ELEC 473 Project Report

Autonomy for Mobile Robots and Kalman Filtering – Individual Submission

A.Patel | 40227663 | BEng. Computer Engineering | Concordia University | achalypatel3403@gmail.com | 2nd Dec, 2025

***Abstract*— This report presents the implementation of a Kalman filter for sensor fusion between IMU and Global Positioning System (GPS) data for autonomous vehicle trajectory estimation. The Kalman filter optimally fuses high-frequency IMU measurements with low-frequency GPS measurements, achieving bounded position error while maintaining smooth trajectory estimates. Results demonstrate the effectiveness of sensor fusion for autonomous vehicle navigation, combining the complementary strengths of INS (high-frequency updates, short-term accuracy) and GPS (absolute positioning, long-term accuracy) to overcome the fundamental limitations of INS-only navigation.**

I. Introduction

This report will move past section 1 and 2 pretty fast or not conrtain them at all but will give detailed report for section 3 which was not included int eh team report that was submitted to not bore you and submit repoeated work

A. ***Objectives and Scope***

The primary objectives are:

1. Characterize accelerometer and gyroscope bias and noise properties from stationary data
2. Reconstruct vehicle trajectories using INS equations with proper bias correction
3. Implement a Kalman filter to fuse IMU and GPS data for optimal trajectory estimation.

B. ***Assumptions***

The following assumptions were made throughout this project:

- **Linear bias model:** Sensor bias is modeled as a linear function of time, $b(t) = b_0 + b_s t$ which is valid for moderate time periods typical of mobile robot operations.
- **Gaussian noise:** Measurement noise is assumed to be zero-mean Gaussian, which is standard for Kalman filter implementation and validated through statistical analysis in Section I.
- **Stationary initial conditions:** Vehicles are assumed to start from rest, with zero initial velocity for trajectory reconstruction.
- **Variable sampling rates:** IMU data exhibits variable sampling rates, and numerical integration accounts for this by computing variable time steps $\Delta t = t[k] - t[k-1]$ for each integration step, rather than assuming constant sampling intervals.
- **Local Gravity:** Standard gravity value $g = 9.805$ m/s² is used for accelerometer z-axis correction.

II. Preliminaries

A. Sensor Models

Accelerometer and gyroscope measurements are modeled as:

$$a_m(t) = a(t) + b_a(t) + \nu_a(t)$$
$$\omega_m(t) = \omega(t) + b_\omega(t) + \nu_\omega(t)$$

where $a_m(t)$ and $\omega_m(t)$ are measured values, $a(t)$ and $\omega(t)$ are true values, $b_a(t)$ and $b_\omega(t)$ are time-varying biases, and $\nu_a(t)$ and $\nu_\omega(t)$ are zero-mean white noise processes.

GPS is are modeled as::

$$\mathbf{p}_m(t) = \begin{bmatrix} p_x(t) + \nu_{p,x}(t) \\ p_y(t) + \nu_{p,y}(t) \end{bmatrix}$$

where:

- $E[\nu_p] = 0$ m, $\mathrm{Var}[\nu_p] = 0.06$ m²

B. Bias Model

Bias is modeled as a linear function of time:

$$b_a(t) = b_{a,0} + b_{a,s} \cdot t$$
$$b_\omega(t) = b_{\omega,0} + b_{\omega,s} \cdot t$$

where $b_{a,0}$ and $b_{\omega,0}$ are initial biases, and $b_{a,s}$ and $b_{\omega,s}$ are bias drift rates.

C. INS Mechanization Equations (Section III)

For 2D vehicle dynamics:

$$\dot{p}_x(t) = v(t)\cos(\theta(t))$$
$$\dot{p}_y(t) = v(t)\sin(\theta(t))$$
$$\dot{v}(t) = a(t)$$
$$\dot{\theta}(t) = \omega(t)$$

As a baseline comparison, the trajectory is computed using INS mechanization alone (without GPS corrections):

$$\begin{bmatrix} p_x[k+1] \\ p_y[k+1] \\ v[k+1] \\ \theta[k+1] \end{bmatrix} = \begin{bmatrix} p_x[k] + v[k]\cos(\theta[k])\Delta t \\ p_y[k] + v[k]\sin(\theta[k])\Delta t \\ v[k] + a[k]\Delta t \\ \theta[k] + \omega[k]\Delta t \end{bmatrix}$$

where p_x, p_y are position coordinates, v is speed, θ is heading angle, a is forward acceleration, and ω is angular rate.

Initial State

$$\mathbf{x}(0) = \begin{bmatrix} p_x(0) \\ p_y(0) \\ v(0) \\ \theta(0) \end{bmatrix} = \begin{bmatrix} 0\text{ m} \\ 0\text{ m} \\ 0\text{ m/s} \\ 83.3^\circ \end{bmatrix}$$

D. Kalman Filter Framework

The discrete-time state-space model is:

$$\mathbf{x}_{[k+1]} = \mathbf{A}_{[k]}\mathbf{x}_{[k]} + \mathbf{B}_{[k]}\mathbf{u}_{[k]} + \mathbf{w}_{[k]}$$
$$\mathbf{z}_{[k]} = \mathbf{H}\mathbf{x}_{[k]} + \mathbf{v}_{[k]}$$

where $\mathbf{x} = [p_x, p_y, v, \theta]^T$ is the state vector, $\mathbf{u} = [a, \omega]^T$ is the control input, $\mathbf{z}$ is the measurement vector, and $\mathbf{w}$, $\mathbf{v}$ are process and measurement noise respectively.

The Kalman filter prediction and correction steps follow standard formulations [1], with state transition matrix $\mathbf{A}_{[k]}$ linearized around the current state estimate.

III. Section I: IMU Sensor calibration

Stationary accelerometer and gyroscope data were loaded from CSV files. For the z-axis accelerometer, gravity correction was applied by subtracting the local gravity value $g = 9.805$ m/s² before bias analysis, as the z-axis measures both motion and gravity when stationary.

A. *Bias Parameters*

Bias parameters were estimated for all axes of both sensors. The accelerometer bias parameters are:

- **X-axis**: $b_{a,0,x} = -2.35 \times 10^{-2}$ m/s², $b_{a,s,x} = 2.43 \times 10^{-6}$ m/s²/s
- **Y-axis**: $b_{a,0,y} = 1.71 \times 10^{-1}$ m/s², $b_{a,s,y} = -2.91 \times 10^{-6}$ m/s²/s
- **Z-axis**: $b_{a,0,z} = -6.01 \times 10^{-2}$ m/s², $b_{a,s,z} = 1.60 \times 10^{-6}$ m/s²/s

The gyroscope bias parameters are:

- **X-axis**: $b_{\omega,0,x} = 2.64 \times 10^{-4}$ rad/s, $b_{\omega,s,x} = -1.06 \times 10^{-7}$ rad/s²
- **Y-axis**: $b_{\omega,0,y} = 3.05 \times 10^{-4}$ rad/s, $b_{\omega,s,y} = 2.54 \times 10^{-8}$ rad/s²
- **Z-axis**: $b_{\omega,0,z} = -1.27 \times 10^{-4}$ rad/s, $b_{\omega,s,z} = -9.20 \times 10^{-8}$ rad/s²

Bias introduces systematic errors that accumulate over time during integration, making accurate calibration essential.

Figure 1 shows the acceleration measurements with fitted bias lines for all three axes. The linear trend is clearly visible, validating the linear bias model assumption.

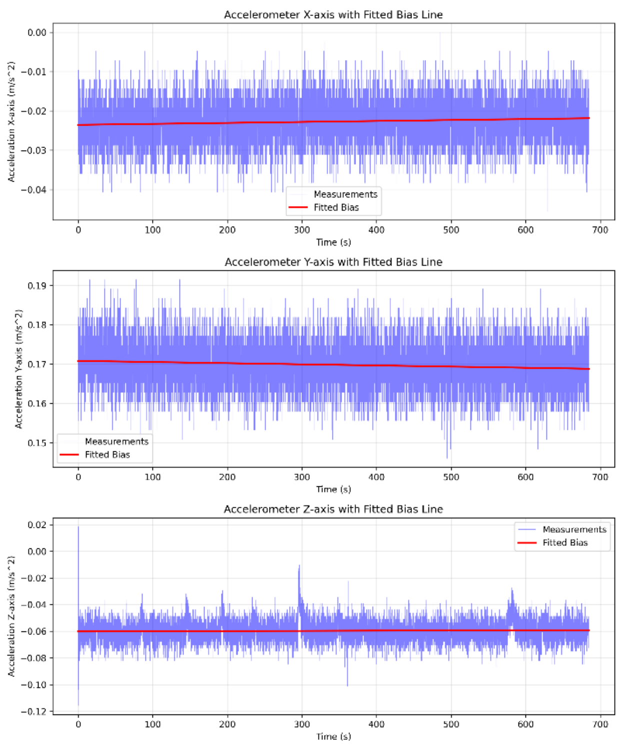

Fig. 1: Accelerometer measurements vs. time with fitted linear bias models for x, y, and z axes. The red lines show the estimated bias trends.

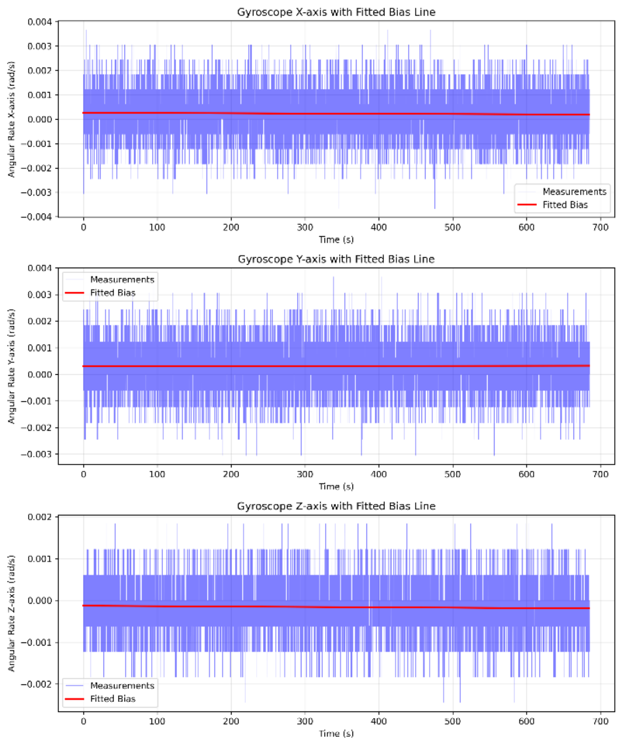

Fig. 2: Gyroscope measurements vs. time with fitted linear bias models for x, y, and z axes. The red lines show the estimated bias trends.

B. *Noise Statistics*

After bias removal, the noise statistics were computed.

***Table 1**: Noise variances for accelerometer and gyroscope sensors*

Axis	Accelerometer Variance (m/s²)²	Gyroscope Variance (rad/s)²
X-axis	2.86×10^{-5}	9.31×10^{-7}
Y-axis	3.33×10^{-5}	8.27×10^{-7}
Z-axis	6.45×10^{-5}	3.78×10^{-7}

The covariance matrices show small but non-zero off-diagonal elements. Accelerometer cross-axis covariances are on the order of 10^{-7} (m/s²)², compared to diagonal variances of 10^{-5} (m/s²)². This indicates weak correlation, suggesting the axes can be treated as largely **independent** for practical purposes.
Figure 3 and 4 shows histograms of accelerometer noise and Gyroscope respectively, with overlaid Gaussian PDFs. The excellent match between histograms and theoretical curves validates the Gaussian noise assumption.

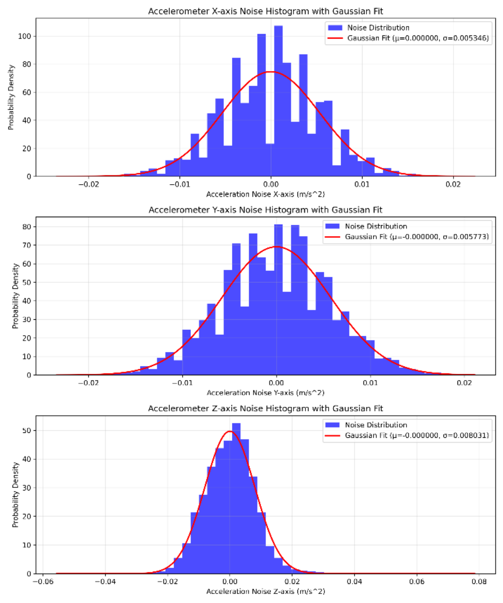

Fig. 3: Histograms of accelerometer measurement noise for x, y, and z axes with overlaid Gaussian probability density functions. The close match validates the Gaussian noise assumption.

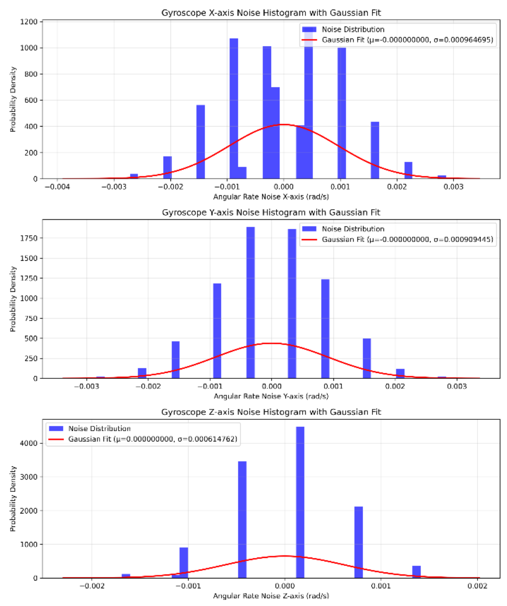

Fig. 4: Histograms of gyroscope measurement noise for x, y, and z axes with overlaid Gaussian probability density functions. The noise follows Gaussian distributions.

C. ***Answers to Section I Questions:***

- **Are bias values the same across axes?** No, biases differ across axes due to manufacturing variations, misalignment, and environmental factors. This is evident from the different bias parameter values listed above.
- **How does bias affect measurements?** Bias introduces systematic errors that accumulate over time during integration. Even small biases lead to significant position errors when double-integrating acceleration.
- **Is noise Gaussian?** Yes, the histograms with overlaid Gaussian PDFs show excellent agreement, validating the Gaussian noise assumption required for Kalman filtering.
- **Is noise independent across axes?** Approximately yes. The covariance matrices show weak cross-axis correlations (off-diagonal elements are 2-3 orders of magnitude smaller than diagonal variances), indicating near-independence.
- **How does noise affect measurements?** Measurement noise introduces uncertainty that accumulates during integration. Higher variance axes (e.g., z-axis accelerometer) contribute more uncertainty to integrated states.

IV. SECTION II: TRAJECTORY CHARACTERIZATION USING IMU DATA

Trajectory reconstruction was performed for two surveillance vehicles operating in a vertical pipeline. Vehicle 1 traveled downward from the top of the pipeline, while Vehicle 2 traveled upward from the bottom.

A. ***Methodology***

1. Data Correction

IMU measurements from moving vehicles were corrected using the following steps:

1. Removing estimated biases using parameters from Section I: The linear bias model $b(t) = b_0 + b_s t$ was applied to both accelerometer and gyroscope measurements.
2. Subtracting gravity from the z-axis accelerometer measurements: The local gravity value $g = 9.805$ m/s² was subtracted from z-axis accelerometer data to isolate motion acceleration.

2. Numerical Integration

Forward Euler integration was used to compute velocity and position:

$$v_{z,[k]} = v_{z,[k-1]} + a_{z,[k]} \cdot \Delta t$$

$$p_{z,[k]} = p_{z,[k-1]} + v_{z,[k]} \cdot \Delta t$$

where $\Delta t = t_{[k]} - t_{[k-1]}$ accounts for variable sampling rates. Critical attention was paid to using variable time steps rather than assuming constant sampling intervals.

3. **Trajectory Analysis**

Stopping points were identified using velocity thresholds (| v_z |< 0.1 m/s). However, due to sensor noise and integration drift, this threshold-based detection produces multiple false positives. Visual inspection of the velocity and position plots reveals that, excluding initial and final conditions, each vehicle has only one distinct stopping period where position remains stable and velocity is consistently near zero. Angular position changes during these distinct stop periods were computed to determine rotation direction and magnitude. Positive angular changes indicate left (counterclockwise) turns, while negative changes indicate right (clockwise) turns.

