

Automated Post-Flight Data and Signal Analysis in Flight Test Instrumentation Applications

Safak Tambova¹, Michael Seeman¹, Osman Birkan Ozseven², Dr. Florian M. Heckmeier³

¹ Depart. of Test Instrumentation, eRC-System GmbH, Ottobrunn, Germany

² Depart of Flight Test Instrumentation System Design, Turkish Aerospace (TA), Istanbul, Turkey

³ Depart of Aerodynamics and Loads, eRC-System GmbH, Ottobrunn, Germany
tambova@erc-system.com

Abstract

Aircraft development and certification depend on flight testing and data collection from sensor measurements. The efficient processing of this large data volume remains essential for analysis purposes while it verifies framework reliability and ensures flight safety. The use of conventional data management methods proves to be time-consuming and lack of smart features, thus demonstrating why reliable automated frameworks should be implemented. A new data analysis framework is demonstrated within this article, which is intended to simplify instrumentation data evaluation from flight tests. The flight-testing platform supports multiple sensor types for aircraft testing, including accelerometers, strain gauges, voltage-current sensors, thermocouples, and flowmeters. The framework maintains efficient dataset management by processing data for both parsing operations and mathematical calculations and by processing Packet Capture (PCAP) files incrementally to decrease memory requirements. The proposed analysis framework ensures data precision through digital signal processing techniques, in the time as well as frequency domains, such as the treatment of noisy data with filters. Furthermore, automated validation and error analysis are introduced as an advanced feature. Finally, the results are presented in nicely structured automated reports. In addition, various analysis verbosity levels are presented, with a "Full" mode for complete sensor data access and a "Quicklook" mode for fast data previews at lower telemetry rates and customized modes for individual sensor investigations. By automating intricate data handling tasks, from initial parsing and calibration to advanced signal analysis and rigorous error checking, this framework significantly enhances the efficiency, accuracy, and reliability of post-flight data interpretation. It equips engineers with a powerful tool to accelerate the flight test evaluation lifecycle, ultimately contributing to improved aircraft safety and more effective framework development.

Key words: Automated Data Analysis, Automated Data Post Processing, Flight Test Instrumentation, Sensor Data Analysis, Signal Processing.

Introduction

Flight test instrumentation is an essential part of the aircraft development process, as it verifies and validates the performance and integrity of aircraft systems. The complexity and quantity of data recorded during flight testing have therefore increased with every new generation of aircraft. For instance, sensors like accelerometers, strain gauges, voltage, current, temperature sensors, and flowmeters are effectively applied on the aircraft in order to measure significant data points. Requisite post-processing of this data is essential for evaluating the performance of aircraft frameworks, diagnosing possible problems, and supporting both real-time progress and decision-making in flight test evaluations [1].

In this article, an efficient framework for managing large-scale sensor data and automating critical operations, including the parsing of Instrumentation Engineering Network Architecture (IENA) packets, sensor calibration, and signal analysis, is presented [2]. To optimize memory usage, especially with large datasets, it employs incremental parsing and parallel processing strategies. The framework is designed with operational flexibility, offering multiple modes: a “Quicklook Mode” for rapid data validation, often using a subset of data or reduced sample rates, and a “Full Mode” for comprehensive analysis with full data resolution. Additionally, sensor-specific analytical modes enable targeted evaluations based on the type of sensor being analyzed. The framework leverages JSON (JavaScript Object Notation) for managing key processing parameters, such as calibration coefficients and sensor-specific thresholds, in addition to XidML (eXtensible Instrumentation Description Markup Language) configurations, which enhances the framework's modularity and adaptability [3].

Existing data processing tools often struggle with the large and heterogeneous datasets generated during flight tests. They frequently lack the requisite flexibility and computational speed for modern analytical demands, and critically, often do not possess comprehensive error and warning mechanisms to identify data quality issues. The framework presented here offers a scalable and automated solution designed to achieve high processing throughput while integrating robust error detection, warning, and calibration mechanisms. These capabilities provide users with critical insights into potential data quality issues such as timing discrepancies, miscalibrations, or EMI, thereby improving the accuracy and reliability of post-flight assessments [4].

In addition to identifying anomalies within the sensor data, the framework performs crucial internal consistency checks to validate the sufficiency and integrity of inputs required for its analytical functions. This pre-analysis validation step is critical in preventing incomplete or erroneous data processing. It ensures that all essential parameters, such as sensor type, calibration coefficients, unit definitions, and operational range limits, are present and correctly specified in the configuration files before detailed analysis commences. If any of these mandatory parameters are missing, the framework flags a critical error and halts processing for the affected data stream [5]. Consequently, this approach mitigates the risk of generating misleading outcomes and promotes more dependable and accurate flight test evaluations.

When examining existing work in the literature, flight test data management has become an increasingly demanding field driven by the growing complexity of modern aircraft systems. Compliance with standards such as IRIG-106, combined with the growing complexity of modern aircraft, has made flight test data management increasingly demanding [1, 2]. In response, a number of tools have employed open formats including JSON and XidML among them to improve interoperability and reduce integration overhead [3, 7]. Rapid iteration in aircraft design has pushed test engineers to seek more adaptable solutions for data tracking and visualization [4, 5, 19]. Beyond volume and source diversity [23, 27, 30], automated fault detection in telemetry streams has proven particularly resistant to straightforward solutions [8, 10, 12, 13, 16, 22]. Prior work has addressed this through machine learning and signal processing approaches [6, 11, 14, 17, 18, 25, 26]. This paper takes a different path, presenting a modular analysis framework grounded in established software design patterns. The framework automatically processes and validates data from hundreds of sensors, with correctness verified at each stage [9, 20, 21, 24, 28, 29, 31].

The article follows this organization: Chapter I of this paper examines the motivation for post-flight sensor data analysis and identifies the technical challenges that make this process complex, along with the specific objectives this study addresses. Chapter II introduces the JSON-based sensor metadata structure at the core of the framework, then covers each functional component in sequence: the parsing pipeline, CSV generation, error detection, analysis routines, report and log output, missing parameter checks, and the parallel processing

approach used to reduce overall processing time. Chapter III presents the experimental results obtained from real flight test data, including a performance evaluation of the framework under operational conditions. Chapter IV evaluates the framework's capabilities against its current limitations. Chapter V summarizes the contributions of this work and identifies real-time data processing and onboard diagnostic integration as the primary directions for future development.

