GPS-IMU Sensor Fusion Analysis

Introduction

This analysis examines the performance of a GPS-IMU sensor fusion system in the CARLA simulator, focusing on its ability to detect and handle various GPS spoofing attacks. The system uses a Kalman filter to combine data from GPS and IMU sensors, providing accurate vehicle positioning while maintaining robustness against spoofing attempts.

Understanding Kalman Filters

High-Level Overview

Imagine you're trying to track a car's position using two different sources of information:

1. A GPS that tells you where the car is, but sometimes gives slightly wrong readings
2. A motion sensor (IMU) that tells you how the car is moving, but can drift over time

The Kalman filter is like a smart assistant that:

- Takes both pieces of information
- Weighs them based on how reliable they are
- Combines them to give you the best possible estimate of where the car actually is
- Learns from past mistakes to improve future estimates

In our research, we're testing how well this system can detect when someone tries to trick the GPS into giving wrong information (GPS spoofing). This is important because:

- GPS is widely used in autonomous vehicles
- Spoofing attacks could be used to make vehicles go to wrong locations
- We need to ensure vehicles can detect and handle such attacks

Technical Details

The Kalman filter implementation uses a 6-state system:

- Position (x, y, z)
- Velocity (vx, vy, vz)

Key components:

1. **State Transition Matrix (A)**

```
A = [
    [1, 0, 0, dt, 0, 0],
    [0, 1, 0, 0, dt, 0],
    [0, 0, 1, 0, 0, dt],
    [0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0],
```

```
    [0, 0, 0, 0, 0, 1]
]
```

- Implements constant velocity model
- dt represents time step between updates
- Accounts for position-velocity coupling

2. **Measurement Matrix (H)**

```
H = [
    [1, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0]
]
```

- Maps state vector to GPS measurements
- Only measures position components
- Assumes direct position measurements

3. **Process Noise Covariance (Q)**

- Set to 0.1 * identity matrix
- Models system uncertainty
- Accounts for:
  - Model imperfections
  - Unmodeled dynamics
  - IMU drift

4. **Measurement Noise Covariance (R)**

- Set to 0.1 * identity matrix
- Models sensor noise
- Accounts for:
  - GPS measurement errors
  - Environmental factors
  - Signal interference

Spoofing Methods Analysis

1. Gradual Drift

- **Implementation**: Sinusoidal drift pattern
- **Parameters**:
  - Drift rate: 0.1 m/s
  - Pattern: Sinusoidal in x and y coordinates
- **Test Results**:
  - Average Error: ~0.1 meters (10 cm)
  - Error Stability: Very stable with minimal fluctuation

    - Error Range: 0.100009 to 0.100010 meters
- **Analysis**:
    - Kalman filter effectively maintains tracking
    - Error remains constant due to drift rate matching filter's adaptation
    - Most challenging to detect due to smooth, natural-looking pattern
    - Demonstrates filter's ability to handle continuous, bounded changes

2. Sudden Jump

- **Implementation**: Random position jumps
- **Parameters**:
    - Jump magnitude: 5.0 meters
    - Probability: 1% per update
- **Test Results**:
    - Error Range: 1.95m to 14.44m
    - Average Error: ~7m
    - Pattern: Shows sudden error increases followed by gradual recovery
- **Analysis**:
    - Filter shows limited ability to handle sudden changes
    - Error accumulates before correction
    - Recovery time proportional to jump magnitude
    - Easiest to detect due to clear discontinuity
    - Demonstrates need for additional detection mechanisms

3. Random Walk

- **Implementation**: Random movements in x and y coordinates
- **Parameters**:
    - Step size: 0.5 meters
    - Independent in x and y coordinates
- **Test Results**:
    - Error Range: 0.05m to 0.53m
    - Average Error: ~0.25m
    - Pattern: Shows consistent error distribution with occasional spikes
- **Analysis**:
    - Filter handles random walk better than sudden jumps
    - Error remains within reasonable bounds
    - Quick recovery from small perturbations
    - More predictable than sudden jumps due to bounded step size
    - Demonstrates filter's effectiveness with bounded random changes

4. Replay Attack

- **Implementation**: Records and replays previous positions
- **Parameters**:
    - Buffer size: 100 positions
    - Replay delay: 2.0 seconds
- **Test Results**:

    - Error Range: 0.0000008m to 4.13m
    - Pattern: Three distinct phases:
        1. Initial stability (0.0000008m - 0.04m)
        2. Rapid error growth (0.04m - 4.13m)
        3. Gradual error reduction (4.13m - 0.0000008m)
- **Analysis**:
    - Clear temporal pattern in error growth
    - Excellent recovery capability
    - Correlation between error and replay delay
    - Demonstrates need for temporal consistency checks