B. ***Vehicle 1 Results***

- Motion direction: **Down** (starting from top)
- Initial position: 16.000 m (top of 16 m pipeline)
- Final position: -3.808 m
- Total distance traveled: 19.81 m
- Number of stops: **6**
- Stop heights: 15.91 m, 7.23 m, 6.73 m, 6.22 m, -3.01 m, -3.16 m
- Distinct stop periods (excluding start/end): **1**
- Stop height: **7.0 m** (main inspection stop around t=30-40s)

Figure 5 shows the complete trajectory for Vehicle 1, including position, velocity, and acceleration profiles. The vehicle exhibits smooth motion with distinct stopping periods where inspections occur.

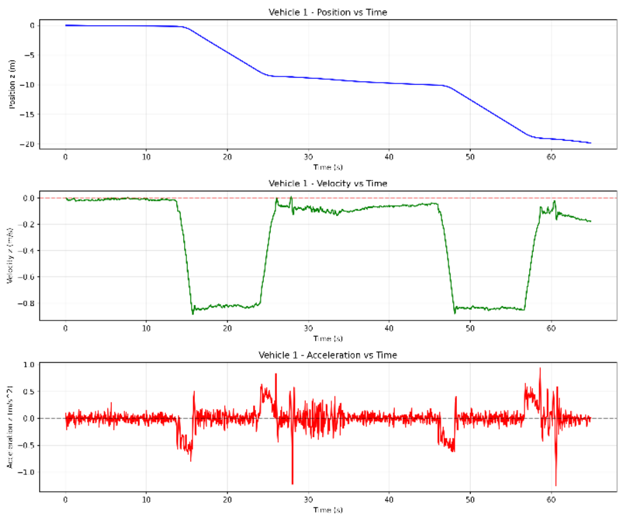

Fig. 5. Vehicle 1 trajectory showing (top) position, (middle) velocity, and (bottom) acceleration as functions of time. The vehicle moves downward with multiple stops for inspections.

Figure 7 shows angular position and angular rate for vehicle 1 during their inspection rotations.

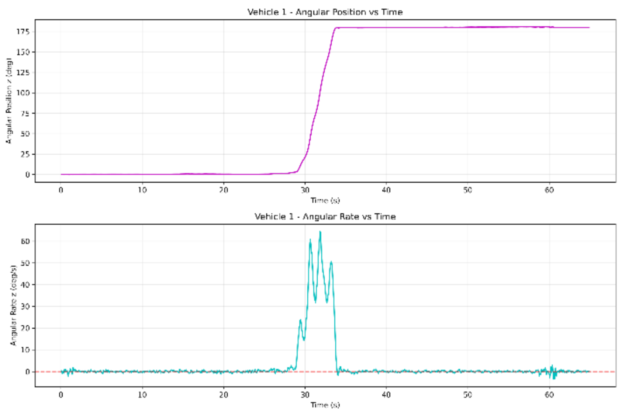

Fig. 7: Angular position and angular rate for Vehicle 1. Rotations occur during stopping periods for visual inspections.

C. ***Vehicle 2 Results***

- Motion direction: **Up** (starting from bottom)
- Initial position: 0.000 m
- Final position: 7.026 m
- Total distance traveled: 7.03 m
- Number of stops detected by algorithm: **5** (where | v_z |< 0.1 m/s, includes noise artifacts)
- Stop heights: -0.03 m, 9.21 m, 8.63 m, 8.49 m, 6.07 m, 9.78 m
- Distinct stop periods (excluding start/end): **1**
- Stop height: **9.2 m** (main inspection stop around t=29s)

Figure 6 shows the trajectory for Vehicle 2. Note the upward motion and different stopping pattern.

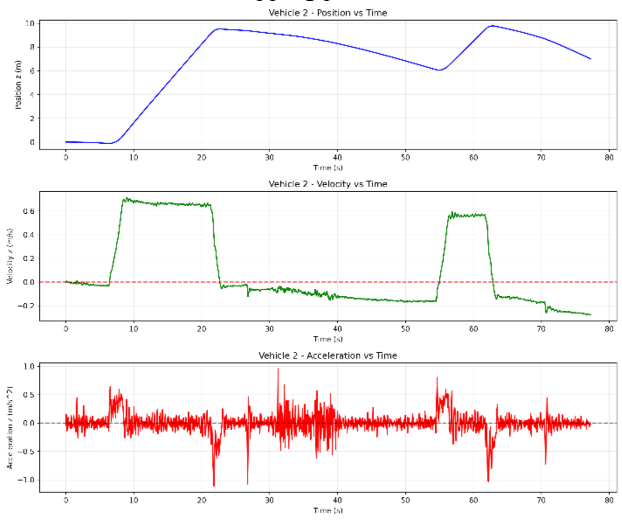

Fig. 5. Vehicle 2 trajectory showing (top) position, (middle) velocity, and (bottom) acceleration as functions of time. The vehicle moves upward with multiple stops.

Figure 8 shows angular position and angular rate for vehicle 2 during their inspection rotations.

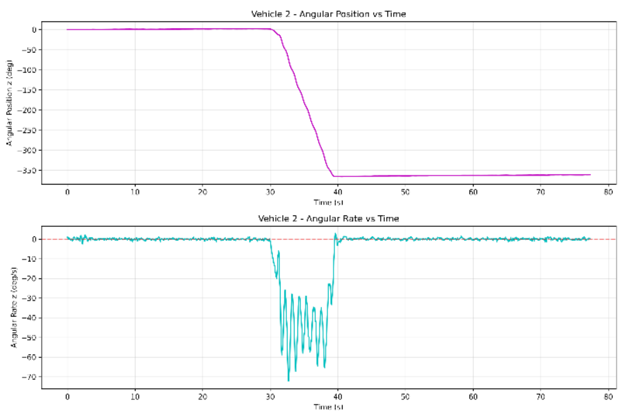

Fig. 8: Angular position and angular rate for Vehicle 2. Rotations occur during stopping periods for visual inspections.

D. ***Answers to Section II Questions:***

- **Which vehicle goes up/down?** Vehicle 1 moves downward (final position -3.8 m < initial 16 m), while Vehicle 2 moves upward (final position 7.03 m > initial 0 m). This is determined by comparing final and initial positions.
- **Stop heights:** The stop detection algorithm identifies any period where $|\ v_z\ | < 0.1$ m/s, which includes noise-induced false positives. Visual inspection of the velocity and position plots (Figures 5 and 6) reveals that, excluding the initial and final conditions, there is only one distinct stopping period for each vehicle where the position remains stable and velocity is consistently near zero. For Vehicle 1, this occurs at approximately 7.0 m height (around t=30-40s), where the vehicle pauses for inspection. For Vehicle 2, this occurs at approximately 9.2 m height (around t=29s), corresponding to the peak position where inspection is performed. The multiple detected stops (6 for Vehicle 1, 5 for Vehicle 2) are artifacts of the threshold-based detection combined with sensor noise and integration drift, which cause velocity to fluctuate around zero during the actual stop period.
- **Full pipeline traversal?** Vehicle 1 traveled approximately 19.65 m, indicating near-complete traversal and some more, going out of the 16 m pipeline. Vehicle 2 traveled only 7.03 m, indicating partial traversal stopping before reaching the top.
- **Rotation direction:** Vehicle 1 rotated 180.1° counter-clockwise (left) throughout the journey. Vehicle 2 rotated 361.2° clockwise (right) throughout the journey, completing slightly over one full 360° rotation.
- **360-degree inspections?** Yes. Vehicle 2 completed one full 360° clockwise rotation (361.2° total), indicating it performed a complete circular inspection. Vehicle 1 rotated 180.1° counter-clockwise, which is a half-rotation, not a full 360° inspection.

Figure 8 provides a 3D visualization of both vehicle trajectories combined.

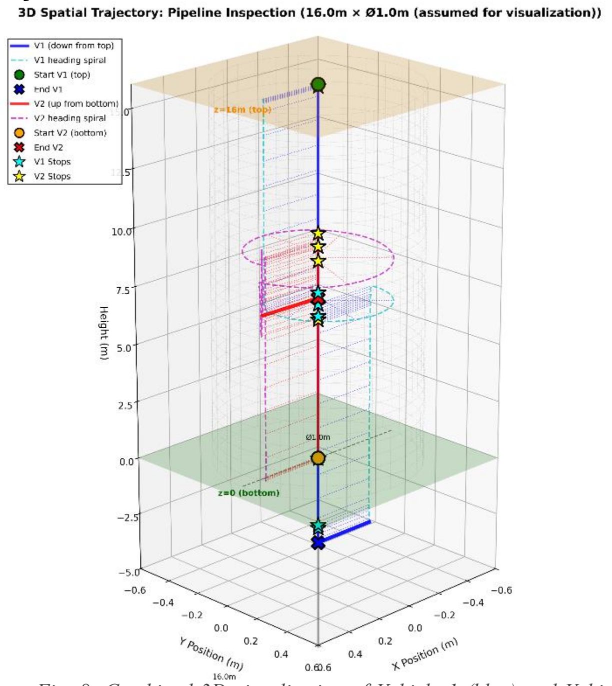

Fig. 8: Combined 3D visualization of Vehicle 1 (blue) and Vehicle 2 (orange) trajectories in the pipeline. The vertical axis represents height, and the spiral trails show rotation during stops.

I encourage you to loop at the separate side by side view of the two vehicles plot in the appendix and to look at the animation made for a better visualization which is attached in the submission folder.

V. Section III: Kalman Filter for sensor fusion

A. ***Methodology***

1. Coordinate Transformation

GPS latitude/longitude coordinates were transformed to local Cartesian coordinates (East-North frame) using:

$$x_{\text{local}} = (\lambda - \lambda_0) \cdot R \cdot cos(\phi_0)$$
$$y_{\text{local}} = (\phi - \phi_0) \cdot R$$

where $R = 6{,}371{,}000$ m is Earth's radius, and (ϕ_0, λ_0) is the reference point (initial position).

2. Continuous-Time State-Space Model

$$\dot{\mathbf{x}}(t) = \mathbf{A}_c(\mathbf{x}(t))\mathbf{x}(t) + \mathbf{B}_c\mathbf{u}(t) + \mathbf{B}_w\mathbf{w}(t)$$

where:

- **State vector**: $\mathbf{x}(t) = \begin{bmatrix} p_x(t) \\ p_y(t) \\ v(t) \\ \theta(t) \end{bmatrix}$

- **Control input**: $\mathbf{u}(t) = \begin{bmatrix} a(t) \\ \omega(t) \end{bmatrix}$

- **Process noise**: $\mathbf{w}(t) = \begin{bmatrix} \nu_a(t) \\ \nu_\omega(t) \end{bmatrix}$

Continuous-time state matrix $\mathbf{A}_c$ (linearized around current state):

$$\mathbf{A}_c = \begin{bmatrix} 0 & 0 & \cos(\theta) & -v\sin(\theta) \\ 0 & 0 & \sin(\theta) & v\cos(\theta) \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Continuous-time control input matrix $\mathbf{B}_c$:

$$\mathbf{B}_c = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

2. State-Space Model Discretization

Forward Euler Method
Using Euler forward approximation with time step Δt:

$$\mathbf{x}[k+1] = \mathbf{x}[k] + \Delta t \cdot \dot{\mathbf{x}}[k]$$

This gives the discrete-time nonlinear dynamics:

$$\begin{bmatrix} p_x[k+1] \\ p_y[k+1] \\ v[k+1] \\ \theta[k+1] \end{bmatrix} = \begin{bmatrix} p_x[k] + v[k]\cos(\theta[k])\Delta t \\ p_y[k] + v[k]\sin(\theta[k])\Delta t \\ v[k] + a[k]\Delta t \\ \theta[k] + \omega[k]\Delta t \end{bmatrix}$$

Zero Order Hold (ZOH) Method

For ZOH discretization, the state transition matrix is computed using matrix exponential:

$$\mathbf{A}[k] = e^{\mathbf{A}_c(\mathbf{x}[k])\cdot\Delta t}$$

The control input matrix for ZOH is:

$$\mathbf{B}[k] = \int_0^{\Delta t} e^{\mathbf{A}_c(\mathbf{x}[k])\lambda}\mathbf{B}_c \, d\lambda$$

B. ***Kalman Filter Implementation***

1. Linearized Discrete-Time State-Space Model

Linearization was done using Forward Euler, for faster processing and ease of implementation. ZOH method was also implemented but led to longer processing times and noisy estimation.

The linearized discrete-time state-space model using Forward Euler Method is:

$$\mathbf{x}[k+1] = \mathbf{A}[k]\mathbf{x}[k] + \mathbf{B}[k]\mathbf{u}[k] + \mathbf{w}[k]$$

Matrix $\mathbf{A}[k]$ *& B[k]* (Forward Euler):

$$\mathbf{A}[k] = \begin{bmatrix} 1 & 0 & \Delta t\cos(\theta[k]) & -\Delta t \cdot v[k]\sin(\theta[k]) \\ 0 & 1 & \Delta t\sin(\theta[k]) & \Delta t \cdot v[k]\cos(\theta[k]) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For this system, since control inputs only directly affect velocity and heading, the discrete-time B matrix simplifies to:

$$\mathbf{B}[k] = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix}$$

2. Kalman Filter Design

State Vector

$$\mathbf{x}[k] = \begin{bmatrix} p_x[k] \\ p_y[k] \\ v[k] \\ \theta[k] \end{bmatrix}$$

Measurement Model
GPS measures position only:

$$\mathbf{z}[k] = \mathbf{H}\mathbf{x}[k] + \mathbf{v}[k]$$

Measurement matrix H:

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

This means GPS measures $p_x[k]$ and $p_y[k]$ directly but **does not** measure speed $v[k]$ or heading $\theta[k]$.

Process Noise Covariance $\mathbf{Q}[k]$:
The process noise enters through control inputs. For Forward Euler discretization:

$$\mathbf{Q}[k] = \begin{bmatrix} q_p & 0 & 0 & 0 \\ 0 & q_p & 0 & 0 \\ 0 & 0 & \sigma_a^2 \Delta t^2 & 0 \\ 0 & 0 & 0 & \sigma_\omega^2 \Delta t^2 \end{bmatrix}$$

$\sigma_a^2 = 12.5$ m²/s⁴ (acceleration noise variance)

- $\sigma_\omega^2 = 0.001$ rad²/s² (angular rate noise variance)
- $q_p = 0.01$ m² (small position noise term for integration errors)
- $\Delta t = t[k] - t[k-1]$ (variable time step)

Measurement Noise Covariance R:

$$\mathbf{R} = \begin{bmatrix} \sigma_p^2 & 0 \\ 0 & \sigma_p^2 \end{bmatrix} = \begin{bmatrix} 0.06 & 0 \\ 0 & 0.06 \end{bmatrix}$$

where $\sigma_p^2 = 0.06$ m² is the GPS position measurement noise variance.

Initial Conditions

$$\hat{\mathbf{x}}(0 \mid 0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 83.3° \end{bmatrix}, \mathbf{P}(0 \mid 0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix}$$

2. Kalman Filter Algorithm
The filter operates in two steps:

Prediction (Time Update):

$$\hat{\mathbf{x}}_{[k|k-1]} = \mathbf{A}_{[k-1]} \hat{\mathbf{x}}_{[k-1|k-1]} + \mathbf{B}_{[k-1]} \mathbf{u}_{[k-1]}$$

$$\mathbf{P}_{[k|k-1]} = \mathbf{A}_{[k-1]} \mathbf{P}_{[k-1|k-1]} \mathbf{A}_{[k-1]}^T + \mathbf{Q}_{[k-1]}$$

Correction (Measurement Update) when GPS available:

$$\mathbf{y}_{[k]} = \mathbf{z}_{[k]} - \mathbf{H} \hat{\mathbf{x}}_{[k|k-1]}$$

$$\mathbf{S}_{[k]} = \mathbf{H} \mathbf{P}_{[k|k-1]} \mathbf{H}^T + \mathbf{R}$$

$$\mathbf{K}_{[k]} = \mathbf{P}_{[k|k-1]} \mathbf{H}^T \mathbf{S}_{[k]}^{-1}$$

$$\hat{\mathbf{x}}_{[k|k]} = \hat{\mathbf{x}}_{[k|k-1]} + \mathbf{K}_{\text{constrained},[k]} \mathbf{y}_{[k]}$$

$$\mathbf{P}_{[k|k]} = (\mathbf{I} - \mathbf{K}_{\text{constrained},[k]} \mathbf{H}) \mathbf{P}_{[k|k-1]}$$

Since GPS only measures position, the Kalman gain was constrained by zeroing rows corresponding to speed and heading, ensuring these states are determined purely by INS integration.