Theory

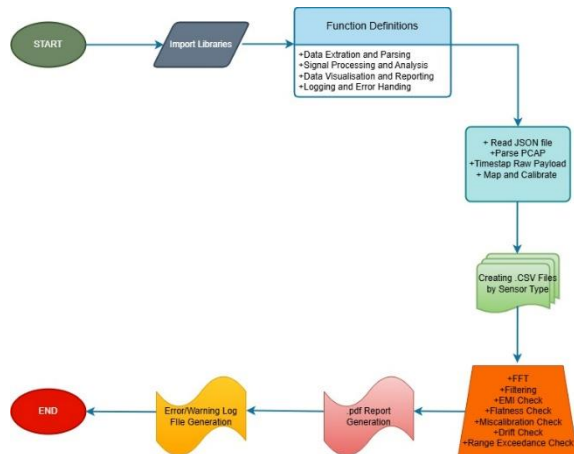


Fig. 1. Flowchart of the framework.

The automated post-flight data and signal analysis process appears in Fig. 1. The workflow starts by importing necessary libraries and establishing functions for data parsing, signal processing, and reporting. The algorithm starts by reading JSON and PCAP files. Then, it parses payload from PCAP files and applies timestamping and sensor data calibration. The system produces CSV files based on sensor types before executing multiple analysis steps, which include FFT analysis, filtering, EMI detection, flatness evaluation, drift assessment and range-exceedance verification. The system produces a PDF report while it creates an error or warning log file before finishing the process [6].

One of the cornerstones of this architecture is the usage of JSON formatting. The proposed data structure's possible use case is to create a flexible data interchange format for data parsing, calibration, and sensor configuration; JSON is a lightweight data interchange format. Apart from the configuration in the XidML file, we used a JSON structure to consolidate the parameters necessary for the calculations in the main code in one place. The details specified from the XidML file have been written out as JSON through another helper script. The system receives the JSON format, which contains additional relevant information that extends

beyond what XidML provides. In the current version, the analysis framework is able to support different sensors, which include accelerometers, temperature sensors, strain gauges, current transducers, voltage transducers, and flowmeters.

The JSON configuration is key, as it serves as the metadata source used for parsing sensor data, including sampling rate, units, calibration parameters, and other relevant details. This modular design enables the framework to accommodate different types of sensor arrays without requiring reconfiguration of the core structure, making it relatively flexible for various kinds of flight tests [7].

Fig. 2 demonstrates a basic system for sensor data integration through a JSON configuration file. The system receives data from various sensors, which are transformed into standardized JSON format through a configuration file. The configuration file serves as a mapping tool which converts different sensor output formats into a single standardized format.

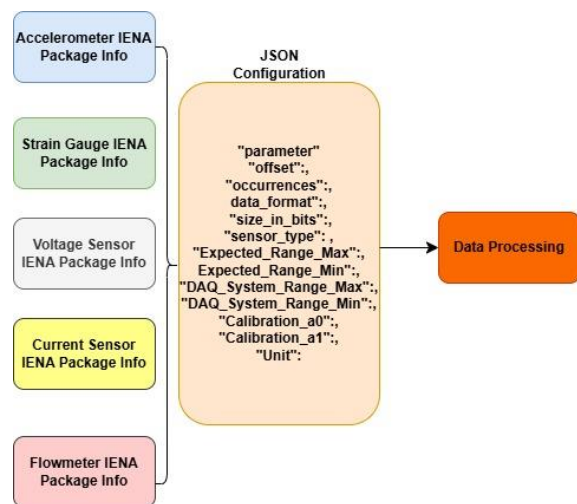


Fig. 2. Sensor data integration via JSON configuration.

Parsing

The data parsing and calibration process begins with the *parse_and_calibrate_interleaved_payload* function, which processes raw IENA packets. Its primary responsibility is to extract raw sensor measurements, accurately timestamp the data, and apply calibration [8]. Once it is finished, it stores the results in a structured dictionary. Any missing calibration data or sensor type triggers an error to ensure data integrity.

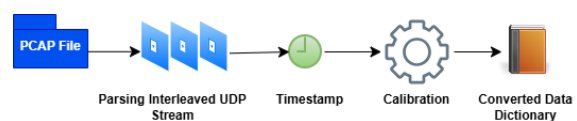


Fig. 3. Sensor data parsing and calibration workflow.

BEGIN Incremental_Parse_PCAP

INPUT: file_path, config, current_year = 1234,
filter_port = 5678, filter_mode = 'exclude'
OUTPUT: yields tuple(DataFrame, sample_rates_dict)

```
all_packet_data ← []
sample_rates_dict ← {}
```

```
FOR EACH packet IN read_pcap(file_path)
  IF filter_mode = 'exclude' AND packet.port =
    filter_port THEN
    CONTINUE
  ELSE IF filter_mode = 'include' AND packet.port ≠
    filter_port THEN
    CONTINUE
  END IF
```

```
packet_data ←
Parse_And_Calibrate_Interleaved_Payload(packet.payload
, config, current_year)
all_packet_data.APPEND(packet_data)
```

```
FOR EACH sensor_parameter IN
packet_data.sensors
  sample_rates_dict[sensor_parameter] ←
packet_data.sensors[sensor_parameter].rate
END FOR
```

```
YIELD convert_to_dataframe(all_packet_data),
sample_rates_dict
END FOR
```

END Incremental_Parse_PCAP

CSV Creation Function

The `create_csv_for_sensor_type_partition` function is responsible for transforming a temporary, pickled pandas data frame, which contains data for a specific sensor type, into a structured Comma-Separated Values (CSV) file. This CSV file is enriched with metadata and formatted to facilitate subsequent data analysis and interpretation.

Fig. 4 shows how CSV files are created through JSON metadata guidance. The pipeline starts with a pickled data frame file, which gets decoded and organized based on the configuration metadata. The CSV creation function performs three essential operations, which include data extraction, organization, and header generation from metadata [9]. As explained in the parsing chapter, any missing metadata in the header triggers an error to ensure data integrity. The process generates an analysis-ready CSV file output while it removes all intermediate files to achieve better performance.

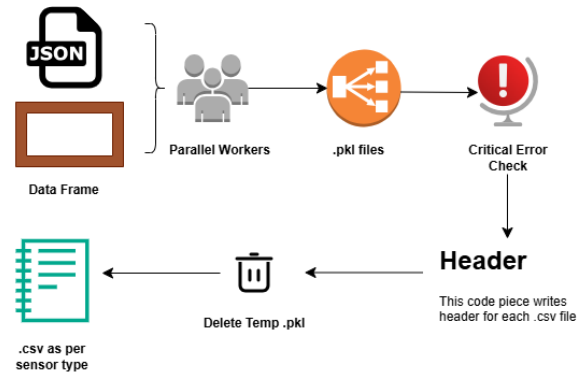


Fig. 4. CSV generation using JSON metadata.

Erroneous and Analysis Functions

Using analytical techniques such as peak-to-peak variation, mean, and drift calculation, the framework provides key information on sensor performance and reliability [10]. This approach serves to detect anomalies in the data, preserve data integrity, and verify the correctness of the analyzes performed on flight tests.

