, 'b-', 'DisplayName', 'PF');
            xlabel('Time (s)');
            ylabel('Position Error (m)');
            title('Position Error Comparison');
            legend('show');
            grid on;

            % Plot heading errors
            subplot(2,1,2);
```

```
            plot(time, rad2deg(ekf_heading_error), 'r-', 'DisplayName', 'EKF');
            hold on;
            plot(time, rad2deg(pf_heading_error), 'b-', 'DisplayName', 'PF');
            xlabel('Time (s)');
            ylabel('Heading Error (degrees)');
            title('Heading Error Comparison');
            legend('show');
            grid on;
        end
    end
end
```

Module 6: Future Challenges in Precise Localization

This section provides students with a comprehensive understanding of the challenges they may face in their future careers. It emphasizes both technical and non-technical aspects, helping them develop a holistic view of the field.

The field of autonomous vehicle localization continues to evolve, presenting several significant challenges that researchers and engineers must address. Understanding these challenges helps us anticipate future developments and guide research directions.

Environmental Resilience

One of the most pressing challenges involves creating localization systems that maintain high precision across all environmental conditions. Current systems often struggle with extreme weather scenarios and environmental variations that we frequently encounter in real-world driving situations.

Rain and snow present particular difficulties because they affect multiple sensing modalities simultaneously. Water droplets can scatter lidar beams, create noise in camera images, and affect radar returns. Snow accumulation can fundamentally alter the appearance of the environment, making it difficult to match sensor data with stored maps. Moreover, wet road surfaces can create reflections that confuse both sensors and recognition algorithms.

Beyond precipitation, we must also consider other environmental factors. Fog and dust can severely limit visibility and sensor range. Seasonal changes affect vegetation appearance and structure, which many mapping systems use as landmarks. Even the position of the sun can create challenging situations, such as direct glare or strong shadows that affect camera-based localization.

Dynamic Environment Adaptation

Our current localization approaches often assume a relatively static environment, but real-world environments are increasingly dynamic. Construction work temporarily alters road geometry and landmarks. New buildings appear while others are demolished. Trees grow or are removed. These changes can quickly make high-definition maps outdated.

Future localization systems will need to:

- Detect and adapt to environmental changes in real-time
- Update their internal representations dynamically
- Share environmental updates across vehicle fleets
- Maintain accurate localization even when significant portions of the map have changed

Multi-Vehicle Collaborative Localization

As more autonomous vehicles enter our roads, we have an opportunity to improve localization accuracy through collaboration. However, this introduces new challenges:

The system must handle relative localization between vehicles while maintaining global consistency. This requires sophisticated data fusion algorithms that can combine information from multiple sources while accounting for communication delays and uncertainties in relative measurements. Additionally, the system needs to maintain privacy and security while sharing location data between vehicles.

Urban Canyon Challenges

Dense urban environments present unique challenges for localization systems. Tall buildings create complex multipath effects for GNSS signals and can block satellite visibility entirely. They also create "urban canyons" that affect other sensors:

- Limited sky view affects not just GNSS but also visual odometry systems that use the sky for orientation
- Complex reflection patterns create ghost targets in radar systems
- Glass buildings and reflective surfaces confuse both lidar and camera systems

Future systems will need to develop robust methods for handling these challenging urban environments, possibly by combining traditional sensing with novel approaches like:

- 5G/6G cellular positioning
- Urban magnetic field mapping
- Underground infrastructure mapping
- Building structural features as landmarks

Semantic Understanding Integration

Future localization systems will likely need deeper semantic understanding of their environment. Rather than just matching geometric features, systems should understand what they're looking at. This semantic understanding can help:

- Distinguish between permanent and temporary features
- Predict which elements of the environment are likely to change
- Identify reliable landmarks even when their appearance changes
- Handle seasonal variations more robustly

Computational Efficiency

As localization systems become more sophisticated, managing computational resources becomes increasingly challenging. Future systems must balance:

- Real-time performance requirements
- Power consumption constraints
- Hardware cost limitations
- System reliability and redundancy

This balance becomes particularly important as we move toward electric vehicles, where power consumption directly affects vehicle range.

Map Data Management

The management of high-definition maps presents several ongoing challenges:

- Storage and transmission of massive amounts of map data
- Efficient updates and version control
- Handling areas with poor connectivity
- Maintaining map accuracy across seasons and construction

Future systems will need to develop more efficient ways to store and update map data, possibly using techniques like:

- Progressive map loading based on location and context
- Automatic map generation and update from vehicle sensor data
- Compressed map representations that maintain necessary precision
- Distributed map storage and updating across vehicle fleets

Sensor Fusion Evolution

As new sensor technologies emerge, localization systems must evolve to incorporate them effectively. Future challenges include:

- Integration of novel sensor types (quantum sensors, new RF technologies)
- Optimal sensor selection based on conditions and requirements
- Graceful degradation when sensors fail
- Cost-effective sensor configurations for different vehicle types

Regulatory Compliance

As autonomous vehicles become more common, regulatory requirements for localization accuracy and reliability will likely become more stringent. Future systems will need to:

- Provide guaranteed minimum accuracy levels
- Demonstrate reliability in safety-critical situations
- Maintain auditable performance records
- Meet different requirements across jurisdictions

Infrastructure Dependency

A key challenge for the future is determining the right balance between vehicle autonomy and infrastructure dependency. While some propose extensive infrastructure support (like precision positioning beacons or magnetic markers), others advocate for fully autonomous solutions. Future systems must consider:

- Cost of infrastructure deployment and maintenance
- Reliability of infrastructure-dependent solutions
- Transition strategies for mixed infrastructure environments
- Backup systems for infrastructure failures

Social and Ethical Considerations

Finally, we must consider the broader implications of precise localization:

- Privacy concerns regarding location tracking
- Data ownership and sharing
- Security against spoofing and jamming
- Fair access to positioning infrastructure
- Environmental impact of required infrastructure

These challenges represent significant opportunities for innovation in the field of autonomous vehicle localization. Success will require advances in multiple domains, from sensor technology and algorithms to system architecture and infrastructure design. Understanding these challenges helps guide research and development efforts toward creating more robust and reliable localization systems for the future of autonomous driving.

Module 7: Sensor-Based Localization and Fusion

7.1 Lidar-Based Localization

Lidar-based localization typically achieves centimeter-level accuracy by matching current lidar scans against either a pre-built map or previous scans. Let's explore the main approaches and their implementations.

7.1.1 Iterative Closest Point (ICP)

ICP is a fundamental algorithm for aligning point clouds. The basic idea is to iteratively minimize the distance between corresponding points in two point clouds. Here's how it works:

