GENERATING SYNTHETIC FLIGHT TRACKS FOR COLLISION RISK MODELING USING GAUSSIAN PROCESSES

by

Oleksandra Snisarevska
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Systems Engineering and Operations Research

Committee:

_____ Dr. Jie Xu, Dissertation Director

_____ Dr. Lance Sherry, Co-Director

_____ Dr. John Shortle, Committee Member

_____ Dr. Ben Seiyon Lee, Committee Member

_____ Dr. Guiping Hu, Department Chair

Date:_____ Fall Semester 2026
George Mason University
Fairfax, VA

Generating Synthetic Flight Tracks for Collision Risk Modeling Using Gaussian Processes

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

by

Oleksandra Snisarevska
Master of Science
George Mason University, 2018
Bachelor of Science
Igor Sikorsky Kyiv Polytechnic Institute, 2015

Director: Jie Xu, Professor
Co-Director: Lance Sherry, Professor
Department of Systems Engineering and Operations Research
Fall Semester 2026
George Mason University
Fairfax, VA

Copyright 2026 Oleksandra Snisarevska
All Rights Reserved

DEDICATION

I dedicate this work to my loving parents and to both of my grandfathers.
Я присвячую цю роботу моїм люблячим батькам і обом дідусям.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep gratitude to my dissertation advisors, Dr. Lance Sherry and Dr. Jie Xu, for their invaluable guidance, mentorship, and inspiration throughout my entire graduate studies. Dr. Sherry introduced me to the world of aeronautic and aerospace systems in my first class as a Master's student in Fall 2016. I have become passionate about aviation and been developing my aerospace career over the last decade. Dr. Xu introduced me to applied predictive modeling back in 2018, laying the foundation for my research interests.

I want to thank my committee members, Dr. John Shortle and Dr. Ben Seiyon Lee, for their generous advice, questions, and suggestions to enhance my research, and for all the support at the Center for Air Transportation Systems Research and the College of Engineering and Computing at George Mason University.

I want to thank my friends and former colleagues, Seungwon, Azin, and Chaitanya, as well as my current colleagues, Shahab, Fahimeh, and Ashim, at the Center for Air Transportation Systems Research, for their contributions to my research presentations.

I want to thank all my friends and colleagues at Bell Textron Inc. for their persistent support, encouragement, and motivation throughout my PhD program.

I want to thank my friends outside of school and work, Mariia, Illia, Svitlana, and Batia, for their strong confidence in me.

I want to thank Austin for his embracing help, encouragement, and inspiration.

Finally, I want to thank my family for their unwavering, unconditional support, inspiration, and dedication throughout my entire educational journey. I could not have done it without you. I love you.

TABLE OF CONTENTS

Page
List of Tables ix
List of Figures x
List of Equations xiii
List of Abbreviations xiv
Abstract xvi
Chapter One: Introduction 1
1.1 Domain 1
1.1.1 National Airspace System 1
1.1.2 Target Level of Safety 2
1.1.3 Rare Events in Aviation 3
1.2 Scope 4
1.2.1 Air-to-Air Collisions 4
1.2.2 Collision Risk Modeling 7
1.3 Problem Description 9
1.4 Approach 13
1.5 Research Objective 14
1.6 Research Questions 15
1.7 Unique Contribution 18
Chapter Two: Literature Review Part I – State-of-the-Art Collision Risk Modeling 19
2.1 Air-to-Air Collision Risk Modeling 19
2.1.1 Collision Models 22
2.1.1.1 Analytical Models 22
2.1.1.2 Simulation Models 24
2.2 Flight Trajectory Models 25
2.2.1 Historical Flight Tracks 25
2.2.2 Simulated Flight Tracks 26

2.2.2.1 Physics-based Models 27
2.2.2.2 Agent-based Models 29
2.2.3 Synthetic Flight Tracks 30
2.2.3.1 Variational Autoencoders 33
2.2.3.2 Generative Adversarial Networks 37
2.2.3.3 Diffusion Models 40
2.2.3.4 Physics-Informed Neural Networks 41
2.2.3.5 Gaussian Processes 42
2.3 Gap Analysis in Flight Trajectories Literature 45
Chapter Three: Literature Review Part II – Gaussian Processes 50
3.1 Overview of Gaussian Processes 50
3.1.1 Definitions 50
3.1.2 Properties of Stationarity and Isotropy 52
3.1.3 Kernel Functions 53
3.1.3.1 Radial Basis Function (RBF) Kernel 54
3.1.3.2 Matérn Kernel 57
3.1.3.3 Linear Kernel 60
3.1.3.4 Periodic Kernel 61
3.2 Gaussian Processes Implementations 62
3.2.1 Gaussian Processes Regression (GPR) 62
3.2.2 Deep Gaussian Process (Deep GP) 67
3.3 Challenges in Gaussian Process Implementations 69
3.3.1 Problem of Non-Stationarity 69
3.3.1.1 Engineering Ad-hoc Solutions 74
3.3.1.2 Partitioning Methods 75
3.3.2 Multi-Output Problem 78
3.4 Gap Analysis in Gaussian Processes Literature 79
Chapter Four: Methodology 81
4.1 Historical Data 82
4.2 Synthetic Flight Trajectories 90
4.2.1 Learning Framework 91
4.2.1.1 Predictor and Response Data Selection 91

4.2.1.2 Gaussian Process (GP) Implementation 92
4.2.1.3 Data Stabilization and Normalization 96
4.2.1.4 Kernel Function Design 96
4.2.1.5 Log Marginal Likelihood Maximization 102
4.2.2 Generating Framework 103
4.2.3 Validation Framework 108
Chapter Five: Results 110
5.1 Historical Flight Track Data for DCA Airport 110
5.1.1 Data Collection 110
5.1.2 Data Preprocessing 115
5.1.3 Data Standardization 120
5.1.3.1 Positional Data Interpolation 120
5.1.4 Data Clustering 124
5.1.4.1 Runway Identification 125
5.1.4.2 Navigation Procedure Identification 130
5.1.5 Data Analysis 136
5.1.5.1 Measurements at Runway Threshold 137
5.1.5.2 Measurements at 3, 6, and 9 NM from Runway Threshold 139
5.1.5.3 Measurements at Waypoints 142
5.1.5.4 Derived Data Parameters 145
5.1.6 Summary 148
5.2 Synthetic Flight Tracks 149
5.2.1 GPR Models with Single Response 154
5.2.1.1 Learning Framework 154
5.2.1.2 Generating Framework 158
5.2.1.3 Validation Framework 172
5.2.2 GPR Models with Single Response – Predict Next n^{th} Position 178
5.2.2.1 Learning Framework 178
5.2.2.2 Generating Framework 182
5.2.3 GPR Model with Encoded Multi-Response 186
5.2.3.1 Learning Framework 186
5.2.3.2 Generating Framework 188

5.2.4 Deep GP with Encoded Multi-Response 195
5.2.4.1 Learning Framework 195
5.2.4.2 Generating Framework 196
5.2.5 GPR Model with Principal Components 202
5.2.5.1 Learning Framework 202
5.2.5.2 Generating Framework 205
5.2.5.3 Validation Framework 210
5.2.6 Summary 217
5.2.6.1 Synthetic Data Quality 217
5.2.6.2 GP Model Scoring 220
Chapter Six: Conclusions 224
6.1 Research Objective 224
6.2 Research Questions and Answers 226
6.3 Future Work and the Next Steps 228
Appendix A – Python Scripts to Generate Kernel Functions 230
Appendix B – Historical Data 240
Appendix C – Waypoint Data 244
Appendix D – Additional Synthetic Results from Model 5 248
Appendix E – Python Scripts for Historical Data 256
Appendix F – Python Scripts for GP Models 300
References 384

LIST OF TABLES

Table Page
Table 1 Literature Review on Synthetic Flight Trajectories 32
Table 2 Strengths and Weaknesses of Flight Trajectory Generation Methods 47
Table 3 Strengths and Weaknesses of Methods Addressing Non-Stationarity in GPs 73
Table 4 Derived Data Parameters 89
Table 5 Strength and Applicability of Kernel Functions 101
Table 6 Model Performance Metrics 106
Table 7 Date and Time Conversion to Unix Timestamp 113
Table 8 Filtering Conditions for Trajectory Removal from Collected Data 117
Table 9 Filtering Conditions for Trajectory Removal from Preprocessed Data 123
Table 10 Independent GPR Models with Single Response 155
Table 11 Response Transformation in Model 1A 156
Table 12 Kernel Functions and Log Marginal Likelihood in Model 1 158
Table 13 Response Reconstruction in Model 1A 163
Table 14 Performance Metrics for Model 1 171
Table 15 GPR Models with Single Responses in Absolute Coordinates 179
Table 16 Predictors and Responses in Models 2A and 2B 180
Table 17 Initial Kernel Functions in Model 2A and 2B 181
Table 18 Modified Kernel Functions in Model 2A and 2B 182
Table 19 Distance between Consecutive Points 187
Table 20 Kernel Function in Observation Layer 195
Table 21 Deep GP Model Performance Metrics 201
Table 22 Model Performance Metrics for GPR with PCs in Response 209
Table 23 Synthetic Data Quality in Model 1 vs Model 5 218
Table 24 Scoring Card for GP Models 222
Table 25 STARs - Standard Terminal Arrivals – SKILS FIVE (RNAV) 244
Table 26 IAPs - Instrument Approach Procedures – RNAV (GPS) Y RWY 19 244

LIST OF FIGURES

Figure Page
Figure 1 Midair Collisions in the US from 2008 to 2024 (NTSB [14]) 5
Figure 2 FAA's Risk Matrix for Commercial Aviation [15] 6
Figure 3 Midair Collisions per Flight Phase in the US from 2008 to 2024 (NTSB [14]) .. 7
Figure 4 Three Pillars in Synthetic Flight Tracks 17
Figure 5 Breakdown of Key CRM Components derived from [32, p. 10] 20
Figure 6 Process Flow for Generic Collision Risk Modeling 22
Figure 7 Geometric Representation of the Reich Model [44] 23
Figure 8 Autoencoder (AE) 34
Figure 9 Variational Autoencoder (VAE) 34
Figure 10 Generative Adversarial Networks (GANs) 38
Figure 11 Flight Track with Spatially Sampled Points at Equidistance of Each Other 43
Figure 12 Samples from Prior Distribution with RBF Kernel 55
Figure 13 Samples from Prior Distribution with Matérn Kernel 59
Figure 14 Samples from Prior Distribution with Linear Kernel 61
Figure 15 Samples from Prior Distribution with Periodic Kernel 62
Figure 16 Prior and Posterior Distributions from GPR Model with Matérn Kernel 66
Figure 17 Example of Non-Stationary Function 70
Figure 18 GPR Application with Matérn Kernel on a Non-Stationary Function 71
Figure 19 Function Splitting and Linearity Removal 74
Figure 20 Algorithm for Performing Collision Risk Modeling 81
Figure 21 Historical Flight Track Data Processing 83
Figure 22 Interpolation between Two Points 87
Figure 23 Modular Framework 90
Figure 24 Matérn Kernel with $v = 0.5$ 98
Figure 25 Composite Kernel with Matérn and Rational Quadratic Functions 101
Figure 26 36 NM Bounding Box around DCA Airport 111
Figure 27 SQL Query to OpenSky Network Trino Database 112
Figure 28 Raw Data for Arrival Flight Trajectories at DCA 115
Figure 29 The First and Last Recorded Positions from Each Flight Track 118
Figure 30 Preprocessed Data for Arrival Flight Trajectories at DCA 120
Figure 31 A Flight Track Before Standardization 121
Figure 32 A Normalized Flight Track 122
Figure 33 Postprocessed Historical Flight Track Data 124
Figure 34 DCA Airport Diagram 126
Figure 35 DBSCAN Clustering for Runway Identification 128

Figure 36 Runway Clusters at DCA 129
Figure 37 Arrivals on Runway 19 at DCA 130
Figure 38 DBSCAN Clustering for Navigation Procedure Identification 131
Figure 39 Runway 19 Clusters at DCA 132
Figure 40 Unclassified Arrivals on Runway 19 at DCA 133
Figure 41 Runway 19 Northern Cluster Data 134
Figure 42 Last Recorded Position Across All Flight Tracks 135
Figure 43 Runway 19 Threshold 135
Figure 44 Interpolated Last Recorded Positions across All Flight Tracks 136
Figure 45 Data Distributions at the Runway Threshold 138
Figure 46 3, 6, and 9 nm Radius around the Runway Threshold 140
Figure 47 Measurements at 3, 6, and 9 NM from the Runway Threshold 141
Figure 48 Standard Terminal Arrivals: SKILS FIVE (RNAV) 143
Figure 49 RNAV (GPS) RWY 19 144
Figure 50 Historical Flight Tracks with Allocated Waypoints 145
Figure 51 Course Change Segments 147
Figure 52 Level and Speed Change Segments 147
Figure 53 Summary for Historical Flight Track Data 148
Figure 54 Train and Seed Tracks 152
Figure 55 Selected Seed Tracks 153
Figure 56 GPR for Change in Latitude in Model 1A 160
Figure 57 GPR for Change in Longitude in Model 1B 160
Figure 58 GPR for Change in Barometric Altitude in Model 1C 161
Figure 59 GPR for Change in Ground Speed in Model 1D 161
Figure 60 GPR for Latitude in Model 1A 164
Figure 61 GPR for Longitude in Model 1B 164
Figure 62 GPR for Flight Tracks in Model 1A and 1B 165
Figure 63 GPR for Barometric Altitude in Model 1C 165
Figure 64 GPR for Ground Speed in Model 1D 166
Figure 65 GPR for Flight Tracks in Model 1A and 1B (Batch = 5) 168
Figure 66 GPR for Barometric Altitude in Model 1C (Batch = 5) 168
Figure 67 GPR for ground Speed in Model 1D (Batch = 5) 169
Figure 68 GPR for Flight Tracks in Model 1A and 1B (Batch = 10) 169
Figure 69 GPR for Barometric Altitude in Model 1C (Batch = 10) 170
Figure 70 GPR for Ground Speed in Model 1D (Batch = 10) 170
Figure 71 Synthetic Flight Tracks from Model 1 172
Figure 72 Along Track Distance Distribution Comparison in Model 1 174
Figure 73 Distribution for Measurements at 3, 6, and 9 NM in Model 1 176
Figure 74 Distribution for Individual Flight Tracks Characteristics in Model 1 176
Figure 75 Synthetic Flight Tracks with Waypoints from Published Arrival Procedures 177
Figure 76 GPR for Latitude in Absolute Coordinates in Model 2A 184
Figure 77 GPR for Longitude in Absolute Coordinates in Model 2B 184
Figure 78 Synthetic Flight Tracks in Model 2A and 2B 185
Figure 79 GPR with Encoded Multi-Response in Model 3 190

Figure 80 GPR for Heading in Model 3 190
Figure 81 GPR for Distance in Model 3 191
Figure 82 Predicted Flight Tracks with Forward Reconstruction in Model 3 192
Figure 83 Predicted Flight Tracks with Backward Reconstruction in Model 3 193
Figure 84 Synthetic Tracks from GPR with Encoded Multi-Response in Model 3 194
Figure 85 Deep GP Model for Encoded Response 197
Figure 86 Deep GP Model for Heading 198
Figure 87 Deep GP Model for Distance 198
Figure 88 Synthetic Flight Tracks from Deep GP Model 199
Figure 89 GPR Model with Principal Components in Response 206
Figure 90 GPR for Flight Tracks in Model 5 (Batch = 10) 207
Figure 91 GPR for Barometric Altitude in Model 5 (Batch = 10) 207
Figure 92 GPR for Ground Speed in Model 5 (Batch = 10) 208
Figure 93 Synthetic Flight Tracks from Model 5 210
Figure 94 Along Track Distance Distribution Comparison in Model 5 (Batch = 10) 212
Figure 95 Distributions for Measurements at 3, 6, and 9 NM in Model 5 (Batch = 10) 213
Figure 96 Synthetic Flight Tracks from Model 5 with Waypoints 215
Figure 97 Distributions for Individual Track Characteristics in Model 5 (Batch = 10) . 216
Figure 98 Snapshot of Collected Historical Data 240
Figure 99 Number of Collected Flight Tracks per Month 241
Figure 100 Number of Collected Flight Tracks per Day 242
Figure 101 Along Track Distance Across All Flight Tracks 243
Figure 102 Lateral Dispersion at Waypoints 245
Figure 103 Vertical Dispersion at Waypoints 246
Figure 104 Speed Dispersion at Waypoints 247
Figure 105 GPR for Flight Tracks in Model 5 (Batch = 5) 248
Figure 106 GPR for Barometric Altitude in Model 5 (Batch = 5) 249
Figure 107 GPR for Ground Speed in Model 5 (Batch = 5) 249
Figure 108 Along Track Distance Distribution Comparison in Model 5 (Batch = 5) 250
Figure 109 Distributions for Measurements at 3, 6, and 9 NM in Model 5 (Batch = 5) 251
Figure 110 Distributions for Individual Track Characteristics in Model 5 (Batch = 5) . 251
Figure 111 GPR for Flight Tracks in Model 5 (Batch = 25) 252
Figure 112 GPR for Barometric Altitude in Model 5 (Batch = 25) 253
Figure 113 GPR for Ground Speed in Model 5 (Batch = 25) 253
Figure 114 Along Track Distance Distribution Comparison in Model 5 (Batch = 25) .. 254
Figure 115 Distributions for Measurements at 3, 6, and 9 NM in Model 5 (Batch=25) 255
Figure 116 Distributions for Individual Track Characteristics in Model 5 (Batch=25) . 255

LIST OF EQUATIONS

Equation Page
Equation 1 Collision Probability 8
Equation 2 Aircraft Position 23
Equation 3 Aircraft Velocity 23
Equation 4 Reconstruction Loss Function 34
Equation 5 KL Divergence Function 34
Equation 6 VAE Loss Function 35
Equation 7 GANs MiniMax Game 38
Equation 8 Definition for Gaussian Process 42
Equation 9 Complete Definition for Gaussian Process 51
Equation 10 Covariance Function 52
Equation 11 RBF Kernel 54
Equation 12 RBF Covariance Matrix 56
Equation 13 RBF Log Marginal Likelihood Function 56
Equation 14 Matérn Kernel 57
Equation 15 Linear Kernel 60
Equation 16 Periodic Kernel 61
Equation 17 Bayes' Theorem 64
Equation 18 Posterior Distribution 64
Equation 19 Deep Gaussian Process Definition 68
Equation 20 Patch-response Function 77
Equation 21 Overall Patch-response Function 77
Equation 22 Multi-output Vector 78
Equation 23 Covariance Function between Outputs 79
Equation 24 Great-Circle Distance 85
Equation 25 Equations for New Standardized Point 86
Equation 26 Response Restructuring 156
Equation 27 Encoded Value for Heading and Distance 187
Equation 28 Example for Encoded Value for Heading and Distance 187
Equation 29 Custom Kernel Function for Model 3 188
Equation 30 Heading Extraction 188
Equation 31 Distance Extraction 188
Equation 32 Example for Decoded Heading and Distance Values 189
Equation 33 Latitude and Longitude 191
Equation 34 Custom Kernel Function for Model 5 204

LIST OF ABBREVIATIONS

Air Traffic Control ..... ATC
Air Traffic Management ..... ATM
Along Track Distance ..... ATD
Application Programming Interface ..... API
Area Navigation ..... RNAV
Artificial Intelligence ..... AI
Autoencoder ..... AE
Automatic Dependent Surveillance-Broadcast ..... ADS-B
Automatic Relevance Determination ..... ARD
Bayesian Belief Network ..... BBN
Collision Risk Analysis ..... CRA
Collision Risk Model ..... CRM
Continuous Ranked Probability Score ..... CRPS
Cumulative Distribution Function ..... CDF
Dallas-Fort Worth International Airport ..... KDFW
Deep Learning ..... DL
Density-Based Spatial Clustering of Applications with Noise ..... DBSCAN
Departure Procedure ..... DP
Electric Vertical Takeoff and Landing ..... eVTOL
Event Sequence Diagram ..... ESD
Federal Aviation Administration ..... FAA
Final Approach Fix ..... FAF
Four-Dimensional 4-D
Gaussian Process ..... GP
Gaussian Process Classification ..... GPC
Gaussian Process for Dynamic Systems ..... GPDS
Gaussian Process Regression ..... GPR
Generative Adversarial Network ..... GAN
Gigabyte ..... GB
Global Positioning System ..... GPS
Graphical Processing Unit ..... GPU
Hierarchical Variational Autoencoders ..... HVAE
Instrument Approach Procedure ..... IAP
Instrument Flight Rules ..... IFR
Instrument Landing System ..... ILS
International Air Transport Association ..... IATA

International Civil Aviation Organization ..... ICAO
Jensen–Shannon Divergence ..... JSD
Kolmogorov–Smirnov ..... KS
Kullback-Leibler ..... KL
Machine Learning ..... ML
Mean Absolute Error ..... MAE
Mean Squared Error ..... MSE
National Airspace System ..... NAS
National Transportation Safety Board ..... NTSB
Nautical Miles ..... nm
Partial Differential Equation ..... PDE
Performance-Based Navigation ..... PBN
Physics-Informed Neural Network ..... PINN
Principal Component Analysis ..... PCA
Principal Component ..... PC
Probabilistic Risk Assessment ..... PRA
Probability Density Function ..... PDF
Radial Basis Function ..... RBF
Recurrent Neural Network ..... PredRNN
Required Navigation Performance ..... RNP
Ronald Reagan Washington National Airport ..... KDCA, DCA
Root Mean Squared Error ..... RMSE
Runway ..... RWY
Standard Terminal Arrival Route ..... STAR
Structured Query Language ..... SQL
Support Vector Machines ..... SVM
System of Systems ..... SoS
Target Level of Safety ..... TLS
Terminal Maneuvering Area ..... TMA
Three-Dimensional 3-D
Total System Error ..... TSE
Total Variational Distance ..... TVD
United States ..... U.S.
Unmanned Aerial Vehicle ..... UAV
Variational Autoencoder ..... VAE
Visual Flight Rules ..... VFR
Wasserstein Autoencoder ..... WAE

ABSTRACT

GENERATING SYNTHETIC FLIGHT TRACKS FOR COLLISION RISK MODELING USING GAUSSIAN PROCESSES

Oleksandra Snisarevska, PhD

George Mason University, 2026

Dissertation Director: Dr. Jie Xu, Co-Director: Dr. Lance Sherry

Introducing emerging technologies into the National Airspace System (NAS) requires comprehensive risk analysis to meet a Target Level of Safety (TLS). The risk analysis aims to estimate the likelihood of rare events, such as air-to-air collisions with TLS of 1×10^{-9} failures per flight hour, using various analytical methods.

Collision Risk Modeling (CRM) comprises a set of tools used by regulatory agencies, such as the Federal Aviation Administration (FAA), to estimate the risk of collisions between pairs of flight trajectories. While CRM requires a substantial number of input trajectories to meet TLS, the data for new developments may be nonexistent or highly limited. The use of Artificial Intelligence (AI)-based methods can alleviate the challenge of limited real data by generating an abundance of synthetic flight trajectories tailored to a specific study. However, the synthetic solution must reflect the realistic aerodynamic characteristics of actual flight trajectories to be incorporated in the risk

analysis. This research examines the applicability and capabilities of Gaussian Process (GP) methods to generate valid synthetic flight tracks.

A conventional GP model with a continuous output, called Gaussian Process Regression (GPR), computes predictions with associated uncertainty estimates for a single target response. Several independent GPR models are required to generate synthetic parameters for a multi-dimensional flight track. As a result, the synthetic solution lacks dependencies among generated outputs. Alternatively, generating flight track parameters in a dependent manner requires transformation of the response data to combine multiple outputs into a single feature. However, the complexities introduced into the response data structure tend to obscure the GPR model, leading to reduced performance. The resulting model meets the required threshold of realism in the synthetic solution by incorporating dependencies between outputs, while adhering to GP mathematical principles by producing a single encoded multi-response function.

This dissertation presents a methodology for generating synthetic flight tracks in a case study using GPR. The case study focused on historical arrival procedures for Runway 19 at Ronald Reagan Washington National Airport (DCA). The GPR models, trained on historical data, generated synthetic parameters for a representative new procedure in the NAS. The results are presented by generating synthetic flight tracks from different GPR models and their variants, such as Deep GP, with distinct underlying functions and transformed response structures. The candidate synthetic solutions were evaluated for realism by examining variance between historical and generated data distributions.

CHAPTER ONE: INTRODUCTION

1.1 DOMAIN

1.1.1 National Airspace System

The primary challenge in introducing new technologies or operational concepts into the National Airspace System (NAS) of the United States (U.S.) is driven by its inherent safety considerations [1]. The NAS enables domestic and oceanic operations, including civil and military aircraft, in both controlled and uncontrolled airspace under the authority of the Federal Aviation Administration (FAA) [2]. The system consists of multiple airspace classes with varying flight-level requirements, where flight rules are defined according to air traffic constraints and meteorological conditions, such as Instrument Flight Rules (IFR) and Visual Flight Rules (VFR) [3].

The NAS is an example of a socio-economic-technological network with high safety requirements, rules, and regulations. The system integrates aircraft technology, air navigation aids, surveillance systems, communication channels, airport infrastructure, and landing sites, including surrounding constraints such as mountains or obstacles near runways. From an operational standpoint, the system includes a wide range of users and stakeholders, including pilots, Air Traffic Controllers (ATC), airport and airline authorities, maintenance personnel, and the growing presence of Unmanned Aerial Vehicle (UAV) operators.

From the Systems Engineering perspective, the NAS is an example of a complex system of systems (SoS) [4] comprising numerous geographically dispersed component systems, each with operational and managerial independence. The components work together to enable emergent behavior and achieve a common mission or functionality, which individual systems cannot accomplish alone. The integrated network of SoS forms interdependent patterns within the system, such as air traffic flow corridors. Safe and efficient operation within the system depends on the interaction of technology, human performance, regulatory oversight, and procedural compliance.

1.1.2 Target Level of Safety

A Target Level of Safety (TLS) is a safety concept that defines an acceptable level of risk for aviation operations. TLS is employed to demonstrate conformity with acceptable and tolerable risk levels for emerging systems and technologies in the NAS. Regulatory authorities, including the FAA and the International Civil Aviation Organization (ICAO), use TLS as a safety performance standard during certification and operational approval processes.

TLS is used to quantify the minimum safety threshold in terms of the probability of catastrophic events that could lead to fatalities or significant aircraft damage, during both airborne and ground operations [5]. TLS is defined for specific flight conditions and system operations based on analytical models of the associated system and functional hazards [6]. The target values vary based on the severity and likelihood of potential failures introduced by the new developments into the NAS.

Historically, the acceptable TLS in the NAS has been defined as 1×10^{-7} hazardous aircraft system failures per flight hour [7]. In addition, regulatory agencies arbitrarily decided that 100 additional instances could lead to catastrophic failures [8]. Although the accepted TLS varies between 1×10^{-7} and 1×10^{-9} failures per flight hour [7], for example, in [9], at least 5×10^{9} data are required to demonstrate a loss of vertical separation between aircraft. The safety considerations include failure propagation effects, common-cause failures, lack of independence between contributing components, and the allocation of system safety objectives across multiple systems. In such cases, lower probability targets may be required to ensure that the overall system safety objectives are met.

1.1.3 Rare Events in Aviation

Rare events in aviation are infrequent occurrences that have the potential to cause severe accidents [10], [11]. Rare events arise from a combination of active and latent failure conditions in the system, including unusual aircraft equipment malfunctions, mechanical failures, severe weather phenomena, and unexpected human errors, rather than a single-point failure [12]. These failures may lead to hazardous and catastrophic system or human-factor failures, such as loss of situation awareness, loss of aircraft control, runway incursions, and mid-air collisions.

Given their low likelihood of occurrence and subsequent scarcity of observational data, aviation safety analysis relies heavily on model-based and mathematical approaches to assess risks associated with rare events. The risk analysis aims to estimate the likelihood and potential consequences of extremely unlikely failures. It employs various

analytical methods, such as Probabilistic Risk Assessment (PRA), Bayesian models, and scenario-based simulations. For example, the ICAO's Collision Risk Model (CRM) [13] evaluates the probability of mid-air collisions by modeling rare deviations in aircraft navigation performance and surveillance accuracy.

Rare events directly influence the definition of the TLS of the proposed changes in the NAS during the regulatory review and certification. Demonstrating compliance with these safety targets requires extensive data, conservative assumptions, and analytical evidence that rare events are adequately captured and their effects are properly mitigated.

This dissertation is organized as follows. Chapter One provides an introduction. Chapters Two and Three review the literature on collision risk modeling and Gaussian Processes, respectively. Chapter Four describes the selected methodology. Chapter Five discusses case studies and results. Chapter Six provides conclusions and summarizes future work.

This Chapter described the domain in Section 1.1. Section 1.2 provides the scope. Section 1.3 states the problem description. Section 1.4 describes the research approach. Section 1.5 outlines the research objective. Section 1.6 addresses the research questions. Section 1.7 lists the unique contributions.

1.2 SCOPE

1.2.1 Air-to-Air Collisions

A loss of vertical or lateral separation between aircraft contributes to air collisions. An air-to-air or, alternatively, midair collision occurs when two or more aircraft are on a path in the same space at the same time. Midair collisions are extremely

rare, averaging eight accidents per year in the U.S. [14]. Although the probability of such an event is very low and considered extremely improbable, its severity is categorized as catastrophic, because it leads to aircraft hull loss and fatalities [3]. Serious and fatal injuries constitute 68% of all collision accidents in the National Transportation Safety Board (NTSB) accident data for the U.S. [14].

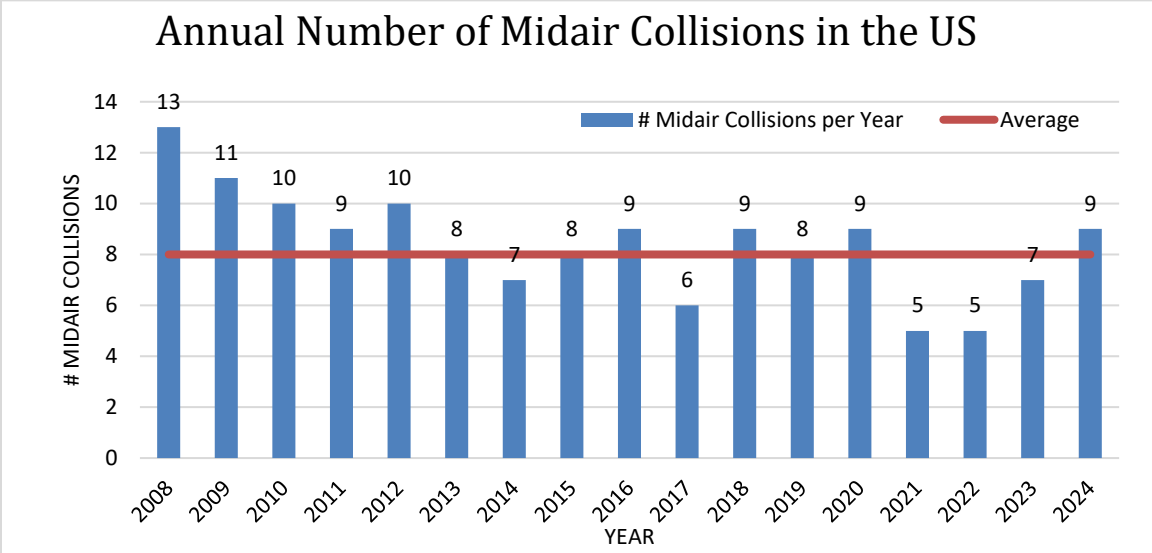


Figure 1 Midair Collisions in the US from 2008 to 2024 (NTSB [14])

In line with the FAA's risk matrix for commercial aviation [15], TLS for pairs of aircraft trajectories on a collision path is defined as less than 1×10^{-9} catastrophic failures per flight hour (Figure 2).

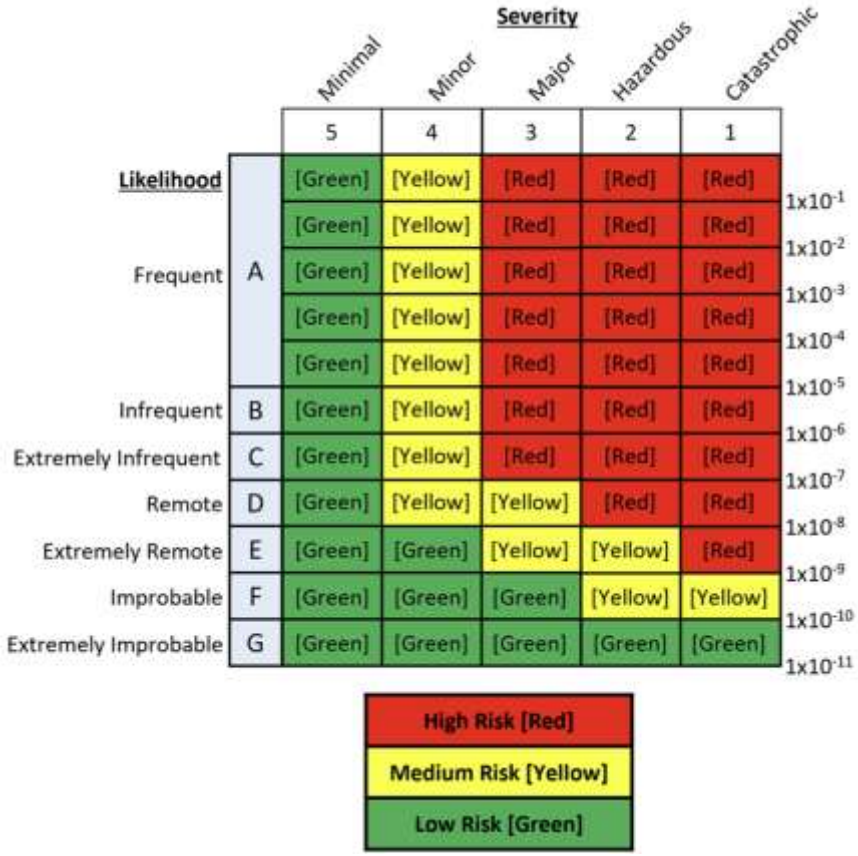


Figure 2 FAA's Risk Matrix for Commercial Aviation [15]

According to ICAO [16], the approach and landing phases of flight are most vulnerable to collisions because aircraft are less maneuverable at lower speeds and altitudes, and because of surrounding environmental constraints, such as mountainous terrain or towers. As described in the FAA library resources [17], midair collisions are more likely to occur in congested airspace and in areas of greatest traffic concentration, such as terminal areas around airports, where many aircraft are situated close to each other.

Notably, in accordance with the NTSB accident data [14] between 2008 and 2024, one-third of midair collisions in the U.S. occurred en-route, while 60% of collision events were recorded during maneuvering and approach phases of flight (Figure 3).

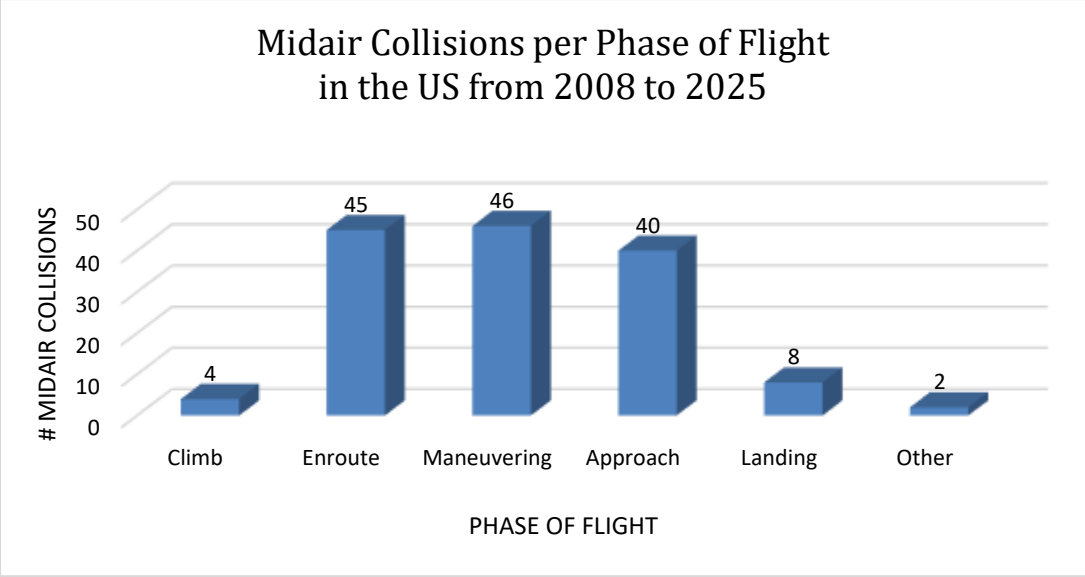


Figure 3 Midair Collisions per Flight Phase in the US from 2008 to 2024 (NTSB [14])

1.2.2 Collision Risk Modeling

Since the middle of the last century, regulatory agencies have driven standardization efforts to ensure safe airport and aircraft operations. In 1980, the U.S. FAA, in conjunction with the ICAO [13], introduced the Collision Risk Model (CRM) to analyze clearance and safe separation between aircraft. The original model was developed to define a cleared airport area required for the aircraft to perform the Instrument Landing System (ILS) approach [18].

Today, Collision Risk Modeling is a set of tools and analytical techniques used to estimate collision risk in aerospace [19]. Regulatory safety agencies use these tools to estimate the risk of an aircraft collision with another aircraft or an obstacle, both in the air and on the ground. In the ICAO manuals [9], risk models account for aircraft on the same or intersecting tracks and the effects of large deviations on the total risk in aerospace. CRMs are developed to determine the likelihood of an aircraft collision with another aircraft or an object during ground (e.g., taxiing) and airborne (e.g., cruise, approach) operations [16].

Collision risk constitutes the predicted severity and likelihood of a potential hazard [20]. According to the ICAO guidance [21], technical collision risk can be related to a loss of vertical or horizontal separation. Quantitatively, the collision risk is expressed in units of probability. Given the probability of a hazardous event, the probability of a collision can be expressed as follows [22]:

Equation 1 Collision Probability

$$Pr\{collision\} = Pr\{event\} \times Pr\{collision \mid event\},$$

where the event corresponds to a loss of vertical or lateral separation.

CRMs are used to perform safety analyses for new development considerations in the NAS. Examples of new developments include the following:

- Emerging technologies, such as new aircraft types, new onboard systems, or expansion of performance envelopes
- New airspace design or weather-driven re-routing strategies leading to changes in separation minima

- Modifications to runway configurations in existing airports affecting departing and arrival procedures near the terminal area
- Traffic growth projections
- Growing presence of emerging actors, such as Unmanned Air Vehicles (UAVs) and electric Vertical Takeoff and Landing (eVTOL) aircraft

In summary, the embedded Collision Risk Analysis (CRA) within CRM is designed to prevent underestimating the frequency of extreme deviations that could lead to catastrophic outcomes [23]. Hence, CRA is valuable for estimating risk levels in emerging systems *before* they are introduced to the existing system.

1.3 Problem Description

The introduction of new or modified procedures to the NAS requires *a very large number* of flight trajectories in the CRM to satisfy TLS for rare events in aviation, such as air-to-air collisions. In combination with the Statistical Rule of Three [24], risk analysis requires three times the amount of data to meet the established safety threshold of 3×10^{-9}, resulting in three billion pairs of flight trajectories. However, historical data for new developments may be nonexistent or highly limited.

For example, Dallas-Fort Worth International Airport (ICAO: KDFW) in Dallas, Texas, recorded 743,203 operations in 2024 [25]. This figure comprises mainly Air Carrier operations (97% of all departures and arrivals) but also includes Air Taxi, General Aviation, and Military operations. Recognizing rare events at a level of 1×10^{-9} would require at least $\frac{10^9}{743{,}203} = 1{,}345$ years of statistics to conduct a safety analysis for a new

procedure at KDFW, which is unreasonable. On top of that, applying the statistical "Rule of Three" [24] and considering pairs of flight trajectories results in a safety level of 6×10^{-9}, which requires $\frac{6\times10^{9}}{743{,}203} = 8{,}073$ years of data collection to perform a risk analysis. In addition, these figures do not include new development considerations.

Flight trajectories tailored for the new environment may be obtained from simulated data [19]. However, flight tracks inherit the restrictive assumptions and system conditions of the simulation model. Simulations can only reflect the assumptions built into them, whereas real-world operations involve many sources of variability and uncertainty. In addition, simulating the vast flight track dataset, which includes rare events with potentially catastrophic outcomes, such as mid-air collisions, requires substantial computational resources and time.

A mix of archived and simulated data yields a more reliable dataset of blended flight tracks. Such an approach alleviates some modeling restrictions on trajectory behavior but fails to incorporate the outlined changes to the NAS effectively.

The limited availability of flight trajectory data in CRMs leads to the definition of the first problem:

> **<u>Problem #1</u>. CRM requires a very large sample size of flight trajectories to perform a comprehensive safety analysis for emerging systems in the NAS. However, data on new developments may be nonexistent, highly limited, or difficult to obtain within a reasonable time.**

As a proposed solution, flight trajectories can be artificially generated using novel mechanisms and advanced algorithms. The use of Artificial Intelligence (AI)-based

approaches can alleviate the challenge posed by the lack of real data by generating an abundance of synthetic flight tracks tailored to a specific study. The synthetic solution learns flight properties from available historical data and applies advanced algorithms to create entirely new tracks. Then, the synthetic data can be incorporated into a separate framework to estimate the effects of the proposed concepts and changes in the NAS.

Generative AI applications are already being successfully employed and are expected to expand even further within the aviation sociotechnical domain [26]. Recent advances in Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL) offer new perspectives and expand the set of tools available for estimating safety parameters. For example, [27] provides a high-level summary of AI/ML/DL practices in the aviation domain, highlighting their growing role in enhancing operational decision-making.

Hypothetically, synthetic flight trajectories must accurately reflect the aerodynamic characteristics of real ones. In addition to the aerodynamic data quality requirement, the synthetic data shall preserve variance and diversity in the resulting solution to enable a comprehensive risk analysis of the proposed changes. The variance in synthetic flight tracks enables the capture of potentially critical and catastrophic deviations, supplementing the safety evaluation [28]. This leads to the definition of the second problem:

> **<u>Problem #2</u>. Synthetic flight trajectories must reflect realistic aerodynamic characteristics, while preserving variety in the resulting solution, to be suitable for collision risk analysis.**

Balancing a realistic flight trajectory profile with reasonable levels of variance entails defining an acceptable threshold in the validation criteria. Conventional verification metrics alone are often insufficient to guarantee the practical validity of solutions produced by AI-based systems. Very often, the results of these algorithms meet verification metrics but provide little value because they fail validation checks. For example, the vertical profile of synthetic flight trajectories may exhibit unrealistic rapid changes in altitudes. Therefore, subject-matter expertise is required to validate the resulting solution rigorously. The gap between verification and validation of the resulting synthetic solutions highlights the need for more robust evaluation frameworks specifically tailored to AI-driven applications [29].

Furthermore, although predominant and practical, generative AI implementations are frequently obscured, and their results can be unpredictable and lack transparency and explainability [29]. Notably, many AI/ML models operate as "black-box" systems, where the underlying decision-making processes are not readily interpretable or transparent. This lack of explainability poses significant concerns in safety-critical domains such as aviation, where trust, accountability, and regulatory compliance are essential. In addition, regulatory agencies have provided little guidance on compliance prospects for determinations involving AI/ML [30], [31]. This leads to the definition of the third problem:

<u>Problem #3</u>. The synthetic solution shall incorporate transparency between modeled assumptions and results.

1.4 APPROACH

This thesis examines the applicability and capabilities of Gaussian Process methods to generate aerodynamically valid synthetic flight tracks. The synthetic flight trajectories are derived from learning flight properties from actual flight trajectories and applying Gaussian Processes to generate entirely new tracks for new developments in the NAS. Synthetic solutions generated by Gaussian Processes aim to resemble the properties of the historical flight trajectories and the bias present in the real data. In addition to realistic flight trajectory characteristics, this approach enables the generation of billions of new flight trajectories in a reasonable time [32].

The Gaussian Process method incorporates several properties that are beneficial to flight trajectory modeling.

1. As a non-parametric framework, Gaussian Process algorithms learn directly from empirical data rather than fitting a fixed equation or function [33] (e.g., unlike linear or logistic regression). The non-parametric property enables the capture of the aerodynamic characteristics of actual flight trajectories by incorporating their behavior into predictive models.
2. As a probabilistic framework, Gaussian Processes provide flexibility by generating averaged prediction estimates with corresponding uncertainty coverage [34]. The probabilistic outputs provide a large search space with preserved variance in the predicted parameters. This enables the capture of outliers and the presentation of a complete picture of synthetic trajectory behavior for comprehensive safety analysis.

3. As a data-efficient framework, Gaussian Processes enable the generation of reasonable predictions from a small number of inputs in the presence of high levels of noise. The data-efficiency property is useful when dealing with inherent noise in actual flight trajectories. This property also allows generating trajectory predictions based on the limited availability of historical data in a reasonable time.
4. As a mathematically interpretable framework, Gaussian Processes comprise explainable parametric functions that preserve transparency between the underlying model structure and the generated predictions [35]. Unlike other generating mechanisms, this algorithm's interpretability ensures reproducible results. This facilitates understanding of the relationship between inputs and predictions in the model, which regulators require for certification and approval decisions.

1.5 Research Objective

This research defines a process of generating a very large number of diverse aircraft trajectories suitable for collision risk modeling. The selected methodology employs Gaussian Processes-based methods to generate synthetic flight tracks tailored for a specified study. The synthetic solution serves as a crucial input to collision risk modeling; thus, it requires a certain level of assurance in the quality and fidelity of the synthetic data.

The overall research objective is to generate realistic, aerodynamically feasible flight trajectories suitable for a comprehensive collision risk analysis for a new procedure, in a reasonable amount of time. The objective breakdown is as follows:

1. Define a suitable Gaussian Process-based method, a set of properties, and a solution space required to generate billions of flight trajectories in a reasonable time.
2. Adapt a Gaussian Process-based methodology to generate a variety of realistic multi-dimensional flight tracks.
3. Derive qualitative and quantitative validation criteria to evaluate synthetic flight tracks for realism from an aerodynamic perspective.

1.6 Research Questions

This section describes research questions related to synthetic flight track generation using Gaussian Processes, along with the associated tasks and activities for their execution.

Research Question #1. How to obtain a large sample size of flight trajectories for a new procedure in the NAS, in a reasonable time?

- Analyze how long it would take to collect historical data to obtain flight trajectories and summarize its limitations.
- Explore simulation methods for generating flight trajectories and summarize their strengths and limitations.
- Explore generative AI methods for generating synthetic flight trajectories and summarize their strengths and limitations.

<u>Research Question #2</u>. How to obtain realistic synthetic flight trajectories using Gaussian Process-based methods?

- Identify strengths and weaknesses in Gaussian Process methods applicable to flight trajectory generation process.
- Develop a framework to incorporate Gaussian Process methods into the flight track data generation algorithm.
- Design a baseline algorithm with a single flight track parameter as an output.
- Generate multi-dimensional synthetic flight track characteristics.
- Preserve dependencies between generated flight-track output parameters.
- Evaluate synthetic flight tracks for realism from an aerodynamic perspective.
- Apply mitigation strategies if physically implausible trajectories are present.

<u>Research Question #3</u>. How to preserve variance and diversity in the synthetic flight tracks while limiting aerodynamically unreasonable samples at the same time?

- Identify mechanisms to extract synthetic flight tracks from the Gaussian Process solution controllably.
- Apply variance control mechanisms in the resulting synthetic solutions by expanding or narrowing the acceptable intervals for generated samples.
- Perform sensitivity analysis on effects of different variance thresholds.
- Derive qualitative and quantitative criteria to evaluate the performance of synthetic flight tracks.

Graphically, the research questions are summarized in Figure 4. When the synthetic solution consists of a large number of realistic, diverse trajectory profiles, it can support the risk analysis within CRM.

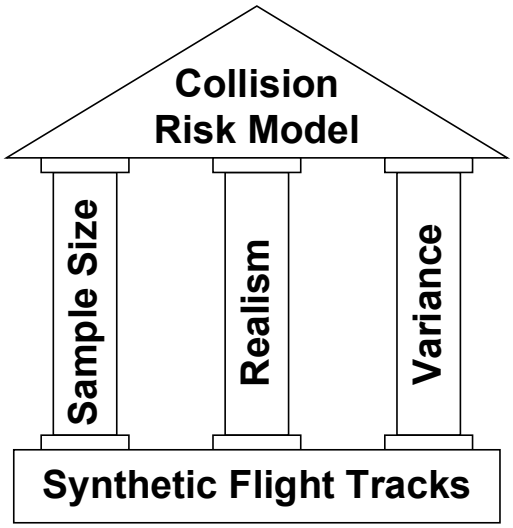


Figure 4 Three Pillars in Synthetic Flight Tracks

1.7 UNIQUE CONTRIBUTION

The unique contributions of this research from both methodological and practical perspectives to the fields of Systems Engineering and Operations Research, as well as AI/ML adaptation in aerospace systems, can be summarized as follows:

1. Developed a reusable modeling framework for training, generating, and validating synthetic flight trajectories.
2. Developed a method for generating multi-dimensional flight trajectories in a dependent manner via the target function transformation in Gaussian Processes.
3. Developed a method for generating flight trajectory profiles using Gaussian Processes that reflect aerodynamic properties of actual trajectories and preserve variance required in CRM's inputs.

CHAPTER TWO: LITERATURE REVIEW PART I – STATE-OF-THE-ART COLLISION RISK MODELING

This Chapter comprises a literature review for collision risk modeling in the aerospace domain. Section 2.1 describes air-to-air collision risk modeling. Section 2.3 focuses on flight trajectory models. Section 2.3 addresses gaps in literature.

2.1 AIR-TO-AIR COLLISION RISK MODELING

A risk analysis of air-to-air collisions follows a structured modeling approach. Sherry et al. [32] conducted a comprehensive survey that defined a complete process for Collision Risk Model (CRM) cycle. The authors reviewed more than 70 scientific and engineering papers describing various types of operational scenarios of CRMs, including the following:

- Different types of approaches on parallel and intersecting runway configurations,
- Intersecting approach and departure trajectories,
- Missed approaches,
- Airport terminal area sequencing and spacing,
- Wake vortex encounters.

Based on the survey, the authors proposed a superset of functions required to derive universal CRM. A selection of the key CRM components for the purpose of this research is defined in Figure 5. Other models that are not mentioned but required for a

complete CRM definition include meteorological and wake vortex models, as well as a subset of models necessary to perform Detect and Avoid functions.

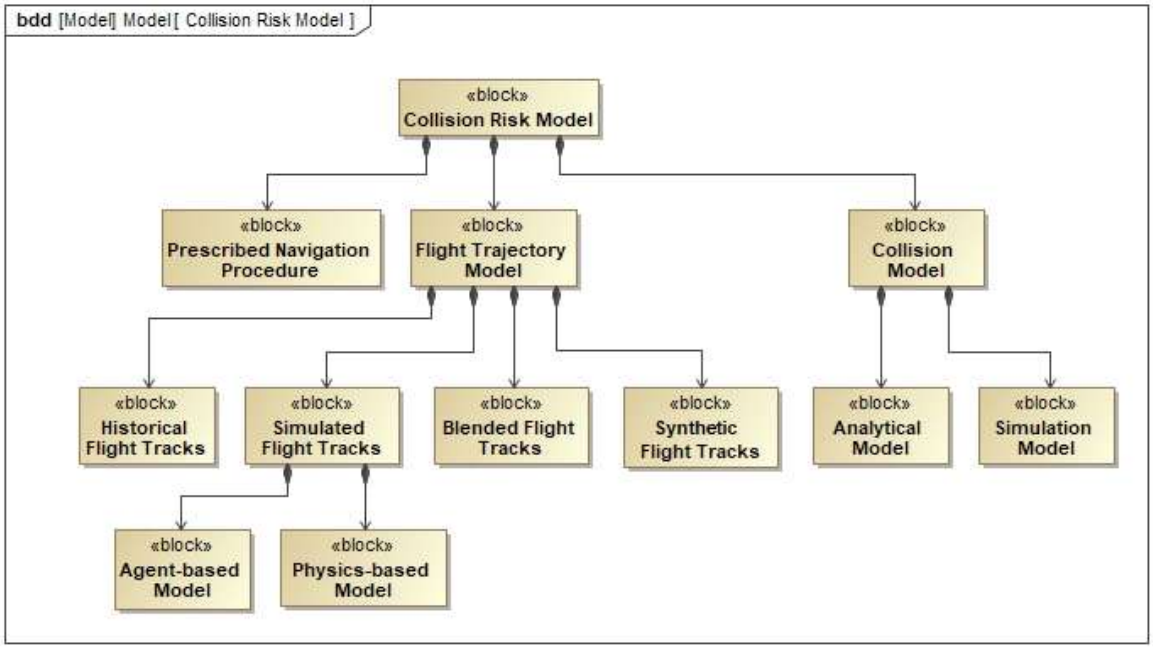


Figure 5 Breakdown of Key CRM Components derived from [32, p. 10]

Prescribed navigation procedure or Required Navigation Performance (RNP) describes a type of ICAO performance-based navigation (PBN) [36] that defines aircraft path in 3-D space. In addition, the RNP type defines the total system error (TSE) comprised of navigation system errors and other technical errors in relation to lateral and longitudinal deviations from the aircraft route [37]. RNP levels encompass departure and approach procedures with specified requirements and restrictions for latitude, longitude, altitude, and velocity parameters.

Standard Terminal Arrival Routes (STARs) and Departure Procedures (DPs) both comprise a prescribed navigation procedure and provide obstacle clearance in high-density traffic terminal area [38]. In the scope of this research, a prescribed navigation procedure formulates requirements for the operating environment and system constraints by defining allowable flow corridors for aircraft paths. This procedure serves as input to other components of the CRM.

A process flow for the CRM's components, starting with the identified prescribed navigation procedure and ending with the risk estimation, is shown in Figure 6. The prescribed navigation procedure is a required input for the generating flight trajectory function. Pairs of flight trajectories must provide a substantial set of data as are necessary for further analysis within the identified prescribed navigation procedure's limits. The flight trajectory model's underlying assumptions, physical properties, sufficient sample size, origin, and transformation significantly impact the overall collision risk estimates. Therefore, to perform a reliable risk assessment, flight trajectories must reflect realistic aerodynamic properties and capture rare events of interest. These verified flight trajectories and their derived distributions ultimately define a consolidated collision model. Next, collision risk can be quantitatively estimated based on the selected collision model.

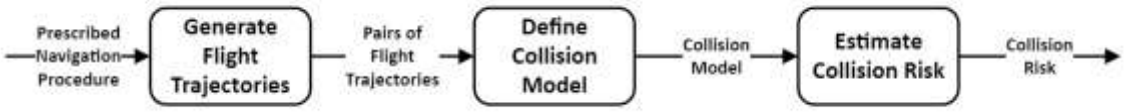


Figure 6 Process Flow for Generic Collision Risk Modeling

The following section describes collision models in detail.

2.1.1 Collision Models

A collision risk estimate can be accounted for in the form of an analytical solution or simulation. Analytical models are described by simple and understandable mathematical equations [39], while the simulation approach relies on advances in computing.

2.1.1.1 Analytical Models

The Reich model is an example of an analytical model. The Reich model was originally published in "Analysis of Long-Range Air Traffic Systems: Separation Standards" [40] and was developed for in-flight separation for transatlantic routes [41]. Although not applicable to the terminal area, the Reich model is valid when specific criteria are met [42], [43]. Those are,

- aircraft traveling on parallel routes and maintaining constant velocity,
- uncertainties in 3-D flight trajectory parameters are independent of each other,
- absence of mitigation strategy in case of a catastrophic event (thus excluding turns or violations to the parallel routes requirement), and
- abundance of data.

This approach applies to a largely homogeneous fleet and data-driven models. The last condition implies a requirement of a sufficient number of flight trajectories to derive probabilistic distribution.

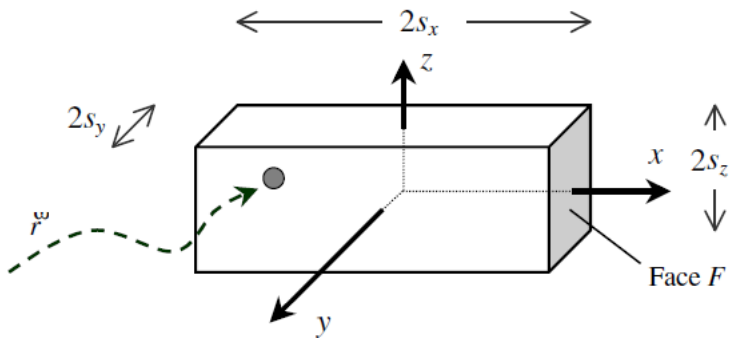


Figure 7 Geometric Representation of the Reich Model [44]

The geometric representation of the Reich model is shown in Figure 7. Here, every aircraft's outer dimensions are summarized by 3-D box parameters, where s_x, s_y, and s_z are along-track length, across-track width, and height, respectively, with a margin. Additionally, the aircraft's i dynamic parameters, such as position $\tilde{r}_i(t)$ and velocity $\tilde{v}_i(t)$ vectors are monitored at a time t. These parameters for a pair of aircraft 1 and 2 are defined as follows:

Equation 2 Aircraft Position

$$\tilde{r}(t) \equiv \left(\tilde{r_x}(t), \tilde{r_y}(t), \tilde{r_z}(t)\right) \equiv \tilde{r_1}(t) - \tilde{r_2}(t),$$

Equation 3 Aircraft Velocity

$$\tilde{v}(t) \equiv \left(\tilde{v_x}(t), \tilde{v_y}(t), \tilde{v_z}(t)\right) \equiv \tilde{v_1}(t) - \tilde{v_2}(t).$$

Hence, the relative position of $\tilde{r}(t) \equiv (0,0,0)$ implies that two aircraft are at the same point in space. A collision occurs when a difference in the relative position for two aircraft becomes too small. Graphically, the collision is recorded when $\tilde{r}(t)$ penetrates the 3-D box of another plane. The Reich model accounts for a probable collision in all three-dimensional space around the aircraft.

The probability of a collision is defined by the joint probability density function (PDF) $f\big(\tilde{r}(t), \tilde{v}(t)\big) \equiv f\big(r_x(t), r_y(t), r_z(t), v_x(t), v_y(t), v_z(t)\big)$ that meets and translates the Reich CRM criteria to the following:

- Uncertainties in the longitudinal x, lateral y, and vertical z dimensions are independent, $f\big(\tilde{r}(t), \tilde{v}(t)\big) = f_x\big(r_x(t), v_x(t)\big) f_y\left(r_y(t), v_y(t)\right) f_z\big(r_z(t), v_z(t)\big)$,

  where f_x, f_y, and f_z are marginal densities of the PDF.
- The constant density within the aircraft's dimensions, $f\big(\tilde{r}(t), \tilde{v}(t)\big) = f\big(0, \tilde{v}(t)\big)$.
- The position and velocity density are independent across all dimensions. For example, in the longitudinal dimensions $f_x\big(r_x(t), v_x(t)\big) = f_{r,x}(r_x) f_{v,x}(v_x)$.

A significant limitation of the Reich CRM is that it lacks input from other external systems or actors such as ATC instructions or pilot's interventive actions [45].

2.1.1.2 Simulation Models

Simulation collision models define a computational framework for risk analysis. Due to recent advances in 3-D technology, it has become significantly easier to present the results in the form of a practical simulation. Simulation models demonstrate detailed, robust, and tangible results for collision risk assessment. However, the model accuracy depends on the overall model fidelity and quality of the input data.

An intuitive example of a simulation-based collision model generates flight trajectories from a prescribed navigation procedure. A modeler may include additional behaviors, use cases, and other details of the operating environment, such as aircraft parameters, weather factors, ATC guidelines, and airport infrastructure restrictions, to expand the simulation profile. However, these models incorporate limitations set by the modeler and may not always provide sufficient insight into rare scenarios.

2.2 Flight Trajectory Models

Pairs of flight trajectories are required to calculate the probability of an air-to-air collision risk between two aircraft on a colliding path. Multiple pairs of flight trajectories constitute a flight trajectory model. CRMs must consume a considerably large number of pairs of flight trajectories to account for the potential presence of rare events in the data that may lead to air-to-air collisions. Although the industry's safety goal defines the TLS in the NAS at 1×10^{-9} per event [9], [46], the statistical "Rule of Three" [24] requires three times the amount of data to meet the established threshold, thus resulting in 3×10^{9} of data for analysis. Hence, CRMs require twice as much data for pairs of flight trajectories (i.e., 6×10^{9}) to meet the safety standards of the NAS environment.

Pairs of flight trajectories, tailored for the new environment to account for proposed concepts and new development in the NAS, may be obtained from simulated data, synthetic data, and combinations of those with historical data [32].

2.2.1 Historical Flight Tracks

Historical flight data collected, for example, from Automatic Dependent Surveillance-Broadcast (ADS-B) or radar sources can be used in collision risk modeling

analysis. For instance, a study based on the ADS-B recorded data alone may provide insights into complex data patterns and anomaly detection [27]. However, historical flight tracks are derived from a limited dataset; hence, observing rare events of interest, such as air-to-air collisions and other accidents, is insufficient.

Noticeably, historical records alone do not reflect the proposed changes that need to be assessed by CRM. These trajectories are better suited for studying anomaly and significant event detection [47], which may also serve as important knowledge for the analysis. In addition to pattern recognition, historical data provide the knowledge needed to derive trajectory profiles using simulations or generative AI applications, as described in the following subsections.

2.2.2 Simulated Flight Tracks

Flight trajectories can be populated using simulation methods, offering several advantages for the CRM domain [32]. Simulation mechanisms can generate millions of flight trajectories, yielding a much larger data sample size than historical records. Simulations also include rare event considerations, expanding the sample size with suitable trajectories for risk analysis. In addition to the generated sample size, specific modeled scenarios can incorporate proposed changes in the NAS. However, the flight trajectories obtained from simulations exhibit certain limitations.

First, the resulting flight tracks inherit modeled restrictions derived from a predefined set of assumptions. The modeled assumptions include aircraft accident scenarios [48], aircraft performance models [49], [50], aerodynamic profiles [51], and pilot behavior rules, such as reaction time and airline compliance rates [52]. Furthermore,

simulations include a predefined traffic flow logic [52], along with underlying conflict-resolution algorithms and ATC intervention scenarios. Next, a simulated approach incorporates limited sources of uncertainty, such as wind fields, in the resulting tracks. Oversimplified or unrealistic assumptions result in an underestimation or misrepresentation of collision risk. Lastly, and most importantly, simulations are computationally expensive [52], necessitating many iterations and requiring high-performance computing clusters and long runtimes for full-airspace scenarios. The computational cost grows nonlinearly with model complexity and flight trajectory density.

As a model-driven approach, the commonly used methodology to derive simulated flight trajectories can be grouped into two major categories: physics-based and agent-based models [53].

2.2.2.1 Physics-based Models

Physics-based models for flight tracks rely on fundamental laws of physics to simulate realistic aircraft motion governed by principles of kinematics, aerodynamics, or control theory. In physics-based models, the aircraft constitutes a point mass, while Newton's law is used to connect the inertial acceleration of aircraft mass with the force and thrust produced by its engines [54]. Either ordinary or stochastic differential equations commonly describe these models to generate deterministic or random flight trajectories, respectively. Given the initial aircraft state along with its mass, thrust, position, velocity, and bank angle parameters, as well as meteorological parameters such

as wind velocity and direction, the flight trajectory can be generated over time intervals [55].

The underlying structure of the model utilizes control laws consisting of kinematic equations with a feedback loop. For example [19], [56], a six-dimensional aircraft state space is defined by the following parameters:

- along-track or longitudinal position, $\dot{X} = V\cos(\psi)\cos(\gamma) + w_1$,
- across-track or lateral position, $\dot{Y} = V\sin(\psi)\sin(\gamma) + w_2$,
- altitude, $\dot{h} = V\sin(\gamma) + w_3$,
- airspeed, $\dot{V} = \frac{1}{m}\big((T\cos(\alpha) - D) - mg\sin(\gamma)\big)$,
- heading, $\dot{\psi} = \frac{1}{mV}(L\sin(\phi) + T\sin(\alpha)\sin(\phi)$,
- and flight-path angle, $\dot{\gamma} = \frac{1}{mV}(L + T\sin(\alpha)\cos(\phi) - mg\cos(\gamma))$,

where $T, L,$ and D are thrust, lift, and drag, respectively; $w_1, w_2,$ and w_3 are wind velocities in all three dimensions; α is angle of attack, ϕ is bank angle, m is mass of aircraft, and g is acceleration due to gravity.

Here, parameters for airspeed, altitude, and heading are adjusted by thrust, angle of attack, and bank angle, respectively; and can be parameters for the feedback loop of the continuous control law system. Noteworthy, parameters of the wind model introduce randomness to the model since they are external disturbances.

Although widely prevalent, this approach may be impractical for generating realistic flight trajectories. In this case, because flight trajectories do not incorporate uncertainty parameters other than wind, assumptions about other contributing factors

must be made (e.g., ATC and/or pilot intervention). Additionally, detailed physics-based aircraft parameters are usually not rapidly available due to security measures and proprietary data, hence expanding the list of assumed values and parameters. Moreover, the simulated trajectories usually inherit conditions of the operating environment prior to the change introduction.

2.2.2.2 Agent-based Models

Agent-based models are used in the collision risk modeling by explicitly representing aircraft behavior, operators, or other agents as independent entities that move and interact inside the simulated environment under predefined conditions and rules [57]. For example, in the collision risk context in [57], an agent-based model usually consists of (1) agents, (2) simulated environment, (3) behavioral rules, and (4) underlying interactions between agents.

Agents can be represented by Aircraft flying approach, departure, or en-route trajectories, as well as ground vehicles operating on taxiways or runways. The environment uses simulations for runway configuration, obstacles, terrain, and navigation aids. In addition, weather conditions such as wind speed and direction can be used to introduce uncertainty in the model. Operational constraints such as separation minima, navigation procedures, and ATC procedures also can be added to the modeled environment.

Behavioral Rules can be used to summarize Aircraft performance, and pilot and ATC actions. Response to conflicts as random variations representing human and technical uncertainties can be also incorporated in the model. Agents continuously assess

proximity to other agents or obstacles as a part of predetermined interactions. Incidents detected when separation standards are violated are evaluated. Collisions are recorded when agents occupy the same space at the same time.

Agent-based models account for some of the shortcomings introduced by physics-based models. These computational models are capable of simulating actions and interactions between multiple agents and components in the system. For example, these models can explicitly capture the behavior and sequence of actions performed by pilots or ATC in pre-defined scenarios. However, overall model fidelity strictly depends on the beforehand established scenarios and interaction rules between the agents, assumptions in the modeled environment, and computational resources for the simulated engine.

Other works include focusing on trajectory prediction by hybrid and simultaneous estimation of aircraft state and flight mode based on data and knowledge provided by ATC regulations and filed flight plan [58]. A mix of archived and simulated data yields a much more reliable flight trajectory. These blended flight tracks exhibit realistic properties of the real flight trajectory. This approach alleviates some modeling restrictions but fails to incorporate the proposed changes to the operating environment effectively.

2.2.3 Synthetic Flight Tracks

As an alternative to the model-driven approach, a synthetic solution derives from learning the flight properties from historical data and applying advanced AI/ML algorithms to generate entirely new tracks [32]. As a data-driven approach, the method falls under the umbrella of data augmentation techniques, which involve artificially

generating new data from existing data [59]. This leads to the creation of many variations of historical data and a generalization of previously unseen data.

Generating synthetic flight trajectories involves replicating and mirroring authentic characteristics, properties, and patterns of the actual flight trajectories from the historical dataset [60]. Ideally, this produces reliable and aerodynamically reasonable flight trajectories. The modeled parameters can also effectively capture and reflect a new development being assessed. This approach allows the generation of millions of flight trajectories [32], thus expanding a sample size for further collision risk modeling. However, the synthetic solution may still exhibit some unrealistic physical characteristics, as there is no guarantee that the generated tracks follow the laws of aerodynamics [61].

Recent advances in methods for generating synthetic flight trajectories are summarized in Table 1. The following subsections describe each method and associated papers in detail.

Table 1 Literature Review on Synthetic Flight Trajectories

Method	Paper	Context	Limitations
Variational Autoencoders (VAEs)	Krauth et al. (2023) [62]	4-D synthetic trajectories for Terminal Maneuvering Area (TMA)	(1) Source of uncertainty (2) Absence of randomly generated go-arounds (i.e., rare events)
	Krauth et al. (2024) [63]	VAE with Subset Simulation for rare events	Fidelity in the synthetic results
	Aref et al. (2024) [64]	3-D synthetic trajectories using a single seed track	Limited variability in the synthetic results
	Aref et al. (2024-2) [65]	4-D synthetic trajectories using a single seed track	
	Aref et al. (2025) [66]	4-D synthetic trajectories using multi-seed track	Rare events considerations
Generative Adversarial Networks (GANs)	Ji et al. (2024) [60]	Wasserstein-based GAN for synthetic trajectories using a seed track	Smoothness in the synthetic results
	Wijnands et al. (2024) [67]	TimeGAN for 4-D synthetic trajectories for TMA	(1) A substantial amount of training data is required (2) Absence of go-arounds and holding patterns in the results
Diffusion Models	Yin et al. (2023) [68]	3-D aircraft trajectory prediction for TMA	Absence of rare events and variability in the study
	Yang et al. (2025) [69]	Aircraft trajectory prediction for general aviation in nontowered TMA	Inherit historical data size and variability considerations
Physics-Informed Neural Networks (PINNs)	Ma et al. (2025) [70]	Recurrent Neural Network in combination with a wind field physical model for aircraft route prediction and optimization	A different perspective on route optimization rather than generation
Gaussian Process	Eerland et al. (2016) [71]	Gaussian Process Regression (GPR) for departure procedures	Independent dimensions for the 3-D aircraft trajectory
	Xu et al. (2024) [72]	Gaussian Process for Dynamic Systems for arrival procedures	Unrealistic synthetic tracks
	Snisarevska et al. (2025) [73]	Gaussian Process Regression for arrival procedures with multi-output generation	(1) Fidelity in the results (2) Independent dimensions for the 3-D flight trajectories

2.2.3.1 Variational Autoencoders

Introduced by Kingma and Welling [74], Variational Autoencoder (VAE) is a popular Bayesian inference framework in image recognition problems [75] but is also applicable for generating synthetic flight trajectories [62], [63], [64], [65], [66]. The method belongs to a generative AI family of algorithms and utilizes two connected neural networks: encoder and decoder [76].

In conventional Autoencoders (AEs) [77], the encoding function $f(x)$ reduces the dimensionality of the underlying image or training data x by extracting its features and disregarding noise into the latent vector $z = f(x)$. Then, the decoder $q(z)$ performs the opposite function through input reconstruction by re-mapping the features from the latent vector back to the original units, resulting in an approximation of the data $\bar{x} = q(z)$ (Figure 8). VAEs expand the latent space by introducing a probabilistic framework and encoding similar pixels (i.e., inputs) with corresponding posterior distributions, while allocating inputs with distinctive features by distant posteriors, $z \sim q(z|x)$. VAE's decoding component then samples variables from the latent distribution to reconstruct the input $\bar{x} \sim p(x|z)$. VAEs incorporated variability ensures generalization and flexibility of the resulting probabilistic model (Figure 9).