Flatness Check

To ensure sensor data reliability, the framework includes a flatness check that monitors the variability within a defined time window. This is done by calculating the peak-to-peak (P2P) variation, which measures the difference between the maximum and minimum values of a sensor signal. If the variation is below a predefined threshold (e.g., 0.5), the data is considered “flat”.

The `check_flatness` function encapsulates this logic and flags such signals, preventing silent errors from going unnoticed during analysis. Detecting flat data is crucial for maintaining the credibility of sensor outputs, which can indicate potential issues such as loose cable connections, frozen sensor outputs, or other forms of sensor malfunction.

The P2P variation is calculated as:

$$P2P = V_{max} - V_{min} \quad (1)$$

Where:

V_{max} is the maximum recorded value

V_{min} is the minimum recorded value

Fig. 5 illustrates the automated analysis results of the flowmeter sensor, highlighting a flat detection condition with nearly constant data values over time.

Sensor Analysis: FLM_VR40

Errors:

- DAQ System Range Exceedance: False
- Flat Detection: True
- Time Errors: False

Warnings:

- Drift Warning: False
- Calibration Warning: False
- Expected Range Exceedance: False

Analysis Results:

- All Values in Unit: lpm
- Max Value: 0.001
- Min Value: -0.003
- Calibration Start At Value: 0.001
- Calibration End At Value: -0.001
- Actual Delta Of Drift: 0.002
- Calibration A0: -17.680
- Calibration A1: 35.359
- Time Errors Found in CSV Line: []

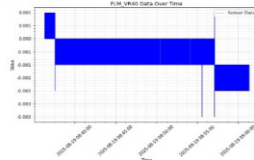
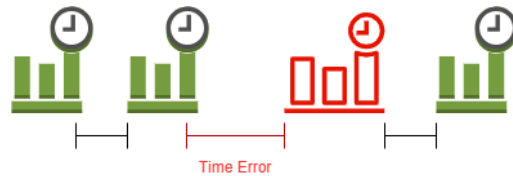


Fig. 5. Flat signal analysis.

Timestamp Delta Validation

The `check_time_deltas` function validates the consistency of time intervals in recorded data. The process takes a pandas series containing timestamps and a defined sample rate as input. It calculates the time deltas between consecutive samples and compares them to the expected time delta. If the comparison is bigger than $1 \mu\text{s}$, it is identified as an error, which may indicate issues such as packet loss, synchronization drift, or irregular sampling [11].

Fig. 6 shows a time delta error situation, which exposes packet loss or irregular data transmission through unexpected gaps between consecutive samples during sensor analysis.



Sensor Analysis: CDH_E411

Errors:

- DAQ System Range Exceedance: False
- Flat Detection: False
- Time Errors: True

Warnings:

- Drift Warning: False
- Calibration Warning: False
- Expected Range Exceedance: False

Analysis Results:

- All Values in Unit: A
- Max Value: 3.370
- Min Value: 1.585
- Calibration Start At Value: 1.941
- Calibration End At Value: 1.899
- Actual Delta Of Drift: 0.042
- Calibration A0: -497.370
- Calibration A1: 1000.000
- Time Errors Found in CSV Line: [762191]

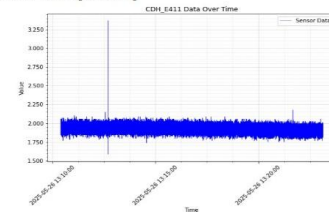


Fig. 6. Time delta checks between data points.

Range Exceedance

One of the key integrity checks implemented in the framework is range exceedance detection, which verifies whether sensor data remains within predefined operational boundaries. The theoretical foundation is:

If Value > MaxThreshold or
If Value < MinThreshold
Then Error (2)

Where:

MinThreshold and *MaxThreshold* are thresholds defined based on expected sensor behaviour. These checks help detect possible sensor malfunctions due to signal saturation, degradation, or environmental interference.

The `check_expected_range_exceedance` function extends this logic by comparing sensor readings with expected limits configured via JSON, and flags data exceeding the valid range. Expected range exceedance typically signals an aircraft-level anomaly

The `check_daq_framework_range_exceedance` function checks whether the readings violate broader data acquisition framework (DAQ) constraints. DAQ range exceedance often implies DAQ level anomaly, such as analogue

digital converter hardware saturation or failure in the test setup itself [12].

Fig. 7 shows the strain gauge sensor analysis, which displayed a major range exceedance occurrence.

Sensor Analysis: SGL_LR01

Errors:

- DAQ System Range Exceedance: False
- Flat Detection: False
- Time Errors: False

Warnings:

- Drift Warning: True
- Calibration Warning: True
- Expected Range Exceedance: True

Analysis Results:

- All Values in Unit: Micrometer/meter
- Max Value: 18775.000
- Min Value: 14815.615
- Calibration Start At Value: 15155.036
- Calibration End At Value: 18733.105
- Actual Delta Of Drift: 3578.069
- Calibration A0: -18836.000
- Calibration A1: 37611.000
- Time Errors Found in CSV Line: []

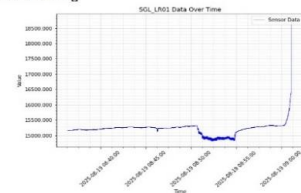


Fig. 7. Range exceedance checks for sensor malfunction detection.

Accelerometer Sensor Analysis

The *analyze_icp_sensor* function analyses the data from the accelerometer sensor and provides drift detection, dominant frequency analysis, and EMI alarms. Accelerometer sensor data is utilized for detecting electromagnetic interference (EMI) since accelerometers are more susceptible to EMI disturbances compared to other sensor types. The produced plots assist engineers in analysing the sensor data in both time and frequency domains.

Fig. 8 shows accelerometer sensor data, indicating an unexpected peak that triggers an EMI warning. The FFT spectrum displays a broad frequency distribution, yet the low-pass and high-pass filtered signals demonstrate how the transient signal interacts with the underlying oscillations. The disturbance pattern indicates that the sensor line received external electrical noise, which caused the disturbance rather than mechanical vibration.