```
function [R, t] = icp_localization(current_scan, reference_scan,
    initial_guess)
    % Initialize transformation
    R = initial_guess.rotation;
    t = initial_guess.translation;

    for iteration = 1:max_iterations
        % 1. Find closest points
        correspondences = find_nearest_neighbors(current_scan,
    reference_scan);

        % 2. Compute centroids
        p_centroid = mean(current_scan(correspondences.query,:));
        q_centroid = mean(reference_scan(correspondences.ref,:));

        % 3. Center the point sets
        p_centered = current_scan(correspondences.query,:) - p_centroid;
        q_centered = reference_scan(correspondences.ref,:) - q_centroid;

        % 4. Compute optimal rotation
        H = p_centered' * q_centered;
        [U, ~, V] = svd(H);
        R_update = V * U';

        % 5. Update transformation
        R = R_update * R;
        t = q_centroid' - R * p_centroid';

        % 6. Check convergence
```

```
        if norm(R_update - eye(3)) < threshold
            break;
        end
    end
end
```

Real-world implementations include several optimizations:

1. Point selection strategies to handle outliers
2. Multi-resolution approaches for faster convergence
3. Robust error metrics like point-to-plane distance
4. Efficient nearest neighbor search using k-d trees

7.1.2 Normal Distributions Transform (NDT)

NDT represents the environment as a grid of Gaussian distributions, which provides a smoother optimization surface than ICP. Here's the core algorithm:

```
function [pose] = ndt_localization(current_scan, ndt_map, initial_pose)
    pose = initial_pose;

    for iteration = 1:max_iterations
        % Transform scan to current pose estimate
        transformed_scan = transform_scan(current_scan, pose);

        % For each point, compute score and derivatives
        score = 0;
        gradient = zeros(6,1);
        hessian = zeros(6,6);

        for point = transformed_scan
            % Get cell distribution
            cell = find_cell(ndt_map, point);

            % Compute probability
            d = point - cell.mean;
            exp_term = exp(-0.5 * d' * cell.inv_cov * d);
            score = score + exp_term;

            % Compute derivatives for optimization
            % [Complex derivative calculations omitted for brevity]
        end

        % Update pose using Newton's method
        pose_update = -hessian \ gradient;
        pose = pose_update + pose;

        if norm(pose_update) < threshold
            break;
        end
    end
end
```

7.1.3 Semantic Lidar Localization

Modern approaches incorporate semantic information to improve robustness:

```
function [pose] = semantic_lidar_localization(current_scan, semantic_map)
    % Extract semantic features from current scan
    pole_features = extract_poles(current_scan);
    building_corners = extract_corners(current_scan);
    ground_plane = extract_ground(current_scan);

    % Match semantic features with map
    matched_features = match_semantic_features(
        pole_features,
        building_corners,
        ground_plane,
        semantic_map
    );

    % Optimize pose using semantic constraints
    pose = optimize_semantic_pose(matched_features);
end
```

7.2 Radar-Based Localization

Radar presents unique challenges and opportunities for localization due to its all-weather capability but lower resolution.

7.2.1 Radar Grid Maps

One effective approach converts radar measurements into occupancy grid maps:

```
function [grid_map] = create_radar_grid(radar_measurements)
    % Initialize probabilistic grid
    grid_map = ones(grid_size) * 0.5;  % Unknown state

    for measurement = radar_measurements
        % Convert radar return to probability
        p_occupied = compute_occupancy_probability(
            measurement.power,
            measurement.range,
            measurement.doppler
        );

        % Update grid using log-odds
        grid_idx = world_to_grid(measurement.position);
        grid_map(grid_idx) = update_log_odds(grid_map(grid_idx), p_occupied);
    end
end
```

7.2.2 Radar Feature Tracking

For dynamic environments, tracking stable radar features improves localization:

```
function [features] = track_radar_features(radar_scan)
    % Extract potential features
    peaks = find_radar_peaks(radar_scan);

    % Classify features by stability
    static_features = classify_static_features(peaks);

    % Track features over time using JPDA
    tracked_features = joint_probabilistic_association(
        static_features,
        previous_features
    );
end
```

7.3 Camera-Based Localization

Camera localization typically involves either visual odometry or visual place recognition.

7.3.1 Visual Odometry

Modern visual odometry often uses direct methods:

```
function [pose_delta] = direct_visual_odometry(image1, image2, depth1)
    % Initialize pose optimization
    pose_delta = eye(4);

    for pyramid_level = max_pyramid:-1:1
        % Build image pyramid
        [I1_pyr, I2_pyr] = build_pyramid(image1, image2, pyramid_level);

        for iteration = 1:max_iterations
            % Compute residuals and jacobians
            [residuals, jacobians] = compute_photometric_error(
                I1_pyr, I2_pyr, depth1, pose_delta
            );

            % Solve normal equations
            update = solve_gauss_newton(jacobians, residuals);
            pose_delta = pose_delta * exp(update);
        end
    end
end
```

7.3.2 Visual Place Recognition

For global localization, we can use learned features:

```
function [location] = visual_place_recognition(query_image, database)
    % Extract global image descriptor
    descriptor = extract_neural_descriptor(query_image);

    % Find nearest neighbors in database
    candidates = find_nearest_neighbors(descriptor, database);

```

```
    % Geometric verification
    for candidate = candidates
        matches = match_local_features(query_image, candidate.image);
        if verify_geometric_consistency(matches)
            location = candidate.location;
            break;
        end
    end
end
```

7.4 Global Precise Localization

Achieving robust global localization requires combining multiple approaches:

7.4.1 Multi-Layer Maps

```
class GlobalLocalizationSystem
    properties
        semantic_map
        geometric_map
        radar_map
        visual_database
    end

    methods
        function initialize_global(obj)
            % Coarse localization using GNSS
            pose = get_gnss_position();

            % Visual place recognition
            visual_pose = obj.visual_database.query(current_image);

            % Radar-based refinement
            radar_pose = obj.radar_map.match(current_radar);

            % Final refinement using lidar
            precise_pose = obj.geometric_map.icp_match(current_lidar);
        end
end
```

7.4.2 Hierarchical Localization

```
function [global_pose] = hierarchical_localize()
    % Level 1: Coarse localization (+/-5m)
    coarse_pose = gnss_locate();

    % Level 2: Area recognition (+/-2m)
    area_pose = visual_place_recognition(coarse_pose);

    % Level 3: Geometric alignment (+/-0.1m)
    precise_pose = geometric_refinement(area_pose);

    % Level 4: Continuous tracking
    global_pose = continuous_track(precise_pose);
```

```
end
```

7.5 Sensor Fusion for High Safety

7.5.1 Multi-Hypothesis Tracking

```
class SafetyLocalization
    properties
        hypotheses  % Multiple pose hypotheses
        sensors     % Available sensors
        safety_level
    end

    methods
        function update(obj, sensor_data)
            % Update each hypothesis
            for hypothesis = obj.hypotheses
                % Independent updates per sensor
                for sensor = obj.sensors
                    sensor_update = sensor.update(hypothesis);
                    hypothesis.incorporate(sensor_update);
                end

                % Compute hypothesis probability
                hypothesis.probability = compute_probability(hypothesis);
            end

            % Safety checks
            obj.safety_level = assess_safety(obj.hypotheses);

            if obj.safety_level < safety_threshold
                trigger_safety_response();
            end
        end
    end
end
```

7.5.2 Fault Detection and Isolation

```
function [valid_sensors] = detect_sensor_faults(sensor_measurements)
    valid_sensors = {};

    % Cross-validation between sensors
    for sensor_i = sensors
        % Compare with other sensors
        conflicts = 0;
        for sensor_j = sensors
            if sensor_i ~= sensor_j
                if measurement_conflict(sensor_i, sensor_j)
                    conflicts = conflicts + 1;
                end
            end
        end

        % Add to valid sensors if consistent
```

```
        if conflicts < fault_threshold
            valid_sensors.add(sensor_i);
        end
    end
end
```

7.5.3 Safety-Weighted Fusion

```
function [fused_state] = safety_fusion(sensor_states)
    % Initialize covariance intersection
    fused_state = zeros(state_dim, 1);
    fused_covariance = zeros(state_dim, state_dim);

    % Compute safety weights
    weights = compute_safety_weights(sensor_states);

    % Covariance intersection with safety weights
    for i = 1:length(sensor_states)
        state = sensor_states(i).state;
        covariance = sensor_states(i).covariance;
        weight = weights(i);

        % Update fusion using covariance intersection
        [fused_state, fused_covariance] = ...
            covariance_intersection(
                fused_state,
                fused_covariance,
                state,
                covariance,
                weight
            );
    end
end
```

Module 8: Practical Exercises

Exercise 1: EKF Implementation

Implement the EKF for a simple 2D robot moving in a plane:

1. Generate simulated GNSS and IMU data
2. Implement the prediction step using IMU data
3. Implement the update step using GNSS measurements
4. Visualize the results and compare with ground truth

Solution

Code

A complete MATLAB implementation of an Extended Kalman Filter for a 2D robot.

Main starting script

```
% Main script for 2D Robot EKF Implementation

% Parameters
dt = 0.1;  % Time step (s)
T = 100;   % Total simulation time (s)
t = 0:dt:T;
n = length(t);

% Process noise parameters
sigma_a = 0.1;  % Acceleration noise
sigma_w = 0.01; % Angular rate noise

% Measurement noise parameters
sigma_gps = 1.0;  % GPS position noise (m)

% Initialize true state
x_true = zeros(5, n);  % [x, y, theta, v, w]
x_true(:,1) = [0; 0; 0; 0; 0];

% Initialize EKF state and covariance
x_est = zeros(5, n);
x_est(:,1) = x_true(:,1) + [0.5; 0.5; 0.1; 0; 0];  % Initial estimate with
    some error
P = diag([1, 1, 0.1, 0.1, 0.1]);  % Initial covariance

% Generate synthetic data
[imu_data, gps_data, x_true] = generate_synthetic_data(x_true, dt, n,
    sigma_a, sigma_w, sigma_gps);
```

```
% Process noise covariance
Q = diag([sigma_a^2, sigma_a^2, sigma_w^2, sigma_a^2, sigma_w^2]);

% Measurement noise covariance
R = eye(2) * sigma_gps^2;

% Main EKF loop
for k = 2:n
    % Prediction step using IMU
    [x_est(:,k), P] = prediction_step(x_est(:,k-1), P, imu_data(:,k), dt, Q
    );

    % Update step using GPS (if available)
    if mod(k, 10) == 0  % GPS update at 1 Hz (assuming IMU at 10 Hz)
        [x_est(:,k), P] = update_step(x_est(:,k), P, gps_data(:,k), R);
    end
end