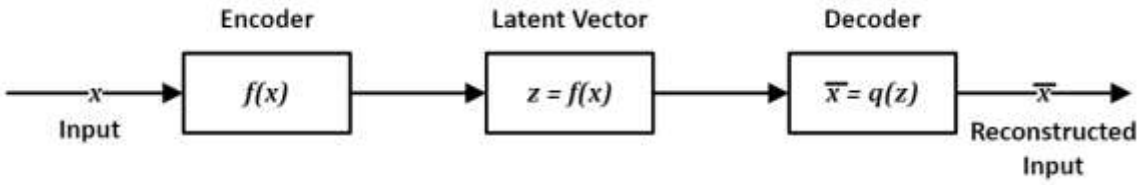


Figure 8 Autoencoder (AE)

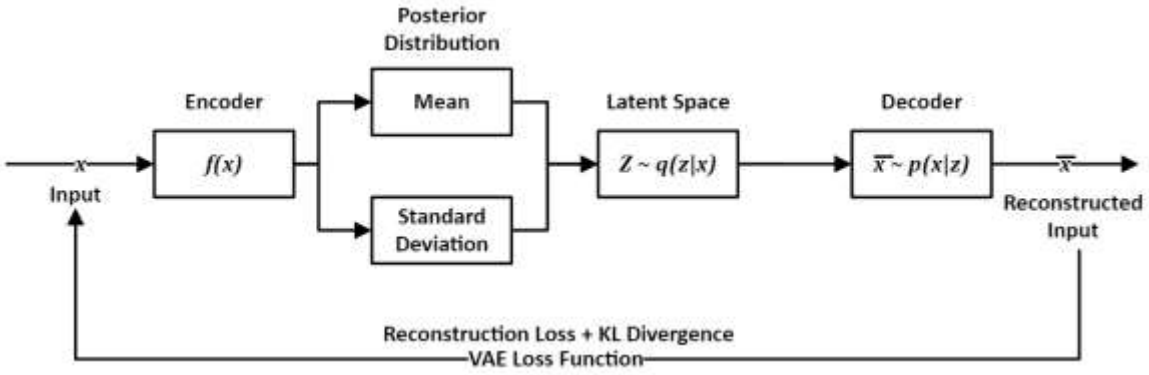


Figure 9 Variational Autoencoder (VAE)

VAE attempts to maximize marginal log-likelihood for each input x to increase the probability of observing the data of the input space. Therefore, the reconstruction loss is a negative expected log-likelihood function for each input x and is denoted as follows:

Equation 4 Reconstruction Loss Function

$$\mathcal{L}_{reconstruction} = -\mathbb{E}_{q(z|x)}(\log p(x|z))$$

The difference between the decoder and latent space distributions is quantified by the Kullback-Leibler (KL) divergence D_{KL} [78] and can be minimized using a gradient descent optimizer:

Equation 5 KL Divergence Function

$$\mathcal{L}_{KL} = D_{KL}(q(z|x)||p(z))$$

Thus, the VAE loss function includes the reconstruction loss and KL divergence:

Equation 6 VAE Loss Function

$$\mathcal{L}_{VAE} = \mathcal{L}_{reconstruction} + \mathcal{L}_{KL}$$

Krauth et al. [62] successfully implemented the VAE framework to synthetically generate 4-dimensional flight trajectories for the Terminal Maneuvering Area (TMA). The resulting distributions matched the distributions of the actual data, proving the realism of the synthetic results. Additionally, the paper introduced multiple sources of uncertainty from various sources, such as weather, aircraft performance, and human factors. However, the authors faced difficulties attributing and identifying the origin of uncertainty for specific pairs of flight trajectories. Furthermore, while generating regular synthetic approaches, another limitation of the method revealed the lack of randomly generated go-around procedures, an example of rare events and a significant focus of interest from the collision risk modeling perspective.

Next, in [63], Krauth et al. again applied the VAE method in combination with subset simulation—a process of sampling one rare event into a sequence of multiple rare events [79]—to account for the rare events and, thus, produce more reliable synthetic aircraft trajectories. However, the authors acknowledged that the generated synthetic solution lacks physical realism. Therefore, their future work requires adding and fusing other models, such as flight dynamics equations and meteorological data, to achieve more realistic results.

Aref, Shortle, and Sherry [64] applied VAEs and adjusted their coefficients to supplement a subset of historical flight tracks using a single nominal (i.e., seed) trajectory from the actual data. The authors effectively doubled the number of original tracks by

adding synthetically generated 3-D flight tracks. In [65], Aref et al. expanded the concept by generating 4-D flight tracks. In both papers, the authors captured and compared historical and synthetic distributions for different parameters at selected locations from the runway threshold to validate the results. Since the seed track represents a single published arrival and approach procedure, the resulting distributions, expectedly, exhibited less variation and were narrower than the historical data.

Later, in [66], Aref et al. proposed a methodology for multi-seed generation of 4-D synthetic flight tracks. The authors described a process for selecting a required number of seed tracks to achieve the desired spatial-temporal variation. Although the method yielded a smaller total variational distance (TVD) (a metric used for distribution comparison) between historical and synthetic results, future work would require further investigation on multiple seed track selection to include representative clusters for different approach maneuvers.

The VAE method has proven effective application in image recognition problems, where graphical instances are defined by a set of similar pixels [80]. VAE's ability to capture the stochastic nature of the underlying data in combination with the loss function attempting to minimize differences between original and synthetic distributions, renders this approach as a lucrative candidate for generating synthetic flight trajectories. However, applications with VAEs require careful consideration and comprehensive selection for rare events in the modeled settings to yield complete and reliable results for further collision risk modeling.

One of the potential ways to combat VAE's shortcomings for rare events encapsulation is to explore Hierarchical Variational Autoencoders (HVAEs). HVAEs are a generalization of the VAE method [81] that allows a depiction of multiple levels of the latent space. Each latent space can be used as a representation class for certain rare events. Although HVAEs provide a multi-level extension to the baseline model, the approach requires careful and comprehensive consideration, selection, categorization, and allocation for the rare events in the latent spaces.

2.2.3.2 Generative Adversarial Networks

Equally to VAEs, the Generative Adversarial Networks (GANs) framework, introduced by Goodfellow in [82], also consists of two neural networks: a generator and discriminator. The generative network G captures the underlying training data distribution from the latent space and outputs high-quality synthetic data. The discriminator D attempts to differentiate between actual samples from real data and synthetic samples derived from G. Adversarial nets utilize competition between the models as a zero-sum game, where one's perceptron advantage contributes to another entity's loss (Figure 10).

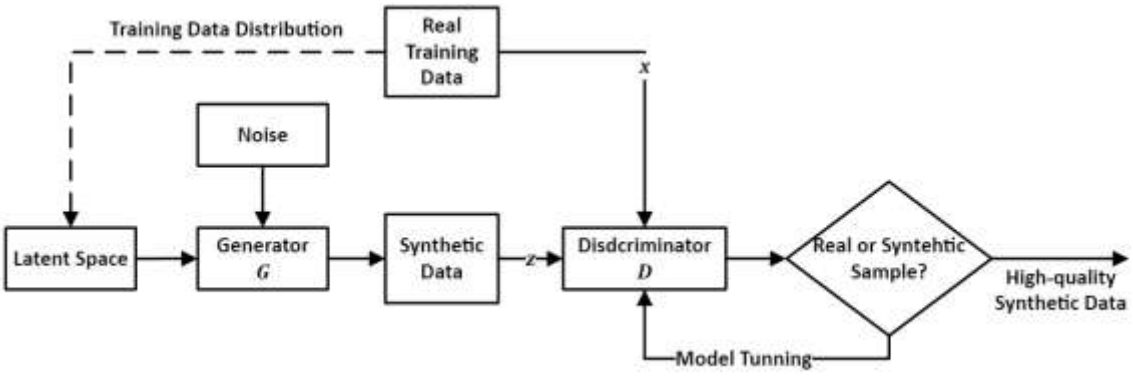


Figure 10 Generative Adversarial Networks (GANs)

In training, GANs attempt to maximize the probability that the discriminator mislabels samples. The value function $V(D,G)$ incorporates $p_{data}(x)$ and $p_z(z)$ distributions for training x and synthetic z data respectively with $D(x)$ and $D(G(z))$ representing the probability that a sample characterizes real or synthetic sample from the corresponding distribution:

Equation 7 GANs MiniMax Game

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}(\log D(x)) + \mathbb{E}_{z\sim p_z(z)}(\log(1 - D(G(z))))$$

The training process is repeated iteratively until the discriminator fails to find the differences between actual and synthetic samples, thus resulting in the model's higher fidelity. However, the algorithm's convergence may fail when the model has a limited sample size. Ji, Shortle, and Sherry [60] applied the Wasserstein Autoencoder (WAE) in conjunction with GANs to overcome the limited number of historical flight trajectories in training data. WAEs follow the same logic as VAEs but rely on the Wasserstein distance parameter instead of KL divergence.

The Wasserstein distance [83] defines the cost of converting one distribution into the other, signifying that a smaller cost implies more similarities between the distributions. The Wasserstein Autoencoder Generative Adversarial Network (WAE-GAN) framework produces robust results for flight trajectory modeling by preserving properties of the historical tracks and introducing high variance in the resulting synthetic model. However, the synthetic flight tracks exhibit smoother behavior compared to the actual tracks, leading to the exclusion of outliers in the training data and potential failure to establish rare events in the resulting synthetic trajectory model.

Wijnands et al. [67] incorporated GANs to generate 4-D aircraft landing trajectories for the TMA. The authors used an expanded version of the GAN concept to include temporal dependencies in data, referred to as TimeGAN, to ensure time consistency in the generated results. The paper highlights the importance of clustering training data to ease the neural network's learning and yield more stable synthetic results. The implemented TimeGAN methodology effectively captured the patterns in the underlying data and ensured high variability in the synthetic data. However, the approach required smoothing in output parameters to guarantee realism in the synthetic results, thus sacrificing outliers and rare events. Moreover, the resulting model exhibited limited diversity due to absence of go-around maneuvers or holding patterns. The authors attributed this restriction to the limited availability of rare events of interest in real-world data.

Although GANs are a potent tool for accurately recognizing and mimicking the underlying data properties and patterns, they heavily rely on a substantial amount of the

training data. The method performs poorly when the original data is limited, which is a common case for historical flight tracks. This makes the method ill-suited for capturing rare events in aircraft trajectories due to the limited number or lack thereof in the recorded data. Furthermore, a large scale of input data contributes to the model's computational complexity.

2.2.3.3 Diffusion Models

Diffusion models are an extension of the GAN framework. Similar to GANs and VAEs, the diffusion method consists of two phases: forward process (noising) and reverse process (denoising) [84], [85]. In the first phase, Gaussian noise gradually and incrementally supplements the input space until the samples lose their distinguishable features. In the second phase, neural networks learn to reserve and remove the noise from the samples. When the training data is represented by time-series inputs, diffusion models can effectively synthesize the underlying features. This approach can be highly applicable in the aircraft trajectory generation process, as the input parameters often denote a set of sequential discrete points whose features must be extracted.

Yin et al. [68] incorporated diffusion models into aircraft trajectory prediction analysis. The authors used a week-long set of time-series type of data from the ADS-B resources over the Singapore airport (ICAO: WSSS) as historical input to the diffusion model. The real flight trajectories were captured in one encoder, while the features of the airport infrastructure (e.g., number of runways, estimated landing time) were summarized in a different encoder. After adding Gaussian noise to both encoders, the decoder was trained to remove the noise from the original flight trajectories and the context encoder.

The resulting model has effectively predicted and captured various behaviors of 3-D aircraft trajectories entering the Singapore TMA. This study demonstrated a successful application of diffusion models to produce diverse and realistic flight trajectories. However, this paper omits the study of the effects of rare events' presence in the data and their effect on the resulting model variability (possibly due to the absence thereof in the limited historical data).

Yang et al. [69] further explore the applicability of the diffusion models to the flight trajectory generation process on the example of a nontowered TMA for general aviation. The authors present a framework capable of goal estimation and trajectory prediction. The goal estimation outlines macro patterns in the airport aerospace, while the trajectory is modeled within the defined boundaries of the operating environment. Notably and similarly to Yin et al. [68] study, the features of the complex environments are divided into two sub-models, which allows the authors to capture the nuances for both very different data representations effectively. Overall, the paper achieved significant results in trajectory prediction accuracy. However, the historical data set presents common limitations in the use case variation. For example, if the flight pattern was not observed in the training data, the diffusion model consequently ignores the potential outlier.

2.2.3.4 Physics-Informed Neural Networks

Physics-Informed Neural Networks (PINNs) describe a data-driven method that combines advances in neural network learning and training processes with physics-informed models and equations. Initially introduced in [86], PINNs have established a

powerful tool that integrates fundamental laws of physics and aerodynamics into a deep learning framework. The underlying physical laws are usually expressed in the form of partial differential equations (PDEs) and effectively capture concepts such as fluid dynamics, stress analysis, aircraft guidance and control.

Ma et al. [70] employed PINNs for aircraft routes planning and optimization in the dynamic wind environment. The wind field prediction model was established and governed by the laws of physics in combination with the Recurrent Neural Network for spatiotemporal predictive learning (PredRNN) [87]. Additionally, the wind field physical constraints were embedded into the loss function to yield more optimal, thus, fuel-efficient, aircraft routes. This paper's findings and lessons learned can be potentially transferred and applied in the synthetic flight trajectories generation paradigm.

2.2.3.5 Gaussian Processes

Another approach for the synthetic flight trajectories utilizes a Gaussian Process-based methodology that also generates solutions by mimicking the properties of the real data [34]. The Gaussian Process $f(x)$ is defined by the mean function $\mu(x)$ and kernel function $k(x_1, x_2)$ [88] that corresponds to a measure of similarity between two given points in the input space $\mathcal{X}$:

Equation 8 Definition for Gaussian Process

$$f(x) \sim GP(\mu(x), k(x_1, x_2)), x \in \mathcal{X}.$$

In the context of flight trajectories, a sequence of spatially sampled points at equidistance of each forms a single trajectory (Figure 11), while multiple trajectories constitute a flight trajectory model. A dispersion at each point of the trajectory defines the distance measured along a 90-degree line (i.e., cross-range) toward the selected

trajectory. Hence, a mean function represents the average of all trajectories, while covariance captures the dispersion at each point along the track across all trajectories in the model [71]. This completes a two-dimensional Gaussian Process model.

By definition [89], a Gaussian Process is an infinite-dimensional representation of the normal distribution. This characteristic enables the expansion of a two-dimensional model representation to include other trajectory properties in higher dimensions. The resulting model captures flight trajectories in the 3-D shape of a flow corridor, where the dispersion at each point along the track is represented by a surface rather than a line [71], [90].

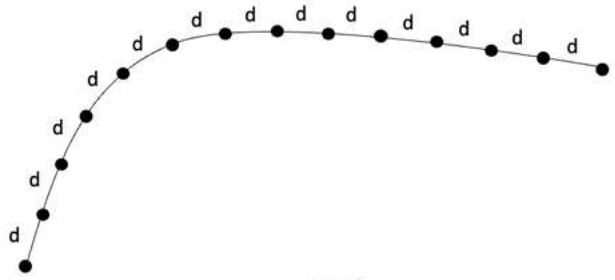


Figure 11 Flight Track with Spatially Sampled Points at Equidistance of Each Other

Chapter Three provides a detailed overview and mathematical definition of Gaussian Processes-based methods.

Eerland et. al [71] utilized a continuous framework of the Gaussian Processes, called the Gaussian Process Regression (GPR), to model the dispersion of aircraft

trajectories. The authors obtained departure procedures at Dallas-Fort Worth Airport (ICAO: KDFW) as a source of historical measurement data. The probabilistic model explicitly demonstrated high-quality and fidelity synthetic results for multiple aircraft trajectories. The authors managed to achieve 13.1% maximum deviation from the single given trajectory, which provides the necessary variability to explore the potential presence of rare events in the synthetic data. However, the dimensions for the resulting 3-D aircraft trajectories (i.e., lateral, horizonal, and vertical) were generated independently. Though the synthetic results are visually not affected by the independent GPR approach, the authors acknowledged that it is essential to capture the correlation between the parameters.

Snisarevska et al. (2025) [73] followed up on the similar approach described in [71], but based on the arrival procedures at Zurich airport (ICAO: LSZH). In the first phase of the study, the authors incorporated separate GPR models for each parameter of the 3-D flight trajectory and encountered analogous limitations as in [71]. However, the outputs were generated simultaneously based on the same (i.e., reused) input data, resulting in the independent multi-output GPR model. The paper concludes that the next step is to generate these parameters interdependently, with a follow-up analysis summarized in [91].

Xu, Sherry, and Shortle (2024) [72] employed Gaussian Process for Dynamic Systems (GPDS) to model new trajectories based on the arriving flight tracks. The interim results demonstrated diverse synthetic flight trajectories that are supplemental to the real tracks. Although, similar to [73], this work is still in progress, the paper discusses

future strategies to deal with unrealistic synthetic tracks. One of the potential solutions is to utilize different kernels for the latent function (see Section 3.1.3 for the detailed mathematical description of the kernel functions).

Gaussian Processes are data efficient methods, where the degree of covariance (i.e., similarity) in the kernel function is measured between data points. Therefore, the predictions are accurate even in regions with sparse observations of the input space. However, Gaussian Processes introduce computational complexity that is mainly attributed to mathematical operations in high-dimensional space with time-series type of data. As concluded in [72], the selection of the Gaussian Process hyperparameters can be cumbersome and highly impactful on the prediction's quality [92], especially in instances when a mix of stationary and non-stationary patterns is present in the data (i.e., the problem of non-stationarity). Furthermore, engineering ad-hoc approaches are required when handling the flight trajectories model's interdependent output parameters (i.e., the multi-output problem) [93].

2.3 Gap Analysis in Flight Trajectories Literature

The subset of literature targeting flight trajectory generation process commonly faces the following challenges:

- Sufficient sample size in generated flight trajectory profiles
- Omission of rare events of interest in the resulting flight trajectories
- Substandard solutions derived from the lack of high-quality training data

- Tradeoff decisions between modeled assumptions and complexity, leading to oversimplified and unrealistic solutions
- Limited control over physical constraints, resulting in the lack of physical realism in the synthetic results
- The lack of transparency between inputs and outputs in the generative AI applications
- Complex training process, model instability, and parameter tuning
- Computational cost, long runtimes, and overall method complexity

In summary, strengths and weaknesses for each method considered in detail in the previous sections are provided in Table 2. The analysis of the existing implementations to generate flight trajectories highlights that addressing the issues in one method can inadvertently create gaps in another, resulting in the inherent trade-offs in each approach. Often, multiple methods are used in a fused or hybrid fashion to achieve better coverage and overcome the common deficiencies. For example, a fusion of GANs and WAE methods in [60], whereas PINNs themselves are an embodiment of GANs in combination with physics-based models, and HVAEs consist of multiple stacked VAEs.

Gaussian Processes were selected as a candidate for generating synthetic flight trajectories for further analysis in the CRM domain. Based on the comprehensive literature review of other existing methods, Gaussian Processes provide a universal approach for generating new flight trajectories with many opportunities for tailoring and tuning the resulting model. Additionally, the Gaussian Processes-based models can also be fused with other methods to achieve a desired level of fidelity in the synthetic results.

Table 2 Strengths and Weaknesses of Flight Trajectory Generation Methods

	Method	Generative Mechanism	Strengths	Weaknesses
Model-driven (simulated)	Physics-based models	Mathematical formulations of physical laws to simulate aircraft motion	(1) Physically consistent flight trajectories (2) Deterministic modeling (3) High interpretability	(1) Requires comprehensive expertise in aircraft performance and dynamics modeling (2) Limited representation of uncertainty sources (3) Dependence on assumptions when required data are proprietary or unavailable (4) Oversimplified modeled assumptions result in underestimated or misrepresented collision risk
	Agent-based models	Behavioral and sequential simulation of interactions among actors and system components	(1) Flexibility (2) Scalability (3) Expandability	(1) Requires expertise in defining behavioral assumptions (2) Large numbers of simulation iterations require high-performance computing clusters and GPU-accelerated physics engines (3) Long runtimes for full-airspace simulations
Data-driven (synthetic)	VAE	Learns a compressed latent representation through an encoder-decoder architecture and generates new trajectories by sampling from the latent space	(1) Generates smooth and continuous flight trajectories (2) Provides a structured latent space for interpolation and clustering (3) Stable training compared to GANs (4) Useful for anomaly detection and representation learning	(1) Requires sufficient representation of rare events in training data (2) Limited control over physical constraints (3) May generate overly smoothed trajectories

	Method	Generative Mechanism	Strengths	Weaknesses
	HVAE	Uses multiple latent layers to represent both high-level and fine-grained trajectory characteristics	(1) Captures multi-scale trajectory patterns (2) Models complex spatial-temporal dependencies (3) More expressive than standard VAEs (4) Better representation of diverse flight behaviors	(1) Requires sufficient representation of rare events in training data (2) Complex model training and hyperparameter tuning (3) More complex architecture and training process (4) Increased computational cost
	WAE	Latent-space sampling through encoder-decoder architecture using optimal transport cost between learned and target latent distributions as the loss function	(1) Produces more diverse and realistic samples than standard VAEs (2) Better latent-space organization (3) Improved sample quality and stability (4) Reduced mode averaging effects	(1) Tends to smooth results and suppress outliers, potentially excluding rare events (2) Computationally expensive optimal transport objectives (3) Sensitive to regularization parameters (4) Limited incorporation of domain-specific constraints
	GAN	Adversarial learning between a generator network and a discriminator network	(1) Realistic and diverse flight trajectories (2) Effective in capturing complex data distributions (3) Generates sharp trajectory patterns without excessive smoothing	(1) Training instability and mode collapse (2) Data inefficiency compared with probabilistic models (3) Convergence issues with limited training data (4) Requires large, high-quality datasets
	Diffusion Models	Learn trajectory generation through iterative forward noising and reverse denoising processes through neural networks	(1) Realistic and diverse trajectories (2) Effectively models multimodal trajectory distributions (3) Capable of separating macro- and micro-level patterns (4) Stable training compared to GANs	(1) High computational cost during training and inference (2) Data-intensive training requirements (3) Long generation times due to iterative denoising (4) Complex architecture and tuning

	Method	Generative Mechanism	Strengths	Weaknesses
	PINN	A loss function with integrated physical laws in conjunction with neural networks	(1) Physically consistent trajectories (2) Supports incorporation of flight dynamics and operational constraints (3) Reduces physically implausible outputs (4) Combines advantages of physics-based and data-driven approaches	(1) Requires comprehensive expertise in physical modeling (2) Complex training and model tuning (3) Lower scalability and adaptability compared with other data-driven methods (4) Limited adaptability when governing equations are uncertain
	Gaussian Process	Bayesian nonparametric probabilistic framework using kernel functions to model predictive distributions	(1) Provides uncertainty quantification (2) High data efficiency (3) Interpretable probabilistic predictions (4) Flexible kernel design (5) Strong performance with limited data (6) Suitable for trajectory prediction and interpolation tasks	(1) Cubic computational complexity with dataset size (2) Poor scalability for large datasets (3) Challenges with high-dimensional trajectory features (4) Difficulties handling non-stationary behaviors (5) Limited capability for modeling multiple correlated outputs without specialized kernels

CHAPTER THREE: LITERATURE REVIEW PART II – GAUSSIAN PROCESSES

This Chapter reviews literature related to the Gaussian Processes methods. Section 3.1 provides an overview and definitions for the concept. Section 3.2 describes different implementations of the Gaussian Processes. Section 3.3 discusses a problem of non-stationarity. Section 3.4 addresses a multi-output problem. Section 3.5 summarizes gaps in literature.

3.1 Overview of Gaussian Processes

3.1.1 Definitions

A *Gaussian Process* is defined as a stochastic process with a joint Gaussian distribution [94]. A *stochastic process* is a mathematical concept represented by a sequence of random variables distributed in space or, more commonly, time. A *random variable* or a *random function* of input space is a function that determines an outcome of a random event or experiment [95].

A continuous random variable that is normally distributed has a *normal distribution*. A bivariate normal distribution is a two-dimensional form of normal distribution between two continuous random variables. A *joint Gaussian distribution* describes the probability distribution of a finite collection of random variables for which any linear combination is Gaussian distributed. It is completely characterized by its mean vector and covariance matrix [95]. A Gaussian Process extends to an infinite-dimensional

space. Sometimes, it can be denoted as a k-variate Gaussian distribution to specify a dimensionality of k. To avoid further confusion, the words 'normal' and 'Gaussian' are used interchangeably; hence, the joint Gaussian distribution is synonymous with a multivariate normal distribution, joint normal distribution, and multivariate Gaussian distribution.

In other words, the Gaussian Process is a collection of random variables, any pair of which has a bivariate normal distribution [94]. Formally [96], let $\mathcal{X}$ denote an input (or feature) space for the unknown output $f(x)$, and $Y(x)$ describe a random variable. When $\mathcal{X}$ is a finite set in the index set L, $Y(x), x \in \mathcal{X}$ is a Gaussian Process if and only if any subset of $x_1, \ldots, x_L$ in $\mathcal{X}$ of the vector consisting of random variables $(Y(x_1), \ldots, Y(x_L))$ has a multivariate normal distribution.

A complete definition of the Gaussian Process $f(x)$ consists of the mean function $\mu(x) \equiv E[Y(x)], x \in \mathcal{X}$, and covariance function $C(x_1, x_2) \equiv Cov[Y(x_1), Y(x_2)]$.

Equation 9 Complete Definition for Gaussian Process

$$f(x) \sim GP(\mu(x), C(x_1, x_2)).$$

The mean function provides an estimate of the prediction. The covariance function describes how similar two random variables are at any given location x of the input space. Covariance measures the joint variability of two random variables by a degree of similarity of distance. In the context of the Gaussian Process, the covariance function is the same as the kernel function, which is explained later. Notably, according to the definition of the Gaussian Process, $y(x)$ is also a random variable.

3.1.2 Properties of Stationarity and Isotropy

A Gaussian Process is stationary if its covariance function $C(x_i, x_j)$ depends only on the relative distance between two given points $d(x_i, x_j)$ [97], thus, is invariant to translations or shifts in the input space,

Equation 10 Covariance Function

$$C(x_i, x_j) = k\left(d(x_i, x_j)\right),$$

where k is a function of distance d.

The property of stationarity implies that if the entire input space is shifted by the same amount, the accuracy of the predictions will not be affected [98]. Contrary to the stationary Gaussian Processes, a non-stationarity implies that the mean and covariance function of the Gaussian Process may vary and exhibit different characteristics across the input space. A non-stationary covariance function captures local variations in smoothness and other parameters, producing more flexible and realistic modeled functions [99]. It depends on the specific location of the input points (i.e., their exact values) instead of the relative distance between them.

A stationary Gaussian Process is isotropic if its covariance function depends only on the Euclidean distance between inputs, $k(x, x') = k(\|x - x'\|)$, making it invariant to rotations of the input space. The isotropic property infers that the modeled process is indifferent to rotations in the input space. The invariance to rotations implies that the covariance function depends only on the distance between a pair of inputs, not their direction, order, or orientation. On the contrary, the anisotropic Gaussian Process reflects variations in magnitude according to the direction of movement. For example, the

anisotropic processes in real life include everything where directionality and trend information are essential. These include stock market fluctuations or arriving flight trajectories, where a descending pattern is observed.

3.1.3 Kernel Functions

The *kernel function* $k(x, x')$ (or *covariance function* $C(x, x')$) controls the modeled function's behavior through a set of hyperparameters, such as shape, scale, and smoothness. Mathematically, it calculates the degree of similarity between two random variables in the input space. The degree of similarity is captured by covariance and, for example, in some cases can be explained by distance between two random variables.

Each kernel presents distinctive assumptions about the modeled function behavior [98]. The underlying function might be monote, linear, quadratic, etc. For example, a squared exponential kernel can be applied to show that the behavior is expected to be smooth and continuous. A set of individual parameters that define the kernel function are its hyperparameters. The hyperparameters include the observational noise variance ω^2, signal variance σ^2, and the length-scale ℓ [97]. The change in noise magnitude can affect any model. The signal variance captures the amplitude of the function variations, while the length-scale parameter describes the function's fluctuations and smoothness. The smoothness is defined by function's continuity and differentiability.

Ideally, the selected kernel captures the underlying behavior of the modeled function. Its hyperparameters are optimized by maximizing the marginal likelihood, enabling the model to fit the observed data while avoiding unnecessary complexity, thus,

covering and explaining more data points in the model. The kernel function outputs the correlation matrix in the joint Gaussian distribution over training data and new input data.

Kernel functions can be stationary or non-stationary. Kernels for various types of data include linear, polynomial, radial basis function (RBF), Matérn, periodic, rational quadratic, and other kernels, as well as an option of combining these kernels and custom kernels. A non-exhaustive list of examples of certain kernel functions with their associated hyperparameters and properties is provided below.

3.1.3.1 Radial Basis Function (RBF) Kernel

The RBF (or squared-exponential) kernel is a widely used stationary kernel function in Machine Learning algorithms such as support vector machines (SVM) and Gaussian Processes, which is given by [100]:

Equation 11 RBF Kernel

$$k(x_i, x_j) = \sigma_f^2 \exp\left(-\frac{d(x_i, x_j)^2}{2\ell^2}\right),$$

where $d(x_i, x_j)$ is the Euclidean distance between two points of the input space $\mathcal{X}$, σ_f is the signal variance and ℓ is the characteristic length-scale hyperparameters. Since the RBF kernel is both stationary and isotropic, it is invariant under rotations of the input space [95]. If the length-scale parameter is a scalar value, then the Gaussian Process is isotropic, otherwise it is anisotropic. The signal variance controls the amplitude or vertical variance of the Gaussian Process.

An example of samples derived from the RBF kernel is shown in Figure 12. A python script to generate samples from the RBF kernel and the following kernel functions is provided in Appendix A – Python Scripts to Generate Kernel Functions. The first plot

(Figure 12a) demonstrates the RBF kernel function with hyperparameters set to $\sigma_f^2 = 1.0$ and $\ell = 1.0$. As the signal variance increases (Figure 12b), the amplitude of the functions rises proportionally. As the length-scale increases (Figure 12c), the fluctuations become smoother. The Gaussian Process with this underlying kernel function is very smooth because it has mean square derivatives of all orders [94].

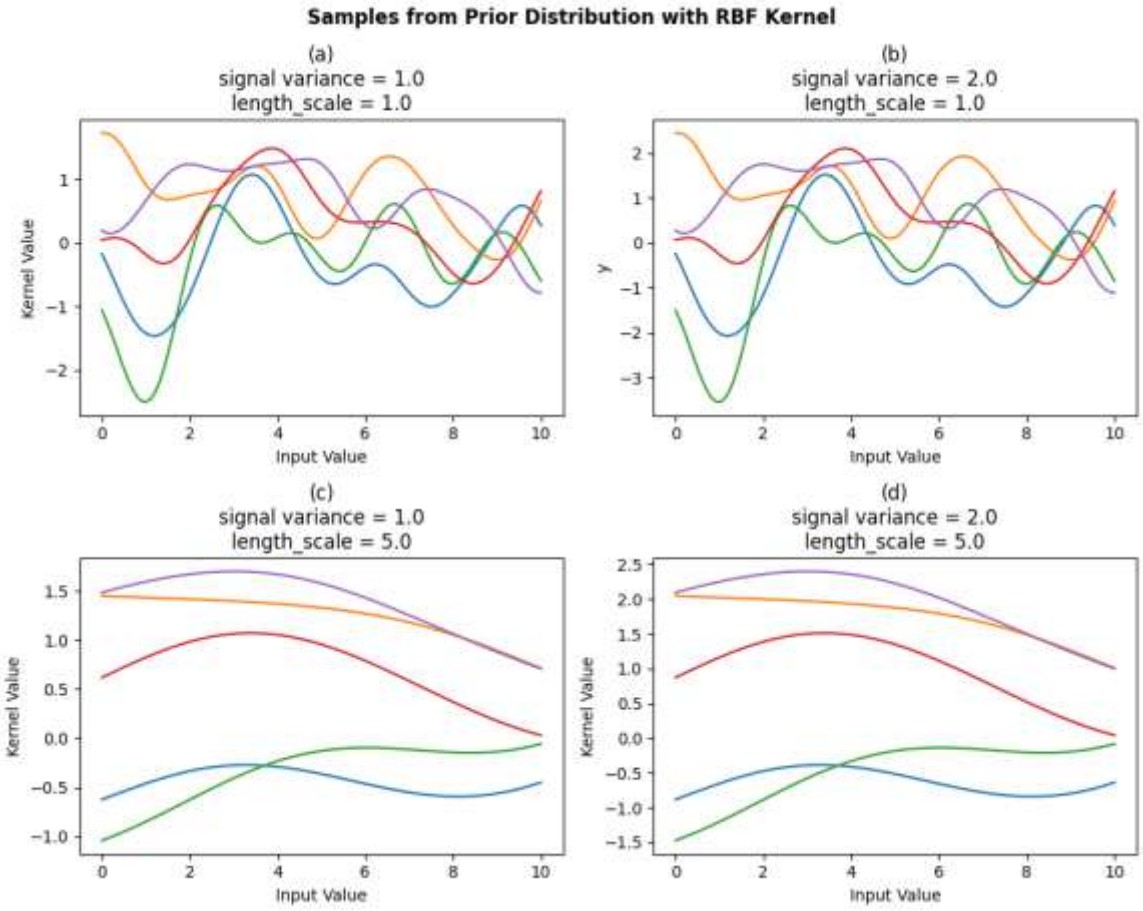


Figure 12 Samples from Prior Distribution with RBF Kernel

Given a finite set of input points $\{x_i, \dots, x_j\}, x \in \mathcal{X}$ and i, j are positive indices in the index set L, $L > 2$, a covariance matrix K can be derived. The number of elements in the input set, n defines the dimensions of the covariance matrix, $n \times n$. Each matrix element K_{ij} represents a covariance between each pair of input points as defined by the RBF kernel:

$$K_{ij} = k(x_i, x_j).$$

The derived covariance matrix is symmetric and positive definite:

Equation 12 RBF Covariance Matrix

$$K_{ij} = \begin{bmatrix} \boldsymbol{k(x_1, x_1)} & k(x_1, x_2) & \dots & k(x_1, x_n) \\ k(x_2, x_1) & \boldsymbol{k(x_2, x_2)} & \dots & k(x_2, x_n) \\ \dots & \dots & \dots & \dots \\ k(x_n, x_1) & k(x_n, x_2) & \dots & \boldsymbol{k(x_n, x_n)} \end{bmatrix}$$

In this case, the covariance matrix demonstrates correlation, implying that the points closer to each other have a higher correlation than the points further away.

A procedure for hyperparameters estimation and optimization of a GPR model with the RBF kernel is defined in [100], [101]. Given training data set, the modeler derives the covariance matrix from the RBF kernel. The kernel's hyperparameters must be set somewhat intuitively within the allowable range at the current stage. Because the RBF kernel typically contains only a few hyperparameters, the maximum marginal likelihood can be estimated using gradient-based optimizers [96]. The log marginal likelihood is given by [94]:

Equation 13 RBF Log Marginal Likelihood Function

$$\log p(\boldsymbol{y}|X, \theta) = -\frac{1}{2} y^T M^{-1} y - \frac{1}{2} \log|M| - \frac{N}{2} \log 2\pi,$$

where $M = K + \sigma_n^2 I$ is a sum of the covariance matrix and observational noise variance.

The larger the number of hyperparameters, the more complex the model; hence, it is computationally burdensome. The gradient-based optimizers process must be repeated until the hyperparameters converge to values that maximize the marginal likelihood.

The computational cost associated with the RBF kernel is $O(N^3)$, where N is the size of training data. This derives from computing covariances for each pair of inputs from the training set (resulting in $O(N^2)$), followed by inverting the covariance matrix ($O(N^3)$) to obtain the marginal likelihood and predictions. Moreover, the covariance matrix's computational cost is $O(DN^2)$ with required $O(N^2)$ of memory [101]. Consequently, the higher the dimensionality of the training data, the longer the optimization lasts.

3.1.3.2 Matérn Kernel

The Matérn kernel is an extension of the RBF kernel and is defined as follows [98]:

Equation 14 Matérn Kernel

$$k(x_i, x_j) = \frac{1}{\Gamma(v)2^{v-1}}\left(\frac{\sqrt{2v}}{l} d(x_i, x_j)\right)^v K_v\left(\frac{\sqrt{2v}}{l} d(x_i, x_j)\right),$$

where v is the additional parameter that controls smoothness of the Gaussian Process. This parameter is selected arbitrarily and not affected by the optimization process.

Commonly, v is selected from the range of intermediate predefined values between $[0.5, 1.5, 2.5, \infty]$ to avoid further computational burden [95], [102]. When v is

infinitively large, the process resembles the RBF kernel. Contrarily, when v is relatively small, the resulting function is rougher. With $v = 0.5$, the Matérn kernel corresponds to the (absolute) exponential kernel. The smoothness value of 1.5 sets the function once differentiable, while $v = 2.5$ makes the function twice differentiable, ensuing a smoother behavior.

An example of samples derived from the Matérn kernel is shown in Figure 13. As v increases from 0.5 to ∞, the functions become smoother and resemble the RBF kernel.

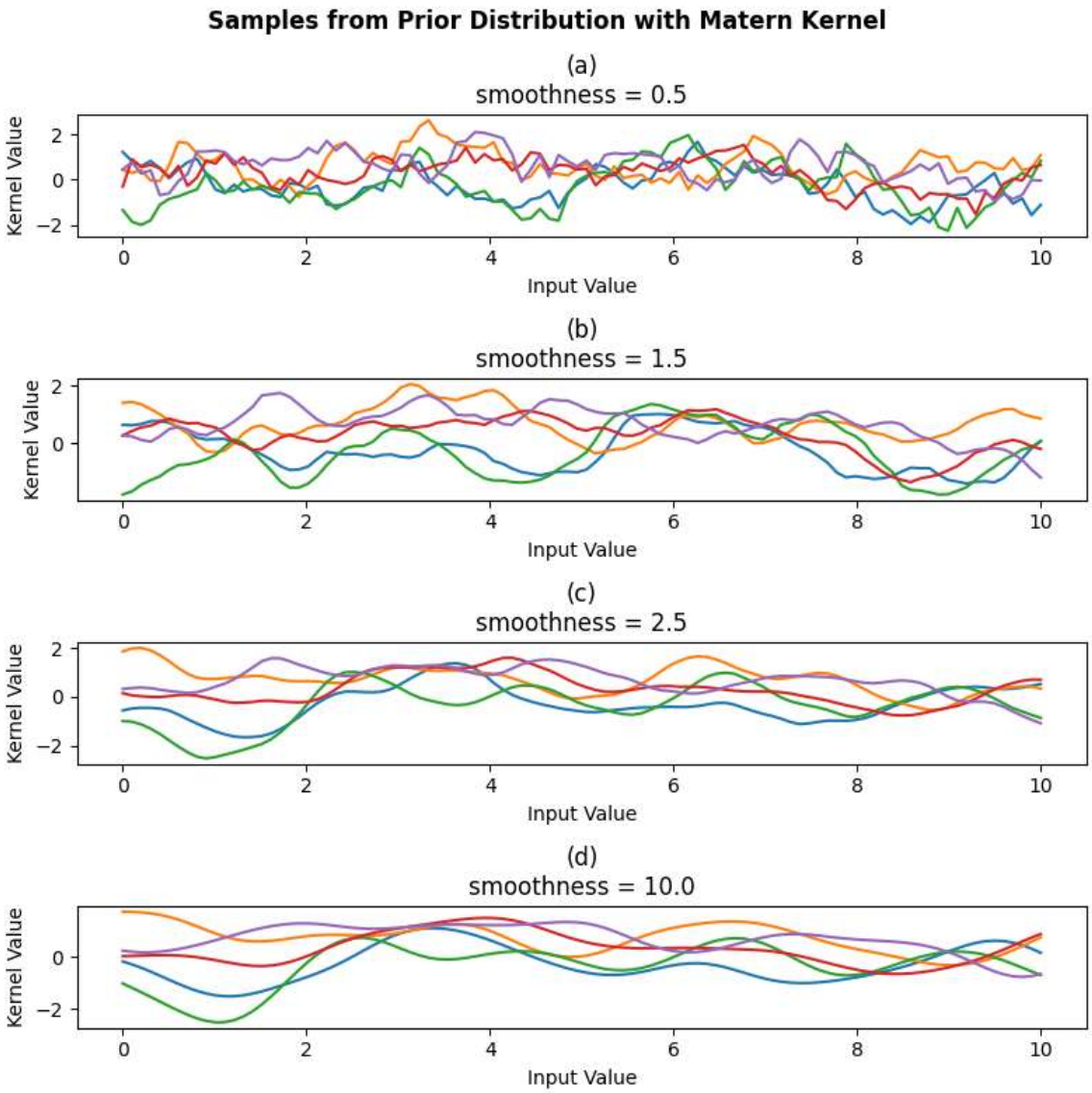


Figure 13 Samples from Prior Distribution with Matérn Kernel

In the Matérn kernel expression, $\Gamma(v)$ denotes the Gamma function, which generalizes the factorial function to real-valued arguments, and K_v represents the modified Bessel function of the second kind of order v. Together, these functions

determine the covariance structure and smoothness properties of the resulting Gaussian process.

Similar to the RBF kernel, the computational cost for the Matérn kernel also scales cubically ($O(N^3)$) with N number of data points in the training set.

3.1.3.3 Linear Kernel

Linear kernel is an example of a non-stationary kernel. Contrary to the stationary kernels, a non-stationary kernel function depends on the specific location of the points across the input space [20] instead of the relative distance between them. Non-stationary kernel implies that the predictions of the Gaussian Process model change when the input space shifted. The kernel function is defined as follows:

Equation 15 Linear Kernel

$$k(x_i, x_j) = \sigma_f^2 (x_i - c)(x_j - c),$$

where c is the offset by which all points are moved.

An example of samples derived from the Linear kernel is shown in Figure 14. As the variance parameter σ_f^2 increases, the sampled functions move further away from the mean function of zero.

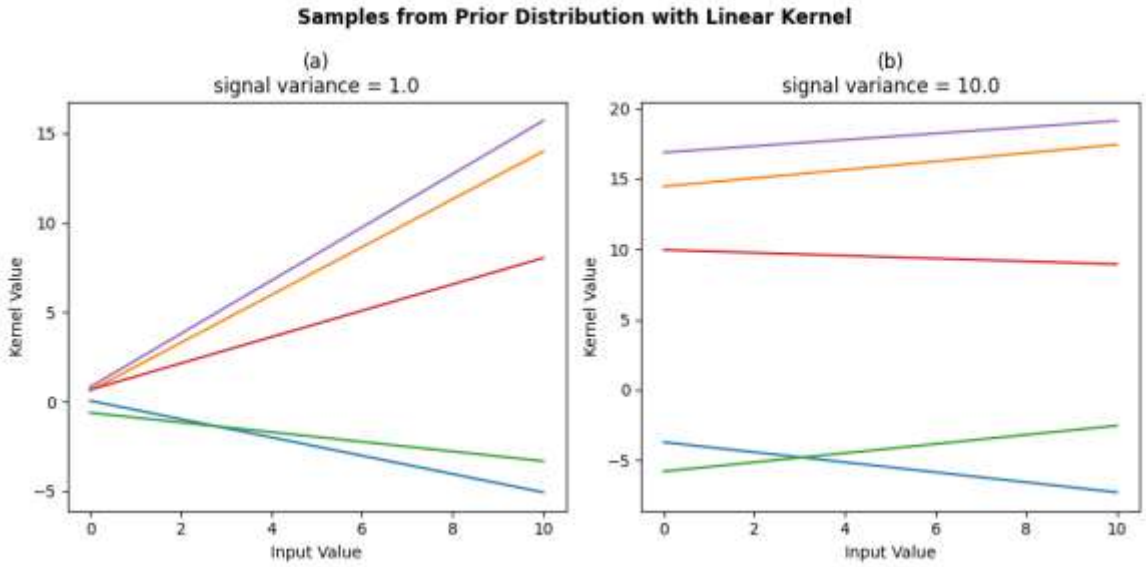


Figure 14 Samples from Prior Distribution with Linear Kernel

The computational cost for the Linear kernel is $O(N)$.

3.1.3.4 Periodic Kernel

The Periodic kernel is another example of a non-stationary function. The periodic functions are repeated across the input space at a specified interval. The kernel is given by [94]:

Equation 16 Periodic Kernel

$$k(x_i, x_j) = \sigma_f^2 \exp\left(-\frac{2}{\ell^2} sin^2\left(\pi \frac{x_i, x_j}{p}\right)\right),$$

where p is the periodicity and defines the distance between periods, while σ and ℓ are the signal variance and length-scale parameters, respectively.

The effect of different periodicity values on the samples is provided in Figure 15.

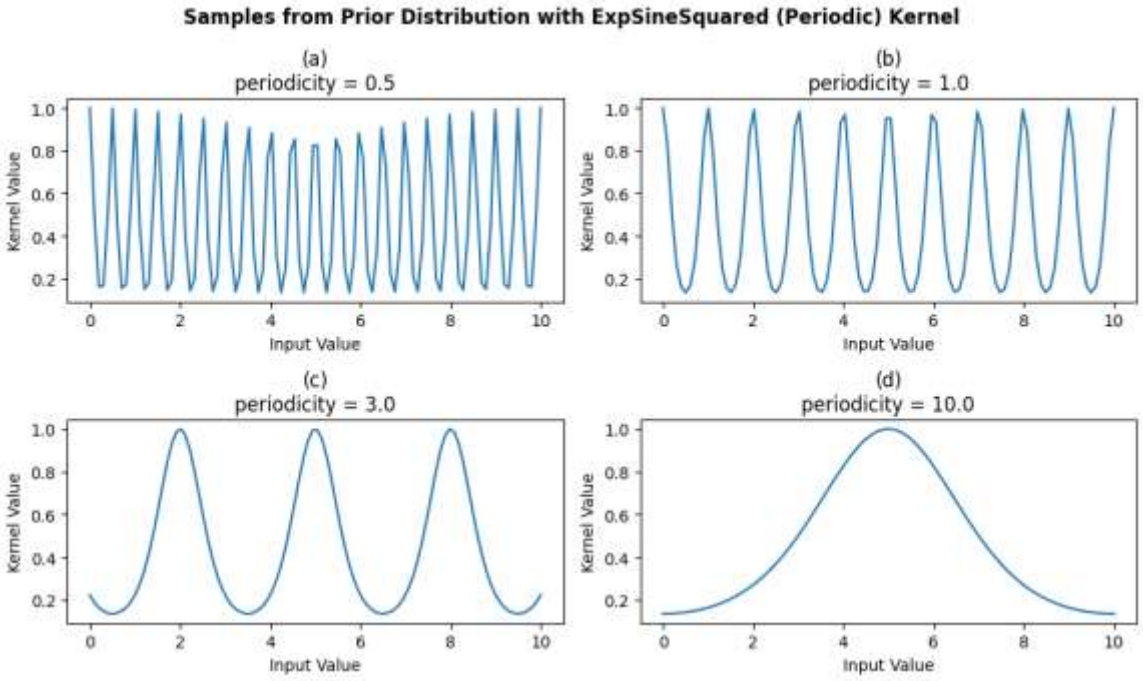


Figure 15 Samples from Prior Distribution with Periodic Kernel

3.2 GAUSSIAN PROCESSES IMPLEMENTATIONS

Gaussian Processes are incorporated into numerous frameworks and algorithms such as Gaussian Process Regression for problems with continuous output, Gaussian Process Classification for problems with discrete output, Deep Gaussian Process, Treed Gaussian Process, Gaussian Process for Dynamic Systems.

3.2.1 Gaussian Processes Regression (GPR)

Machine Learning methods can be broadly categorized into supervised and unsupervised learning. Supervised learning searches for patterns in data by mapping and describing relationships between the model's inputs and outputs. In supervised learning, regression refers to predicting continuous-valued outputs, whereas classification involves

assigning inputs to discrete categories. A supervised learning technique based on the Gaussian Process with continuous output is called the *Gaussian Process Regression (GPR)* [102].

The GPR is a non-parametric model that incorporates Bayesian inference [102]. In non-parametric models, predictors are derived directly from the data, unlike prediction models where input and output relationships are described by an underlying function (e.g., an equation for linear regression). Non-parametricity becomes valid when the shape of the inherent function (i.e., a flight trajectory) is unknown or highly non-linear. The GPR's key components include observations and Bayesian inference distributions, such as prior, posterior, and predictive distributions [96].

The *observations* are an allocated set of input-output training data from which the model learns. The test samples from the original data are stored separately to verify prediction accuracy later. In the context of collision risk modeling, observations describe flight trajectories.

Like in Bayesian statistics, the *prior distribution* expresses our beliefs about the form and smoothness of the modeled function before observing any data. The prior distribution also incorporates our uncertainty about the function's behavior. In our case, the prior distribution describes our assumptions of the flight trajectories behavior before the training process starts.

A *kernel function* is selected to describe a prior distribution and capture our beliefs about observations' shape and smoothness [94]. The selected kernel function

computes and attempts to maximize marginal likelihood, which defines the probability of observed data coverage by a given model.

The prior distribution, combined with the likelihood of the observation data, defines the *posterior distribution*. After model training, the posterior distribution provides updated beliefs about the underlying function. The definition for the posterior probabilities derives directly from Bayes' theorem [103]:

Equation 17 Bayes' Theorem

$$P(H|E) = \frac{P(E|H) \times P(H)}{P(E)},$$

- where $P(H|E)$ is a posterior or conditional probability for hypothesis H given evidence E,
- $P(H)$ is the estimate of the probability H before observing any data provided by E. This is the prior probability captured by a kernel function,
- $P(E)$ is a marginal probability or model evidence without any given conditions,
- $P(E|H)$ is a conditional probability or likelihood of observing data E given H.

In other words,

Equation 18 Posterior Distribution

$$Posterior = \frac{Likelihood\ of\ the\ observations\ \times Prior}{Marginal\ likelihood}$$

The *predictive distribution* derives predictions for the test samples from the posterior distribution. The prediction estimate is derived from the posterior mean, while the posterior covariance captures the prediction uncertainty.

The prior and posterior distributions of the GPR model based on a Matérn kernel with two different smoothness parameters are provided in Figure 16. The underlying

modeled function with selected points for observations is shown in Figure 16 (a). At this point, the behavior, shape, and smoothness of the function are unknown to the GPR model.

Prior samples, obtained from the Matérn kernel function, are shown as dotted lines with smoothness of $v = 0.5$ and $v = 1.5$ in Figure 16 (b) and (d), respectively. These samples reflect our assumptions about the parameters of the modeled function before observing any data.

The posterior mean, uncertainty intervals, and posterior samples are illustrated in Figure 16 (c) and (e). The posterior mean, represented by a solid black line, provides an averaged estimate over the functions after observing the training data. The 95% confidence interval, indicated by the gray shaded areas, encapsulates the prediction uncertainty. Posterior samples are drawn from the predictive distribution. Evidently, as the smoothness parameter v increases, the resulting synthesized predictions better resemble the underlying function.

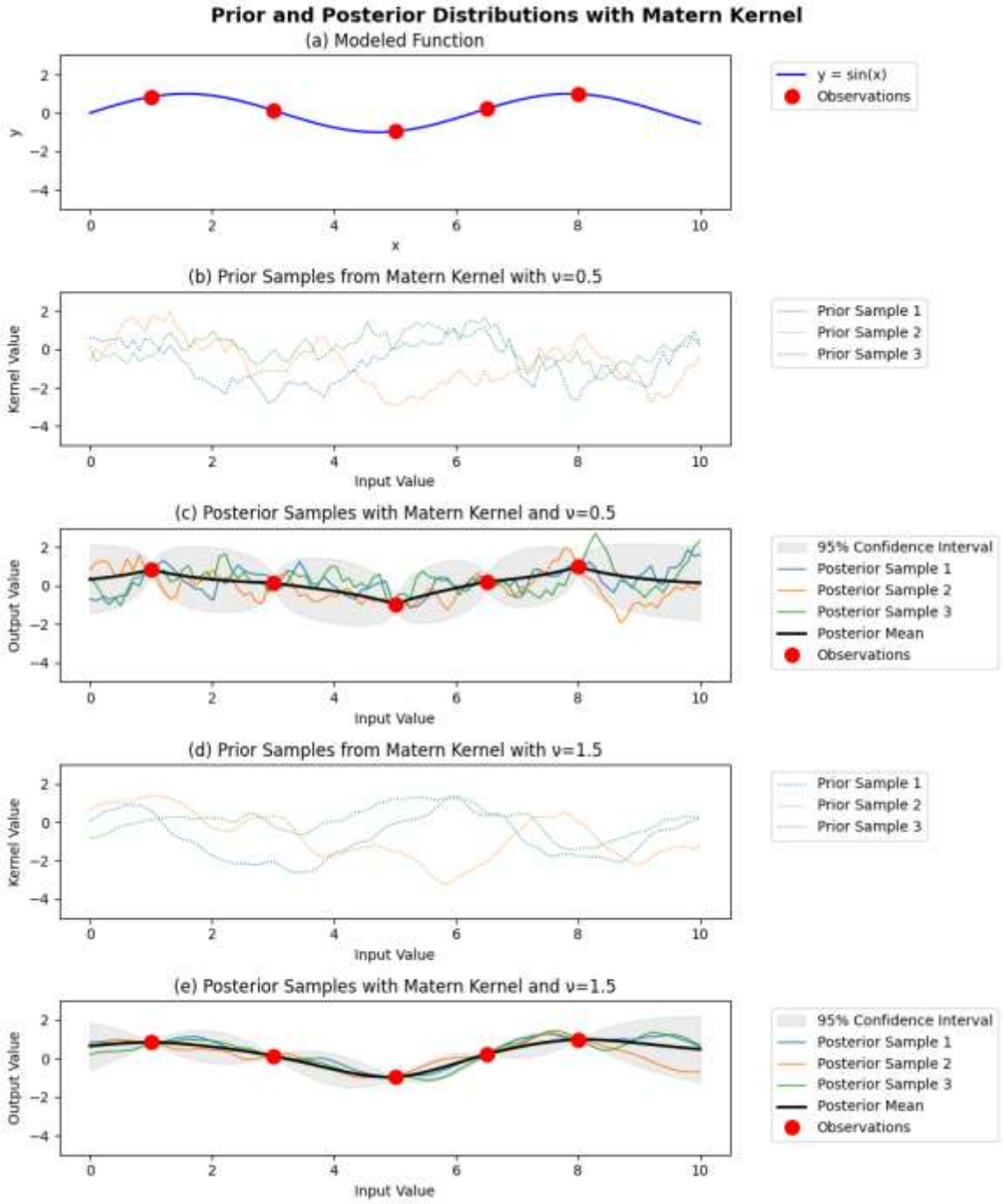


Figure 16 Prior and Posterior Distributions from GPR Model with Matérn Kernel

The GPR models ensure flexibility and versatility by learning directly from the data rather than fitting the data into a pre-determined equation. GPR operates within a probabilistic framework, allowing it to estimate the mean value and associated uncertainty for each prediction. As shown in the examples in Figure 16, this method is data-efficient and does not require a large training dataset. However, the model loses interpretability when employing complex kernel functions. Scalability and computational complexity are additional challenges when dealing with high-dimensional cases. Furthermore, GPR is not well-suited for non-stationary functions, as discussed in Section 3.4.

3.2.2 Deep Gaussian Process (Deep GP)

Deep Gaussian Process (Deep GP) is a generalization framework represented by a cascade of Gaussian Processes. A Deep GP model comprises stacked hidden layers of latent (or observed) variables where interactions and mapping of the entire composition are governed by the Gaussian Process [104]. In this case, a Gaussian Process defines a hierarchy where one layer's output becomes another layer's input [105]. As a result, this method enables modeling intricate behaviors by utilizing a set of plain Gaussian Processes with simple underlying kernel functions and fewer hyperparameters.

An output of a conventional Gaussian Process, such as GPR—or a single-layered Deep GP—directly defines the observed model response $\boldsymbol{y}$. In the context of the stacked composition, the output of one layer serves as an input to the next layer. When this process is repeated L times, it constitutes a Deep GP model with L hidden layers [106]. Because each layer introduces its own set of latent variables, the resulting hierarchical

dependencies between the layers make inference and posterior distribution approximation computationally challenging [107]. A method discovered in [108] overcomes issues induced by the uncorrelated posterior by removing independence between the layers. However, while preserving non-linearity, this approach contributes to a loss of analytical tractability.

A complete definition of Deep GP consists of a product of its prior and likelihood. In D-dimensional space, $\boldsymbol{Y}$ is the matrix for i observations, $\boldsymbol{y}_i \in \mathbb{R}^D$. The output dimensions at layer l are denoted as D^l. Given a stochastic vector of functions at the design $F^1, \dots, F^L$, the outputs at layer l are F_d^l, while the corresponding inputs are F^{l-1}. The inducing function values $\boldsymbol{U}^l$ for each dimension with the corresponding inducing locations $\boldsymbol{Z}^{l-1}$ at each layer. The regression joint Deep GP model is defined as follows [108], [106]:

Equation 19 Deep Gaussian Process Definition

$$p(\boldsymbol{Y}, \{\boldsymbol{F}^l, \boldsymbol{U}^l\}_{l=1}^L) = \prod_{i=1}^{N} p(\boldsymbol{y}_i|\boldsymbol{f}_i^L) \prod_{l=1}^{L} p(\boldsymbol{F}^l|\boldsymbol{U}^l; \boldsymbol{F}^{l-1}, \boldsymbol{Z}^{l-1}) p(\boldsymbol{U}^l; \boldsymbol{Z}^{l-1}),$$

where the first component is likelihood, and the second component is Deep GP prior. Here, $\boldsymbol{F}^0 = \boldsymbol{X}$, where X is a vector of N observation values at the design locations. The semicolon in the equation specifies the Gaussian Process inputs.

According to [108], Deep GP models consistently exceed the results compared to conventional single-layered Gaussian Process models. Nevertheless, stationary kernel functions used in the Deep GP derive from the outputs of prior layers, thus constituting scalability and dimensionality challenges. This results in a highly nonlinear model that, on the one hand, addresses non-stationarity issues but becomes highly intractable and less

interpretable [109]. Moreover, ongoing research efforts are focused on reducing computational complexity and cost associated with the Deep GP approach.

3.3 Challenges in Gaussian Process Implementations

3.3.1 Problem of Non-Stationarity

In general, non-stationary behavior in GP can be expressed in various forms: some functions can exhibit smooth and gradual changes in their behaviors, while others can present rapid and sharp variations [110].

One of the significant limitations of the GPR method is that it cannot handle non-stationary data effectively. The GPR algorithm assumes that the modeled function behavior does not change over time. The modeled function's behavior derives from the covariance function parameters such as the observational noise variance ω^2, signal variance σ^2, and the length-scale ℓ [97]. The change in noise magnitude can affect any model. The signal variance captures its amplitude, while the length-scale parameter determines how rapidly the function can vary with changes in the input. The smoothness is defined by function's continuity and differentiability. These parameters can change over time, thus, display non-stationarity.

When applying an original GPR model to non-stationary data, the model generally can correctly identify only a portion of the observed data. For example, if the underlying function exhibits oscillations before stabilizing, the periodic kernel might reflect the initial behavior satisfactorily but later add fluctuations to the linear segment of the function as well. This constitutes a *spatial modeling problem*.

Figure 17 illustrates an example of a non-stationary underlying function. In this example, the second half of the input space demonstrates linear behavior, contrasting with the rest of the function. The applied GPR model, which employs a Matérn kernel (as shown in Figure 18), failed to grasp this variability and uniformly estimated uncertainty across the entire function.

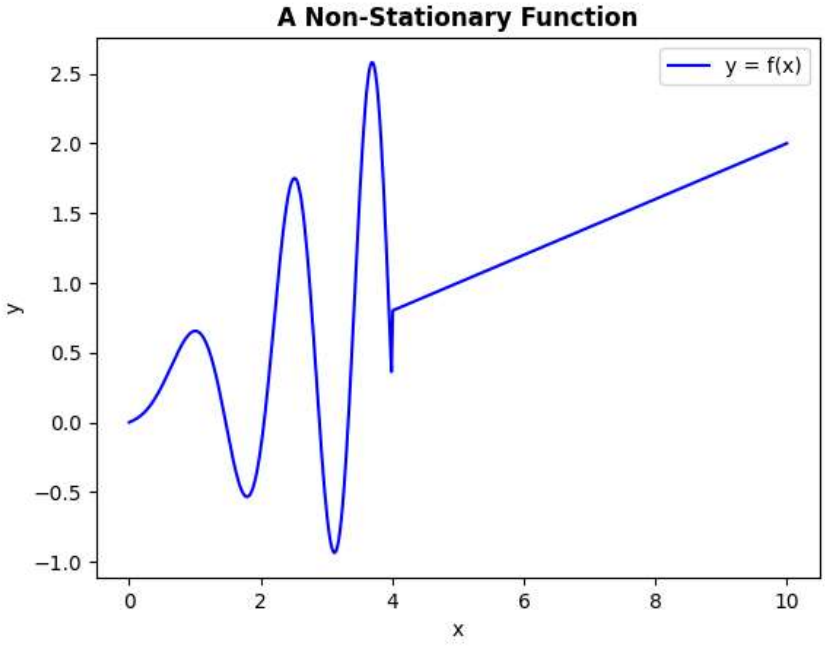


Figure 17 Example of Non-Stationary Function

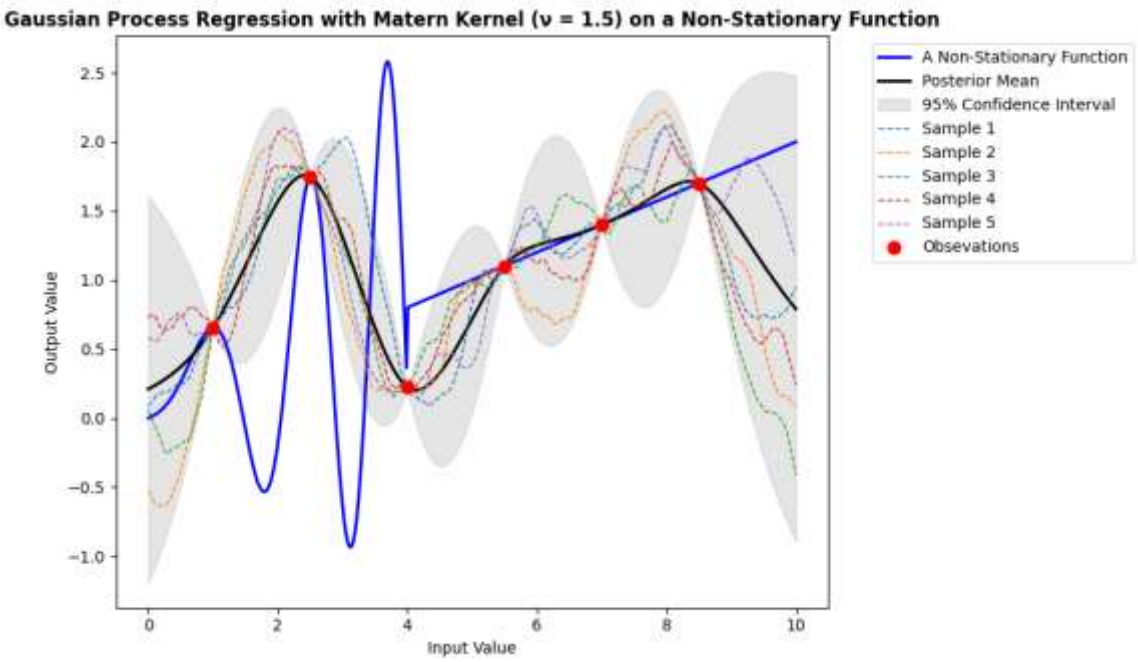


Figure 18 GPR Application with Matérn Kernel on a Non-Stationary Function

Generally, non-stationarity in GP constitutes a much more complex model with a variety of additional parameters and expanded dimensionality [99]. These models arguably tend to lose the interpretability and simplicity provided by stationary GP applications. In addition, non-stationary GP models lead to larger computational costs due to non-stationary covariance matrix operations.

The spatial modeling problem is crucial from the Gaussian Process perspective as well as in its applicability in the collision risk modeling domain. For example, the flight trajectories naturally exhibit non-stationarity in different phases of a flight. Removing nonlinear functions or introducing substantial structural modifications to the underlying function may result in a loss of realism when generating synthetic tracks. Therefore, a

significant portion of this research focuses on the opportunities with non-stationary Gaussian Processes.

Different strategies to address the spatial modeling problem in the context of non-stationarity exhibited by the underlying modeled function are summarized in Table 3. The following subsections describe each method in detail.

Table 3 Strengths and Weaknesses of Methods Addressing Non-Stationarity in GPs

Approach	Method	Strength	Weaknesses
Engineering ad-hoc solutions	Function splitting and linearity removal	(1) Robustness (2) Simplicity	(1) Impractical for complex problems (2) Irreversibly alters the nature and properties of the target function (3) Risk of model overfitting
	Combinations and custom creation of multiple kernel functions ([110])	Tailored solution for specific problems	(1) Custom solutions may lead to overfitting (2) Requires extensive programming expertise (3) Computational complexity may be difficult to estimate
Partitioning methods	Treed partitioning ([111])	(1) Computationally efficient (2) Probabilistic of partition thresholds	Predictions near partition boundaries may be unreliable
	Spatial deformations ([111])	(1) Flexible and adaptive to local variations in the input space (2) Performs well with simple kernel functions	(1) Introduces additional complexity through latent-space modeling (2) High computational cost (3) Reduced interpretability
	Process convolutions ([112], [113])	Smooth transitions between partition regions and boundaries	(1) High computational cost (2) Reduced interpretability
Other GP implementations	Deep Gaussian Process ([104], [105], [106], [107], [108], [109])	Capable of capturing highly non-linear and non-stationary functional behavior	(1) High computational complexity (2) Model instability caused by inter-layer kernel dependencies (3) Risk of overfitting (4) Reduced interpretability

3.3.1.1 Engineering Ad-hoc Solutions

A simple concept to solve a spatial modeling problem is to remove linearity from the model altogether by breaking the function space into separate models, as shown in Figure 19. This approach may improve the fitting of individual models by applying separate and different kernels targeting homogeneous regions of the function. However, the splitting of the functional space irreversibly changes the core function definition. The removal of linear regions in data functions may lead to a loss of understandability and interpretability of the overall concept. Additionally, identifying the border between different types of functional behavior, followed by the removal of the linear data, may be impractical and burdensome.

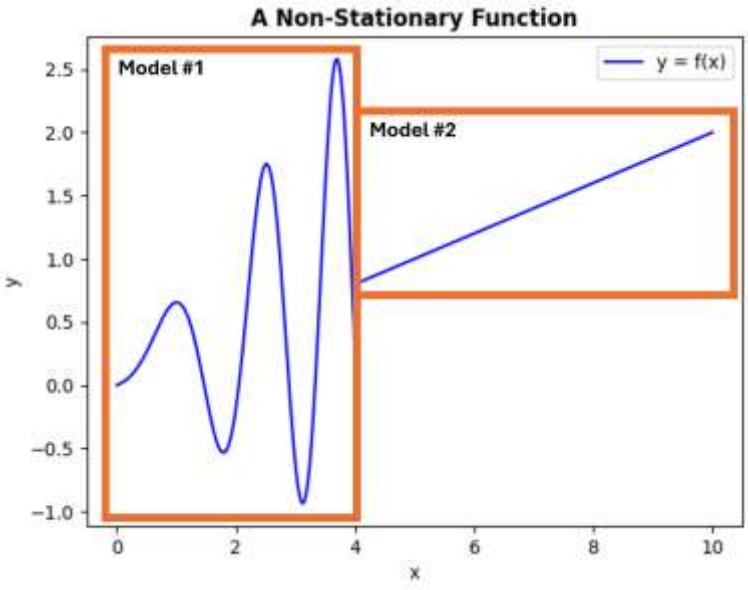


Figure 19 Function Splitting and Linearity Removal

Alternatively, creating custom solutions by combining different kernels is another option to solve the spatial modeling problem in an engineering way. This approach derives a non-stationary kernel function from two or more locally stationary kernels whilst maintaining underlying function continuity [110]. However, like with many Machine Learning problems, it may lead to overfitting. This approach requires extensive knowledge, skills, and expertise in software development. Furthermore, the entirely custom solution would apply only to a specific case, reducing the GPR's flexibility by making the model unresponsive to other scenarios or problems.

3.3.1.2 Partitioning Methods

An alternative solution to the spatial modeling problem is to utilize a hierarchical approach [114], such as a partition model. A hierarchical partition model comprises a collection of sub-models targeting different regions of the input space. The resulting sub-models contain other areas of the original model, where each area can be treated as a stationary process within the partition. A predefined set of functional characteristics, such as differences in smoothness, shape, and scale, describes the variance between the regions. This approach includes *treed partitioning* [111].

Treed partition models recursively divide the input space into smaller subregions. Each new partition is created by splitting a parent partition according to a threshold value of one of the input variables. The threshold is typically selected probabilistically as part of the model construction process.

Treed partition models split the input space iteratively and recursively, so the latest partition belongs to the previous partition. The split is determined by a threshold

value drawn from a probability distribution over the random variable. This is not a concurrent approach, where if one child is partitioned, the others also create children. This method allows us to specify the regions of interest on the input space that can fit separate stationary models without adding redundancy. However, the predictions near the partition boundaries may be less reliable. From the resource standpoint, the treed partition effectively reduces computational effort by handling smaller sub-models. Consequently, the treed partitioning method is well-suited for working with large models.

Other strategies for handling non-stationary models focus on spatial deformations and process convolutions [111].

The spatial deformation method involves transferring non-stationary inputs from their original input space into a latent space, where the Gaussian Process displays stationarity. This deformation reshapes the input data space so that the underlying function appears stationary, allowing application of simpler kernel functions. Additionally, spatially varying kernels allow the covariance function of the Gaussian process to adapt to local variations over the input space. However, the presence of the probabilistic latent layer requires the inclusion of additional methods and techniques, such as neural networks, for feature extraction.

Process convolutions apply spatially varying kernels, mimicking stationarity over non-stationary functions. The process convolutions method adds uncertainty to the kernel function; thus, the resulting Gaussian Process model is flexible but becomes less tailored to the underlying function [113].

Formally [115], a Gaussian Process $f(x)$ with a mean function $\mu(x)$ and kernel function $k(x_1, x_2)$ also utilizes a so-called patch-response function $g: \mathbb{R}^D \to \mathbb{R}$, where D is the size of the input space:

Equation 20 Patch-response Function

$$g \sim GP(\mu(x), k(x_1, x_2))$$

The patch-response function represents a collection of convolutional patches P used in process convolution's domain application [112]. The overall function $f(\cdot)$ corresponds to the sum of all patch responses and is defined as follows:

Equation 21 Overall Patch-response Function

$$f = \sum_{p} g\left(x^{[P]}\right),$$

$$f \sim GP\left(\mu(x), \sum_{p_1=1}^{P} \sum_{p_2=1}^{P} k\left(x_1^{[p]}, x_2^{[p]}\right)\right),$$

where $x_i^{[p]}$ denotes the p^{th} patch used the input space.

By decomposing the initial subset of input points, the process convolution method utilizes a hierarchical approach like other partitioning procedures. However, in contrast with, for example, the treed partitioning, the process convolutions apply the same function $g(\cdot)$ to all decomposed subsets. This allows all patches in the decomposition to inform the cumulative value of the resulting patch-response function. Additionally, the process convolution method provides smooth transitions between the partitioned regions, unlike the treed partitioning.

The last two techniques focused on spatial deformation and process convolutions, significantly increasing GPR flexibility along with computational complexity and loss of

interpretability as a downside. Other methods targeting non-stationarity include considerations of advanced Gaussian Process-based techniques such as Deep Gaussian Process, and Gaussian Process for Dynamic Systems [72].

3.3.2 Multi-Output Problem

Conventional Gaussian Processes-based algorithms produce one output at a time. However, for example, a flight trajectory is simultaneously described by numerous parameters such as lateral and horizontal position, altitude, ground speed, acceleration, orientation, applied forces, time component, among others. This formulates a need to extend a conventional Gaussian framework with capability of producing multiple outputs at once.

In single-output GPR models, the target variable y is represented by $y = f(\mathbf{x}) + \varepsilon,\ x \in \mathcal{X}$, where $\mathbf{x}$ is the input vector in the $\mathcal{X}$ input space, f is the function value, and ε is the Gaussian noise [94]. In the multi-output problem definition, the target variable y must be represented by a vector consisting of multiple functional values for different output parameters:

Equation 22 Multi-output Vector

$$y = [f_1(x), f_2(x), \dots, f_T(x)], \qquad x \in \mathcal{X}$$

where T defines a number of output parameters.

The formulation in Equation 22 is also called a *vector-valued function* [116].

Given the vector-valued function, standard Gaussian Processes implementation models, such as GPR and Deep GP, can be refined to generate a required number of outputs simultaneously. However, the dependencies between outputs add complexity to the model by introducing a complex cross-covariance structure. In conventional Gaussian

Processes, the covariance function defines correlation between a part of inputs, $C(x_1, x_2) \equiv Cov[Y(x_1), Y(x_2)]$. While in multi-output Gaussian Processes, the additional covariance function is required for each pair of outputs:

Equation 23 Covariance Function between Outputs

$$C(y, y_2) \equiv Cov[Y(x_1), Y(x_2)],$$

where $Y(x)$ describe a random variable.

Although the multi-output concept may reduce computational efficiency and interpretability compared to conventional single-output models, it enables the joint modeling of correlated outputs, resulting in more realistic and statistically consistent synthetic data.

3.4 Gap Analysis in Gaussian Processes Literature

This section addresses current gaps in the literature related to the Gaussian Processes.

Gaussian Processes present a powerful framework for generating accurate predictions with their uncertainty estimates [94]. However, multiple challenges arise from the framework's application in the aerospace domain. Commonly, flight trajectories constitute time-series datasets with numerous parameters exhibiting various behaviors, such as a mixture of linear and non-linear patterns, as well as stationary and non-stationary underlying functions. Complexities in trajectory behavior increase the dimensionality of the input space and make model inference more challenging. Furthermore, flight trajectories exhibit complex interdependent patterns that require multi-output implementation. Balancing these issues is an ongoing research effort in [92], [97], [110], [111].