GPS measurements were synchronized to IMU time base using Zero Order Hold interpolation, holding the most recent GPS measurement until the next sample arrives.

*C. **Trajectory Comparison***

The Kalman filter successfully fuses IMU and GPS data, producing optimal trajectory estimates. Figure 9 shows the speed comparison between INS-only and Kalman filter estimates. The Kalman filter speed closely tracks the INS speed, confirming that speed is correctly derived from INS integration and not affected by GPS corrections.

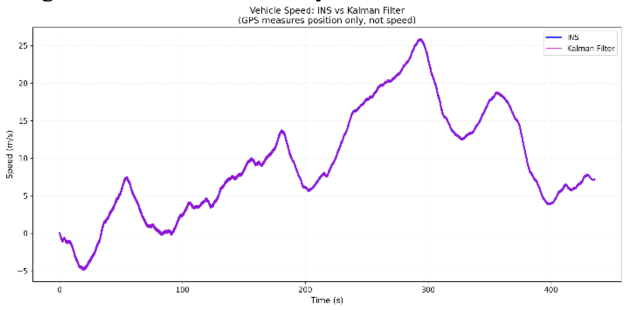

Fig 9: Speed comparison between INS-only (orange) and Kalman filter (blue) estimates. Both tracks are nearly identical, confirming that GPS corrections do not affect speed estimation.

Figure 10 shows the heading comparison. Again, the Kalman filter heading matches the INS heading, validating the constraint implementation.

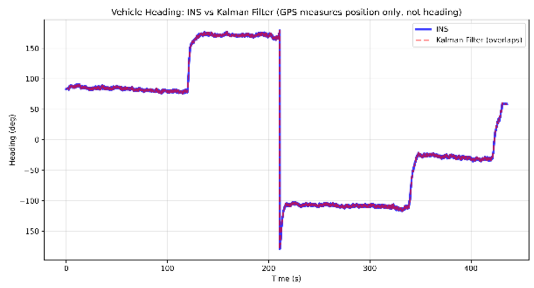

Fig. 10: Heading comparison between INS-only (orange) and Kalman filter (blue) estimates. The close match confirms heading is derived from INS integration only

Figures 11 and 12 show position comparisons. The INS-only trajectory exhibits significant drift over time, while the Kalman filter trajectory follows GPS measurements closely while maintaining smooth interpolation between GPS updates.

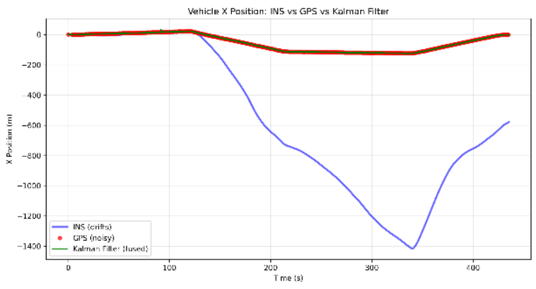

Fig 11: X-position comparison showing INS-only (orange), GPS (green), and Kalman filter (blue) estimates. The Kalman filter optimally combines both sources, reducing drift while maintaining high-frequency updates.

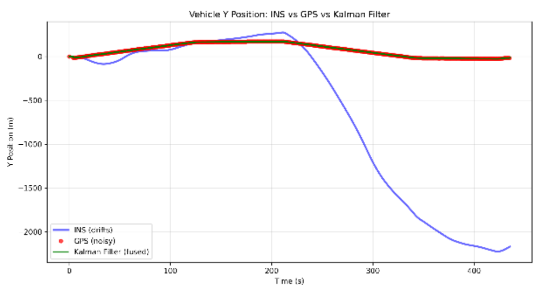

Fig 12: Y-position comparison showing INS-only (orange), GPS (green), and Kalman filter (blue) estimates. The Kalman filter provides accurate position estimates with bounded errors.

Figure 13 shows the complete 2D trajectory. The INS-only path drifts significantly, while the Kalman filter trajectory closely follows the GPS path with smooth interpolation.

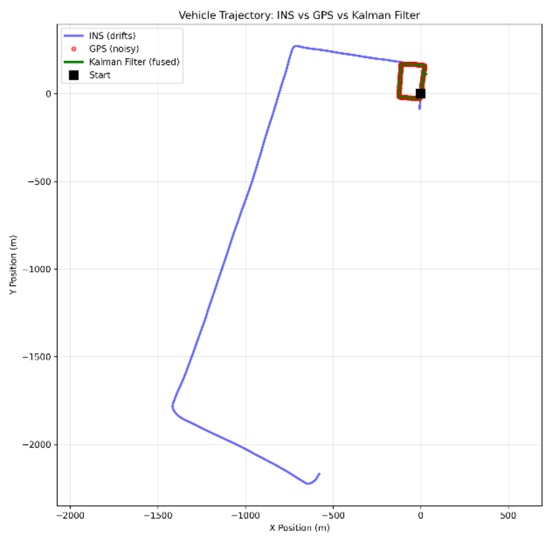

Fig 13: Complete 2D trajectory comparison. INS-only (orange) shows significant drift, GPS (green) is noisy but accurate, and Kalman filter (blue) optimally combines both for accurate, smooth estimates.

The Kalman filter demonstrates dramatically reduced drift compared to INS-only navigation, while providing smoother estimates than raw GPS measurements. The filter successfully maintains accuracy over the entire trajectory duration.

However, a critical limitation is observed in the heading estimation. Figure 14 shows how, even after Kalman filter implementation which correctly corrects for INS position bias, the heading angle drifts over time and no longer points in the direction of motion. The heading arrow becomes progressively misaligned with the actual vehicle trajectory. This occurs because GPS provides only position measurements (latitude and longitude, converted to p_x and p_y) and contains no heading information whatsoever. The measurement matrix $\mathbf{H}$ extracts only position coordinates, and the GPS data file contains no heading data. Consequently, the heading estimate θ must be derived exclusively from z-axis gyroscope integration, which accumulates bias and noise errors over time. Unlike position, which can be corrected by GPS measurements, there is simply no external heading measurement available to correct the gyroscope-derived heading. As the only source of heading data is the gyroscope, which drifts over time due to integration and residual bias effects, the heading estimate becomes increasingly inaccurate despite accurate position tracking. This demonstrates the fundamental limitation of using gyroscope-only heading in the absence of external heading references such as magnetometers, etc

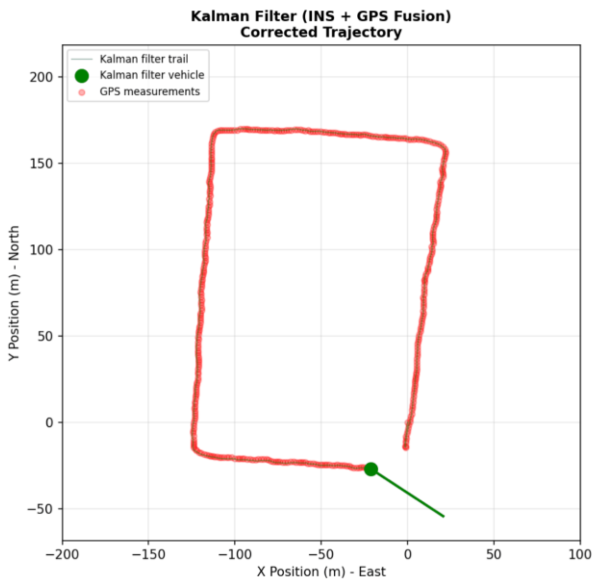

Fig 14: Animation frame showing Kalman filter trajectory with heading arrow. Despite accurate position tracking, the heading angle (indicated by the green arrow) drifts from the direction of motion over time due to gyroscope integration errors. The heading estimate matches the IMU-provided heading exactly but progressively becomes misaligned with the actual motion direction as visible in the trajectory path, highlighting the limitation of gyroscope-only heading estimation.

NOTE: Please refer to the animation attached in the project submission folder for better visualization of KF implementation

VI. Discussion

A. Advantages and Limitations of INS and GPS

INS Advantages: High update rate (100+ Hz), self-contained operation, short-term accuracy, direct measurement of acceleration and angular rate, continuous trajectory estimates.
INS Limitations: Integration drift grows quadratically ($\epsilon_p(t) \propto \frac{1}{2}\epsilon_a t^2$), bias instability requiring recalibration, need for known initial conditions, no absolute position reference, numerical integration errors accumulate.
GPS Advantages: Absolute global positioning, bounded error (1-5 m), no integration required, long-term accuracy without drift.
GPS Limitations: Low update rate (1-10 Hz) requires line-of-sight to satellites, multipath errors in urban areas, initialization delays, measurement noise on order of meters.

Complementary Nature: INS provides high-frequency updates but suffers unbounded drift; GPS provides absolute positioning but at low rates. Kalman filtering optimally combines both modalities.

B. Advantages and Limitations of Kalman Filter

Advantages: Optimal estimation under Gaussian noise (MMSE), computationally efficient recursive processing, uncertainty quantification via covariance matrices, natural sensor fusion framework, adaptive gain balancing prediction and measurement.

Limitations: Requires linear/linearized models (nonlinear systems need EKF/UKF), Gaussian noise assumption, performance depends on accurate system and noise models, $O(n^3)$ computational complexity for matrix inversion, requires careful tuning of noise covariances, can diverge if assumptions violated

C. Difficulties Encountered and Solutions

Section I:

- **Z-axis accelerometer measures both motion and gravity:** solved by subtracting local gravity before bias analysis.
- **Constant bias model inadequate**: implemented linear bias model $b(t) = b_0 + b_s t$ using least-squares fitting.

Section II:

- **Variable sampling rates caused errors**: calculated $\Delta t = t_{[i]} - t_{[i-1]}$ for each integration step.
- **Stop detection complicated by noise**: used velocity threshold with grouping, recognizing that visual inspection reveals only one distinct stop per vehicle (excluding start/end).
- **Rotation direction ambiguity**: normalized angular differences to $[-\pi, \pi]$ using atan2.

Section III Challenges:

- **Coordinate transformation**: GPS coordinates needed conversion to local frame. Solution: Implemented flat Earth transformation using Earth radius.
- **ZOH discretization**: Matrix exponential implementation initially caused instability. Solution: Simplified B matrix to prevent direct position corrections from control inputs.
- **Kalman gain constraint**: GPS was affecting unmeasured states. Solution: Zeroed speed and heading rows of Kalman gain matrix.
- **Heading alignment**: Coordinate system conventions required verification. Solution: Mathematically verified heading conversions between GPS and INS conventions.
- **GPS interpolation**: Needed synchronization with IMU time base. Solution: Implemented Zero Order Hold to hold most recent GPS measurement until next sample.

Key Insights: Sensor fusion is essential to overcome INS drift. Accurate calibration directly impacts trajectory quality. Variable time steps must be accounted for in numerical integration. Integration acts as a low-pass filter, smoothing high-frequency noise while accumulating low-frequency errors.

VII. CONTRIBUTION

This project was completed individually by me (Achal Patel). Although Section I was initially completed by a teammate during the group project phase, all work presented in this report—including implementation, analysis, visualization, and documentation—was performed independently by the me to deepen understanding of sensor fusion and Kalman filtering principles.

- **Section I**: Independent re-implementation of IMU sensor calibration, including bias parameter estimation using linear least-squares fitting, comprehensive noise analysis with statistical validation, and generation of all visualization plots. This included handling gravity correction for z-axis accelerometer measurements and implementing linear bias models with time-varying drift.

- **Section II**: Complete trajectory reconstruction implementation using INS mechanization equations, including Forward Euler numerical integration with variable time step handling. Developed algorithms for vehicle motion analysis, stopping point detection using velocity thresholding, and rotation direction analysis with proper angle normalization. Created comprehensive trajectory visualizations and animations.

- **Section III**: Full Kalman filter design and implementation from first principles, including state-space model formulation, coordinate transformation from GPS (latitude/longitude) to local Cartesian coordinates, Zero Order Hold (ZOH) discretization using matrix exponentials, and sensor fusion algorithm with proper constraint handling. Implemented visualization tools including animated comparisons of INS drift versus Kalman filter corrections.

- **Report Writing**: Complete technical report writing in IEEE-style format, including mathematical derivations, figure generation with appropriate captions, and comprehensive documentation. All plots and animations were generated programmatically using matplotlib.

All code was developed from scratch following mathematical formulations taught in class, with appropriate use of **NumPy** and **SciPy** libraries for numerical computations as detailed in the Appendix. The implementation strictly adheres to the state-space model approach, using explicit A, B, H, Q, and R matrices without relying on Jacobian-based linearization methods.

VIII. Conclusion

This project successfully demonstrated the complete pipeline from IMU sensor calibration to trajectory estimation using Kalman filtering. Key achievements include:

1. **Accurate sensor characterization**: Linear bias models were successfully fitted, and Gaussian noise properties were validated through statistical analysis. Bias values differ across sensor axes as expected, and noise exhibits weak cross-axis correlations.

2. **Successful trajectory reconstruction**: Vehicle trajectories were accurately reconstructed despite integration drift. The analysis successfully identified motion directions, stopping points, and rotation characteristics. Vehicle 1 completed near-full pipeline traversal, while Vehicle 2 performed partial traversal.

3. **Effective sensor fusion**: The Kalman filter successfully combines IMU and GPS measurements, achieving bounded position error while maintaining high-frequency updates. The filter demonstrates dramatically reduced drift compared to INS-only navigation, with smooth estimates superior to raw GPS measurements.

The results validate the effectiveness of sensor fusion for autonomous vehicle navigation, combining the complementary strengths of INS (high frequency, short-term accuracy) and GPS (absolute position, long-term accuracy). The implementation properly constrains GPS corrections to position only, ensuring physically meaningful speed and heading estimates from INS integration.

This project was both intellectually rewarding and enjoyable, providing hands-on experience with sensor fusion algorithms and their practical applications. The process of debugging numerical instabilities, refining coordinate transformations, and achieving stable filter performance through careful constraint design was particularly educational.

Future work could explore Unscented Kalman Filter (UKF) for improved nonlinear handling without requiring Jacobian computations, adaptive filtering techniques for automatic noise covariance tuning based on innovation statistics, and multi-sensor fusion incorporating additional sensors such as magnetometers for heading reference or wheel odometry for velocity measurements.

Additional Context: The knowledge gained from this project has been successfully applied to a practical robotics application. The author implemented a Kalman filter to fuse velocity estimates from a ZED2i camera (derived from position measurements) and an onboard IMU (derived from acceleration integration) on **CRALWR**, a rover platform at Concordia University's Aerospace Robotics Lab. Upon analysis, this implementation effectively functions as a complementary filter with an alpha gain favoring IMU measurements over camera-derived velocities, which is appropriate given that differentiated position measurements exhibit higher noise characteristics. This practical application demonstrates the transferability of sensor fusion concepts learned in this course to real-world autonomous navigation systems.

References

[1] L. Rodrigues, "Fundamentals of Navigation Systems," Draft, September 2024. (Navbook.pdf - Class Lecture Textbook)

[2] Phyphox - Physical phone experiments. Available: https://phyphox.org/.

[3] SciPy Documentation. Available: https://docs.scipy.org/.

[4] Kalman Filtering (SciPy Cookbook). Available: https://scipy-cookbook.readthedocs.io/items/KalmanFiltering.html

[5] The Kalman Filter. Available: https://thekalmanfilter.com/.