% Visualize results
visualize_results(t, x_true, x_est, gps_data);
```

Generate synthetic data

```
% Function to generate synthetic data
function [imu_data, gps_data, x_true] = generate_synthetic_data(x_true, dt,
     n, sigma_a, sigma_w, sigma_gps)
    % Initialize data arrays
    imu_data = zeros(2, n);  % [a, w]
    gps_data = zeros(2, n);  % [x, y]

    % Generate true trajectory (circle + straight line)
    for k = 2:n
        if k < n/2
            % Circular motion
            x_true(4,k) = 1;  % Constant velocity
            x_true(5,k) = 0.2;  % Constant angular velocity
        else
            % Straight line
            x_true(4,k) = 1;  % Constant velocity
            x_true(5,k) = 0;  % No angular velocity
        end

        % Update true state
        x_true(1,k) = x_true(1,k-1) + x_true(4,k-1)*cos(x_true(3,k-1))*dt;
        x_true(2,k) = x_true(2,k-1) + x_true(4,k-1)*sin(x_true(3,k-1))*dt;
        x_true(3,k) = x_true(3,k-1) + x_true(5,k-1)*dt;

        % Generate noisy IMU measurements
        imu_data(1,k) = (x_true(4,k) - x_true(4,k-1))/dt + randn*sigma_a;
        imu_data(2,k) = x_true(5,k) + randn*sigma_w;

        % Generate noisy GPS measurements
        gps_data(1,k) = x_true(1,k) + randn*sigma_gps;
        gps_data(2,k) = x_true(2,k) + randn*sigma_gps;
    end
end
```

Prediction function

```
% Prediction step function
function [x_pred, P_pred] = prediction_step(x, P, imu, dt, Q)
    % State: [x, y, theta, v, w]
    % Input: [a, w]

    % Predict state
    x_pred = zeros(5,1);
    x_pred(1) = x(1) + x(4)*cos(x(3))*dt;
    x_pred(2) = x(2) + x(4)*sin(x(3))*dt;
    x_pred(3) = x(3) + x(5)*dt;
    x_pred(4) = x(4) + imu(1)*dt;
    x_pred(5) = imu(2);

    % Compute Jacobian
    F = eye(5);
    F(1,3) = -x(4)*sin(x(3))*dt;
    F(1,4) = cos(x(3))*dt;
    F(2,3) = x(4)*cos(x(3))*dt;
    F(2,4) = sin(x(3))*dt;
    F(3,5) = dt;

    % Predict covariance
    P_pred = F*P*F' + Q;
end
```

Update function

```
% Update step function
function [x_update, P_update] = update_step(x, P, gps, R)
    % Measurement model
    h = x(1:2);  % Position measurements only

    % Measurement Jacobian
    H = [1 0 0 0 0;
         0 1 0 0 0];

    % Innovation
    y = gps - h;

    % Kalman gain
    S = H*P*H' + R;
    K = P*H'/S;

    % Update state and covariance
    x_update = x + K*y;
    P_update = (eye(5) - K*H)*P;
end
```

Visualization function

```
% Visualization function
function visualize_results(t, x_true, x_est, gps_data)
    figure('Position', [100, 100, 1200, 400]);

    % Trajectory plot
    subplot(1,2,1);
    plot(x_true(1,:), x_true(2,:), 'g-', 'LineWidth', 2, 'DisplayName', '
   True');
    hold on;
```

```
    plot(x_est(1,:), x_est(2,:), 'b--', 'LineWidth', 2, 'DisplayName', '
   Estimated');
    plot(gps_data(1,1:10:end), gps_data(2,1:10:end), 'r.', 'MarkerSize',
   10, 'DisplayName', 'GPS');
    grid on;
    legend('Location', 'best');
    xlabel('X (m)');
    ylabel('Y (m)');
    title('Robot Trajectory');
    axis equal;

    % Error plot
    subplot(1,2,2);
    pos_error = sqrt((x_true(1,:)-x_est(1,:)).^2 + (x_true(2,:)-x_est(2,:))
   .^2);
    plot(t, pos_error, 'b-', 'LineWidth', 2);
    grid on;
    xlabel('Time (s)');
    ylabel('Position Error (m)');
    title('Position Error');
end
```

Code components breakdown

1. Main Script:
   - Sets up simulation parameters
   - Initializes state vectors and covariance matrices
   - Runs the main EKF loop
   - Calls visualization functions
2. Data Generation:
   - Creates a realistic trajectory (circular motion followed by straight line)
   - Generates noisy IMU data (acceleration and angular velocity)
   - Generates noisy GPS measurements at 1 Hz
3. EKF Implementation:
   - Prediction step using IMU measurements
   - Update step using GPS measurements when available
   - Properly computed Jacobian matrices
   - Noise covariance handling
4. Visualization:
   - Plots true trajectory, estimated trajectory, and GPS measurements
   - Shows position error over time

To run the code

5. Copy the entire code into a MATLAB script file
6. Run the script
7. Two plots will be generated showing the results

The simulation parameters can be adjusted:

- `dt`: Time step
- `T`: Total simulation time
- `sigma_a`, `sigma_w`: IMU noise parameters
- `sigma_gps`: GPS noise parameter

Exercise 2: Particle Filter Implementation

Implement a particle filter for global localization:

1. Initialize particles uniformly in the environment
2. Implement motion model using IMU data
3. Implement measurement model using GNSS and compass data
4. Implement resampling step
5. Visualize particle evolution and compare with ground truth

Solution

Code

A complete MATLAB implementation of a particle filter for global localization.

Main starting script

```matlab
% Main script for Particle Filter Implementation

% Parameters
dt = 0.1;  % Time step (s)
T = 100;   % Total simulation time (s)
t = 0:dt:T;
n = length(t);
N = 1000;  % Number of particles