However, most of the literature primarily focuses on reducing computational complexity and its associated cost and execution time, which is important. When addressing high dimensionality and multi-output problems in Gaussian Process applications, some authors decompose complex models into a subset of simpler models, which yields significant computational advantages [93], [111], [117]. The proposed solutions lead to compromises and significant simplifications of the modeled assumptions to achieve the results in a reasonable time, thereby sacrificing the model's fidelity. Nevertheless, the excessive pursuit of realism within a given model often leads to model overfitting, rendering the solution unsuitable for modifications or applications even in a slightly different domain [59].

CHAPTER FOUR: METHODOLOGY

This Chapter describes the selected methodology approach organized as follows. Section 4.1 outlines steps required for historical data collection and processing. Section 4.2 describes the process for generating synthetic flight trajectories using Gaussian Processes.

A generic process for collision risk analysis is summarized in Figure 20. First, this method requires collecting reliable historical data at an airport or in a geographical area of interest for departure, approach, or a mix of both procedures. Notably, the methodology described in this section applies to operations in proximity to airports and excludes collision risk assessment during the en-route flight phase. Next, the historical flight data are used to train a ML model to generate flight trajectories. Last, the synthetic flight trajectories are used as an input to the collision model to estimate the risk between pairs of flight trajectories on a collision course.

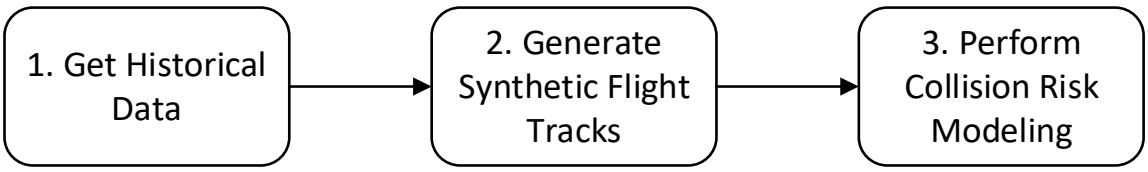


Figure 20 Algorithm for Performing Collision Risk Modeling

Getting a reliable historical dataset is a prerequisite for generating flight trajectories. This research focuses on methods for generating synthetic flight tracks, while the collision risk modeling is deferred to future work. The subsequent sections describe the method in detail. The subsequent sections describe the process in detail.

4.1 HISTORICAL DATA

The first step in collision risk modeling is to collect reliable historical flight trajectory data for a given airport from open sources, such as OpenSky Network [113], ADS-B Exchange [114], or FlightAware [115]. Since each source of ADS-B data has its own limitations, subsequent steps, summarized in Figure 21 and outlined below, are required to ensure the quality of the collected data.

Step 1. Draw a 36 nautical miles (nm) bounding box around the given airport.

The output of this step provides four pairs of coordinates (i.e., latitude and longitude) each direction from the airport: Northeast, Southeast, Southwest, and Northwest, clockwise. While the airport lies within the bounding box, a 36 nm radius in each direction captures departing and arriving flight trajectories. The 36 nm radius is deemed sufficient to capture immediate operations around the airport of interest, but the radius can be expanded if necessary. Tools like Google Maps can be used to estimate the bounding box coordinates.

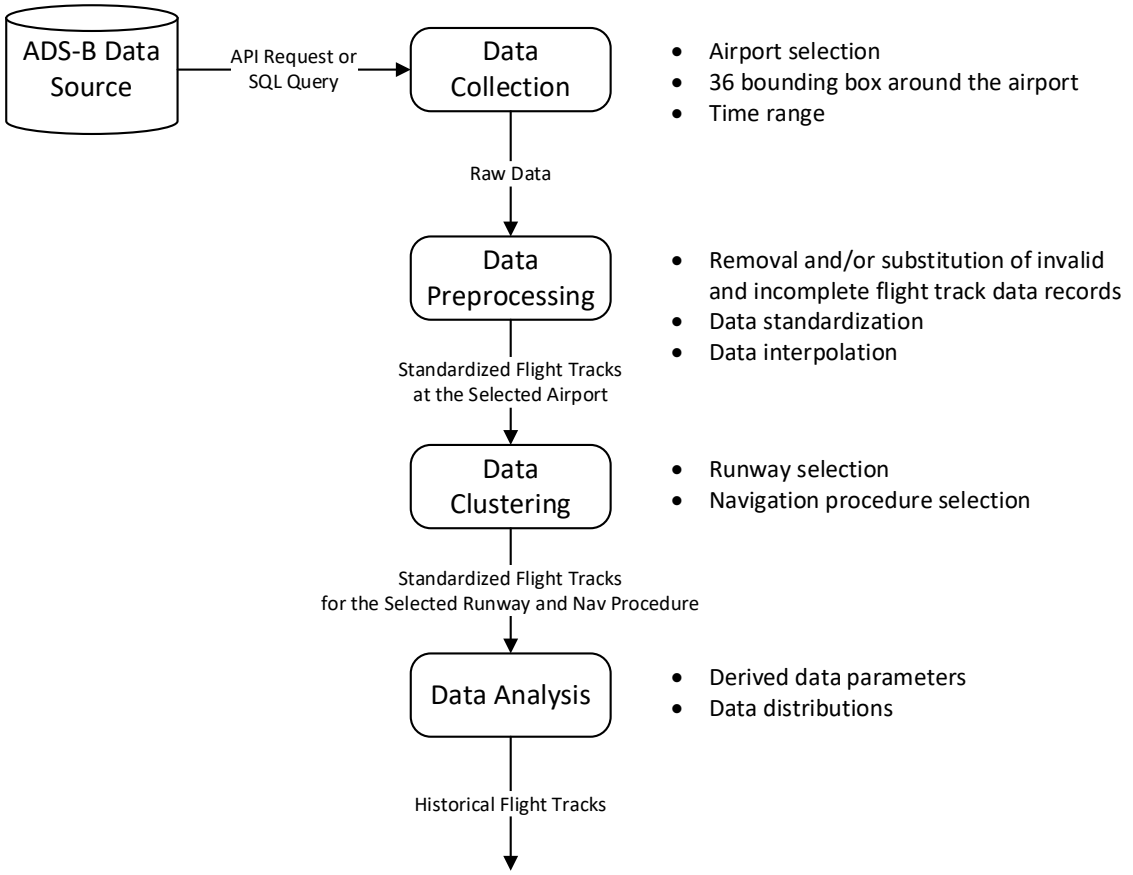


Figure 21 Historical Flight Track Data Processing

Step 2. Select a time range for historical data.

The general recommendation is to select at least three months of data from a peak season to ensure a larger sample size and diversity among historical flight trajectories. For example, according to the Bureau of Transportation Statistics (BTS) [118], monthly data show that passenger enplanements typically peak in the summer months and decline slightly in late winter and early spring.

Step 3. Collect the historical data from an open source database.

Based on the source configuration, a database query or an API request is required to collect flight data with the parameters identified in the previous steps. At a minimum, the data should include the parameters for date, time, flight identifier, flight position (i.e., latitude and longitude), altitude, and velocity. Airport and runway identifiers for departing and landing aircraft should also be included as additional parameters, if available.

The collected raw data can be stored in a .csv file or another data-table-readable format. Notably, these files may contain multiple Gigabytes of data, which requires extensive computer resources such as memory and data storage capacity.

Step 4. Preprocess the data.

Since each data source has its own limitations, data preprocessing is required to ensure the quality of the collected data. It is recommended to filter the data to create separate homogeneous dataset focused on either departing or approaching flight trajectories. For example, filtering flight trajectories by altitude with an ascending pattern yields data for approaching trajectories.

Further preprocessing considerations include removing noise from the data, such as unreasonable flight patterns and incomplete or invalid flight trajectories. Additionally, outliers and unreasonable single track data records can be replaced or smoothed along the flight track.

Step 5. Standardize the total number of data points for each flight track.

Since the number of data points for each collected flight trajectory varies, data standardization is performed to ensure that all tracks in the dataset have the same total

number of data points. Data standardization is required to produce fixed-sized flight tracks, which substantially simplifies the subsequent processing of input flight trajectories by machine learning models. Furthermore, this standardization makes the flight tracks easier to store, manipulate, compare, cluster, and visualize.

The detailed algorithm to perform data standardization for each flight track in the dataset is outlined in Aref et al. [64]. The algorithm standardizes flight tracks so that every track contains the same number of points, regardless of its original length or number of recorded observations. It does this by creating evenly spaced segments along the cumulative distance traveled.

First, data standardization begins with computing the along-track distance. The great-circle distance [119] is computed between consecutive points on the flight track given their latitudes and longitudes (Equation 24). Then, the individual distances between points are summed to produce the along-track distance for the portion of the recorded flight trajectories in the collected dataset.

Equation 24 Great-Circle Distance

$$d = R \cdot arccos(\sin(lat_1)\sin(lat_2) + \cos(lat_1)\cos(lat_2)\cos(lon_1 - lon_2)),$$

where $R = 6371.137\ km$ is the Earth's radius.

Second, the track is divided into a predefined number of segments, where each segment represents the target distance between consecutive standardized points.

For example, when the track length is 1000 nm and the number of segments is 200, the target distance is $\frac{1000}{200} = 5\ nm$.

The initial recorded point in the dataset represents longitude, latitude, and altitude (x_A, y_A, z_A). Given the initial recorded point and the target distance of the segment, the next point (x_B, y_B, z_B) is determined on the boundaries of the segment's length. If the calculated distance between points A and B is less than or equal to the target distance of the segment, then point B can be included in the standardized track data without adjustments. Otherwise, a set of equations below (Equation 25) is required to interpolate coordinates and altitude for the next standardized point along the track [64]. In this equation, x_{New}, y, and z_{New} represent longitude, latitude, and altitude of the new standardized point along the track, respectively. This new point is added to the standardized track data.

Equation 25 Equations for New Standardized Point

$$x_{New} = x_B + \left(\frac{d - d_{AB}}{d_{AB}}\right)(x_B - x_A),$$

$$y_{New} = y_B + \left(\frac{d - d_{AB}}{d_{AB}}\right)(y_B - y_A),$$

$$z_{New} = z_B + \left(\frac{d - d_{AB}}{d_{AB}}\right)(z_B - z_A).$$

Standardized interpolation between two points along the flight track is shown in Figure 22 [64]. The steps above must be performed until standardized tracks are formed with evenly spaced points along the entire track.

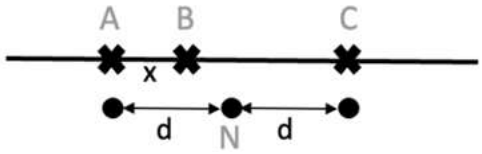


Figure 22 Interpolation between Two Points

Step 6. Standardize the endpoint for each track in the dataset.

Since the ADS-B data contains inherent noise, use the same technique outlined in Step 5 to interpolate the last point of the track. This ensures that all tracks are recorded at the runway threshold, excluding on-ground maneuvers.

Step 7. Apply data clustering to select departure or arrival procedures for a specific runway, direction, or other navigational patterns of interest.

While there are many methods to apply data clustering, it is recommended to use the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. DBSCAN is an unsupervised machine learning clustering method for grouping densely packed data points [120]. Unlike centroid-based clustering methods such as K-means, DBSCAN does not require the number of clusters to be specified in advance and can detect clusters with arbitrary shapes. These characteristics make DCSCAN especially suitable for analyzing trajectory data due to tracks exhibiting complex spatial patterns.

The DBSCAN algorithm contains two parameters: the neighborhood radius (ε) and the minimum number of samples [121]. The epsilon (ε) defines the radius of a neighborhood around a given data point. The minimum number of data points within a specific radius is required for a point to be considered a core point. Core points form the

basis of clusters and can connect to other nearby core points. Points that fall inside the neighborhood of a core point but do not satisfy the minimum number of points requirements are classified as border points and are assigned to the corresponding cluster. Points that are neither core points nor border points are considered noise or outliers. Appropriate selection of ε and the minimum number of samples is critical, as these parameters directly influence the desired number, size, and density of the resulting clusters. For example, smaller values for ε and the minimum number of samples yield a larger number of clusters with more unique features.

When clustering arriving flight tracks, the last recorded positions in the track data can be used to identify the landing runway. Once clusters are formed, each cluster can be associated with a specific runway, enabling automatic classification of arriving trajectories by landing runway or navigation procedure.

Step 8. Analyze the historical data.

The next step in processing historical flight tracks is deriving analytical distributions. For example, lateral, vertical, and speed dispersion measurements for arriving flight trajectories should be collected at 0, 3, 6, and 9 nm away from the runway threshold. These analytical distributions are required for comparison and validation of the generated synthetic data in the following steps. In addition, if published airport standard procedures are available, they can be used for comparison of the measurements at known waypoint locations.

Last, various data parameters are derived based on original parameters in the historical flight track data (Table 4).

Table 4 Derived Data Parameters

Original Parameter	Derived Parameter	Value
Heading	Course change	Greater than 5 degrees; Boolean indicator
	Magnitude of course change	Heading value
	Cumulative magnitude of course change	A sum of heading values per segment
Altitude	Level flight, ascending segment, or descending segment	Greater than 25 feet; Boolean indicator
	Magnitude of ascend/descend	Altitude value
	Cumulative magnitude of altitude change	A sum of altitude values per segment
Velocity	Constant speed, acceleration segment, or deceleration segment	Greater than 2 knots; Boolean indicator
	Magnitude of acceleration/ deceleration	Ground speed value
	Cumulative magnitude of speed change	A sum of ground speed values per segment

4.2 SYNTHETIC FLIGHT TRAJECTORIES

The selected methodology for generating synthetic flight tracks is based on a reusable modular framework. The framework's architecture consists of *Learning*, *Generating*, and *Validation* modules, with the process flow summarized in Figure 23. The benefits of a modular approach include the ability to use different machine learning models and algorithms at the core of the Learning module, while applying the same or highly similar techniques to generate and validate synthetic data. This dissertation focuses on the application of Gaussian Processes (GP) to generate synthetic flight tracks. Therefore, the methodology outlined in this section is tailored for GP-based algorithms in the Learning module.

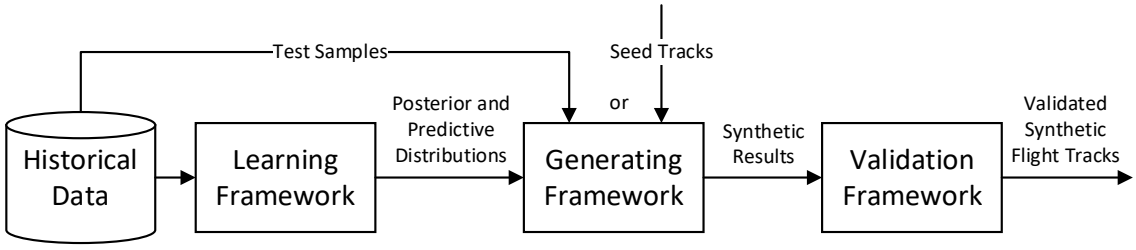


Figure 23 Modular Framework

In Figure 23, the historical flight tracks serve as inputs to the Learning module to train a model with an underlying GP function. The Learning module provides GP posterior and predictive distributions. These distributions, along with test or seed tracks, are used in the Generating Framework to provide millions of new flight tracks. The test

samples, which were not used for model training, are selected from historical data. The test tracks allow verification of the performance, responsiveness to the training, and overall quality of the results of the GP model. If the model's performance is deemed inadequate, adjustments to the training model are required. However, when the model performance meets the verification metrics, seed tracks can be used to generate synthetic data. The seed tracks represent a simulated flight procedure whose features need to be replicated and studied in future collision risk analysis. Finally, the synthetic flight tracks retrieved from the Generating module are evaluated for realism within the Validation framework. The following subsections describe the framework in detail.

4.2.1 Learning Framework

The Learning framework incorporates a structured set of methods for constructing a GP model by performing common supervised machine learning tasks and activities. The framework ensures that the resulting model is both accurate and robust, while properly capturing uncertainty in predictions. The main activities performed in the Learning phase are outlined in the following sections.

4.2.1.1 Predictor and Response Data Selection

Domain knowledge and study objectives dictate a selection of predictors (i.e., inputs) and responses (i.e., outputs) for predictive modeling. In supervised machine learning, such as in GP-based algorithms, the mapping between inputs and outputs is critical for model training.

A collection of relevant predictors ensures that the model captures the most informative relationships while avoiding redundant or noisy features. In contrast, poorly

selected predictors tend to either obscure the GP model performance or lead to overfitting.

In the context of GP, a large number of predictors increases the model complexity due to high-dimensional matrix operations. Conversely, a smaller set of inputs simplifies data manipulation but risks overlooking important details and dependencies linked to the omitted parameters.

Notably, aircraft trajectories in collision risk analysis require at least latitude, longitude, altitude, and speed. Therefore, the response data parameters must be selected accordingly. Other flight path parameters, such as heading or duration, can be derived from trajectory data. When the modeling approach requires the response data transformation (e.g., logarithmic scaling, differencing), outputs are restructured accordingly. The response reconstruction ensures compatibility with GP assumptions (e.g., smoothness, continuity) and can improve model stability and predictive performance.

4.2.1.2 Gaussian Process (GP) Implementation

A specific GP implementation is applied based on the model objectives and response data type. For example, Gaussian Process Classification (GPC) is used for prediction models with categorical output [122]. Although the GPC implementation is not applicable for flight trajectory generation, it provides important insights into trajectory behavior, such as recognition of course or flight segment changes. Alternatively, Gaussian Process Regression (GPR) and its variants, such as the Deep GP algorithm, are suitable for problems with continuous output (i.e., numerical time-series response data).

Given the time-series type of trajectory data with continuous outputs, this dissertation did not use GPC, instead focusing on GPR and its variants.

4.2.1.2.1 Single Target Response

A mathematically stable GP model generates predictions with associated uncertainty estimates for a single target response variable. The same principle applies to both GPC and GPR models. In the context of trajectory data parameters, the process involves implementing multiple GPR models to generate synthetic samples for each target separately. However, this approach lacks dependencies among separately generated outputs, such as latitude, longitude, altitude, and speed.

Specifically, when combining generated samples from individual GPR models for collision risk analysis, inconsistencies between lateral trajectories and their associated altitude and speed parameters may be observed. Although models for latitude and longitude rely on the same set of inputs, inconsistencies in the separately generated samples are attributed to differences in underlying GP function behavior, settings, and noise levels. Advanced indexing techniques may be necessary to combine responses from separate models into individual flight tracks, ensuring aerodynamic consistency in the resulting data. In addition, repeated model training on the same predictor set results in inefficient use of computational and processing resources.

4.2.1.2.2 Multiple Responses

An alternative approach to single target response is to generate flight tracks simultaneously and in a dependent manner. The method incorporates response data transformation to encompass multiple parameters into a single feature. The resulting

model meets the required threshold of realism in the synthetic solution by incorporating dependencies between outputs, while adhering to GP mathematical principles by producing a single encoded multi-response function. However, this solution introduces several challenges.

First, response transformation techniques depend on the specific problem domain and data availability, requiring careful consideration on a case-by-case basis. Second, the transformed outputs require subsequent conversion to their original units to facilitate the interpretability of the results. Consequently, extra steps are necessary to encode and decode the response data, further complicating the modeling process. Finally, the complexities introduced into the response data structure tend to obscure the GPR model, thereby reducing performance.

As an example, two separate GPC models predict correlated Boolean outputs. Each output represents a *True/False* value, resulting in a two-category classification performed by each model. An encoded multi-response data set consists of a pair of Boolean values combined into a single feature, resulting in four categories: *True+True*, *True+False*, *False+True*, and *False+False*, where the values in each class correspond to the first and second outputs, respectively. The model trains on the encoded response data and generates the associated predictions. Afterward, the predictions are decoded by splitting the response into two separate values based on the position in the class.

4.2.1.2.3 Principal Component Analysis

Another option for combining multiple responses into a single target variable is to employ the Principal Component Analysis (PCA) technique. PCA for regression models

defines a dimensionality-reduction technique in GP models with high-dimensional input space. Conventionally, PCA mitigates the curse of dimensionality in predictor sets with highly correlated features (e.g., descending altitude and deceleration speed segments in arrival flight trajectories) by reducing multicollinearity and noise. Additionally, PCA reduces the computational burden of multi-dimensional matrix calculations by performing kernel hyperparameter operations on fewer dimensions. Hence, PCA compresses the feature space, while GPR performs flexible nonlinear regression in the reduced space.

Mathematically, given an input matrix with n observations and p predicting features, $X \in \mathbb{R}^{n \times p}$, PCA finds orthogonal directions that maximize variance. The component scores Z contain eigenvectors W of the covariance matrix, $Z = XW$. The first component explains the largest amount of variance, while the second component explains the next largest variance, and so on. Usually, only the first k components are retained, $k < p$, so most of the original variance in features is preserved.

Principal Components (PCs) can be tailored to describe multiple response data parameters, such as latitude and longitude, as a single output. First, responses across the model, including train and test samples, are standardized using $X_{st\ dev} = \frac{X-\mu}{\sigma}$, to ensure all variables contribute equally. Then, the analysis computes PCs and retains k components using $Z_k = PCA(X_{st\ dev})$, where $Z_k \in \mathbb{R}^{n \times p}$. Finally, a GPR model learns a mapping between predictors and a single response that represents multiple combined outputs.

The challenges of employing PCA for flight trajectories include potential information loss and reduced model interpretability. While this method maximizes variance in the response variables, PCA may remove predictive information from low-variance components. PCA transforms original features of the response data into linear combinations. This response data transformation via PCA encoding reduces the interpretability of the results, necessitating subsequent decoding steps.

4.2.1.3 Data Stabilization and Normalization

Aircraft trajectories present a wide range of units in their parameters. For example, altitude differences along the flight path are measured in hundreds of feet. At the same time, changes in coordinates result in decimal-point variations, creating an unstable input space for unit measurements. Therefore, both predictors and response variables require scaling (e.g., standardization to zero mean and unit variance, or min–max normalization). Data normalization is particularly important for GP models because kernel functions are sensitive to input scale, and improper scaling can lead to poor numerical conditioning and suboptimal hyperparameter estimates.

4.2.1.4 Kernel Function Design

A kernel function defines the similarity or covariance between data points of the input space in GP algorithms [98]. The kernel plays a central role in determining the prior assumptions about the underlying function (i.e., the response), including its smoothness, periodicity, continuity, and overall behavior. Commonly used kernels include the Radial Basis Function (RBF), Matérn, Rational Quadratic, and periodic kernels. The choice of kernel determines the GP model's ability to accurately represent different patterns within

the data. For example, periodic kernels are suited to repeating patterns in the underlying function (e.g., sine- or cosine-type oscillations) [94]. In contrast, Matérn kernels can model rougher, less smooth functions that frequently occur in real-world applications.

Kernel functions reflect individual characteristics of the corresponding dataset, requiring a selection of specific kernels for each constructed model. For aircraft trajectory prediction and modeling, the ***Matérn kernel*** function is commonly used as the prior distribution because it offers greater flexibility than the RBF kernel in representing trajectories with abrupt changes, irregular movements, and varying levels of smoothness. As described in Section 3.1.3.2 Matérn Kernel, the Matérn kernel comprises the signal variance, length scale, and smoothness hyperparameters [97].

The *signal variance* σ^2, often represented by a scaling factor or kernel amplitude, controls the overall variance of the functions sampled from the prior distribution. Large scaling factors, such as 10^2, allow the generated functions to exhibit substantial variation around their mean, resulting in larger amplitudes and greater uncertainty. In contrast, smaller scaling factors constrain the function amplitudes, implying that the underlying process is expected to remain closer to its mean. Therefore, the scaling factor directly influences the magnitude of deviations observed in the flight trajectories.

The *length scale* parameter characterizes the degree of fluctuations in the underlying function. It determines how quickly the function changes with respect to the input variables. Large length-scale values, typically $\ell \geq 10$, indicate that observations remain correlated over long distances and that functions change slowly over the input space. Such models capture gradual changes in flight trajectories and long-term motion

trends. Conversely, using smaller length scales, $\ell \leq 1$, assumes that the underlying functions vary rapidly over short distances, making the function more sensitive to nearby observations. Consequently, smaller length scales are useful for capturing abrupt maneuvers or short-term deviations from the nominal flight path.

The *smoothness* parameter controls the differentiability of the underlying function. Unlike the RBF kernel, which assumes infinite differentiability, the Matérn kernel allows control of the roughness of the generated functions. The smoothness parameter v of 0.5 indicates that the corresponding functions are not expected to be smooth or differentiable, thereby capturing abrupt changes and irregular patterns in the underlying functions. Such functions are particularly useful for modeling noisy or highly dynamic systems. Before observing the training data, samples from the Matérn kernel with $v = 0.5$ are shown in Figure 24.

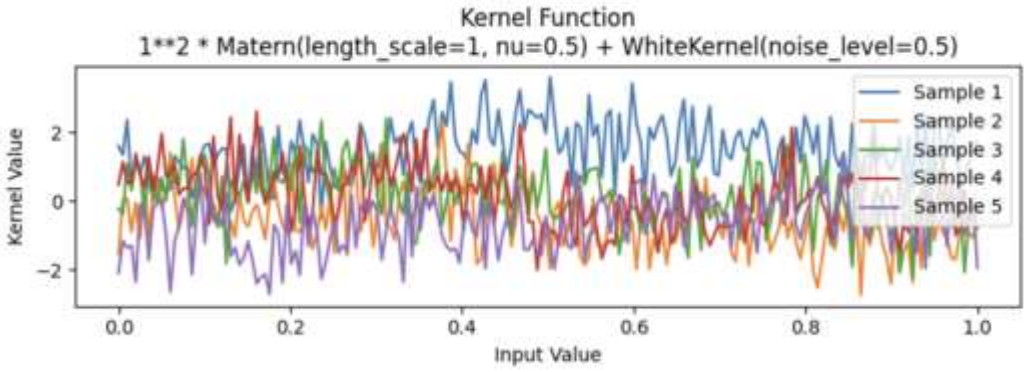


Figure 24 Matérn Kernel with $v = 0.5$

As the smoothness parameter increases, the resulting functions become progressively smoother. For example, the smoothness of 1.5 indicates moderate roughness; thus, the underlying function is once differentiable. Similarly, $v = 2.5$ produces twice-differentiable functions that are noticeably smoother yet remain more flexible than those obtained using the RBF kernel.

In contrast, the ***Radial Basis Function (RBF) kernel***, also known as the Squared Exponential kernel, assumes that the underlying function is infinitely differentiable and very smooth. The RBF kernel is particularly suitable when the modeled process is expected to evolve continuously without abrupt changes. However, this strong smoothness assumption can become restrictive in applications involving aircraft trajectories, where sudden maneuvers, external disturbances, and operational constraints may lead to non-smooth behavior. For higher values of the smoothness parameter, such as $v = 2.5$ or larger, the Matérn kernel increasingly resembles the behavior of the RBF kernel.

Another commonly used covariance function is the ***Rational Quadratic kernel***, which can be interpreted as a scale mixture of infinitely many RBF kernels with different length scales. Unlike a single RBF kernel, which assumes a single characteristic length scale across the dataset, the Rational Quadratic kernel can model processes that vary across multiple spatial or temporal scales. This property makes it particularly useful for complex trajectory datasets where both long-term trends and short-term local variations are present simultaneously.

To improve generalization and avoid overfitting, covariance functions often include the ***White kernel***. The White kernel models observation noise by introducing independent Gaussian noise to each measurement. This component acknowledges that real-world sensor measurements, including position, altitude, and velocity observations from aircraft systems, are inherently noisy. By accounting for measurement uncertainty, the GP avoids fitting every minor fluctuation in the training data and produces more robust predictions.

Combining multiple kernels often leads to more expressive models. For instance, a composite kernel formed by the sum or product of a Matérn kernel and a Rational Quadratic kernel can simultaneously capture local irregularities, varying smoothness, and multi-scale behavior. Such hybrid kernel structures provide greater flexibility in modeling aircraft trajectories, enabling the GP to represent both gradual changes in flight dynamics and sudden deviations due to pilot actions, weather conditions, or air traffic control instructions. Samples drawn from the composite kernel consisting of the Matérn and Rational Quadratic kernels are shown in Figure 25.

In summary, kernel functions, their strength, and applicability for flight trajectory data modeling are provided in Table 5.

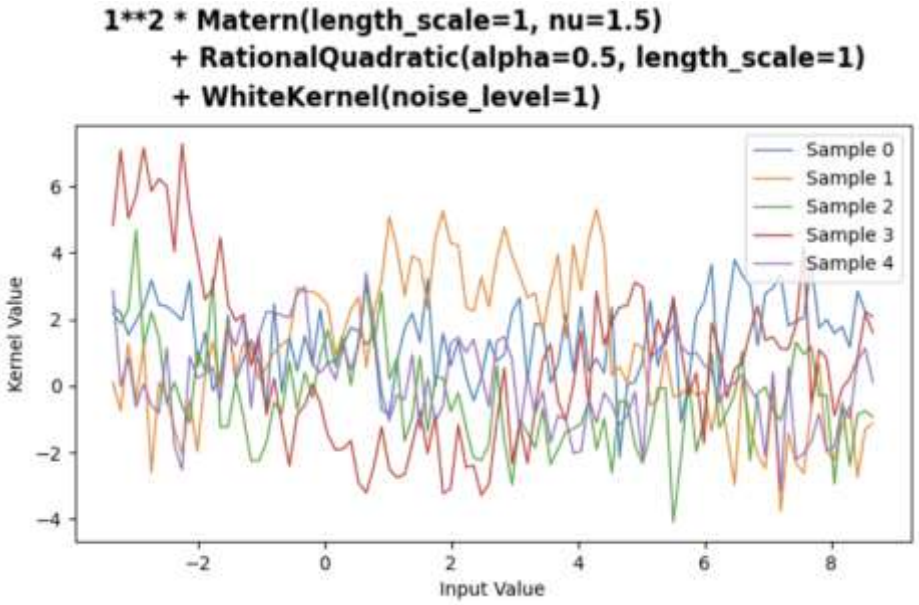


Figure 25 Composite Kernel with Matérn and Rational Quadratic Functions

Table 5 Strength and Applicability of Kernel Functions

Kernel Function	Strength	Applicability
Matérn Kernel	Flexible smoothness control	Flight trajectory profiles with abrupt changes and varying dynamics
RBF Kernel	Very smooth functions	Continuous processes with no sudden maneuvers
Rational Quadratic Kernel	Multi-scale modeling	Data containing both local fluctuations and global trends
White Kernel	Noise modeling	Handling inherent noise in ADS-B data and measurement uncertainty
Composite Kernels	Combines flexibility of different kernel functions and multi-scale behavior	Complex aircraft trajectories with both smooth trends and sudden maneuvers

4.2.1.5 Log Marginal Likelihood Maximization

Unlike parametric machine learning models that fit explicit model coefficients, a GP model learns the characteristics of the covariance function through tuning of kernel hyperparameters. These hyperparameters determine the behavior of the prior distribution and significantly influence the model's predictive performance, uncertainty estimates, and generalization capability.

The hyperparameter optimization maximizes the log marginal likelihood [94], also known as the model evidence. The log marginal likelihood measures model fit by computing the logarithm of the probability of the observed data under the GP model, integrated (or, equivalently, marginalized) over all possible latent functions. Unlike conventional machine learning models that rely on a predefined validation set for hyperparameter tuning, GPs provide a probabilistic framework that directly evaluates the likelihood of the observed data given a set of hyperparameters. The optimization process therefore seeks the hyperparameter values that maximize the probability of observing the training data while maintaining an appropriate level of model complexity. A large positive log marginal likelihood generally indicates that the model explains the observed data well. Efficient optimization techniques (e.g., gradient-based methods) are typically used.

During optimization, the hyperparameters are iteratively adjusted to maximize the log marginal likelihood. For example:

- Increasing the length scale leads to smoother functions and stronger correlations between distant observations.

- Increasing the signal variance permits larger deviations from the mean function and greater output variability.
- Increasing the noise variance allows the model to attribute more variation to measurement noise rather than the underlying process.
- Modifying the smoothness parameter in the Matérn kernel changes the degree of differentiability and roughness of the latent function.

As an output, the Learning framework provides a trained and optimized GP model.

4.2.2 Generating Framework

Given a trained GP model from the Learning framework, the Generating framework builds posterior and predictive distributions for seed tracks whose features need to be predicted. The model was trained on historical data that provide insights into flight patterns and aerodynamic characteristics of the tracks. The model uses seed tracks, which represent a modified navigation procedure not included in the training dataset, to predict and replicate millions of new samples.

The framework comprises methods for re-transformation of predictors and response variables to their original units. Data rescaling and reconstruction enable the generation of predictions and associated uncertainty estimates from the predictive distributions in an interpretable format. Then, the synthetic flight track data parameters are sampled from the posterior distributions using confidence intervals or empirical ranges based on standard-deviation bounds. Synthetic samples are also reconstructed

and/or decoded to a format consistent with data re-transformation and encodings applied in the Learning Framework.

During posterior sampling, the confidence intervals or standard deviation ranges control the degree of variation in the generated synthetic parameters. An expansion of uncertainty bounds increases variance and diversity in the output data, which are necessary for the large sample size in the collision risk analysis. However, wide corridors for sampling synthetic parameters may lead to inclusion of inconsistent trajectory patterns, such as unrealistic altitude or speed indications on the final approach segment. Therefore, a trade-off analysis between variance in posterior samples and their reasonableness is required to ensure a stable solution and limit unrealistic outliers in the confidence intervals.

In addition to the bounds control, the posterior samples require additional post-processing. While the predictive or posterior mean can resemble a reasonable flight-track parameter, individual samples from the posterior distribution may exhibit some roughness. This resulting roughness can be attributed to the kernels used in the underlying GP function (e.g., the Matérn kernel). Therefore, posterior samples can be combined into batches to reach a required level of smoothness in the output data.

The raw samples form a batch of a selected number of samples. Then, the batch's mean is used as a representative of the synthetic parameter. However, the number of posterior samples in a batch requires careful determination. Even though batching limits unreasonable trajectory patterns, it significantly reduces the sample size of the generated synthetic flight track data parameters. Moreover, the generating algorithm requires

substantially more posterior samples to produce the same number of synthetic tracks, thereby increasing the computational burden.

Eventually, the Generating framework collects multiple verification metrics to evaluate the GP model's responsiveness and performance, with the list provided in Table 6.

The primary metrics comprise Root Mean Squared Error (RMSE) [123], the coefficient of determination (R^2) [124], and the Continuous Ranked Probability Score (CRPS) [125]. RMSE quantifies the model's mean squared error, with lower values indicating lower prediction error. The R^2 score measures the proportion of variance in the response variable explained by the model, with values closer to one indicating stronger explanatory power. CRPS assesses the quality of probabilistic forecasts by comparing the predicted cumulative distribution function (CDF) with the observed value. On a normalized scale, CRPS values are expected to be lower than RMSE. A CRPS value of zero represents a perfect forecast, while values in the range of 0.05 to 0.1 indicate reasonable predictive capability.

If (1) the GP model passes verification criteria, provided by metrics in Table 6, and (2) the preliminary analysis finds the generated data suitable for further evaluation, then the Validation framework evaluates the synthetic flight tracks for realism. Otherwise, the model needs to be re-trained in the Learning framework using different kernel functions, predictor set, response structure, or entirely different GP implementation approach.

Table 6 Model Performance Metrics

Score	Description	Units	Range	Interpretation (Good Value)
Mean Absolute Error (MAE)	Measures average absolute difference between actual and predicted values: $\frac{1}{n}\sum_{i=1}^{n}\lvert y-\hat{y}_i\rvert$	Same as the target variable (original units)	[0, ∞)	• A good score is smaller than the target's typical value • Less sensitive to outliers
Mean Squared Error (MSE)	Measures average squared difference between actual and predicted values: $\frac{1}{n}\sum_{i=1}^{n}(y-\hat{y}_i)^2$	Squared units of the target variable (difficult to interpret)	[0, ∞)	• A good score is closer to 0 • Sensitive to outliers • Useful when large errors are especially undesirable
Root Mean Squared Error (RMSE)	Measures the squared root of MSE: $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y-\hat{y}_i)^2}$	Same as the target variable (original units)	[0, ∞)	• A good score is closer to 0 • Sensitive to outliers • Interpretable in original units
R^2 (Coefficient of Determination)	Measures how well the model explains the variance in the target variable: $1-\frac{SS_{reg}}{SS_{total}}$, $SS_{reg}=\sum_{i=1}^{n}(y-\hat{y}_i)^2$ $SS_{total}=\sum_{i=1}^{n}(y-\hat{y}_i)^2$	Unitless	[-∞,1]	• A good score is closer to 1, with the score of 1 meaning a perfect fit • The score of 0 means same as the predicting mean • The score <0 is worse than mean
Explained Variance	Measures of how much of the variance in the target variable is explained by the model: $1-\frac{Var(y-\hat{y})}{Var(y)}$	Percentage	[0,1]	• A good score is closer to 1 and R^2 • The score of 1 means perfect fit

Score	Description	Units	Range	Interpretation (Good Value)
Continuous Ranked Probability Score (CRPS)	• Measures difference between predicted CDF and observed value • Generalizes MAE to probabilistic forecasts	Sample as the target variable (unless on the normalized scale)	[0, ∞) or [0,1] if normalized	CRPS should be lower than RMSE for the same normalized scale. For Min-Max normalized data (0–1): • CRPS of 0 means a perfect forecast • CRPS of 0.01–0.05 is excellent • CRPS of 0.05–0.10 is reasonable

4.2.3 Validation Framework

The Validation Framework quantifies the quality and fidelity of synthetic flight tracks. The framework operates on the number of collected parameters and their distributions in both historical and synthetic datasets.

First, the framework collects measurements of lateral, vertical, and speed deviations at 3, 6, and 9 nm from the runway threshold in both historical and synthetic flight tracks. Second, the along track distance is computed for each synthetic track and subsequently compared with historical figures. Next, individual flight track characteristics, such as the number of courses, levels, and speed changes, are accounted for across all tracks in accordance with Table 4. An indication of course changes greater than 5° derives from the heading data, while levels greater than 25 feet and speed changes greater than 2 knots are based on altitude and speed profiles, respectively. Last, the historical and synthetic data distributions for all collected parameters are compared using distribution comparison metrics.

In addition to the measurements collected above, the framework applies waypoint data from published navigation procedures for the selected airport to the synthetic tracks. The existing waypoint data allow for the collection of additional measurements of lateral, vertical, and speed deviations at the applicable segments of the synthetic flight tracks. Hence, the distributions at waypoints can be used to further validate the generated data.

The primary metrics for distribution comparison include the Wasserstein Distance, the Kolmogorov–Smirnov (KS) Statistics, and the Jensen–Shannon Divergence (JSD).

The Wasserstein distance [126] measures the cost of an optimal transport plan to convert one distribution into the other. Because the Wasserstein distance is sensitive to the geometry of distributions, it indicates the similarity in shape between historical and synthetic distributions. When these distributions are identical, the Wasserstein distance is zero.

The KS Statistics measures the maximum difference between two CDFs and captures the single largest deviation [127]. KS Statistics values range from 0 to 1, with 0 indicating that the CDFs are identical. On a normalized scale, KS Statistics scores less than 0.2 show that the CDFs are moderately different, while values greater than 0.3 indicate a significant statistical difference.

The JSD [128] measures similarity between two probability distributions using symmetric, smoothed Kullback-Leibler (KL) divergence. Similar to the KS Statistics, lower values indicate greater similarity between the distributions, while values greater than 0.3 indicate significant differences.

If the synthetic flight tracks satisfy validation criteria, they can be incorporated into a collision risk model with a degree of certainty in accuracy, quality, and realism of the artificially generated data.

CHAPTER FIVE: RESULTS

This Chapter presents findings and results for generating synthetic flight trajectories using a case study example. The case study applies arrival procedures at Ronald Reagan Washington National Airport (ICAO: KDCA; IATA: DCA) in Arlington, Virginia, to generate synthetic flight trajectories.

The Chapter is organized as follows. Section 5.1 describes the historical data collected for the case study. Section 5.2 demonstrates how GPR and its variants can be used to generate synthetic flight tracks.

The following sections outline the detailed findings and results achieved from the case study.

5.1 HISTORICAL FLIGHT TRACK DATA FOR DCA AIRPORT

Python scripts for historical data preprocessing are provided in Appendix E – Python Scripts for Historical Data.

5.1.1 Data Collection

In accordance with the methodology described in Chapter 4, the first step was to collect historical flight trajectory data at DCA airport.

A 36 nm bounding box surrounding DCA airport is shown in Figure 26. The four pairs of coordinates at the corners of the bounding box were required to capture flight traffic in the DCA region:

- DCA airport: 38.8514403, -77.0377214
- Northeastern corner: 39.4518486486, -76.2689290239
- Southeastern corner: 38.2505513514, -76.2689290239
- Southwestern corner: 38.2505513514, -77.8114709761
- Northwestern corner: 39.4518486486, -77.8114709761

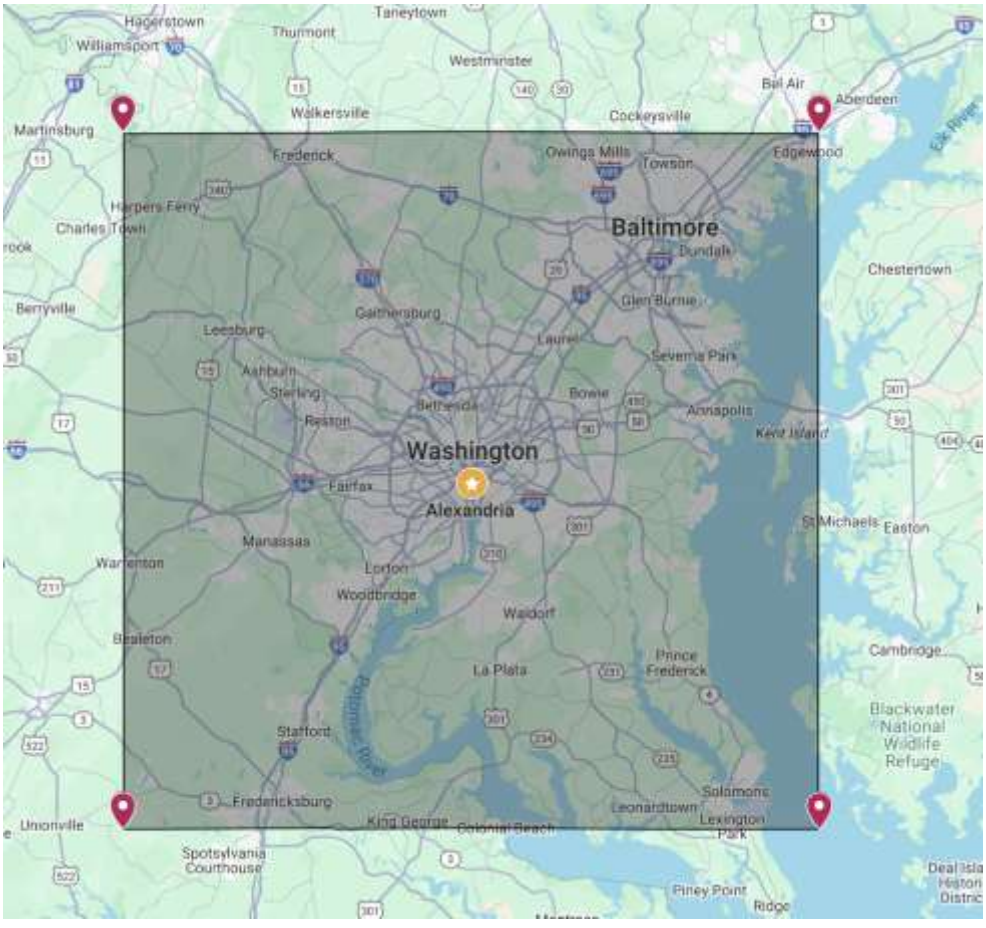


Figure 26 36 NM Bounding Box around DCA Airport

The data were collected in the busiest summer months of the year: from June 1st to August 31st, 2025. The coordinates at the box boundaries, along with the time range, were used to generate a SQL query to the `state_vectors_data4` and `flights_data4` schemas in the OpenSky Network Trino database [129]. In the database, the state of an aircraft is a summary of all tracking information (mainly position, velocity, and identity) at a certain point in time. The SQL query included the details provided in Figure 27.

```
SELECT
CAST(from_unixtime(sv."time") AS timestamp) AS time, -- convert
Unix Timestamp and drop time zone
sv.icao24,
sv.lat,
sv.lon,
(sv.velocity * 1.94384) AS velocity, -- convert m/s to knots
sv.heading,
sv.vertrate,
sv.onground,
sv.baroaltitude,
CONCAT(REPLACE(sv.callsign, ' ', ''), '-',
CAST(CAST(from_unixtime(sv."time") AS date) AS varchar)) AS
flight_id,
0 AS distance,
fd.estdepartureairport,
fd.estarrivalairport
FROM minio.osky.state_vectors_data4 sv
LEFT JOIN minio.osky.flights_data4 fd
ON sv.icao24 = fd.icao24
AND sv.callsign = fd.callsign
AND CAST(sv."time" AS bigint) BETWEEN fd."day" AND fd."day" + 86400
-- assuming "day" is start of day
WHERE sv.lat >= 38.2505513514 AND sv.lat <= 39.4518486486
AND sv.lon >= -77.8114709761 AND sv.lon <= -76.2689290239
AND sv."hour" >= 1748761200 AND sv."hour" <= 1756710000
and sv.vertrate < 5
AND fd.estarrivalairport = 'KDCA';
```

Figure 27 SQL Query to OpenSky Network Trino Database

The following considerations were included in the query:

- Dates had to be converted to Unix Timestamps (Table 7) to retrieve the data from the database and then converted back to datetime format in the output file for readability.
- The flight identifier field (`flight_id`) was generated by combining the flight date with a callsign to avoid duplication. The resulting format for `flight_id` was *CALLSIGN-yyyy-MM-dd*.
- The velocity parameter was converted from m/s to knots for convenience.
- The trajectories were selected at a vertical rate less than 5 m/s to exclude ascending (i.e., departing) flight trajectories.
- Estimated departure and arrival airport information was pulled from `flights_data4`. Despite many missing or unreliable records, the query indicated "KDCA" as the arrival airport.

Table 7 Date and Time Conversion to Unix Timestamp

Parameter	Unix Timestamp	Date
Start Hour	1748761200	Sunday June 1, 2025, at 1 AM EDT
End Hour	1756710000	Monday September 1, 2025, at 1 AM EDT

The query produced 8.32 GB of raw data in a .csv format. The snapshot of the file, with corresponding columns, is provided in Figure 98 in Appendix B – Historical Data:

- Date and time
- ICAO 24-bit address of the aircraft's transponder
- Latitude
- Longitude
- Velocity
- Heading
- Vertical rate
- On ground indicator
- Barometric altitude
- Flight ID
- Estimated departure airport
- Estimated arrival airport

The raw data yielded 35,116 flight trajectories. A snapshot of the horizontal profile, altitude, and ground speed for one week of collected data is provided in Figure 28. Data preprocessing steps were required to remove incomplete or invalid flight trajectories and corrupt data records.

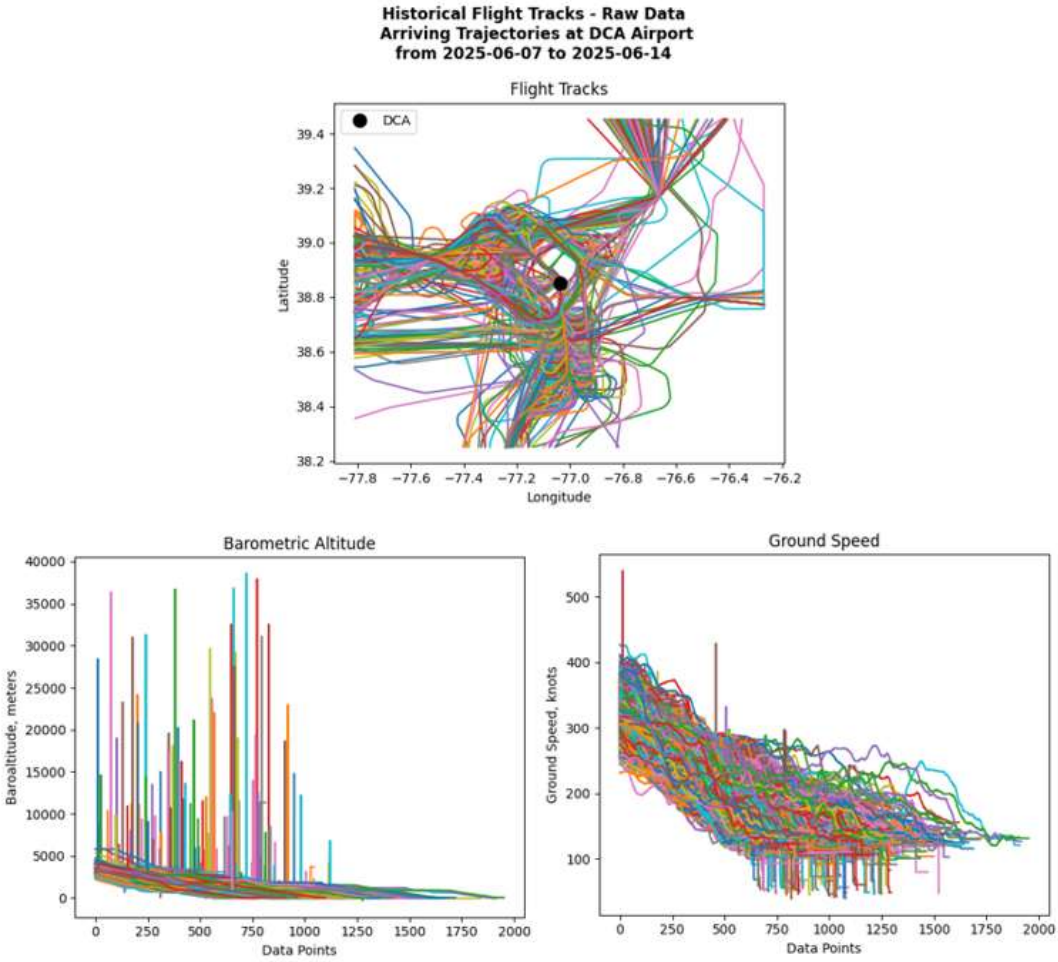


Figure 28 Raw Data for Arrival Flight Trajectories at DCA

5.1.2 Data Preprocessing

After careful examination, the horizontal profile in Figure 28 shows many incomplete flight trajectories flying over the DCA area. These incomplete trajectories needed to be removed from the collected dataset. The vertical profile exhibited high noise, with individual records along the flight path randomly reaching tens of thousands of feet in elevation. The inherent noise in data is common for ADS-B sources, caused by

signal loss, multi-path interference, and hardware limitations [130]. Similar to the altitude data, sudden changes in the ground speed profile were also observed. Since raw data contained significant noise, rigorous filtering and preprocessing were required to produce a homogeneous set of arrival flight trajectories at DCA.

Data filtering conditions for trajectory removal from the dataset are provided in Table 8. Based on the original parameter or criteria in the first column, conditions and rationale for removal are described in the subsequent columns. The threshold values were selected arbitrarily based on the qualitative analysis of the raw data.

Table 8 Filtering Conditions for Trajectory Removal from Collected Data

Parameter	Filtering Condition	Rationale for Trajectory Removal
Latitude and Longitude	The first recorded position of the flight track did not originate at either side of the 36 nm bounding box.	This condition ensured that only full-length flight tracks are included in the historical data for further analysis. The first recorded positions across all flight tracks in the post-filtered dataset are provided in Figure 29. Here, each track originated on the boundary of the 36 nm bounding box.
	The last recorded position of the flight track did not have an "on-ground" indication.	This condition removed flight tracks that have not landed at the airport.
	The last recorded position of the flight track was not within DCA airport coordinates.	This condition helped eliminate arrivals at nearby airports (e.g., BWI, IAD). The last recorded positions across all tracks in the post-filtered dataset are provided in Figure 29. Here, all tracks ended in the vicinity of DCA airport. Further interpolation to the landing runways is required in the later steps.
The Length of Tracks	A flight track had fewer than 500 data records.	This condition removed short tracks, which are equivalent to 8-10 minutes of flight time, from the dataset.
	A track had more than 2,000 data records.	This condition helped eliminate some of the helicopter traffic data records.
Barometric Altitude	The altitude indication at the first position was less than 2,200 meters.	An altitude below 2,200 meters at the bounding box's origin indicated helicopter traffic in the area. Therefore, this condition helped eliminate helicopter traffic data records.
	The altitude indication at the first position was greater than 8,000 meters.	An altitude above 8,000 meters at the origin of the bounding box indicated passing-by flight trajectories in the area that did not intend to land at DCA airport.
	The difference in altitude between the first and 300th positions was less than 100 meters.	This condition helped eliminate flight trajectories that did not exhibit a descending pattern.
	The altitude was greater than 15,000 ft at any point in the track data.	This condition helped eliminate either passing-by flights or trajectories with invalid data records.
	The altitude indication of the last position was greater than 1,000 ft.	This condition helped eliminate passing-by flight trajectories in the area that did not intend to land at DCA airport.

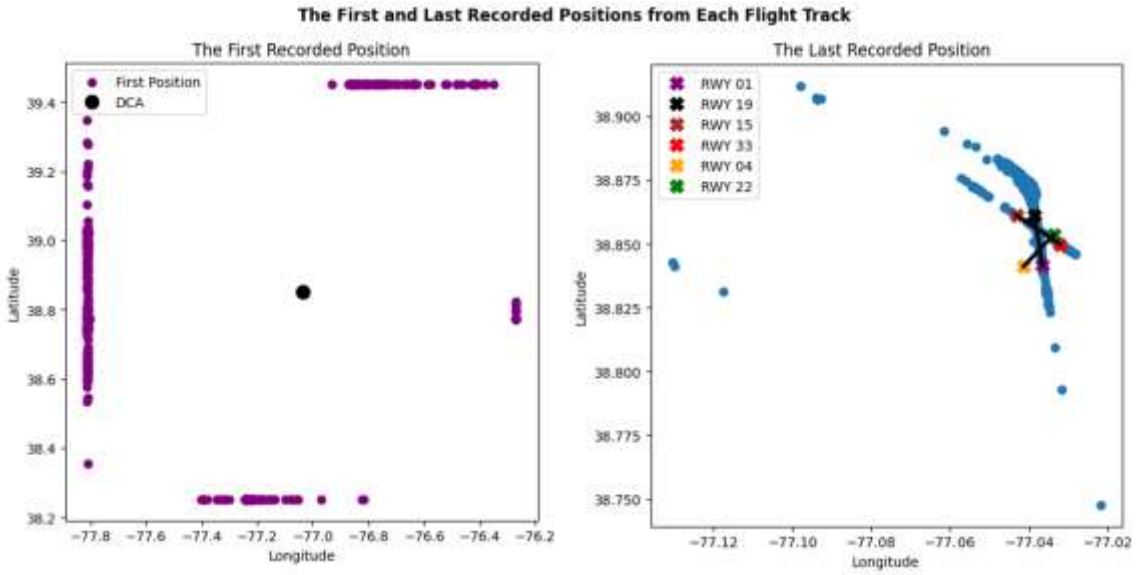


Figure 29 The First and Last Recorded Positions from Each Flight Track

The rationale for removing helicopter traffic from historical data samples was based on the decision first to form a homogeneous dataset of arriving flight tracks. Future work can build on this case study's dataset by including other traffic patterns, such as departure procedures and helicopter trajectories.

The vertical and ground speed profiles contained several unreasonable outliers along the track, such as drastic increases or decreases over a few seconds. Therefore, some of these invalid records can be substituted rather than the entire trajectory data being completely removed from the dataset. Invalid record substitution conditions are provided below for each parameter separately.

Data substitution conditions by ***barometric altitude***:

- Convert barometric altitude units of meters to feet by multiplying the original value by 3.28084. This conversion could be done when the SQL query was generated.
- If the absolute value of the difference in altitude between the consecutive data points was greater than 1,000 ft, then assign the previous altitude to the current data point.
- If there was a missing value in the altitude data, assign the previous altitude to it.
- If the altitude was negative at any point in the track data, set that record's altitude to zero.

Data substitution conditions by ***ground speed***:

- If the absolute value of the difference in ground speed between the consecutive data points was greater than 50 knots, then assign the previous ground speed to the current data point.

An example of one-week pre-processed data is shown in Figure 30. Data filtering and invalid-record substitution achieved a more stable and representative sample of arrival trajectories at DCA airport. However, unreasonable data records from an aerodynamic standpoint should be removed or substituted at later stages.

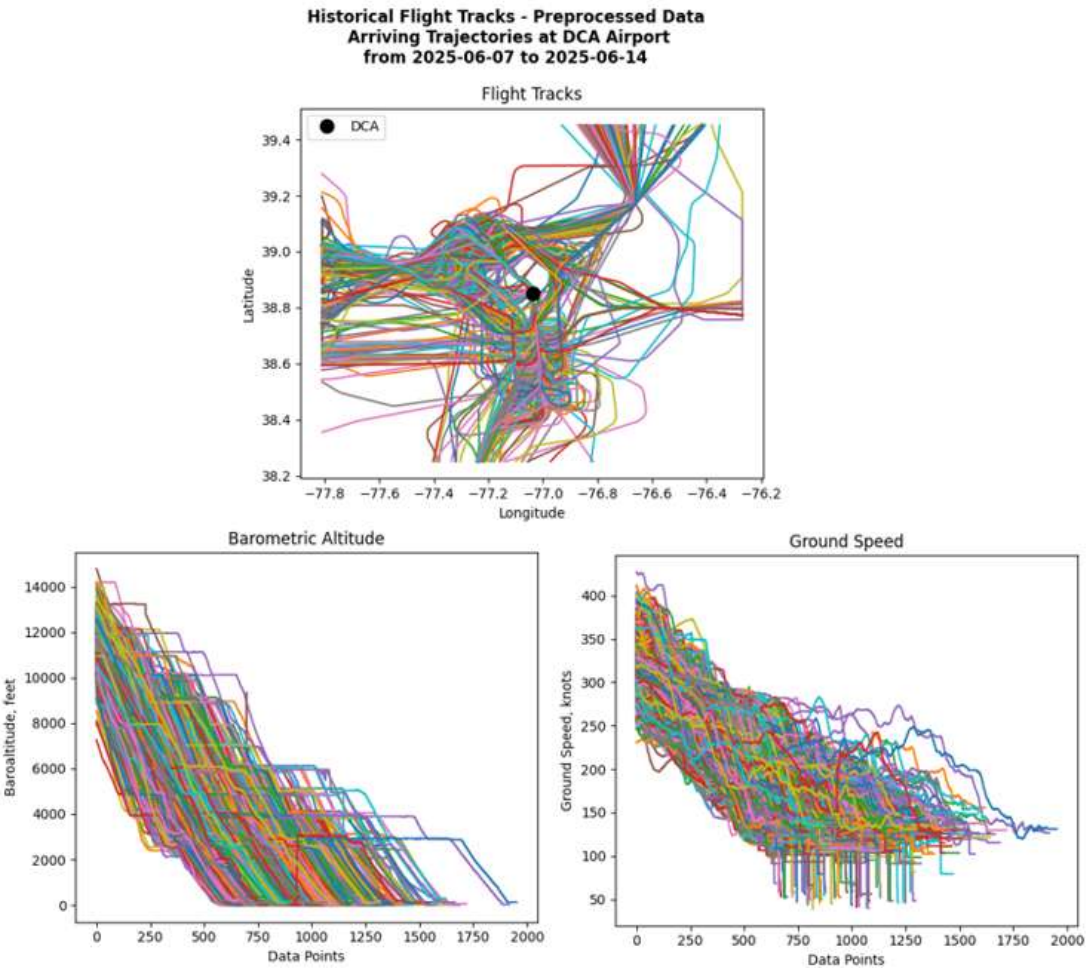


Figure 30 Preprocessed Data for Arrival Flight Trajectories at DCA

5.1.3 Data Standardization

5.1.3.1 Positional Data Interpolation

The length of the flight tracks in the preprocessed data ranged from 500 to 2000 data records per track. Furthermore, the distance between points within the track also varied. An example of the original flight track before standardization is shown in Figure 31. This track contained 1,437 data points with varying distances between them.

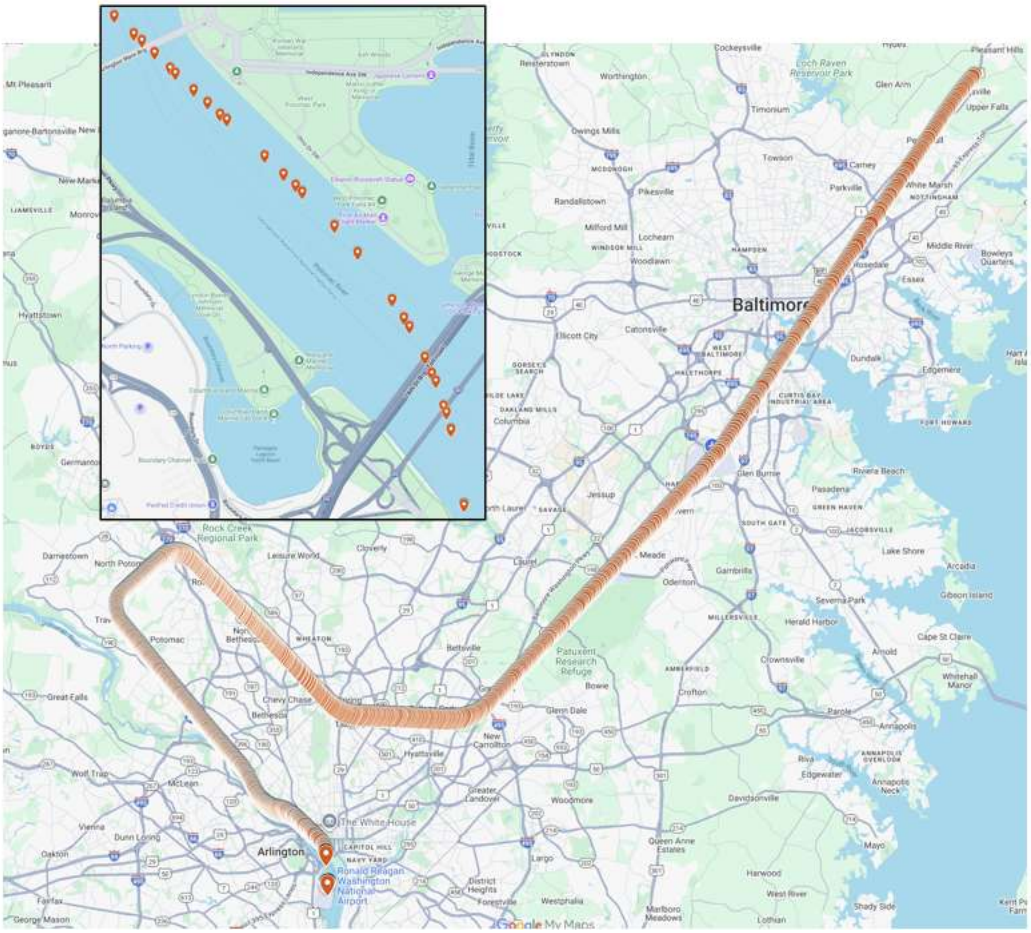


Figure 31 A Flight Track Before Standardization

In accordance with the algorithm described in Section 4.1 and Equation 25, data interpolation was used to standardize the length of all flight tracks. A fixed-length track of 200 records reduced data volume and provided a homogeneous input space for further modeling and analysis. An example of the same track in Figure 31 after standardization is shown in Figure 32. The normalized track contained precisely 200 data points, each spaced the same distance apart. This procedure was repeated for all tracks in the dataset.

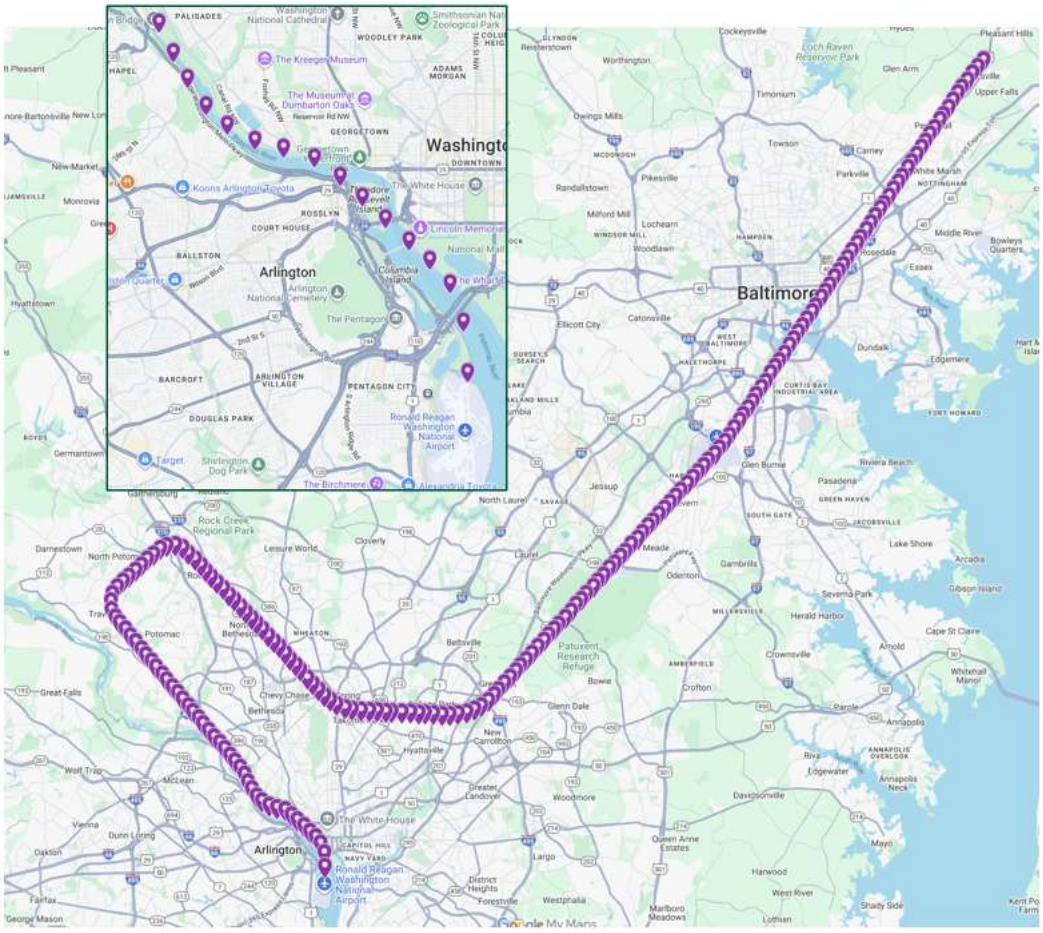


Figure 32 A Normalized Flight Track

Further filtering conditions applied on the standardized data to remove the remaining helicopter traffic and passing-by trajectories at higher altitudes in the area (Table 9). The threshold values were selected arbitrarily based on the qualitative analysis of the preprocessed data.

Table 9 Filtering Conditions for Trajectory Removal from Preprocessed Data

Parameter	Filtering Condition	Rationale for Trajectory Removal
Along Track Distance	The along track distance was greater than 100 nm	These conditions helped eliminate helicopter traffic in the area.
Barometric Altitude	The altitude indication between 100th and 125th positions was less than 2,000 ft.	
	The altitude indication between 125th and 150th positions was less than 1,500 ft.	
	The altitude indication between 150th and 190th positions was greater than 8,000 ft.	This condition helped eliminate passing-by air traffic at higher altitudes.

The preprocessing activities provided 18,802 standardized and filtered flight tracks arriving at DCA airport across three months. The length of each track was normalized to 200 data points. The post-processed flight track data is shown in Figure 33.

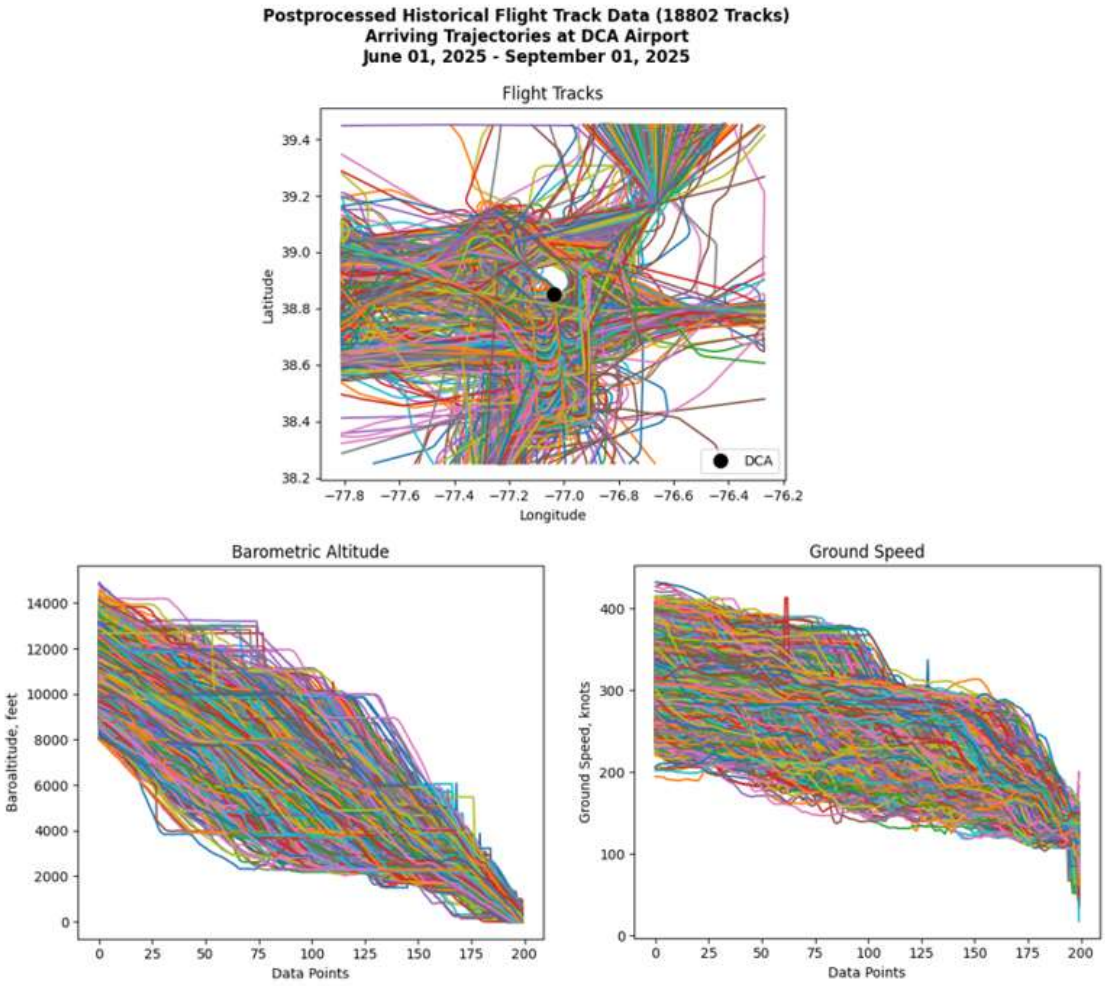


Figure 33 Postprocessed Historical Flight Track Data

5.1.4 Data Clustering

The collected data did not provide runway information. Therefore, a two-step data clustering procedure was required to identify (1) landing runways and (2) navigation procedures (e.g., directions of the air traffic for the selected runway).

5.1.4.1 Runway Identification

An official FAA airport diagram for DCA [131] is shown in Figure 34 to facilitate runway identification. DCA is a constrained urban airport located along the Potomac River and features six intersecting runways:

- Runway 1/19 (7,169 feet)
- Runway 15/33 (5,204 feet)
- Runway 4/22 (5,000 feet)

Runways 1 and 19 are the primary arrival and departure runways and are the longest at the airport, accommodating most commercial air carrier operations. Runway 1 is typically used for aircraft arriving from the south, while Runway 19 is commonly used for arrivals from the north, depending on prevailing wind conditions and air traffic requirements.

Runway 15/33 and Runway 4/22 serve as secondary runways and provide operational flexibility during changing weather conditions, varying wind patterns, runway maintenance activities, or periods of high traffic demand.