Sensor Analysis: AA0_VR11_Y

Errors:

- DAQ System Range Exceedance: False
- Flat Detection: False
- Time Errors: False

Warnings:

- EMI Warning: True
- Drift Warning: False
- Calibration Warning: False
- Expected Range Exceedance: False

Analysis Results:

- All Values in Unit: g
- Max Value: 7.078
- Min Value: -4.737
- Calibration Start At Value: -0.048
- Calibration End At Value: -0.045
- Actual Delta Of Drift: -0.003
- Dominant Frequency: ['0.006', '0.004', '0.011', '0.013', '0.008', '0.019', '0.017', '0.009', '0.021', '0.026']
- Calibration A0: -49.650
- Calibration A1: 99.310
- Time Errors Found in CSV Line: []

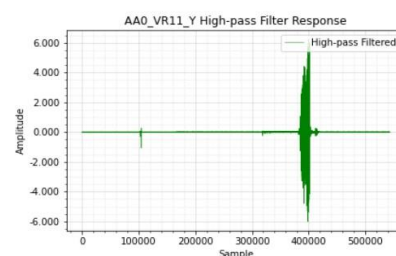
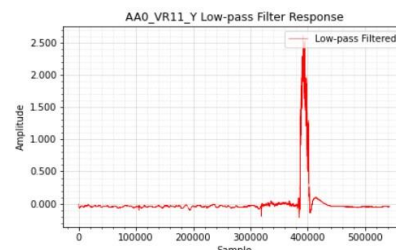
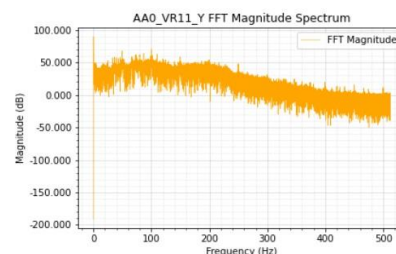
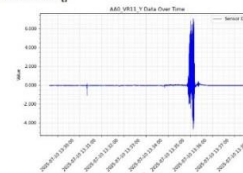


Fig. 8. Accelerometer sensor analysis showing EMI-induced transient response and corresponding frequency spectrum.

Calibration and Drift Detection

In order to assess the long-term reliability of accelerometer sensor readings, the framework evaluates calibration drift by comparing the average sensor values at the beginning and end of the data set. These segments represent the initial and final calibration states. The calibration values are calculated as follows:

$$\text{Calibration Start} = \frac{1}{N} \sum_{i=1}^N V_i^{\text{start}} \quad (3)$$

$$\text{Calibration End} = \frac{1}{N} \sum_{i=1}^N V_i^{\text{end}} \quad (4)$$

Where:

V_i^{start} and V_i^{end} are the sensor values at the start and end of the dataset, respectively.

N is the number of points considered in each segment. This comparison highlights whether the sensor's baseline has shifted over time.

The calibration comparison is then performed by checking if the mean of the first five seconds of sensor values exceeds the miscalibration threshold:

$$\text{if} | \text{Calibration}_{\text{start}} | > \text{MiscalibrationThreshold} \quad (5)$$

This check shows how much the DC shift is during the first five seconds. A mean value higher than the predefined miscalibration threshold indicates that the sensor is out of calibration, resulting in inaccurate data measurement.

To detect possible drift, the framework computes the absolute difference between these two values:

$$\Delta_{\text{drift}} = | \text{Calibration End} - \text{Calibration Start} |$$

$$\text{if} | \Delta_{\text{drift}} | > \text{DriftThreshold} \quad (6)$$

If Δ_{drift} exceeds the drift threshold, the framework raises a warning, indicating that the sensor may have experienced significant drift, suggesting potential sensor malfunction. This mechanism ensures that any unexpected behavior in sensor data is caught early, preventing misinterpretation of flight test data due to unreliable measurements [13].

Fig. 9 shows the results of strain gauge sensor calibration and drift detection analysis. The first sensor shows a steady rise in readings, which indicates positive drift during the measurement period, but the second sensor displays a DC offset because of miscalibration.

Sensor Analysis: SGL_LR01

Errors:

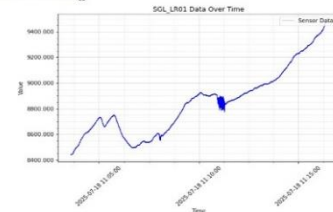
- DAQ System Range Exceedance: False
- Flat Detection: False
- Time Errors: False

Warnings:

- Drift Warning: True
- Calibration Warning: True
- Expected Range Exceedance: False

Analysis Results:

- All Values in Unit: Micrometer/meter
- Max Value: 9486.314
- Min Value: 8437.786
- Calibration Start At Value: 8441.892
- Calibration End At Value: 9461.318
- Actual Delta Of Drift: 1019.426
- Calibration A0: -18836.000
- Calibration A1: 37611.000
- Time Errors Found in CSV Line: []



Sensor Analysis: SGL_LR04

Errors:

- DAQ System Range Exceedance: False
- Flat Detection: False
- Time Errors: False

Warnings:

- Drift Warning: False
- Calibration Warning: True
- Expected Range Exceedance: False

Analysis Results:

- All Values in Unit: Micrometer/meter
- Max Value: -559.929
- Min Value: -652.328
- Calibration Start At Value: -607.801
- Calibration End At Value: -610.554
- Actual Delta Of Drift: 2.753
- Calibration A0: -18836.000
- Calibration A1: 37611.000
- Time Errors Found in CSV Line: []

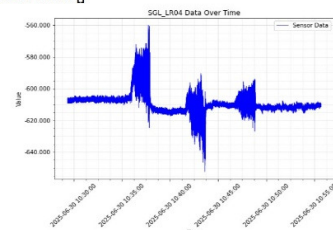


Fig. 9. Calibration and drift detection results of sensors.

Frequency Domain Analysis (FFT)

To analyze periodic patterns and detect mechanical anomalies in aircraft sensor data, the framework transforms time-domain signals into the frequency domain using Fast Fourier Transform (FFT). This transformation enables the identification of dominant frequency

components, which may correspond to vibrations, resonances, or structural issues. Detecting such frequencies is essential to determine if abnormal oscillatory behaviors occurred during the flight. The FFT is mathematically represented as:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n / N} \quad (7)$$

Where:

x_n is the time-domain signal,

N is the number of samples,

X_k is the resulting complex-valued frequency component at index k .

Frequency domain analysis helps engineers to observe high-amplitude frequencies and assess whether they fall within expected limits, or indicate potential issues like imbalance, misalignment, or component fatigue.

Butterworth Filtering

Butterworth filters are applied to refine sensor data. These filters can be used to remove undesirable frequencies (high-pass filter) or to smooth out high-frequency noise (low-pass filter), thus cleaning up the data to concentrate on meaningful signals. The pipeline also uses low-pass and high-pass Butterworth filters on the data [14].

The Butterworth filter is mathematically expressed as follows:

$$H(s) = \frac{1}{\sqrt{1 + \left(\frac{s}{w_c}\right)^{2n}}} \quad (8)$$

or equivalently in terms of frequency response:

$$|H(j\omega)|^2 = \frac{1}{1 + \left(\frac{\omega}{w_c}\right)^{2n}} \quad (9)$$

Where:

$H(s)$ is the transfer function,

w_c is the cut-off angular frequency,

n is the filter order,

Butterworth filters are used to filter the accelerometer sensor data and remove unwanted frequencies by making the signal clearer.