% Environment boundaries
x_min = -50; x_max = 50;
y_min = -50; y_max = 50;

% Measurement noise parameters
sigma_gps = 2.0;    % GPS position noise (m)
sigma_compass = 0.1; % Compass heading noise (rad)

% Motion noise parameters
sigma_v = 0.5;      % Velocity noise
sigma_omega = 0.1;  % Angular velocity noise

% Initialize true state [x, y, theta]
true_state = zeros(3, n);
true_state(:,1) = [0; 0; 0];

% Initialize particles [x, y, theta, weight]
particles = zeros(4, N, n);
particles(1:2, :, 1) = [unifrnd(x_min, x_max, 1, N);
                        unifrnd(y_min, y_max, 1, N)];
particles(3, :, 1) = unifrnd(-pi, pi, 1, N);
particles(4, :, 1) = 1/N * ones(1, N);  % Initial weights
```

```
% Generate synthetic data
[imu_data, gps_data, compass_data, true_state] = generate_synthetic_data(true_state, dt, n);

% Initialize estimated state
est_state = zeros(3, n);
est_state(:,1) = mean(particles(1:3, :, 1), 2);

% Main particle filter loop
for k = 2:n
    % Predict step - Motion model
    particles(1:3, :, k) = predict_particles(particles(1:3, :, k-1), ...
        imu_data(:,k), dt, sigma_v, sigma_omega);

    % Update step - Measurement model
    if mod(k, 10) == 0  % GPS update at 1 Hz
        particles(4, :, k) = measurement_model(particles(1:3, :, k), ...
            gps_data(:,k), compass_data(k), sigma_gps, sigma_compass);

        % Resample particles
        particles(:, :, k) = resample_particles(particles(:, :, k));
    else
        particles(4, :, k) = particles(4, :, k-1);
    end

    % Calculate estimated state
    est_state(:,k) = mean(particles(1:3, :, k), 2);

    % Visualize every 10 steps
    if mod(k, 10) == 0
        visualize_particles(particles(:, :, k), true_state(:,k), est_state(:,k), ...
            gps_data(:,k), x_min, x_max, y_min, y_max);
        drawnow;
    end
end

% Final trajectory visualization
visualize_trajectory(t, true_state, est_state, gps_data);
```

Generate synthetic data

```
% Function to generate synthetic data
function [imu_data, gps_data, compass_data, true_state] = generate_synthetic_data(true_state, dt, n)
    % Initialize data arrays
    imu_data = zeros(2, n);    % [v, omega]
    gps_data = zeros(2, n);    % [x, y]
    compass_data = zeros(1, n); % theta

    % Generate true trajectory (figure-8 pattern)
    for k = 2:n
        t = (k-1)*dt;
        % Figure-8 trajectory parameters
        v = 2;  % Constant velocity
        omega = 0.5*sin(0.2*t);  % Time-varying angular velocity

        % Update true state
```

```
        true_state(1,k) = true_state(1,k-1) + v*cos(true_state(3,k-1))*dt;
        true_state(2,k) = true_state(2,k-1) + v*sin(true_state(3,k-1))*dt;
        true_state(3,k) = true_state(3,k-1) + omega*dt;

        % Generate noisy IMU measurements
        imu_data(1,k) = v + randn*0.2;      % Noisy velocity
        imu_data(2,k) = omega + randn*0.05; % Noisy angular velocity

        % Generate noisy GPS and compass measurements
        gps_data(1,k) = true_state(1,k) + randn*2.0;
        gps_data(2,k) = true_state(2,k) + randn*2.0;
        compass_data(k) = true_state(3,k) + randn*0.1;
    end
end
```

Predict particle motion function

```
% Function to predict particle motion
function pred_particles = predict_particles(particles, imu, dt, sigma_v,
    sigma_omega)
    N = size(particles, 2);
    pred_particles = zeros(size(particles));

    % Add noise to velocity and angular velocity
    v = imu(1) + randn(1, N)*sigma_v;
    omega = imu(2) + randn(1, N)*sigma_omega;

    % Update particles using motion model
    pred_particles(1,:) = particles(1,:) + v.*cos(particles(3,:))*dt;
    pred_particles(2,:) = particles(2,:) + v.*sin(particles(3,:))*dt;
    pred_particles(3,:) = particles(3,:) + omega*dt;

    % Normalize angles to [-pi, pi]
    pred_particles(3,:) = wrapToPi(pred_particles(3,:));
end
```

Measurement likelihood function

```
% Function to compute measurement likelihood
function weights = measurement_model(particles, gps, compass, sigma_gps,
    sigma_compass)
    % Compute position likelihood
    pos_likelihood = exp(-0.5*((particles(1,:) - gps(1)).^2 + ...
        (particles(2,:) - gps(2)).^2)/(sigma_gps^2));

    % Compute heading likelihood
    heading_diff = wrapToPi(particles(3,:) - compass);
    heading_likelihood = exp(-0.5*(heading_diff.^2)/(sigma_compass^2));

    % Combine likelihoods
    weights = pos_likelihood .* heading_likelihood;

    % Normalize weights
    weights = weights / sum(weights);
end
```

Resampling function

```
% Function to resample particles
function resampled_particles = resample_particles(particles)
```

```
    N = size(particles, 2);
    weights = particles(4,:);

    % Systematic resampling
    positions = (rand + (0:N-1))/N;
    cumsum_weights = cumsum(weights);

    % Initialize resampled particles
    resampled_particles = zeros(size(particles));
    i = 1;
    j = 1;

    while i <= N
        if positions(i) < cumsum_weights(j)
            resampled_particles(:,i) = particles(:,j);
            i = i + 1;
        else
            j = j + 1;
        end
    end