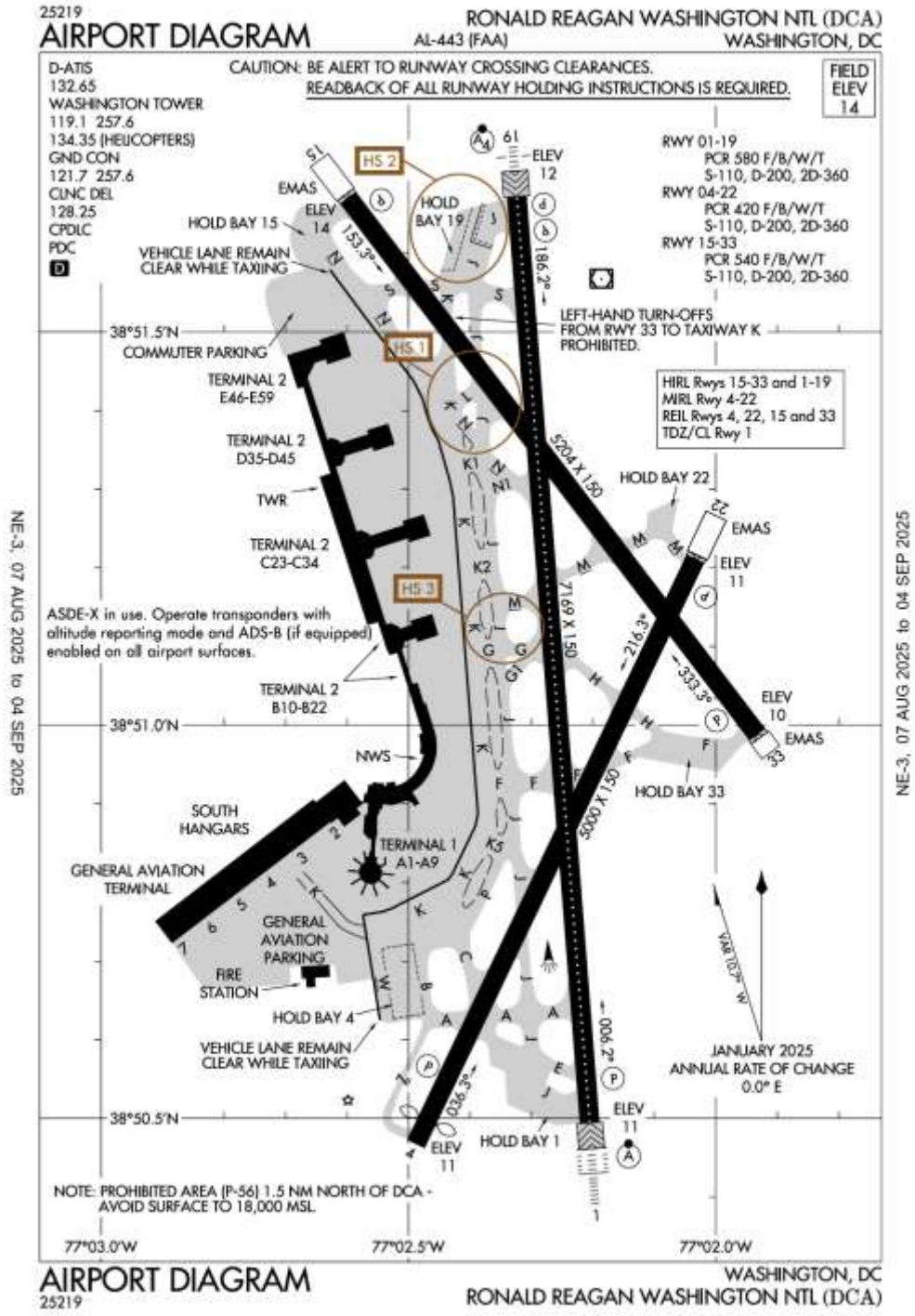


Figure 34 DCA Airport Diagram

The DBSCAN algorithm was used to classify arrivals by landing runway. Clustering was performed using latitude and longitude coordinates at the following positions at the end of each flight track: [190, 192, 194, 196, 198, 199]. The 200th position indicates a touchdown.

DBSCAN was configured with an epsilon (ε) value of 0.01, which defines the radius of the neighborhood surrounding each data point. Selecting a relatively small epsilon creates tighter local neighborhoods and increases the algorithm's responsiveness to subtle spatial variations in the approach paths. The epsilon value enables detection of small differences in latitude and longitude that distinguish trajectories associated with nearby runways.

The minimum number of samples required to form a cluster was set to 5. Lowering this parameter in DBSCAN made the algorithm less strict, allowing sparse regions to form clusters. As a result, fewer trajectories were classified as noise while preserving the ability to detect distinct arrival patterns associated with individual runways.

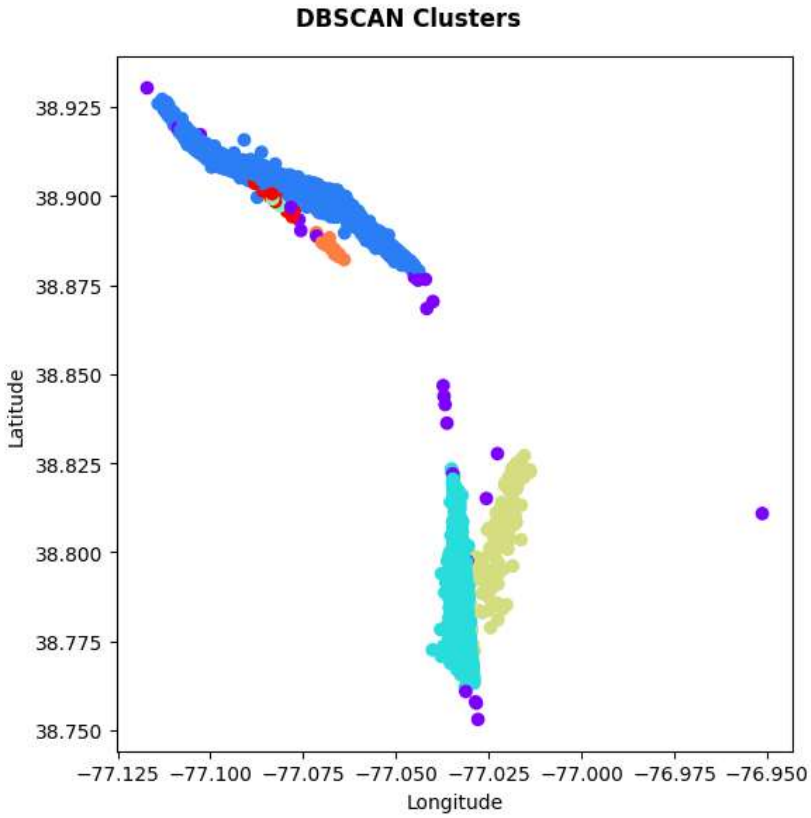


Figure 35 DBSCAN Clustering for Runway Identification

Historical data were captured across six clusters, which correspond to the number of runways at DCA (Figure 35). The resulting clusters are shown in Figure 36. Cluster #0 represented arriving flight trajectories landing on Runway 19. This was the largest cluster with 9,447 landing tracks. Cluster #1 contained 9,129 trajectories landing from the South direction on Runway 1. Clusters #2 and #4 represented only 28 tracks landing on Runway 15, which is not heavily utilized at the airport. Cluster #5 had 139 tracks landing on Runway 33. Only 59 tracks were not classified by the clustering algorithm.

The largest cluster for Runway 19 arrivals was selected for further processing due to its largest sample size of tracks.

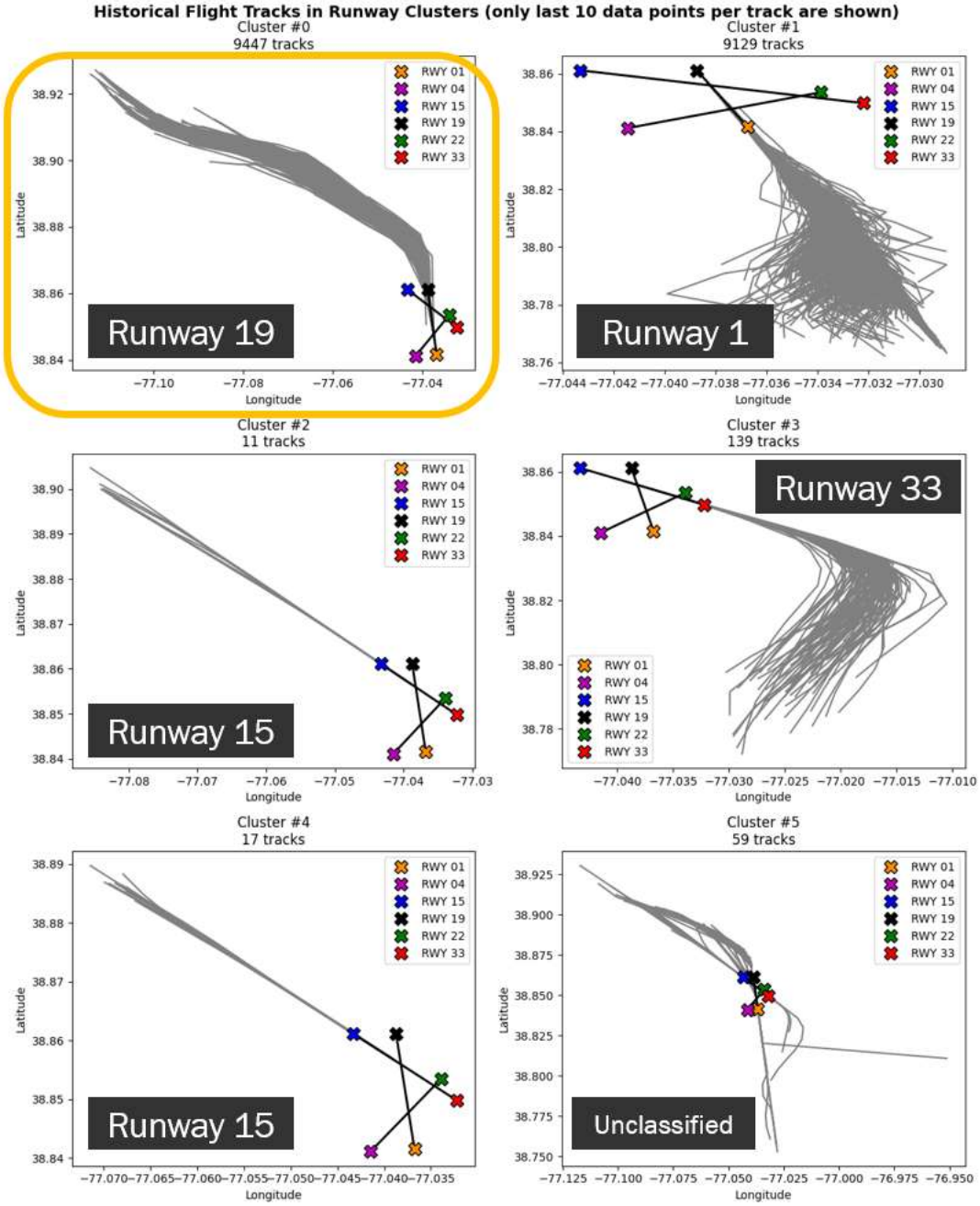


Figure 36 Runway Clusters at DCA

5.1.4.2 Navigation Procedure Identification

Arrivals on Runway 19 are shown in Figure 37. Since flights arrive from every direction on the runway, the second-step clustering is required to classify arrivals.

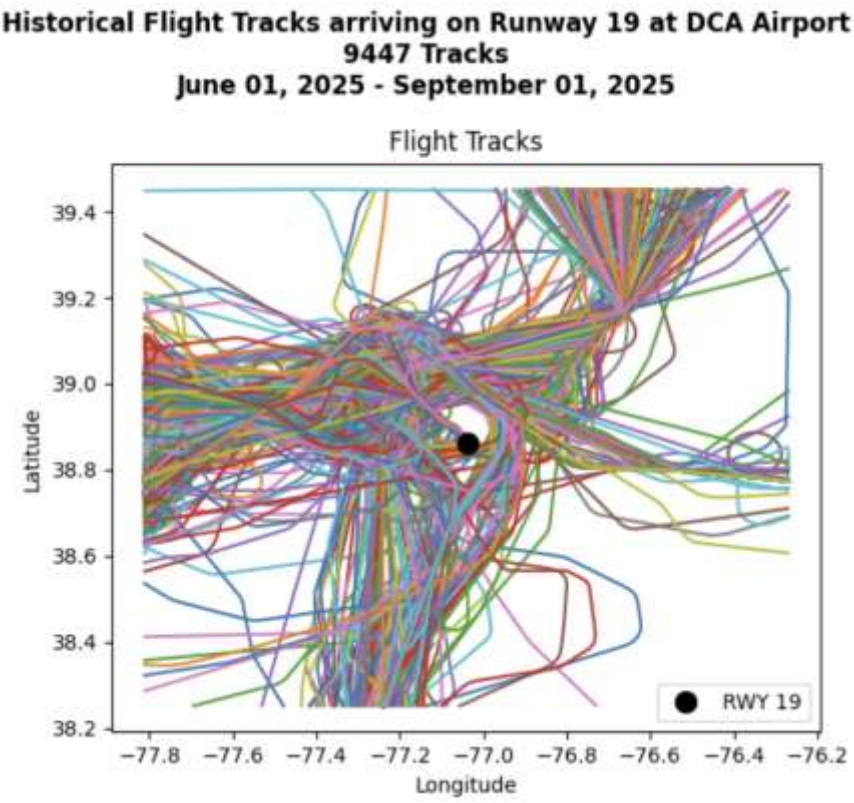


Figure 37 Arrivals on Runway 19 at DCA

The DBSCAN algorithm was applied to latitudes and longitudes at the following positions along the tracks landing on Runway 19: [50, 60, 100, 120] (Figure 38). Epsilon was set to 0.08, while the minimum number of samples was 10. Six resulting clusters for

different navigation procedures are shown in Figure 39, while 229 tracks were unclassified (Figure 40).

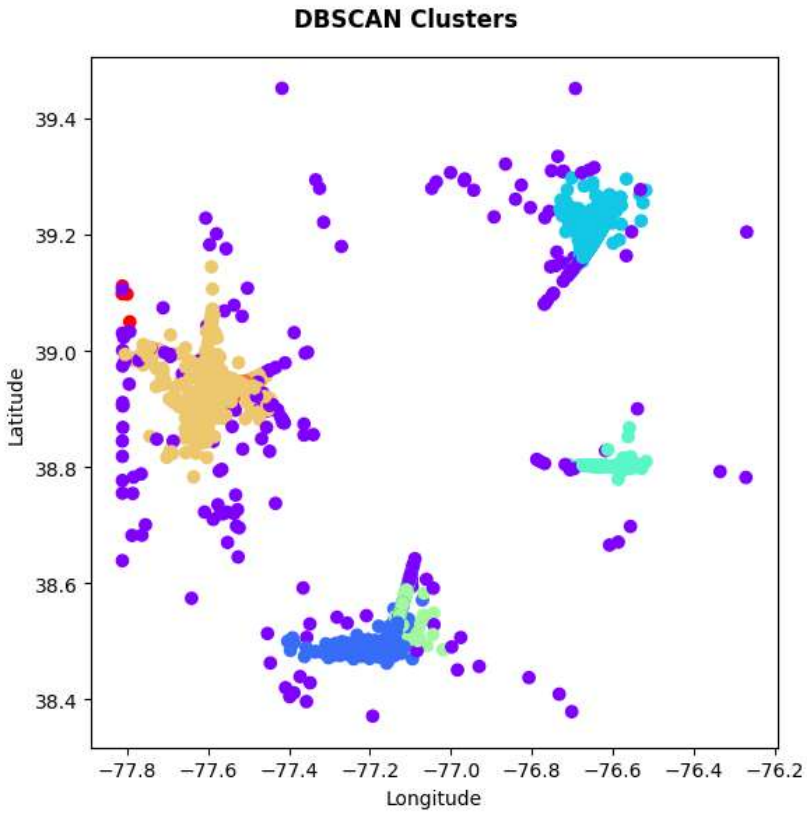


Figure 38 DBSCAN Clustering for Navigation Procedure Identification

Cluster #0 contained 2,600 tracks arriving from the southern direction. Cluster #1 had 1,710 tracks from the northern direction. The largest Cluster #4 showed complex arrival procedures from the West. Cluster #5 recognized holding patterns. The remaining Clusters #2 and #3 contained 189 and 324 distinctive tracks arriving from the East and South, respectively.

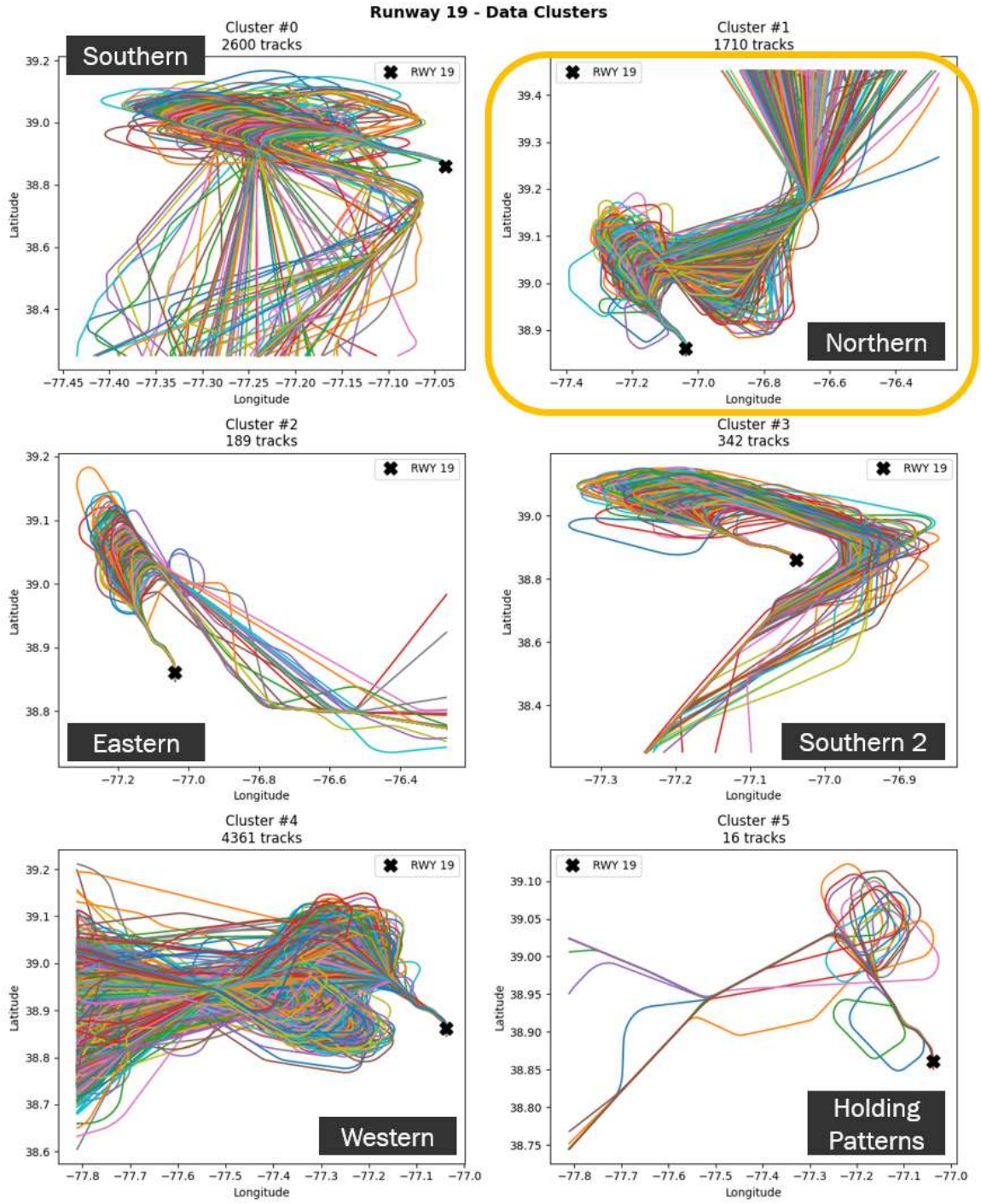


Figure 39 Runway 19 Clusters at DCA

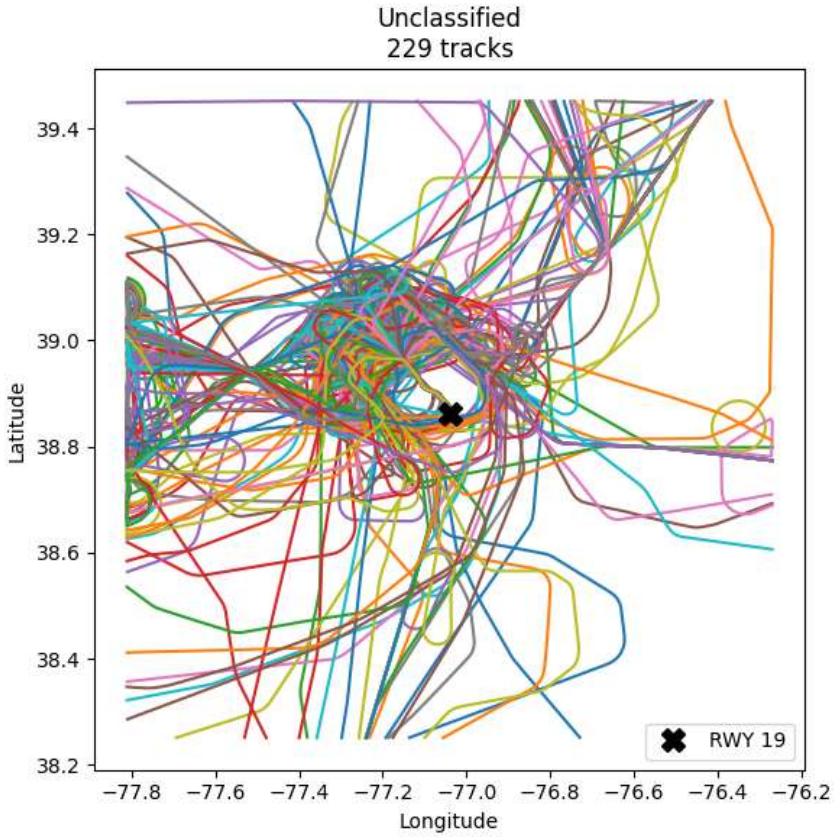


Figure 40 Unclassified Arrivals on Runway 19 at DCA

Cluster #1 with Northern arrivals on Runway 19 at DCA was selected for further analysis. The flight tracks, along with their vertical and speed profiles, across three months at DCA are shown in Figure 41.

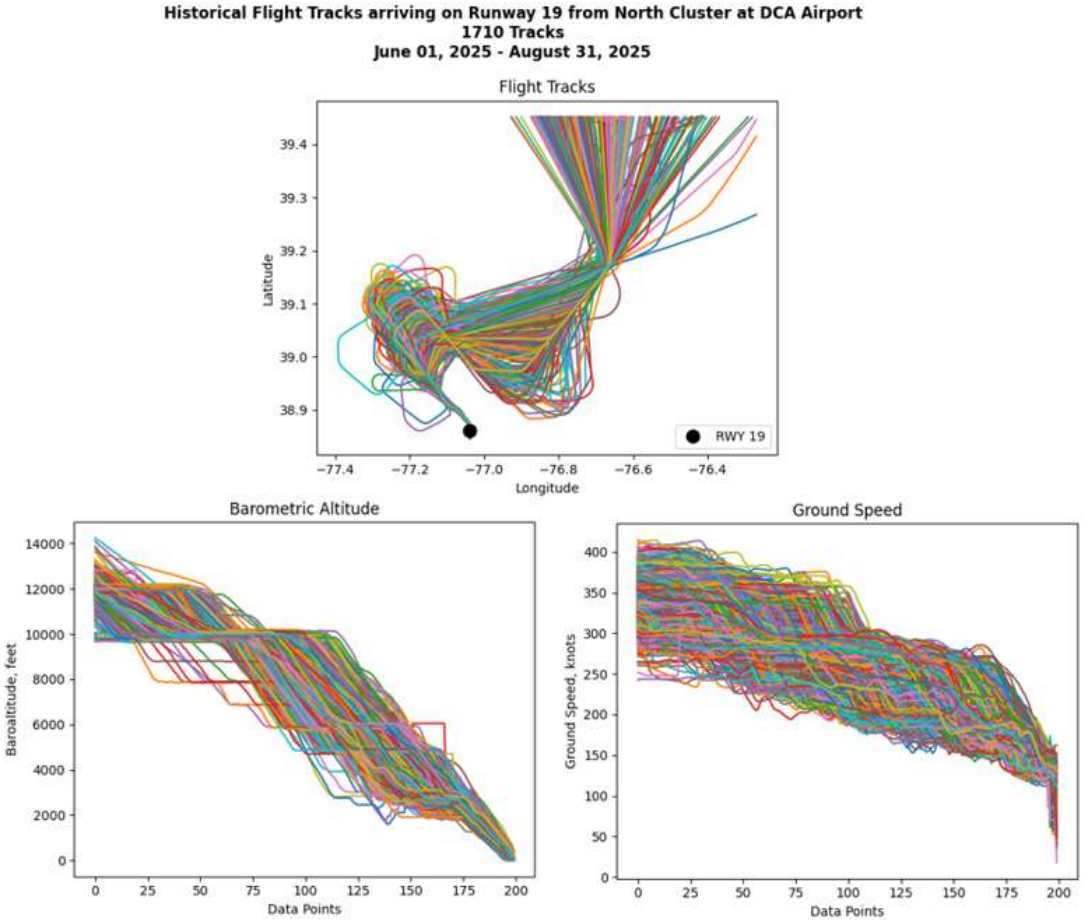


Figure 41 Runway 19 Northern Cluster Data

The final step in processing flight track data was to ensure all selected tracks end at the same location. The last recorded position of the flight tracks is shown in Figure 42. Since the last recorded positions varied across the tracks, the runway threshold was determined to represent a touchdown location (Figure 43).

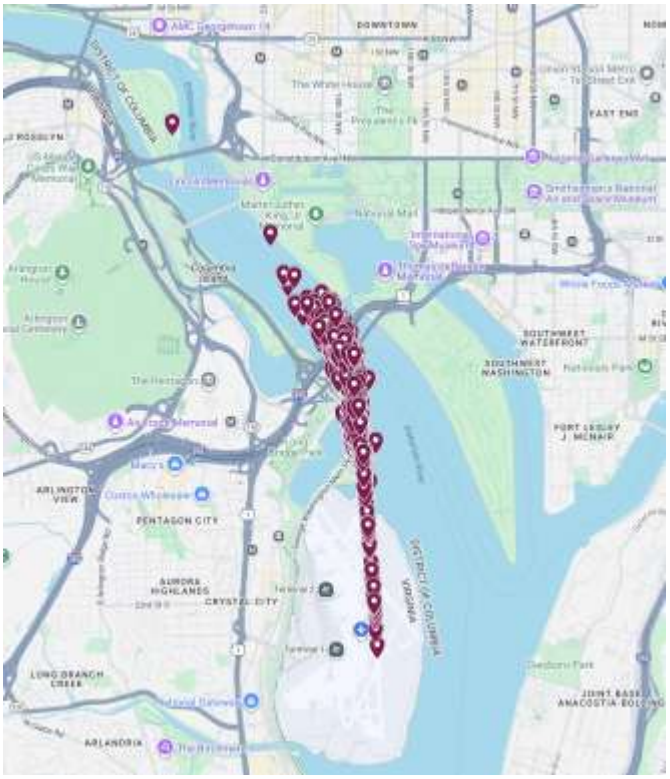


Figure 42 Last Recorded Position Across All Flight Tracks



Figure 43 Runway 19 Threshold

Positional data interpolation was applied to the last data point for all tracks. The resulting flight tracks that stop being recorded precisely at the runway threshold are shown in Figure 44.

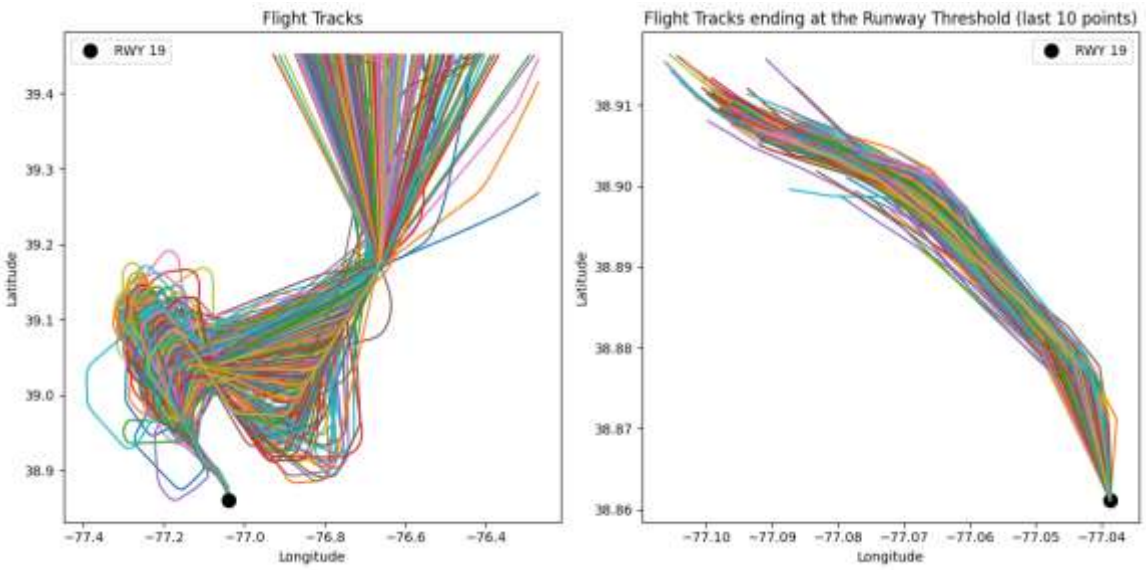


Figure 44 Interpolated Last Recorded Positions across All Flight Tracks

This section concludes data preprocessing and provides post-processed flight track data suitable for further analysis.

5.1.5 Data Analysis

Given the post-processed flight track data, the next steps were to collect measurements for data distributions and derive additional parameters. As described in the Methodology Chapter, these distributions and derived parameters are required in the Validation framework to generate synthetic flight trajectories.

Measurements for lateral, vertical, and speed dispersions were collected at the following locations along the flight tracks:

- At the runway threshold
- 3 nm away from the runway threshold
- 6 nm away from the runway threshold
- 9 nm away from the runway threshold
- At multiple waypoints from published standard procedures

5.1.5.1 Measurements at Runway Threshold

Measurements at the runway threshold, including the along-track distance and flight duration, are provided in the form of data distributions in Figure 45.

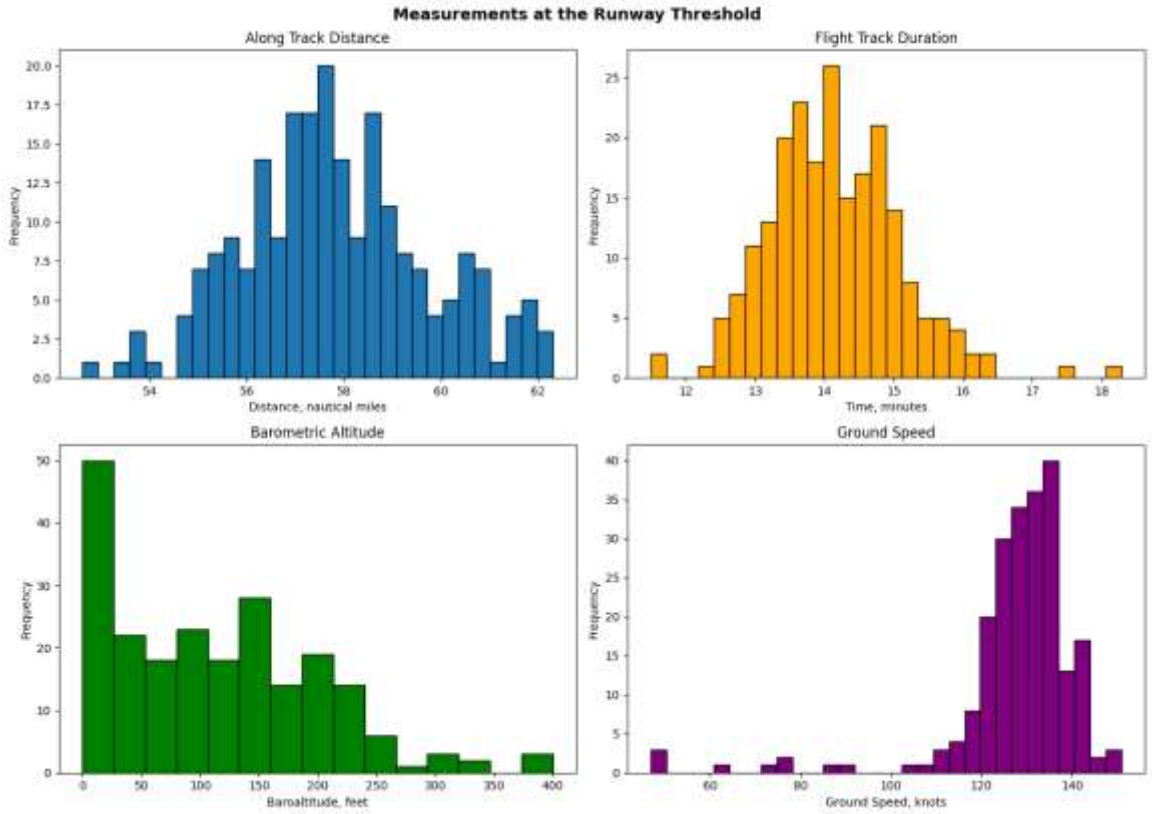


Figure 45 Data Distributions at the Runway Threshold

The along-track distance distribution is roughly unimodal and symmetric, with most observations concentrated between 56 and 60 nautical miles. The relatively narrow spread indicated a high degree of consistency in the location of aircraft relative to the runway threshold. A small number of observations appeared at both tails of the distribution, which implied occasional variations in the approach geometry.

The flight track duration distribution resembles a normal distribution. Most trajectories had durations between 13 and 15 minutes, with a peak around 14 minutes.

The barometric altitude distribution is strongly right-skewed. Most observations occurred at lower altitudes, primarily below 100 ft, while frequencies steadily decreased

with altitude. The distribution contained a long tail extending to approximately 400 ft, indicating that although most aircraft cross the runway threshold at relatively low altitudes.

The ground speed figures were mainly distributed between approximately 120 and 140 knots, with a pronounced peak near 130 to 135 knots. Overall, the narrow concentration around the mode suggests that aircraft maintain steady approach speeds at the runway threshold.

5.1.5.2 Measurements at 3, 6, and 9 NM from Runway Threshold

Since the variability of flight tracks increases with distance from the runway threshold, measurements for 3 (blue), 6 (purple), and 9 (brown) nm from the runway threshold were recorded at the corresponding radii drawn around the runway threshold (Figure 46). The reference points were arbitrarily selected for each radius. Data distributions for the lateral (blue), vertical (purple), and ground speed (green) parameters are shown in Figure 47.

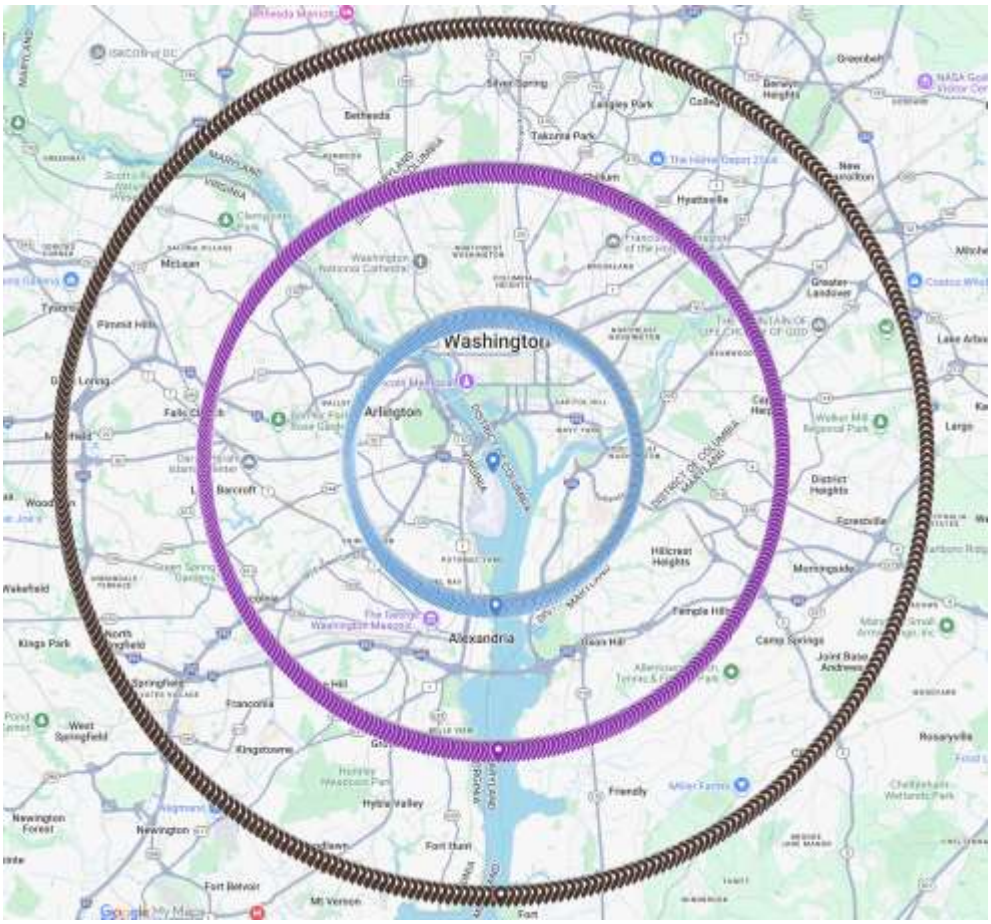


Figure 46 3, 6, and 9 nm Radius around the Runway Threshold

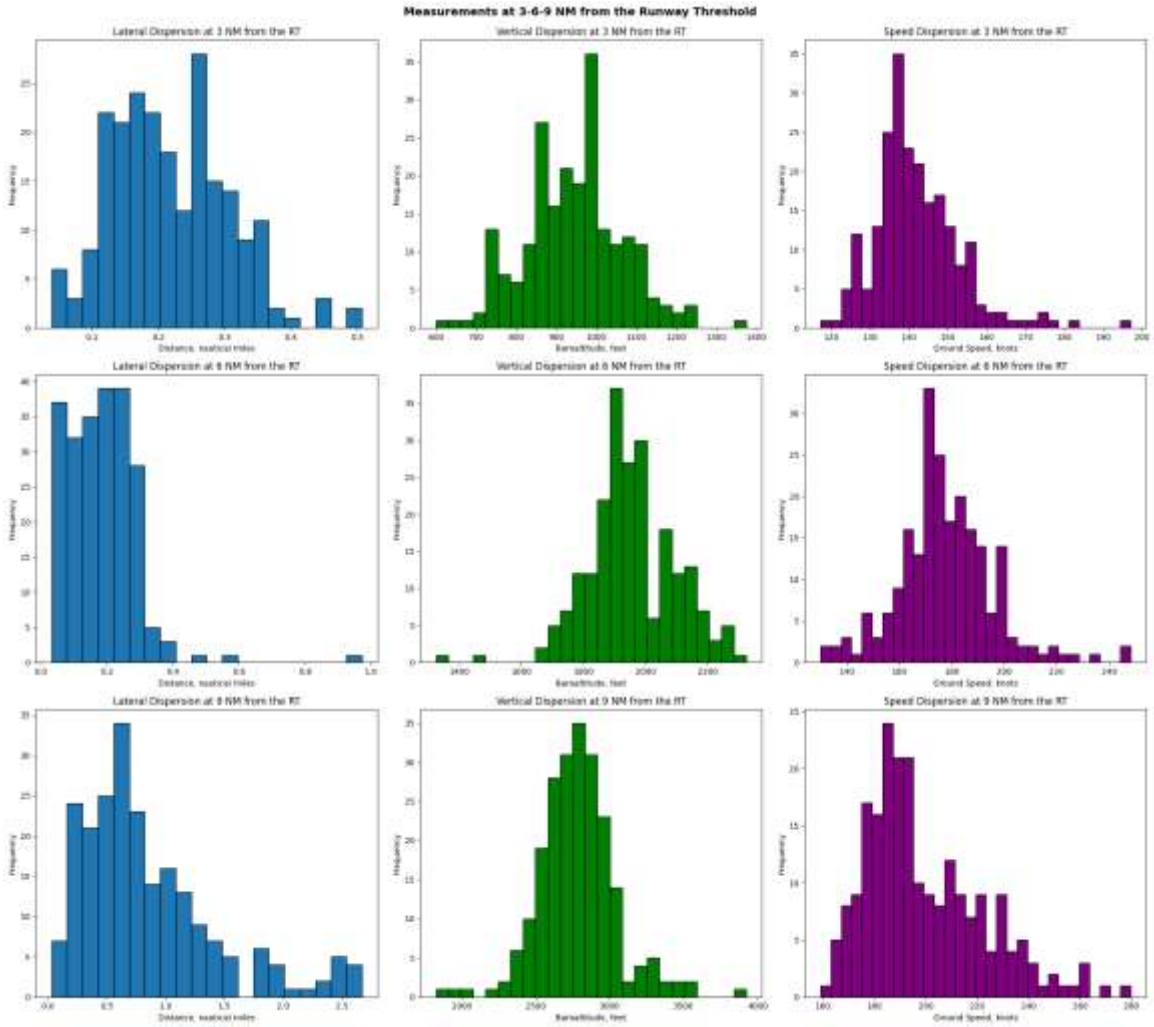


Figure 47 Measurements at 3, 6, and 9 NM from the Runway Threshold

At 3 nm from the runway threshold, lateral dispersion was relatively small, with most flight tracks located within 0.1 to 0.3 nm of the reference approach path. Vertical dispersion exhibited a roughly unimodal distribution centered near 900 to 1,000 ft, while ground speeds were concentrated between 130 and 150 knots, with a peak near 140 knots.

At 6 nm from the runway threshold, the distributions remained largely unimodal but showed greater spread than those observed at 3 nm. Specifically, vertical dispersion

shifted upward, with most tracks between 1,800 and 2,100 ft. Ground speed also increased, with the distribution centered near 175 to 185 knots and displaying moderate variability.

At 9 nm from the runway threshold, the largest variability was observed across all parameters. The histograms demonstrate the narrowing of flight tracks position and speed distributions as aircraft approach the runway threshold.

5.1.5.3 Measurements at Waypoints

Aircraft adhered to published standard arrival and approach procedures. Specifically, the collected flight trajectories closely follow the routing defined by the SKILS FIVE Area Navigation (RNAV) Standard Terminal Arrival Route (STAR), shown in Figure 48, and the RNAV (GPS) Y Instrument Approach Procedure (IAP), shown in Figure 49 [131]. Applicable waypoints are highlighted in yellow. The detailed waypoint data were summarized in Table 25 and Table 26 for both procedures, respectively, in Appendix C – Waypoint Data. These procedures outline the lateral and vertical guidance used by arriving aircraft transitioning from the en-route flight phase to the final approach segment.

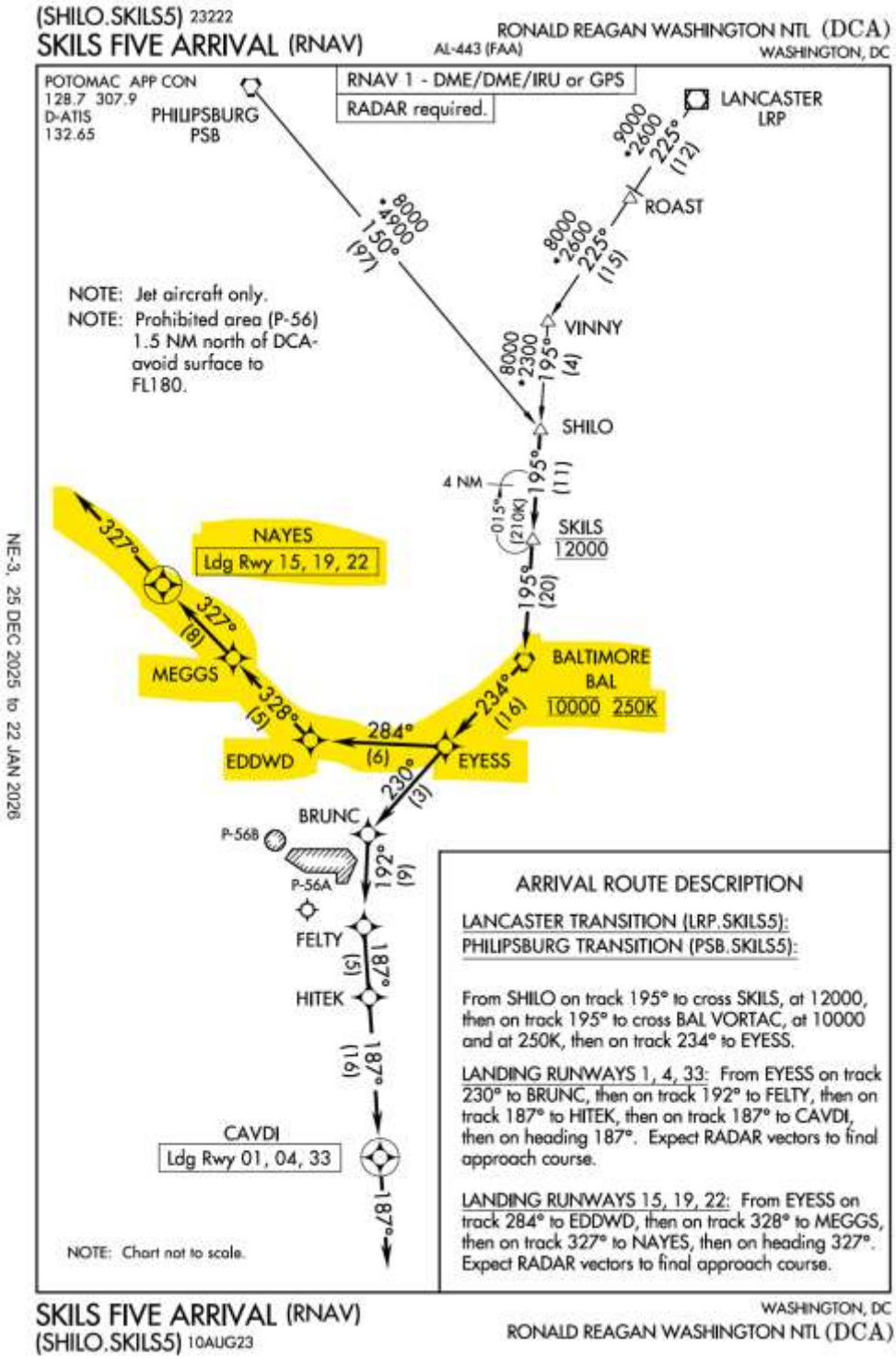


Figure 48 Standard Terminal Arrivals: SKILS FIVE (RNAV)

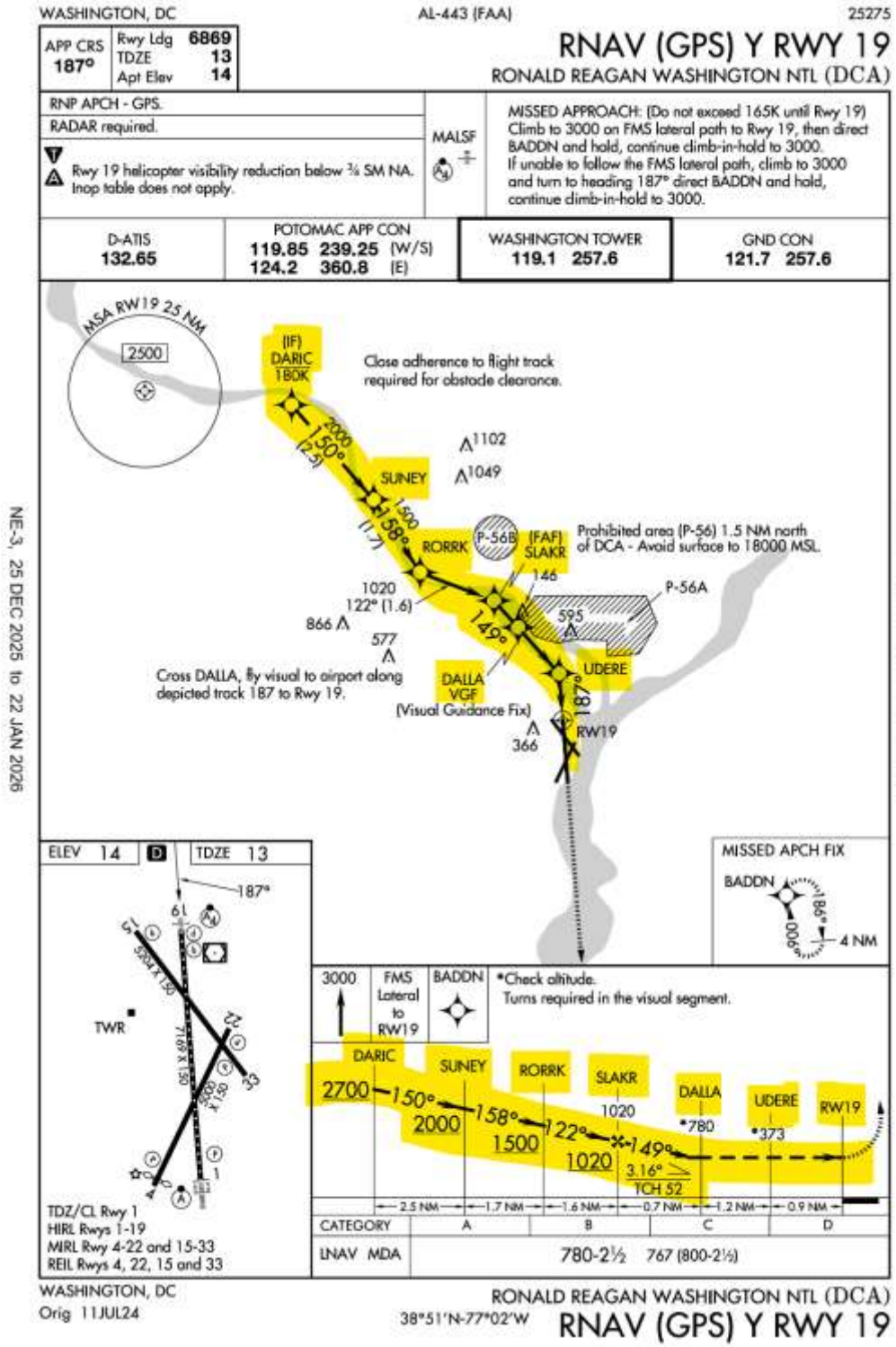


Figure 49 RNAV (GPS) RWY 19

Historical flight trajectories with allocated waypoints from BAL to NAYES in SKILS FIVE, and DARIC onwards in the IAPs are shown in Figure 50. The alignment between the observed flight trajectories and the published procedures indicated a high level of procedural compliance and demonstrates that the collected dataset represents typical arrival operations at DCA.

Data distributions for lateral, vertical, and speed dispersions at waypoints are provided in Appendix C – Waypoint Data .

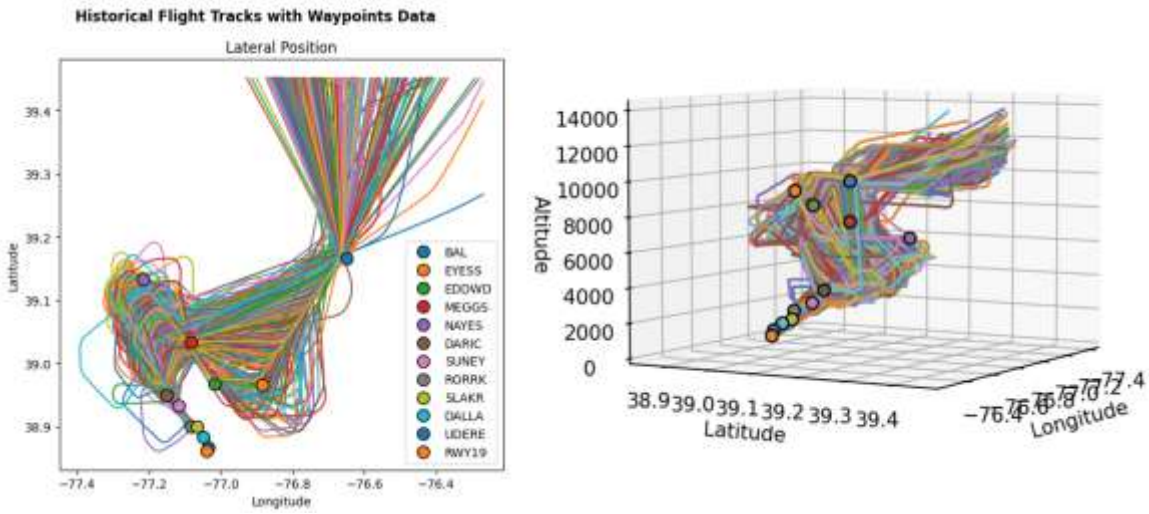


Figure 50 Historical Flight Tracks with Allocated Waypoints

5.1.5.4 Derived Data Parameters

In accordance with Table 4, individual flight track characteristics, such as the number of courses, levels, and speed changes, were derived from flight track data. An

indication of a course change greater than 5° was derived from the heading parameter, while levels greater than 25 ft and speed changes greater than 2 knots were determined from altitude and speed profiles, respectively.

The number of course changes and their associated magnitudes are shown in Figure 51. On average, flight tracks exhibited 3 course changes, each with a magnitude between 5° and 50°. The level- and speed-change segments are shown in Figure 52. There was an average of 7 level changes in the vertical profile of the track data. The number of speed changes varied from 5 to 15 across the tracks.

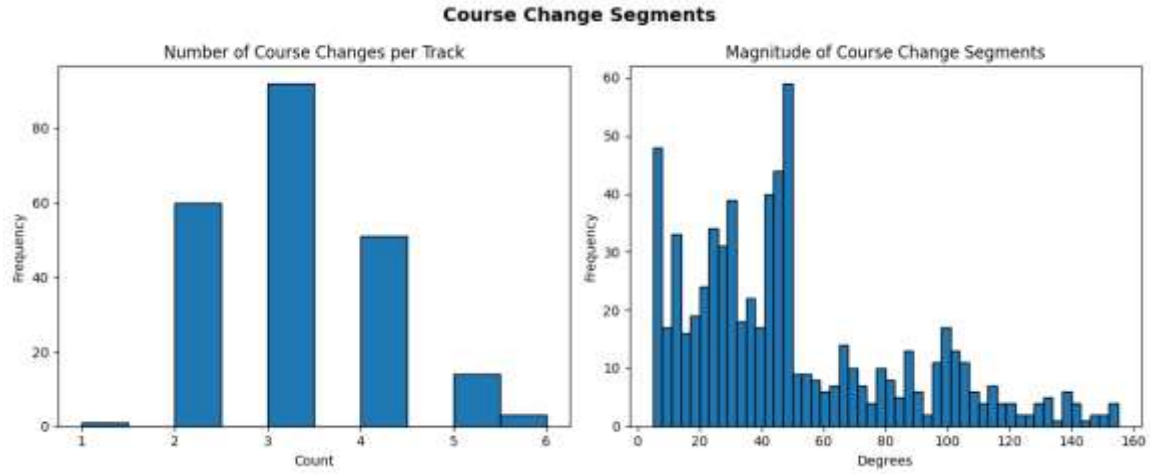


Figure 51 Course Change Segments

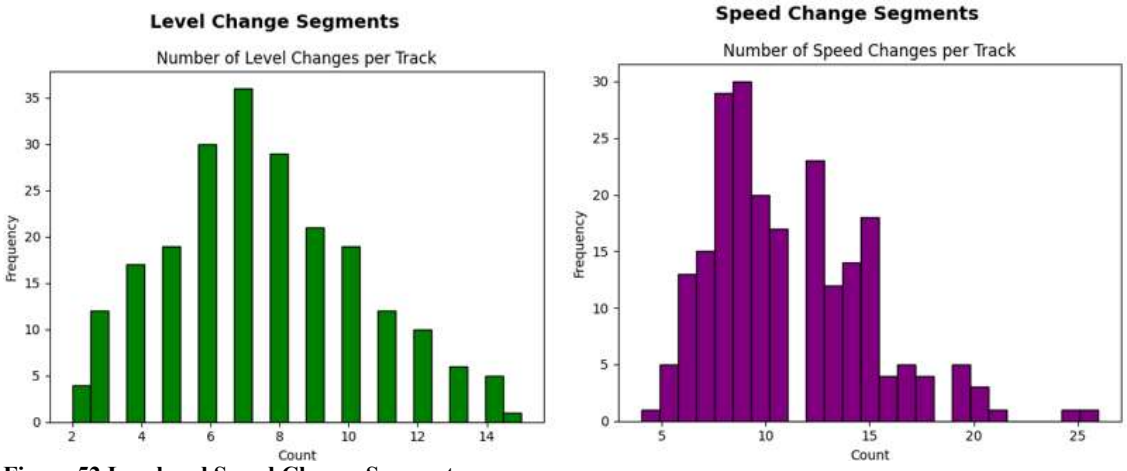


Figure 52 Level and Speed Change Segments

5.1.6 Summary

In summary, after following all steps outlined in Chapter 4.1, historical post-processed data yielded 1,710 flight tracks arriving on Runway 19 from the northern direction at DCA during the three-month period.

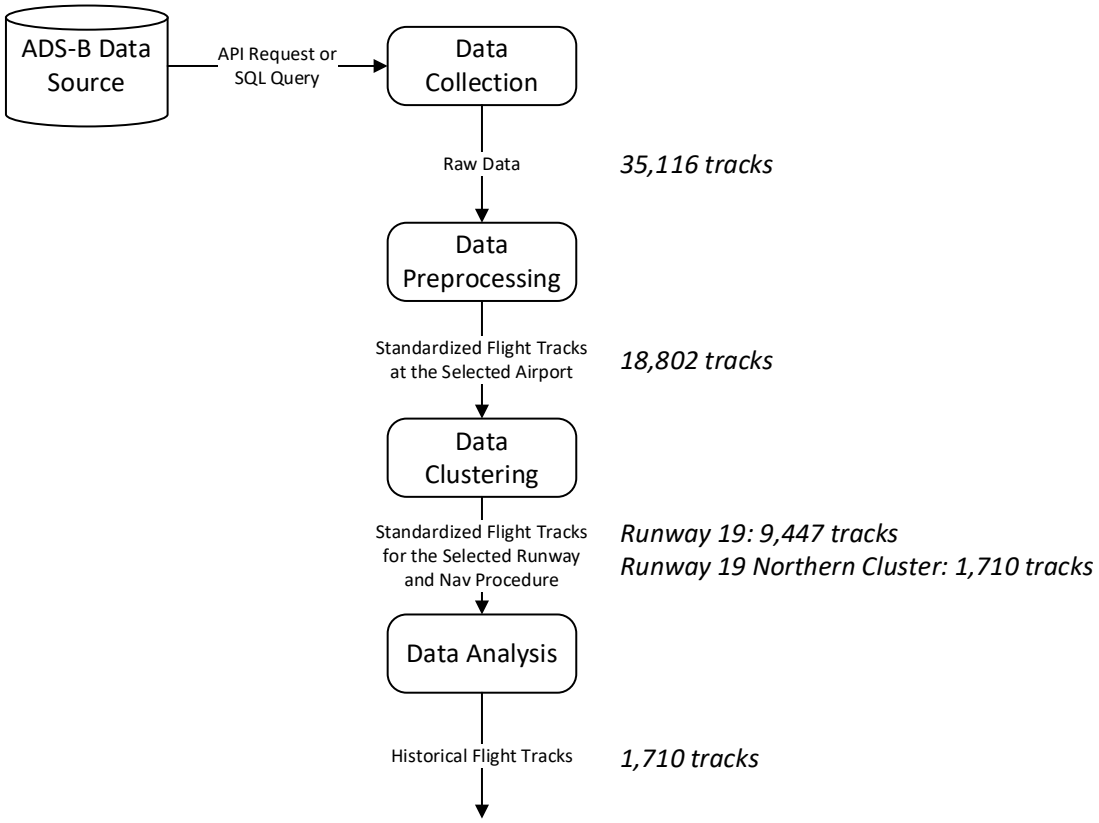


Figure 53 Summary for Historical Flight Track Data

5.2 Synthetic Flight Tracks

This section describes the implementation of the following GP models for generating synthetic flight tracks, with the models numbered based on a response variable:

- GPR with a single response predicting changes in flight track parameters (i.e., latitude, longitude, altitude, and speed), resulting in four independent models, 1A through 1D
- GPR with a single response predicting the next n^{th} position of track data on an example of lateral and longitudinal parameters, resulting in two independent models, 2A and 2B
- GPR with a combined encoded multi-response, Model 3
- Deep GP with a combined encoded multi-response, Model 4, as an extension to Model 3
- GPR with principal components in a multi-response, Model 5

Python scripts for GP models are provided in Appendix F – Python Scripts for GP Models.

In accordance with the modular modeling approach described in Section 4.2, all GP models were constructed in the Learning framework. Then, the synthetic results were generated and evaluated for realism in the Generating and Validation frameworks, respectively.

In the Learning framework, the GP models differed in the predictor allocation and response generation, as well as in the specific GP implementation and associated kernel

function design. However, all models relied on the same training data and seed tracks for testing, facilitating fair comparison of the resulting tracks.

The historical data provide training samples for flight trajectories arriving on Runway 19 from the northern direction at DCA airport (Figure 41). The seed tracks represent a new arrival procedure at DCA. The new procedure can be obtained, for instance, from flight trajectories flown by pilots in a simulator. Although simulated arrival procedures were not available at the time of this research, a different approach was used to obtain the seed tracks.

Flight Trajectories that closely follow the SKILS FIVE RNAV published procedure for the Northern cluster of Runway 19 (Figure 48) were selected as training samples (i.e., observations). Available training flight tracks (931 tracks in total), along with their associated barometric altitude and ground speed profiles, are shown in gray in Figure 54. Due to the GP algorithm's data-efficiency property, not all training samples needed to be used for model learning. By limiting the number of observations, smaller input dimensions facilitate faster, more efficient use of computational resources. Unless otherwise noted, the training data comprised 50 randomly selected flight tracks, with each track standardized to 200 data points, as shown in Figure 54. The input space is characterized by dimensions $50 \times 200 \times p$, where p is the number of predictors.

The tracks that deviated from the arrival procedure were designated as seed samples (shown in dark blue in Figure 54) to represent a new development in the NAS for this case study. Six flight tracks were selected from the seed samples, resulting in 6×200-dimensional outputs for each response variable (Figure 55). The Validation

framework used the remaining 215 seed tracks to compare against generated samples and assess the realism of the synthetic data.

The Generating framework provides a means to output synthetic flight tracks in an interpretable format consistent with the description provided in Section 4.2.2. The best-performing synthetic results for only one of the six seed samples were selected for validation. The Validation framework was used only when (1) the generated models met verification metrics and (2) the preliminary analysis determined the suitability of the synthetic flight tracks for further evaluation.

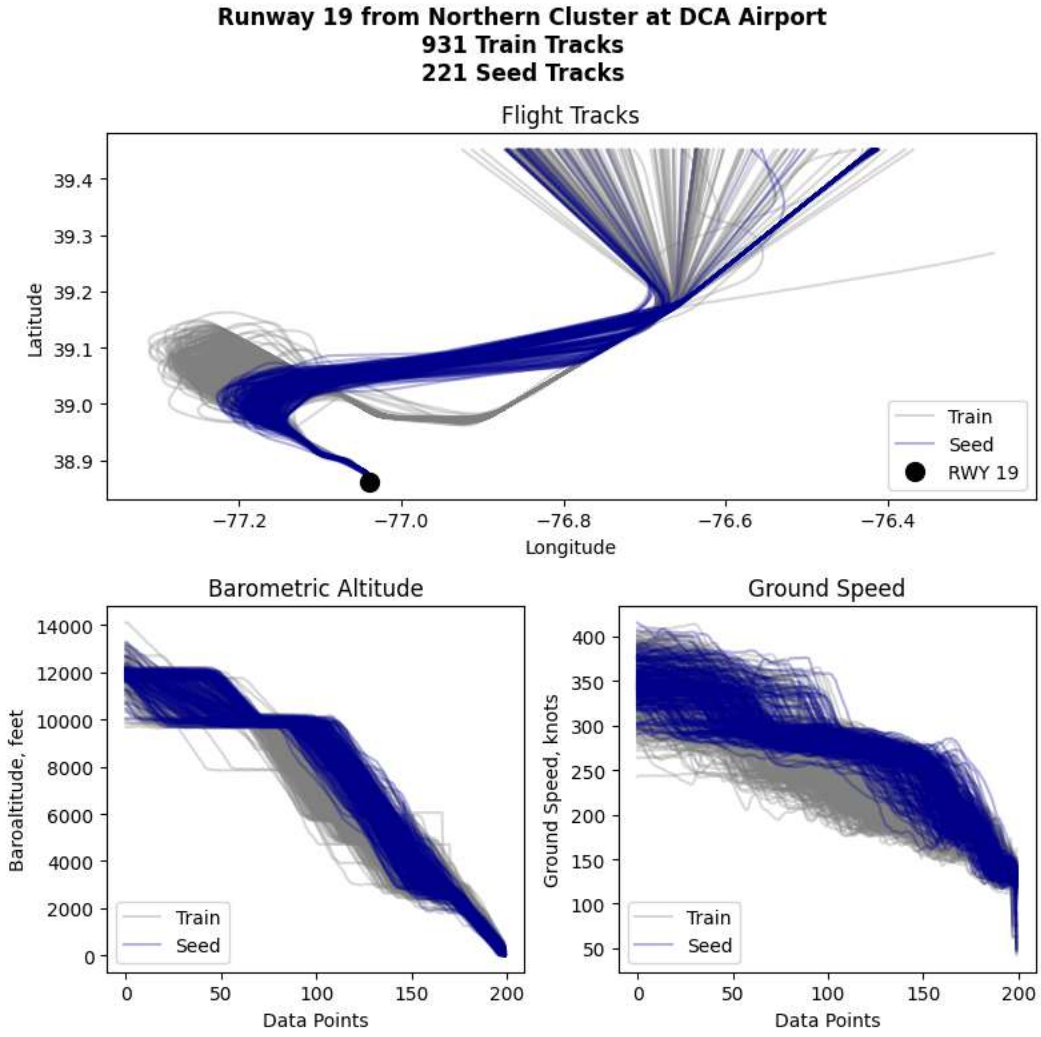


Figure 54 Train and Seed Tracks

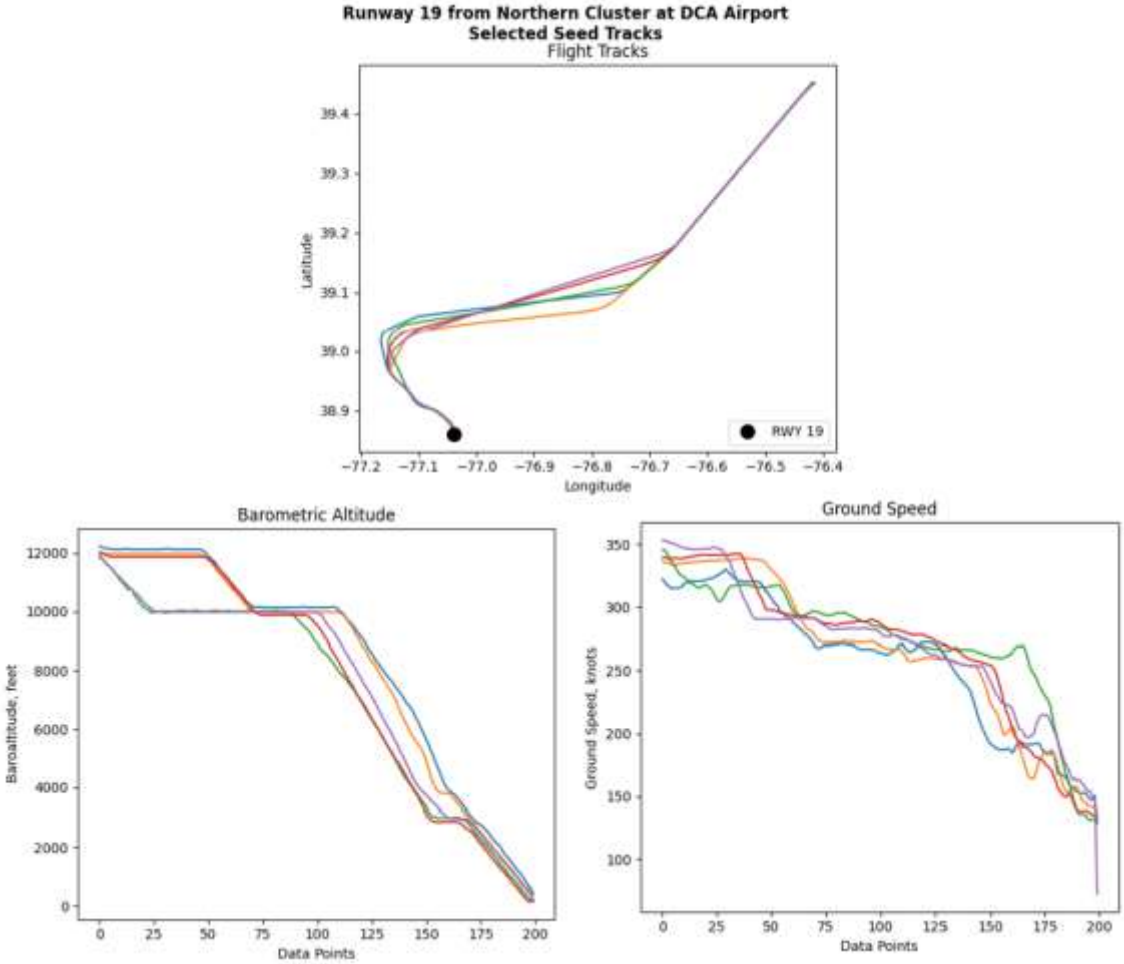


Figure 55 Selected Seed Tracks

5.2.1 GPR Models with Single Response

This section provides synthetic flight tracks generated using GPR with a single response variable. Since track data require latitude, longitude, altitude, and velocity parameters at a minimum, four independent GPR models were constructed for each synthetic track output.

5.2.1.1 Learning Framework

In the Learning framework of the modular modeling approach, the first step was to allocate a set of predictors with a target response variable for each model. The predictor set was comprised of selected flight-track parameters. Prior to applying the Learning framework, the historical flight track data were standardized (see Section 5.1.3 Data Standardization), so each parameter of the track in the dataset contained exactly 200 data points (i.e., positions). Each model had multiple predictors, resulting in an input data dimension of $X \in \mathbb{R}^{n \times 200 \times p}$, where X is the input space, $n = 50$ is the number of flight tracks used for model training, and p is the number of predictors. Consequently, the response in each model represents a corresponding output of $y \in \mathbb{R}^{n \times 200}$ dimensions. The models, along with their associated predictors and responses, are provided in Table 10.

Table 10 Independent GPR Models with Single Response

GPR Model	Predictors	Response
Model 1A	$X \in \mathbb{R}^{50 \times 200 \times 6}$: 1. Position 2. Latitude 3. Longitude 4. Heading 5. Ground Speed 6. Barometric Altitude	Latitude
Model 1B	$X \in \mathbb{R}^{50 \times 200 \times 6}$: 1. Position 2. Latitude 3. Longitude 4. Heading 5. Ground Speed 6. Barometric Altitude	Longitude
Model 1C	$X \in \mathbb{R}^{50 \times 200 \times 5}$: 1. Position 2. Ground Speed 3. Barometric Altitude 4. Descend Indication 5. Cumulative Descend	Barometric Altitude
Model 1D	$X \in \mathbb{R}^{50 \times 200 \times 5}$: 1. Position 2. Ground Speed 3. Deceleration Indication 4. Magnitude of Deceleration 5. Barometric Altitude	Ground Speed

Evidently, both predictors and responses contained the same parameters across all models. Therefore, the response data needed to be restructured to enable the model prediction capability. The response restructuring method across all models is summarized in Equation 26 and illustrated as an example based on Model 1A in Table 11.

Equation 26 Response Restructuring

$$y_0 = \begin{bmatrix} \Delta lat_0 \\ \Delta lon_0 \\ \Delta alt_0 \\ \Delta speed_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$y_t = \begin{bmatrix} \Delta lat_t \\ \Delta lon_t \\ \Delta alt_t \\ \Delta speed_t \end{bmatrix} = \begin{bmatrix} lat_t - lat_{t-1} \\ lon_t - lon_{t-1} \\ alt_t - alt_{t-1} \\ speed_t - speed_{t-1} \end{bmatrix},$$

where t is a position along the track.