Performance Analysis

Overall System Performance

1. **Gradual Drift**
    - Best performance
    - Consistent error of ~10cm
    - Filter effectively compensates
    - Hardest to detect
2. **Random Walk**
    - Good performance
    - Error range: 0.05m - 0.53m
    - Quick recovery
    - Moderate detection difficulty
3. **Sudden Jump**
    - Poorest performance
    - Error range: 1.95m - 14.44m
    - Slow recovery
    - Easiest to detect
4. **Replay Attack**
    - Variable performance
    - Error range: 0.0000008m - 4.13m
    - Excellent recovery
    - Clear temporal pattern

Key Findings

1. **Filter Effectiveness**
    - Excellent with continuous, bounded changes
    - Poor with sudden, large changes
    - Good recovery capability in most cases

- Limited by current noise models

2. **Detection Capabilities**
    - Natural drift hardest to detect
    - Sudden jumps easiest to detect
    - Random walk and replay require additional checks
    - Temporal patterns provide detection opportunities
3. **System Limitations**
    - Limited ability to handle sudden changes
    - No built-in temporal validation
    - Fixed noise models
    - No velocity consistency checks

Detailed Analysis of GPS Spoofing Attacks

Attack Mechanisms and Real-World Implications

1. Gradual Drift Attack

Mechanism:

- Implements a sinusoidal drift pattern in GPS coordinates
- Drift rate: 0.1 m/s, simulating natural GPS drift
- Smooth, continuous position changes
- Maintains velocity consistency

Real-World Equivalent:

- Similar to GPS signal degradation in urban canyons
- Mimics atmospheric interference effects
- Resembles GPS signal multipath issues
- Common in areas with tall buildings or dense urban environments

Detection Challenges:

- Difficult to distinguish from natural GPS drift
- No sudden changes to trigger anomaly detection
- Maintains realistic velocity profiles
- Smooth transition makes it hard to detect statistically

Implementation Details:

```
def _gradual_drift(self, true_position):
    drift = np.array([
        self.drift_rate * np.sin(self.time * 0.1),
        self.drift_rate * np.cos(self.time * 0.1),
        0
```

```
        ])
        return true_position + drift
```

2. Sudden Jump Attack

Mechanism:

- Random position jumps of up to 5 meters
- 1% probability per update
- Instantaneous position changes
- No velocity consistency

Real-World Equivalent:

- Similar to GPS signal hijacking
- Resembles GPS signal spoofing attacks
- Common in military GPS spoofing scenarios
- Used in GPS signal jamming attacks

Detection Challenges:

- Easy to detect due to discontinuity
- Large error spikes are obvious
- Recovery time is proportional to jump magnitude
- Can be detected through velocity inconsistency

Implementation Details:

```
def _sudden_jump(self, true_position):
    if np.random.random() < 0.01:  # 1% chance
        jump = np.random.uniform(-5, 5, size=3)
        return true_position + jump
    return true_position
```

3. Random Walk Attack

Mechanism:

- Random movements in x and y coordinates
- Step size: 0.5 meters per update
- Independent movement in each axis
- Bounded random changes

Real-World Equivalent:

- Similar to GPS signal interference
- Resembles GPS signal noise
- Common in areas with electronic interference

- Used in GPS signal disruption attacks

Detection Challenges:

- More difficult to detect than sudden jumps
- Maintains bounded error range
- Quick recovery from small perturbations
- Requires statistical analysis for detection

Implementation Details:

```
def _random_walk(self, true_position):
    step = np.random.uniform(-0.5, 0.5, size=3)
    return true_position + step
```

4. Replay Attack

Mechanism:

- Records and replays previous positions
- Buffer size: 100 positions
- Replay delay: 2.0 seconds
- Temporal inconsistency

Real-World Equivalent:

- Similar to GPS signal replay attacks
- Resembles GPS signal recording and playback
- Common in GPS signal spoofing research
- Used in GPS signal manipulation attacks

Detection Challenges:

- Clear temporal pattern
- Three distinct phases:
    1. Initial stability
    2. Rapid error growth
    3. Gradual error reduction
- Requires temporal consistency checks
- Can be detected through position history analysis

Implementation Details:

```
def _replay(self, true_position):
    self.position_buffer.append(true_position)
    if len(self.position_buffer) > self.buffer_size:
        return self.position_buffer.pop(0)
    return true_position
```

Attack Characteristics Comparison

Attack Type	Error Range	Detection Difficulty	Recovery Time	Real-World Likelihood
Gradual Drift	0.10m	High	N/A	High
Sudden Jump	1.95m-14.44m	Low	Long	Medium
Random Walk	0.05m-0.53m	Medium	Short	High
Replay	0.0000008m-4.13m	Medium	Medium	Low