    % Reset weights
    resampled_particles(4,:) = 1/N * ones(1,N);
end
```

Visualization functions

```
% Function to visualize particles
function visualize_particles(particles, true_state, est_state, gps, x_min,
    x_max, y_min, y_max)
    clf;
    % Plot particles
    scatter(particles(1,:), particles(2,:), 5, 'b.', 'MarkerAlpha', 0.3);
    hold on;

    % Plot true position
    plot(true_state(1), true_state(2), 'g*', 'MarkerSize', 10, 'LineWidth',
     2);

    % Plot estimated position
    plot(est_state(1), est_state(2), 'r+', 'MarkerSize', 10, 'LineWidth',
    2);

    % Plot GPS measurement
    plot(gps(1), gps(2), 'k.', 'MarkerSize', 15);

    % Plot particle directions (for subset of particles)
    subset = 1:50:length(particles);
    quiver(particles(1,subset), particles(2,subset), ...
        cos(particles(3,subset)), sin(particles(3,subset)), 0.5, 'b');

    grid on;
    xlim([x_min x_max]);
    ylim([y_min y_max]);
    legend('Particles', 'True Position', 'Estimated Position', 'GPS
    Measurement');
    title('Particle Filter Localization');
    xlabel('X (m)');
    ylabel('Y (m)');
```

```
end

% Function to visualize complete trajectory
function visualize_trajectory(t, true_state, est_state, gps_data)
    figure;

    % Plot trajectories
    subplot(2,1,1);
    plot(true_state(1,:), true_state(2,:), 'g-', 'LineWidth', 2);
    hold on;
    plot(est_state(1,:), est_state(2,:), 'r--', 'LineWidth', 2);
    plot(gps_data(1,1:10:end), gps_data(2,1:10:end), 'k.', 'MarkerSize',
   10);
    grid on;
    legend('True Trajectory', 'Estimated Trajectory', 'GPS Measurements');
    title('Robot Trajectory');
    xlabel('X (m)');
    ylabel('Y (m)');

    % Plot position error
    subplot(2,1,2);
    pos_error = sqrt((true_state(1,:)-est_state(1,:)).^2 + ...
        (true_state(2,:)-est_state(2,:)).^2);
    plot(t, pos_error, 'b-', 'LineWidth', 2);
    grid on;
    xlabel('Time (s)');
    ylabel('Position Error (m)');
    title('Position Error Over Time');
end
```

Code components breakdown

1. Initialization:
    - Uniformly distributes particles across the environment
    - Sets up simulation parameters and noise models
    - Initializes true state and measurement data
2. Motion Model:
    - Uses velocity and angular velocity from IMU
    - Includes noise in the prediction step
    - Updates particle positions and orientations
3. Measurement Model:
    - Combines GPS and compass measurements
    - Computes particle weights based on measurement likelihood
    - Handles both position and heading measurements
4. Resampling:
    - Implements systematic resampling
    - Maintains particle diversity
    - Resets weights after resampling
5. Visualization:

- Real-time visualization of particles and robot state
- Shows particle distribution and headings
- Plots complete trajectory and position error

To run the code:

Copy the entire code into a MATLAB script Run the script You'll see real-time visualization of the particle filter and final trajectory plots

The key parameters you can adjust:

- `N`: Number of particles
- `sigma_gps`, `sigma_compass`: Measurement noise parameters
- `sigma_v`, `sigma_omega`: Motion model noise parameters
- Environment boundaries (`x_min`, `x_max`, `y_min`, `y_max`)

Exercise 3: Comparison Study

Compare the performance of EKF and Particle Filter:

1. Generate test scenarios with different noise levels
2. Implement error metrics (RMSE, consistency)
3. Compare computational requirements
4. Analyze failure cases for each method

Solution

Code

A comprehensive MATLAB comparison between EKF and Particle Filter for robot localization.

Main starting script

```
% Comparison of EKF and Particle Filter for Robot Localization
clear all; close all; clc;

% Simulation parameters
dt = 0.1;  % Time step (s)
T = 100;   % Total simulation time (s)
t = 0:dt:T;
n = length(t);
N = 1000;  % Number of particles for PF

% Test scenarios with different noise levels
noise_levels = struct('low', struct('gps', 1.0, 'imu_v', 0.1, 'imu_w',
    0.05), ...
                     'medium', struct('gps', 2.0, 'imu_v', 0.3, 'imu_w',
    0.1), ...
                     'high', struct('gps', 4.0, 'imu_v', 0.5, 'imu_w', 0.2)
    );

% Initialize results structure
results = struct();
scenarios = fieldnames(noise_levels);

```

```
% Run simulations for each noise level
for s = 1:length(scenarios)
    scenario = scenarios{s};
    noise = noise_levels.(scenario);

    % Generate true trajectory and measurements
    [true_state, imu_data, gps_data] = generate_data(t, dt, n, noise);

    % Run EKF
    tic;
    ekf_state = run_ekf(imu_data, gps_data, dt, n, noise);
    results.(scenario).ekf_time = toc;

    % Run Particle Filter
    tic;
    pf_state = run_particle_filter(imu_data, gps_data, dt, n, N, noise);
    results.(scenario).pf_time = toc;

    % Calculate metrics
    results.(scenario).metrics = calculate_metrics(true_state, ekf_state, pf_state, n);

    % Store states for visualization
    results.(scenario).true_state = true_state;
    results.(scenario).ekf_state = ekf_state;
    results.(scenario).pf_state = pf_state;
    results.(scenario).gps_data = gps_data;
end

% Visualize results
visualize_results(results, scenarios, t);
```

Data generation function

```

%% Helper Functions

function [true_state, imu_data, gps_data] = generate_data(t, dt, n, noise)
    % Initialize states [x, y, theta, v, w]
    true_state = zeros(5, n);
    imu_data = zeros(2, n);    % [v, w]
    gps_data = zeros(2, n);    % [x, y]