Table 11 Response Transformation in Model 1A

Track #1, 2, 3, …, n; n = number of tracks in the model						
Predictors						**Response #1**
Position	**Latitude**	**Longitude**	**Heading**	**Ground Speed**	**Barometric Altitude**	**Δ Latitude**
0	lat_0	lon_0	h_0	gs_0	$alt0_0$	$\Delta\ lat_0 = 0$
1	lat_1	lon_1	h_1	gs_1	alt_1	$\Delta\ lat_1 = lat_1 - lat_0$
2	lat_2	lon_2	h_2	gs_2	alt_2	$\Delta\ lat_2 = lat_2 - lat_1$
…						
199	lat_{199}	lon_{199}	h_{199}	gs_{199}	alt_{199}	$\Delta\ lat_{199} = lat_{199} - lat_{198}$

Each response was restructured to represent a difference in consecutive values of the associated output required to move along the flight path. The initial response at the first position was set to zero. The predictors at the current position of the flight track were tied to the response, which represents the difference in values between the current and

next positions, and so on. The restructuring sequence was repeated for each track in the model dataset.

Before observing the training samples, prior distributions represented by covariance functions were determined. Primarily, the Matérn kernel was selected to summarize assumptions about the behavior of the underlying responses, expecting roughness and irregular patterns in Δ responses. After observing the training data, hyperparameters in kernel functions were optimized through log marginal likelihood maximization (Table 12).

A small signal variance σ^2 across all models was consistent with the amplitude of Δ scale in the parameters rather than their absolute values. The smoothness v of 0.5 in Models 1A and 1B indicated abrupt changes and strong nonlinear patterns in Δ latitude and Δ longitude profiles. In contrast, the smoothness v of 1.5 demonstrated that Δ altitude and Δ ground speed in Models 1C and 1D, respectively, exhibit only moderate roughness, consistent with the presence of level and constant-speed segments in the flight trajectories. In addition, larger length scale values in Models 1A and 1B captured a gradual change in Δ latitude and Δ longitude. Conversely, length scale values of 0.317 and 0.326 indicate that Δ altitude and Δ ground speed varied rapidly over short distances, signifying overall descent and deceleration patterns.

A large positive log marginal likelihood generally indicates that the model explains the observed data well. However, the GPR model for the speed profile yielded a negative log marginal likelihood value. The negative value was the largest value obtained after 10 model training iterations. The negative log marginal likelihood does not always

imply poor model performance because data scale, noise levels, and model complexity influence the metric.

This concludes the activities performed in the Learning framework for independent GPR models with a single response.

Table 12 Kernel Functions and Log Marginal Likelihood in Model 1

GPR Model	Kernel Function	Log Marginal Likelihood
Model 1A	Matérn Kernel with White Noise: $k(x,x') = 0.00776^2 \cdot k_{Matern}(x,x';\ell = 56,\ v = 0.5) + \sigma_n^2\delta(x,x');\ \sigma_n^2 = 5.52e-07$	22200.925433
Model 1B	Matérn Kernel with White Noise: $k(x,x') = 0.00955^2 \cdot k_{Matern}(x,x';\ell = 135,\ v = 0.5) + \sigma_n^2\delta(x,x');\ \sigma_n^2 = 4.68e-07$	22217.093604
Model 1C	Matérn Kernel with White Noise: $k(x,x') = 1.01^2 \cdot k_{Matern}(x,x';\ell = 0.317,\ v = 1.5) + \sigma_n^2\delta(x,x');\ \sigma_n^2 = 0.192$	2966.779678
Model 1D	Composite Kernel with White Noise: $k(x,x') = 1.71^2 \cdot k_{Matern}(x,x';\ell = 0.326,\ v = 1.5) + k_{RBF}(x,x';\alpha = 2.01,\ \ \ \ \ell = 1.6) + \sigma_n^2\delta(x,x')\ ;\ \sigma_n^2 = 0.23$	-2958.811057

5.2.1.2 Generating Framework

The trained GPR models that predicted changes in latitude, longitude, altitude, and speed profiles are shown in Figure 56 through Figure 59, respectively. Each model provided the posterior and predictive distributions, along with uncertainty estimates, for

the selected seed samples in Figure 55 and their corresponding response parameters. The uncertainty estimates were derived from 95% confidence intervals for each prediction point along the flight track. In the plots, the yellow dots represent a collection of training samples. The black lines show actual seed parameters from historical data. The orange lines show the prediction mean derived from the predictive distribution, with associated confidence intervals for uncertainty estimates shown in light blue.

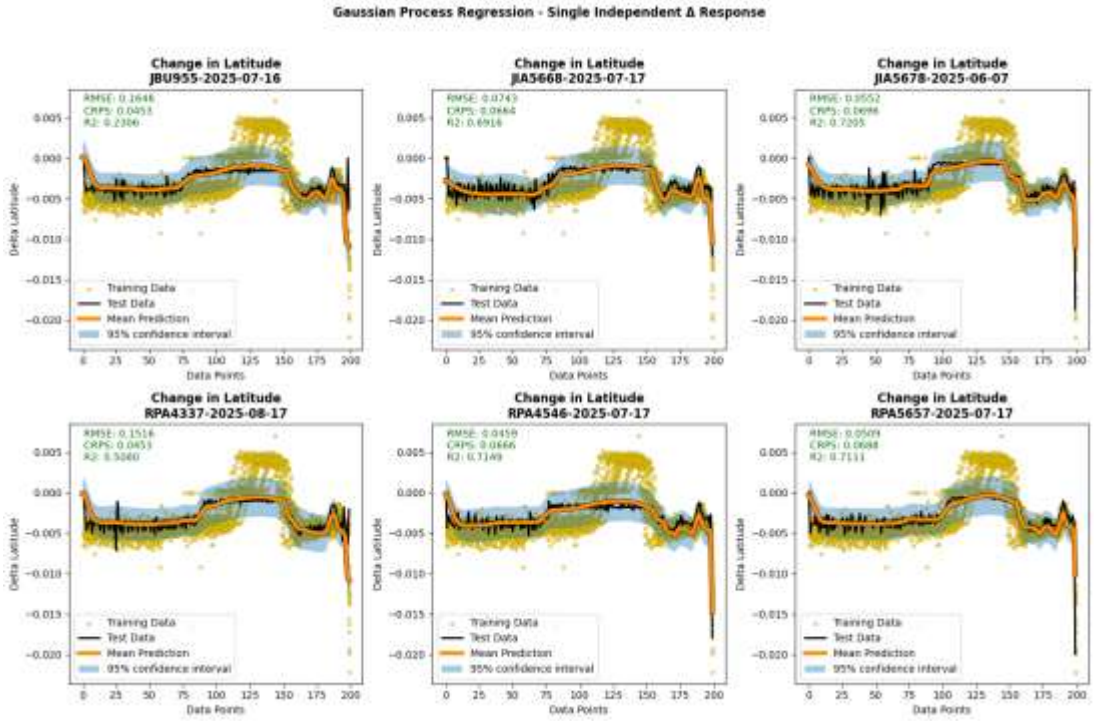


Figure 56 GPR for Change in Latitude in Model 1A

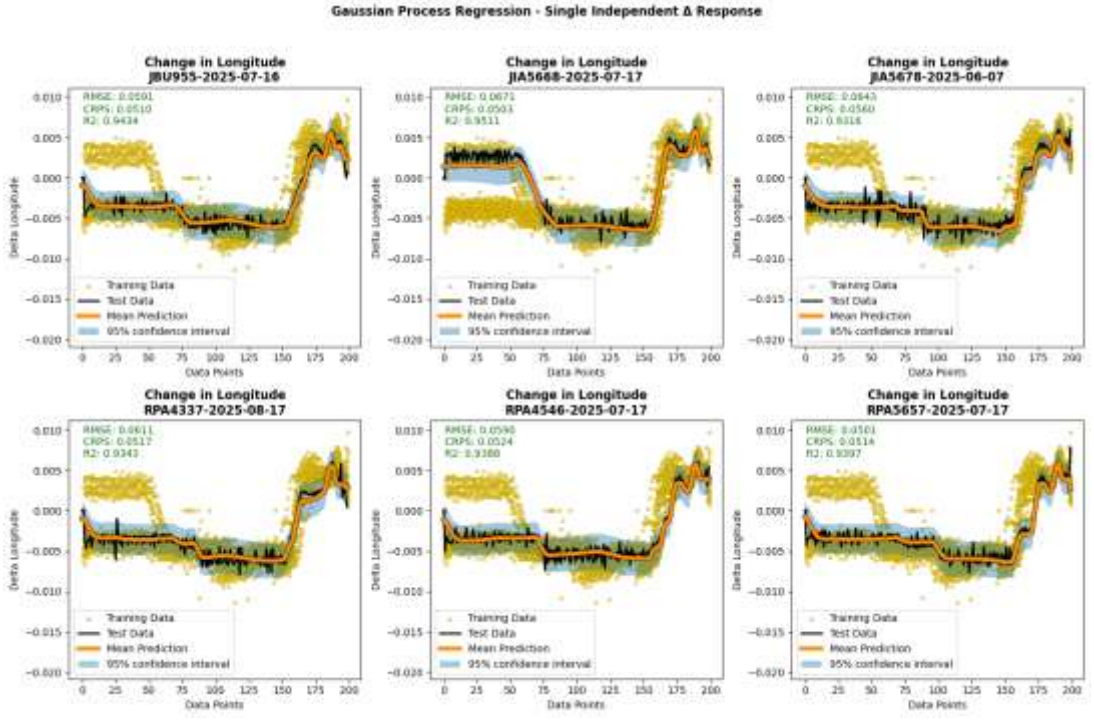


Figure 57 GPR for Change in Longitude in Model 1B

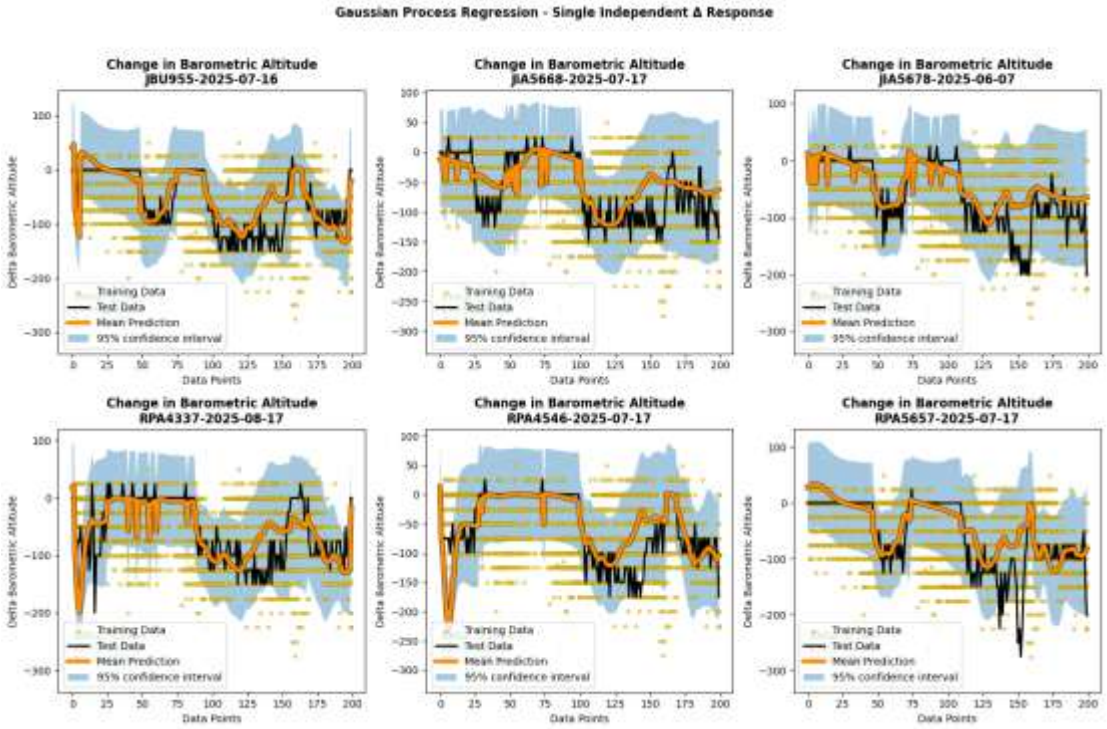


Figure 58 GPR for Change in Barometric Altitude in Model 1C

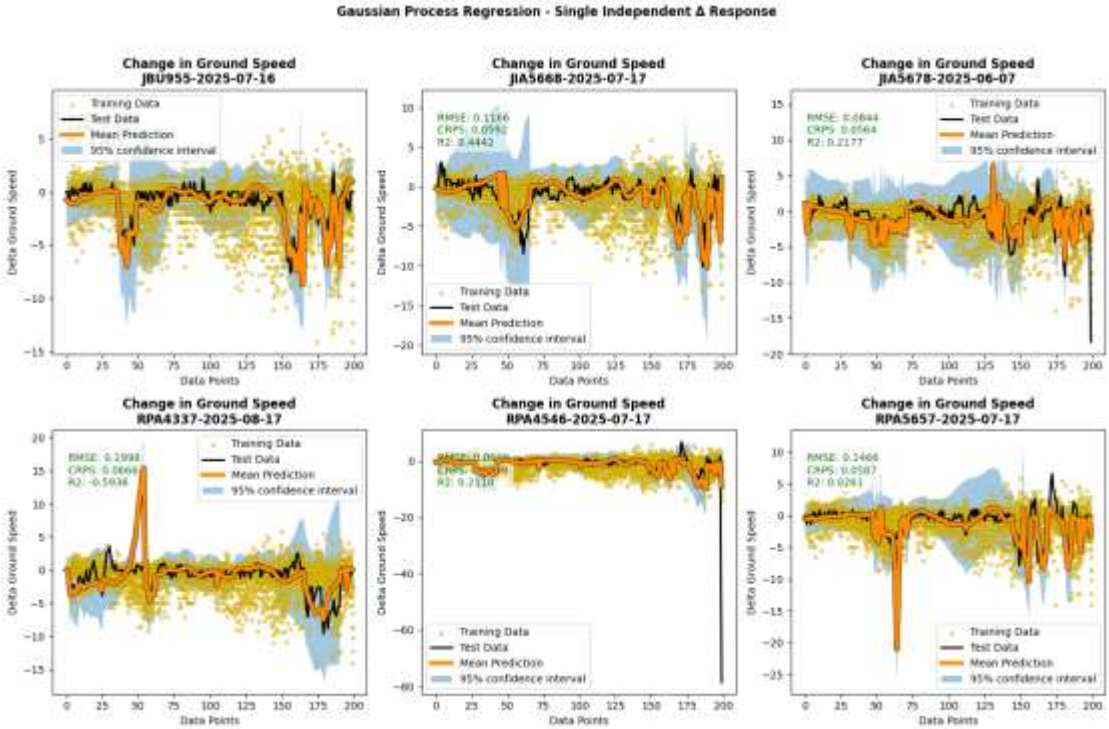


Figure 59 GPR for Change in Ground Speed in Model 1D

Since the results encompassed changes in the parameter values and are not easily interpretable, the response data needed to be reconstructed to the original geographic units of coordinates, feet, and knots. The steps for the response reconstruction were as follows:

1. For each model, set the fixed reference by using values for the corresponding parameter (i.e., latitude, longitude, barometric altitude, and ground speed) at the runway threshold.
2. Assign the reference to the last position of each track.
3. Iterate backward through track data by adding or subtracting the Δ values at the current position to the derived value in the previous position.
4. Repeat for all tracks in the model.

An example of Δ latitude reconstruction is provided in Table 13. A backward iteration through the track data was intentionally selected to ensure that the reconstructed tracks end precisely at the runway threshold. In contrast, a forward iteration led to cumulative error propagation in the models, resulting in predictions for latitude and longitude that missed the runway threshold coordinates.

Table 13 Response Reconstruction in Model 1A

	Response #1	Reconstructed Response #1
Position	**Δ Latitude**	**Latitude**
1	$\Delta\ lat_1$	$lat_1 = lat_2 + \Delta\ lat_1$
2	$\Delta\ lat_2$	$lat_2 = lat_3 + \Delta\ lat_2$
3	$\Delta\ lat_3$	$lat_3 = lat_4 + \Delta\ lat_3$
…		
198	$\Delta\ lat_{198}$	$lat_{198} = lat_{199} + \Delta\ lat_{198}$
199	$\Delta\ lat_{199}$	$lat_{199} = lat_{200} + \Delta\ lat_{199}$
200	$\Delta\ lat_{200}$	$lat_{200} = lat_{rt}$

The reconstructed latitude and longitude in absolute coordinates for the seed tracks are shown in Figure 60 and Figure 61, respectively. For context, the predicted flight track coordinates, with independently generated latitude and longitude parameters plotted together, are shown in Figure 62. The reconstructed vertical and speed profiles, generated by separate models, are shown in Figure 63 and Figure 64. In the plots, a collection of gray lines represents samples drawn from the corresponding posterior distribution. The posterior samples represent synthetic flight data parameters for each model. Thus, the data present one million synthetic parameters, generated independently, for each seed track, resulting in $1{,}000{,}000 \times 6 = 6{,}000{,}000$ new synthetic flight tracks (only 10,000 tracks shown).

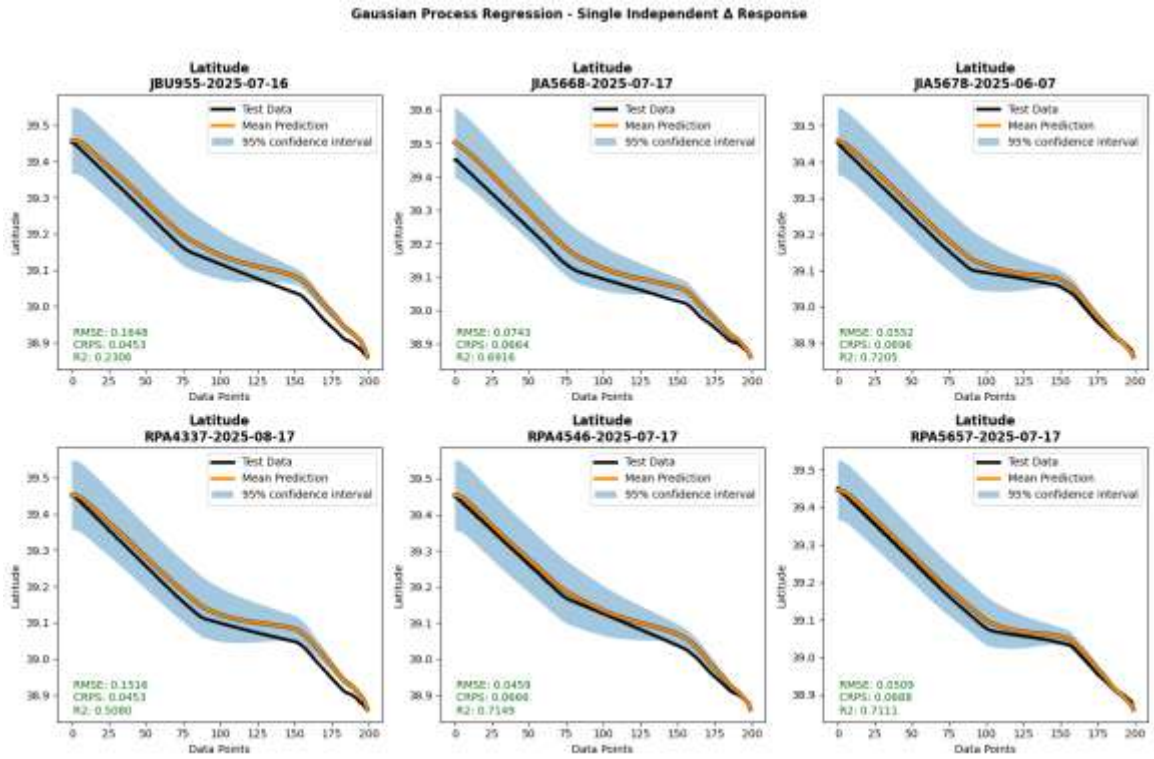


Figure 60 GPR for Latitude in Model 1A

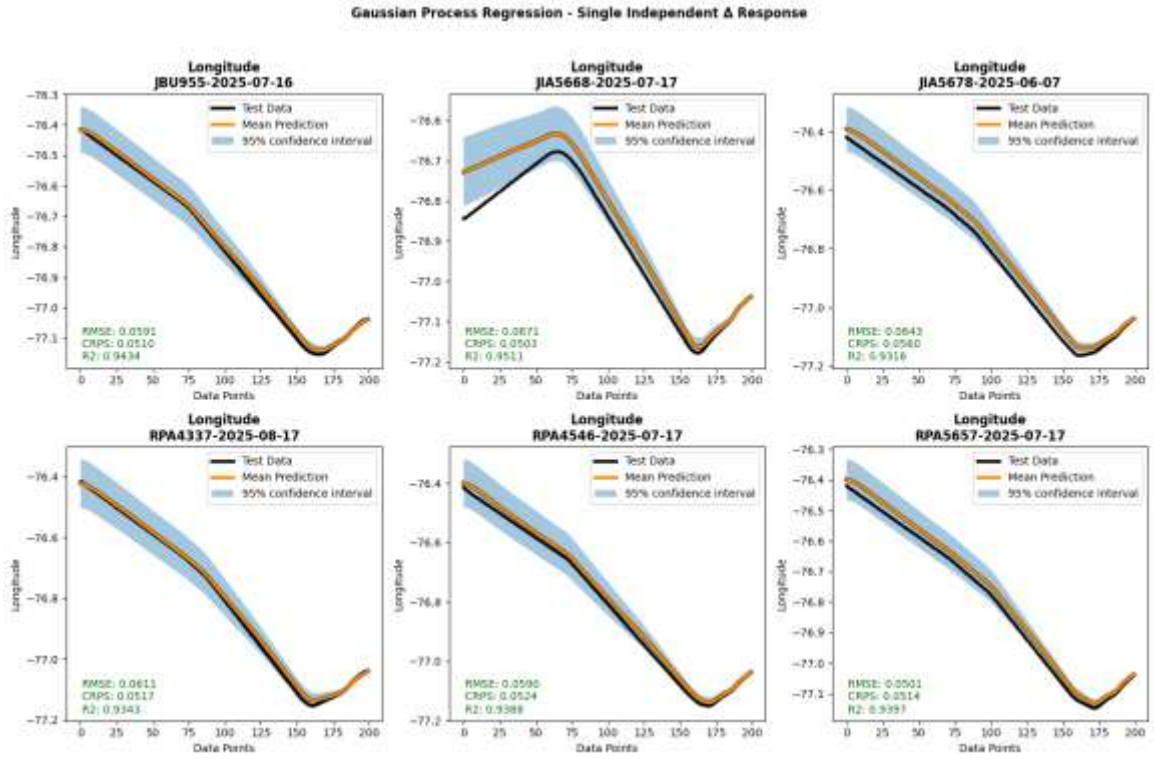


Figure 61 GPR for Longitude in Model 1B

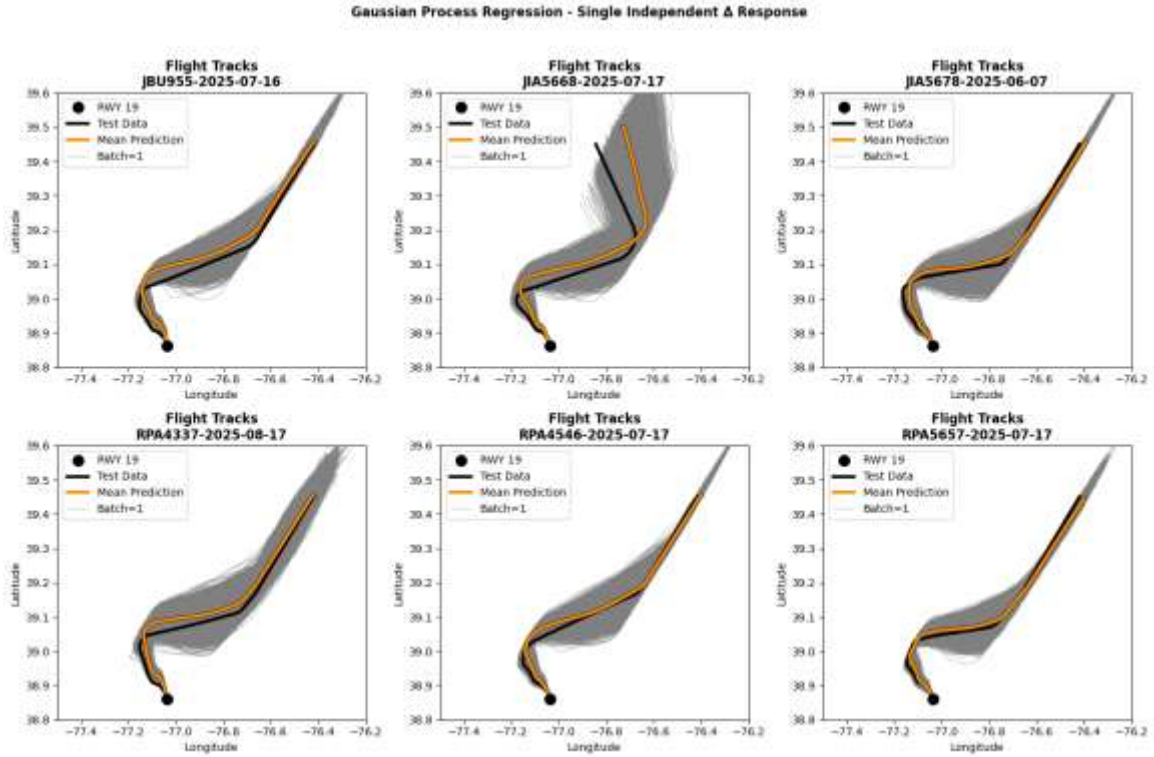


Figure 62 GPR for Flight Tracks in Model 1A and 1B

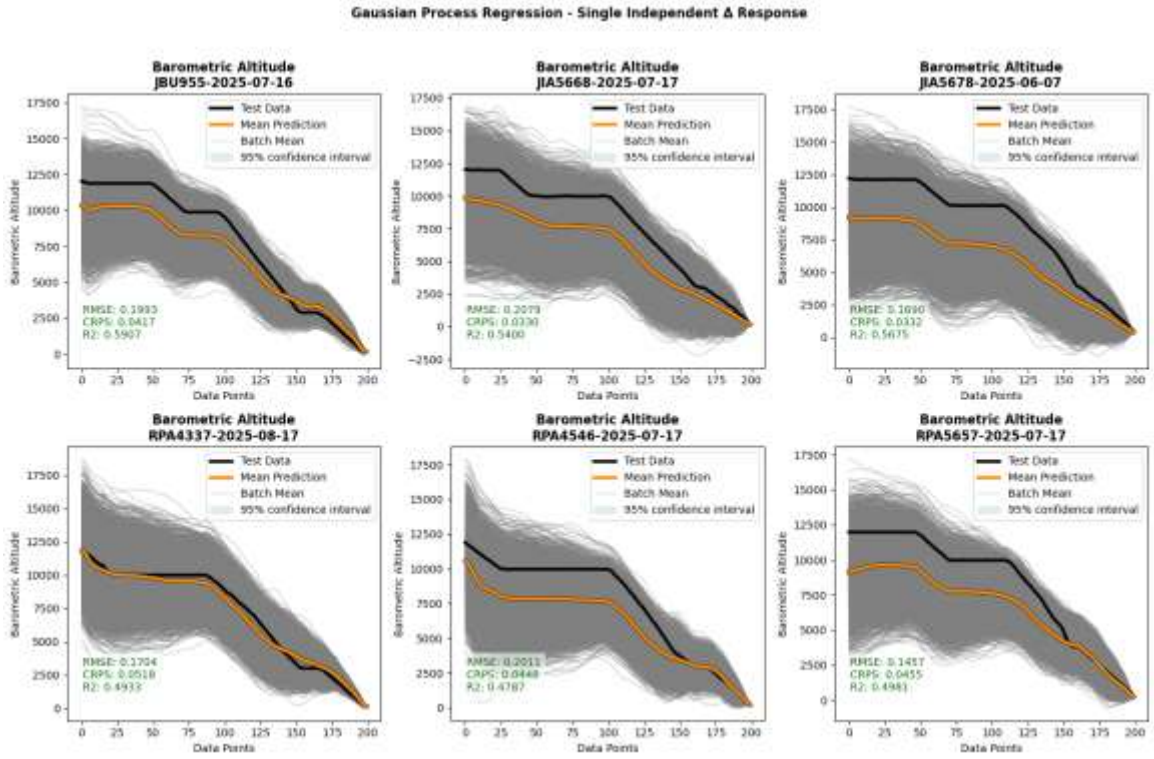


Figure 63 GPR for Barometric Altitude in Model 1C

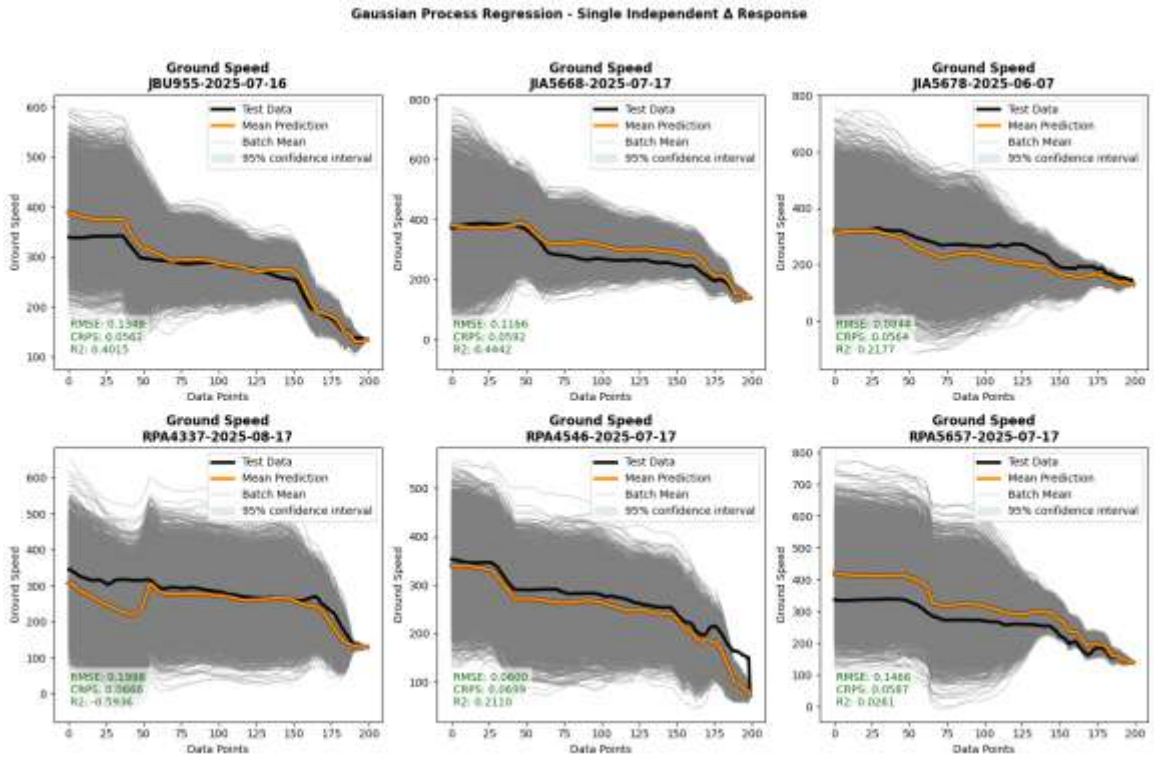


Figure 64 GPR for Ground Speed in Model 1D

Based on the preliminary analysis, the generated synthetic data exhibited high variance in the resulting parameters. Although the GPR algorithm captured model variance through uncertainty estimates and provided the required large sample size for collision risk modeling, some of the generated tracks still exhibited unrealistic aerodynamic properties. Specifically, the vertical and speed profiles presented both unrealistically high and low arrival patterns. Therefore, the posterior samples had to be combined into batches to ensure a smooth, narrow solution, and to limit unrealistic outliers in the confidence intervals.

The resulting synthetic tracks, combined in batches of 5, are shown in Figure 65. The associated batches for vertical and speed synthetic profiles are shown in Figure 66 and Figure 67, respectively. Although this reduced the number of unreasonable trajectories, the small number of posterior samples in each batch did not yield the desired performance. While flight tracks incorporated required variance, the vertical and speed profile still contained samples inconsistent with approach procedures. Hence, the batch size increased to 10, as shown in Figure 68, Figure 69, and Figure 70. Overall, the synthetic data contained one million new tracks per seed generated from $batch_size \times 1{,}000{,}000 = 10{,}000{,}000$ posterior samples.

The model performance metrics across all seed samples on the normalized scale are provided in Table 14. On average, models with latitude and longitude responses had lower RMSE values compared to Models 1C and 1D, which indicated fewer errors in their corresponding GPR implementation. CRPS values were lower than RMSE across the models, suggesting good predictive capability. Similar to RMSE, R^2 was higher in the first two models, suggesting a better explanation of data variance. Lower R^2 scores for altitude and speed data primarily reflect the inherent noise in these parameters.

The next step was to evaluate the synthetic data parameters for realism. The synthetic samples for the first seed track (*JBU955-2025-07-16*) were selected for validation as a potential candidate solution for collision risk modeling.

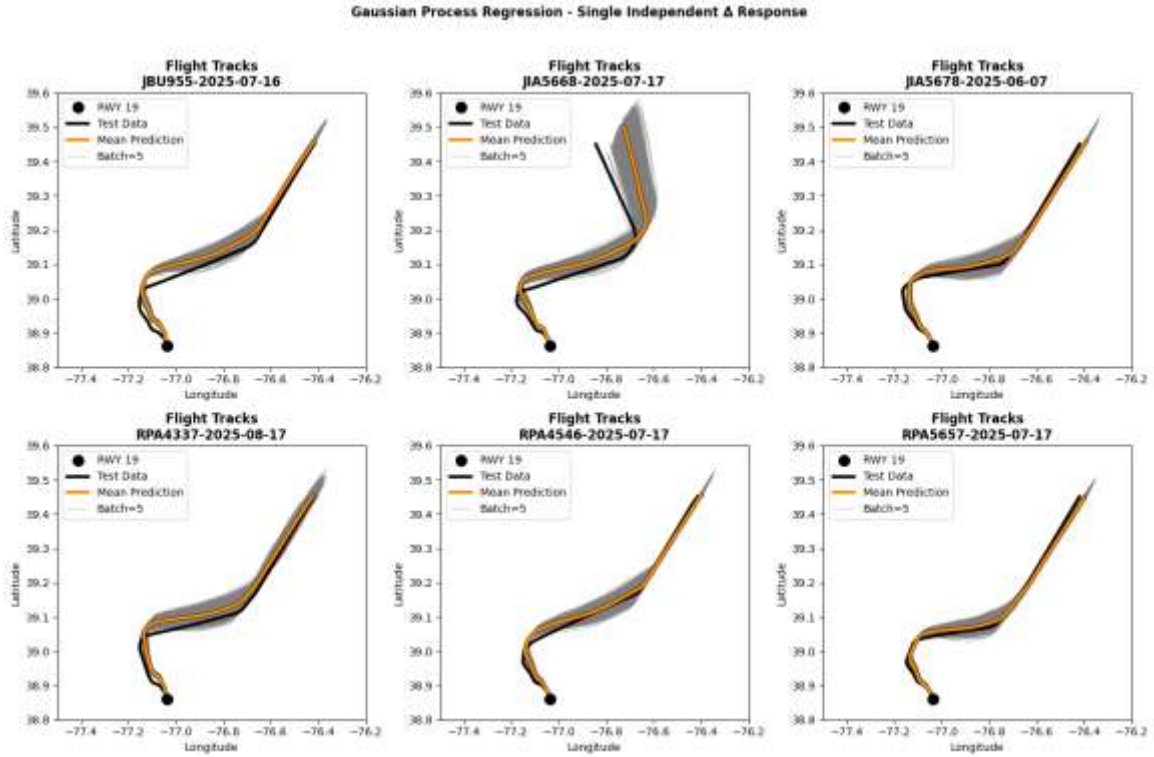


Figure 65 GPR for Flight Tracks in Model 1A and 1B (Batch = 5)

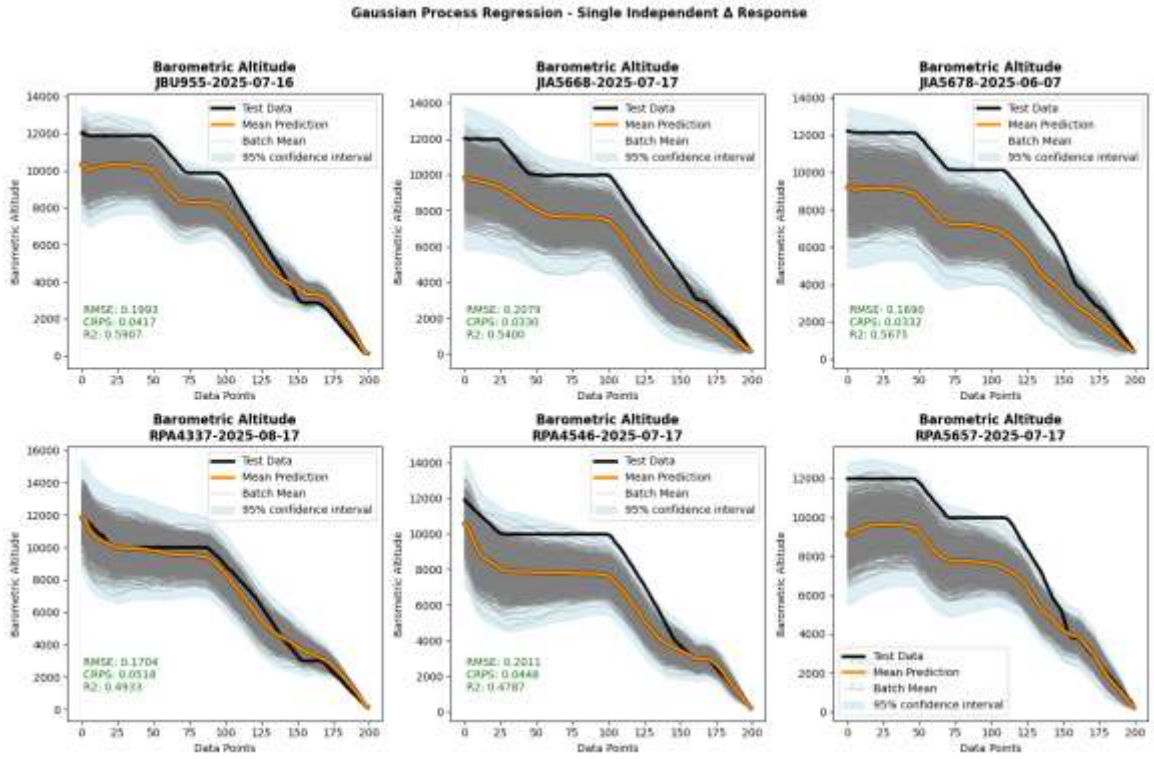


Figure 66 GPR for Barometric Altitude in Model 1C (Batch = 5)

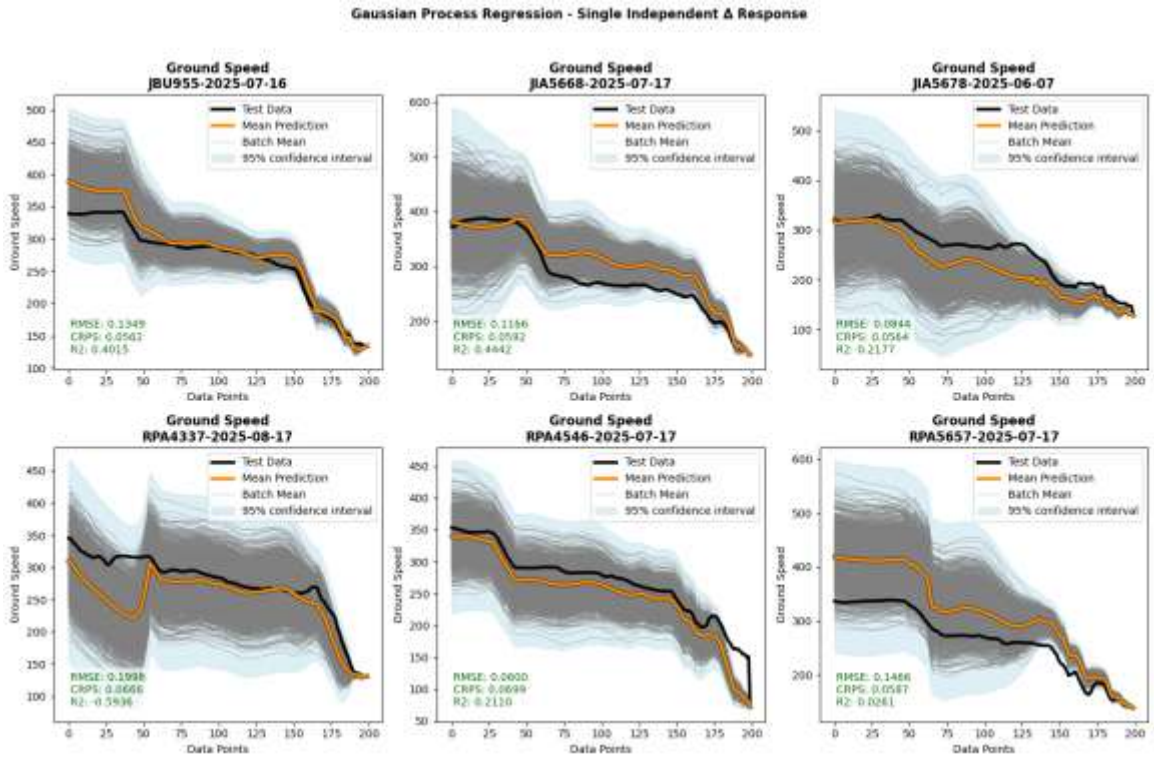


Figure 67 GPR for ground Speed in Model 1D (Batch = 5)

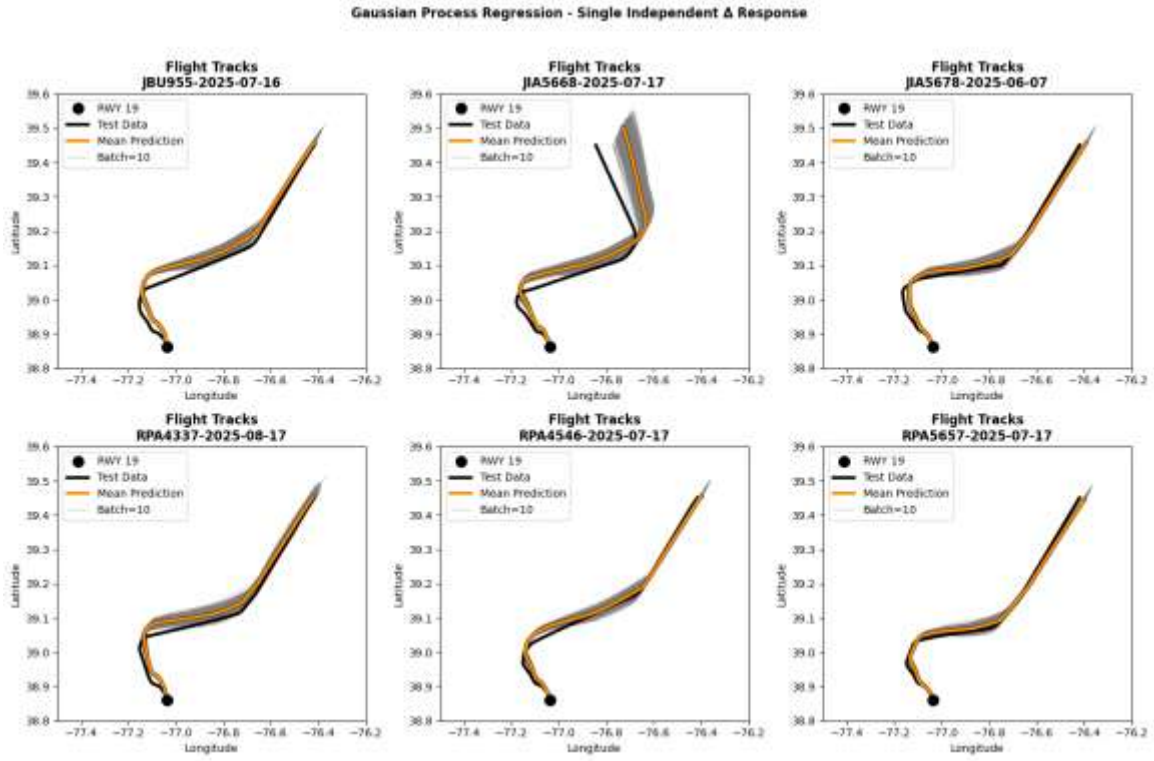


Figure 68 GPR for Flight Tracks in Model 1A and 1B (Batch = 10)

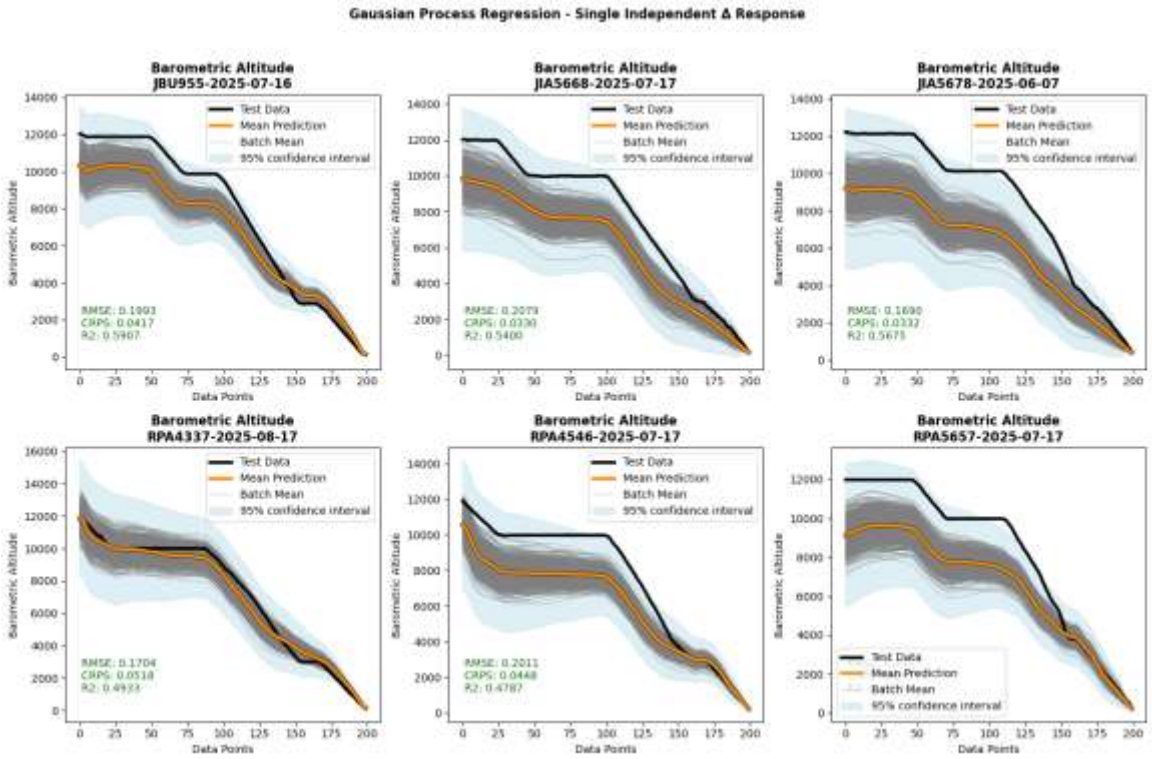


Figure 69 GPR for Barometric Altitude in Model 1C (Batch = 10)

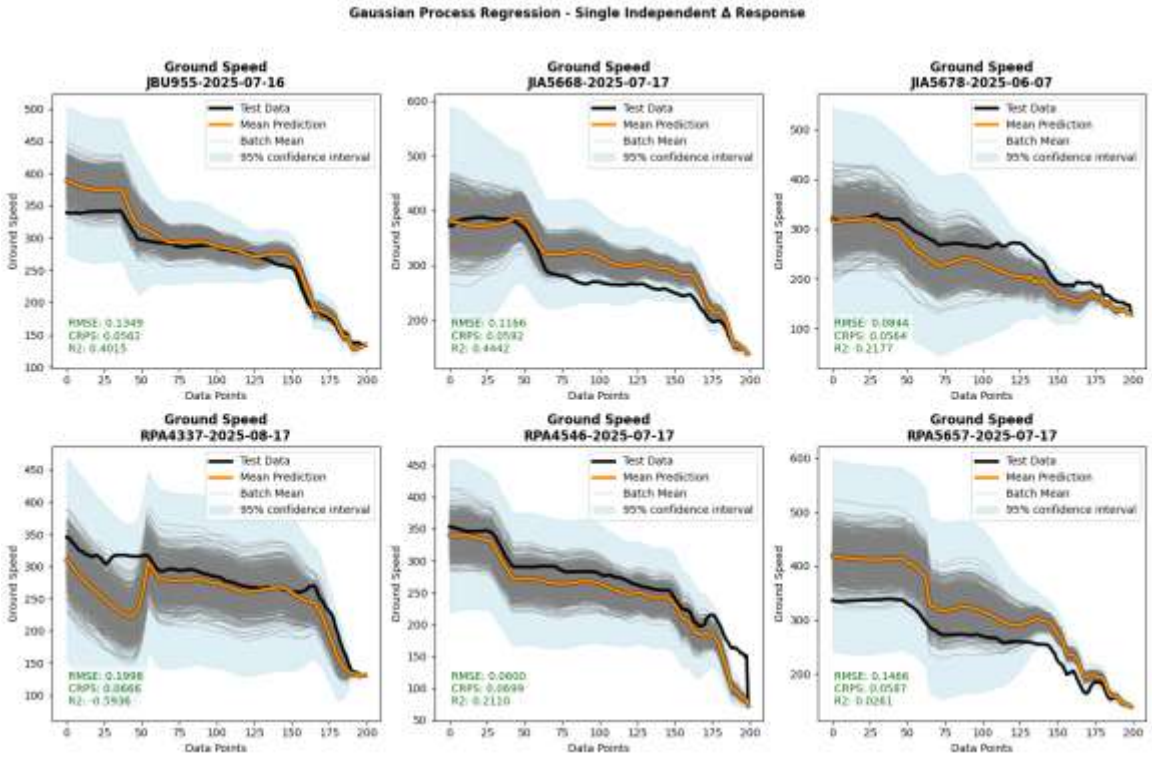


Figure 70 GPR for Ground Speed in Model 1D (Batch = 10)

Table 14 Performance Metrics for Model 1

GPR Model	Seed Track	RMSE	CRPS	R^2
Model 1A Latitude	JBU955-2025-07-16	0.1648	0.0453	0.2306
	JIA5668-2025-07-17	0.0743	0.0663	0.6916
	JIA5678-2025-06-07	0.0552	0.0695	0.7205
	RPA4337-2025-08-17	0.1516	0.0452	0.5080
	RPA4546-2025-07-17	0.0459	0.0665	0.7149
	RPA5657-2025-07-17	0.0509	0.0688	0.7111
	Average	**0.0905**	**0.0603**	**0.5961**
Model 1B Longitude	JBU955-2025-07-16	0.0591	0.0510	0.9434
	JIA5668-2025-07-17	0.0671	0.0504	0.9511
	JIA5678-2025-06-07	0.0643	0.0560	0.9316
	RPA4337-2025-08-17	0.0611	0.0516	0.9343
	RPA4546-2025-07-17	0.0590	0.0524	0.9388
	RPA5657-2025-07-17	0.0501	0.0514	0.9397
	Average	**0.0601**	**0.0521**	**0.9398**
Model 1C Barometric Altitude	JBU955-2025-07-16	0.1993	0.0463	0.5907
	JIA5668-2025-07-17	0.2079	0.0370	0.5400
	JIA5678-2025-06-07	0.1690	0.0358	0.5675
	RPA4337-2025-08-17	0.1704	0.0546	0.4933
	RPA4546-2025-07-17	0.2011	0.0517	0.4787
	RPA5657-2025-07-17	0.1457	0.0492	0.4981
	Average	**0.1822**	**0.0458**	**0.5281**
Model 1D Ground Speed	JBU955-2025-07-16	0.1349	0.0608	0.4015
	JIA5668-2025-07-17	0.1166	0.0607	0.4442
	JIA5678-2025-06-07	0.0844	0.0644	0.2177
	RPA4337-2025-08-17	0.1998	0.0695	-0.5936
	RPA4546-2025-07-17	0.0600	0.0761	0.2110
	RPA5657-2025-07-17	0.1466	0.0643	0.0261
	Average	**0.1237**	**0.0660**	**0.1178**

5.2.1.3 Validation Framework

One million synthetic flight tracks, along with associated barometric altitude and ground speed profiles, were extracted from the GPR models for a single representative seed sample (*JBU955-2025-07-16*), as shown in Figure 71. If the synthetic solution incorporates realistic trajectory behavior, an unlimited number of synthetic samples can be further extracted from the model.

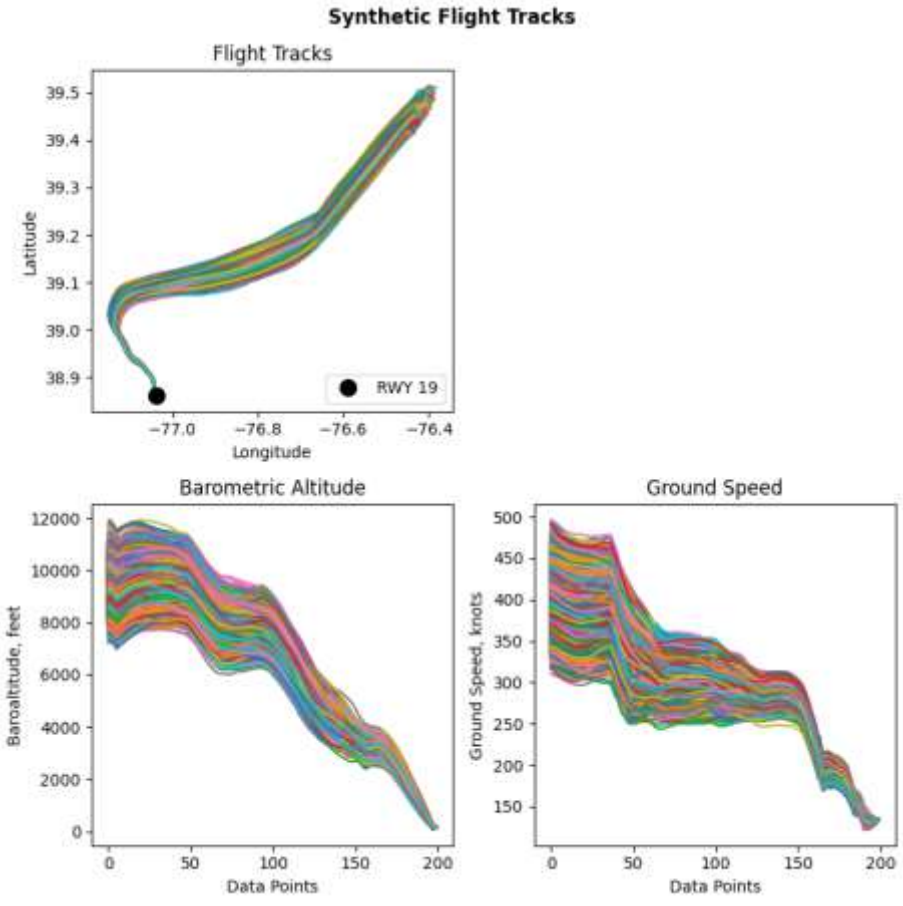


Figure 71 Synthetic Flight Tracks from Model 1

Several measurements in the synthetic parameters were collected for comparison against historical records described in Section 5.1.5. The historical dataset contains the original set of 215 seed tracks shown in Figure 54, which were preserved for validation. The collected measurements included lateral, vertical, and speed dispersion at various locations along the track from the runway threshold, and the overall flight track length.

The Along-Track Distance was calculated across all tracks in both synthetic and historical datasets (Figure 72). The distances for synthetic tracks, shown in blue, were normally distributed with a mean of 55 nm. In contrast, the historical figures, shown in gray, exhibited roughly unimodal, symmetric distributions, with most observations concentrated between 56 and 60 nm. The Wasserstein distance of 0.2596 indicated strong statistical differences in the provided distributions. The shorter distances observed in normally distributed synthetic tracks can be explained by the concentration of generated samples around a single representative seed track.

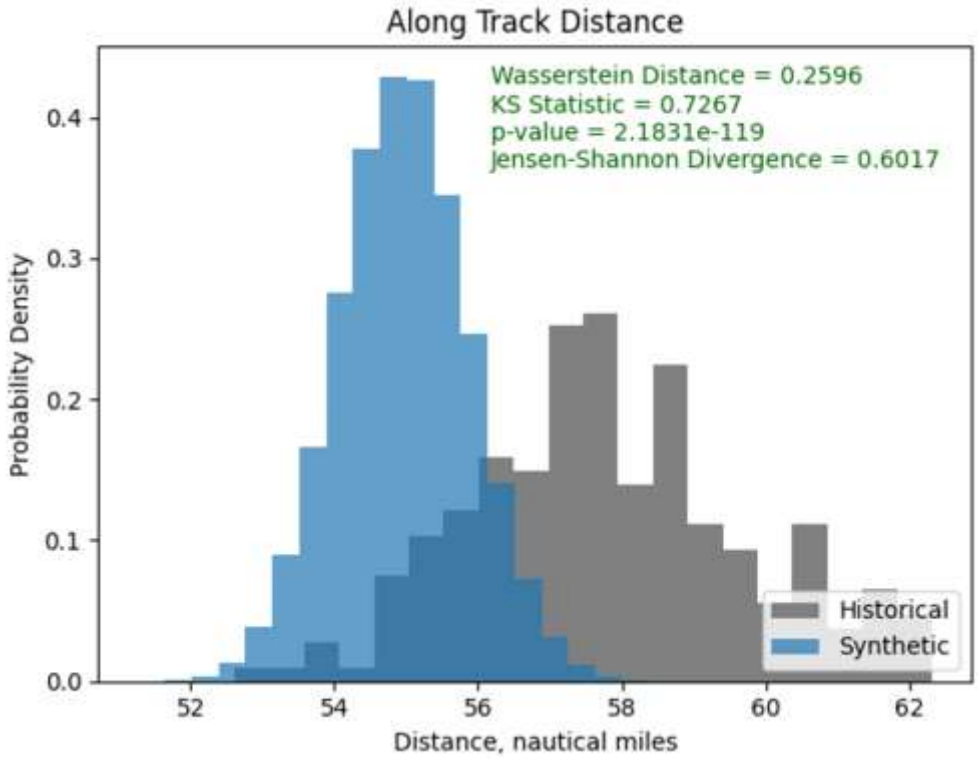


Figure 72 Along Track Distance Distribution Comparison in Model 1

The measurements for lateral, vertical, and speed dispersion at 3, 6, and 9 nm from the runway threshold summarized corresponding differences between synthetic and historical tracks (Figure 73). The synthetic data across all parameters were normally distributed due to the large sample size of generated samples.

The synthetic distributions for ground speed values (purple) resulted in a narrow concentration around the original seed sample. Although the synthetic speeds exhibited low variance, they fell within the historical data range, implying realistic patterns in the profile. The resulting narrow distributions can be attributed to the increased batch size used in the Generating framework.

Noticeably, the vertical dispersion (green) at 9 nm from the runway threshold in both synthetic and historical data presented identical distributions. The statistical similarities were confirmed by low scores in the Wasserstein distance and KS Statistics, 0.022 and 0.05, respectively. The remaining vertical dispersion near the runway threshold led to overlapping distributions with lower mean altitudes.

In contrast, the lateral deviations (blue) in the synthetic tracks were populated further to the right on the historical data spectrum. Large discrepancies demonstrate significant deviations at the final approach fix between synthetic and archival flight trajectories. The allocated waypoints from the RNAV (GPS) Y approach procedure for Runway 19 at DCA (Figure 49) show a mismatch between synthetic and historical data points in Figure 75. However, the synthetic tracks were supposed to capture this approach more accurately. Therefore, the synthetic track profile failed to resemble the original flight path.

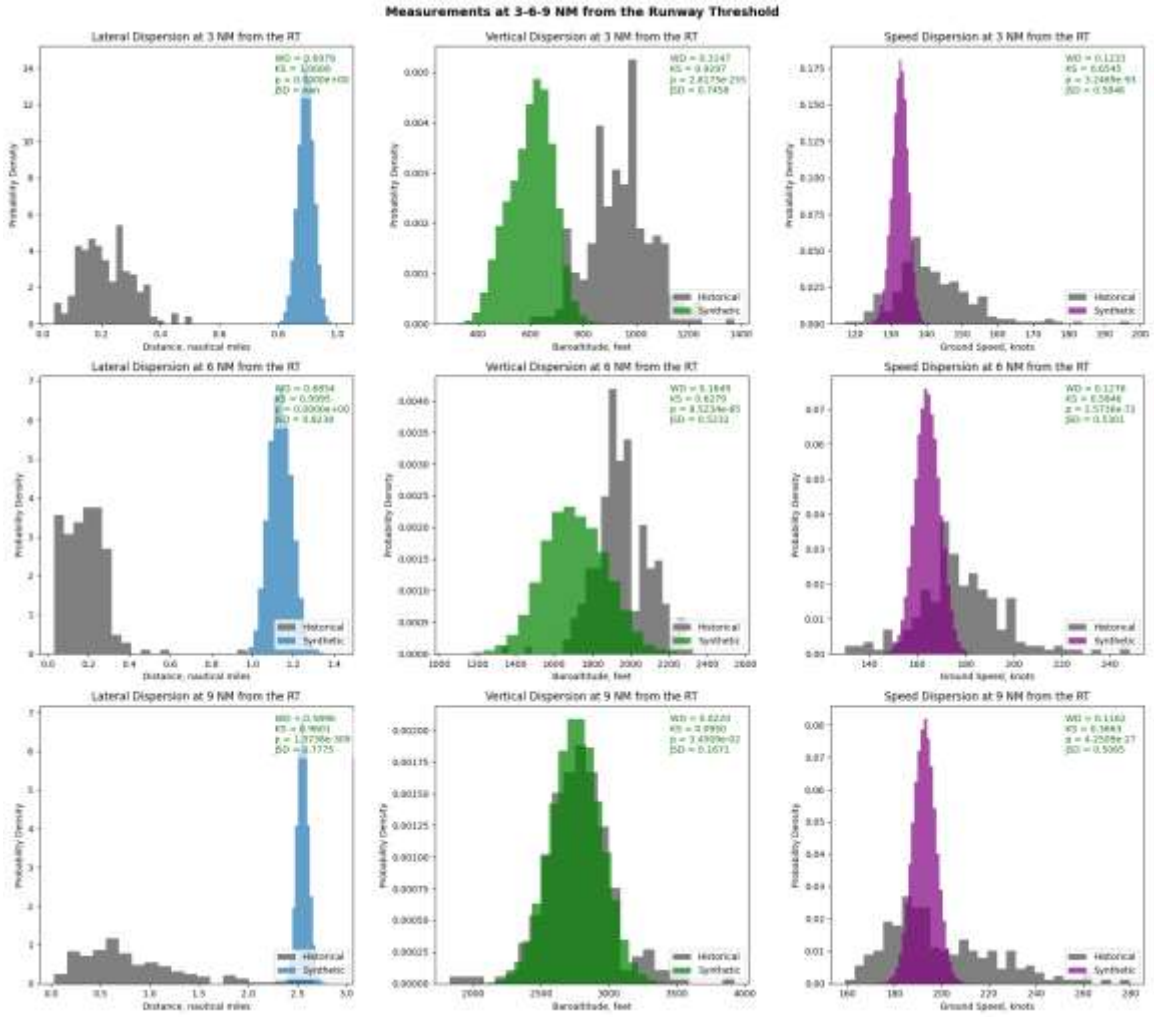


Figure 73 Distribution for Measurements at 3, 6, and 9 NM in Model 1

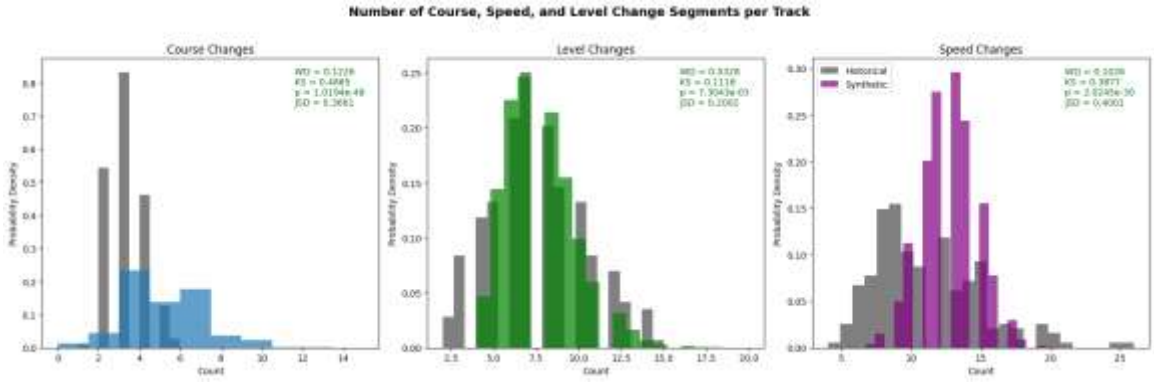


Figure 74 Distribution for Individual Flight Tracks Characteristics in Model 1

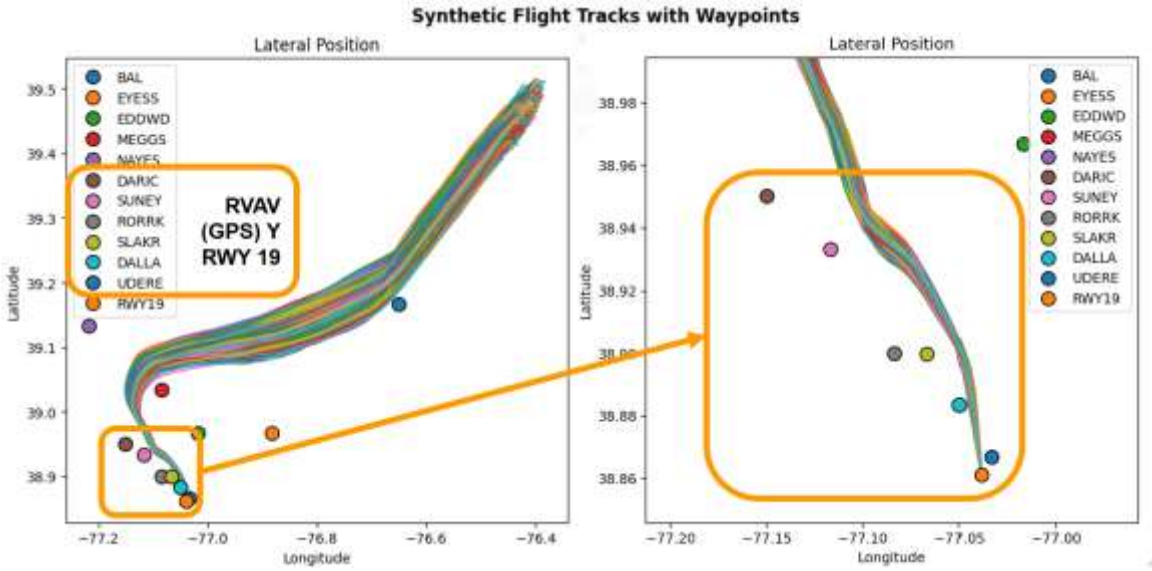


Figure 75 Synthetic Flight Tracks with Waypoints from Published Arrival Procedures

Despite significant statistical differences in the overall synthetic profiles, the individual flight-track characteristics showed strong resemblance between the artificially generated and actual flight patterns. In Figure 74, the numbers for level and speed changes accurately represented the behavior of aircraft trajectories, verified by low statistical scores in both corresponding distributions. However, the average number of course changes in synthetic tracks was moderately higher than in the representative seed tracks. The slight inconsistencies in the number of course changes support the overall conclusion that the artificially generated tracks failed to capture trajectory behavior.

5.2.2 GPR Models with Single Response – Predict Next n^{th} Position

This section describes another variation of the GPR models with a single response, but with a modified set of predictors and responses. A different GPR implementation presents an opportunity to perform a sensitivity analysis of the training parameter selection. Due to the modular design approach, a modified strategy was employed for predictors and responses allocation in the Learning framework, while the generating and validating mechanisms were reused from the previous section. The train and seed data are the same as in the previous implementation.

5.2.2.1 Learning Framework

Two independent GPR models, each with a single target response variable, were constructed to predict the latitude and longitude of the synthetic flight tracks. The models, along with the initial set of predictor parameters, are provided in Table 15. Instead of converting the response data into delta values, as in the previous set of GPR models, latitude and longitude values were provided in absolute coordinates.

Table 15 GPR Models with Single Responses in Absolute Coordinates

GPR Model	Predictors	Response
Model 2A	$X \in \mathbb{R}^{50 \times 200 \times 6}$: 1. Position 2. Latitude 3. Longitude 4. Heading 5. Ground Speed 6. Barometric Altitude	Latitude
Model 2B	$X \in \mathbb{R}^{50 \times 200 \times 6}$: 1. Position 2. Latitude 3. Longitude 4. Heading 5. Ground Speed 6. Barometric Altitude	Longitude

The model dataset for predictors and responses was organized as follows:

1. Row Alignment: each row of predictors corresponds to the position in the track where the response is predicted.
2. Sequential Mapping: the first row predicts the response at the n^{th} position. The second row predicts the response at the $(n+1)^{th}$ position, and so on.
3. End-of-Track Mapping: the responses for the last n positions of the track are paired with their corresponding predictor rows to maintain alignment.

The organization of predictors and responses, with n set to 5, is summarized in Table 16.

Table 16 Predictors and Responses in Models 2A and 2B

Tracks #1, 2, 3, …, model size = train size + test size;						n = 5	
Predictors						Response #1	Response #2
Position	Latitude	Longitude	Heading	Speed	Altitude	Latitude	Longitude
1	lat_1	lon_1	h_1	gs_1	alt_1	lat_5	lon_5
2	lat_2	lon_2	h_2	gs_2	alt_2	lat_6	lon_6
3	lat_3	lon_3	h_3	gs_3	alt_3	lat_7	lon_7
4	lat_4	lon_4	h_4	gs_4	alt_4	lat_8	lon_8
5	lat_5	lon_5	h_5	gs_5	alt_5	lat_9	lon_9
6	lat_6	lon_6	h_6	gs_6	alt_6	lat_{10}	lon_{10}
7	lat_7	lon_7	h_7	gs_7	alt_7	lat_{11}	lon_{11}
8	lat_8	lon_8	h_8	gs_8	alt_8	lat_{12}	lon_{12}
9	lat_9	lon_9	h_9	gs_9	alt_9	lat_{13}	lon_{13}
10	lat_{10}	lon_{10}	h_{10}	gs_{10}	alt_{10}	lat_{14}	lon_{14}
…							
191	lat_{191}	lon_{191}	h_{191}	gs_{191}	alt_{191}	lat_{195}	lon_{195}
192	lat_{192}	lon_{192}	h_{192}	gs_{192}	alt_{192}	lat_{196}	lon_{196}
193	lat_{193}	lon_{193}	h_{193}	gs_{193}	alt_{193}	lat_{197}	lon_{197}
194	lat_{194}	lon_{194}	h_{194}	gs_{194}	alt_{194}	lat_{198}	lon_{198}
195	lat_{195}	lon_{195}	h_{195}	gs_{195}	alt_{195}	lat_{199}	lon_{199}
196	lat_{196}	lon_{196}	h_{196}	gs_{196}	alt_{196}	lat_{200}	lon_{200}
197	lat_{197}	lon_{197}	h_{197}	gs_{197}	alt_{197}	lat_{197}	lon_{197}
198	lat_{198}	lon_{198}	h_{198}	gs_{198}	alt_{198}	lat_{198}	lon_{198}
199	lat_{199}	lon_{199}	h_{199}	gs_{199}	alt_{199}	lat_{199}	lon_{199}
200	lat_{200}	lon_{200}	h_{200}	gs_{200}	alt_{200}	lat_{200}	lon_{200}

Both GPR models contained the same predictors (Table 15). After observing the training data, hyperparameters were optimized through log marginal likelihood maximization (Table 17). In contrast to the previous GPR model for Δ latitude, the smoothness parameter in the Matérn kernel was set to 2.5 in Model 2A instead of 0.5. A

higher smoothness value produced twice-differentiable, much smoother functions, which represented the flight path rather than the differences in values along the track. Similar to Model 2A, a much smoother RBF kernel in Model 2B captured gradual changes in the longitudinal values.