Appendix

A. *Additional Plots*

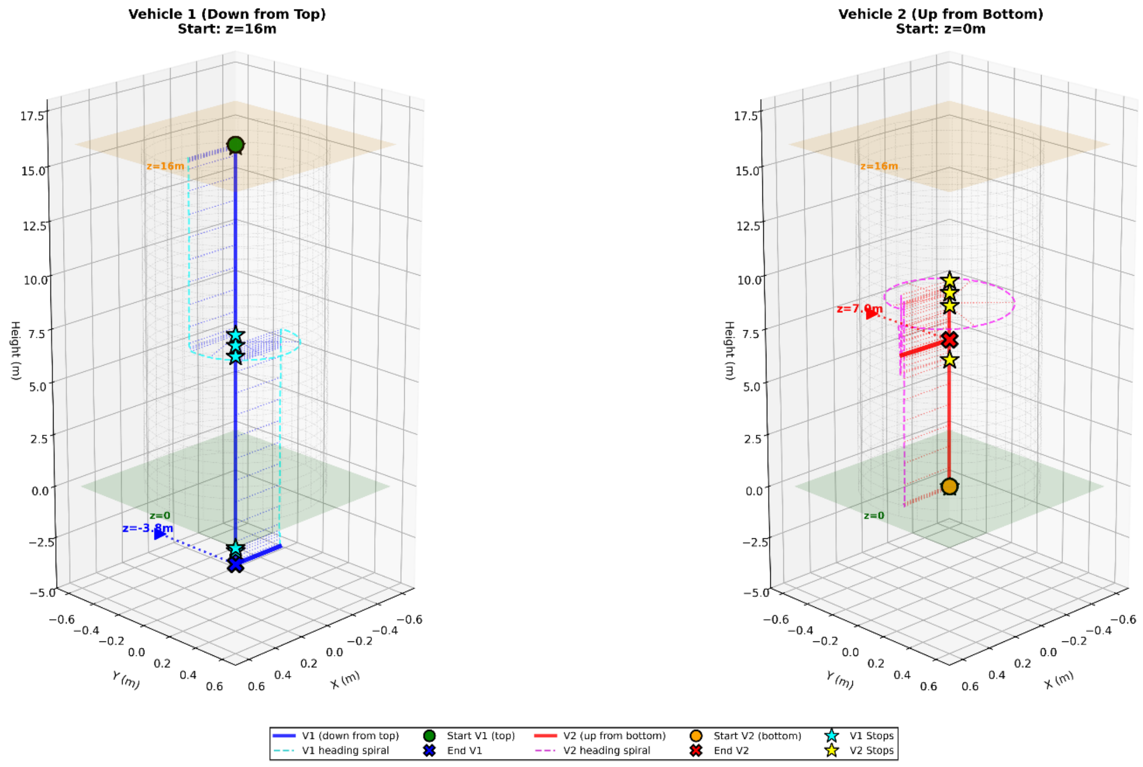

Fig. 15: Side-by-side 3D visualization of Vehicle 1 (left) and Vehicle 2 (right) trajectories in the pipeline. Vehicle 1 (blue) descends from the top (z=16 m, green start marker) with spiral rotations during stops (cyan dashed lines, cyan star markers) to its final position (z=-3.8 m, black X). Vehicle 2 (red) ascends from the bottom (z=0 m, orange start marker) with spiral rotations during stops (magenta dashed lines, yellow star markers) to its final position (z=7.0 m, red X). The translucent planes indicate the top (orange, z=16 m) and bottom (green, z=0 m) of the 16 m pipeline.

B. ***Library Function Used***

The following NumPy and SciPy functions were used for mathematical operations. While these could theoretically be implemented manually, using well-tested library functions ensures numerical stability.

Python NumPy functions:

- *np.polyfit():* Polynomial fitting for bias estimation (least-squares regression)
- *np.mean(), np.var(), np.cov():* Statistical computations (mean, variance, covariance)
- *np.matmul() / @:* Matrix multiplication for state-space operations
- *np.cos(), np.sin(), np.arctan2():* Trigonometric functions for coordinate transformations and INS equations
- *np.exp(), np.sqrt():* Mathematical operations

SciPy functions:

- scipy.linalg.expm(): Matrix exponential for ZOH discretization of state transition matrix

C. ***Code Structure***

This project consists of three main Python modules:

- **src/section_1.py**: IMU sensor calibration, bias fitting, noise analysis
- **src/section_2.py**: Trajectory reconstruction using INS equations

D. Code

Section1.py:

```python
"""
Section 1: IMU Sensor Calibration
Analyzes stationary IMU data to determine bias and measurement noise characteristics.
"""
•
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
import os
import json
•
# Constants
GRAVITY = 9.805  # m/s^2
•
def least_squares_line_fit(t, y):
    """
    Manual least-squares line fitting: y = b0 + b_s * t

    Solution: [b0, b_s] = (A^T * A)^(-1) * A^T * y
    where A = [1, t] (design matrix)

    Returns: b0 (intercept), b_s (slope)
    """
    n = len(t)
    # Design matrix: each row is [1, t_i]
    A = np.column_stack([np.ones(n), t])  # Stack 1's and t values horizontally

    # Normal equation: A^T * A
    ATA = A.T @ A  # @ is matrix multiplication

    # A^T * y
    ATy = A.T @ y

    # Solve: (A^T * A)^(-1) * A^T * y
    params = np.linalg.solve(ATA, ATy)  # Solves linear system ATA * params = ATy

    return params[0], params[1]  # b0, b_s
•
def main(data_dir, plots_dir, results_dir):
    """
    Run Section 1 calibration

    Returns:
    - acc_bias_params: dict with accelerometer bias parameters
    - gyr_bias_params: dict with gyroscope bias parameters
    """

    print("[INFO] Running Section 1: IMU Sensor Calibration")

    ACC_FILE = os.path.join(data_dir, "secI_acc.csv")
    GYR_FILE = os.path.join(data_dir, "secI_gyr.csv")

    print(f"[FILE] Loading accelerometer data from: {ACC_FILE}")
    print(f"[FILE] Loading gyroscope data from: {GYR_FILE}")

    # Load data
    acc_data = pd.read_csv(ACC_FILE)
    gyr_data = pd.read_csv(GYR_FILE)

    # Extract time and sensor readings
    t_acc = acc_data.iloc[:, 0].values
    acc_x = acc_data.iloc[:, 1].values
    acc_y = acc_data.iloc[:, 2].values
    acc_z = acc_data.iloc[:, 3].values

    t_gyr = gyr_data.iloc[:, 0].values
    gyr_x = gyr_data.iloc[:, 1].values
    gyr_y = gyr_data.iloc[:, 2].values
    gyr_z = gyr_data.iloc[:, 3].values

    print(f"[INFO] Accelerometer data: {len(t_acc)} samples")
    print(f"[INFO] Gyroscope data: {len(t_gyr)} samples")

    # Correct z-axis accelerometer for gravity
    acc_z_corrected = acc_z - GRAVITY

    # Fit bias models for accelerometer (linear model: b(t) = b0 + b_s * t)
    acc_axes = {'x': acc_x, 'y': acc_y, 'z': acc_z_corrected}
    acc_bias_params = {}

    for axis in ['x', 'y', 'z']:
        b0, b_s = least_squares_line_fit(t_acc, acc_axes[axis])
        acc_bias_params[axis] = {'b0': float(b0), 'b_s': float(b_s)}
        print(f"[OUTPUT] Accelerometer {axis}-axis: b0 = {b0:.6e}, b_s = {b_s:.9e}")

    # Fit bias models for gyroscope
    gyr_axes = {'x': gyr_x, 'y': gyr_y, 'z': gyr_z}
    gyr_bias_params = {}

    for axis in ['x', 'y', 'z']:
        b0, b_s = least_squares_line_fit(t_gyr, gyr_axes[axis])
        gyr_bias_params[axis] = {'b0': float(b0), 'b_s': float(b_s)}
        print(f"[OUTPUT] Gyroscope {axis}-axis: b0 = {b0:.9e}, b_s = {b_s:.12e}")

    # Remove bias from data to get noise
    acc_noise = {}
    gyr_noise = {}

     for axis in ['x', 'y', 'z']:
         # Inline bias calculation: b(t) = b0 + b_s * t
         bias_acc = acc_bias_params[axis]['b0'] + acc_bias_params[axis]['b_s'] * t_acc
         acc_noise[axis] = acc_axes[axis] - bias_acc

         bias_gyr = gyr_bias_params[axis]['b0'] + gyr_bias_params[axis]['b_s'] * t_gyr
         gyr_noise[axis] = gyr_axes[axis] - bias_gyr

     # Compute statistics
     acc_stats = {}
     gyr_stats = {}

     for axis in ['x', 'y', 'z']:
         acc_stats[axis] = {
             'mean': float(np.mean(acc_noise[axis])),
             'var': float(np.var(acc_noise[axis], ddof=0)),
         }
         gyr_stats[axis] = {
             'mean': float(np.mean(gyr_noise[axis])),
             'var': float(np.var(gyr_noise[axis], ddof=0)),
         }
         print(f"[OUTPUT] Accelerometer {axis}: mean = {acc_stats[axis]['mean']:.6e}, variance = {acc_stats[axis]['var']:.6e}")
         print(f"[OUTPUT] Gyroscope {axis}: mean = {gyr_stats[axis]['mean']:.9e}, variance = {gyr_stats[axis]['var']:.9e}")

    # Compute covariance matrices
    acc_noise_matrix = np.column_stack([acc_noise['x'], acc_noise['y'], acc_noise['z']])  # Stack columns
    gyr_noise_matrix = np.column_stack([gyr_noise['x'], gyr_noise['y'], gyr_noise['z']])

    # np.cov computes covariance matrix (each row is a variable, so transpose)
    acc_cov = np.cov(acc_noise_matrix.T).tolist()  # Convert to list for JSON serialization
    gyr_cov = np.cov(gyr_noise_matrix.T).tolist()

    # Create output directories
    section_plots_dir = os.path.join(plots_dir, "section_1")
    os.makedirs(section_plots_dir, exist_ok=True)
    os.makedirs(results_dir, exist_ok=True)

    # Plot accelerometer data with fitted lines
    fig, axes = plt.subplots(3, 1, figsize=(10, 12))

    for idx, axis in enumerate(['x', 'y', 'z']):
        data = acc_z_corrected if axis == 'z' else acc_axes[axis]
        bias_fit = acc_bias_params[axis]['b0'] + acc_bias_params[axis]['b_s'] * t_acc

        axes[idx].plot(t_acc, data, 'b-', alpha=0.5, linewidth=0.5, label='Measurements')
        axes[idx].plot(t_acc, bias_fit, 'r-', linewidth=2, label='Fitted Bias')
        axes[idx].set_xlabel('Time (s)')
        axes[idx].set_ylabel(f'Acceleration {axis.upper()}-axis (m/s^2)')
        axes[idx].set_title(f'Accelerometer {axis.upper()}-axis with Fitted Bias Line')
        axes[idx].grid(True, alpha=0.3)
        axes[idx].legend()

    plt.tight_layout()
    plot_path = os.path.join(section_plots_dir, "accelerometer_bias_fit.png")
    plt.savefig(plot_path, dpi=300)
    print(f"[FILE] Saved: {plot_path}")
    plt.close()

    # Plot gyroscope data with fitted lines
    fig, axes = plt.subplots(3, 1, figsize=(10, 12))

    for idx, axis in enumerate(['x', 'y', 'z']):
        bias_fit = gyr_bias_params[axis]['b0'] + gyr_bias_params[axis]['b_s'] * t_gyr

        axes[idx].plot(t_gyr, gyr_axes[axis], 'b-', alpha=0.5, linewidth=0.5, label='Measurements')
        axes[idx].plot(t_gyr, bias_fit, 'r-', linewidth=2, label='Fitted Bias')
        axes[idx].set_xlabel('Time (s)')
        axes[idx].set_ylabel(f'Angular Rate {axis.upper()}-axis (rad/s)')
        axes[idx].set_title(f'Gyroscope {axis.upper()}-axis with Fitted Bias Line')
        axes[idx].grid(True, alpha=0.3)
        axes[idx].legend()

    plt.tight_layout()
    plot_path = os.path.join(section_plots_dir, "gyroscope_bias_fit.png")
    plt.savefig(plot_path, dpi=300)
    print(f"[FILE] Saved: {plot_path}")
    plt.close()

    # Plot histograms with Gaussian fits for accelerometer
    fig, axes = plt.subplots(3, 1, figsize=(10, 12))

    for idx, axis in enumerate(['x', 'y', 'z']):
        noise = acc_noise[axis]
        mean = acc_stats[axis]['mean']
        std = np.sqrt(acc_stats[axis]['var'])

        axes[idx].hist(noise, bins=50, density=True, alpha=0.7, color='blue', label='Noise Distribution')

        x_gauss = np.linspace(noise.min(), noise.max(), 200)
        gauss_fit = stats.norm.pdf(x_gauss, mean, std)
        axes[idx].plot(x_gauss, gauss_fit, 'r-', linewidth=2, label=f'Gaussian Fit (μ={mean:.6f}, σ={std:.6f})')

        axes[idx].set_xlabel(f'Acceleration Noise {axis.upper()}-axis (m/s^2)')
        axes[idx].set_ylabel('Probability Density')
        axes[idx].set_title(f'Accelerometer {axis.upper()}-axis Noise Histogram with Gaussian Fit')
        axes[idx].grid(True, alpha=0.3)
        axes[idx].legend()

    plt.tight_layout()
    plot_path = os.path.join(section_plots_dir, "accelerometer_noise_histogram.png")
    plt.savefig(plot_path, dpi=300)
    print(f"[FILE] Saved: {plot_path}")
    plt.close()

    # Plot histograms with Gaussian fits for gyroscope
    fig, axes = plt.subplots(3, 1, figsize=(10, 12))

    for idx, axis in enumerate(['x', 'y', 'z']):
        noise = gyr_noise[axis]
        mean = gyr_stats[axis]['mean']
        std = np.sqrt(gyr_stats[axis]['var'])

        axes[idx].hist(noise, bins=50, density=True, alpha=0.7, color='blue', label='Noise Distribution')

        x_gauss = np.linspace(noise.min(), noise.max(), 200)
        gauss_fit = stats.norm.pdf(x_gauss, mean, std)
        axes[idx].plot(x_gauss, gauss_fit, 'r-', linewidth=2, label=f'Gaussian Fit (μ={mean:.9f}, σ={std:.9f})')

        axes[idx].set_xlabel(f'Angular Rate Noise {axis.upper()}-axis (rad/s)')
        axes[idx].set_ylabel('Probability Density')
        axes[idx].set_title(f'Gyroscope {axis.upper()}-axis Noise Histogram with Gaussian Fit')
        axes[idx].grid(True, alpha=0.3)
        axes[idx].legend()

    plt.tight_layout()
    plot_path = os.path.join(section_plots_dir, "gyroscope_noise_histogram.png")
    plt.savefig(plot_path, dpi=300)
    print(f"[FILE] Saved: {plot_path}")
    plt.close()

    # Save results to JSON
    results_data = {
        "accelerometer": {
            "bias_parameters": acc_bias_params,
            "noise_statistics": acc_stats,
            "covariance_matrix": acc_cov
        },
        "gyroscope": {
            "bias_parameters": gyr_bias_params,
            "noise_statistics": gyr_stats,
            "covariance_matrix": gyr_cov
        }
    }

    results_file = os.path.join(results_dir, "section_1_results.json")
    with open(results_file, 'w') as f:
        json.dump(results_data, f, indent=4)
    print(f"[FILE] Saved results to: {results_file}")

    print("[INFO] Section 1 completed\n")

    # Return bias parameters for use by other sections
    return acc_bias_params, gyr_bias_params

if __name__ == "__main__":
    # If run as standalone script
    SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
    PROJECT_ROOT = os.path.dirname(SCRIPT_DIR)
    DATA_DIR = os.path.join(PROJECT_ROOT, "data")
    PLOTS_DIR = os.path.join(PROJECT_ROOT, "plots")
    RESULTS_DIR = os.path.join(PROJECT_ROOT, "results")

    main(DATA_DIR, PLOTS_DIR, RESULTS_DIR)