    % Generate figure-8 trajectory
    for k = 2:n
        % True motion
        time = t(k);
        v = 2 + 0.5*sin(0.1*time);
        w = 0.5*sin(0.2*time);

        % Update true state
        true_state(4:5,k) = [v; w];
        true_state(1:3,k) = true_state(1:3,k-1) + ...
            [v*cos(true_state(3,k-1))*dt;
             v*sin(true_state(3,k-1))*dt;
             w*dt];

        % Generate noisy IMU data
        imu_data(:,k) = [v; w] + ...
```

```
            [randn*noise.imu_v; randn*noise.imu_w];

        % Generate noisy GPS data (at 1 Hz)
        if mod(k, 10) == 0
            gps_data(:,k) = true_state(1:2,k) + ...
                randn(2,1)*noise.gps;
        end
    end
end
```

Running filters functions

```

function ekf_state = run_ekf(imu_data, gps_data, dt, n, noise)
    % Initialize EKF
    ekf_state = zeros(5, n);
    P = diag([1, 1, 0.1, 0.1, 0.1]);
    Q = diag([noise.imu_v^2, noise.imu_v^2, noise.imu_w^2, ...
        noise.imu_v^2, noise.imu_w^2]);
    R = eye(2)*noise.gps^2;

    for k = 2:n
        % Prediction
        ekf_state(:,k) = predict_ekf(ekf_state(:,k-1), imu_data(:,k), dt);
        F = compute_jacobian(ekf_state(:,k-1), dt);
        P = F*P*F' + Q;

        % Update if GPS available
        if any(gps_data(:,k))
            H = [1 0 0 0 0; 0 1 0 0 0];
            y = gps_data(:,k) - ekf_state(1:2,k);
            S = H*P*H' + R;
            K = P*H'/S;
            ekf_state(:,k) = ekf_state(:,k) + K*y;
            P = (eye(5) - K*H)*P;
        end
    end
end

function pf_state = run_particle_filter(imu_data, gps_data, dt, n, N, noise
    )
    % Initialize particles [x, y, theta, v, w, weight]
    particles = zeros(6, N, n);
    particles(1:2,:,1) = randn(2,N)*10;  % Initial position
    particles(3,:,1) = randn(1,N)*0.1;   % Initial heading
    particles(6,:,1) = 1/N;              % Initial weights

    pf_state = zeros(5, n);
    pf_state(:,1) = mean(particles(1:5,:,1), 2);

    for k = 2:n
        % Predict
        particles(1:5,:,k) = predict_pf(particles(1:5,:,k-1), ...
            imu_data(:,k), dt, noise);
        particles(6,:,k) = particles(6,:,k-1);

        % Update if GPS available
        if any(gps_data(:,k))
            particles(6,:,k) = compute_weights(particles(1:2,:,k), ...
```

```
                gps_data(:,k), noise.gps);

            % Resample if effective sample size is too low
            if 1/sum(particles(6,:,k).^2) < N/2
                particles(:,:,k) = resample_particles(particles(:,:,k));
            end
        end

        % Compute state estimate
        pf_state(:,k) = sum(particles(1:5,:,k).*particles(6,:,k), 2);
    end
end
```

Metrics calculation function

```

function metrics = calculate_metrics(true_state, ekf_state, pf_state, n)
    % Calculate RMSE
    ekf_rmse = sqrt(mean((true_state(1:2,:) - ekf_state(1:2,:)).^2, 2));
    pf_rmse = sqrt(mean((true_state(1:2,:) - pf_state(1:2,:)).^2, 2));

    % Calculate consistency (NEES - Normalized Estimation Error Squared)
    ekf_err = true_state - ekf_state;
    pf_err = true_state - pf_state;

    ekf_nees = mean(sum(ekf_err.^2, 1));
    pf_nees = mean(sum(pf_err.^2, 1));

    metrics = struct('ekf_rmse', ekf_rmse, 'pf_rmse', pf_rmse, ...
        'ekf_nees', ekf_nees, 'pf_nees', pf_nees);
end
```

Visualization function

```
function visualize_results(results, scenarios, t)
    % Create figure for trajectories
    figure('Position', [100 100 1200 800]);

    for i = 1:length(scenarios)
        scenario = scenarios{i};
        data = results.(scenario);

        % Plot trajectories
        subplot(2,2,i);
        plot(data.true_state(1,:), data.true_state(2,:), 'g-', 'LineWidth',
 2);
        hold on;
        plot(data.ekf_state(1,:), data.ekf_state(2,:), 'b--', 'LineWidth',
 1.5);
        plot(data.pf_state(1,:), data.pf_state(2,:), 'r:', 'LineWidth',
 1.5);
        plot(data.gps_data(1,:), data.gps_data(2,:), 'k.', 'MarkerSize',
 10);
        grid on;
        title(['Trajectory - ' scenario ' noise']);
        legend('True', 'EKF', 'PF', 'GPS');
        xlabel('X (m)'); ylabel('Y (m)');
    end

    % Plot performance metrics
```

```
    figure('Position', [100 100 1200 400]);

    % RMSE comparison
    subplot(1,2,1);
    rmse_ekf = zeros(1,length(scenarios));
    rmse_pf = zeros(1,length(scenarios));
    for i = 1:length(scenarios)
        rmse_ekf(i) = mean(results.(scenarios{i}).metrics.ekf_rmse);
        rmse_pf(i) = mean(results.(scenarios{i}).metrics.pf_rmse);
    end
    bar([rmse_ekf; rmse_pf]');
    set(gca, 'XTickLabel', scenarios);
    legend('EKF', 'PF');
    title('Average RMSE');
    ylabel('meters');