Table 17 Initial Kernel Functions in Model 2A and 2B

GPR Model	Kernel Function	Log Marginal Likelihood
Model 2A	Matérn Kernel with White Noise: $k(x,x') = 14.5^2 \cdot k_{Matern}\begin{pmatrix} x,x';\ell = 32.3, \\ v = 2.5 \end{pmatrix} + \sigma_n^2\delta(x,x');$ $\sigma_n^2 = 0.000232$	9702.299854
Model 2B	RBF Kernel with White Noise: $k(x,x') = 14.5^2 \cdot k_{RBF}(x,x';\ell = 100000) + \sigma_n^2\delta(x,x');$ $\sigma_n^2 = 0.35$	-961.797489

While the log marginal likelihood remained large in Model 2A, the log marginal likelihood presented a negative value in Model 2B. Therefore, to optimize the model learning, the predictor set was modified to exclude the *Heading* parameter in Model 2B. Due to the inherent dependencies between latitude and longitude profiles, the same modified set of inputs was used for Model 2A. The exclusion of *Heading* yielded much larger positive log marginal likelihood values in both models (Table 18).

Table 18 Modified Kernel Functions in Model 2A and 2B

GPR Model	Kernel Function	Log Marginal Likelihood
Model 2A	Matérn Kernel with White Noise: $k(x,x') = 16.3^2 \cdot k_{Matern}\left(x,x';\ell = 24.2,\ \nu = 2.5\right) + \sigma_n^2\delta(x,x');\ \sigma_n^2 = 0.00000209$	19958.549101
Model 2B	Matérn Kernel with White Noise: $k(x,x') = 32.3^2 \cdot k_{RBF}(x,x';\ell = 4.46) + \sigma_n^2\delta(x,x');\ \sigma_n^2 = 0.00000153$	20400.259631

5.2.2.2 Generating Framework

The trained GPR models that predicted latitude and longitude in absolute coordinates for the seed tracks are shown in Figure 76 and Figure 77, respectively. In contrast to Models 1A and 1B, the responses did not require further restructuring because the resulting coordinates are in the original format. Each model provided the posterior and predictive distributions, along with uncertainty estimates, for the seed tracks and their corresponding response parameters. The uncertainty estimates were derived from 95% confidence intervals for each prediction point along the flight path. In the plots, the purple lines represent a collection of training tracks. The black lines show actual seeds from historical data. The orange lines show the prediction mean derived from the predictive distribution, with associated confidence intervals for uncertainty estimates shown in light blue.

To facilitate the interpretability of the results, predicted flight tracks for each seed, with independently generated latitude and longitude parameters, were plotted together in

Figure 78. In the plots, a collection of gray lines represents 1000 samples drawn from the corresponding posterior distribution.

The resulting synthetic tracks remarkably resembled the seed samples, indicating either great GPR predictive capability or overfitting. However, a low variance observed in posterior samples indicates overfitting to the training data in the generated models. Therefore, the entire set of synthetic tracks generated by Models 2A and 2B cannot be used for collision risk modeling.

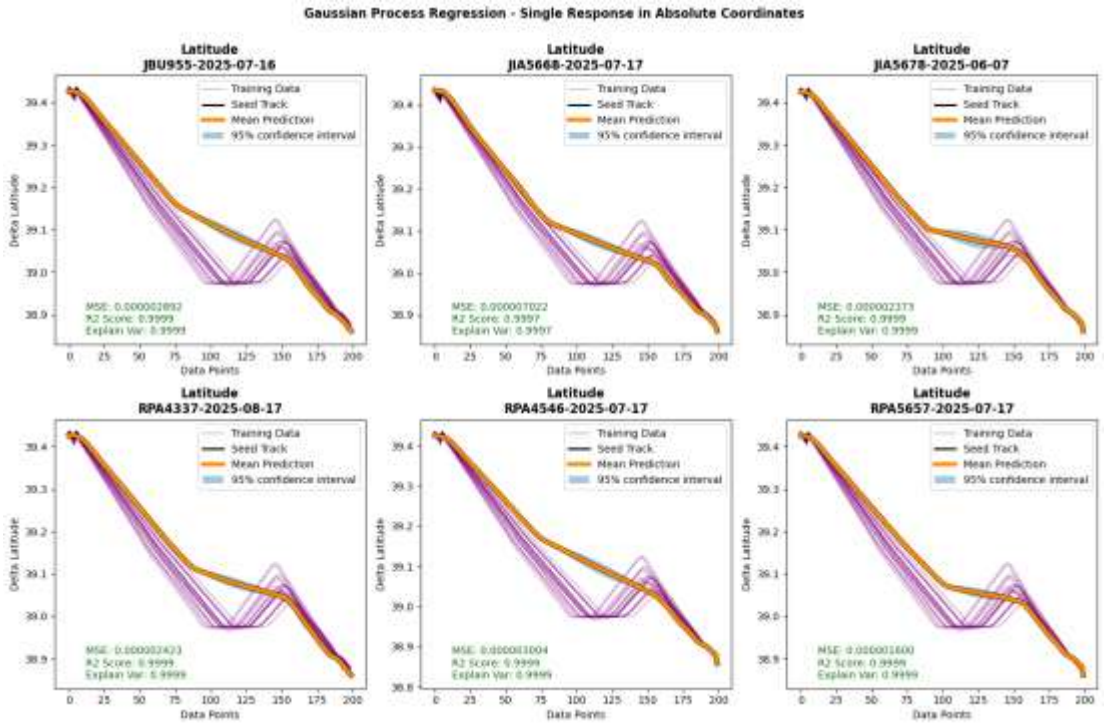


Figure 76 GPR for Latitude in Absolute Coordinates in Model 2A

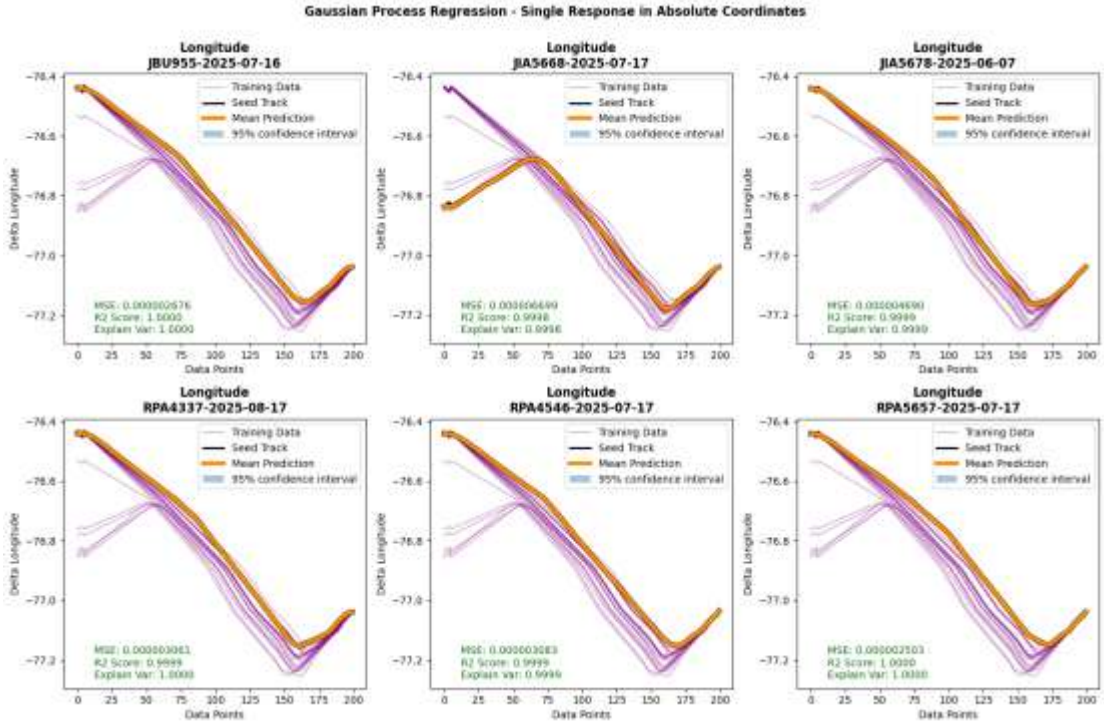


Figure 77 GPR for Longitude in Absolute Coordinates in Model 2B

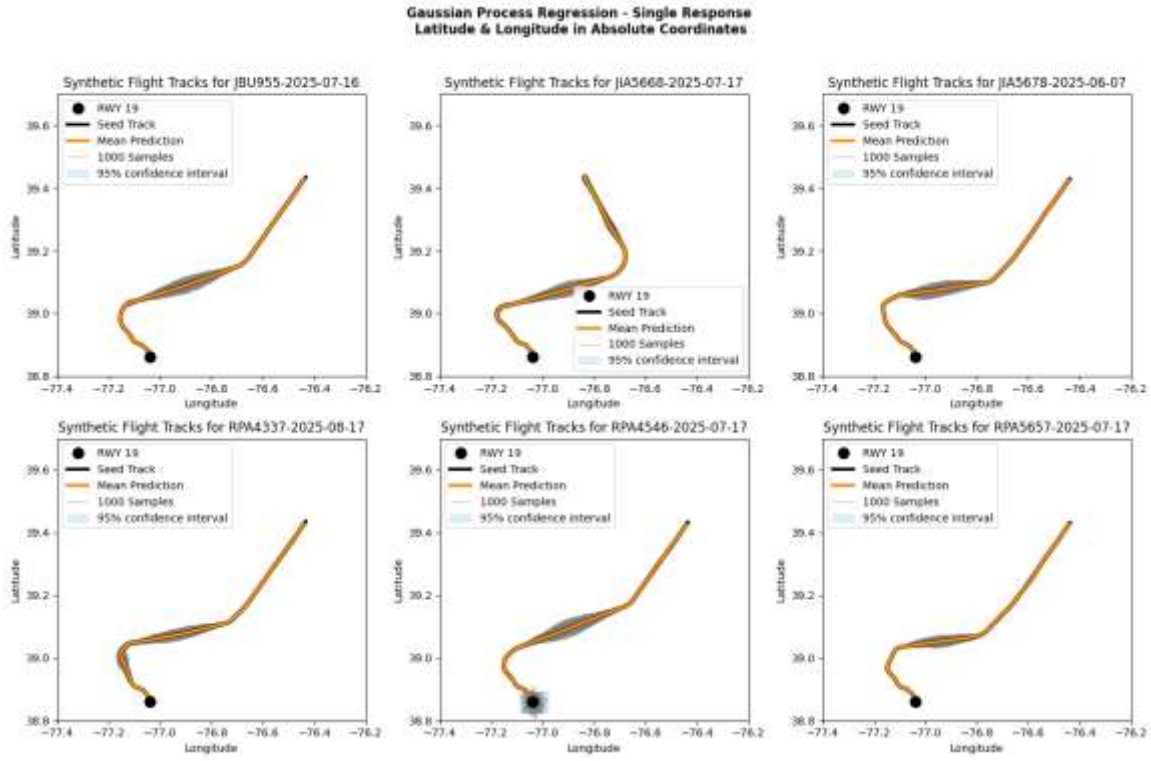


Figure 78 Synthetic Flight Tracks in Model 2A and 2B

5.2.3 GPR Model with Encoded Multi-Response

This section describes a GPR model that simultaneously and dependently generates flight-track outputs by encoding them into a single feature. The resulting encoded multi-response preserves dependencies among the track parameters while adhering to GP mathematical principles by producing a single output.

5.2.3.1 Learning Framework

A pair of coordinates (i.e., latitude and longitude) can be encoded into a single response through a combination of Heading and Distance indications along the flight track. Given the reference point, the coordinates can be extracted from the response during the Generating framework's decoding phase.

An encoding method relied on numerical placement of heading and distance parameters in the resulting combinations. The heading always ranges from 0° to 360°, requiring three places in the encoded response. The average distance between consecutive points across all flight tracks was 0.32 nm, while the minimum and maximum distances were 0 and 1.36 nm, respectively (Table 19). Consequently, the distance parameter also required at least three numerical positions in the encoded response. Therefore, each parameter was scaled by a factor of 1000 to preserve three decimal places for distance, so their values do not overlap. The sum of the two values yielded the final encoded response.

Table 19 Distance between Consecutive Points

Minimum	0
Maximum	1.36054
Average	0.323892
Median	0.318584
Standard Deviation	0.079579

Equation 27 Encoded Value for Heading and Distance

$$encoded\ value = heading\ \times 1000 + distance\ \times 1000$$

As an example, for the heading of 230° and distance of 0.323892 nm, the encoded value is as follows:

Equation 28 Example for Encoded Value for Heading and Distance

$$encoded\ value = 230^{\circ} \times 1000 + 0.323892 \times 1000 = 230000 + 323.892 = 230324$$

The encoded values were converted to integers to drop decimal points for additional simplicity. Importantly, predictors and response data had to be scaled and normalized to facilitate GPR training.

The predictor set, $X \in \mathbb{R}^{50 \times 200 \times 5}$, comprised the following parameters across all training tracks:

1. Position on the flight track
2. Latitude
3. Longitude
4. An indication for the course change (a Boolean value with *True* or *False* categories)
5. Accumulated distance travelled

A custom kernel function was designed to capture discrete structure in the underlying GP function (Equation 29). The composite kernel function consisted of the Hamming kernel, the Matérn kernel, and White noise. The Hamming kernel measured covariance between bit (i.e., placement) patterns by assigning higher similarity to patterns that differ in fewer bits based on Hamming distance. Similar to the prior GP implementations, the Matérn kernel allowed for capturing non-linear smoothness of the underlying function. The White kernel added independent Gaussian noise to the model.

Equation 29 Custom Kernel Function for Model 3

$$k(x, x') = k_{Hamming}(x, x'; \ell = 10) + k_{Matern}(x, x'; \ell = 1,\ v = 1.5) + \sigma_n^2 \delta(x, x')$$

After hyperparameter optimization, the log marginal likelihood was large and negative (-802.381540), indicating challenges in the underlying response structure.

5.2.3.2 Generating Framework

A single trained GPR model that predicted the encoded combination of heading and distance for each seed sample is shown in Figure 79. The decoding sequence is as follows:

1. Inverse the scaling applied in the Learning framework
2. Divide the encoded value by 1000 to extract the heading parameter:

Equation 30 Heading Extraction

$$heading = \frac{encoded\ value}{1000}$$

3. Subtract the heading multiplied by 1000 from the encoded value and divide the resulting value by 1000 to extract the distance parameter:

Equation 31 Distance Extraction

$$distance = \frac{encoded\ value\ - heading \times 1000}{1000}$$

For the example in Equation 28, the decoded heading and distance are as follows:

Equation 32 Example for Decoded Heading and Distance Values

$$heading = \frac{230324}{1000} = 230.324 \approx 230°,$$

$$distance = \frac{230324 - 230 \times 1000}{1000} = 0.324\, nm.$$

Following the decoding sequence, the GPR model was split into two models for the heading and distance responses, as shown in Figure 80 and Figure 81, respectively. The mean predictions, with associated 95% confidence intervals, successfully captured the variance in heading values along the track. However, the mean predictions for the distances between points on the track were substantially higher than those for the seed samples.

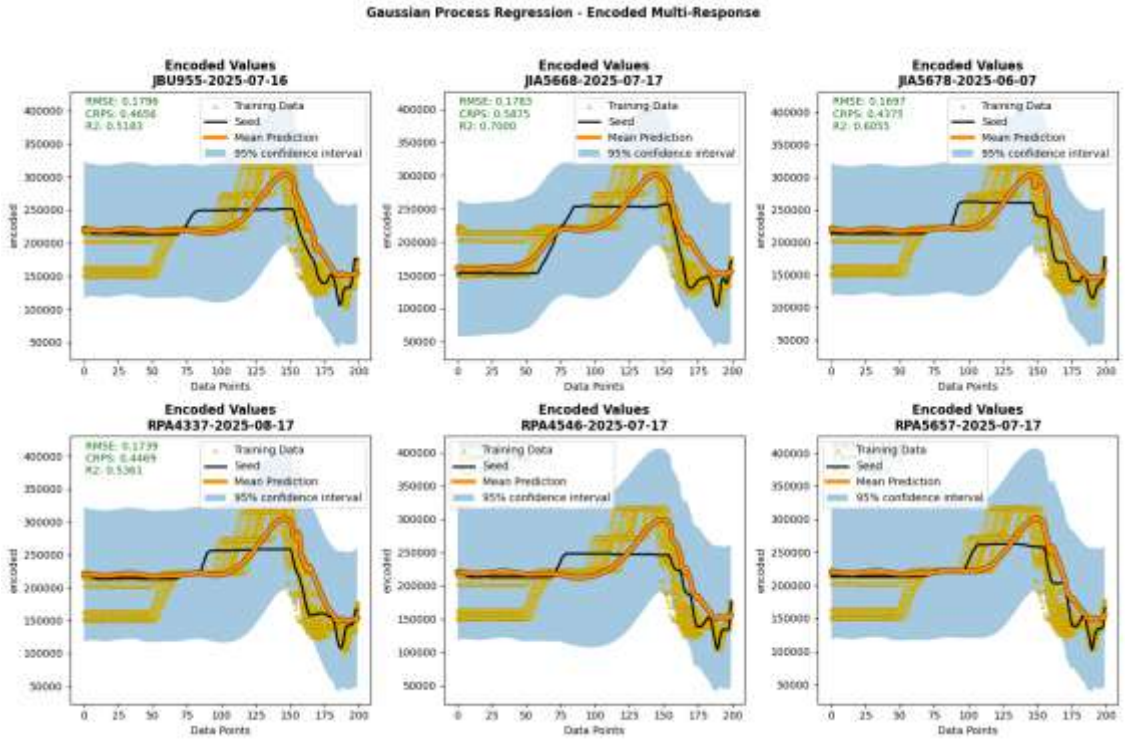


Figure 79 GPR with Encoded Multi-Response in Model 3

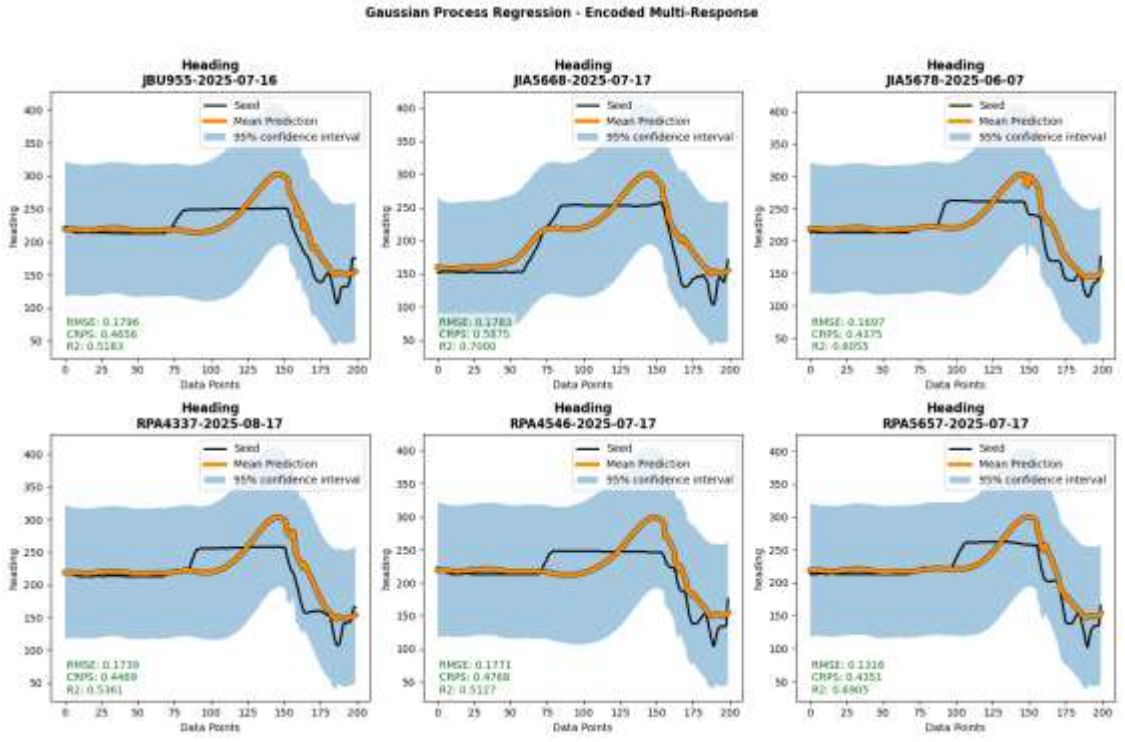


Figure 80 GPR for Heading in Model 3

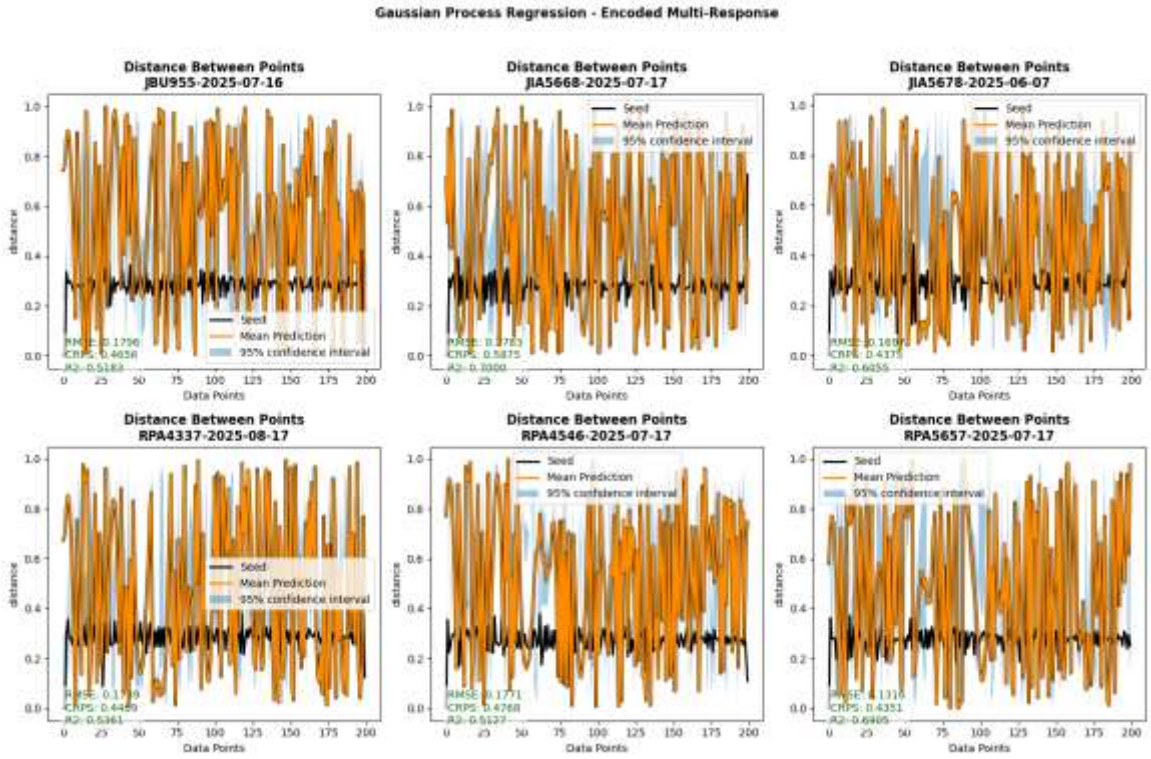


Figure 81 GPR for Distance in Model 3

Nevertheless, the next step focused on reconstructing latitude and longitude by setting the initial reference coordinates at the beginning of the track. The reference point, together with heading and distance indications, in the great-circle distance formula produced individual latitude and longitude values along the tracks (Figure 82):

Equation 33 Latitude and Longitude

$$lat_2 = arcsin(\sin(lat_1)\cos\delta + \cos(lat_1)\sin\delta\cos\theta),$$

$$lon_2 = lon_1 + atan2(\sin\theta\sin\delta\cos(lat_1)\cos\delta - \sin(lat_1)\sin(lat_2)),$$

where lat_1 is starting latitude, lat_2 is destination latitude, lon_1 is starting longitude, lon_2 is destination longitude δ is angular distance travelled, and θ is heading.

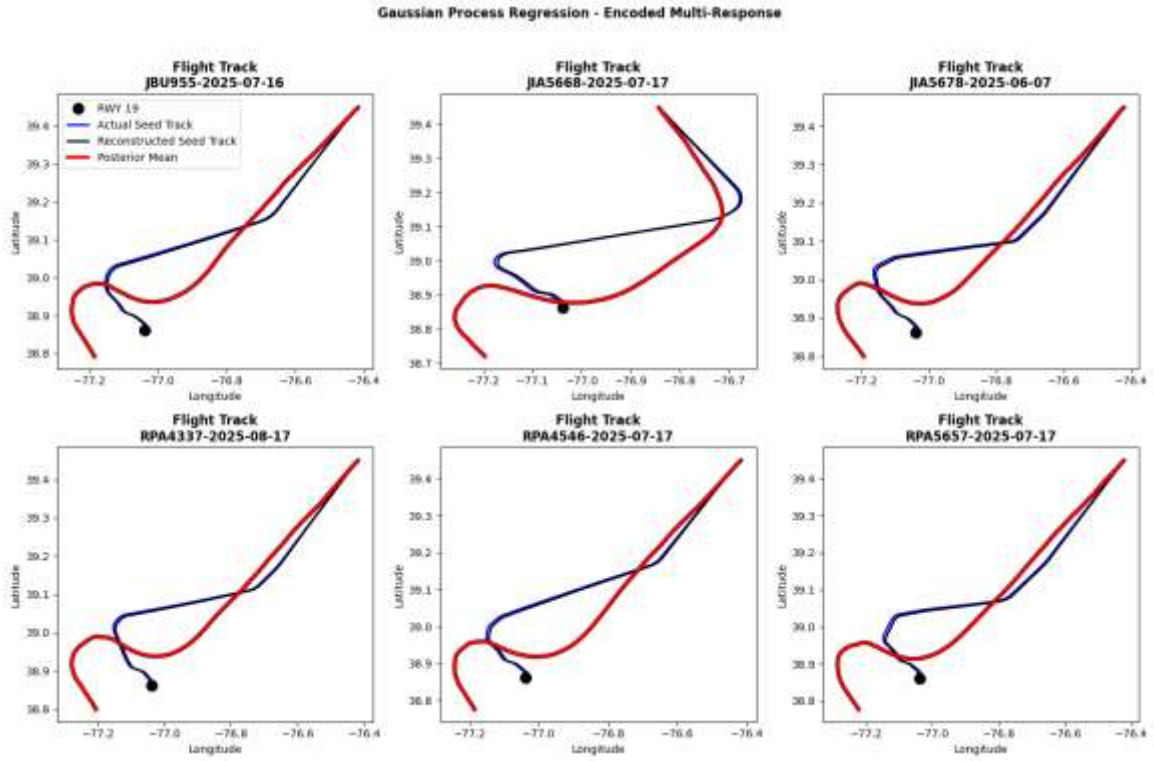


Figure 82 Predicted Flight Tracks with Forward Reconstruction in Model 3

Evidently, the forward reconstruction with the reference point at the beginning of the track resulted in trajectories that substantially missed the runway. Therefore, the forward reconstruction was replaced with backward reconstruction by setting the reference coordinate pair at the runway threshold.

The resulting trajectories based on the backward reconstruction are shown in Figure 83. Although all predicted flight tracks, shown in red by posterior mean, ended precisely at the runway threshold, the tracks failed to mimic the seed samples. Due to the model's much larger predicted distances, the resulting tracks tended to be much longer. In

addition to track length, the errors in heading values were attributed to the vastly different shapes of the predicted tracks.

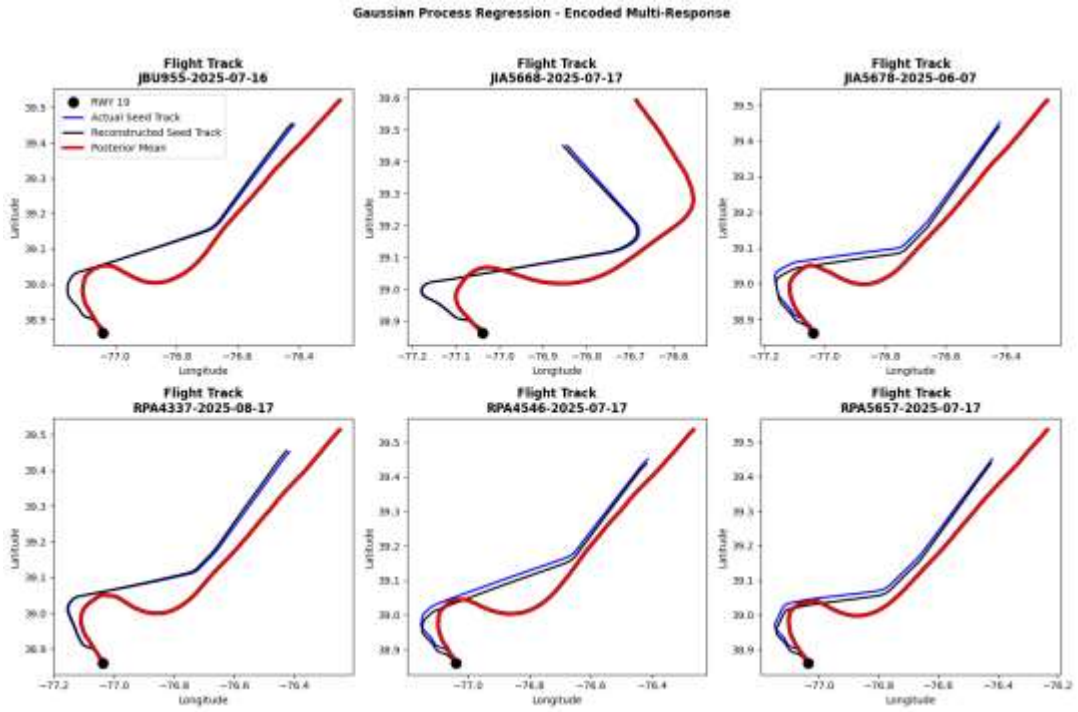


Figure 83 Predicted Flight Tracks with Backward Reconstruction in Model 3

The synthetic flight tracks for the best-performing model, *RPA5657-2025-07-17*, were generated from the posterior distribution in batches of 10 and 100 samples, as shown in Figure 84. However, the synthetic samples failed to capture trends and patterns in the seed track accurately. Overall, this GPR implementation failed to predict seed tracks despite generating multiple outputs simultaneously.

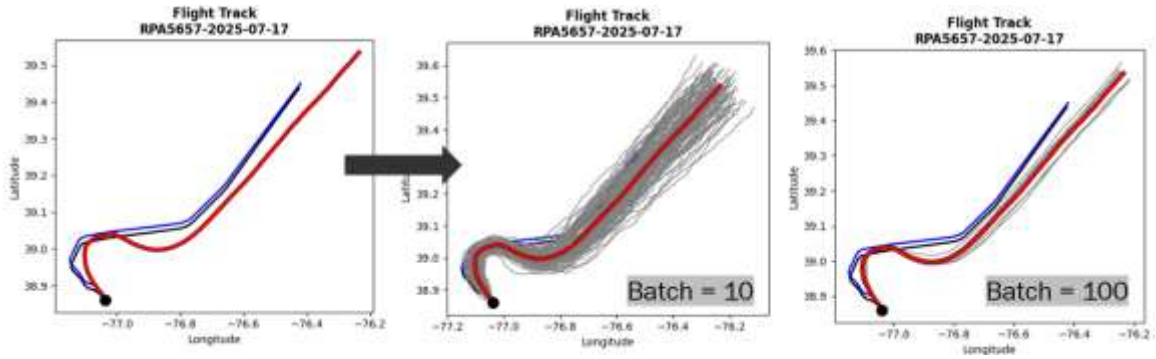


Figure 84 Synthetic Tracks from GPR with Encoded Multi-Response in Model 3

5.2.4 Deep GP with Encoded Multi-Response

This section describes the Deep GP implementation to address the pitfalls of the GPR model with combined encoded multi-response presented in the previous section.

5.2.4.1 Learning Framework

A Deep GP method comprises multiple stacked GPR in latent layers. While the Deep GP model used the same predictors and encoding sequence as in the previous section, its architecture employed a different mathematical approach.

The model architecture consisted of three GP layers, $Input\ Features \rightarrow Latent\ Layer(3) \rightarrow Latent\ Layer(1) \rightarrow Output(1)$, specified by $layers = [1, 3, 1, d]$, where:

- d is the number of input features,
- The first layer corresponds to the observed output y,
- Latent layers represent intermediate hidden layers.

The input layer used a composite kernel consisting of (1) the Matérn kernel capturing nonlinear effects, (2) the Linear kernel describing linear trends, and (3) the White noise preserving constant offsets in the underlying function:

Table 20 Kernel Function in Observation Layer

$$k(x, x') = k_{Matern}(x, x') + k_{Linear}(x, x') + \sigma_n^2 \delta(x, x')$$

Each hidden layer used the Matérn kernel with $\nu = 1.5$ assuming the generated functions were continuous and once differentiable. Additionally, each layer had Automatic Relevance Determination (ARD) set to *True*, implying that each dimension had its own length scale.

Instead of using all 50 training samples, the Deep GP model summarized the observations through 100 representative inducing points, thus significantly reducing computational complexity.

The multi-stage optimization was used to determine hyperparameter values in each layer, and consisted of three phases:

- Initial optimization to fix noise variance for 200 iterations
- Kernel hyperparameter refinement for another 200 iterations
- Full optimization to allow function adaptation to optimized hyperparameters for 1000 iterations

The optimized hyperparameter values in each layer were as follows:

- Observation layer: signal variance $\sigma^2 = 2.5046$, length scale $\ell = 3.0$, noise variance $\sigma_n^2 = 0.00000175$
- First latent layer: signal variance $\sigma^2 = 0.0138$, length scale $\ell = 5.1521$, noise variance $\sigma_n^2 = 0.0112$
- Second latent layer: signal variance $\sigma^2 = 0.4432$, length scale $\ell = 5.0$, noise variance $\sigma_n^2 = 0.6104$

The obtained log marginal likelihood was -1717.0918, signifying complexities in the generated function.

5.2.4.2 Generating Framework

Similar to the GPR implementation in the previous model, the Deep GP model that predicted the encoded combination of heading and distance for each seed sample is

shown in Figure 85. Following the same decoding sequence, the Deep GP model was split into two models for the heading and distance responses, as shown in Figure 86 and Figure 87, respectively. The 95% confidence interval for the heading uncertainty estimates was much narrower than the corresponding interval in the GPR implementation, indicating higher precision and lower uncertainty. Notably, the decoded distances contained far fewer errors in the Deep GP model than in the corresponding GPR model, indicating better performance.

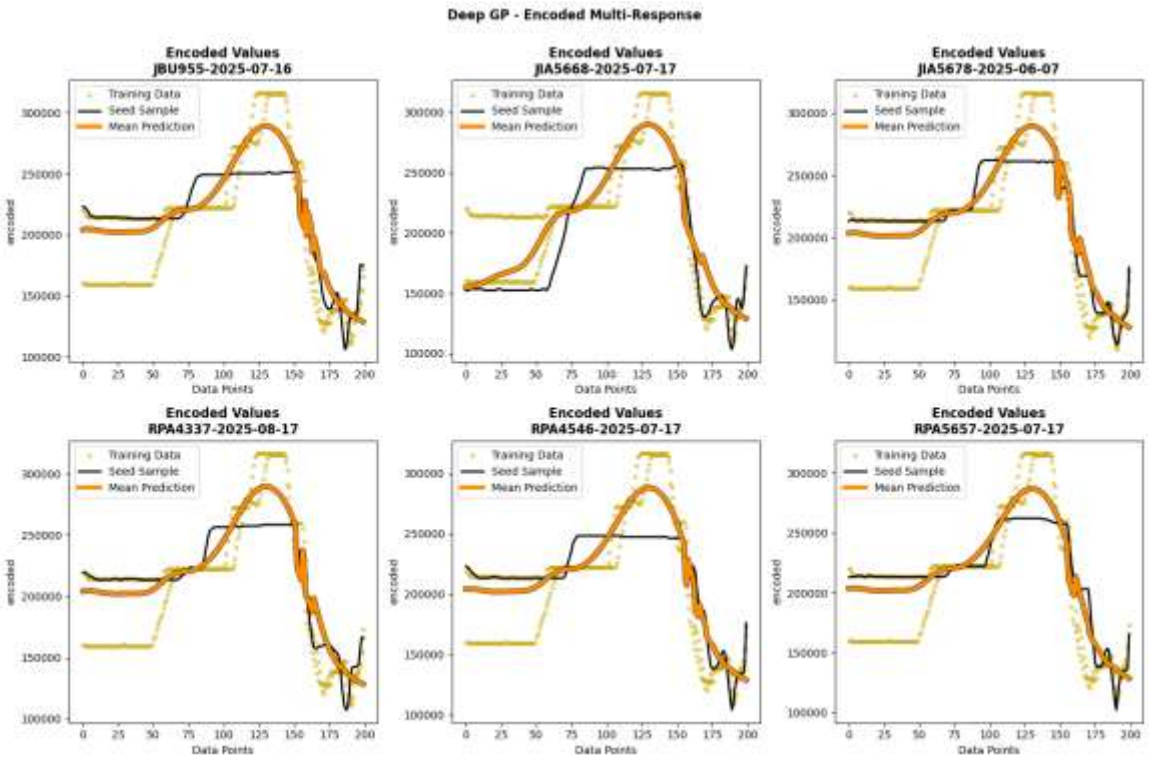


Figure 85 Deep GP Model for Encoded Response

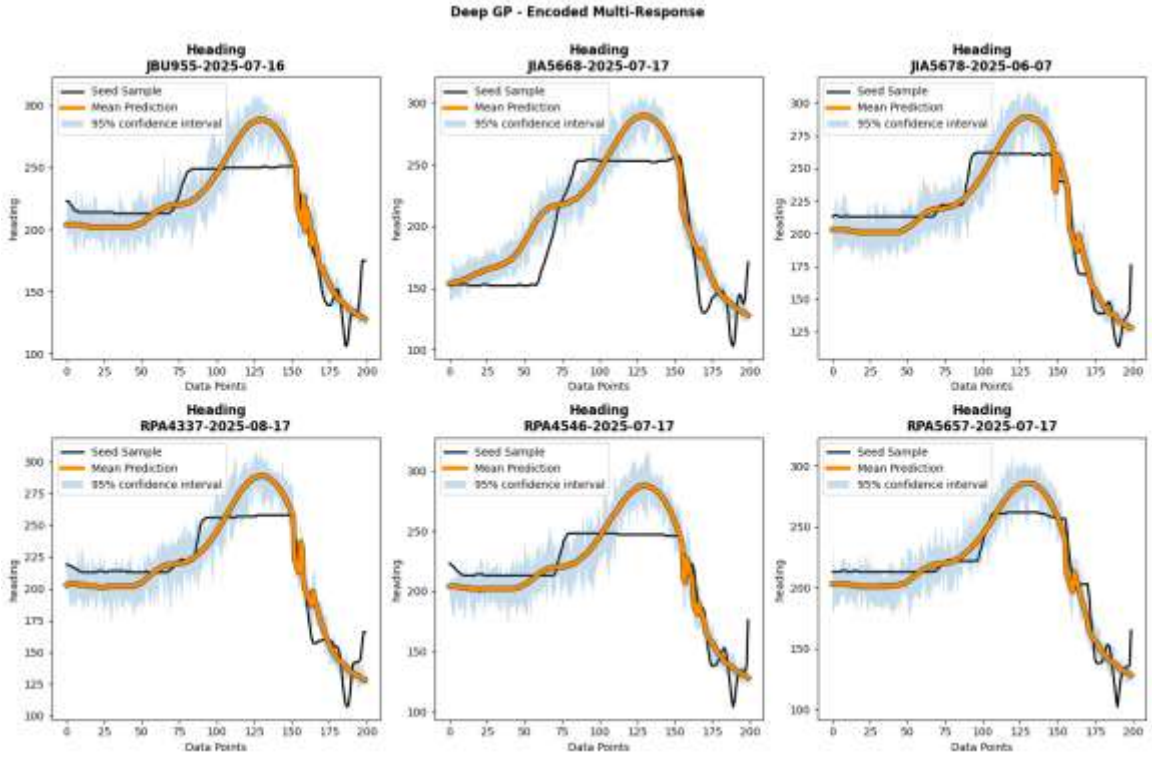


Figure 86 Deep GP Model for Heading

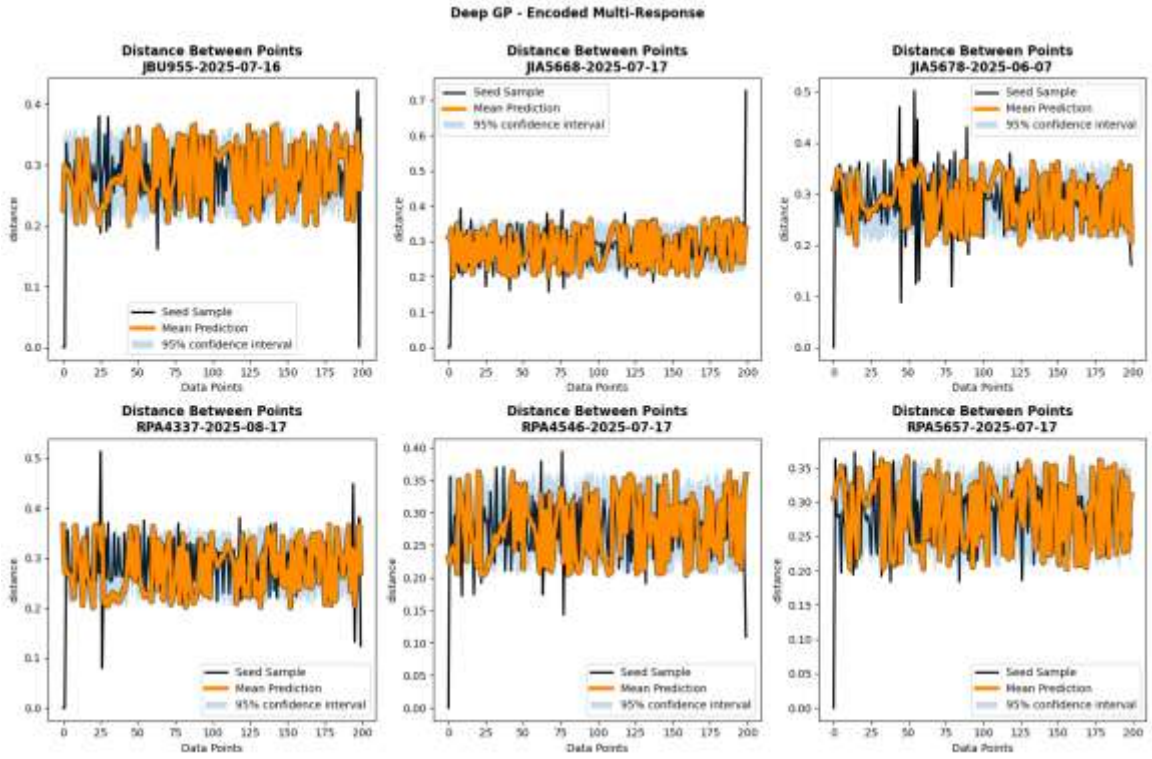


Figure 87 Deep GP Model for Distance

The resulting trajectories based on the backward reconstruction are shown in Figure 88. In contrast to GPR, the overall synthetic track length was similar to that of the actual seed tracks due to the model's accurate distance predictions. However, the incorrect shape of the predicted tracks was similar to that of the GPR model, indicating equivalent stacked propagated errors in headings.

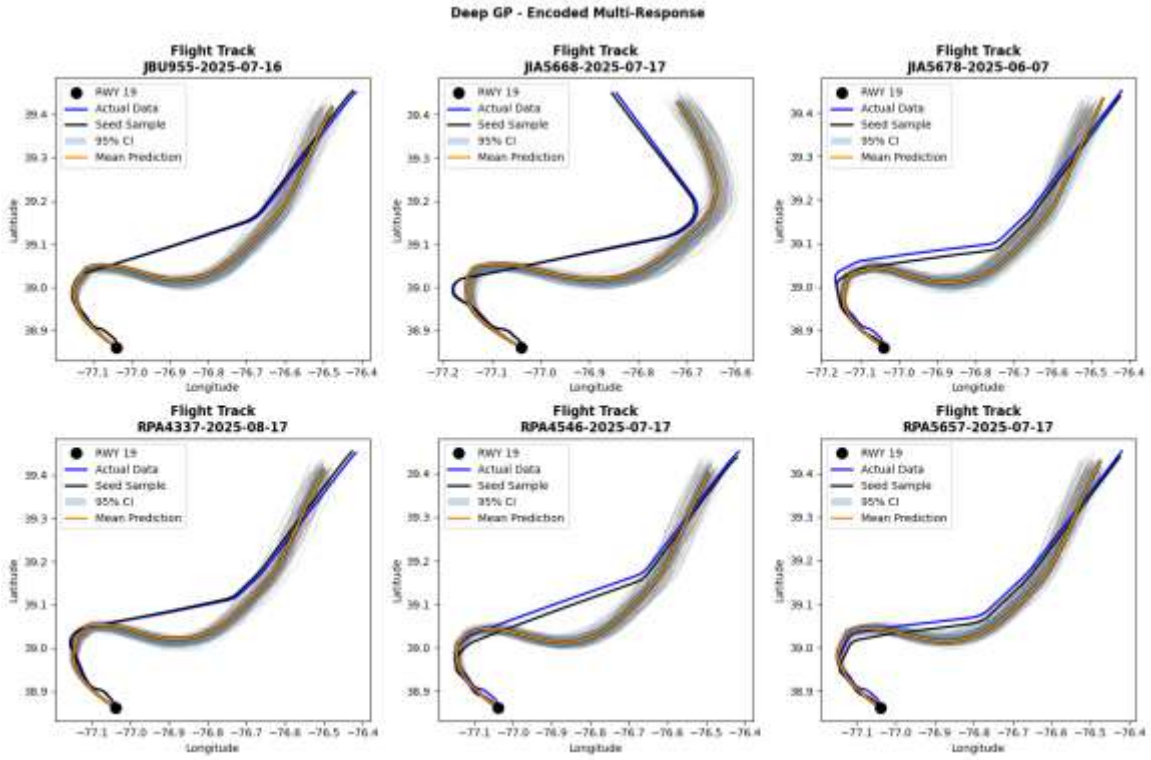


Figure 88 Synthetic Flight Tracks from Deep GP Model

The Deep GP model performance metrics are provided in Table 21. Quantitatively, RMSE values were low across all seed tracks, with an average error of 0.1173 on a normalized scale. The explained model variance, as captured by R2, was 0.8041 on average across all seeds.

Similar to GPR, the Deep GP model generated the best-performing synthetic results for the last seed, *RPA5657-2025-07-17*. The synthetic track data exhibited the lowest RMSE score (0.082) and the highest R^2 (0.8797). In comparison, the synthetic tracks for the same seed sample in the GPR model scored 0.1316 in RMSE and 0.6905 in R^2, indicating significant improvement in the Deep GP model performance.

The generated tracks followed the actual seeds much more closely than the tracks produced by the GPR model. In addition to good performance scores and an overall better shape of the tracks, batching was not required in the Deep GP model, reducing the computational burden and preserving variance in the results. However, due to the synthetic tracks limitation to capture the entire original flight path, these tracks were not selected as a candidate solution for collision risk modeling. Therefore, the Validation framework was not used.

Table 21 Deep GP Model Performance Metrics

Seed Track	RMSE	R^2
JBU955-2025-07-16	0.1297	0.7489
JIA5668-2025-07-17	0.1568	0.7678
JIA5678-2025-06-07	0.097	0.871
RPA4337-2025-08-17	0.1067	0.8255
RPA4546-2025-07-17	0.1314	0.7318
RPA5657-2025-07-17	0.082	0.8797
Average	**0.1173**	**0.8041**

5.2.5 GPR Model with Principal Components

This section provides an alternative approach to generating synthetic flight track data parameters simultaneously and in a dependent manner by encoding a response function with Principal Components (PCs). The resulting model encompassed all four track features (i.e., latitude, longitude, barometric altitude, and ground speed) in a single response.

5.2.5.1 Learning Framework

5.2.5.1.1 Response Encoding with Principal Components

Similar to the GPR implementation for Models 1A through 1D in Section 5.2.1, Principal Component Analysis (PCA) was used to encode the response based on a collection of Δ parameters along the tracks, (Δ latitude, Δ longitude, Δ barometric altitude, Δ ground speed). The method used PCA, as described in Section 4.2.1.2.3, to represent a 4-dimensional vector by separating its main features from less contributing details in the trajectory behavior. PCA was performed on the outputs in the entire model, including both training and test (i.e., seed) tracks, before the model learning.

Each contributing response variable was scaled so that each Δ parameter was on a comparable scale, resulting in a vector $Y_{scaled} = \begin{bmatrix} \Delta\,\text{latitude} \\ \Delta\,\text{longitude} \\ \Delta\,\text{altitude} \\ \Delta\,\text{speed} \end{bmatrix}$. The data standardization prevented parameters at a smaller scale from being excluded based on their contribution to variance.

The first PC, $PC1$, searched for directions in 4-dimensional space along which the flight trajectories varied the most. Mathematically, PCA found a unit vector, W, such that

the projection, $PC1 = WY_{scaled}$, captured the most variance in the response data. Residuals, R, captured the remaining variance in the response Δ parameters that $PC1$ could not explain, $R = Y_{scaled} - \hat{Y}_{scaled} = \begin{bmatrix} \mathrm{r}_{\Delta latitude} \\ \mathrm{r}_{\Delta longitude} \\ \mathrm{r}_{\Delta \mathrm{altitude}} \\ \mathrm{r}_{\Delta \mathrm{speed}} \end{bmatrix}$.

The resulting encoded response, Z, comprised a single PC describing the dominant trajectory pattern and four remaining values capturing deviations from the main pattern, $Z = \begin{bmatrix} PC1 \\ R_{\Delta latitude} \\ \mathrm{R}_{\Delta longitude} \\ R_{\Delta \mathrm{altitude}} \\ \mathrm{R}_{\Delta \mathrm{speed}} \end{bmatrix}$. Thus, the encoded response preserved all information needed to recover the original vector in the Generating framework. In other words, PCA compressed a 4-dimensional response into a principal score plus correction terms. During decoding in the Generating framework, the response reconstructed the exact original parameters by adding the corrections back into the outputs.

5.2.5.1.2 Predictors Allocation

The predictor set, $X \in \mathbb{R}^{50 \times 200 \times 9}$, comprised the following selection of the parameters in the specified order for each training track:

1. Position on the flight tracks
2. Latitude
3. Longitude
4. Heading
5. Course Change indication derived from the heading parameter and provided by a Boolean *True* or *False* value

6. Barometric Altitude
7. Descend indication derived from the altitude parameter and provided by a Boolean *True* or *False* value
8. Ground Speed
9. Deceleration indication derived from the speed parameter and provided by a Boolean *True* or *False* value

The order of predictions assumed that the flight path directionality contributed the most variance to the response, followed by altitude and speed.

5.2.5.1.3 Kernel Function Design

The composite covariance function comprised the Matérn, Rational Quadratic, and White kernels. After hyperparameter refinement, the resulting kernel function was as follows:

Equation 34 Custom Kernel Function for Model 5

$$k(x, x') = k_{Matern}(x, x', \sigma^2 = 0.429, \ell = 0.265, \upsilon = 1.5) + k_{Rational\ Quadratic}(x, x', \alpha = 95900, \ell = 2.13\ ) + \sigma_n^2 \delta(x, x');$$

$$\sigma_n^2 = 0.0712$$

In the Matérn kernel, a small signal variance σ of 0.429^2 indicated that the generated response had low amplitude, consistent with the delta scale in the encoded parameters. A small length scale ℓ of 0.265 indicated that the response changed rapidly across the input space, accurately capturing the overall trend in the trajectory. The smoothness υ of 1.5 resulted in the underlying function being once differentiable, with moderate roughness.

The Rational Quadratic kernel captured local irregularities, varying smoothness, and multi-scale behavior in the response variable. A relatively small length scale ℓ of 2.13 meant the function changed rapidly over short distances. A large alpha $\alpha \to \infty$ implied that the kernel converged into a standard squared exponential kernel with a single, uniform length scale, supplementing the Matérn kernel's assumptions about the response behavior.

The obtained log marginal likelihood was a large negative value of -17960.181939989518, indicating complex patterns and challenges in the predicted response.

5.2.5.2 Generating Framework

A single trained GPR model that predicts the parameters of flight tracks, encoded via PCA, is shown in Figure 89.

The response decoding sequence was as follows. First, the PCA process was reversed by extracting a single PC and the remaining residuals from the encoded response Z. Second, the PC was reconstructed to approximate the trajectory, $\hat{Y}_{scaled}$. Next, the residuals were added to the reconstructed PC, $Y_{scaled} = \hat{Y}_{scaled} + R$. Lastly, the scaled response vector, Y_{scaled}, was transformed back into physical units, yielding the original vector $\begin{bmatrix} \Delta \text{ latitude} \\ \Delta \text{ longitude} \\ \Delta \text{ altitude} \\ \Delta \text{ speed} \end{bmatrix}$.

Next, the delta values in the obtained vector of predicted delta parameters were reconstructed to the original physical scale in accordance with the procedure outlined in Section 5.2.1.2. The reconstituted outputs for flight track, altitude, and speed profiles, all

generated dependently, are shown in Figures 89-91, respectively. In the plots, black lines represent actual seed parameters. Red lines provide posterior means for the associated outputs. The shaded gray regions show 95% confidence intervals around the posterior means. A collection of gray lines describes synthetic samples in the corresponding outputs. The synthetic flight track data parameters were extracted from batches with 10 posterior samples to facilitate more realistic patterns in the predicted trajectory parameters. The synthetic flight track data parameters generated from batches with other number of posterior samples (batch size = 5 and batch size = 25) are provided in Appendix D – Additional Synthetic Results from Model 5.

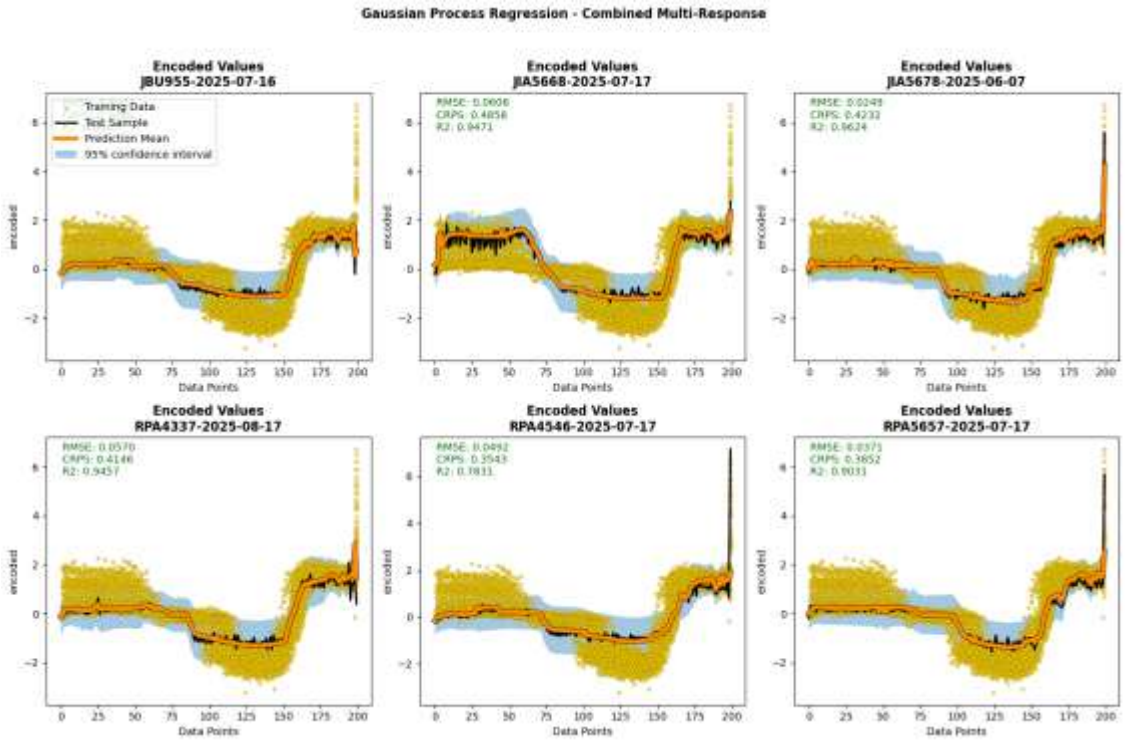


Figure 89 GPR Model with Principal Components in Response

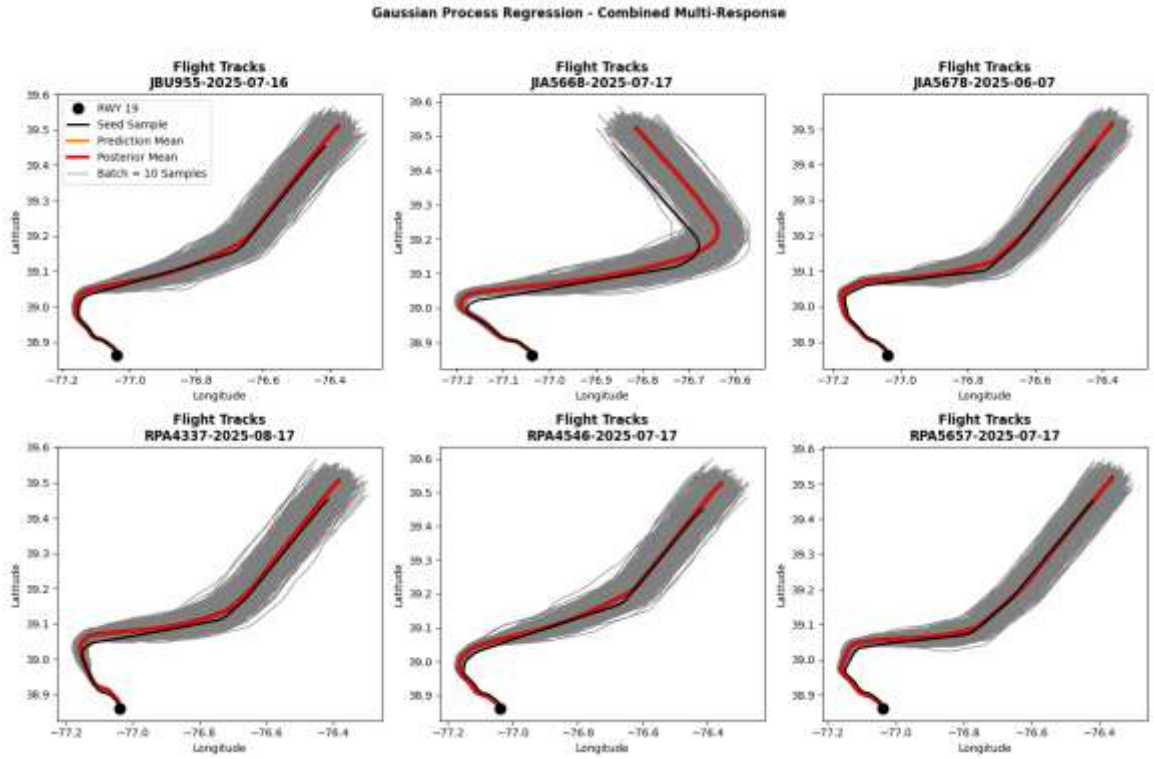


Figure 90 GPR for Flight Tracks in Model 5 (Batch = 10)

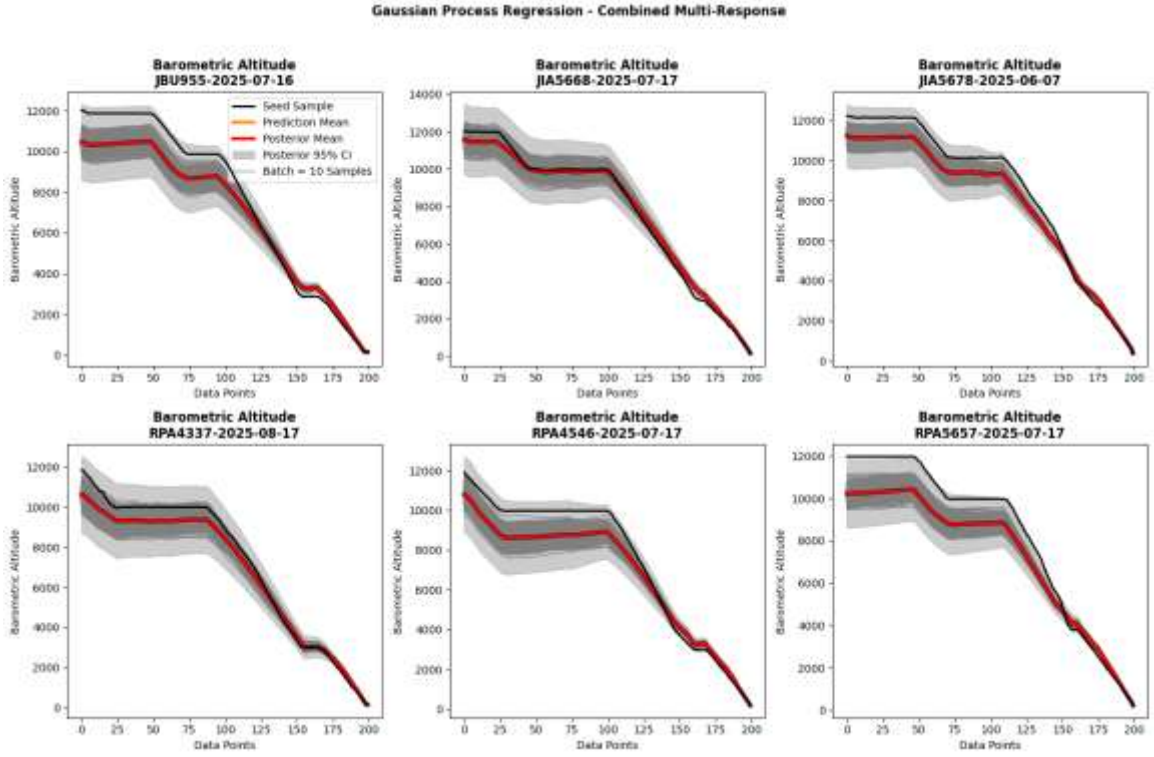


Figure 91 GPR for Barometric Altitude in Model 5 (Batch = 10)

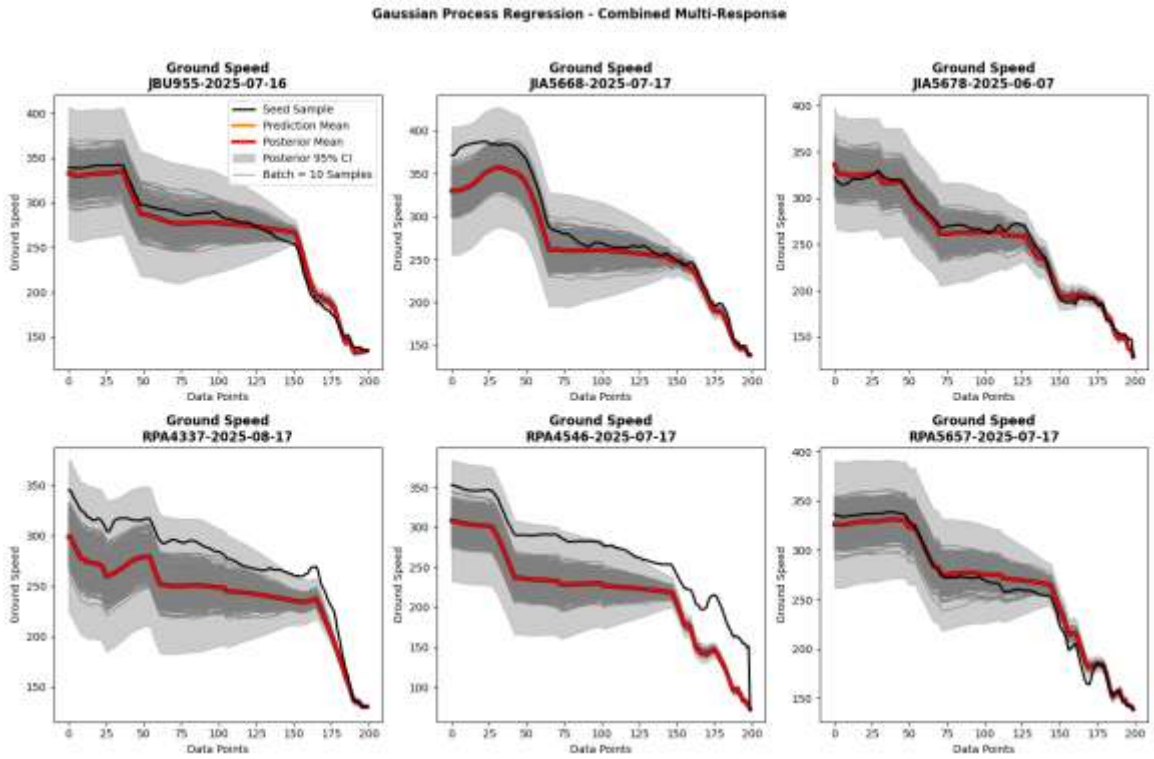


Figure 92 GPR for Ground Speed in Model 5 (Batch = 10)

The model performance scores are provided in Table 22. Each seed model yielded low RMSE scores, with an average of 0.0465 on the normalized scale across all seed samples. Low RMSE scores demonstrated the GPR model's ability to predict the data accurately. Low CRPS scores across all seeds indicated the model's good predictive performance (average score of 0.0402). In contrast, high R^2 scores, with an average of 0.9203, verified that the model explained the underlying variance in the response data well.

Based on the model verification metrics and preliminary analysis of the synthetic data, the synthetic samples across all six seeds were suitable for evaluation in the

Validation framework. Nevertheless, the results for the first seed track, JBU955-2025-07-16, were selected for further analysis, as it was done in Section 5.2.1.3 for GPR model with a single response, facilitating a fair comparison between the performance of synthetic flight tracks.

Table 22 Model Performance Metrics for GPR with PCs in Response

Seed Track	RMSE	CRPS	R^2
JBU955-2025-07-16	0.0503	0.0352	0.9692
JIA5668-2025-07-17	0.0606	0.0484	0.9501
JIA5678-2025-06-07	0.0249	0.0420	0.9649
RPA4337-2025-08-17	0.057	0.0417	0.9471
RPA4546-2025-07-17	0.0492	0.0355	0.7852
RPA5657-2025-07-17	0.0371	0.0383	0.9053
Average	0.0465	0.0402	0.9203

5.2.5.3 Validation Framework

The GPR model with principal components in a single response generated 1 million synthetic samples for each flight track parameter. The synthetic values in the corresponding track parameter represented a mean of 10 posterior samples combined in a batch. If the synthetic solution incorporates realistic trajectory behavior, an unlimited number of synthetic samples can be extracted from the model. To facilitate graphical processing, a selection of 10,000 synthetic flight tracks, along with associated barometric ltitude and ground speed profiles, is shown in Figure 93.

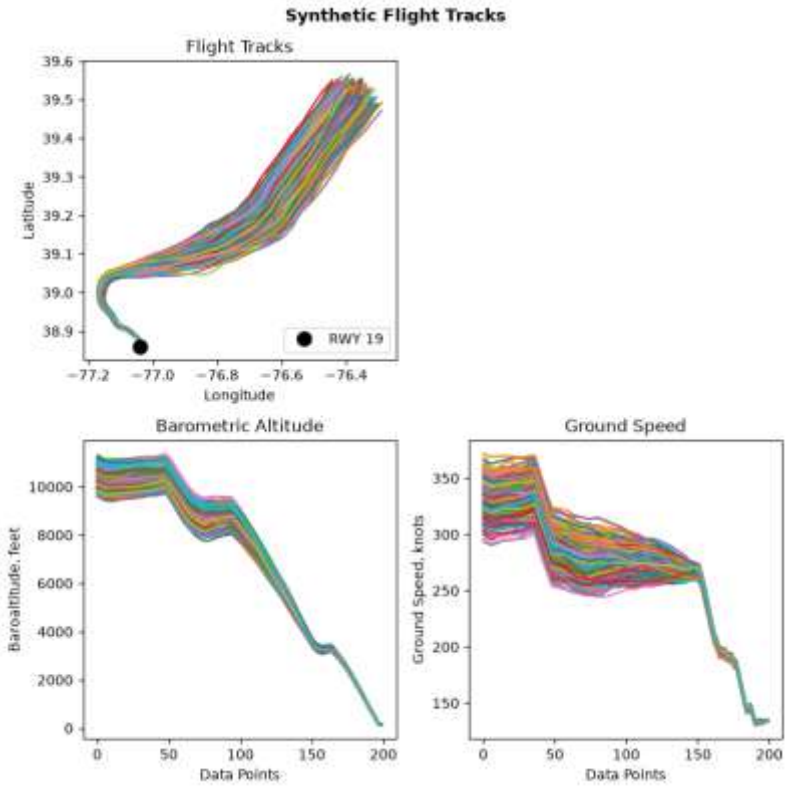


Figure 93 Synthetic Flight Tracks from Model 5

The Along-Track Distance was calculated across all tracks in both synthetic and historical datasets (Figure 94). The distances for synthetic tracks, shown in blue, were normally distributed with a mean of approximately 61 nm. In contrast, the historical figures, shown in gray, were mostly concentrated farther to the left, between 56 and 60 nm. In addition, the synthetic flight tracks from Model 1 had shorter distances on average, with a mean of 55 nm (Figure 72). A narrower distribution of samples from Model 5 can be explained by their concentration around a single representative seed track and a much larger sample size (1 million synthetic samples against 215 historical tracks). Longer distances in the synthetic tracks resulted from rougher trajectory patterns, which can be attributed to a smaller number of posterior samples per batch. By increasing the batch size, as shown in Figure 114, synthetic trajectories become smoother, yielding shorter distances

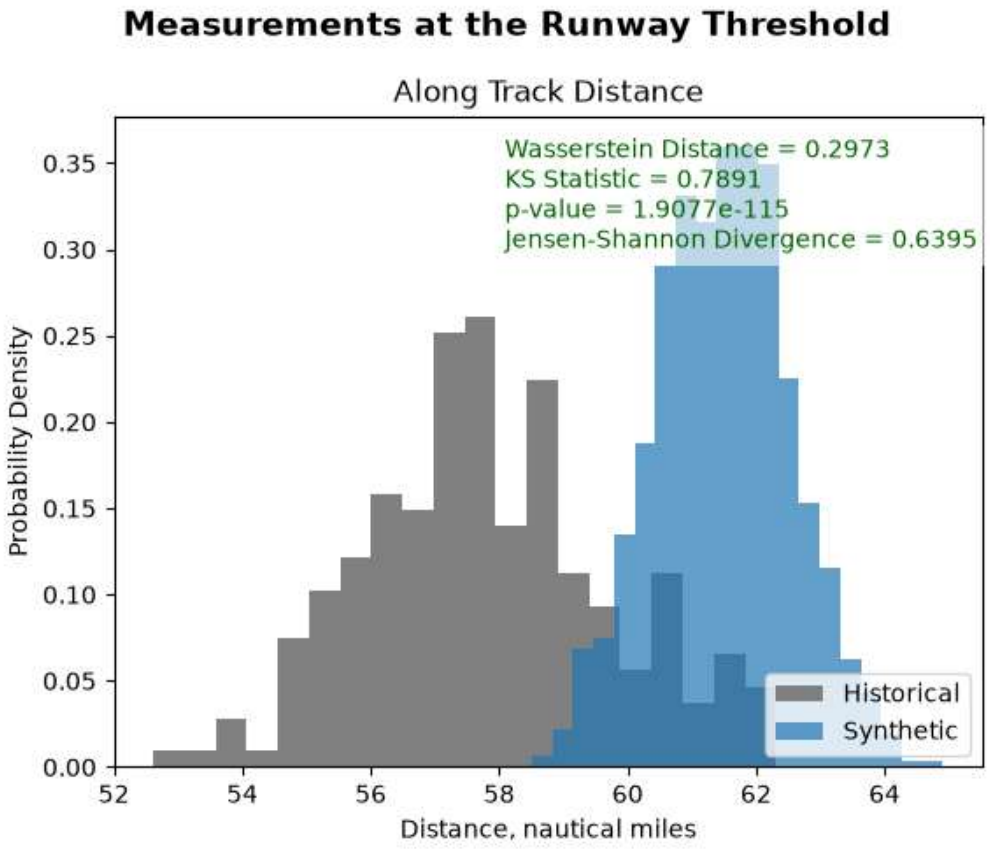


Figure 94 Along Track Distance Distribution Comparison in Model 5 (Batch = 10)

Despite the narrower concentration of samples and slightly longer distances, the synthetic flight tracks from Model 5 exhibited realistic behavior due to the existing overlap between historical and synthetic distributions.