Electromagnetic Interference (EMI) Detection

The process of solving electromagnetic interference (EMI) problems in eVTOL development needs substantial additional work beyond typical aircraft development programs. The dedicated function exists to tackle all EMI-related problems. To identify potential EMI in sensor data, the framework applies a low-pass

filter to isolate low-frequency components where EMI should not typically be present. After filtering, it performs a peak-to-peak (P2P) variation check on the resulting signal. A significant variation suggests the presence of low-frequency noise that has leaked into the low-frequency spectrum, possibly indicating conducted EMI [15]. Although this method does not explicitly separate EMI components in the frequency domain, it offers an indirect means of detection by revealing unexpected fluctuations or irregular peaks within frequency bands that are expected to remain stable under nominal operating conditions.

Fig. 8 in the accelerometer sensor analysis section presents the EMI induced in transient responses. The corresponding pseudocode provided below outlines the automated detection and filtering process.

BEGIN Check_EMI_Effect

INPUT: signal, fs, cutoff = 4

OUTPUT: Boolean indicating EMI presence

```
// Apply low-pass filter to the signal
filtered_signal ← Butter_Lowpass_Filter(signal, cutoff,
fs)
```

```
// Check if peak-to-peak exceeds threshold
emi_detected ← peak_to_peak(filtered_signal) ≥ 1
```

RETURN emi_detected

END Check_EMI_Effect

Temperature Sensor Analysis

The `analyze_temperature_sensor` function detects unexpected temperature changes that occur without physical reasons while the sensor remains attached to a structure to alert users about system failures or external interference.

BEGIN Analyze_Temperature_Sensor

INPUT: values, sample_rate, timestamps, sensor_name

OUTPUT: analysis_results, warnings

```
// Compute differences between consecutive
temperature readings
temperature_diff ← diff(values)
```

```
// Define threshold for sudden changes in degrees
Celsius
sudden_change_threshold ← 1.0
```

```
// Detect sudden changes
sudden_change_detected ← any(abs(temperature_diff)
> sudden_change_threshold)
sudden_changes_count ← sum(abs(temperature_diff) >
sudden_change_threshold)
```

```
// Prepare analysis results
```

```

analysis_results ← {
  "Sudden Changes Count": sudden_changes_count
}

// Prepare warnings
warnings ← {
  "Sudden Change Warning": sudden_change_detected
}

```

RETURN analysis_results, warnings

END Analyze_Temperature_Sensor

This analysis highlights potential faults or environmental disturbances that may have influenced sensor response during flight [16].

Generic Sensor Analysis

For overall sensor analysis, the *analyze_generic_sensor* function handles the analysis of various sensors by performing the aforementioned checks, like drift and miscalibration, according to sensor type-specific thresholds. This functionality makes the framework flexible and extensible, allowing it to dynamically address various sensor types.

Plotting Functions

Flight analysis depends on plot functions to show test data, which helps users identify patterns, spot problems, and share their findings clearly. The *generate_time_series_plot* function shows sensor information through the time domain to help users identify sudden changes or unusual patterns in the data. The *generate_fft_plot* function converts signals into frequency domain data to show both resonant frequencies and vibration patterns. The *generate_filter_plots* function shows both unfiltered and filtered data through Butterworth filters, which helps engineers check filter performance and maintain essential signal features.

Fig. 10 illustrates the time series, FFT magnitude spectrum and corresponding low-pass and high-pass filter responses of the accelerometer sensor, highlighting vibration activity and signal components across different frequency ranges.

Sensor Analysis: AA0_FSR1_Z

Errors:

- DAQ System Range Exceedance: False
- Flat Detection: False
- Time Errors: False

Warnings:

- EMI Warning: False
- Drift Warning: False
- Calibration Warning: False
- Expected Range Exceedance: False

Analysis Results:

- All Values in Unit: g
- Max Value: 2.903
- Min Value: -3.238
- Calibration Start At Value: -0.049
- Calibration End At Value: -0.050
- Actual Delta Of Drift: 0.001
- Dominant Frequency: ['72.680', '75.377', '72.681', '75.378', '72.680', '74.396', '75.376', '75.368', '75.289', '74.395']
- Calibration A0: -53.310
- Calibration A1: 106.610
- Time Errors Found in CSV Line: []

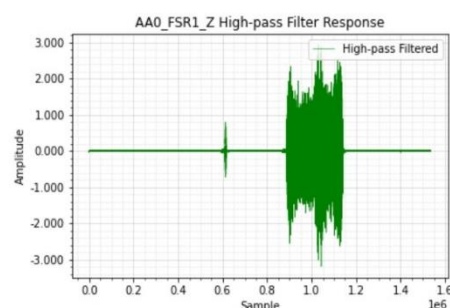
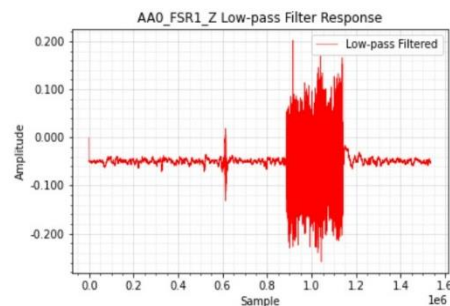
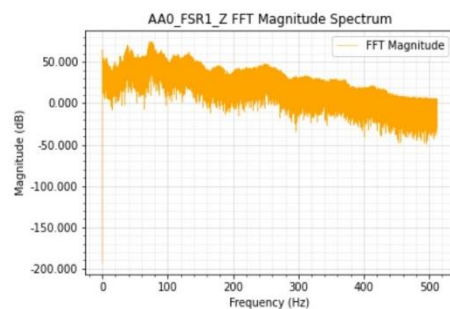
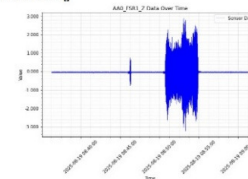


Fig. 10. Time series, Frequency domain and filter-based analysis.

Analyze Sensor Data

The *analyze_sensor_data* function serves as the main processing hub of the framework. This approach is a series of dedicated analysis functions such as flatness check, range exceedance check, calibration drift assessment, time synchronization validation, and frequency domain evaluation. Rather than performing complex decision-making or automatic sensor-type-based processing internally, it orchestrates and dispatches the appropriate sub-functions in a structured order [17]. This dispatch is based on the available sensor data and the parameters defined in the JSON configuration. The function also handles logging of analysis steps, reporting of errors and warnings, and the generation of visualizations like time-series plots, ensuring that all relevant checks are applied consistently across different sensors. This centralized structure simplifies code maintenance, enhances the modularity of the framework, and ensures that new diagnostic checks or analytical routines can be integrated with minimal modifications to the overall workflow.

Report Generation

The generation of a comprehensive report is essential for representing the outcomes of the data analytics process. Such reports typically include a summary of the investigation, a description of any errors or anomalies observed, and a presentation of the analysis results. This documentation is important for maintaining transparency and accountability in flight data analysis.