```

Section2.py

```python
"""
Section 2: Trajectory Characterization Using IMU Data
Analyzes vehicle trajectories by integrating accelerometer and gyroscope data.
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import json

# Constants
GRAVITY = 9.805  # m/s^2

def integrate_forward_euler(t, values):
    """Integrate using Forward Euler method (matching class example style)"""
    n = len(t)
    integrated = np.zeros(n)  # Initialize array of zeros (same size as input)
    for i in range(1, n):
        # Forward Euler: integral[i] = integral[i-1] + f[i] * dt
        dt = t[i] - t[i-1]  # Calculate actual dt for each step (not constant!)
        integrated[i] = integrated[i-1] + values[i] * dt

    return integrated

def process_vehicle_data(acc_file, gyr_file, vehicle_name, acc_bias_params, gyr_bias_params):
    """Process IMU data for a single vehicle to compute trajectory"""

    # Load data
    acc_data = pd.read_csv(acc_file)
    gyr_data = pd.read_csv(gyr_file)
    print(f"[INFO] Loaded {len(acc_data)} accelerometer and {len(gyr_data)} gyroscope samples for {vehicle_name}")

    # Extract time and measurements
    t_acc = acc_data.iloc[:, 0].values
    acc_z = acc_data.iloc[:, 3].values

    t_gyr = gyr_data.iloc[:, 0].values
    gyr_z = gyr_data.iloc[:, 3].values

    # Remove bias from accelerometer (inline: b(t) = b0 + b_s * t)
    acc_z_corrected = acc_z - (acc_bias_params['z']['b0'] + acc_bias_params['z']['b_s'] * t_acc)

    # Remove gravity from z-axis
    acc_z_corrected = acc_z_corrected - GRAVITY

    # Remove bias from gyroscope (inline: b(t) = b0 + b_s * t)
    gyr_z_corrected = gyr_z - (gyr_bias_params['z']['b0'] + gyr_bias_params['z']['b_s'] * t_gyr)

    # Integrate z-axis acceleration to get velocity (Forward Euler)
    v_z = integrate_forward_euler(t_acc, acc_z_corrected)

    # Integrate velocity to get position (Forward Euler)
    p_z = integrate_forward_euler(t_acc, v_z)

    # Integrate z-axis angular rate to get angular position (Forward Euler)
    theta_z = integrate_forward_euler(t_gyr, gyr_z_corrected)

    print(f"[INFO] Trajectory computed for {vehicle_name}")
    return {
        'time_acc': t_acc,
        'time_gyr': t_gyr,
        'acceleration_z': acc_z_corrected,
        'velocity_z': v_z,
        'position_z': p_z,
        'angular_rate_z': gyr_z_corrected,
        'angular_position_z': theta_z
    }

def analyze_trajectory(results, vehicle_name):
    """Analyze trajectory characteristics: direction, stops, rotations"""

    p_z = results['position_z']
    v_z = results['velocity_z']
    theta_z = results['angular_position_z']
    t_acc = results['time_acc']
    t_gyr = results['time_gyr']

    # Determine motion direction
    final_position = p_z[-1]
    direction = "up" if final_position > p_z[0] else "down"
    print(f"[OUTPUT] {vehicle_name} is moving {direction} (final position: {final_position:.2f} m)")

    # Calculate TOTAL rotation from start to finish
    total_rotation_rad = theta_z[-1] - theta_z[0]
    total_rotation_deg = np.degrees(total_rotation_rad)
    num_full_rotations = total_rotation_deg / 360.0
    rotation_direction = "right (clockwise)" if total_rotation_deg < 0 else "left (counter-clockwise)"

    print(f"[OUTPUT] TOTAL rotation: {total_rotation_deg:.1f}° {rotation_direction}")
    print(f"[OUTPUT] Number of full 360° rotations: {abs(num_full_rotations):.2f}")

    # Find stopping points (velocity near zero)
    velocity_threshold = 0.1  # m/s
    stopping_indices = np.where(np.abs(v_z) < velocity_threshold)[0]  # np.where returns indices where condition is True

    # Identify distinct stopping periods
    stops = []
    if len(stopping_indices) > 0:
        stop_groups = []
        current_group = [stopping_indices[0]]

        for i in range(1, len(stopping_indices)):
            if stopping_indices[i] - stopping_indices[i-1] < 10:
                current_group.append(stopping_indices[i])
            else:
                stop_groups.append(current_group)
                current_group = [stopping_indices[i]]
        stop_groups.append(current_group)

        for group in stop_groups:
            if len(group) > 5:
                center_idx = group[len(group)//2]
                stop_time = t_acc[center_idx]
                stop_height = p_z[center_idx]
                stops.append({'time': stop_time, 'height': stop_height, 'indices': group})
                print(f"[OUTPUT] Stop detected at t={stop_time:.2f}s, height={stop_height:.2f}m")

    # Analyze rotations during stops
    rotations = []
    for stop in stops:
        stop_start_idx = stop['indices'][0]
        stop_end_idx = stop['indices'][-1]

        stop_time_start = t_acc[stop_start_idx]
        stop_time_end = t_acc[stop_end_idx]
        # np.argmin finds the index of the minimum value
        gyr_start_idx = np.argmin(np.abs(t_gyr - stop_time_start))  # Find closest gyro timestamp
        gyr_end_idx = np.argmin(np.abs(t_gyr - stop_time_end))

        if gyr_end_idx > gyr_start_idx:
            theta_change = theta_z[gyr_end_idx] - theta_z[gyr_start_idx]

            # Normalize to [-π, π] range
            while theta_change > np.pi:
                theta_change -= 2*np.pi
            while theta_change < -np.pi:
                theta_change += 2*np.pi

            rotation_deg = np.degrees(theta_change)

            if abs(rotation_deg) > 10:
                direction_rot = "left" if rotation_deg > 0 else "right"
                full_rotation = abs(rotation_deg) >= 350
                rotations.append({
                    'stop_height': stop['height'],
                    'rotation': rotation_deg,
                    'direction': direction_rot,
                    'full_360': full_rotation
                })
                print(f"[OUTPUT] Rotation: {rotation_deg:.1f}° {direction_rot} ({'full 360°' if full_rotation else 'partial'})")

    return {
        'direction': direction,
        'stops': stops,
        'rotations_during_stops': rotations,  # Rotation only during stop periods
        'total_rotation_deg': float(total_rotation_deg),  # TOTAL rotation throughout journey
        'total_rotation_direction': rotation_direction,
        'num_full_rotations': float(abs(num_full_rotations)),
        'final_position': final_position,
        'initial_position': p_z[0]
    }

def main(data_dir, plots_dir, results_dir, acc_bias_params, gyr_bias_params):
    """
    Run Section 2 trajectory analysis

    Parameters:
    - data_dir: path to data directory
    - plots_dir: path to plots directory
    - results_dir: path to results directory
    - acc_bias_params: accelerometer bias parameters from Section 1
    - gyr_bias_params: gyroscope bias parameters from Section 1
    """

    print("[INFO] Running Section 2: Trajectory Characterization")

    # Process vehicle 1
    vehicle1_results = process_vehicle_data(
            os.path.join(data_dir, "secII_acc_1.csv"),
            os.path.join(data_dir, "secII_gyr_1.csv"),
        "Vehicle 1",
        acc_bias_params,
        gyr_bias_params
    )

    # Process vehicle 2
    vehicle2_results = process_vehicle_data(
            os.path.join(data_dir, "secII_acc_2.csv"),
            os.path.join(data_dir, "secII_gyr_2.csv"),
        "Vehicle 2",
        acc_bias_params,
        gyr_bias_params
    )

    # Analyze trajectories
    vehicle1_analysis = analyze_trajectory(vehicle1_results, "Vehicle 1")
    vehicle2_analysis = analyze_trajectory(vehicle2_results, "Vehicle 2")

    # Create plots
    section_plots_dir = os.path.join(plots_dir, "section_2")
    os.makedirs(section_plots_dir, exist_ok=True)
    os.makedirs(results_dir, exist_ok=True)

    vehicles = [
        (vehicle1_results, vehicle1_analysis, "Vehicle 1"),
        (vehicle2_results, vehicle2_analysis, "Vehicle 2")
    ]

    # Plot position, velocity, acceleration for each vehicle
    for results, analysis, name in vehicles:
        fig, axes = plt.subplots(3, 1, figsize=(12, 10))

        t = results['time_acc']
        p_z = results['position_z']
        v_z = results['velocity_z']
        a_z = results['acceleration_z']

        axes[0].plot(t, p_z, 'b-', linewidth=1.5)
        axes[0].set_xlabel('Time (s)')
        axes[0].set_ylabel('Position z (m)')
        axes[0].set_title(f'{name} - Position vs Time')
        axes[0].grid(True, alpha=0.3)

        axes[1].plot(t, v_z, 'g-', linewidth=1.5)
        axes[1].set_xlabel('Time (s)')
        axes[1].set_ylabel('Velocity z (m/s)')
        axes[1].set_title(f'{name} - Velocity vs Time')
        axes[1].grid(True, alpha=0.3)
        axes[1].axhline(y=0, color='r', linestyle='--', alpha=0.5)

        axes[2].plot(t, a_z, 'r-', linewidth=1.5)
        axes[2].set_xlabel('Time (s)')
        axes[2].set_ylabel('Acceleration z (m/s^2)')
        axes[2].set_title(f'{name} - Acceleration vs Time')
        axes[2].grid(True, alpha=0.3)
        axes[2].axhline(y=0, color='k', linestyle='--', alpha=0.5)

        plt.tight_layout()
        filename = os.path.join(section_plots_dir, f"{name.lower().replace(' ', '_')}_trajectory.png")
        plt.savefig(filename, dpi=300)
        print(f"[FILE] Saved: {filename}")
        plt.close()

    # Create 3D spatial visualization - vehicles at centerline rotating to scan walls
    from mpl_toolkits.mplot3d import Axes3D
    from matplotlib.lines import Line2D

    PIPELINE_RADIUS = 0.5   # m (assumed for viz)
    PIPELINE_LENGTH = 16.0  # m (given in project description)

    def plot_trajectory_3d_separate(v1_results, v1_analysis, z1, theta1,
                                    v2_results, v2_analysis, z2, theta2,
                                    plots_dir, pipe_r, pipe_len):
        """Create 1x2 grid with separate 3D plots for each vehicle."""
        fig = plt.figure(figsize=(18, 9))

        # Common plot setup function
        def setup_vehicle_plot(ax, z, theta, results, analysis, v_name, color, start_color, end_color, stop_color, spiral_color):
            # Draw pipeline cylinder
            theta_cyl = np.linspace(0, 2*np.pi, 30)
            z_cyl = np.linspace(0, pipe_len, 30)
            Theta_cyl, Z_cyl = np.meshgrid(theta_cyl, z_cyl)
            X_cyl = pipe_r * np.cos(Theta_cyl)
            Y_cyl = pipe_r * np.sin(Theta_cyl)
            ax.plot_wireframe(X_cyl, Y_cyl, Z_cyl, alpha=0.2, color='gray', linewidth=0.5, linestyle='--')

            # Vehicle at centerline
            x = np.zeros_like(z)
            y = np.zeros_like(z)

            # Plot trajectory
            ax.plot(x, y, z, color=color, linewidth=3, alpha=0.8, label=f'{v_name} path')
            ax.scatter(x[0], y[0], z[0], color=start_color, s=200, marker='o', edgecolors='black', linewidths=2, zorder=5)
            ax.scatter(x[-1], y[-1], z[-1], color=end_color, s=200, marker='X', edgecolors='black', linewidths=2, zorder=5)

            # Draw heading spiral
            view_len = 0.35
            skip = max(1, len(z) // 50)
            heading_x, heading_y, heading_z = [], [], []
            for i in range(0, len(z), skip):
                heading_x.extend([0, view_len * np.cos(theta[i]), np.nan])
                heading_y.extend([0, view_len * np.sin(theta[i]), np.nan])
                heading_z.extend([z[i], z[i], np.nan])
            ax.plot(heading_x, heading_y, heading_z, color=color, linestyle=':', linewidth=1, alpha=0.5)

            spiral_x = view_len * np.cos(theta)
            spiral_y = view_len * np.sin(theta)
            ax.plot(spiral_x, spiral_y, z, color=spiral_color, linestyle='--', linewidth=1.5, alpha=0.7)

            # Final heading arrow
            ax.plot([0, view_len * np.cos(theta[-1])], [0, view_len * np.sin(theta[-1])],
                   [z[-1], z[-1]], color=color, linewidth=4, zorder=5, alpha=0.9)

            # Stop markers
            for stop in analysis['stops']:
                stop_idx = np.argmin(np.abs(results['time_acc'] - stop['time']))
                ax.scatter(x[stop_idx], y[stop_idx], z[stop_idx], color=stop_color, s=300, marker='*',
                          edgecolors='black', linewidths=1.5, zorder=10)

            # FINAL POSITION INDICATOR - dotted line from z-axis to final position
            final_z = z[-1]
            # Horizontal dotted line from z-axis (at y=-0.6) to centerline (y=0) at final z height
            ax.plot([0, 0], [-0.6, 0], [final_z, final_z],
                   color=color, linestyle=':', linewidth=2, alpha=0.8)
            # Marker on z-axis edge
            ax.scatter([0], [-0.6], [final_z], color=color, s=100, marker='>', zorder=10)
            # Label showing final z value
            ax.text(0, -0.7, final_z, f'z={final_z:.1f}m', fontsize=10, fontweight='bold',
                   color=color, ha='center', va='center')

            # Reference planes
            xx, yy = np.meshgrid([-0.6, 0.6], [-0.6, 0.6])
            ax.plot_surface(xx, yy, np.zeros_like(xx), alpha=0.15, color='green', edgecolor='none')
            ax.text(0.65, 0, 0, 'z=0', fontsize=9, color='darkgreen', fontweight='bold')
            ax.plot_surface(xx, yy, np.ones_like(xx) * pipe_len, alpha=0.15, color='orange', edgecolor='none')
            ax.text(0.65, 0, pipe_len, f'z={pipe_len:.0f}m', fontsize=9, color='darkorange', fontweight='bold')

            ax.set_xlabel('X (m)', fontsize=10, labelpad=8)
            ax.set_ylabel('Y (m)', fontsize=10, labelpad=8)
            ax.set_zlabel('Height (m)', fontsize=10, labelpad=8)
            ax.set_xlim([-0.7, 0.7])
            ax.set_ylim([-0.7, 0.7])
            ax.set_zlim([-5, pipe_len + 2])
            ax.set_box_aspect([1, 1, 2])
            ax.view_init(elev=20, azim=45)
            ax.grid(True, alpha=0.3)

        # Vehicle 1 plot (left)
        ax1 = fig.add_subplot(121, projection='3d')
        setup_vehicle_plot(ax1, z1, theta1, v1_results, v1_analysis,
                          'V1', 'blue', 'green', 'blue', 'cyan', 'cyan')
        ax1.set_title('Vehicle 1 (Down from Top)\nStart: z=16m', fontsize=12, fontweight='bold', pad=15)

        # Vehicle 2 plot (right)
        ax2 = fig.add_subplot(122, projection='3d')
        setup_vehicle_plot(ax2, z2, theta2, v2_results, v2_analysis,
                          'V2', 'red', 'orange', 'red', 'yellow', 'magenta')
        ax2.set_title('Vehicle 2 (Up from Bottom)\nStart: z=0m', fontsize=12, fontweight='bold', pad=15)

        # Create shared legend for both plots
        legend_elements = [
            Line2D([0], [0], color='b', linewidth=3, alpha=0.8, label='V1 (down from top)'),
            Line2D([0], [0], color='c', linewidth=1.5, linestyle='--', alpha=0.7, label='V1 heading spiral'),
            Line2D([0], [0], marker='o', color='w', markerfacecolor='green', markersize=10,