    % Computation time comparison
    subplot(1,2,2);
    time_ekf = zeros(1,length(scenarios));
    time_pf = zeros(1,length(scenarios));
    for i = 1:length(scenarios)
        time_ekf(i) = results.(scenarios{i}).ekf_time;
        time_pf(i) = results.(scenarios{i}).pf_time;
    end
    bar([time_ekf; time_pf]');
    set(gca, 'XTickLabel', scenarios);
    legend('EKF', 'PF');
    title('Computation Time');
    ylabel('seconds');
end
```

EKF functions

```

function x_pred = predict_ekf(x, imu, dt)
    x_pred = zeros(size(x));
    x_pred(1) = x(1) + x(4)*cos(x(3))*dt;
    x_pred(2) = x(2) + x(4)*sin(x(3))*dt;
    x_pred(3) = x(3) + x(5)*dt;
    x_pred(4) = imu(1);
    x_pred(5) = imu(2);
end

function F = compute_jacobian(x, dt)
    F = eye(5);
    F(1,3) = -x(4)*sin(x(3))*dt;
    F(1,4) = cos(x(3))*dt;
    F(2,3) = x(4)*cos(x(3))*dt;
    F(2,4) = sin(x(3))*dt;
    F(3,5) = dt;
end
```

Particle Filter functions

```

function particles_pred = predict_pf(particles, imu, dt, noise)
    N = size(particles, 2);
    particles_pred = zeros(size(particles));

    % Add noise to velocity and angular velocity
```

```
    v = imu(1) + randn(1,N)*noise.imu_v;
    w = imu(2) + randn(1,N)*noise.imu_w;

    particles_pred(1,:) = particles(1,:) + v.*cos(particles(3,:))*dt;
    particles_pred(2,:) = particles(2,:) + v.*sin(particles(3,:))*dt;
    particles_pred(3,:) = particles(3,:) + w*dt;
    particles_pred(4,:) = v;
    particles_pred(5,:) = w;
end

function weights = compute_weights(particle_pos, gps, noise)
    % Compute likelihood based on GPS measurement
    innovation = particle_pos - gps;
    weights = exp(-0.5*sum(innovation.^2, 1)/(noise^2));
    weights = weights / sum(weights);  % Normalize
end

function particles_new = resample_particles(particles)
    N = size(particles, 2);
    weights = particles(6,:);

    % Systematic resampling
    positions = (rand + (0:N-1))/N;
    cumsum_weights = cumsum(weights);

    particles_new = zeros(size(particles));
    i = 1;
    j = 1;

    while i <= N
        if positions(i) < cumsum_weights(j)
            particles_new(:,i) = particles(:,j);
            i = i + 1;
        else
            j = j + 1;
        end
    end

    % Reset weights
    particles_new(6,:) = 1/N;
end
```

Code components breakdown

1. Test Scenarios:
    - Three noise levels (low, medium, high)
    - Different GPS and IMU noise characteristics
    - Figure-8 trajectory for testing non-linear motion
2. Performance Metrics:

- RMSE for position accuracy
- NEES for filter consistency
- Computation time measurement

- Trajectory visualization

3. Implementation Details:

- EKF with proper Jacobian computation
- Particle filter with adaptive resampling
- Consistent noise models across both filters

4. Key Findings: EKF Characteristics:
   - Pros:
     - Computationally efficient
     - Good performance with low noise
     - Consistent state estimation
   - Cons:
     - Performance degrades with high noise
     - Can diverge with poor initialization
     - Assumes Gaussian noise

Particle Filter Characteristics: * Pros: * More robust to high noise * Better handles non-Gaussian noise * Can recover from poor initialization * Cons: * Computationally more intensive * Performance depends on particle count * Can suffer from particle depletion

To run the comparison:

1. Copy the code into MATLAB
2. Run the script
3. Two figures will be generated:
   - Trajectories for each noise scenario
   - Performance metrics comparison

Final Quiz

1. Which statement best describes the relationship between GNSS accuracy and autonomous driving requirements?
   GNSS accuracy of 5-10 meters is sufficient for autonomous driving
   Autonomous driving requires centimeter-level accuracy
   Meter-level accuracy is adequate for all autonomous operations
   GNSS accuracy is not important for autonomous driving
2. Why is the Extended Kalman Filter needed instead of a standard Kalman Filter for vehicle localization?
   It's computationally faster
   It can handle non-linear vehicle dynamics
   It requires less memory
   It's easier to implement
3. What is the main advantage of particle filters over EKF?
   They are always more accurate
   They require less computation
   They can represent multi-modal distributions
   They work better with linear systems
4. Which sensor fusion combination is most commonly used for basic vehicle localization?
   Camera + Lidar
   GNSS + IMU
   Radar + Sonar
   Compass + Speedometer
5. What is the purpose of the resampling step in particle filters?
   To reduce computational complexity
   To prevent particle degeneracy
   To improve accuracy
   To linearize the system
6. Which factor most significantly affects GNSS accuracy in urban environments?
   Temperature variations
   Vehicle speed
   Multipath effects
   Satellite clock errors
7. What is the primary reason for maintaining a covariance matrix in the EKF?
   To track system uncertainty
   To improve computation speed
   To store sensor measurements
   To handle non-linear dynamics

8. Why is sensor fusion necessary for robust localization?
   To reduce system cost
   To compensate for individual sensor limitations
   To simplify calculations
   To meet regulatory requirements
9. What is the main challenge in implementing particle filters?
   They are difficult to program
   Choosing the appropriate number of particles
   They don't work with GNSS data
   They require special hardware
10. Which statement about IMU integration is correct?
    IMU data alone provides drift-free position estimates
    IMU bias must be estimated and compensated
    IMU measurements are always accurate
    IMU drift is not a significant problem

Answer Key:

1. b
2. b
3. c
4. b
5. b
6. c
7. a
8. b
9. b
10. b