Attack Detection Strategies

1. **Gradual Drift Detection**
    - Long-term statistical analysis
    - Velocity consistency checks
    - Multiple sensor fusion
    - Environmental context awareness
2. **Sudden Jump Detection**
    - Immediate error threshold checks
    - Velocity discontinuity detection
    - Position history analysis
    - Multiple sensor validation
3. **Random Walk Detection**
    - Statistical anomaly detection
    - Bounded error analysis
    - Pattern recognition
    - Multiple sensor correlation
4. **Replay Detection**
    - Timestamp validation
    - Position history analysis
    - Temporal consistency checks
    - Multiple sensor synchronization

Security Implications

1. **Autonomous Vehicle Security**
    - GPS spoofing can lead to incorrect navigation
    - Potential for vehicle hijacking
    - Safety concerns in urban environments
    - Need for robust detection systems

2. **Critical Infrastructure**
    - GPS timing attacks on power grids
    - Navigation system vulnerabilities
    - Communication system dependencies
    - Need for backup systems
3. **Military Applications**
    - GPS spoofing in conflict zones
    - Navigation system security
    - Weapon system vulnerabilities
    - Need for secure alternatives
4. **Commercial Applications**
    - Fleet management security
    - Navigation system reliability
    - Location-based services
    - Need for spoofing detection

Conclusion

The sensor fusion system demonstrates varying performance against different spoofing attacks, with the Kalman filter providing a solid foundation for sensor fusion. The system excels at handling continuous, bounded changes but struggles with sudden, large changes. The results suggest that additional detection mechanisms, particularly temporal consistency checks and velocity validation, would significantly improve the system's robustness against spoofing attacks.

The findings have important implications for autonomous vehicle security, highlighting the need for:

1. Multiple detection mechanisms
2. Temporal validation
3. Adaptive filtering
4. Sophisticated motion models

These improvements would enhance the system's ability to detect and handle various types of GPS spoofing attacks in real-world scenarios.

In this research project, I've developed and tested a GPS-IMU sensor fusion system using the CARLA simulator to detect and analyze various GPS spoofing attacks. The system employs a Kalman filter to combine data from GPS and IMU sensors, providing accurate vehicle positioning while maintaining robustness against spoofing attempts. I've implemented and tested four distinct spoofing methods: gradual drift, sudden jump, random walk, and replay attacks, each simulating different real-world GPS spoofing scenarios. The results demonstrate varying levels of effectiveness in detecting different types of attacks. The system performs exceptionally well against gradual drift (maintaining ~10cm error), moderately against random walk (0.05m-0.53m error range), and replay attacks (0.0000008m-4.13m error range), but struggles with sudden jumps (1.95m-14.44m error range). These findings highlight the importance of implementing multiple detection mechanisms and temporal validation to enhance the system's robustness against sophisticated spoofing attacks.

Phase 3: Detection and Correction Implementation

1. Detection System Architecture

```
class SpoofingDetector:
    def __init__(self):
        # Detection thresholds
        self.gradual_drift_threshold = 0.15  # meters
        self.sudden_jump_threshold = 1.0     # meters
        self.random_walk_threshold = 0.4     # meters
        self.replay_threshold = 0.1          # meters

        # Temporal analysis
        self.position_history = []
        self.velocity_history = []
        self.timestamp_history = []

        # Statistical analysis
        self.error_mean = 0
        self.error_variance = 0
        self.window_size = 100

    def detect_spoofing(self, current_position, current_velocity, timestamp):
        # Update history
        self._update_history(current_position, current_velocity, timestamp)

        # Run detection algorithms
        detection_results = {
            'gradual_drift': self._detect_gradual_drift(),
            'sudden_jump': self._detect_sudden_jump(),
            'random_walk': self._detect_random_walk(),
            'replay': self._detect_replay()
        }

        return detection_results
```

2. Detection Methods

2.1 Gradual Drift Detection

```
def _detect_gradual_drift(self):
    """
    Detects gradual drift by analyzing:
    1. Long-term position trends
    2. Velocity consistency
    3. Error accumulation pattern
    """
    # Calculate position trend
    position_trend = self._calculate_position_trend()
```

```
    # Check velocity consistency
    velocity_consistency = self._check_velocity_consistency()

    # Analyze error accumulation
    error_pattern = self._analyze_error_pattern()

    return {
        'detected': position_trend > self.gradual_drift_threshold,
        'confidence': self._calculate_confidence(position_trend,
velocity_consistency, error_pattern)
    }
```