                   markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='Start V1 (top)'),
            Line2D([0], [0], marker='X', color='w', markerfacecolor='blue', markersize=10,
                   markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='End V1'),
            Line2D([0], [0], color='r', linewidth=3, alpha=0.8, label='V2 (up from bottom)'),
            Line2D([0], [0], color='m', linewidth=1.5, linestyle='--', alpha=0.7, label='V2 heading spiral'),
            Line2D([0], [0], marker='o', color='w', markerfacecolor='orange', markersize=10,
                   markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='Start V2 (bottom)'),
            Line2D([0], [0], marker='X', color='w', markerfacecolor='red', markersize=10,
                   markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='End V2'),
            Line2D([0], [0], marker='*', color='w', markerfacecolor='cyan', markersize=14,
                   markeredgecolor='black', markeredgewidth=1, linestyle='None', label='V1 Stops'),
            Line2D([0], [0], marker='*', color='w', markerfacecolor='yellow', markersize=14,
                   markeredgecolor='black', markeredgewidth=1, linestyle='None', label='V2 Stops')
        ]
        fig.legend(handles=legend_elements, loc='upper center', ncol=5, fontsize=9,
                  framealpha=0.9, edgecolor='black', fancybox=False, bbox_to_anchor=(0.5, 0.02))

        plt.tight_layout(rect=[0, 0.08, 1, 1])  # Leave space at bottom for legend
        filename = os.path.join(plots_dir, "trajectory_3d.png")
        plt.savefig(filename, dpi=300, bbox_inches='tight')
        print(f"[FILE] Saved: {filename}")
        plt.close()

    # Vehicles stay at centerline (x=0, y=0), only move in z and rotate around z-axis
    # The rotation angle tells us which direction they're "looking" at the walls

    # Vehicle 1 - goes DOWN, so starts at TOP (z=16m)
    z1_raw = vehicle1_results['position_z']
    theta1_interp = np.interp(vehicle1_results['time_acc'], vehicle1_results['time_gyr'], vehicle1_results['angular_position_z'])
    x1 = np.zeros_like(z1_raw)  # Stay at centerline
    y1 = np.zeros_like(z1_raw)
    z1 = z1_raw + PIPELINE_LENGTH  # Offset to start at top (z=16m)

    # Vehicle 2 - goes UP, so starts at BOTTOM (z=0)
    z2_raw = vehicle2_results['position_z']
    theta2_interp = np.interp(vehicle2_results['time_acc'], vehicle2_results['time_gyr'], vehicle2_results['angular_position_z'])
    x2 = np.zeros_like(z2_raw)  # Stay at centerline
    y2 = np.zeros_like(z2_raw)
    z2 = z2_raw  # No offset, starts at bottom (z=0)

    print(f"[INFO] V1 (down from top): z=[{z1.min():.1f}, {z1.max():.1f}]m")
    print(f"[INFO] V2 (up from bottom): z=[{z2.min():.1f}, {z2.max():.1f}]m")

    # Create 3D plot
    fig = plt.figure(figsize=(14, 10))
    ax = fig.add_subplot(111, projection='3d')

    # Draw pipeline cylinder walls (dotted wireframe)
    theta_cyl = np.linspace(0, 2*np.pi, 30)
    z_cyl = np.linspace(0, 16, 30)
    Theta_cyl, Z_cyl = np.meshgrid(theta_cyl, z_cyl)
    X_cyl = PIPELINE_RADIUS * np.cos(Theta_cyl)
    Y_cyl = PIPELINE_RADIUS * np.sin(Theta_cyl)
    ax.plot_wireframe(X_cyl, Y_cyl, Z_cyl, alpha=0.2, color='gray', linewidth=0.5, linestyle='--')

    # Draw pipeline dimensions reference
    ax.plot([-PIPELINE_RADIUS, PIPELINE_RADIUS], [0, 0], [0, 0], 'k--', linewidth=1, alpha=0.5)
    ax.text(0, 0, 0.8, f'Ø{2*PIPELINE_RADIUS:.1f}m', fontsize=8, ha='center')
    ax.plot([0], [0], [-PIPELINE_LENGTH, 0], 'k-', linewidth=2, alpha=0.4)
    ax.text(0.7, 0, -PIPELINE_LENGTH/2, f'{PIPELINE_LENGTH}m', fontsize=8, ha='left', rotation=90)

    # Plot Vehicle 1 trajectory (blue - centerline, going down from TOP)
    ax.plot(x1, y1, z1, 'b-', linewidth=3, alpha=0.8, label='Vehicle 1 path (down from top)')
    ax.scatter(x1[0], y1[0], z1[0], color='green', s=200, marker='o', edgecolors='black', linewidths=2, label='Start V1 (top)', zorder=5)
    ax.scatter(x1[-1], y1[-1], z1[-1], color='blue', s=200, marker='X', edgecolors='black', linewidths=2, label='End V1', zorder=5)

    # Draw rotation spiral for V1 - dotted lines showing heading at intervals
    view_len = 0.35
    skip_v1 = max(1, len(z1) // 50)  # Show ~50 heading lines
    heading_x1 = []
    heading_y1 = []
    heading_z1 = []
    for i in range(0, len(z1), skip_v1):
        # Line from center to wall showing heading direction
        heading_x1.extend([0, view_len * np.cos(theta1_interp[i]), np.nan])
        heading_y1.extend([0, view_len * np.sin(theta1_interp[i]), np.nan])
        heading_z1.extend([z1[i], z1[i], np.nan])
    ax.plot(heading_x1, heading_y1, heading_z1, 'b:', linewidth=1, alpha=0.5)

    # Draw spiral connecting the heading tips (shows rotation pattern)
    spiral_x1 = view_len * np.cos(theta1_interp)
    spiral_y1 = view_len * np.sin(theta1_interp)
    ax.plot(spiral_x1, spiral_y1, z1, 'c--', linewidth=1.5, alpha=0.7, label='V1 heading spiral')

    # Final heading arrow (solid)
    hx1 = [0, view_len * np.cos(theta1_interp[-1])]
    hy1 = [0, view_len * np.sin(theta1_interp[-1])]
    hz1 = [z1[-1], z1[-1]]
    ax.plot(hx1, hy1, hz1, 'b-', linewidth=4, zorder=5, alpha=0.9)

    # Add stop markers for Vehicle 1
    for stop in vehicle1_analysis['stops']:
        stop_idx = np.argmin(np.abs(vehicle1_results['time_acc'] - stop['time']))
        ax.scatter(x1[stop_idx], y1[stop_idx], z1[stop_idx], color='cyan', s=300, marker='*',
                  edgecolors='black', linewidths=1.5, zorder=10)

    # Plot Vehicle 2 trajectory (red - centerline, going up from BOTTOM)
    ax.plot(x2, y2, z2, 'r-', linewidth=3, alpha=0.8, label='Vehicle 2 path (up from bottom)')
    ax.scatter(x2[0], y2[0], z2[0], color='orange', s=200, marker='o', edgecolors='black', linewidths=2, label='Start V2 (bottom)', zorder=5)
    ax.scatter(x2[-1], y2[-1], z2[-1], color='red', s=200, marker='X', edgecolors='black', linewidths=2, label='End V2', zorder=5)

    # Draw rotation spiral for V2 - dotted lines showing heading at intervals
    skip_v2 = max(1, len(z2) // 50)
    heading_x2 = []
    heading_y2 = []
    heading_z2 = []
    for i in range(0, len(z2), skip_v2):
        heading_x2.extend([0, view_len * np.cos(theta2_interp[i]), np.nan])
        heading_y2.extend([0, view_len * np.sin(theta2_interp[i]), np.nan])
        heading_z2.extend([z2[i], z2[i], np.nan])
    ax.plot(heading_x2, heading_y2, heading_z2, 'r:', linewidth=1, alpha=0.5)

    # Draw spiral connecting the heading tips
    spiral_x2 = view_len * np.cos(theta2_interp)
    spiral_y2 = view_len * np.sin(theta2_interp)
    ax.plot(spiral_x2, spiral_y2, z2, 'm--', linewidth=1.5, alpha=0.7, label='V2 heading spiral')

    # Final heading arrow (solid)
    hx2 = [0, view_len * np.cos(theta2_interp[-1])]
    hy2 = [0, view_len * np.sin(theta2_interp[-1])]
    hz2 = [z2[-1], z2[-1]]
    ax.plot(hx2, hy2, hz2, 'r-', linewidth=4, zorder=5, alpha=0.9)

    # Add stop markers for Vehicle 2
    for stop in vehicle2_analysis['stops']:
        stop_idx = np.argmin(np.abs(vehicle2_results['time_acc'] - stop['time']))
        ax.scatter(x2[stop_idx], y2[stop_idx], z2[stop_idx], color='yellow', s=300, marker='*',
                  edgecolors='black', linewidths=1.5, zorder=10)

    # # Update cylinder to match pipeline bounds (0 to 16m)
    # theta_cyl2 = np.linspace(0, 2*np.pi, 30)
    # z_cyl2 = np.linspace(0, PIPELINE_LENGTH, 30)
    # Theta_cyl2, Z_cyl2 = np.meshgrid(theta_cyl2, z_cyl2)
    # X_cyl2 = PIPELINE_RADIUS * np.cos(Theta_cyl2)
    # Y_cyl2 = PIPELINE_RADIUS * np.sin(Theta_cyl2)
    # ax.plot_wireframe(X_cyl2, Y_cyl2, Z_cyl2, alpha=0.3, color='brown', linewidth=0.8, linestyle='-')

    ax.set_xlabel('X Position (m)', fontsize=11, labelpad=10)
    ax.set_ylabel('Y Position (m)', fontsize=11, labelpad=10)
    ax.set_zlabel('Height (m)', fontsize=11, labelpad=10)
    ax.set_title(f'3D Spatial Trajectory: Pipeline Inspection ({PIPELINE_LENGTH}m × Ø{2*PIPELINE_RADIUS}m (assumed for visualization))',
                fontsize=13, pad=20, fontweight='bold')

    # Create custom legend with star marker
    from matplotlib.lines import Line2D
    legend_elements = [
        Line2D([0], [0], color='b', linewidth=3, alpha=0.8, label='V1 (down from top)'),
        Line2D([0], [0], color='c', linewidth=1.5, linestyle='--', alpha=0.7, label='V1 heading spiral'),
        Line2D([0], [0], marker='o', color='w', markerfacecolor='green', markersize=10,
               markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='Start V1 (top)'),
        Line2D([0], [0], marker='X', color='w', markerfacecolor='blue', markersize=10,
               markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='End V1'),
        Line2D([0], [0], color='r', linewidth=3, alpha=0.8, label='V2 (up from bottom)'),
        Line2D([0], [0], color='m', linewidth=1.5, linestyle='--', alpha=0.7, label='V2 heading spiral'),
        Line2D([0], [0], marker='o', color='w', markerfacecolor='orange', markersize=10,
               markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='Start V2 (bottom)'),
        Line2D([0], [0], marker='X', color='w', markerfacecolor='red', markersize=10,
               markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='End V2'),
        Line2D([0], [0], marker='*', color='w', markerfacecolor='cyan', markersize=14,
               markeredgecolor='black', markeredgewidth=1, linestyle='None', label='V1 Stops'),
        Line2D([0], [0], marker='*', color='w', markerfacecolor='yellow', markersize=14,
               markeredgecolor='black', markeredgewidth=1, linestyle='None', label='V2 Stops')
    ]
    ax.legend(handles=legend_elements, loc='upper left', fontsize=9, framealpha=0.9,
             edgecolor='black', fancybox=False, shadow=False, ncol=1, labelspacing=0.8)

    ax.view_init(elev=20, azim=45)
    ax.grid(True, alpha=0.3)

    # Set z-axis range to include both vehicles within pipeline
    ax.set_xlim([-0.6, 0.6])
    ax.set_ylim([-0.6, 0.6])
    ax.set_zlim([-5, 16])  # Pipeline is 0-16m, allow some margin

    # Set aspect ratio to stretch z-axis
    ax.set_box_aspect([1, 1, 2])  # x:y:z = 1:1:2.5

    # Add reference planes at pipeline boundaries
    # XY plane at z=0 (BOTTOM of pipeline - where V2 starts)
    xx, yy = np.meshgrid([-0.6, 0.6], [-0.6, 0.6])
    zz = np.zeros_like(xx)
    ax.plot_surface(xx, yy, zz, alpha=0.2, color='green', edgecolor='none')
    ax.text(0.65, 0, 0, 'z=0 (bottom)', fontsize=9, color='darkgreen', fontweight='bold')

    # XY plane at z=16m (TOP of pipeline - where V1 starts)
    zz_top = np.ones_like(xx) * PIPELINE_LENGTH
    ax.plot_surface(xx, yy, zz_top, alpha=0.2, color='orange', edgecolor='none')
    ax.text(0.65, 0, PIPELINE_LENGTH, 'z=16m (top)', fontsize=9, color='darkorange', fontweight='bold')

    plt.tight_layout()
    filename = os.path.join(section_plots_dir, "trajectory_3d_combined.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    print(f"[FILE] Saved: {filename}")
    plt.close()

    # Create 1x2 grid with separate plots for each vehicle
    plot_trajectory_3d_separate(
        vehicle1_results, vehicle1_analysis, z1, theta1_interp,
        vehicle2_results, vehicle2_analysis, z2, theta2_interp,
        section_plots_dir, PIPELINE_RADIUS, PIPELINE_LENGTH
    )

    # Plot angular position and rate
    for results, analysis, name in vehicles:
        fig, axes = plt.subplots(2, 1, figsize=(12, 8))

        t_gyr = results['time_gyr']
        theta_z = results['angular_position_z']
        omega_z = results['angular_rate_z']

        theta_z_deg = np.degrees(theta_z)
        omega_z_deg = np.degrees(omega_z)

        axes[0].plot(t_gyr, theta_z_deg, 'm-', linewidth=1.5)
        axes[0].set_xlabel('Time (s)')
        axes[0].set_ylabel('Angular Position z (deg)')
        axes[0].set_title(f'{name} - Angular Position vs Time')
        axes[0].grid(True, alpha=0.3)

        axes[1].plot(t_gyr, omega_z_deg, 'c-', linewidth=1.5)
        axes[1].set_xlabel('Time (s)')
        axes[1].set_ylabel('Angular Rate z (deg/s)')
        axes[1].set_title(f'{name} - Angular Rate vs Time')
        axes[1].grid(True, alpha=0.3)
        axes[1].axhline(y=0, color='r', linestyle='--', alpha=0.5)

        plt.tight_layout()
        filename = os.path.join(section_plots_dir, f"{name.lower().replace(' ', '_')}_angular.png")
        plt.savefig(filename, dpi=300)
        print(f"[FILE] Saved: {filename}")
        plt.close()

    # Save analysis results to JSON
    results_data = {}
    for results, analysis, name in vehicles:
        vehicle_key = name.lower().replace(' ', '_')
        results_data[vehicle_key] = {
            "direction": analysis['direction'],
            "initial_position": float(analysis['initial_position']),
            "final_position": float(analysis['final_position']),
                "total_rotation_deg": analysis['total_rotation_deg'],
                "total_rotation_direction": analysis['total_rotation_direction'],
                "num_full_rotations": analysis['num_full_rotations'],
            "stops": [{"time": float(s['time']), "height": float(s['height'])} for s in analysis['stops']],
                "rotations_during_stops": [{
                "rotation_deg": float(r['rotation']),
                "direction": r['direction'],
                "height": float(r['stop_height']),
                    "full_360": bool(r['full_360'])
                } for r in analysis['rotations_during_stops']]
        }

    results_file = os.path.join(results_dir, "section_2_results.json")
    with open(results_file, 'w') as f:
        json.dump(results_data, f, indent=4)

        print(f"[FILE] Results saved to: {results_file}")
    print("[INFO] Section 2 completed\n")

if __name__ == "__main__":
    # If run as standalone script, need to load bias params from file
    SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
    PROJECT_ROOT = os.path.dirname(SCRIPT_DIR)
    DATA_DIR = os.path.join(PROJECT_ROOT, "data")
    PLOTS_DIR = os.path.join(PROJECT_ROOT, "plots")
    RESULTS_DIR = os.path.join(PROJECT_ROOT, "results")

    # Load bias parameters from JSON file (if running standalone)
    import json
    bias_file = os.path.join(RESULTS_DIR, "section_1_bias_parameters.json")
    if os.path.exists(bias_file):
        with open(bias_file, 'r') as f:
            data = json.load(f)
        acc_bias_params = data["accelerometer"]
        gyr_bias_params = data["gyroscope"]
        print(f"[FILE] Loaded bias parameters from: {bias_file}")
    else:
        print("[WARNING] Bias parameters file not found. Run Section 1 first or run main.py")
        exit(1)

    main(DATA_DIR, PLOTS_DIR, RESULTS_DIR, acc_bias_params, gyr_bias_params)
```