The framework is equipped with a powerful reporting framework capable of issuing detailed PDF reports based on the performed analysis. The *generate_pdf_report* function is designed to create informative reports that summarize the analytical findings, present metrics on data quality (errors and warnings), and include visual representations of sensor data [18]. These reports serve as key documents for engineers and stakeholders, providing a clear and comprehensive record for understanding the results of flight tests.

Log File Generation

The framework incorporates a robust logging mechanism to record key events, warnings, and errors encountered during the analysis process. This detailed logging is invaluable for auditing, debugging, and verifying the processing steps. Central to this are the *configure_logging* function and a custom logging level, NOISSUE.

A custom logging level, NOISSUE (numerically defined as 25, between INFO and WARNING), was added to the standard Python logging module [19]. This level is used to explicitly

indicate that a particular check or analysis for a sensor was completed without finding any issues. The *noissue* method is a helper function added to the logging. Logger class to facilitate this:

BEGIN Define_Custom_Logging_Level

```
// Define a custom logging level called 'NOISSUE' with
numeric value 25
NOISSUE_LEVEL ← 25
Add_Logging_Level_Name(NOISSUE_LEVEL,
'NOISSUE')
```

```
// Define custom logging method for 'NOISSUE'
messages
FUNCTION noissue(self, message, *args, **kwargs)
    IF self.isEnabledFor(NOISSUE_LEVEL) THEN
        self._log(NOISSUE_LEVEL, message, args,
kwargs)
    END IF
END FUNCTION
```

```
// Attach custom method to Logger class
Logger.noissue ← noissue
```

END Define_Custom_Logging_Level

This function is called to log a confirmation message when the analysis for a specific sensor has been completed successfully, and no problems were detected. It enhances the clarity and completeness of the logs by explicitly stating that sensor data passed all relevant checks [20].

The *configure_logging* function establishes the logging framework to ensure structured and comprehensive logging throughout the analysis.

BEGIN Configure_Logging

```
INPUT: info_log_filename, error_log_filename
// Create or get the root logger
logger ← Get_Logger()
```

```
// Remove any existing handlers
FOR EACH handler IN Copy_Of(logger.handlers)
    logger.Remove_Handler(handler)
END FOR
```

```
// Set the base logging level to INFO
logger.Set_Level(INFO)
```

END Configure_Logging

Perform Analysis

The *perform_analysis* function serves as the main entry point and orchestrator for the entire data analysis pipeline. It manages the sequence of operations from initial file gathering and configuration loading to PCAP parsing, computational analysis, and final report generation.

Parallel Processing and Worker Function

The framework utilizes parallel processing to efficiently handle large volumes of flight test data. Instead of sequentially processing sensor data, it distributes the workload across multiple CPU cores using Python's multiprocessing module. This significantly speeds up operations such as parsing PCAP files via *parse_pcap_worker*, generating sensor-specific CSVs via *create_csv_by_sensor_type_parallel*, and performing mathematical analyzes via *perform_mathematical_analysis_parallel*. This approach is especially beneficial in environments where sufficient CPU and memory resources are available [21].

Parallel PCAP Parsing

The *parse_pcap_worker* function is designed to handle the concurrent parsing of individual PCAP files. Each worker process takes a PCAP file path and associated configuration as arguments and invokes the *incremental_parse_pcap* function to perform the actual parsing and calibration.

BEGIN Parse_PCAP_Worker

```

INPUT: file_path, config, current_year, filter_port,
filter_mode

// Initialize logger
logger ← Get_Logger()

TRY
    // Incrementally parse the PCAP file
    results ← Incremental_Parse_PCAP(file_path,
config, current_year, filter_port, filter_mode)

    RETURN results AS list
CATCH Exception AS e
    // Log critical error
    logger.Critical("Critical error while parsing PCAP
file", file_path, "Error:", e)
    // Re-raise the exception
    RAISE e

```

END Parse_PCAP_Worker

Parallel CSV Creation

The creation of CSV files, partitioned by sensor type, is also parallelized to enhance efficiency. The *create_csv_by_sensor_type_parallel* function orchestrates this. It first groups the combined, parsed data frame by sensor type. For each sensor type, the relevant data subset is temporarily saved to a pickled file. The paths to these temporary files, along with other necessary arguments, are then passed to worker processes that execute function [22]. The function, as detailed previously, handles the conversion of a pickled pandas data frame for a single sensor type into a structured CSV file,

complete with metadata headers. The function manages the distribution of these tasks to a pool of worker processes. This parallel execution significantly reduces the time required for generating multiple CSV files, especially when dealing with a large number of distinct sensor types.

Parallel Mathematical Analysis

Similarly, the *perform_mathematical_analysis_parallel* function distributes the computational load of data analyzing different sensors [23]. It groups the main data frame by sensor name, and then each group (representing all data for a single sensor) is passed as an argument to a worker process that executes the *analyze_sensor_data* function.

This parallelization of the mathematical analysis phase, where functions like filtering, FFT, range checks, and sensor-specific diagnostics are performed, ensures that the processing time scales favorably with the number of sensors and the volume of data, rather than being bottlenecked by sequential execution [24].

Results AND Findings

The framework systematically processes PCAP recordings from the raw data stream to the final analysis and reporting. Using a JSON-based configuration file to decode packets and define sensor structures, it converts digital readings into engineering units with precise timestamps. Processed data are organized into structured tables and exported as sensor-specific CSV files containing calibration and timing information. During analysis, the framework verifies time-interval consistency, miscalibration and drift, applies FFT-based spectral analysis with low- and high-pass filtering to identify dominant frequency patterns, flatness, and range violations. The framework presents the results in a nicely structured report. Systematic logging of all detected anomalies ensures transparent, traceable evaluation results. [25,26].

The framework demonstrates its analytical abilities by generating multiple visual outputs. The time-series plot in Figure 10 demonstrates error-free patterns of the accelerometer sensor signal during normal operation. Figures 5, 6, 7, 8, and 9 show signal anomalies and irregularities which help in the detection of potential malfunctions or disturbances. Overall, the collection of the above figures offers an extensive understanding of sensor performance and signal integrity, and framework dependability from start to finish of the testing process.

The framework shows different patterns of resource usage throughout its operational

stages. The decoding and calibration stages require the most computational power because they perform complex data extraction and structural mapping, and large sensor dataset conversion into engineering units. The Fast Fourier Transform and digital filtering operations during signal processing increase computational needs because of their complex algorithms. The output generation tasks that produce CSV files and reports require minimal system resources because they mainly perform input/output operations.