2.2 Sudden Jump Detection

```
def _detect_sudden_jump(self):
    """
    Detects sudden jumps by analyzing:
    1. Position discontinuities
    2. Velocity inconsistencies
    3. Error spikes
    """
    # Calculate position difference
    position_diff = self._calculate_position_difference()

    # Check for velocity anomalies
    velocity_anomaly = self._check_velocity_anomaly()

    # Detect error spikes
    error_spike = self._detect_error_spike()

    return {
        'detected': position_diff > self.sudden_jump_threshold,
        'confidence': self._calculate_confidence(position_diff, velocity_anomaly,
error_spike)
    }
```

2.3 Random Walk Detection

```
def _detect_random_walk(self):
    """
    Detects random walk by analyzing:
    1. Statistical distribution of movements
    2. Step size patterns
    3. Direction changes
    """
    # Calculate movement statistics
    movement_stats = self._calculate_movement_statistics()
```

```
    # Analyze step size pattern
    step_pattern = self._analyze_step_pattern()

    # Check direction changes
    direction_changes = self._analyze_direction_changes()

    return {
        'detected': movement_stats['variance'] > self.random_walk_threshold,
        'confidence': self._calculate_confidence(movement_stats, step_pattern,
direction_changes)
    }
```

2.4 Replay Detection

```
def _detect_replay(self):
    """
    Detects replay attacks by analyzing:
    1. Temporal patterns
    2. Position history correlation
    3. Velocity consistency
    """
    # Check temporal patterns
    temporal_pattern = self._check_temporal_pattern()

    # Analyze position history
    position_correlation = self._analyze_position_correlation()

    # Verify velocity consistency
    velocity_consistency = self._verify_velocity_consistency()

    return {
        'detected': temporal_pattern > self.replay_threshold,
        'confidence': self._calculate_confidence(temporal_pattern,
position_correlation, velocity_consistency)
    }
```

3. Correction Strategies

3.1 Gradual Drift Correction

```
def correct_gradual_drift(self, detected_drift):
    """
    Corrects gradual drift by:
    1. Estimating drift rate
    2. Applying counter-drift
    3. Adjusting Kalman filter parameters
    """
    # Estimate drift rate
```

```
    drift_rate = self._estimate_drift_rate()

    # Apply counter-drift
    corrected_position = self._apply_counter_drift(drift_rate)

    # Adjust Kalman filter
    self._adjust_kalman_filter(drift_rate)

    return corrected_position
```

3.2 Sudden Jump Correction

```
def correct_sudden_jump(self, detected_jump):
    """
    Corrects sudden jumps by:
    1. Rejecting anomalous measurements
    2. Using IMU prediction
    3. Gradually re-integrating GPS
    """
    # Reject anomalous measurement
    self._reject_anomalous_measurement()

    # Use IMU prediction
    predicted_position = self._use_imu_prediction()

    # Re-integrate GPS
    self._reintegrate_gps()

    return predicted_position
```

3.3 Random Walk Correction

```
def correct_random_walk(self, detected_walk):
    """
    Corrects random walk by:
    1. Filtering high-frequency changes
    2. Applying statistical smoothing
    3. Weighting measurements
    """
    # Apply low-pass filter
    filtered_position = self._apply_low_pass_filter()

    # Apply statistical smoothing
    smoothed_position = self._apply_statistical_smoothing()

    # Adjust measurement weights
    self._adjust_measurement_weights()
```

```
    return smoothed_position
```

3.4 Replay Correction

```
def correct_replay(self, detected_replay):
    """
    Corrects replay attacks by:
    1. Detecting replay phase
    2. Using alternative sensors
    3. Implementing temporal validation
    """
    # Detect replay phase
    phase = self._detect_replay_phase()

    # Use alternative sensors
    alternative_position = self._use_alternative_sensors()

    # Apply temporal validation
    validated_position = self._apply_temporal_validation()

    return validated_position
```

4. Implementation Steps

1. **Setup Detection System**
    - Implement `SpoofingDetector` class
    - Configure detection thresholds
    - Initialize history buffers
2. **Implement Detection Methods**
    - Add detection algorithms for each attack type
    - Implement confidence calculation
    - Add temporal analysis
3. **Implement Correction Strategies**
    - Add correction methods for each attack type
    - Implement sensor fusion adjustments
    - Add validation mechanisms
4. **Integration with Existing System**
    - Modify `SensorFusion` class to use detector
    - Add correction feedback loop
    - Implement logging and monitoring

5. Testing Strategy

1. **Unit Testing**
    - Test each detection method
    - Verify correction algorithms
    - Validate confidence calculations
2. **Integration Testing**
    - Test with real sensor data
    - Verify system response
    - Measure correction effectiveness
3. **Performance Testing**
    - Measure detection latency
    - Verify correction accuracy
    - Test system robustness