Section3.py:

```python
"""
Section 2: Trajectory Characterization Using IMU Data
Analyzes vehicle trajectories by integrating accelerometer and gyroscope data.
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import json

# Constants
GRAVITY = 9.805  # m/s^2

def integrate_forward_euler(t, values):
    """Integrate using Forward Euler method (matching class example style)"""
    n = len(t)
    integrated = np.zeros(n)  # Initialize array of zeros (same size as input)
    for i in range(1, n):
        # Forward Euler: integral[i] = integral[i-1] + f[i] * dt
        dt = t[i] - t[i-1]  # Calculate actual dt for each step (not constant!)
        integrated[i] = integrated[i-1] + values[i] * dt

    return integrated

def process_vehicle_data(acc_file, gyr_file, vehicle_name, acc_bias_params, gyr_bias_params):
    """Process IMU data for a single vehicle to compute trajectory"""

    # Load data
    acc_data = pd.read_csv(acc_file)
    gyr_data = pd.read_csv(gyr_file)
    print(f"[INFO] Loaded {len(acc_data)} accelerometer and {len(gyr_data)} gyroscope samples for {vehicle_name}")

    # Extract time and measurements
    t_acc = acc_data.iloc[:, 0].values
    acc_z = acc_data.iloc[:, 3].values

    t_gyr = gyr_data.iloc[:, 0].values
    gyr_z = gyr_data.iloc[:, 3].values

    # Remove bias from accelerometer (inline: b(t) = b0 + b_s * t)
    acc_z_corrected = acc_z - (acc_bias_params['z']['b0'] + acc_bias_params['z']['b_s'] * t_acc)

    # Remove gravity from z-axis
    acc_z_corrected = acc_z_corrected - GRAVITY

    # Remove bias from gyroscope (inline: b(t) = b0 + b_s * t)
    gyr_z_corrected = gyr_z - (gyr_bias_params['z']['b0'] + gyr_bias_params['z']['b_s'] * t_gyr)

    # Integrate z-axis acceleration to get velocity (Forward Euler)
    v_z = integrate_forward_euler(t_acc, acc_z_corrected)

    # Integrate velocity to get position (Forward Euler)
    p_z = integrate_forward_euler(t_acc, v_z)

    # Integrate z-axis angular rate to get angular position (Forward Euler)
    theta_z = integrate_forward_euler(t_gyr, gyr_z_corrected)

    print(f"[INFO] Trajectory computed for {vehicle_name}")
    return {
        'time_acc': t_acc,
        'time_gyr': t_gyr,
        'acceleration_z': acc_z_corrected,
        'velocity_z': v_z,
        'position_z': p_z,
        'angular_rate_z': gyr_z_corrected,
        'angular_position_z': theta_z
    }

def analyze_trajectory(results, vehicle_name):
    """Analyze trajectory characteristics: direction, stops, rotations"""

    p_z = results['position_z']
    v_z = results['velocity_z']
    theta_z = results['angular_position_z']
    t_acc = results['time_acc']
    t_gyr = results['time_gyr']

    # Determine motion direction
    final_position = p_z[-1]
    direction = "up" if final_position > p_z[0] else "down"
    print(f"[OUTPUT] {vehicle_name} is moving {direction} (final position: {final_position:.2f} m)")

    # Calculate TOTAL rotation from start to finish
    total_rotation_rad = theta_z[-1] - theta_z[0]
    total_rotation_deg = np.degrees(total_rotation_rad)
    num_full_rotations = total_rotation_deg / 360.0
    rotation_direction = "right (clockwise)" if total_rotation_deg < 0 else "left (counter-clockwise)"

    print(f"[OUTPUT] TOTAL rotation: {total_rotation_deg:.1f}° {rotation_direction}")
    print(f"[OUTPUT] Number of full 360° rotations: {abs(num_full_rotations):.2f}")

    # Find stopping points (velocity near zero)
    velocity_threshold = 0.1  # m/s
    stopping_indices = np.where(np.abs(v_z) < velocity_threshold)[0]  # np.where returns indices where condition is True

    # Identify distinct stopping periods
    stops = []
    if len(stopping_indices) > 0:
        stop_groups = []
        current_group = [stopping_indices[0]]

        for i in range(1, len(stopping_indices)):
            if stopping_indices[i] - stopping_indices[i-1] < 10:
                current_group.append(stopping_indices[i])
            else:
                stop_groups.append(current_group)
                current_group = [stopping_indices[i]]
        stop_groups.append(current_group)

        for group in stop_groups:
            if len(group) > 5:
                center_idx = group[len(group)//2]
                stop_time = t_acc[center_idx]
                stop_height = p_z[center_idx]
                stops.append({'time': stop_time, 'height': stop_height, 'indices': group})
                print(f"[OUTPUT] Stop detected at t={stop_time:.2f}s, height={stop_height:.2f}m")

    # Analyze rotations during stops
    rotations = []
    for stop in stops:
        stop_start_idx = stop['indices'][0]
        stop_end_idx = stop['indices'][-1]

        stop_time_start = t_acc[stop_start_idx]
        stop_time_end = t_acc[stop_end_idx]
        # np.argmin finds the index of the minimum value
        gyr_start_idx = np.argmin(np.abs(t_gyr - stop_time_start))  # Find closest gyro timestamp
        gyr_end_idx = np.argmin(np.abs(t_gyr - stop_time_end))

        if gyr_end_idx > gyr_start_idx:
            theta_change = theta_z[gyr_end_idx] - theta_z[gyr_start_idx]

            # Normalize to [-π, π] range
            while theta_change > np.pi:
                theta_change -= 2*np.pi
            while theta_change < -np.pi:
                theta_change += 2*np.pi

            rotation_deg = np.degrees(theta_change)

            if abs(rotation_deg) > 10:
                direction_rot = "left" if rotation_deg > 0 else "right"
                full_rotation = abs(rotation_deg) >= 350
                rotations.append({
                    'stop_height': stop['height'],
                    'rotation': rotation_deg,
                    'direction': direction_rot,
                    'full_360': full_rotation
                })
                print(f"[OUTPUT] Rotation: {rotation_deg:.1f}° {direction_rot} ({'full 360°' if full_rotation else 'partial'})")

    return {
        'direction': direction,
        'stops': stops,
        'rotations_during_stops': rotations,  # Rotation only during stop periods
        'total_rotation_deg': float(total_rotation_deg),  # TOTAL rotation throughout journey
        'total_rotation_direction': rotation_direction,
        'num_full_rotations': float(abs(num_full_rotations)),
        'final_position': final_position,
        'initial_position': p_z[0]
    }

def main(data_dir, plots_dir, results_dir, acc_bias_params, gyr_bias_params):
    """
    Run Section 2 trajectory analysis

    Parameters:
```

```python
    - data_dir: path to data directory
    - plots_dir: path to plots directory
    - results_dir: path to results directory
    - acc_bias_params: accelerometer bias parameters from Section 1
    - gyr_bias_params: gyroscope bias parameters from Section 1
    """

    print("[INFO] Running Section 2: Trajectory Characterization")

    # Process vehicle 1
    vehicle1_results = process_vehicle_data(
            os.path.join(data_dir, "secII_acc_1.csv"),
            os.path.join(data_dir, "secII_gyr_1.csv"),
        "Vehicle 1",
        acc_bias_params,
        gyr_bias_params
    )

    # Process vehicle 2
    vehicle2_results = process_vehicle_data(
            os.path.join(data_dir, "secII_acc_2.csv"),
            os.path.join(data_dir, "secII_gyr_2.csv"),
        "Vehicle 2",
        acc_bias_params,
        gyr_bias_params
    )

    # Analyze trajectories
    vehicle1_analysis = analyze_trajectory(vehicle1_results, "Vehicle 1")
    vehicle2_analysis = analyze_trajectory(vehicle2_results, "Vehicle 2")

    # Create plots
    section_plots_dir = os.path.join(plots_dir, "section_2")
    os.makedirs(section_plots_dir, exist_ok=True)
    os.makedirs(results_dir, exist_ok=True)

    vehicles = [
        (vehicle1_results, vehicle1_analysis, "Vehicle 1"),
        (vehicle2_results, vehicle2_analysis, "Vehicle 2")
    ]

    # Plot position, velocity, acceleration for each vehicle
    for results, analysis, name in vehicles:
        fig, axes = plt.subplots(3, 1, figsize=(12, 10))

        t = results['time_acc']
        p_z = results['position_z']
        v_z = results['velocity_z']
        a_z = results['acceleration_z']

        axes[0].plot(t, p_z, 'b-', linewidth=1.5)
        axes[0].set_xlabel('Time (s)')
        axes[0].set_ylabel('Position z (m)')
        axes[0].set_title(f'{name} - Position vs Time')
        axes[0].grid(True, alpha=0.3)

        axes[1].plot(t, v_z, 'g-', linewidth=1.5)
        axes[1].set_xlabel('Time (s)')
        axes[1].set_ylabel('Velocity z (m/s)')
        axes[1].set_title(f'{name} - Velocity vs Time')
        axes[1].grid(True, alpha=0.3)
        axes[1].axhline(y=0, color='r', linestyle='--', alpha=0.5)

        axes[2].plot(t, a_z, 'r-', linewidth=1.5)
        axes[2].set_xlabel('Time (s)')
        axes[2].set_ylabel('Acceleration z (m/s^2)')
        axes[2].set_title(f'{name} - Acceleration vs Time')
        axes[2].grid(True, alpha=0.3)
        axes[2].axhline(y=0, color='k', linestyle='--', alpha=0.5)

        plt.tight_layout()
        filename = os.path.join(section_plots_dir, f"{name.lower().replace(' ', '_')}_trajectory.png")
        plt.savefig(filename, dpi=300)
        print(f"[FILE] Saved: {filename}")
        plt.close()

    # Create 3D spatial visualization - vehicles at centerline rotating to scan walls
    from mpl_toolkits.mplot3d import Axes3D
    from matplotlib.lines import Line2D

    PIPELINE_RADIUS = 0.5   # m (assumed for viz)
    PIPELINE_LENGTH = 16.0  # m (given in project description)

    def plot_trajectory_3d_separate(v1_results, v1_analysis, z1, theta1,
                                     v2_results, v2_analysis, z2, theta2,
                                     plots_dir, pipe_r, pipe_len):
        """Create 1x2 grid with separate 3D plots for each vehicle."""
        fig = plt.figure(figsize=(18, 9))

        # Common plot setup function
        def setup_vehicle_plot(ax, z, theta, results, analysis, v_name, color, start_color, end_color, stop_color, spiral_color):
            # Draw pipeline cylinder
            theta_cyl = np.linspace(0, 2*np.pi, 30)
            z_cyl = np.linspace(0, pipe_len, 30)
            Theta_cyl, Z_cyl = np.meshgrid(theta_cyl, z_cyl)
            X_cyl = pipe_r * np.cos(Theta_cyl)
            Y_cyl = pipe_r * np.sin(Theta_cyl)
            ax.plot_wireframe(X_cyl, Y_cyl, Z_cyl, alpha=0.2, color='gray', linewidth=0.5, linestyle='--')

            # Vehicle at centerline
            x = np.zeros_like(z)
            y = np.zeros_like(z)

            # Plot trajectory
            ax.plot(x, y, z, color=color, linewidth=3, alpha=0.8, label=f'{v_name} path')
            ax.scatter(x[0], y[0], z[0], color=start_color, s=200, marker='o', edgecolors='black', linewidths=2, zorder=5)
            ax.scatter(x[-1], y[-1], z[-1], color=end_color, s=200, marker='X', edgecolors='black', linewidths=2, zorder=5)

            # Draw heading spiral
            view_len = 0.35
            skip = max(1, len(z) // 50)
            heading_x, heading_y, heading_z = [], [], []
            for i in range(0, len(z), skip):
                heading_x.extend([0, view_len * np.cos(theta[i]), np.nan])
                heading_y.extend([0, view_len * np.sin(theta[i]), np.nan])
                heading_z.extend([z[i], z[i], np.nan])
            ax.plot(heading_x, heading_y, heading_z, color=color, linestyle=':', linewidth=1, alpha=0.5)

            spiral_x = view_len * np.cos(theta)
            spiral_y = view_len * np.sin(theta)
            ax.plot(spiral_x, spiral_y, z, color=spiral_color, linestyle='--', linewidth=1.5, alpha=0.7)

            # Final heading arrow
            ax.plot([0, view_len * np.cos(theta[-1])], [0, view_len * np.sin(theta[-1])],
                   [z[-1], z[-1]], color=color, linewidth=4, zorder=5, alpha=0.9)

            # Stop markers
            for stop in analysis['stops']:
                stop_idx = np.argmin(np.abs(results['time_acc'] - stop['time']))
                ax.scatter(x[stop_idx], y[stop_idx], z[stop_idx], color=stop_color, s=300, marker='*',
                          edgecolors='black', linewidths=1.5, zorder=10)

            # FINAL POSITION INDICATOR - dotted line from z-axis to final position
            final_z = z[-1]
            # Horizontal dotted line from z-axis (at y=-0.6) to centerline (y=0) at final z height
            ax.plot([0, 0], [-0.6, 0], [final_z, final_z],
                   color=color, linestyle=':', linewidth=2, alpha=0.8)
            # Marker on z-axis edge
            ax.scatter([0], [-0.6], [final_z], color=color, s=100, marker='>', zorder=10)
            # Label showing final z value
            ax.text(0, -0.7, final_z, f'z={final_z:.1f}m', fontsize=10, fontweight='bold',
                   color=color, ha='center', va='center')

            # Reference planes
            xx, yy = np.meshgrid([-0.6, 0.6], [-0.6, 0.6])
            ax.plot_surface(xx, yy, np.zeros_like(xx), alpha=0.15, color='green', edgecolor='none')
            ax.text(0.65, 0, 0, 'z=0', fontsize=9, color='darkgreen', fontweight='bold')
            ax.plot_surface(xx, yy, np.ones_like(xx) * pipe_len, alpha=0.15, color='orange', edgecolor='none')
            ax.text(0.65, 0, pipe_len, f'z={pipe_len:.0f}m', fontsize=9, color='darkorange', fontweight='bold')

            ax.set_xlabel('X (m)', fontsize=10, labelpad=8)
            ax.set_ylabel('Y (m)', fontsize=10, labelpad=8)
            ax.set_zlabel('Height (m)', fontsize=10, labelpad=8)
            ax.set_xlim([-0.7, 0.7])
            ax.set_ylim([-0.7, 0.7])
            ax.set_zlim([-5, pipe_len + 2])
            ax.set_box_aspect([1, 1, 2])
            ax.view_init(elev=20, azim=45)
            ax.grid(True, alpha=0.3)

        # Vehicle 1 plot (left)
        ax1 = fig.add_subplot(121, projection='3d')
        setup_vehicle_plot(ax1, z1, theta1, v1_results, v1_analysis,
                          'V1', 'blue', 'green', 'blue', 'cyan', 'cyan')
        ax1.set_title('Vehicle 1 (Down from Top)\nStart: z=16m', fontsize=12, fontweight='bold', pad=15)

        # Vehicle 2 plot (right)
        ax2 = fig.add_subplot(122, projection='3d')
        setup_vehicle_plot(ax2, z2, theta2, v2_results, v2_analysis,
                          'V2', 'red', 'orange', 'red', 'yellow', 'magenta')
        ax2.set_title('Vehicle 2 (Up from Bottom)\nStart: z=0m', fontsize=12, fontweight='bold', pad=15)

        # Create shared legend for both plots
        legend_elements = [
            Line2D([0], [0], color='b', linewidth=3, alpha=0.8, label='V1 (down from top)'),
            Line2D([0], [0], color='c', linewidth=1.5, linestyle='--', alpha=0.7, label='V1 heading spiral'),
            Line2D([0], [0], marker='o', color='w', markerfacecolor='green', markersize=10,