Performance Evaluation

Evaluations were performed on a 12th Gen Intel Core i7-12700H (2.30 GHz) with 32 GB of RAM. By distributing the `perform_analysis` function across a `ProcessPoolExecutor`, the system processes multiple PCAP files in parallel, substantially reducing end-to-end runtimes [27]. A 12.72-minute recording was processed in 4.36 minutes (2.90 minutes for parsing), whereas a 68.15-minute recording completed in 28.18 minutes (7.77 minutes for parsing).

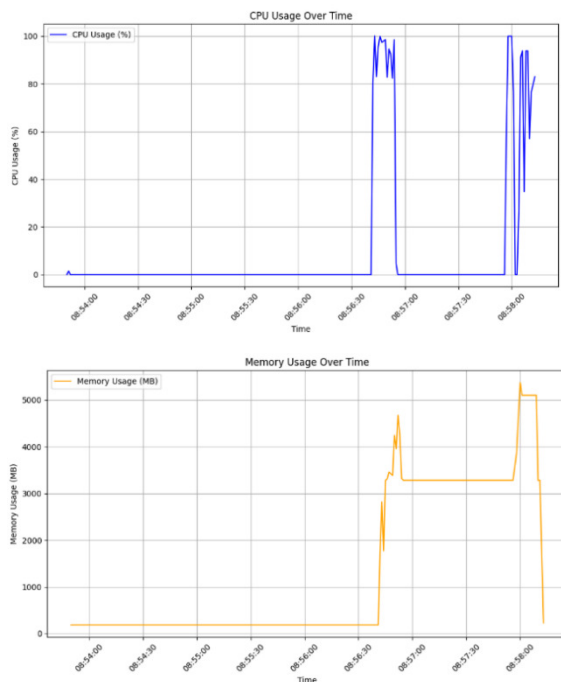


Fig. 11. Short Recording

When we evaluate the short recording performance results resource metrics shows where processing demand is concentrated across the pipeline. During the 2.90-minute parsing phase (08:53:50 – 08:56:44), the `Incremental_Parse_PCAP` function kept CPU and memory usage at low levels, which aligns with the expected behavior of chunk-based data processing in the short recording. At 08:56:45, the CSV generation phase started and ran for 1.20 minutes, during which CPU utilization reached 100% and memory allocation rose to

5,000 MB. The mathematical analysis and PDF report generation stages of the short recording completed in 0.22 and 0.03 minutes respectively, after which resource usage dropped back to baseline. Overall, these measurements show that data processing in the short recording places limited strain on the system, while converting large datasets into output formats remains the main source of high resource demand.

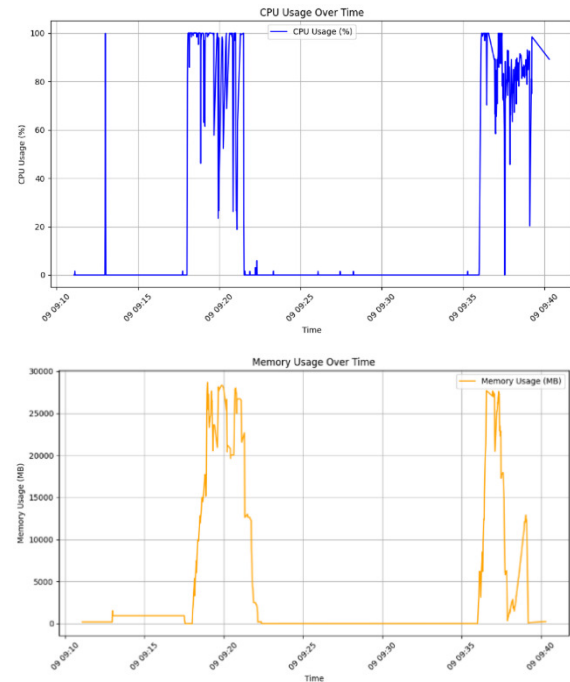


Fig. 12. Long Recording

For the long recordings, it is observed that how resource demand scales with data volume. The 7.77-minute parsing phase (09:11:05 – 09:18:51) maintained low CPU and memory usage throughout, consistent with the behavior observed in the short recording and confirming that the `Incremental_Parse_PCAP` function handles long recording without placing undue stress on the system. At 09:19:18, the CSV generation phase of the long recording began and ran for 16.72 minutes, during which CPU utilization reached 100% and memory peaked at nearly 28,000 MB. The mathematical analysis phase of the long recording added a further 3.18 minutes of elevated resource usage, a stage that was far less prominent in the short recording.

Across both sessions, the parsing phase remained stable regardless of dataset size, whereas the CSV generation phase grew from 1.20 minutes to 16.72 minutes as data volume increased. These findings suggest that incremental parsing scales well, while the conversion of large datasets into structured

output formats is the primary factor driving resource demand and processing time.

Discussion

The proposed framework provides users with a powerful tool to automate post-data processing.

Strengths of the Framework:

- **Modularity:** The code is modular, in terms of functions and classes, which allows one to reuse the code and maintain its sustainability [28].
- **Scalability:** The framework provides parallel processing facilities, allowing scaling to computational resources and is thus applicable for large-scale data analysis [29].
- **Configurability:** It is possible to have flexible sensor mappings, gain and bias calibration coefficients and operation settings through a JSON configuration file.
- **Comprehensive Analysis:** The framework implements various analyzes from simple false checks to advanced signal processing techniques of sensor data [30].
- **Automated Reporting:** Results are sent via email as PDFs with the help of ReportLab.

Limitations and Future Enhancements:

Although the framework is very general, code analysis suggests reinforcements:

- **Real-time Processing:** The current implementation is tailored to post-processing of PCAP files. Extending the framework to handle real-time data streams would boost its utility for live monitoring purposes.
- **Extensibility:** The implementation of additional sensor types and complex calibration models might require code modifications to the framework. Such a custom plugin framework and better configuration handling would enable additional extensibilities [31].
- **User Interface:** A web-based GUI or dashboard implementation could enhance the user experience for users who prefer web interfaces over command-line interfaces.
- **Programming Language Selection:** The selection of Python over C and C++ occurred because Python provides an extensive library ecosystem, fast prototyping abilities, simple data analysis and visualization tool integration (NumPy, Pandas, Matplotlib). The research framework benefits from Python's flexibility, maintainability and shorter development cycles because it needs to undergo many algorithmic changes. If a higher execution speed is required, then C/C++ can be preferred.

Conclusion

The research presents a complete Python framework for post-flight data analysis, which handles IENA packets from PCAP files through organized parsing and calibration and analysis steps. The system contains a flexible design which performs automatic packet decoding and sensor data calibration based on JSON parameters and checks timing accuracy to identify packet transmission problems or synchronization issues. The system performs advanced signal processing through FFT and Butterworth filtering for frequency and noise analysis. It also includes anomaly detection tools to detect EMI effects and range exceedances, and flatline behaviors. The system uses multiprocessing technology together with memory optimization to process big flight datasets at high speeds. The system produces calibrated CSV files and time-series, spectral plots and a detailed PDF report, which contains all analysis findings for quick and reliable post-flight assessment.

The adaptable design of the framework enables future development to incorporate real-time data processing capabilities and adjustable configuration options for different aircraft and improved operator interfaces for enhanced workflow integration. The tool will achieve operational efficiency improvements while becoming an essential capability for flight test data analysis by solving existing framework limitations, which will enable data-driven airworthiness assessments and informed decision-making throughout the aircraft test and evaluation lifecycle.

References

- [1] Z. Yin, et al., Real-time Flight Test Data Management Framework Design, *IEEE Access* 10, 36121–36133 (2022); doi:10.1109/ACCESS.2022.3163473
- [2] X. Zhang, et al., Parsing and Reconstruction of Complex IENA Packets in Avionics Networks, *Journal of Aerospace Information Frameworks* 17, 250–259 (2020); doi:10.2514/1.1010785
- [3] R. Sánchez-González, et al., A Modular Data Acquisition and Processing Architecture for Aerospace Flight Testing, *Sensors* 21, (2021); doi: 10.3390/s21124030
- [4] J. Almeida, et al., Impact of EMI on Digital Signal Acquisition Frameworks in Aircraft, *IEEE Aerospace and Electronic Frameworks Magazine*, 36, 28–37 (2021); doi:10.1109/MAES.2021.3051293
- [5] X. Qin, et al., A Reconfigurable Embedded Architecture for Flight test Instrumentation, *IEEE Transactions on Aerospace and Electronic Frameworks* 59, 470–482 (2023); doi:10.1109/TAES.2022.3203745

- [6] C. Niu, et al., An Automated Framework for Flight Data Post-processing Using Model-based Design, *Information Sciences* 598, 169–185 (2022).
- [7] M. Savić, et al., Flexible Framework for Aircraft Telemetry and Data Analysis Using JSON Configurations, *Aerospace* 8, article 182 (2021); doi:10.3390/aerospace8060182
- [8] Z. Li, et al., A Lightweight Telemetry Protocol for Real-time Avionics Data Streaming, *IEEE Transactions on Aerospace and Electronic Frameworks* 57, 4123–4135 (2021); doi: 10.1109/TAES.2021.3069056
- [9] L. Cao, et al., High-Fidelity PCAP-Based Flight Data Recorder for UAVs, *IEEE Access* 10, 66143–66152 (2022), doi:10.1109/ACCESS.2022.3183321
- [10] S. Song, et al., Drift Detection in Flight Data Sensors Using Change Point Analysis, *IEEE Sensors Journal* 23, 2235–2243 (2023); doi:10.1109/JSEN.2023.3236781
- [11] J. Elson, L. Girod and D. Estrin, Fine-grained Network Time Synchronization Using Reference Broadcasts, *ACM SIGOPS Operating Frameworks Review* 36, 147–163 (2002); doi:10.1145/844128.844143
- [12] X. Liu et al., Application of Fast Fourier Transform in Vibration Analysis of Aircraft Engines, *IEEE Access* 9, 55123–55132 (2021); doi:10.1109/ACCESS.2021.3070612
- [13] R. B. Randall and J. Antoni, Rolling Element Bearing Diagnostics—A Tutorial, *Mechanical Frameworks and Signal Processing* 25, 485–520 (2011); doi:10.1016/j.ymssp.2010.07.017.
- [14] Y. Chen, et al., Butterworth Filter Optimization for Inertial Sensor Noise Reduction, *IEEE Transactions on Instrumentation and Measurement* 68, 739–748 (2019); doi:10.1109/TIM.2018.2859018.
- [15] H. Bai, et al., EMI Detection in Aircraft Sensor Networks Using Spectral Analysis, *IEEE Transactions on Electromagnetic Compatibility* 61, 1412–1420 (2019); doi:10.1109/TEM.2018.2883376
- [16] D. Yang, et al., Flight Test Sensor Fault Detection and Calibration Using Redundant Measurements, *Sensors* 19, 3104 (2019); doi: 10.3390/s19143104
- [17] Y. Zhou et al., "Parallel Processing for Large-scale Flight Data Analysis Using Python Multiprocessing," *Future Generation Computer Frameworks*, 118:78–90, 2021, doi:10.1016/j.future.2020.12.009.
- [18] F. He, et al. Scalable Signal Analysis Framework Using Multicore Processing for Aircraft Testing, *Journal of Aerospace Information Frameworks* 19, 86–98 (2022); doi:10.2514/1. I010974
- [19] Q. Huang, et al., Interactive Dashboards for Visual Flight Test Data Exploration, *Aerospace* 8, 89 (2021); doi:10.3390/aerospace8030089
- [20] L. Fernandez, et al., Using PDF and Markdown Reporting in Post-flight Analysis Tools, *IEEE Access* 8, 136543–136552 (2020); doi:10.1109/ACCESS.2020.3011367
- [21] D. Beazley and B. K. Jones, *Python Cookbook: Recipes for Mastering Python 3*, 3rd ed., O'Reilly Media, ch. 12, pp. 527–556 (2013).
- [22] P. Martínez, et al., Unified Configuration File Approach for Multi-sensor Flight Test Data Frameworks, *Sensors* 21, 3156 (2021); doi:10.3390/s21093156
- [23] D. W. Aha, L. Breslow and H. Muñoz-Avila, Parallelizing Data Analysis Tasks Using Data Partitioning and Multiprocessing, *Journal of Parallel and Distributed Computing* 98, 52–64 (2016); doi:10.1016/j.jpdc.2016.06.003
- [24] A. Verma, et al., Large-Scale Cluster Management at Google with Borg, *Proceedings of the European Conference on Computer Frameworks (EuroSys)* 1–17 (2015); doi: 10.1145/2741948.2741964
- [25] M. Blanke, *Diagnosis and Fault-Tolerant Control*, 2nd ed., Springer, (2016); doi:10.1007/978-3-319-21476-6
- [26] W. Zhou, et al., Design and Implementation of a Digital Filter for Aerospace Sensor Signal Processing, *Aerospace Science and Technology*, 113, Art. no. 106699 (2022); doi:10.1016/j.ast.2021.106699
- [27] J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, *Communications of the ACM* 51, 107–113 (2008); doi:10.1145/1327452.1327492.
- [28] T. White, *Hadoop: The Definitive Guide*, 4th ed., O'Reilly Media, (2015).
- [29] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, (2003).
- [30] J. Smith and R. Jones, Automated Data Analysis and Reporting Frameworks for Large-scale Sensor Networks, *IEEE Transactions on Industrial Informatics* 15, 2253–2263 (2019); doi:10.1109/TII.2018.2885472
- [31] E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, ch. 4, pp. 139–216 (1994).