                   markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='Start V1 (top)'),
            Line2D([0], [0], marker='X', color='w', markerfacecolor='blue', markersize=10,
                   markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='End V1'),
            Line2D([0], [0], color='r', linewidth=3, alpha=0.8, label='V2 (up from bottom)'),
            Line2D([0], [0], color='m', linewidth=1.5, linestyle='--', alpha=0.7, label='V2 heading spiral'),
            Line2D([0], [0], marker='o', color='w', markerfacecolor='orange', markersize=10,
                   markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='Start V2 (bottom)'),
            Line2D([0], [0], marker='X', color='w', markerfacecolor='red', markersize=10,
                   markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='End V2'),
            Line2D([0], [0], marker='*', color='w', markerfacecolor='cyan', markersize=14,
                   markeredgecolor='black', markeredgewidth=1, linestyle='None', label='V1 Stops'),
            Line2D([0], [0], marker='*', color='w', markerfacecolor='yellow', markersize=14,
                   markeredgecolor='black', markeredgewidth=1, linestyle='None', label='V2 Stops')
        ]
        fig.legend(handles=legend_elements, loc='upper center', ncol=5, fontsize=9,
                  framealpha=0.9, edgecolor='black', fancybox=False, bbox_to_anchor=(0.5, 0.02))

        plt.tight_layout(rect=[0, 0.08, 1, 1])  # Leave space at bottom for legend
        filename = os.path.join(plots_dir, "trajectory_3d.png")
        plt.savefig(filename, dpi=300, bbox_inches='tight')
        print(f"[FILE] Saved: {filename}")
        plt.close()

    # Vehicles stay at centerline (x=0, y=0), only move in z and rotate around z-axis
    # The rotation angle tells us which direction they're "looking" at the walls

    # Vehicle 1 - goes DOWN, so starts at TOP (z=16m)
    z1_raw = vehicle1_results['position_z']
    theta1_interp = np.interp(vehicle1_results['time_acc'], vehicle1_results['time_gyr'], vehicle1_results['angular_position_z'])
    x1 = np.zeros_like(z1_raw)  # Stay at centerline
    y1 = np.zeros_like(z1_raw)
    z1 = z1_raw + PIPELINE_LENGTH  # Offset to start at top (z=16m)

    # Vehicle 2 - goes UP, so starts at BOTTOM (z=0)
    z2_raw = vehicle2_results['position_z']
    theta2_interp = np.interp(vehicle2_results['time_acc'], vehicle2_results['time_gyr'], vehicle2_results['angular_position_z'])
    x2 = np.zeros_like(z2_raw)  # Stay at centerline
    y2 = np.zeros_like(z2_raw)
    z2 = z2_raw  # No offset, starts at bottom (z=0)

    print(f"[INFO] V1 (down from top): z=[{z1.min():.1f}, {z1.max():.1f}]m")
    print(f"[INFO] V2 (up from bottom): z=[{z2.min():.1f}, {z2.max():.1f}]m")

    # Create 3D plot
    fig = plt.figure(figsize=(14, 10))
    ax = fig.add_subplot(111, projection='3d')

    # Draw pipeline cylinder walls (dotted wireframe)
    theta_cyl = np.linspace(0, 2*np.pi, 30)
    z_cyl = np.linspace(0, 16, 30)
    Theta_cyl, Z_cyl = np.meshgrid(theta_cyl, z_cyl)
    X_cyl = PIPELINE_RADIUS * np.cos(Theta_cyl)
    Y_cyl = PIPELINE_RADIUS * np.sin(Theta_cyl)
    ax.plot_wireframe(X_cyl, Y_cyl, Z_cyl, alpha=0.2, color='gray', linewidth=0.5, linestyle='--')

    # Draw pipeline dimensions reference
    ax.plot([-PIPELINE_RADIUS, PIPELINE_RADIUS], [0, 0], [0, 0], 'k--', linewidth=1, alpha=0.5)
    ax.text(0, 0, 0.8, f'Ø{2*PIPELINE_RADIUS:.1f}m', fontsize=8, ha='center')
    ax.plot([0], [0], [-PIPELINE_LENGTH, 0], 'k-', linewidth=2, alpha=0.4)
    ax.text(0.7, 0, -PIPELINE_LENGTH/2, f'{PIPELINE_LENGTH}m', fontsize=8, ha='left', rotation=90)

    # Plot Vehicle 1 trajectory (blue - centerline, going down from TOP)
    ax.plot(x1, y1, z1, 'b-', linewidth=3, alpha=0.8, label='Vehicle 1 path (down from top)')
    ax.scatter(x1[0], y1[0], z1[0], color='green', s=200, marker='o', edgecolors='black', linewidths=2, label='Start V1 (top)', zorder=5)
    ax.scatter(x1[-1], y1[-1], z1[-1], color='blue', s=200, marker='X', edgecolors='black', linewidths=2, label='End V1', zorder=5)

    # Draw rotation spiral for V1 - dotted lines showing heading at intervals
    view_len = 0.35
    skip_v1 = max(1, len(z1) // 50)  # Show ~50 heading lines
    heading_x1 = []
    heading_y1 = []
    heading_z1 = []
    for i in range(0, len(z1), skip_v1):
        # Line from center to wall showing heading direction
        heading_x1.extend([0, view_len * np.cos(theta1_interp[i]), np.nan])
        heading_y1.extend([0, view_len * np.sin(theta1_interp[i]), np.nan])
        heading_z1.extend([z1[i], z1[i], np.nan])
    ax.plot(heading_x1, heading_y1, heading_z1, 'b:', linewidth=1, alpha=0.5)

    # Draw spiral connecting the heading tips (shows rotation pattern)
    spiral_x1 = view_len * np.cos(theta1_interp)
    spiral_y1 = view_len * np.sin(theta1_interp)
    ax.plot(spiral_x1, spiral_y1, z1, 'c--', linewidth=1.5, alpha=0.7, label='V1 heading spiral')

    # Final heading arrow (solid)
    hx1 = [0, view_len * np.cos(theta1_interp[-1])]
    hy1 = [0, view_len * np.sin(theta1_interp[-1])]
    hz1 = [z1[-1], z1[-1]]
    ax.plot(hx1, hy1, hz1, 'b-', linewidth=4, zorder=5, alpha=0.9)

    # Add stop markers for Vehicle 1
    for stop in vehicle1_analysis['stops']:
        stop_idx = np.argmin(np.abs(vehicle1_results['time_acc'] - stop['time']))
        ax.scatter(x1[stop_idx], y1[stop_idx], z1[stop_idx], color='cyan', s=300, marker='*',
                  edgecolors='black', linewidths=1.5, zorder=10)

    # Plot Vehicle 2 trajectory (red - centerline, going up from BOTTOM)
    ax.plot(x2, y2, z2, 'r-', linewidth=3, alpha=0.8, label='Vehicle 2 path (up from bottom)')
    ax.scatter(x2[0], y2[0], z2[0], color='orange', s=200, marker='o', edgecolors='black', linewidths=2, label='Start V2 (bottom)', zorder=5)
    ax.scatter(x2[-1], y2[-1], z2[-1], color='red', s=200, marker='X', edgecolors='black', linewidths=2, label='End V2', zorder=5)

    # Draw rotation spiral for V2 - dotted lines showing heading at intervals
    skip_v2 = max(1, len(z2) // 50)
    heading_x2 = []
    heading_y2 = []
    heading_z2 = []
    for i in range(0, len(z2), skip_v2):
        heading_x2.extend([0, view_len * np.cos(theta2_interp[i]), np.nan])
        heading_y2.extend([0, view_len * np.sin(theta2_interp[i]), np.nan])
        heading_z2.extend([z2[i], z2[i], np.nan])
    ax.plot(heading_x2, heading_y2, heading_z2, 'r:', linewidth=1, alpha=0.5)

    # Draw spiral connecting the heading tips
    spiral_x2 = view_len * np.cos(theta2_interp)
    spiral_y2 = view_len * np.sin(theta2_interp)
    ax.plot(spiral_x2, spiral_y2, z2, 'm--', linewidth=1.5, alpha=0.7, label='V2 heading spiral')

    # Final heading arrow (solid)
    hx2 = [0, view_len * np.cos(theta2_interp[-1])]
    hy2 = [0, view_len * np.sin(theta2_interp[-1])]
    hz2 = [z2[-1], z2[-1]]
    ax.plot(hx2, hy2, hz2, 'r-', linewidth=4, zorder=5, alpha=0.9)

    # Add stop markers for Vehicle 2
    for stop in vehicle2_analysis['stops']:
        stop_idx = np.argmin(np.abs(vehicle2_results['time_acc'] - stop['time']))
        ax.scatter(x2[stop_idx], y2[stop_idx], z2[stop_idx], color='yellow', s=300, marker='*',
                  edgecolors='black', linewidths=1.5, zorder=10)

    # # Update cylinder to match pipeline bounds (0 to 16m)
    # theta_cyl2 = np.linspace(0, 2*np.pi, 30)
    # z_cyl2 = np.linspace(0, PIPELINE_LENGTH, 30)
    # Theta_cyl2, Z_cyl2 = np.meshgrid(theta_cyl2, z_cyl2)
    # X_cyl2 = PIPELINE_RADIUS * np.cos(Theta_cyl2)
    # Y_cyl2 = PIPELINE_RADIUS * np.sin(Theta_cyl2)
    # ax.plot_wireframe(X_cyl2, Y_cyl2, Z_cyl2, alpha=0.3, color='brown', linewidth=0.8, linestyle='-')

    ax.set_xlabel('X Position (m)', fontsize=11, labelpad=10)
    ax.set_ylabel('Y Position (m)', fontsize=11, labelpad=10)
    ax.set_zlabel('Height (m)', fontsize=11, labelpad=10)
    ax.set_title(f'3D Spatial Trajectory: Pipeline Inspection ({PIPELINE_LENGTH}m × Ø{2*PIPELINE_RADIUS}m (assumed for visualization))',
                fontsize=13, pad=20, fontweight='bold')

    # Create custom legend with star marker
    from matplotlib.lines import Line2D
    legend_elements = [
        Line2D([0], [0], color='b', linewidth=3, alpha=0.8, label='V1 (down from top)'),
        Line2D([0], [0], color='c', linewidth=1.5, linestyle='--', alpha=0.7, label='V1 heading spiral'),
        Line2D([0], [0], marker='o', color='w', markerfacecolor='green', markersize=10,
               markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='Start V1 (top)'),
        Line2D([0], [0], marker='X', color='w', markerfacecolor='blue', markersize=10,
               markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='End V1'),
        Line2D([0], [0], color='r', linewidth=3, alpha=0.8, label='V2 (up from bottom)'),
        Line2D([0], [0], color='m', linewidth=1.5, linestyle='--', alpha=0.7, label='V2 heading spiral'),
        Line2D([0], [0], marker='o', color='w', markerfacecolor='orange', markersize=10,
               markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='Start V2 (bottom)'),
        Line2D([0], [0], marker='X', color='w', markerfacecolor='red', markersize=10,
               markeredgecolor='black', markeredgewidth=1.5, linestyle='None', label='End V2'),
        Line2D([0], [0], marker='*', color='w', markerfacecolor='cyan', markersize=14,
               markeredgecolor='black', markeredgewidth=1, linestyle='None', label='V1 Stops'),
        Line2D([0], [0], marker='*', color='w', markerfacecolor='yellow', markersize=14,
               markeredgecolor='black', markeredgewidth=1, linestyle='None', label='V2 Stops')
    ]
    ax.legend(handles=legend_elements, loc='upper left', fontsize=9, framealpha=0.9,
             edgecolor='black', fancybox=False, shadow=False, ncol=1, labelspacing=0.8)

    ax.view_init(elev=20, azim=45)
    ax.grid(True, alpha=0.3)

    # Set z-axis range to include both vehicles within pipeline
    ax.set_xlim([-0.6, 0.6])
    ax.set_ylim([-0.6, 0.6])
    ax.set_zlim([-5, 16])  # Pipeline is 0-16m, allow some margin

    # Set aspect ratio to stretch z-axis
    ax.set_box_aspect([1, 1, 2])  # x:y:z = 1:1:2.5

    # Add reference planes at pipeline boundaries
    # XY plane at z=0 (BOTTOM of pipeline - where V2 starts)
    xx, yy = np.meshgrid([-0.6, 0.6], [-0.6, 0.6])
    zz = np.zeros_like(xx)
    ax.plot_surface(xx, yy, zz, alpha=0.2, color='green', edgecolor='none')
    ax.text(0.65, 0, 0, 'z=0 (bottom)', fontsize=9, color='darkgreen', fontweight='bold')

    # XY plane at z=16m (TOP of pipeline - where V1 starts)
    zz_top = np.ones_like(xx) * PIPELINE_LENGTH
    ax.plot_surface(xx, yy, zz_top, alpha=0.2, color='orange', edgecolor='none')
    ax.text(0.65, 0, PIPELINE_LENGTH, 'z=16m (top)', fontsize=9, color='darkorange', fontweight='bold')

    plt.tight_layout()
    filename = os.path.join(section_plots_dir, "trajectory_3d_combined.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    print(f"[FILE] Saved: {filename}")
    plt.close()

    # Create 1x2 grid with separate plots for each vehicle
    plot_trajectory_3d_separate(
        vehicle1_results, vehicle1_analysis, z1, theta1_interp,
        vehicle2_results, vehicle2_analysis, z2, theta2_interp,
        section_plots_dir, PIPELINE_RADIUS, PIPELINE_LENGTH
    )

    # Plot angular position and rate
    for results, analysis, name in vehicles:
        fig, axes = plt.subplots(2, 1, figsize=(12, 8))

        t_gyr = results['time_gyr']
        theta_z = results['angular_position_z']
        omega_z = results['angular_rate_z']

        theta_z_deg = np.degrees(theta_z)
        omega_z_deg = np.degrees(omega_z)

        axes[0].plot(t_gyr, theta_z_deg, 'm-', linewidth=1.5)
        axes[0].set_xlabel('Time (s)')
        axes[0].set_ylabel('Angular Position z (deg)')
        axes[0].set_title(f'{name} - Angular Position vs Time')
        axes[0].grid(True, alpha=0.3)

        axes[1].plot(t_gyr, omega_z_deg, 'c-', linewidth=1.5)
        axes[1].set_xlabel('Time (s)')
        axes[1].set_ylabel('Angular Rate z (deg/s)')
        axes[1].set_title(f'{name} - Angular Rate vs Time')
        axes[1].grid(True, alpha=0.3)
        axes[1].axhline(y=0, color='r', linestyle='--', alpha=0.5)

        plt.tight_layout()
        filename = os.path.join(section_plots_dir, f"{name.lower().replace(' ', '_')}_angular.png")
        plt.savefig(filename, dpi=300)
        print(f"[FILE] Saved: {filename}")
        plt.close()

    # Save analysis results to JSON
    results_data = {}
    for results, analysis, name in vehicles:
        vehicle_key = name.lower().replace(' ', '_')
        results_data[vehicle_key] = {
            "direction": analysis['direction'],
            "initial_position": float(analysis['initial_position']),
            "final_position": float(analysis['final_position']),
                "total_rotation_deg": analysis['total_rotation_deg'],
                "total_rotation_direction": analysis['total_rotation_direction'],
                "num_full_rotations": analysis['num_full_rotations'],
            "stops": [{"time": float(s['time']), "height": float(s['height'])} for s in analysis['stops']],
                "rotations_during_stops": [{
                "rotation_deg": float(r['rotation']),
                "direction": r['direction'],
                "height": float(r['stop_height']),
                    "full_360": bool(r['full_360'])
                } for r in analysis['rotations_during_stops']]
        }

    results_file = os.path.join(results_dir, "section_2_results.json")
    with open(results_file, 'w') as f:
        json.dump(results_data, f, indent=4)

        print(f"[FILE] Results saved to: {results_file}")
    print("[INFO] Section 2 completed\n")

if __name__ == "__main__":
    # If run as standalone script, need to load bias params from file
    SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
    PROJECT_ROOT = os.path.dirname(SCRIPT_DIR)
    DATA_DIR = os.path.join(PROJECT_ROOT, "data")
    PLOTS_DIR = os.path.join(PROJECT_ROOT, "plots")
    RESULTS_DIR = os.path.join(PROJECT_ROOT, "results")

    # Load bias parameters from JSON file (if running standalone)
    import json
    bias_file = os.path.join(RESULTS_DIR, "section_1_bias_parameters.json")
    if os.path.exists(bias_file):
        with open(bias_file, 'r') as f:
            data = json.load(f)
        acc_bias_params = data["accelerometer"]
        gyr_bias_params = data["gyroscope"]
        print(f"[FILE] Loaded bias parameters from: {bias_file}")
    else:
        print("[WARNING] Bias parameters file not found. Run Section 1 first or run main.py")
        exit(1)

    main(DATA_DIR, PLOTS_DIR, RESULTS_DIR, acc_bias_params, gyr_bias_params)
```