The measurements for lateral, vertical, and speed dispersion at 3, 6, and 9 nm from the runway threshold summarized corresponding differences between synthetic and historical tracks (Figure 95).

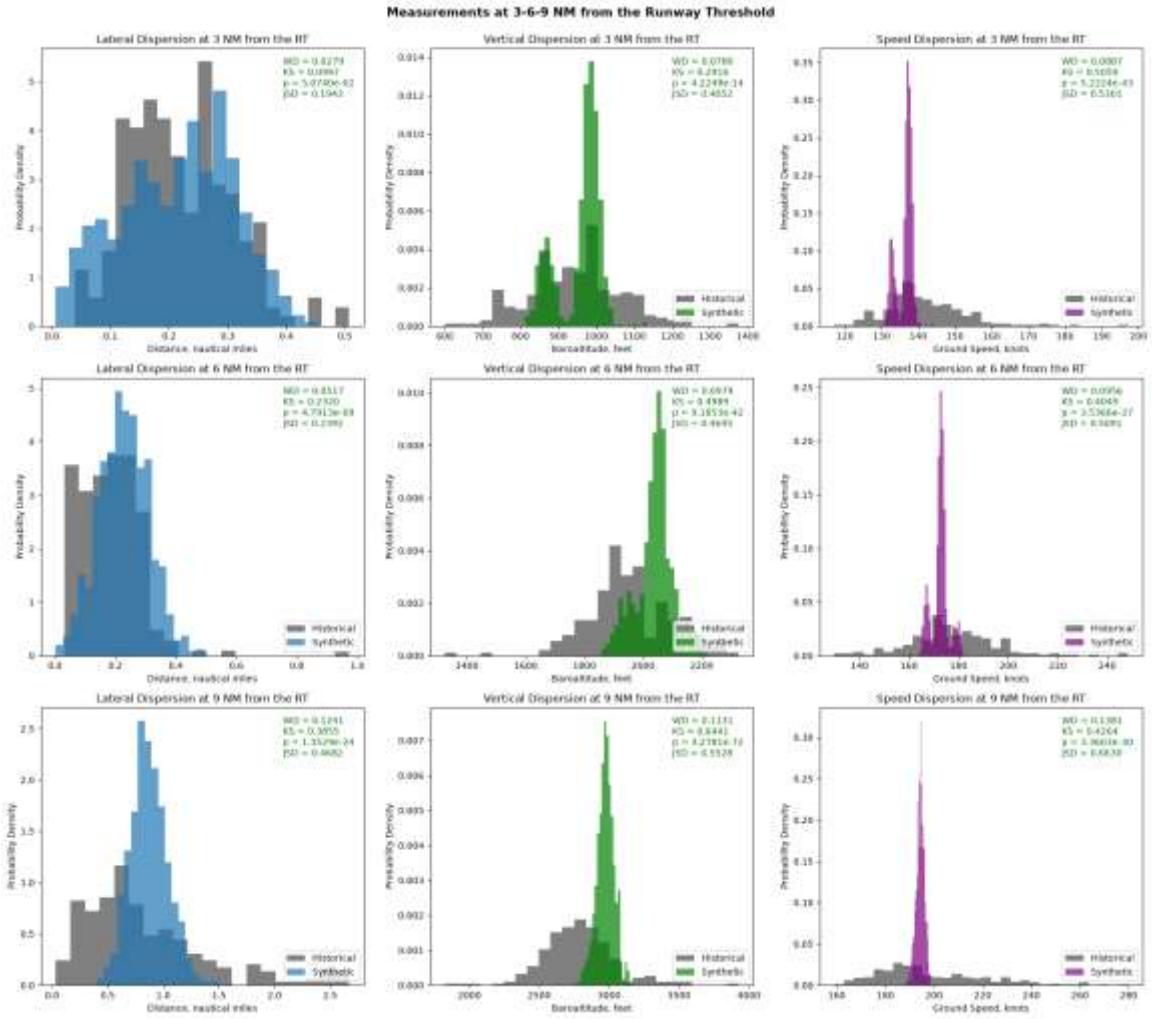


Figure 95 Distributions for Measurements at 3, 6, and 9 NM in Model 5 (Batch = 10)

Historical and Synthetic data distributions for lateral dispersion across all measured locations exhibited significant statistical similarities. Specifically, the measurements of lateral dispersion at 3 nm from the runway threshold yielded scores of 0.0279, 0.0997, and 0.1943 for the Wasserstein distance, KS Statistics, and JSD, respectively, indicating that the compared distributions were almost identical. The lateral dispersion computed at 6 nm from the runway threshold yielded a Wasserstein distance

of 0.0517, indicating strong resemblance between the historical and synthetic distributions, albeit with slightly different shapes (KS Statistics = 0.232 and JSD = 0.2392). The overlapping distributions at 9 nm demonstrated that the synthetic tracks resembled the behavior of the historical trajectories.

Similar to the results from Model 1, synthetic altitudes and speeds across all three recorded locations yielded narrower distributions than the historical profiles. The Wasserstein distances were <=0.1 in both vertical and speed dispersions across all measurements, indicating moderate statistical similarities. Overlaps in synthetic and historical figures prove that the generated parameters exhibited realistic features.

Overall, the synthetic tracks' shape accurately reflected flight characteristics at the final approach fix, suggesting high quality of the synthetic data. Furthermore, the waypoints from the RNAV (GPS) Y approach procedure for Runway 19 at DCA (Figure 49) were properly captured by the synthetic flight tracks (Figure 96).

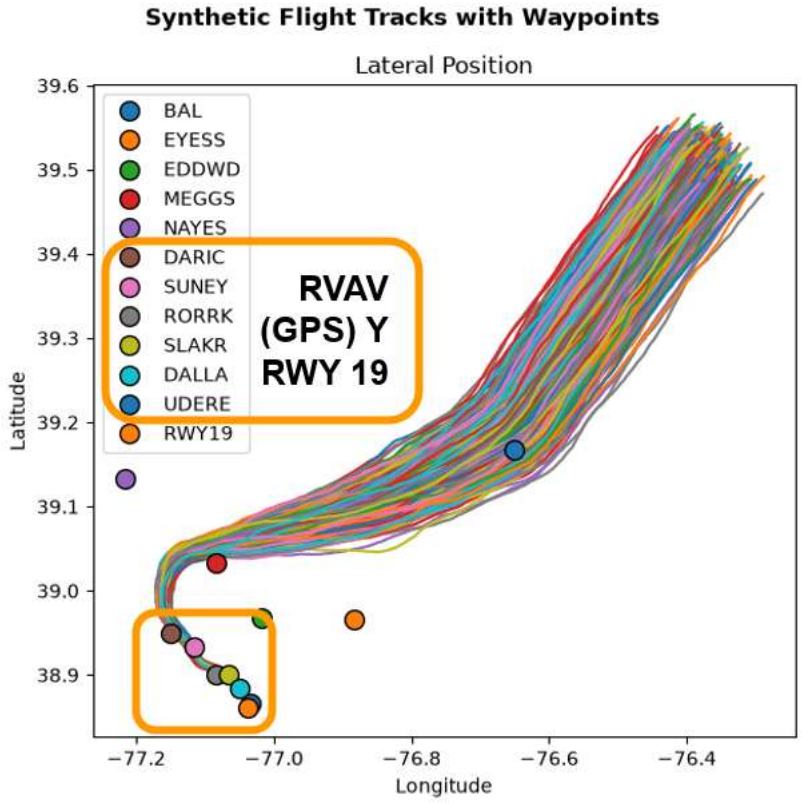


Figure 96 Synthetic Flight Tracks from Model 5 with Waypoints

In Figure 97, the numbers for level changes in the synthetic vertical profile fell within the range provided by the recorded aircraft trajectories. However, the number of course changes in synthetic tracks was moderately higher than in the representative seed tracks due to rougher trajectory patterns. As shown in Figure 116, increasing the batch size makes synthetic trajectories smoother, yielding fewer course changes. In contrast, the number of speed changes was lower in the synthetic data. As shown in Figure 110, decreasing the batch size makes synthetic speeds less smooth, thereby incorporating fewer constant-speed segments.

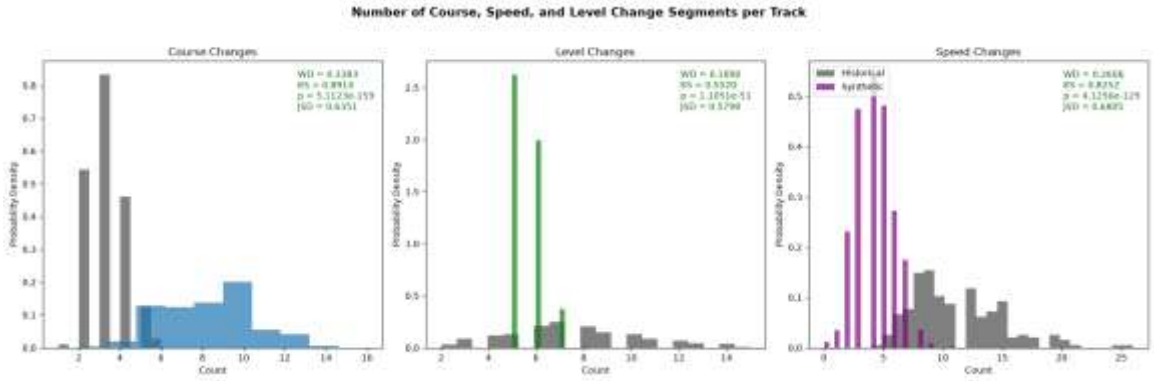


Figure 97 Distributions for Individual Track Characteristics in Model 5 (Batch = 10)

5.2.6 Summary

5.2.6.1 Synthetic Data Quality

Overall, five different GPR models and their variants, such as Deep GP, were implemented with distinct underlying functions, predictor set selections, and response structures. Only two of the five models were capable of generating reasonable 4-dimensional synthetic flight trajectories, comprising a pair of coordinates, altitude, and speed. First, Model 1, encompassing four independent GPR sub-models, each with a single response, was used to predict changes in flight-track data parameters separately. Second, Model 5, characterized by principal components in the encoded multi-response of the GPR function, generated the same trajectory parameters simultaneously and in a dependent manner.

The resulting synthetic solutions from both Models were evaluated for realism by examining the variance between the historical and generated data distributions. The measurements of lateral, vertical, and speed deviations at 3, 6, and 9 nm from the runway threshold were collected in both historical and synthetic flight tracks. The along-track distance was computed for each synthetic track and subsequently compared with historical values. The primary distribution comparison metrics included the Wasserstein Distance, the Kolmogorov–Smirnov (KS) Statistics, and the Jensen–Shannon Divergence (JSD). Lower scores in each metric indicate greater similarity between synthetic and historical data, suggesting a higher level of quality and realism in the synthetic data.

The data comparison between synthetic results from Model 1 and Model 5 is shown in Table 23.

Table 23 Synthetic Data Quality in Model 1 vs Model 5

Model / Parameter	Model 1	Model 5	Model 5 % Improvement	Model 1	Model 5	Model 5 % Improvement	Model 1	Model 5	Model 5 % Improvement
Lateral Dispersion	**3 nm from the RT**			**6 nm from the RT**			**9 nm from the RT**		
Wasserstein Distance	0.6979	0.0279	96%	0.6854	0.0517	92%	0.5896	0.1241	79%
KS Statistic	1	0.0997	90%	0.9995	0.232	77%	0.9601	0.3855	60%
JSD	nan	0.1943	nan	0.8238	0.2392	71%	0.7775	0.4682	40%
Vertical Dispersion	**3 nm from the RT**			**6 nm from the RT**			**9 nm from the RT**		
Wasserstein Distance	0.3147	0.078	75%	0.1649	0.0979	41%	0.022	0.1131	-414%
KS Statistic	0.9297	0.2916	69%	0.6279	0.4989	21%	0.095	0.6441	-578%
JSD	0.7458	0.4052	46%	0.5232	0.4645	11%	0.1671	0.5528	-231%
Speed Dispersion	**3 nm from the RT**			**6 nm from the RT**			**9 nm from the RT**		
Wasserstein Distance	0.1233	0.0887	28%	0.1276	0.0956	25%	0.1162	0.1381	-19%
KS Statistic	0.6545	0.5059	23%	0.5846	0.4049	31%	0.3663	0.4264	-16%
JSD	0.5846	0.5301	9%	0.5301	0.5091	4%	0.5065	0.663	-31%
Along Track Distance	**Runway Threshold**								
Wasserstein Distance	0.2596	0.2973	-15%						
KS Statistic	0.7267	0.7891	-9%						
JSD	0.6017	0.6395	-6%						

Model 5 consistently outperformed Model 1 across all lateral dispersion measurements. At 3 nm from the runway threshold, improvements were substantial, reaching 96% for Wasserstein Distance and 90% for the KS Statistic. Model 5 performance remained strong at 6 nm, with reductions of 92%, 77%, and 71% in Wasserstein Distance, KS Statistic, and JSD, respectively. Although the percentage decreased with distance, Model 5 achieved notable gains at 9 nm, reducing the three metrics by 79%, 60%, and 40%. Overall, the synthetic flight tracks in Model 5 demonstrated a markedly better representation of lateral dispersion than in Model 1.

In terms of vertical dispersion, Model 5's performance declined with increasing distance from the runway threshold. At 3 nm, substantial improvements were observed, with reductions of 75%, 69%, and 46% in Wasserstein Distance, KS Statistic, and JSD, respectively. At 6 nm, the gains became more modest, ranging from 11% to 41%. However, at 9 nm, Model 5 performed significantly worse than Model 1, as indicated by the large negative percent improvements (−414%, −578%, and −231%). This suggests that while Model 5 captured vertical dispersion well near the runway, its accuracy deteriorated farther away.

For speed dispersion, Model 5 improvements over Model 1's performance were generally moderate. At 3 nm, Model 5 reduced Wasserstein Distance by 28%, KS Statistic by 23%, and JSD by 9%. Similar gains were observed at 6 nm, with improvements ranging from 4% to 31%. At 9 nm, however, performance declines, with all three metrics showing negative values (−19%, −16%, and −31%). Thus, Model 5

provides a better representation of speed dispersion near the runway but loses accuracy at greater distances.

Unlike the dispersion metrics, analytically, Model 5 performs slightly worse than Model 1 at the runway threshold. Wasserstein Distance increases by 15%, the KS Statistic by 9%, and JSD by 6%, indicating that Model 1 more closely matches the reference along-track distribution.

Overall, Model 5 delivered the greatest improvements in lateral dispersion, consistently achieving large reductions in all metrics across the distances examined. For vertical and speed dispersion, improvements were greatest near the runway threshold but diminished with distance, eventually becoming negative at 9 nm. In contrast, along-track distance showed a slight performance degradation, suggesting that the enhancements introduced in Model 5 primarily benefit the lateral characteristics of the trajectory distributions.

5.2.6.2 GP Model Scoring

A scoring card generated for all five GP models implemented in Sections 5.2.1 through 5.2.5 is shown in Table 24. The scoring criteria derived from the following considerations:

- Model complexity in terms of algorithmic and implementation challenges
- Model computational complexity in terms of computer resources usage
- Synthetic parameters availability for collision risk modeling, indicating that four (i.e., latitude, longitude, altitude, and speed) parameters in synthetic data were generated in models

- Dependencies between synthetic parameters
- Generated sample size in the synthetic data
- Quality of the synthetic data (summarized in the previous Section)

The scoring scale represented the following:

- Green – Good, score = 5
- Yellow – Average, score = 4
- Orange – Below Average, score = 3
- Red – Bad, score = 2
- Gray – Not Applicable (N/A), score = 0

Model 5 for GPR with principal components in the response scored the highest (24). The second-best implementation was Model 1 for independent GPR models with single responses (22). Next, Model 3 for Deep GP with encoded multi-response scored 18 points. Lastly, Model 3 for GPR with encoded multi-response and Model 2 for GPR with a single response predicting the next nth position scored 15 and 14 points, respectively.

In summary, the synthetic flight tracks, along with their associated vertical and speed profiles, from Model 5 can be incorporated in collision risk modeling to study effects of a new arrival procedure represented by seed tracks.

Table 24 Scoring Card for GP Models

	Model Implementation Complexity	Model Computational Complexity	Synthetic Parameters Availability	Dependencies between Synthetic Parameters	Sample Size in Synthetic Data	Synthetic Data Quality	Score
Model 1	• Multiple independent GPR models • *Identical model structure* • Response data transformation into Δ values in the Learning Framework • Response data reconstruction in the Generating Framework • Predictor and response data scaling and normalization in the Learning Framework • Predictor and response data inverse scaling in the Generating framework	Moderate computation burden due to repeated identical matrix calculations by multiple models	All four synthetic parameters are available (latitude, longitude, altitude, and speed)	No, the synthetic parameters generated from independent GPR models	Adjustable; batching required, reducing the sample size and increasing computational burden		22
Model 2	• Multiple independent GPR models • *Identical model structure* • *Minimum response data manipulation* • Predictor and response data scaling and normalization in the Learning Framework • Predictor and response data inverse scaling in the Generating framework	Moderate computation burden due to repeated identical matrix calculations by multiple models	Only latitude and longitude are available, but altitude and speed can be easily added	No, the synthetic parameters generated from independent GPR models			14
Model 3	• *Single model* • Complex response encoding and decoding sequences • Complex response extraction and computation • Predictor and response data scaling and normalization in the Learning Framework • Predictor and response data inverse scaling in the Generating framework • Complex composite kernel structure	• *Single model operation* • Large batch size is required to combine synthetic samples	Only latitude and longitude are available. Encoding additional parameters can be unreasonable given the model complexity	Dependencies only between latitude and longitude	Batch size of 100 is required, reducing the sample size and increasing computational burden		15

	Model Implementation Complexity	Model Computational Complexity	Synthetic Parameters Availability	Dependencies between Synthetic Parameters	Sample Size in Synthetic Data	Synthetic Data Quality	Score
Model 4	• *Single model* • Complex response encoding and decoding sequences • Complex response extraction and computation • Predictor and response data scaling and normalization in the Learning Framework • Predictor and response data inverse scaling in the Generating framework • Complex composite kernel structure in observation layer • Multiple kernel functions in hidden layers	• *Single model operation* • *No batching required*	Only latitude and longitude are available. Encoding additional parameters can be unreasonable given the model complexity	Dependencies only between latitude and longitude	Adjustable; *batching not required*		18
Model 5	• *Single model* • Response data encoding and decoding with PCA • Predictor and response data scaling and normalization in the Learning Framework • Predictor and response data inverse scaling in the Generating framework • Complex composite kernel structure • Large predictors set	Moderate computation burden due to complex response transformation, multi-dimensional computations, and batch size	All four synthetic parameters are available (latitude, longitude, altitude, and speed)	Dependencies between synthetic parameters	Batch size of 10 is required, reducing the sample size and increasing computational burden		24

The text in *Italics* indicates positive factors in the Model Implementation Complexity, Model Computational Complexity, and Sample Size criteria.

CHAPTER SIX: CONCLUSIONS

This Chapter summarizes the results, future work, and the next steps in this research.

This dissertation proposed a modular framework for training, generating, and validating synthetic flight tracks for collision risk modeling using GP-based methods. A total of five different GP models were developed using the framework. The resulting GP models produced multi-dimensional flight tracks based on historical arrival procedures at DCA. Each GP implementation exhibited distinct modeled behaviors and provided important insights into the effects of various underlying GP functions, predictor selections, and response structures.

6.1 Research Objective

The overall research objective was to generate **realistic** flight trajectories suitable **for collision risk analysis** of a **new procedure** within a **reasonable time** frame. The following description provides a breakdown of the objective setting.

New procedure: The entirely new procedure at DCA could be obtained from a flight trajectory flown by pilots using aircraft simulators. Given the absence of such a trajectory at the time of this research, a representative procedure was selected from the historical dataset. Although the use of the representative procedure did not allow for a collision risk analysis of an actual change to the NAS, it provided certain benefits. First,

the representative procedure did not participate in training; thus, it did not influence the modeled predictions. Second, the representative procedure originated from the remaining historical data distribution. This historical distribution was used for variance comparison against the generated data distributions. The distribution comparison enabled quantitative assessment of the resulting synthetic flight tracks.

Realistic feasible flight trajectories: Two out of the five developed GP models generated reasonable synthetic flight tracks, confirmed by metrics in Table 23 and Table 24. The quality and fidelity of the resulting synthetic solitons were confirmed by comparing historical and synthetic distributions across various data parameters. Although both models, specifically Model 1 and Model 5, generated multi-dimensional synthetic flight tracks, Model 5 marginally outperformed Model 1. Therefore, the synthetic flight trajectories from Model 5 could be incorporated into collision risk modeling as a realistic representation of the new procedure development.

Reasonable time: Due to the data-efficient property of Gaussian Processes, only a subset of the available historical data was used for model training. Specifically, the same 50 historical flight tracks, with 200 data points per track, were used by each model, although with different predictor set sizes. The limited number of training tracks enabled efficient use of computational resources during model learning. In addition to the training, synthetic flight trajectories were generated from samples in the posterior distribution of the corresponding GP model. Although synthetic trajectory sampling did not require extensive use of memory or operational resources, response reconstruction negatively affected processing time. However, millions of synthetic flight tracks can be

generated in under 15 minutes using a standard personal computer. Otherwise, generating billions of synthetic flight trajectories requires supercomputer power to facilitate sampling, which was still faster than in simulation models.

Suitability for CRA: In addition to the high quality of the synthetic solution, the generated flight track's suitability criteria for collision risk modeling include a large sample size and preserved variance in the resulting trajectories. First, the GP-based methods' ability to generate billions of new flight trajectories made them a strong option for further consideration in collision risk modeling. Second, GP models generated predictions for each flight-track parameter along with their uncertainty estimates. The confidence interval or the standard deviation range enabled control of the degree of uncertainty in the synthetic solution. These mechanisms for variance control, in combination with evaluation of the resulting solutions, enabled the generation of very diverse synthetic flight tracks, making them a viable candidate for collision risk modeling and analysis.

In conclusion, the overall research objective was met.

6.2 Research Questions and Answers

The following outlines the research questions in Chapters 2 and 3 with provided answers.

Research Question #1. How to obtain a large sample size of flight trajectories for a new procedure in the NAS, in a reasonable time?

Answer to Research Question #1: Flight trajectory models in CRM comprise historical, simulated, blended, or synthetic flight tracks. First, historical flight tracks

represent a limited sample of existing flight trajectories and do not incorporate proposed changes to the NAS, making them unsuitable as a sole source of CRM's inputs. Second, simulated flight tracks enable new development considerations; however, they are subject to inherent modeling assumptions and operational restrictions. Moreover, simulation at a scale of billions of new tracks may take a significant amount of time to complete. Next, a blended mix of historical and simulated flight tracks mitigates some restrictive modeling assumptions in the resulting dataset. However, it still faces the same challenges of substantial computational resources and time-consuming operations. Finally, the synthetic flight tracks, which capture properties of real aircraft trajectories, enable unlimited sample generation for a new procedure required for comprehensive collision risk modeling.

Research Question #2. How to obtain realistic synthetic flight trajectories using Gaussian Process-based methods?

Answer to Research Question #2: This dissertation presented results for five GP implementations with different response structures. The resulting response types can be categorized into three categories: single response in Models 1 and 2, encoded multi-response in Models 3 and 4, and response with principal components in Model 5. Several independent GPR models with single responses were required to generate synthetic parameters for a multi-dimensional flight track. As a result, the synthetic solution lacked dependencies among generated outputs. Alternatively, generating flight track parameters in a dependent manner required transformation of the response data to combine multiple outputs into a single feature. The complexities introduced into the response data structure

tended to obscure the GPR model, leading to more steps in the algorithm execution. However, the resulting model met the required threshold of realism in the synthetic solution by incorporating dependencies between outputs, while adhering to GP mathematical principles by producing a single encoded multi-response function.

Research Question #3. How to preserve variance and diversity in the synthetic flight tracks while limiting aerodynamically unreasonable samples at the same time?

Answer to Research Question #3: A combination of activities performed within both the Generating and Validation frameworks of the proposed modular architecture enabled the generation of aerodynamically valid flight trackers while preserving their variance and diversity. First, this research explored the effects of different batch sizes used in the results generation process. Second, the synthetic results were evaluated for realism within the Validation framework by examining variance in the resulting distributions for the corresponding batch sizes. Lastly, the synthetic solution with the best balance between variance and realism in track data parameters was presented.

6.3 Future Work and the Next Steps

Future work in this research can be extended in several ways. First, the generated synthetic flight tracks can be incorporated into collision risk modeling, thereby completing the modeling process. The end-to-end process encompasses historical data collection, followed by the generation of synthetic data, and the incorporation of the flight trajectory model in CRM. Second, a separate set of synthetic tracks can be generated from other historical procedures, such as departure or helicopter trajectories, enabling fusion of flight trajectories. The resulting fused dataset presents an opportunity

to conduct a risk analysis of mixed procedures near the terminal area, where collision probability is highest. Third, additional qualitative and analytical validation criteria can be derived to evaluate the realism of synthetic solutions. Lastly, a continuous search for the next-best GP implementation and its parameters could yield better ways to generate synthetic data faster and more reliably.

APPENDIX A – PYTHON SCRIPTS TO GENERATE KERNEL FUNCTIONS

RBF Kernel:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF

title = "Samples from Prior Distribution with RBF Kernel"
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10,8), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

# Generate sample data points
X = np.linspace(0, 10, 100).reshape(-1, 1)

##########################################################################
chart = '(a)'
# Define the RBF kernel with a specific length scale
length_scale = 1.0
variance = 1.0
kernel = variance * RBF(length_scale=length_scale)

# Create a Gaussian Process model with the RBF kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

# Plot the samples from the prior distribution
for i in range(samples.shape[1]):
    ax[0,0].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[0,0].set_title(f'{chart}\nsignal variance = {variance}\nlength_scale = {length_scale}')
ax[0,0].set_xlabel('Input Value')
ax[0,0].set_ylabel('Kernel Value')

##########################################################################
chart = '(b)'
# Define the RBF kernel with a specific length scale
length_scale = 1.0
variance = 2.0
kernel = variance * RBF(length_scale=length_scale)

# Create a Gaussian Process model with the RBF kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

# Plot the samples from the prior distribution
for i in range(samples.shape[1]):
    ax[0,1].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[0,1].set_title(f'{chart}\nsignal variance = {variance}\nlength_scale = {length_scale}')
```

```
ax[0,1].set_xlabel('Input Value')
ax[0,1].set_ylabel('y')

#############################################################################
chart = '(c)'
# Define the RBF kernel with a specific length scale
length_scale = 5.0
variance = 1.0
kernel = variance * RBF(length_scale=length_scale)

# Create a Gaussian Process model with the RBF kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

# Plot the samples from the prior distribution
for i in range(samples.shape[1]):
    ax[1,0].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[1,0].set_title(f'{chart}\nsignal variance = {variance}\nlength_scale = {length_scale}')
ax[1,0].set_xlabel('Input Value')
ax[1,0].set_ylabel('Kernel Value')

#############################################################################
chart = '(d)'
# Define the RBF kernel with a specific length scale
length_scale = 5.0
variance = 2.0
kernel = variance * RBF(length_scale=length_scale)

# Create a Gaussian Process model with the RBF kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

# Plot the samples from the prior distribution
for i in range(samples.shape[1]):
    ax[1,1].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[1,1].set_title(f'{chart}\nsignal variance = {variance}\nlength_scale = {length_scale}')
ax[1,1].set_xlabel('Input Value')
ax[1,1].set_ylabel('Kernel Value')

#############################################################################

plt.show()
```

Matérn Kernel:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern

title = "Samples from Prior Distribution with Matern Kernel"
fig, ax = plt.subplots(nrows=4, ncols=1, figsize=(8,8), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

# Generate sample data points
X = np.linspace(0, 10, 100).reshape(-1, 1)

#########################################################################
chart = '(a)'

# Define the Matern kernel with specific length scale and nu
length_scale = 1.0
nu = 0.5
kernel = Matern(length_scale=length_scale, nu=nu)

# Create a Gaussian Process model with the Matern kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

for i in range(samples.shape[1]):
    ax[0].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[0].set_title(f'{chart}\nsmoothness = {nu}')
ax[0].set_xlabel('Input Value')
ax[0].set_ylabel('Kernel Value')

#########################################################################
chart = '(b)'

# Define the Matern kernel with specific length scale and nu
length_scale = 1.0
nu = 1.5
kernel = Matern(length_scale=length_scale, nu=nu)

# Create a Gaussian Process model with the Matern kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

for i in range(samples.shape[1]):
    ax[1].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[1].set_title(f'{chart}\nsmoothness = {nu}')
ax[1].set_xlabel('Input Value')
ax[1].set_ylabel('Kernel Value')

#########################################################################
chart = '(c)'

# Define the Matern kernel with specific length scale and nu
length_scale = 1.0
nu = 2.5
kernel = Matern(length_scale=length_scale, nu=nu)

# Create a Gaussian Process model with the Matern kernel
```

```
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

for i in range(samples.shape[1]):
    ax[2].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[2].set_title(f'{chart}\nsmoothness = {nu}')
ax[2].set_xlabel('Input Value')
ax[2].set_ylabel('Kernel Value')

##########################################################################
chart = '(d)'

# Define the Matern kernel with specific length scale and nu
length_scale = 1.0
nu = 10.0
kernel = Matern(length_scale=length_scale, nu=nu)

# Create a Gaussian Process model with the Matern kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

for i in range(samples.shape[1]):
    ax[3].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[3].set_title(f'{chart}\nsmoothness = {nu}')
ax[3].set_xlabel('Input Value')
ax[3].set_ylabel('Kernel Value')

##########################################################################

plt.show()
```

Linear Kernel:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import DotProduct

title = "Samples from Prior Distribution with Linear Kernel"
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10,5), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

########################################################################
chart = '(a)'

# Define the Linear kernel with a specific sigma_0
sigma_0 = 1.0
kernel = DotProduct(sigma_0=sigma_0)

# Generate sample data points
X = np.linspace(0, 10, 100).reshape(-1, 1)

# Create a Gaussian Process model with the Linear kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

# Plot the samples from the prior distribution
for i in range(samples.shape[1]):
    ax[0].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[0].set_title(f'{chart}\nsignal variance = {sigma_0}')
ax[0].set_xlabel('Input Value')
ax[0].set_ylabel('Kernel Value')

########################################################################
chart = '(b)'

# Define the Linear kernel with a specific sigma_0
sigma_0 = 10.0
kernel = DotProduct(sigma_0=sigma_0)

# Generate sample data points
X = np.linspace(0, 10, 100).reshape(-1, 1)

# Create a Gaussian Process model with the Linear kernel
gp = GaussianProcessRegressor(kernel=kernel)

# Sample from the prior distribution (mean=0, covariance=kernel)
samples = gp.sample_y(X, n_samples=5, random_state=42)

# Plot the samples from the prior distribution
for i in range(samples.shape[1]):
    ax[1].plot(X, samples[:, i], label=f'Sample {i+1}')
ax[1].set_title(f'{chart}\nsignal variance = {sigma_0}')
ax[1].set_xlabel('Input Value')
ax[1].set_ylabel('Kernel Value')

plt.show()
```

Periodic Kernel:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process.kernels import ExpSineSquared

# Generate input values
X = np.linspace(0, 10, 100).reshape(-1, 1)
x_ref = np.array([[5.0]])

# Define different periodicity values
periodicities = [0.5, 1.0, 3.0, 10.0]
chart = ['(a)', '(b)', '(c)', '(d)']

# Create subplots
title = "Samples from Prior Distribution with ExpSineSquared (Periodic) Kernel"
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)
ax = ax.flatten()

for i, p in enumerate(periodicities):
    kernel = ExpSineSquared(length_scale=1.0, periodicity=p)
    k_values = kernel(x_ref, X).reshape(-1)

    ax[i].plot(X.reshape(-1), k_values)
    ax[i].set_title(f'{chart[i]}\nperiodicity = {p}')
    ax[i].set_xlabel("Input Value")
    ax[i].set_ylabel("Kernel Value")

plt.show()
```

Prior and Posterior Distributions with Matern Kernel:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern

# Generate input values
X = np.linspace(0, 10, 100).reshape(-1, 1)
y = np.sin(X).ravel()

# Define training data (observations)
X_train = np.array([[1.0], [3.0], [5.0], [6.5], [8.0]])
y_train = np.sin(X_train).ravel()

# Define different smoothness parameters for Matern kernel
nus = [0.5, 1.5]

# Create subplots
fig, ax = plt.subplots(nrows=5, ncols=1, figsize=(10, 12), tight_layout=True)
fig.subplots_adjust(hspace=0.5)
fig.suptitle("Prior and Posterior Distributions with Matern Kernel", weight='bold', fontsize=14)

ax[0].plot(X, y, label='y = sin(x)', color='blue')
ax[0].plot(X_train, y_train, 'ro', markersize=10, label='Observations')
ax[0].set_title(f'(a) Modeled Function')
ax[0].set_xlabel('x')
ax[0].set_ylabel('y')
ax[0].legend(loc='upper left', bbox_to_anchor=(1.05, 1))
ax[0].set_ylim(-5, 3)

for i, nu in enumerate(nus):
    # Define the Matern kernel
    kernel = Matern(length_scale=1.0, nu=nu)

    # Prior: sample from GP before fitting
    gp_prior = GaussianProcessRegressor(kernel=kernel, alpha=1e-6)
    y_prior_samples = gp_prior.sample_y(X, n_samples=3, random_state=0)

    # Posterior: fit GP to training data
    gp_posterior = GaussianProcessRegressor(kernel=kernel, alpha=1e-6)
    gp_posterior.fit(X_train, y_train)
    y_mean, y_std = gp_posterior.predict(X, return_std=True)
    y_post_samples = gp_posterior.sample_y(X, n_samples=3, random_state=42)

    if i == 0:
        # Plot prior samples
        for j in range(y_prior_samples.shape[1]):
            ax[1].plot(X, y_prior_samples[:, j], lw=1, ls=':', label=f'Prior Sample {j+1}')
            ax[1].set_title(f'(b) Prior Samples from Matern Kernel with v={nu}')
            ax[1].set_xlabel('Input Value')
            ax[1].set_ylabel('Kernel Value')
            ax[1].legend(loc='upper left', bbox_to_anchor=(1.05, 1))
            ax[1].set_ylim(-5, 3)

            # Plot confidence interval
        ax[2].fill_between(X.ravel(), y_mean - 1.96 * y_std, y_mean + 1.96 * y_std,
                           alpha=0.15, color='gray', label='95% Confidence Interval')

        # Plot posterior samples
        for j in range(y_post_samples.shape[1]):
            ax[2].plot(X, y_post_samples[:, j], lw=1,  label=f'Posterior Sample {j+1}')

        # Plot posterior mean
```

```
        ax[2].plot(X, y_mean, 'k', lw=2, label='Posterior Mean')

        # Plot training data
        ax[2].plot(X_train, y_train, 'ro', markersize=10, label='Observations')
        ax[2].set_title(f'(c) Posterior Samples with Matern Kernel and v={nu}')
        ax[2].set_xlabel('Input Value')
        ax[2].set_ylabel('Output Value')
        ax[2].legend(loc='upper left', bbox_to_anchor=(1.05, 1))
        ax[2].set_ylim(-5, 3)

    if i==1:
        # Plot prior samples
        for j in range(y_prior_samples.shape[1]):
            ax[3].plot(X, y_prior_samples[:, j], lw=1, ls=':', label=f'Prior Sample {j+1}')
            ax[3].set_title(f'(d) Prior Samples from Matern Kernel with v={nu}')
            ax[3].set_xlabel('Input Value')
            ax[3].set_ylabel('Kernel Value')
            ax[3].legend(loc='upper left', bbox_to_anchor=(1.05, 1))
            ax[3].set_ylim(-5, 3)

            # Plot confidence interval
        ax[4].fill_between(X.ravel(), y_mean - 1.96 * y_std, y_mean + 1.96 * y_std,
                           alpha=0.15, color='gray', label='95% Confidence Interval')

        # Plot posterior samples
        for j in range(y_post_samples.shape[1]):
            ax[4].plot(X, y_post_samples[:, j], lw=1,  label=f'Posterior Sample {j+1}')

        # Plot posterior mean
        ax[4].plot(X, y_mean, 'k', lw=2, label='Posterior Mean')

        # Plot training data
        ax[4].plot(X_train, y_train, 'ro', markersize=10, label='Observations')
        ax[4].set_title(f'(e) Posterior Samples with Matern Kernel and v={nu}')
        ax[4].set_xlabel('Input Value')
        ax[4].set_ylabel('Output Value')
        ax[4].legend(loc='upper left', bbox_to_anchor=(1.05, 1))
        ax[4].set_ylim(-5, 3)

plt.show()
```

A Non-Stationary Function Example:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0, 10, 500)
y = np.piecewise(
    X,
    [X > 4, X <= 4],
    [lambda x: 0.2 * x, lambda x: (0.2 * x) + (0.5 * x * np.sin(2 * x**1.5))]
)

plt.plot(X, y, label='y = f(x)', color='blue')
plt.title('A Non-Stationary Function', weight='bold')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc='upper right')
plt.show()
```

GPR with Matern Kernel on Non-Stationary Function Example:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern

# Generate input values
X = np.linspace(0, 10, 500).reshape(-1, 1)

# Define the piecewise function
y_true = np.piecewise(
    X,
    [X > 4, X <= 4],
    [lambda x: 0.2 * x, lambda x: (0.2 * x) + (0.5 * x * np.sin(2 * x**1.5))]
)

# Define training data (observations)
X_train = np.array([[1.0], [2.5], [4.0], [5.5], [7.0], [8.5]])
y_train = np.piecewise(
    X_train.ravel(),
    [X_train.ravel() > 4, X_train.ravel() <= 4],
    [lambda x: 0.2 * x, lambda x: (0.2 * x) + (0.5 * x * np.sin(2 * x**1.5))]
)

# Define the Matern kernel
kernel = Matern(length_scale=1.0, nu=1.5)

# Create and fit the Gaussian Process model
gp = GaussianProcessRegressor(kernel=kernel, alpha=1e-6)
gp.fit(X_train, y_train)

# Predict mean and standard deviation
y_mean, y_std = gp.predict(X, return_std=True)

# Sample from the posterior distribution
y_samples = gp.sample_y(X, n_samples=5, random_state=42)

# Plot the true function
plt.figure(figsize=(10, 6))
plt.plot(X, y_true, 'b-', label='A Non-Stationary Function', linewidth=2)

# Plot posterior mean and confidence interval
plt.plot(X, y_mean, 'k', lw=2, label='Posterior Mean')
plt.fill_between(X.ravel(), y_mean - 1.96 * y_std, y_mean + 1.96 * y_std,
                 alpha=0.2, color='gray', label='95% Confidence Interval')

# Plot posterior samples
for i in range(y_samples.shape[1]):
    plt.plot(X, y_samples[:, i], lw=1, ls='--', label=f'Sample {i+1}')

# Plot training data
plt.plot(X_train, y_train, 'ro', label='Obsevations', markersize=8)

# Add labels and legend
plt.title('Gaussian Process Regression with Matern Kernel (v = 1.5) on a Non-Stationary Function',
weight='bold')
plt.xlabel('Input Value')
plt.ylabel('Output Value')
plt.legend(loc='upper left', bbox_to_anchor=(1.05, 1))
plt.tight_layout()
plt.show()
```

APPENDIX B – HISTORICAL DATA

Collected Historical Data

	A	B	C	D	E	F	G	H	I	J	L	M
1	time	icao24	lat	lon	velocity	heading	vertrate	onground	baroaltitude	flight_id	departure_airport	arrival_airport
2	6/5/2025 20:22	a73ab6	38.96305925	-77.81140137	226.8736943	80.86982352	-0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
3	6/5/2025 20:22	a73ab6	38.96356201	-77.80726557	224.8892637	80.76897346	0	FALSE	3017.52	JIA5318-2025-06-05	KBNA	KDCA
4	6/5/2025 20:22	a73ab6	38.9641571	-77.80254862	222.9252889	80.7066914	0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
5	6/5/2025 20:22	a73ab6	38.96470642	-77.79836903	222.9252889	80.7066914	0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
6	6/5/2025 20:22	a73ab6	38.96539307	-77.79317441	221.9384762	80.66500154	0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
7	6/5/2025 20:22	a73ab6	38.96580606	-77.78953221	221.1824797	81.94316587	0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
8	6/5/2025 20:22	a73ab6	38.96621704	-77.78409875	220.274129	87.13758477	-0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
9	6/5/2025 20:22	a73ab6	38.96621704	-77.780158	220.1833149	92.34261201	0	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
10	6/5/2025 20:22	a73ab6	38.96585238	-77.77508545	219.4281072	96.54203815	0	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
11	6/5/2025 20:22	a73ab6	38.96529375	-77.7699685	220.05384	97.83552511	0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
12	6/5/2025 20:22	a73ab6	38.96492134	-77.76629639	221.1824797	98.05683413	0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
13	6/5/2025 20:23	a73ab6	38.96431616	-77.76086426	222.0353281	97.76516602	0	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
14	6/5/2025 20:23	a73ab6	38.96388245	-77.75705089	223.0261961	97.73045533	0	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
15	6/5/2025 20:23	a73ab6	38.96333313	-77.75179656	222.8838862	97.47573739	0	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
16	6/5/2025 20:23	a73ab6	38.96296614	-77.74798684	223.8854209	97.44244124	-0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
17	6/5/2025 20:23	a73ab6	38.9624176	-77.74260148	223.8854209	97.44244124	-0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
18	6/5/2025 20:23	a73ab6	38.96198855	-77.73852539	223.8854209	97.44244124	-0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
19	6/5/2025 20:23	a73ab6	38.9614563	-77.7334064	223.8854209	97.44244124	-0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
20	6/5/2025 20:23	a73ab6	38.96101095	-77.72930908	223.8854209	97.44244124	-0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
21	6/5/2025 20:23	a73ab6	38.96045232	-77.72381592	223.8854209	97.44244124	-0.32512	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
22	6/5/2025 20:23	a73ab6	38.96008301	-77.71997203	222.8938862	97.47573739	0	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
23	6/5/2025 20:23	a73ab6	38.95857947	-77.71501624	222.7658868	97.2207219	0	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
24	6/5/2025 20:23	a73ab6	38.95907593	-77.71000074	222.7658868	97.2207219	0	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
25	6/5/2025 20:23	a73ab6	38.95882299	-77.70727539	222.6425054	96.96541848	-0.65024	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
26	6/5/2025 20:23	a73ab6	38.95826437	-77.70147705	221.6499222	96.99676636	-1.95072	FALSE	3025.14	JIA5318-2025-06-05	KBNA	KDCA
27	6/5/2025 20:23	a73ab6	38.9578454	-77.69683838	221.5303328	96.7400561	-2.92608	FALSE	3017.52	JIA5318-2025-06-05	KBNA	KDCA
28	6/5/2025 20:23	a73ab6	38.95733643	-77.69149117	221.5303328	96.7400561	-3.57632	FALSE	3002.28	JIA5318-2025-06-05	KBNA	KDCA
29	6/5/2025 20:23	a73ab6	38.95697021	-77.68755042	221.5303328	96.7400561	-3.90144	FALSE	2994.66	JIA5318-2025-06-05	KBNA	KDCA
30	6/5/2025 20:24	a73ab6	38.95655823	-77.68337084	221.5303328	96.7400561	-4.22656	FALSE	2979.42	JIA5318-2025-06-05	KBNA	KDCA
31	6/5/2025 20:24	a73ab6	38.95612296	-77.67828369	222.408826	96.45398659	-3.90144	FALSE	2971.8	JIA5318-2025-06-05	KBNA	KDCA
32	6/5/2025 20:24	a73ab6	38.95565744	-77.67303467	222.408826	96.45398659	-3.90144	FALSE	2956.56	JIA5318-2025-06-05	KBNA	KDCA
33	6/5/2025 20:24	a73ab6	38.95528502	-77.66912842	221.4151953	96.48307369	-3.90144	FALSE	2941.32	JIA5318-2025-06-05	KBNA	KDCA
34	6/5/2025 20:24	a73ab6	38.95486606	-77.66412354	221.4151953	96.48307369	-4.22656	FALSE	2926.08	JIA5318-2025-06-05	KBNA	KDCA
35	6/5/2025 20:24	a73ab6	38.95440364	-77.65997314	221.4151953	96.48307369	-4.55168	FALSE	2918.46	JIA5318-2025-06-05	KBNA	KDCA
36	6/5/2025 20:24	a73ab6	38.95404053	-77.65459144	222.2986419	96.19787763	-4.8768	FALSE	2895.6	JIA5318-2025-06-05	KBNA	KDCA
37	6/5/2025 20:24	a73ab6	38.95370225	-77.65081787	222.2986419	96.19787763	-4.8768	FALSE	2887.98	JIA5318-2025-06-05	KBNA	KDCA

Figure 98 Snapshot of Collected Historical Data

Historical Data Statistics

The monthly number of collected historical flight tracks is shown in Figure 99, with an average of 6,551 collected tracks per month at DCA. The daily number of collected flight tracks is shown in Figure 100. There were 213 flight trajectories per day. The number of flights has substantially dropped at the beginning of July 2025. Server connectivity issues can explain this drop during the data download.

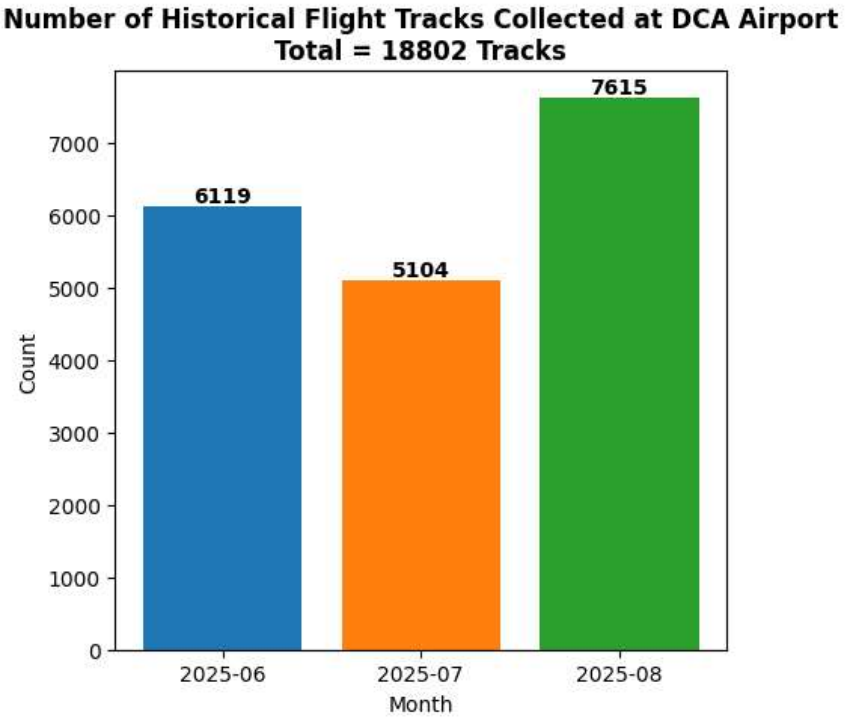


Figure 99 Number of Collected Flight Tracks per Month

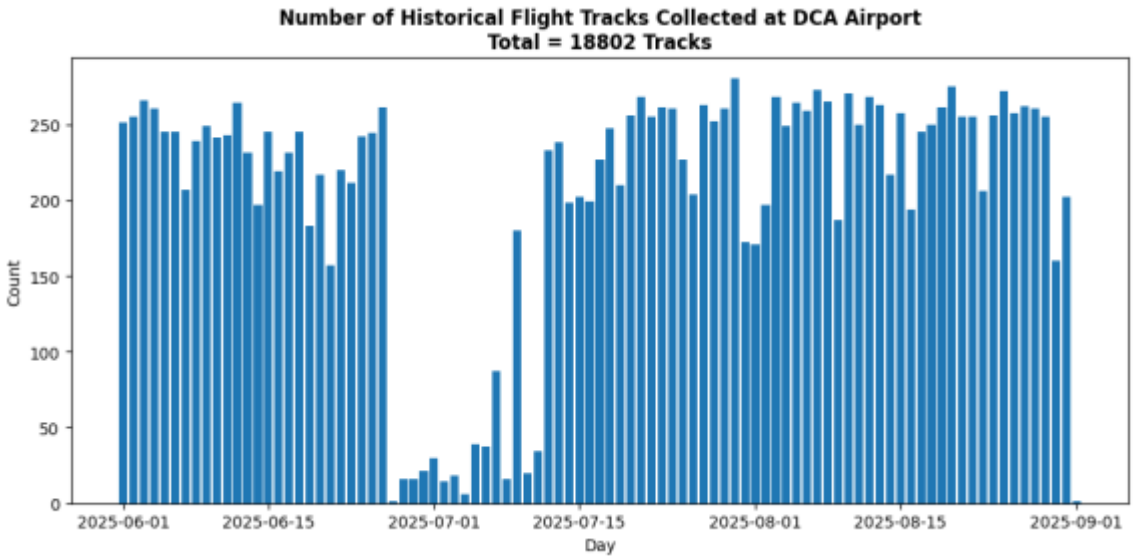


Figure 100 Number of Collected Flight Tracks per Day

The total along track distance across all captured flight tracks is shown in Figure 101. The distribution is multimodal and positively skewed, indicating the presence of longer trajectories. The histogram suggests a heterogeneous traffic pattern composed of multiple arrival streams. Most aircraft traveled 35 to 45 nm or 60 nm on average. Navigation procedures and landing patterns determine the range of distances traveled.

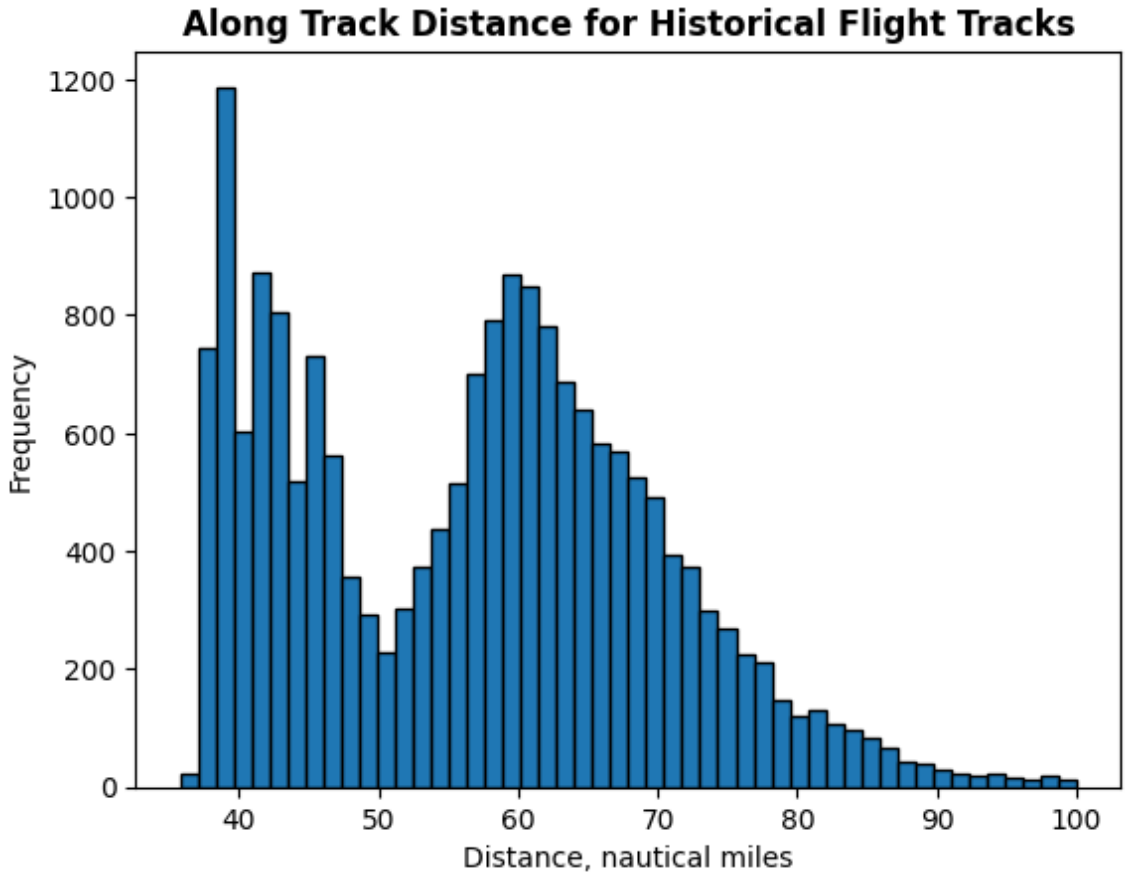


Figure 101 Along Track Distance Across All Flight Tracks

APPENDIX C – WAYPOINT DATA

Table 25 STARs - Standard Terminal Arrivals – SKILS FIVE (RNAV)

Waypoint	Latitude	Longitude	Altitude, ft	Speed, knots	Course	Distance to the next waypoint, nm
Baltimore BAL	39.16673889	-76.65018889	10000	250	234°	16
EYESS	38.96675278	-76.88355833	~9000		284°	6
EDDWD	38.96682778	-77.01684444	~8000		328°	5
MEGGS	39.03336944	-77.08359167	~7000		327°	8
NAYES	39.13333889	-77.21673889	~6000		327°	

Table 26 IAPs - Instrument Approach Procedures – RNAV (GPS) Y RWY 19

Waypoint	Latitude	Longitude	Altitude, ft	Speed, knots	Course	Distance to the next waypoint, nm
DARIC	38.95025	-77.15005278	2700	180	150°	2.5
SUNEY	38.93334167	-77.11669722	2000		158°	1.7
RORRK	38.90016389	-77.08359167	1500		122°	1.6
SLAKR	38.90000833	-77.06668333	1020		149°	0.7
DALLA	38.88347222	-77.050125	780		149°	1.2
UDERE	38.86683056	-77.03345	373		149°	0.9
RWY 19	38.861193	-77.038730	0		149°	0

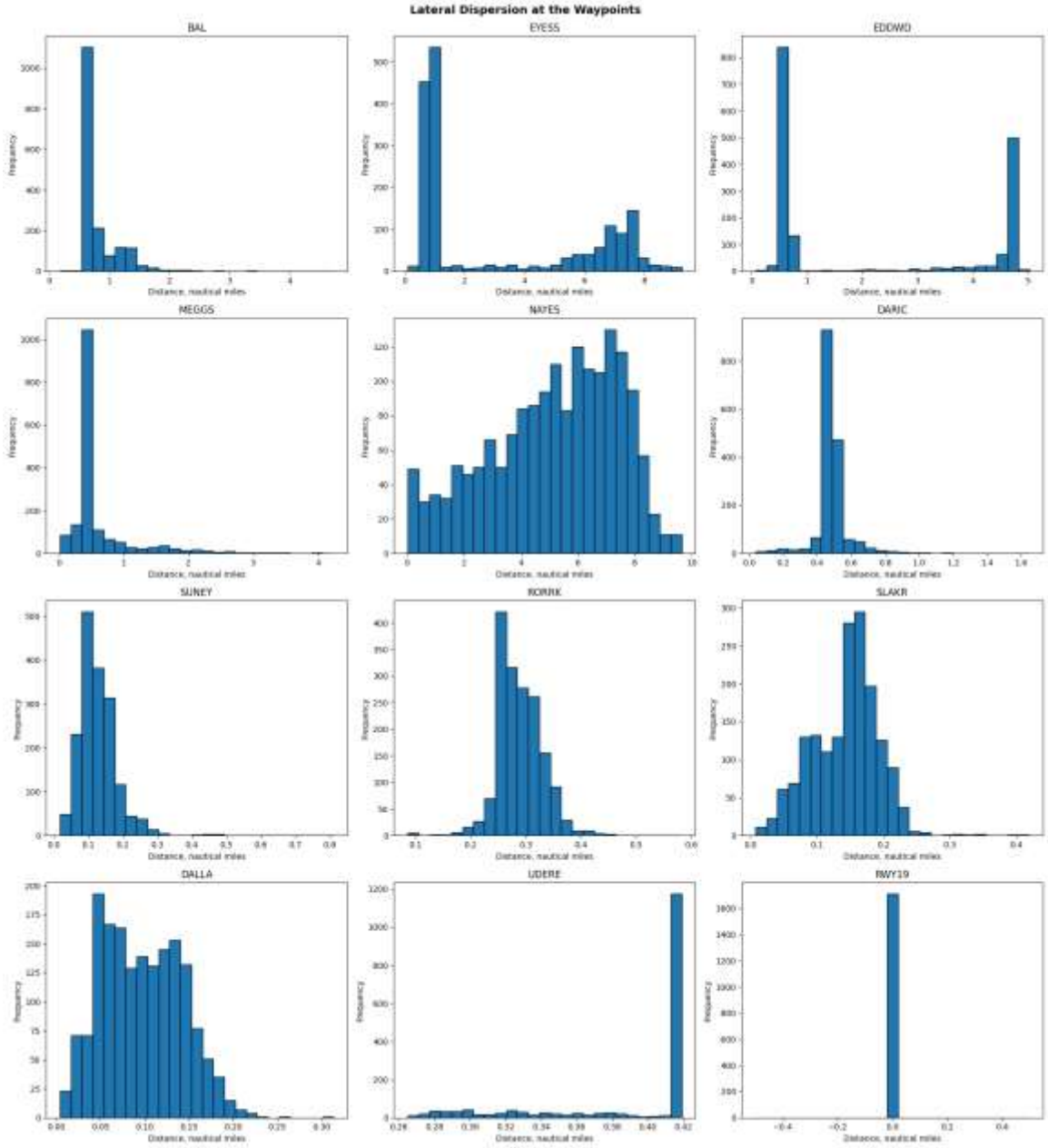


Figure 102 Lateral Dispersion at Waypoints

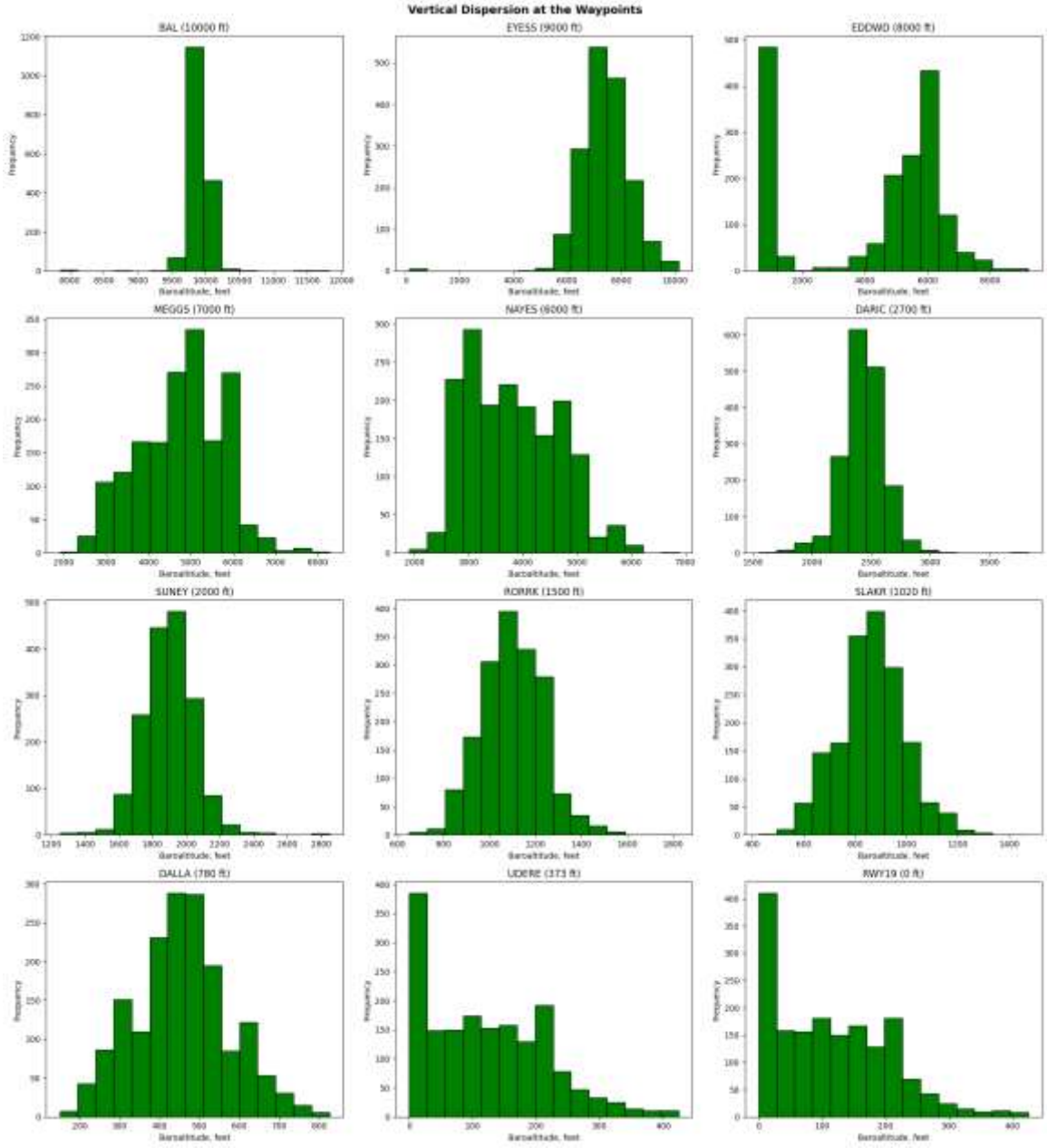


Figure 103 Vertical Dispersion at Waypoints

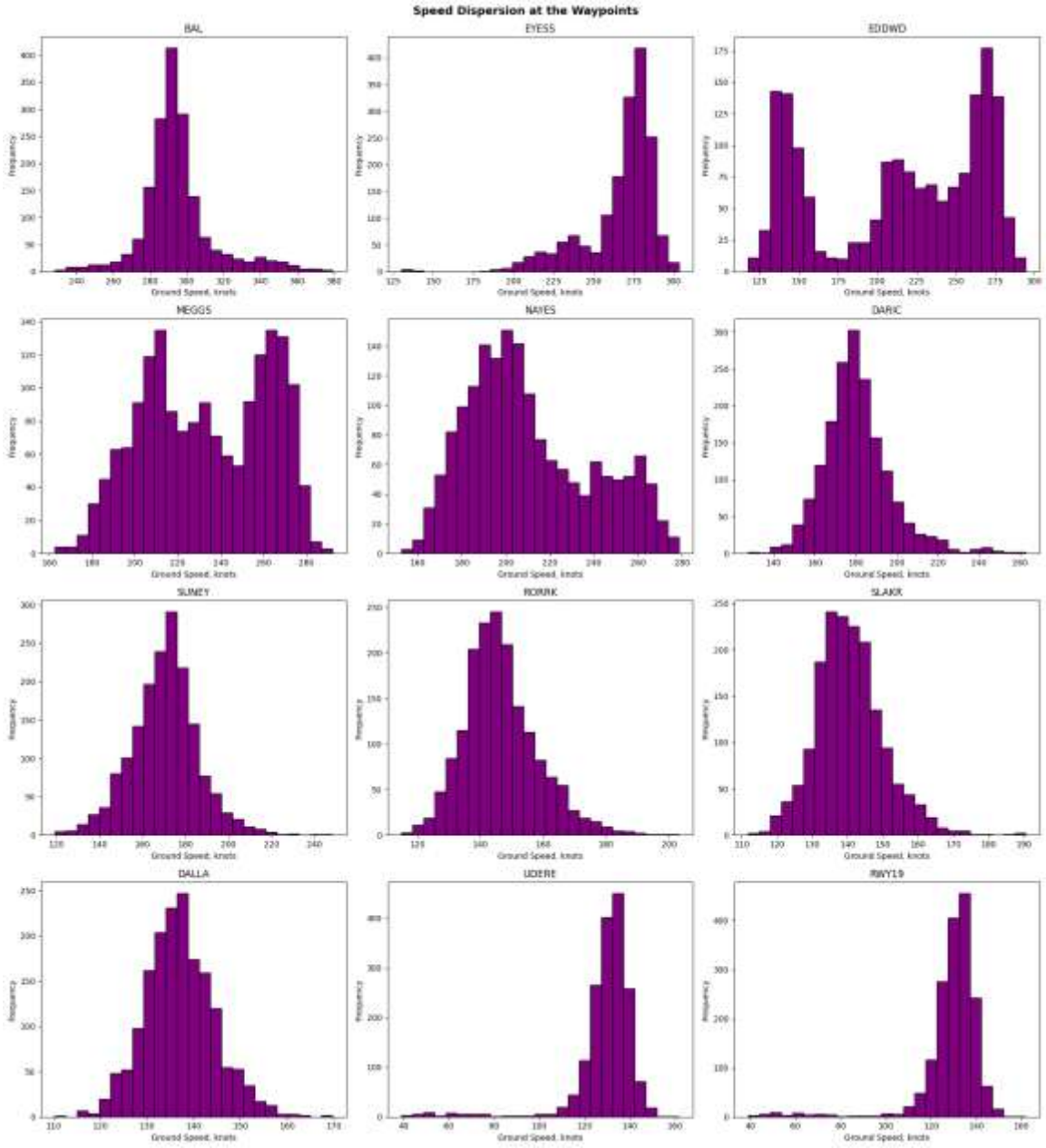


Figure 104 Speed Dispersion at Waypoints

APPENDIX D – ADDITIONAL SYNTHETIC RESULTS FROM MODEL 5

Batch = 5

The synthetic flight track data parameters extracted from batches with 5 posterior samples in Model 5 are shown in Figure 105, Figure 106, and Figure 107.

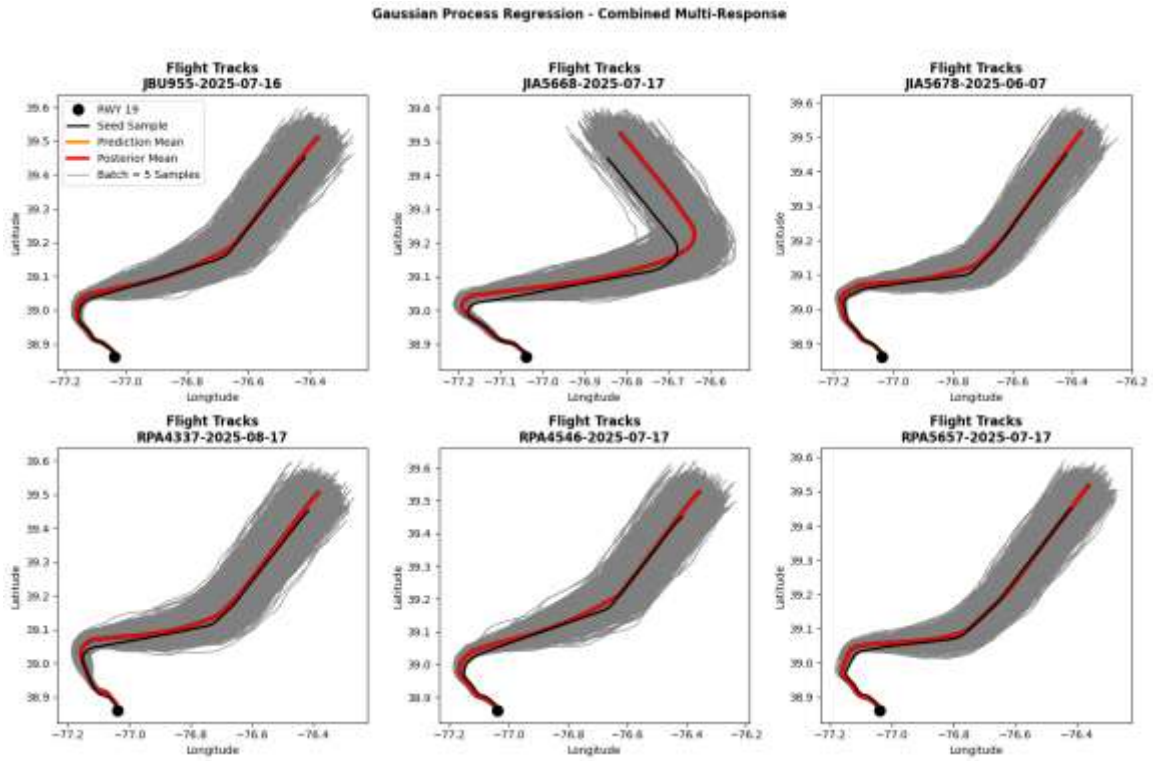


Figure 105 GPR for Flight Tracks in Model 5 (Batch = 5)

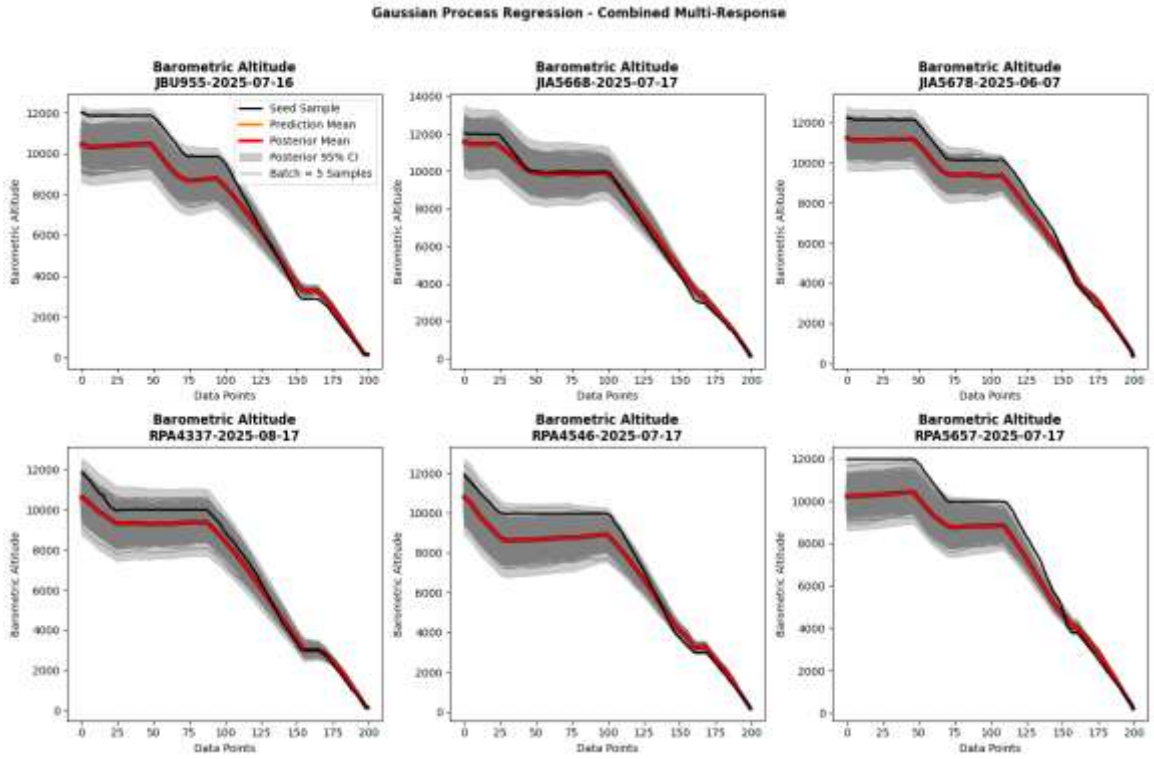


Figure 106 GPR for Barometric Altitude in Model 5 (Batch = 5)

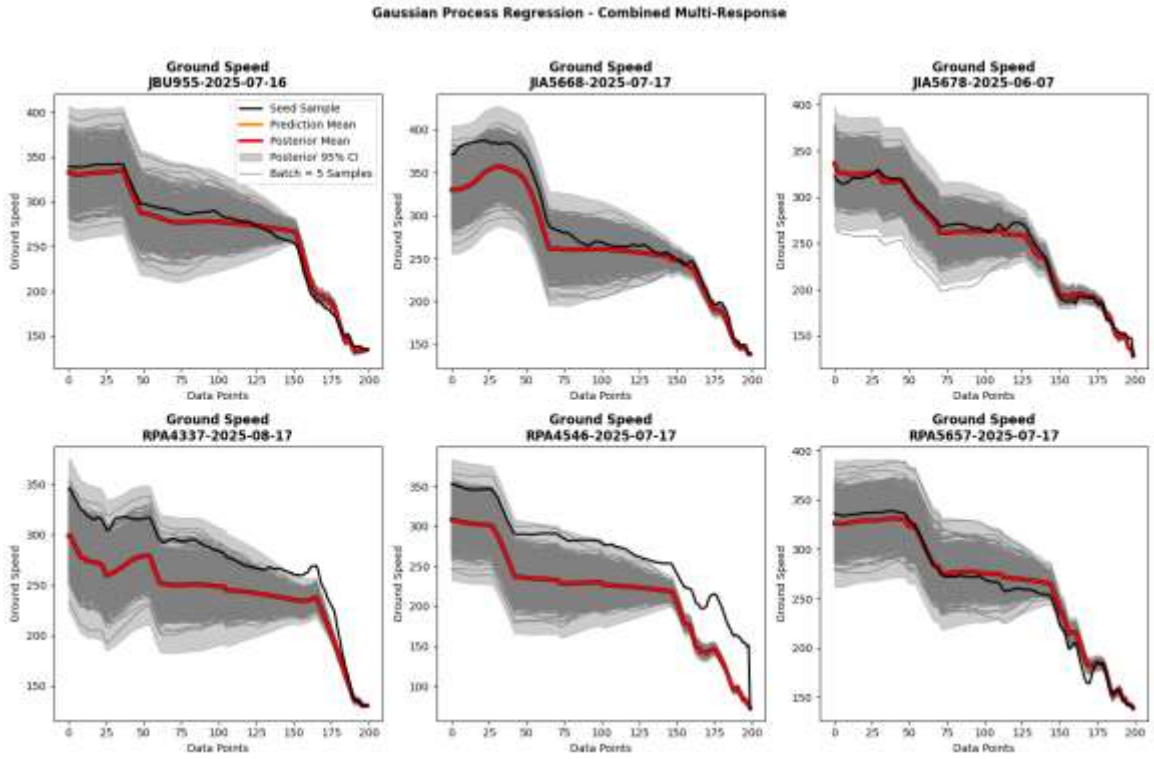


Figure 107 GPR for Ground Speed in Model 5 (Batch = 5)

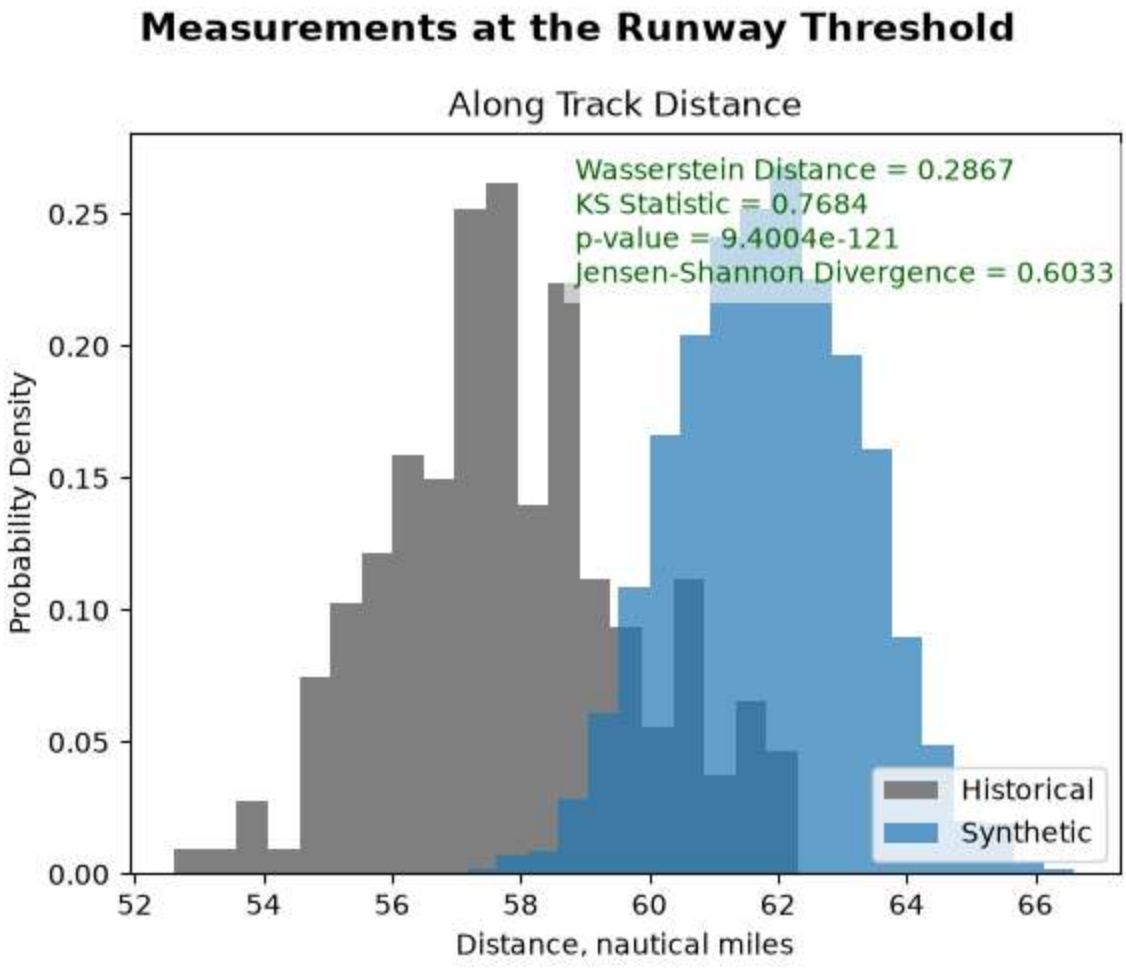


Figure 108 Along Track Distance Distribution Comparison in Model 5 (Batch = 5)

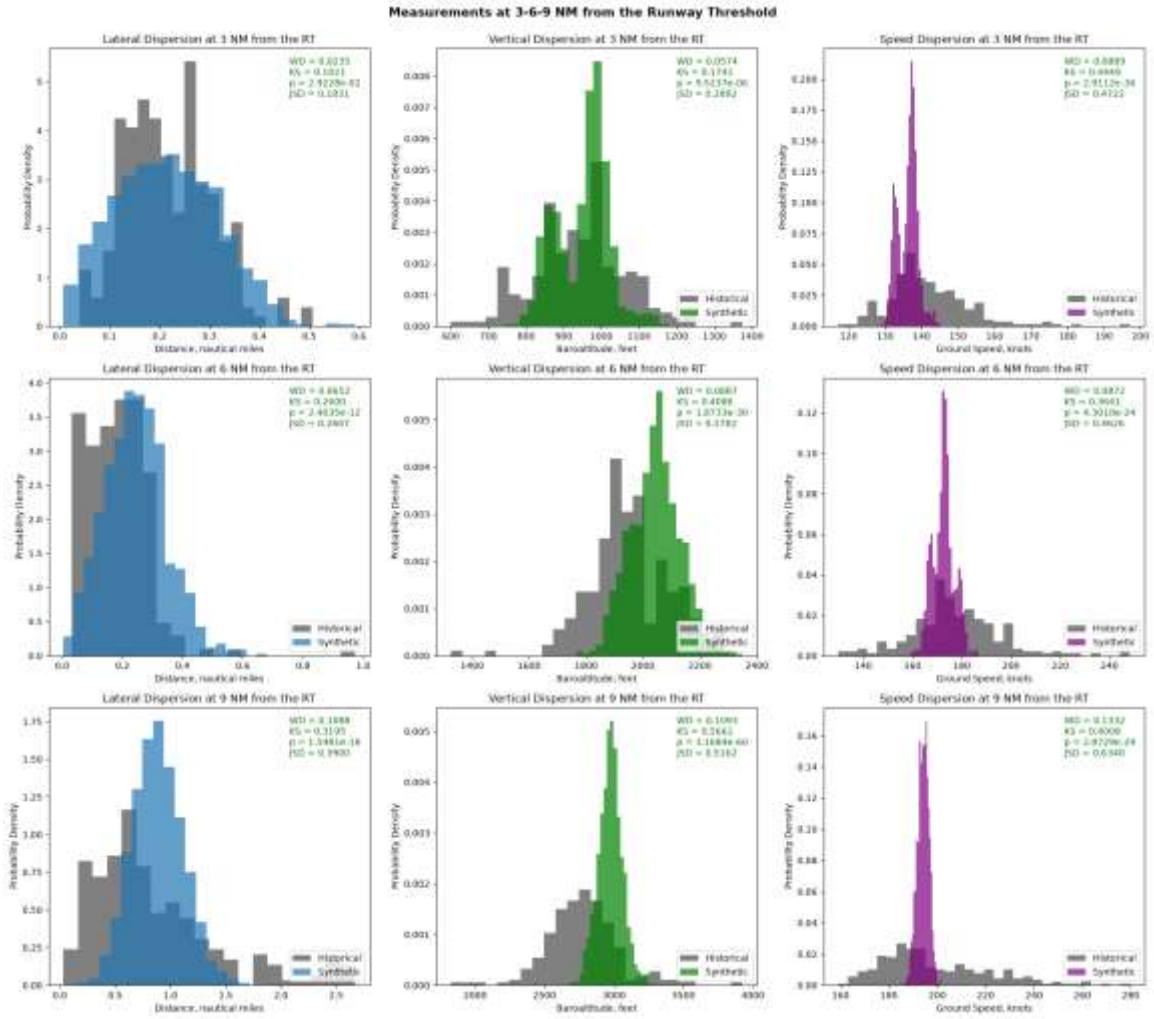


Figure 109 Distributions for Measurements at 3, 6, and 9 NM in Model 5 (Batch = 5)

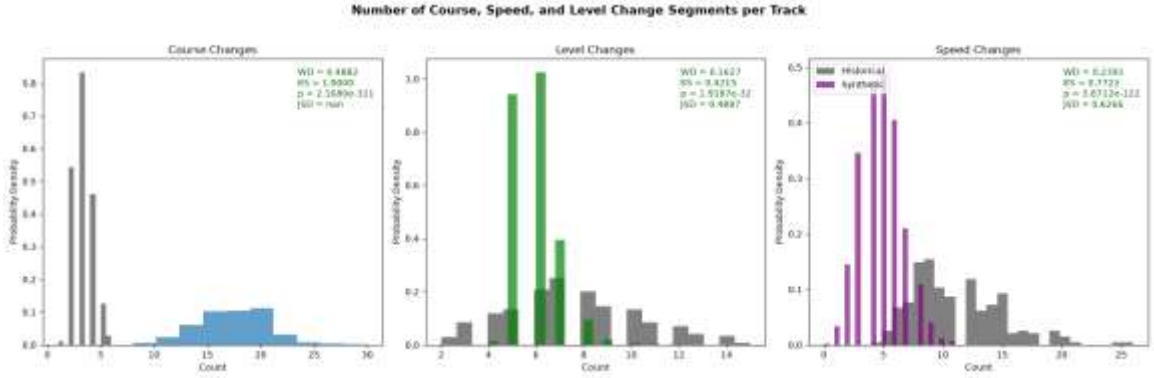


Figure 110 Distributions for Individual Track Characteristics in Model 5 (Batch = 5)

Batch = 25

The synthetic flight track data parameters extracted from batches with 25 posterior samples in Model 5 are shown in Figure 111, Figure 112, and Figure 113.

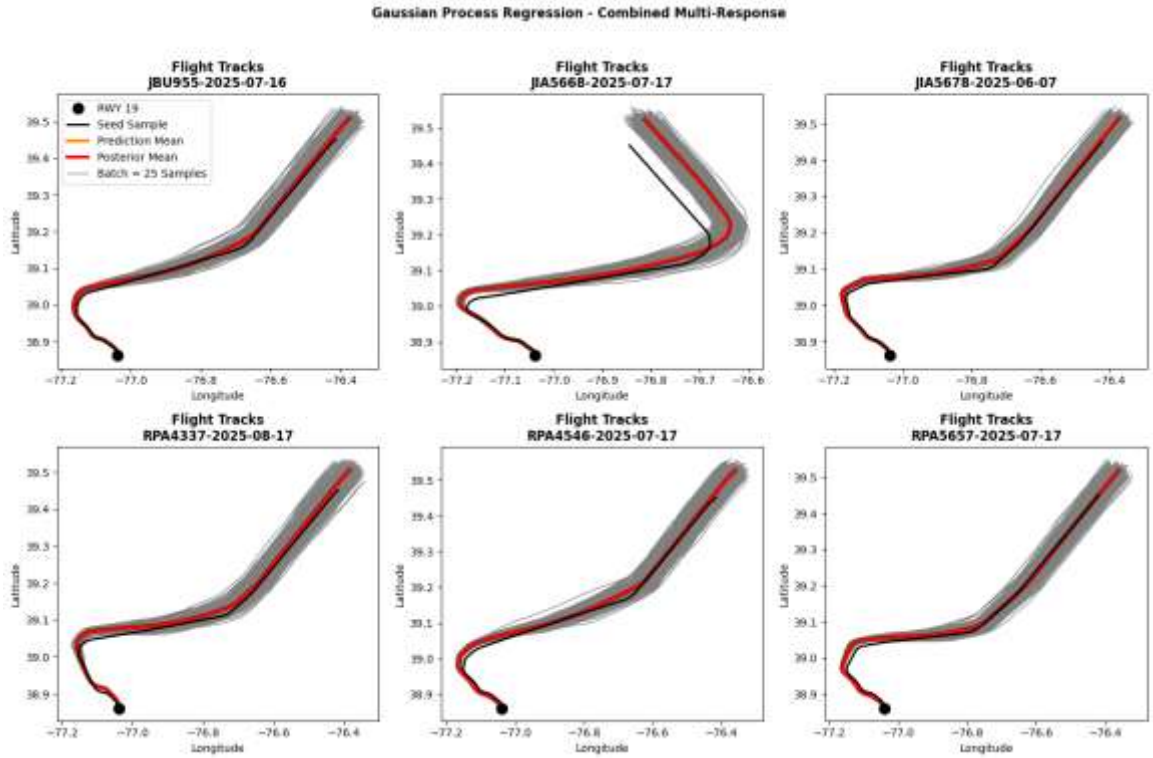


Figure 111 GPR for Flight Tracks in Model 5 (Batch = 25)

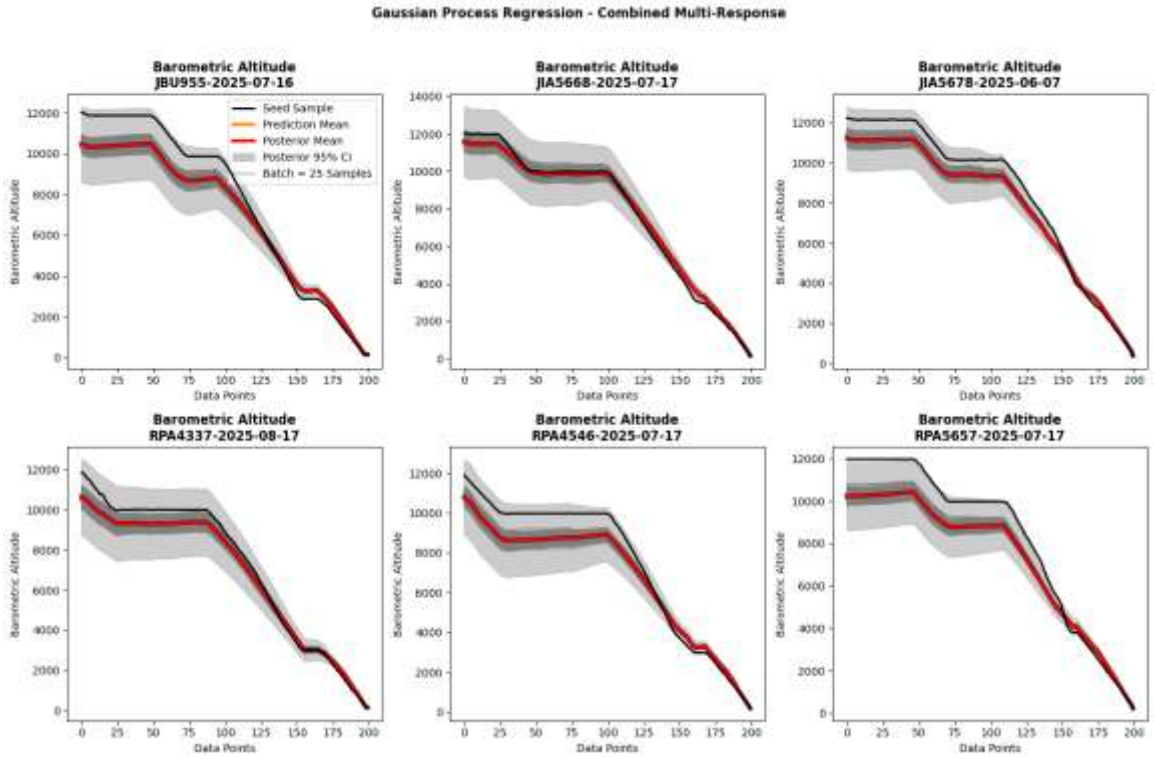


Figure 112 GPR for Barometric Altitude in Model 5 (Batch = 25)

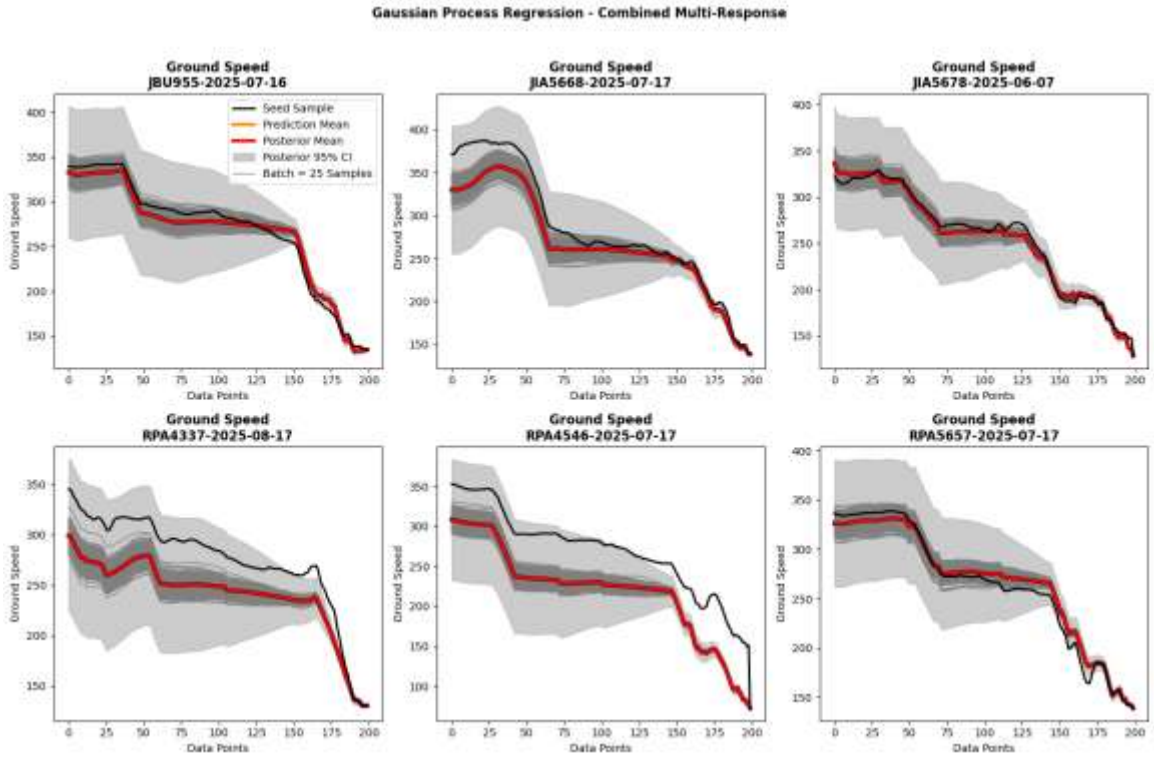


Figure 113 GPR for Ground Speed in Model 5 (Batch = 25)

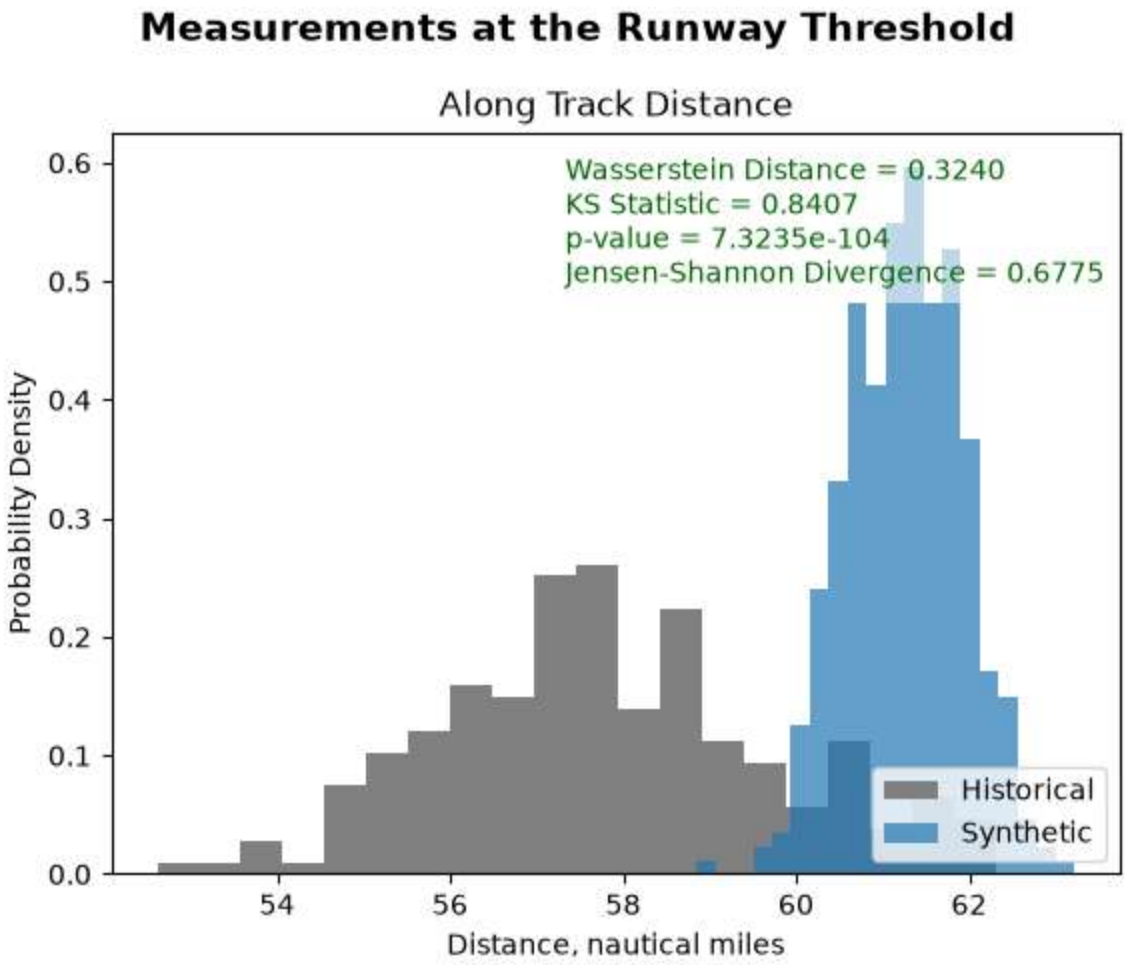


Figure 114 Along Track Distance Distribution Comparison in Model 5 (Batch = 25)

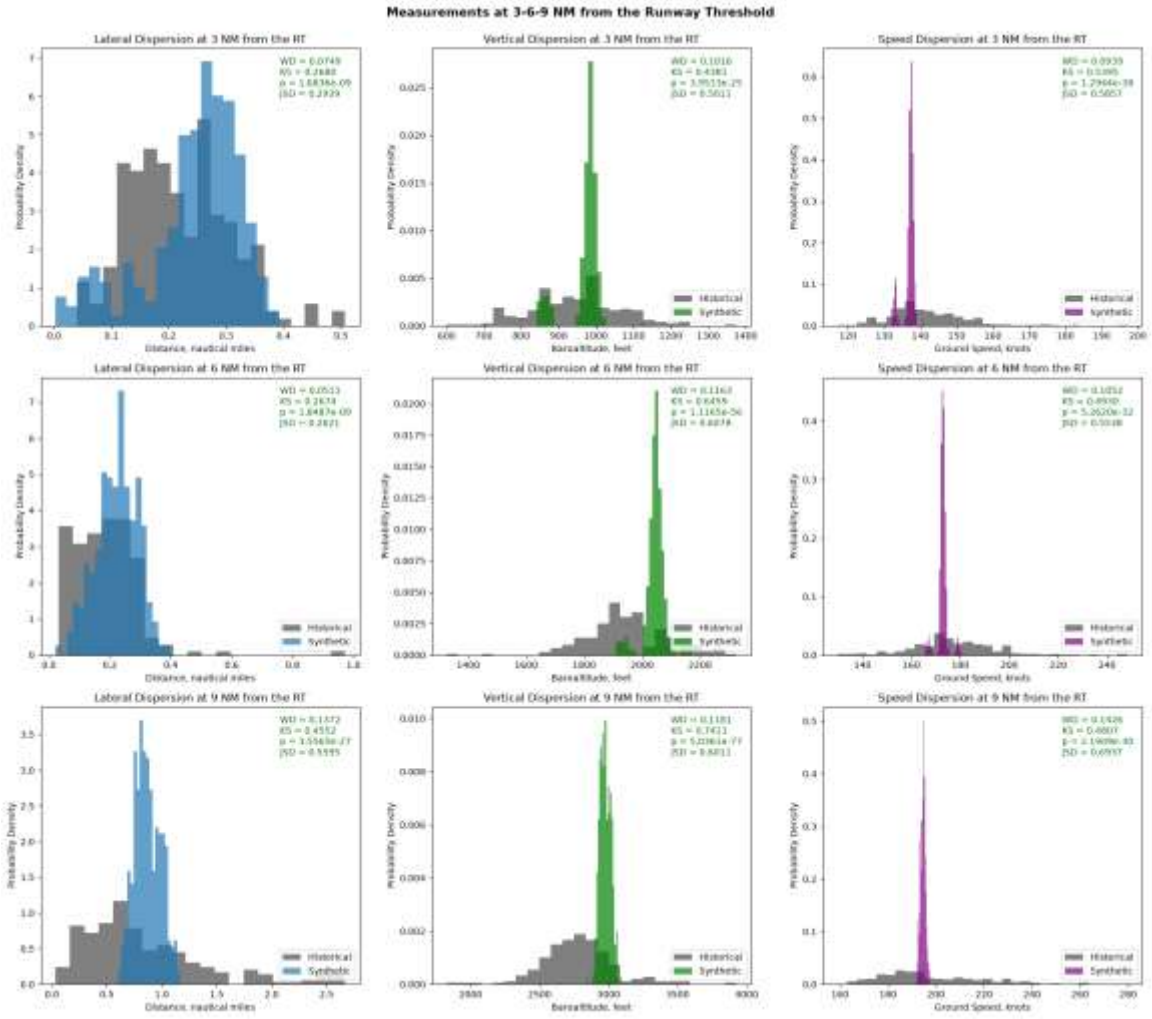


Figure 115 Distributions for Measurements at 3, 6, and 9 NM in Model 5 (Batch=25)

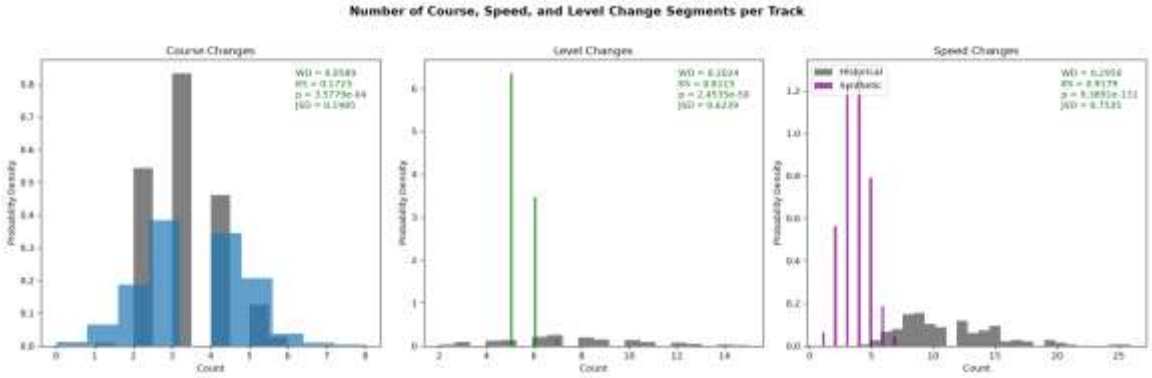


Figure 116 Distributions for Individual Track Characteristics in Model 5 (Batch=25)

APPENDIX E – PYTHON SCRIPTS FOR HISTORICAL DATA

Data Preprocessing:

```
# %% [markdown]
# Import and install the necessary libraries and packages:

# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
from math import radians, sin, cos, acos
import glob

# %% [markdown]
# Indicate the CSV data file location and output directory:

# %%
file_dir = "C:\\Raw Data\\" # File directory
output_dir = "C:\\Files\\"
file_path = os.path.join(file_dir, "DCA_Arrivals.csv") # File name

# %% [markdown]
# Read the CSV data:

# %%
data = pd.read_csv(file_path)

# %% [markdown]
# Create the data dictionary:

# %%
columns = {'time':0, 'icao24':1, 'lat':2, 'lon':3, 'velocity':4, 'heading':5, 'vertrate':6,
'onground':7, 'baroaltitude':8, 'flight_id':9, 'distance':10, 'departure_airport':11,
'arrival_airport':12, 'distance_between_points':13,}

# %% [markdown]
# Convert Timestamp into Datetime:

# %%
data.loc[:, 'time'] = pd.to_datetime(data['time'], format="%Y-%m-%d %H:%M:%S.%f")

# %% [markdown]
# Create a list of tracks and assign an unique flight id:

# %%
unique_tracks = data['flight_id'].unique() # Unique flight IDs
print(len(unique_tracks))

# %%
# Get min and max dates
min_date = data['time'].min()
```

```
max_date = data['time'].max()

print("Min date:", min_date)
print("Max date:", max_date)

# %% [markdown]
# Create a list of tracks:

# %%
# Sort once globally by flight_id and time
data_sorted = data.sort_values(by=['flight_id', 'time']).reset_index(drop=True)

# Group by flight_id
tracks = [group.to_numpy() for _, group in data_sorted.groupby('flight_id')]

# %% [markdown]
# DCA and BWI airport coordinates:

# %%
dca = {'lat':38.8514403, 'lon':-77.0377214}
bwi = {'lat':39.1757283, 'lon':-76.6689911}

# %%
def get_great_circle_distance(lon1, lat1, lon2, lat2):
    r = 6371.137 # The Earth's radius in km

    # Convert to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # Compute cosine of central angle
    cos_angle = sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) * cos(lon1 - lon2)

    # Clamp the value to avoid math domain error
    cos_angle = min(1, max(-1, cos_angle))

    # Great circle distance in km
    d = r * acos(cos_angle)

    # Convert km to nm:
    d = d * 0.5399568035

    return d

# %%
tracks_filtered = []

for i, track in enumerate(tracks):
    if track.size > 1:

        # The last recorded track’s position is flagged as “on-ground”
        if track[-1, columns['onground']] == True:

            # The first recorded track’s position has  originated at either side of the 36NM
bounding box
            if (track[0, columns['lon']] <= -77.75 and track[0, columns['lat']] >= 38.1 and
track[0, columns['lat']] <= 39.5) or\
            (track[0, columns['lon']] >= -76.3 and track[0, columns['lat']] >= 38.1 and track[0,
columns['lat']] <= 39.5) or\
            (track[0, columns['lat']] >= 39.45 and track[0, columns['lon']] >= -77.9 and track[0,
columns['lon']] <= -76.1) or\
            (track[0, columns['lat']] <= 38.3 and track[0, columns['lon']] >= -77.9 and track[0,
columns['lon']] <= -76.1):

                for i in range(len(track)):
```

```
                    # The last recorded track's position is within the DCA airport
coordinates
                    if track[i, columns['lat']] >= 38.83 and track[i, columns['lat']] <= 38.87
and\
                    track[i, columns['lon']] >= -77.046 and track[i, columns['lon']] <= -77.03
and\
                    track[i, columns['onground']]:
                        lat_last = track[i, columns['lat']]
                        lon_last = track[i, columns['lon']]
                        d = get_great_circle_distance(lon_last, lat_last, dca['lon'], dca['lat'])

                        # The distance between the last recorded position and the airport is less
than 1nm:
                        if d < 0.4:

                            # At least 500 records are available and eliminate helicopter traffic
                            if i > 500 and i < 2000:

                                # Not cruising altitude
                                if track[0, columns['baroaltitude']] < 8000 and track[0,
columns['baroaltitude']] > 2200:

                                    # Not a level flight
                                    if track[0, columns['baroaltitude']] - track[300,
columns['baroaltitude']] > 100:
                                        tracks_filtered.append(track[0:i, :].copy())
                                        break

# %% [markdown]
# Plots for position, altitude, and ground speed profiles:

# %%
title = f"Historical Flight Tracks - Raw Data\n Arriving Trajectories at DCA Airport\nfrom
{min_date.date()} to {max_date.date()}\n"
# fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(18,6), tight_layout=True)
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

for track in tracks_filtered:
    x = np.arange(0, track.shape[0])
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend()

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, meters')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

plt.show()

# %%
# Remove tracks where the last latitude is more than 38.90
tracks_filtered = [
    track for track in tracks_filtered
```

```
    if float(track[-1, columns['lat']]) < 38.90
]

# %% [markdown]
# Filter by position:

# %%
first_positions = np.array([(float(track[0, columns['lat']]), float(track[0, columns['lon']])) for
track in tracks_filtered])
last_positions = np.array([(float(track[-1, columns['lat']]), float(track[-1, columns['lon']]))
for track in tracks_filtered])

# Get max latitude and max longitude
max_lat = np.max(last_positions[:, 0])
max_lon = np.max(last_positions[:, 1])

print(f"Max Latitude: {max_lat}")
print(f"Max Longitude: {max_lon}")

title = "The First and Last Recorded Positions from Each Flight Track"
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

ax[0].scatter(first_positions[:, 1], first_positions[:, 0], color='purple', label='First
Position')
ax[0].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')
ax[0].set_title('The First Recorded Position')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend()

ax[1].scatter(last_positions[:, 1], last_positions[:, 0], color='orange', label='Last Position')
ax[1].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')
ax[1].set_title('The Last Recorded Position')
ax[1].set_xlabel('Longitude')
ax[1].set_ylabel('Latitude')
ax[1].legend()

# %% [markdown]
# Convert meters to feet in altitude data and substitute invalid data records:

# %%
next_alt = True
next_vel = True

for track in tracks_filtered:

    has_nan = np.isnan(track[:, columns['baroaltitude']].astype(float)).any()

    # Convert meters to feet
    track[0, columns['baroaltitude']] = track[0, columns['baroaltitude']] * 3.28084

    for i in range(1, len(track)):

        # Substitute NaN values from altitude data
        if has_nan:
            track[i, columns['baroaltitude']] = track[i - 1, columns['baroaltitude']]

        # Convert meters to feet
        track[i, columns['baroaltitude']] = track[i, columns['baroaltitude']] * 3.28084

        # Substitute negative values from altitude data
        if track[i, columns['baroaltitude']] < 0:
             track[i, columns['baroaltitude']] = track[i - 1,columns['baroaltitude']]
```

```
        # Substitute invalid records from altitude data
        if np.abs(track[i, columns['baroaltitude']] - track[i - 1,columns['baroaltitude']]) > 500
and next_alt:
            track[i, columns['baroaltitude']] = track[i - 1,columns['baroaltitude']]
            next_alt = False
        else:
            next_alt = True

        # Substitute invalid records from ground speed data
        if np.abs( track[i,columns['velocity']] - track[i - 1,columns['velocity']]) > 50 and
next_vel:
            track[i,columns['velocity']] = track[i - 1,columns['velocity']]
            next_vel = False
        else:
            next_vel = True

# %%
# Remove tracks where altitude is greater than 15000 ft at any point
# and the last altitude is greater than 1000 ft
tracks_filtered = [
    track for track in tracks_filtered
    if np.nanmax(track[:, columns['baroaltitude']].astype(float)) <= 15000 and
       float(track[-1, columns['baroaltitude']]) <= 1000
]

# %%
title = f"Historical Flight Tracks - Preprocessed Data\n Arriving Trajectories at DCA
Airport\nfrom {min_date.date()} to {max_date.date()}\n"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

for track in tracks_filtered:
    x = np.arange(0, track.shape[0])
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend()

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %%
def get_interpolated_point(xa, xb, ya, yb, za, zb, va, vb, d):

    d_ab = get_great_circle_distance(xa, ya, xb, yb)

    if d_ab <= d:
        x = xb
        y = yb
        z = zb
        v = vb

    else:
```

```
        x = xb + ((d - d_ab) / d_ab) * (xb - xa)
        y = yb + ((d - d_ab) / d_ab) * (yb - ya)
        z = zb + ((d - d_ab) / d_ab) * (zb - za)
        v = vb + ((d - d_ab) / d_ab) * (vb - va)

    return x, y, z

# %%
# Add a new column for distance_between_points
tracks_filtered = [
    np.hstack([track, np.zeros((track.shape[0], 1))])
    for track in tracks_filtered
]

# %%
for track in tracks_filtered:
    track_length = 0
    track[0, columns['distance']] = 0
    for i in range(1, len(track)):
        lon1 = track[i-1, columns['lon']]
        lat1 = track[i-1, columns['lat']]
        lon2 = track[i, columns['lon']]
        lat2 = track[i, columns['lat']]
        d = get_great_circle_distance(lon1, lat1, lon2, lat2)
        track_length += d
        track[i, columns['distance_between_points']] = d
        track[i, columns['distance']] = track_length # Cumulative distance

# %%
num_segments = 200
standardized_tracks = []

for track in tracks_filtered:
    # the total length of the track
    track_length = track[-1, columns['distance']]

    # The length of the segment for the track
    d_desired = track_length / num_segments

    # Declare new ndarray
    standardized_track = np.zeros((num_segments, len(columns)), dtype=object)

    # Save initilal point
    standardized_track[0,:] = track[0,:]

    # First distance:
    accumulated_distance = d_desired

    index = 1
    for segment_index in range(1, num_segments):
        for i in range(index, len(track) - 1):
            if track[i - 1, columns['distance']] <= accumulated_distance and track[i,
columns['distance']] > accumulated_distance:

                # Coordinates of previous and current point
                xa, ya, za, va = track[i - 1, columns['lon']], track[i - 1, columns['lat']],
track[i - 1, columns['baroaltitude']], track[i - 1, columns['velocity']]
                xb, yb, zb, vb = track[i, columns['lon']], track[i, columns['lat']], track[i,
columns['baroaltitude']], track[i, columns['velocity']]
                x, y, z = get_interpolated_point(xa, xb, ya, yb, za, zb, va, vb, d_desired)

                index = i - 1
                break

        accumulated_distance += d_desired
```

```
        # Save new point
        standardized_track[segment_index,:] = np.copy(track[index,:])
        standardized_track[segment_index, columns['lon']] = x
        standardized_track[segment_index, columns['lat']] = y
        standardized_track[segment_index, columns['baroaltitude']] = z
        standardized_track[segment_index, columns['distance']] = accumulated_distance

    # Save last point
    standardized_track[-1,:] = track[-1,:].copy()

    # Add to tracks list
    standardized_tracks.append(standardized_track)

# %%
title = f"Historical Flight Tracks - Standardized Data\n Arriving Trajectories at DCA
Airport\nfrom {min_date.date()} to {max_date.date()}\n"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

# Generate an array of segment indices (0 to num_segments-1)
x = np.arange(num_segments)

for track in standardized_tracks:
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend()

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %%
# Remove helicopter traffic
standardized_tracks = [
    track for track in standardized_tracks
    if not np.any(track[100:125, columns['baroaltitude']].astype(float) <= 2000) and
       not np.any(track[125:150, columns['baroaltitude']].astype(float) <= 1500) and
       not np.any(track[150:190, columns['baroaltitude']].astype(float) >= 8000) and
       float(track[0, columns['baroaltitude']]) > 8000 and
       float(track[-1, columns['distance']]) < 100
]

# %%
title = f"Historical Flight Tracks - Standardized Data\n Arriving Trajectories at DCA
Airport\nfrom {min_date.date()} to {max_date.date()}\n"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

# Generate an array of segment indices (0 to num_segments-1)
x = np.arange(num_segments)

for track in standardized_tracks:
```

```
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend()

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %% [markdown]
# Save to .csv files:

# %%
file_path = output_dir + f'DCA_Arrivals.csv'
file_arrivals = open(file_path, 'a')

headers =
"time,icao24,lat,lon,velocity,heading,vertrate,onground,baroaltitude,flight_id,distance_between_po
ints,accumulated_distance,departure_airport,arrival_airport"

if os.path.exists(file_path) and os.path.getsize(file_path) == 0:
    file_arrivals.write(headers)

for track in standardized_tracks:
    for i in range(len(track)):
        line = f"\n{str(track[i, columns['time']])},{track[i, columns['icao24']]},{track[i,
columns['lat']]},{track[i, columns['lon']]},\
            {track[i, columns['velocity']]},{track[i, columns['heading']]},{track[i,
columns['vertrate']]},{track[i, columns['onground']]},\
                {track[i, columns['baroaltitude']]},{track[i, columns['flight_id']]},{track[i,
columns['distance_between_points']]},\
                        {track[i, columns['distance']]},{track[i,
columns['departure_airport']]},{track[i, columns['arrival_airport']]}"
        file_arrivals.write(line)
file_arrivals.close()
```

Data Clustering:

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from datetime import datetime
from mpl_toolkits.mplot3d import Axes3D

# %%
output_dir = "C:\\Files\\"
file_path = os.path.join(output_dir, "DCA_Arrivals.csv") # File name

# %%
data = pd.read_csv(file_path)

# %%
columns = {'time':0, 'icao24':1, 'lat':2, 'lon':3, 'velocity':4, 'heading':5, 'vertrate':6,
'onground':7, 'baroaltitude':8, 'flight_id':9, 'distance_between_points':10,
'accumulated_distance':11, 'departure_airport':12, 'arrival_airport':13}

# %%
# Get min and max dates
min_date = data['time'].min()
max_date = data['time'].max()

min_date_str = datetime.strptime(min_date, '%Y-%m-%d %H:%M:%S').strftime('%B %d, %Y')
max_date_str = datetime.strptime(max_date, '%Y-%m-%d %H:%M:%S').strftime('%B %d, %Y')

print("Min date:", min_date)
print("Max date:", max_date)

# %%
# Convert 'time' column to datetime
data['time'] = pd.to_datetime(data['time'])

# Extract unique dates
unique_dates = data['time'].dt.date.unique()

# Extract unique months
unique_months =  data['time'].dt.to_period('M').unique()

# Convert to string format
unique_months_str = [str(month) for month in unique_months]

# Count unique flight IDs per day
unique_tracks_per_day = data.groupby(data['time'].dt.date)['flight_id'].nunique().to_numpy()

# Count unique flight IDs per month
unique_tracks_per_month = data.groupby(data['time'].dt.to_period('M'))['flight_id'].nunique()

# %%
data['flight_id'] = data['flight_id'].astype(str)

# Sort once globally by flight_id and time
data_sorted = data.sort_values(by=['flight_id', 'time']).reset_index(drop=True)

# Generate an array of segment indices
num_segments = 200
x = np.arange(num_segments)

# Group by flight_id
```

```
tracks = []
for _, group in data_sorted.groupby('flight_id'):
    if len(group) == num_segments:
        tracks.append(group.to_numpy(dtype=object))

# %%
dca = {'lat':38.8514403, 'lon':-77.0377214}
bwi = {'lat':39.1757283, 'lon':-76.6689911}

# %%
title = f"Postprocessed Historical Flight Track Data ({len(tracks)} Tracks)\nArriving Trajectories
at DCA Airport\n{min_date_str} - {max_date_str}\n"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

for track in tracks:
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend(loc='lower right')

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %%
# Remove outliers
tracks = [
    track for track in tracks
    if not np.any(track[190:, columns['velocity']].astype(float) >= 200) and
       not np.any(track[65:100, columns['velocity']].astype(float) >= 400) and
       not np.any(track[75:100, columns['baroaltitude']].astype(float) <= 2000) and
       not np.any(track[180:, columns['baroaltitude']].astype(float) >= 3500) and
       not np.any(track[40:, columns['baroaltitude']].astype(float) >= 13500) and
       not np.any(track[100:, columns['baroaltitude']].astype(float) >= 12000)
]

# %%
title = f"Postprocessed Historical Flight Track Data ({len(tracks)} Tracks)\nArriving Trajectories
at DCA Airport\n{min_date_str} - {max_date_str}\n"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

for track in tracks:
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend(loc='lower right')
```

```
ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %%
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10,5), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
title = f'Number of Historical Flight Tracks Collected at DCA Airport\nTotal = {len(tracks)}
Tracks'

ax.bar(unique_dates, unique_tracks_per_day)
ax.set_title(title, weight='bold')
ax.set_xlabel('Day')
ax.set_ylabel('Count')

plt.show()

# %%
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(5, 5), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)

bars = []
# Create bars dynamically for all months
for i in range(len(unique_months_str) - 1):
    bars.append(ax.bar(unique_months_str[i], unique_tracks_per_month[i]))

# Add value labels on top of each bar
for bar_group in bars:
    for rect in bar_group:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2, height,
                str(height), ha='center', va='bottom', fontsize=10, weight='bold')

ax.set_title(title, weight='bold')
ax.set_xlabel('Month')
ax.set_ylabel('Count')

plt.show()

# %%
distances =  np.zeros(len(tracks))

for i in range(len(tracks)):
    distances[i] = tracks[i][-1, columns['accumulated_distance']]

plt.hist(distances, bins=50, edgecolor='black')
title = f"Along Track Distance for Historical Flight Tracks"
plt.title(title, weight='bold')
plt.xlabel('Distance, nautical miles')
plt.ylabel('Frequency')
plt.show()

# %%
first_positions = np.array([(float(track[0, columns['lat']]), float(track[0, columns['lon']])) for
track in tracks])
last_positions = np.array([(float(track[-1, columns['lat']]), float(track[-1, columns['lon']]))
for track in tracks])

# Get max latitude and max longitude
max_lat = np.max(last_positions[:, 0])
max_lon = np.max(last_positions[:, 1])
```

```
print(f"Max Latitude: {max_lat}")
print(f"Max Longitude: {max_lon}")

title = "The First and Last Recorded Positions from Each Flight Track"
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

ax[0].scatter(first_positions[:, 1], first_positions[:, 0], color='purple', label='First
Position')
ax[0].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')
ax[0].set_title('The First Recorded Position')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend()

ax[1].scatter(last_positions[:, 1], last_positions[:, 0], color='orange', label='Last Position')
ax[1].plot(dca['lon'], dca['lat'], 'ko', markersize=10, label='DCA')
ax[1].set_title('The Last Recorded Position')
ax[1].set_xlabel('Longitude')
ax[1].set_ylabel('Latitude')
ax[1].legend()

# %%
positions = [190,192,194,196,198,199]

rows = len(tracks)
cols = len(positions)
position = np.zeros((rows, 2 * cols)) # Two columns required for lat and lon

# Cluster trajectories based on recorded position (lat, lon) for positions:
for row in range(rows):
    for col in range(cols):
        position[row, col*2:col*2+2] = tracks[row][positions[col],[columns['lat'],
columns['lon']]]

# Create DBSCAN model
model = DBSCAN(eps=0.01, min_samples=5)
model.fit(position)
cluster_labels = model.labels_
num_clusters = len(set(cluster_labels)) - (1 if -1 in cluster_labels else 0)
print("Number of clusters:", num_clusters)

# Plot
title = "DBSCAN Clusters"
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

ax.scatter(position[:, 1], position[:, 0], c=cluster_labels, cmap='rainbow')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')

# %%
# Put tracks into different clusters:
clusters = [[] for _ in range(num_clusters)];
for i in range(len(tracks)):
    clusters[cluster_labels[i]].append(tracks[i])

# Check how many tracks in each cluster:
num_tracks_cluster = [len(cluster) for cluster in clusters]
print(f'The number of tracks in each cluster is {num_tracks_cluster}. The total number is
{sum(num_tracks_cluster)}')
```

```
# %%
# Runway thresholds
rwy_01 = {'lat': 38.8415682, 'lon': -77.0367484}
rwy_04 = {'lat': 38.8411290, 'lon': -77.0414542}
rwy_15 = {'lat': 38.8611614, 'lon': -77.0432970}
rwy_19 = {'lat': 38.861193, 'lon': -77.038730}
rwy_22 = {'lat': 38.8535110, 'lon': -77.0338808}
rwy_33 = {'lat': 38.8498039, 'lon': -77.0322089}

# %%
title = "Historical Flight Tracks in Runway Clusters (only last 10 data points per track are
shown)"
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(12, 15), tight_layout=True)
fig.subplots_adjust(hspace=0.4)
fig.suptitle(title, weight='bold', fontsize=14)

# Flatten axes for easy indexing
axes = axes.flatten()

for i in range(len(clusters)):

    ax = axes[i]

    # Plot tracks
    for track in clusters[i]:
        ax.plot(track[190:, columns['lon']],track[190:, columns['lat']], color='grey')

    # Plot runways
    ax.plot([rwy_01['lon'], rwy_19['lon']], [rwy_01['lat'], rwy_19['lat']], color='black',
linewidth=2)
    ax.plot([rwy_15['lon'], rwy_33['lon']], [rwy_15['lat'], rwy_33['lat']], color='black',
linewidth=2)
    ax.plot([rwy_04['lon'], rwy_22['lon']], [rwy_04['lat'], rwy_22['lat']], color='black',
linewidth=2)

    # Plot runway thresholds
    ax.plot(rwy_01['lon'], rwy_01['lat'], 'X', color='darkorange', markersize=12,
markeredgecolor='black', label='RWY 01')
    ax.plot(rwy_04['lon'], rwy_04['lat'], 'X', color='m', markersize=12, markeredgecolor='black',
label='RWY 04')
    ax.plot(rwy_15['lon'], rwy_15['lat'], 'X', color='blue', markersize=12,
markeredgecolor='black', label='RWY 15')
    ax.plot(rwy_19['lon'], rwy_19['lat'], 'X', color='black', markersize=12,
markeredgecolor='black', label='RWY 19')
    ax.plot(rwy_22['lon'], rwy_22['lat'], 'X', color='green', markersize=12,
markeredgecolor='black', label='RWY 22')
    ax.plot(rwy_33['lon'], rwy_33['lat'], 'X', color='red', markersize=12,
markeredgecolor='black', label='RWY 33')

    ax.set_title(f'Cluster #{i}\n{len(clusters[i])} tracks')
    ax.set_ylabel('Latitude')
    ax.set_xlabel('Longitude')
    ax.legend()

# %%
runway_19 = clusters[0]
runway_01 = clusters[1]
runway_33 = clusters[3]
runway_15 = clusters[2] + clusters[4]

# %% [markdown]
# Save to .csv files:

# %%
runway_data = runway_19
```

```
file_path = output_dir + f'RWY_19.csv'
file_runway = open(file_path, 'a')

headers =
"time,icao24,lat,lon,velocity,heading,vertrate,onground,baroaltitude,flight_id,distance_between_po
ints,accumulated_distance,departure_airport,arrival_airport"
file_runway.write(headers)

for track in runway_data:
    for i in range(len(track)):
        line = f"\n{str(track[i, columns['time']])},{track[i, columns['icao24']]},{track[i,
columns['lat']]},{track[i, columns['lon']]},\
            {track[i, columns['velocity']]},{track[i, columns['heading']]},{track[i,
columns['vertrate']]},{track[i, columns['onground']]},\
                {track[i, columns['baroaltitude']]},{str(track[i,
columns['flight_id']])},{track[i, columns['distance_between_points']]},\
                    {track[i, columns['accumulated_distance']]},{track[i,
columns['departure_airport']]},{track[i, columns['arrival_airport']]}"
        file_runway.write(line)
file_runway.close()
```

Runway Data Clustering:

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from mpl_toolkits.mplot3d import Axes3D
from datetime import datetime
import math

# %%
output_dir = "C:\\Files\\"
file_path = os.path.join(output_dir, "RWY_19.csv") # File name

# %%
data = pd.read_csv(file_path)

# %%
columns = {'time':0, 'icao24':1, 'lat':2, 'lon':3, 'velocity':4, 'heading':5, 'vertrate':6,
'onground':7, 'baroaltitude':8, 'flight_id':9, 'distance_between_points':10,
'accumulated_distance':11, 'departure_airport':12, 'arrival_airport':13}

# %%
# Get min and max dates
min_date = data['time'].min()
max_date = data['time'].max()

min_date_str = datetime.strptime(min_date, '%Y-%m-%d %H:%M:%S').strftime('%B %d, %Y')
max_date_str = datetime.strptime(max_date, '%Y-%m-%d %H:%M:%S').strftime('%B %d, %Y')

print("Min date:", min_date)
print("Max date:", max_date)

# %%
# Convert 'time' column to datetime
data['time'] = pd.to_datetime(data['time'])

# Extract unique dates
unique_dates = data['time'].dt.date.unique()

# Extract unique months
unique_months =  data['time'].dt.to_period('M').unique()

# Convert to string format
unique_months_str = [str(month) for month in unique_months]

# Count unique flight IDs per day
unique_tracks_per_day = data.groupby(data['time'].dt.date)['flight_id'].nunique().to_numpy()

# Count unique flight IDs per month
unique_tracks_per_month = data.groupby(data['time'].dt.to_period('M'))['flight_id'].nunique()

# %%
# Sort once globally by flight_id and time
data_sorted = data.sort_values(by=['flight_id', 'time']).reset_index(drop=True)

# Group by flight_id

tracks = []
for _, group in data_sorted.groupby('flight_id'):
    if len(group) == 200:
        tracks.append(group.to_numpy())
```

```
# %%
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10,5), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
title = f'Number of Historical Flight Tracks Collected for Runway 19 at DCA Airport\nTotal =
{len(tracks)} Tracks'

ax.bar(unique_dates, unique_tracks_per_day)
ax.set_title(title, weight='bold')
ax.set_xlabel('Day')
ax.set_ylabel('Count')

plt.show()

# %%
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(5,5), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)

bars = []
# Create bars dynamically for all months
for i in range(len(unique_months_str) - 1):
    bars.append(ax.bar(unique_months_str[i], unique_tracks_per_month[i]))

# Add value labels on top of each bar
for i, bar_group in enumerate(bars):
    for rect in bar_group:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2, height,
                str(height), ha='center', va='bottom', fontsize=10, weight='bold')

ax.set_title(title, weight='bold')
ax.set_xlabel('Month')
ax.set_ylabel('Count')

plt.show()

# %%
runway_threshold = {'lat': 38.861193, 'lon': -77.038730}

# %%
title = f"Historical Flight Tracks arriving on Runway 19 at DCA Airport\n{len(tracks)}
Tracks\n{min_date_str} - {max_date_str}\n"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

# Generate an array of segment indices (0 to num_segments-1)
num_segments = 200
x = np.arange(num_segments)

for track in tracks:
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(runway_threshold['lon'], runway_threshold['lat'], 'ko', markersize=10, label='RWY 19')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend(loc='lower right')

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')
```

```
ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %%
title = "Historical Flight Tracks arriving on Runway 19 at DCA Airport\n(only last 50 data points
per track are shown)"
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

for track in tracks:
    ax.plot(track[150:, columns['lon']], track[150:, columns['lat']])
ax.plot(runway_threshold['lon'], runway_threshold['lat'], 'ko', markersize=10, label='RWY 19')
ax.set_title('')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.legend()

plt.show()

# %%
positions = [50, 60, 100, 120]

rows = len(tracks)
cols = len(positions)
position = np.zeros((rows, 2 * cols)) # Two columns required for lat and lon

# Cluster trajectories based on recorded position (lat, lon) for positions:
for row in range(rows):
    for col in range(cols):
        position[row, col*2:col*2+2] = tracks[row][positions[col],[columns['lat'],
columns['lon']]]

# Create DBSCAN model
model = DBSCAN(eps=0.08, min_samples=5)
model.fit(position)
cluster_labels = model.labels_
num_clusters = len(set(cluster_labels)) - (1 if -1 in cluster_labels else 0)
print("Number of clusters:", num_clusters)

# Plot
title = "DBSCAN Clusters"
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

ax.scatter(position[:, 1], position[:, 0], c=cluster_labels, cmap='rainbow')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')

# %%
# Put tracks into different clusters:
clusters = [[] for _ in range(num_clusters)];
for i in range(len(tracks)):
    clusters[cluster_labels[i]].append(tracks[i])

# Check how many tracks in each cluster:
num_tracks_cluster = [len(cluster) for cluster in clusters]
print(f'The number of tracks in each cluster is {num_tracks_cluster}. The total number is
{sum(num_tracks_cluster)}')

# %%
title = "Runway 19 - Data Clusters"
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(12, 15), tight_layout=True)
```

```
fig.subplots_adjust(hspace=0.4)
fig.suptitle(title, weight='bold', fontsize=14)

# Flatten axes for easy indexing
axes = axes.flatten()

for i in range(len(clusters) - 1):

    ax = axes[i]

    # Plot tracks
    for track in clusters[i]:
        ax.plot(track[0:, columns['lon']],track[0:, columns['lat']])
    ax.plot(runway_threshold['lon'], runway_threshold['lat'], 'X', color='black', markersize=12,
markeredgecolor='black', label='RWY 19')
    ax.set_title(f'Cluster #{i}\n{len(clusters[i])} tracks')
    ax.set_ylabel('Latitude')
    ax.set_xlabel('Longitude')
    ax.legend()

# %%
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 6), tight_layout=True)
fig.subplots_adjust(hspace=0.4)

for track in clusters[-1]:
    ax.plot(track[0:, columns['lon']],track[0:, columns['lat']])
ax.plot(runway_threshold['lon'], runway_threshold['lat'], 'X', color='black', markersize=12,
markeredgecolor='black', label='RWY 19')
ax.set_title(f'Unclassified\n{len(clusters[-1])} tracks')
ax.set_ylabel('Latitude')
ax.set_xlabel('Longitude')
ax.legend()

# %%
cluster_data = clusters[1] # North

file_path = output_dir + f'RWY_19_Cluster_North.csv'
file_runway = open(file_path, 'a')

headers =
"time,icao24,lat,lon,velocity,heading,vertrate,onground,baroaltitude,flight_id,distance_between_po
ints,accumulated_distance,departure_airport,arrival_airport"
file_runway.write(headers)

for track in cluster_data:
    for i in range(len(track)):
        line = f"\n{str(track[i, columns['time']])},{track[i, columns['icao24']]},{track[i,
columns['lat']]},{track[i, columns['lon']]},\
            {track[i, columns['velocity']]},{track[i, columns['heading']]},{track[i,
columns['vertrate']]},{track[i, columns['onground']]},\
                {track[i, columns['baroaltitude']]},{track[i, columns['flight_id']]},{track[i,
columns['distance_between_points']]},\
                        {track[i, columns['accumulated_distance']]},{track[i,
columns['departure_airport']]},{track[i, columns['arrival_airport']]}"
        file_runway.write(line)
file_runway.close()
```

Cluster Analysis:

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import radians, sin, cos, acos
from mpl_toolkits.mplot3d import Axes3D
import math
from datetime import datetime
from collections import defaultdict
import csv

# %%
output_dir = "C:\\Files\\"
file_path = os.path.join(output_dir, "RWY_19_Cluster_North.csv") # File name
file_path_seeds = os.path.join(output_dir, "RWY_19_Northern_Cluster_Seed_Tracks_All.csv")
cluster = 'North'

# %%
data = pd.read_csv(file_path)

data_seeds = pd.read_csv(file_path_seeds)
# unique_tracks = data_seeds['flight_id'].unique() # Unique flight IDs
unique_tracks = data['flight_id'].unique() # Unique flight IDs

# %%
columns = {'time':0, 'icao24':1, 'lat':2, 'lon':3, 'velocity':4, 'heading':5, 'vertrate':6,
'onground':7, 'baroaltitude':8, 'flight_id':9, 'distance_between_points':10,
'accumulated_distance':11, 'departure_airport':12, 'arrival_airport':13, 'duration':14,
'lateral_dispersion':15, 'course_change':16, 'magnitude_course_change':17,
'cumulative_course_change':18, 'deceleration':19, 'magnitude_deceleration':20,
'cumulative_deceleration':21, 'descend':22, 'magnitude_descend':23, 'cumulative_descend':24,
'speed_change':25, 'level_change':26}

# %%
# Sort once globally by flight_id and time
data_sorted = data.sort_values(by=['flight_id', 'time']).reset_index(drop=True)

# Filter to only flight_ids that exist in unique_tracks
data_filtered = data_sorted[data_sorted['flight_id'].isin(unique_tracks)]

# Generate an array of segment indices
num_segments = 200
x = np.arange(num_segments)

# Group by flight_id
tracks = []
for _, group in data_filtered.groupby('flight_id'):
    if len(group) == num_segments:
        tracks.append(group.to_numpy())

# %%
# Add column for new parameters
for idx in range(len(tracks)):
    rows, cols = tracks[idx].shape
    new_cols = np.full((rows, 15), np.zeros)
    tracks[idx] = np.hstack([tracks[idx], new_cols])

# %%
# Get min and max dates
min_date = data['time'].min()
max_date = data['time'].max()
```

```
min_date_str = datetime.strptime(min_date, '%Y-%m-%d %H:%M:%S').strftime('%B %d, %Y')
max_date_str = datetime.strptime(max_date, '%Y-%m-%d %H:%M:%S').strftime('%B %d, %Y')

print("Min date:", min_date)
print("Max date:", max_date)

# %%
runway_treshold = {'lat':38.861193, 'lon':-77.038730}

# %%
title = f"Historical Flight Tracks arriving on Runway 19 from {cluster} Cluster at DCA
Airport\n{len(tracks)} Tracks\n{min_date_str} - {max_date_str}\n"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

for track in tracks:
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY 19')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend(loc='lower right')

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %%
# Ensure all tracks end at runway threshold and have 200 points
tracks_filtered = []
num_segments = 200  # Desired length for all tracks

for track in tracks:
    # Adjust last point to runway threshold
    if track[-1, columns['lat']] <= runway_treshold['lat']:  # Overshoot
        for j in reversed(range(len(track) - 1)):
            if track[j, columns['lat']] > runway_treshold['lat']:
                new_point = track[j].copy()
                new_point[columns['lon']] = runway_treshold['lon']
                new_point[columns['lat']] = runway_treshold['lat']
                track = np.vstack((track[:j], new_point))
                break
    else:  # Undershoot
        track[-1, columns['lon']] = runway_treshold['lon']
        track[-1, columns['lat']] = runway_treshold['lat']

    # Interpolate or truncate to 200 points
    current_len = len(track)
    standardized_track = np.zeros((num_segments, track.shape[1]), dtype=object)

    # Keep first and last points
    standardized_track[0, :] = track[0, :]
    standardized_track[-1, :] = track[-1, :]

    # Interpolate intermediate points
    for i in range(1, num_segments - 1):
```

```
        # Compute proportional index in original track
        idx = int(i * (current_len - 1) / (num_segments - 1))
        standardized_track[i, :] = track[idx, :]

    tracks_filtered.append(standardized_track)

# %%
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)

ax[1].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY 19')

for track in tracks_filtered:
    ax[0].plot(track[0:, columns['lon']], track[0:, columns['lat']])
    ax[1].plot(track[190:, columns['lon']], track[190:, columns['lat']])

ax[0].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY 19')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend()

ax[1].set_title('Flight Tracks ending at the Runway Threshold (last 10 points)')
ax[1].set_xlabel('Longitude')
ax[1].set_ylabel('Latitude')
ax[1].legend()

# %%
time_format = "%Y-%m-%d %H:%M:%S"

for track in tracks_filtered:
    duration = 0
    track[0, columns['duration']] = 0
    for i in range(1, len(track)):

        # Convert strings to datetime
        t1 = datetime.strptime(track[i-1, columns['time']], time_format)
        t2 = datetime.strptime(track[i, columns['time']], time_format)

        # Calculate time difference in minute
        time_diff = (t2 - t1).total_seconds() / 60
        duration += time_diff
        track[i, columns['duration']] = duration # Cumulative time

# %%
def get_great_circle_distance(lon1, lat1, lon2, lat2):
    r = 6371.137 # The Earth's radius in km

    # Convert to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # Compute cosine of central angle
    cos_angle = sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) * cos(lon1 - lon2)

    # Clamp the value to avoid math domain error
    cos_angle = min(1, max(-1, cos_angle))

    # Great circle distance in km
    d = r * acos(cos_angle)

    # Convert km to nm:
    d = d * 0.5399568035

    return d
```

```
# %%
for track in tracks_filtered:
    track_length = 0
    track[0, columns['accumulated_distance']] = 0
    for i in range(1, len(track)):
        lon1 = track[i-1, columns['lon']]
        lat1 = track[i-1, columns['lat']]
        lon2 = track[i, columns['lon']]
        lat2 = track[i, columns['lat']]
        d = get_great_circle_distance(lon1, lat1, lon2, lat2)
        track_length += d
        track[i, columns['distance_between_points']] = d
        track[i, columns['accumulated_distance']] = track_length # Cumulative distance

# %%
size = len(tracks)
distances_at_rt =  np.zeros(size)
durations_at_rt = np.zeros(size)
altitudes_at_rt = np.zeros(size)
ground_speeds_at_rt = np.zeros(size)

for i in range(size):
    distances_at_rt[i] = tracks_filtered[i][-1, columns['accumulated_distance']]
    durations_at_rt[i] = tracks_filtered[i][-1, columns['duration']]
    altitudes_at_rt[i] = tracks_filtered[i][-1, columns['baroaltitude']]
    ground_speeds_at_rt[i] = tracks_filtered[i][-1, columns['velocity']]

# %%
title = "Measurements at the Runway Threshold"
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 10), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

axes[0, 0].hist(distances_at_rt, bins=30, edgecolor='black')
axes[0, 0].set_title('Along Track Distance')
axes[0, 0].set_xlabel('Distance, nautical miles')
axes[0, 0].set_ylabel('Frequency')

axes[0, 1].hist(durations_at_rt, bins=30, edgecolor='black', color='orange')
axes[0, 1].set_title('Flight Track Duration')
axes[0, 1].set_xlabel('Time, minutes')
axes[0, 1].set_ylabel('Frequency')

axes[1, 0].hist(altitudes_at_rt, bins=15, edgecolor='black', color='green')
axes[1, 0].set_title('Barometric Altitude')
axes[1, 0].set_xlabel('Baroaltitude, feet')
axes[1, 0].set_ylabel('Frequency')

axes[1, 1].hist(ground_speeds_at_rt, bins=30, edgecolor='black', color='purple')
axes[1, 1].set_title('Ground Speed')
axes[1, 1].set_xlabel('Ground Speed, knots')
axes[1, 1].set_ylabel('Frequency')

plt.show()

# %%
def get_treshold_point(lat, lon, heading, distance):
    r = 6371.137 # The Earth's radius in km

    # Convert to radians
    lat1 = math.radians(lat)
    lon1 = math.radians(lon)
    bearing = math.radians(heading + 180)

    # Angular distance
```

```
    d_r = (distance * 1.852) / r # Convert nm to km

    # Destination latitude
    lat2 = math.asin(math.sin(lat1) * math.cos(d_r) +
                     math.cos(lat1) * math.sin(d_r) * math.cos(bearing))

    # Destination longitude
    lon2 = lon1 + math.atan2(math.sin(bearing) * math.sin(d_r) * math.cos(lat1),
                             math.cos(d_r) - math.sin(lat1) * math.sin(lat2))

    return math.degrees(lat2), math.degrees(lon2)

# %%
def process_tracks_at_distance(tracks_filtered, distance_nm, nominal_index):
    positions = []
    track_data = []

    # Generate circle positions
    for i in range(360):
        lat_nm, lon_nm = get_treshold_point(runway_treshold['lat'], runway_treshold['lon'], i,
distance_nm)
        positions.append([i, lat_nm, lon_nm])
        if i == nominal_index:
            nominal_lat = lat_nm
            nominal_lon = lon_nm

    first = True

    # Process each track
    for track in tracks_filtered:
        for i in reversed(range(len(track) - 2)):
            d1 = get_great_circle_distance(track[i, columns['lon']], track[i, columns['lat']],
                                           runway_treshold['lon'], runway_treshold['lat'])
            d2 = get_great_circle_distance(track[i - 1, columns['lon']], track[i - 1,
columns['lat']],
                                           runway_treshold['lon'], runway_treshold['lat'])

            if d1 <= distance_nm and d2 > distance_nm:
                # Find closest point on the circle
                closest_point = min(positions, key=lambda pos: get_great_circle_distance(
                    track[i - 1, columns['lon']], track[i - 1, columns['lat']], pos[2], pos[1]))
                distance = get_great_circle_distance(track[i - 1, columns['lon']], track[i - 1,
columns['lat']],
                                                     closest_point[2], closest_point[1])

                if first:
                    nominal_alt = track[i, columns['baroaltitude']]
                    first = False

                # Create threshold value array
                treshold_value = np.zeros((track.shape[1]))
                treshold_value[columns['lat']] = closest_point[1]
                treshold_value[columns['lon']] = closest_point[2]
                treshold_value[columns['baroaltitude']] = track[i, columns['baroaltitude']]
                treshold_value[columns['velocity']] = track[i, columns['velocity']]
                treshold_value[columns['accumulated_distance']] += distance
                treshold_value[columns['lateral_dispersion']] = get_great_circle_distance(
                    track[i - 1, columns['lon']], track[i - 1, columns['lat']], nominal_lon,
nominal_lat)

                track_data.append(treshold_value)
                break

    return positions, track_data
```

```
# Call for 3NM, 6NM, 9NM
positions_at_3nm, track_data_at_3nm = process_tracks_at_distance(tracks_filtered, 3, 147)
positions_at_6nm, track_data_at_6nm = process_tracks_at_distance(tracks_filtered, 6, 140)
positions_at_9nm, track_data_at_9nm = process_tracks_at_distance(tracks_filtered, 9, 136)

# %%
title = "Measurements at 3NM from the Runway Threshold"
fig, axes = plt.subplots(nrows=3, ncols=1, figsize=(6, 15), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

track_data = np.array(track_data_at_3nm)

altitude_3nm = np.zeros(size)
ground_speed_3nm = np.zeros(size)
lateral_dispersion_3nm = np.zeros(size)

for i in range(len(tracks)):
    altitude_3nm[i] = track_data[i, columns['baroaltitude']]
    ground_speed_3nm[i] = track_data[i, columns['velocity']]
    lateral_dispersion_3nm[i] = track_data[i, columns['lateral_dispersion']]

axes[0].hist(lateral_dispersion_3nm, bins=20, edgecolor='black')
axes[0].set_title('Lateral Dispersion')
axes[0].set_xlabel('Distance, nautical miles')
axes[0].set_ylabel('Frequency')

axes[1].hist(altitude_3nm, bins=25, edgecolor='black', color='green')
axes[1].set_title('Vertical Dispersion')
axes[1].set_xlabel('Baroaltitude, feet')
axes[1].set_ylabel('Frequency')

axes[2].hist(ground_speed_3nm, bins=30, edgecolor='black', color='purple')
axes[2].set_title('Speed Dispersion')
axes[2].set_xlabel('Ground Speed, knots')
axes[2].set_ylabel('Frequency')

plt.show()

# %%
title = "Measurements at 6NM from the Runway Threshold"
fig, axes = plt.subplots(nrows=3, ncols=1, figsize=(6, 15), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

track_data = np.array(track_data_at_6nm)
size = len(tracks)

altitude_6nm = np.zeros(size)
ground_speed_6nm = np.zeros(size)
lateral_dispersion_6nm = np.zeros(size)

for i in range(len(tracks)):
    altitude_6nm[i] = track_data[i, columns['baroaltitude']]
    ground_speed_6nm[i] = track_data[i, columns['velocity']]
    lateral_dispersion_6nm[i] = track_data[i, columns['lateral_dispersion']]

axes[0].hist(lateral_dispersion_6nm, bins=20, edgecolor='black')
axes[0].set_title('Lateral Dispersion')
axes[0].set_xlabel('Distance, nautical miles')
axes[0].set_ylabel('Frequency')

axes[1].hist(altitude_6nm, bins=25, edgecolor='black', color='green')
axes[1].set_title('Vertical Dispersion')
axes[1].set_xlabel('Baroaltitude, feet')
axes[1].set_ylabel('Frequency')
```

```
axes[2].hist(ground_speed_6nm, bins=30, edgecolor='black', color='purple')
axes[2].set_title('Speed Dispersion')
axes[2].set_xlabel('Ground Speed, knots')
axes[2].set_ylabel('Frequency')

plt.show()

# %%
title = "Measurements at 9NM from the Runway Threshold"
fig, axes = plt.subplots(nrows=3, ncols=1, figsize=(6, 15), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

track_data = np.array(track_data_at_9nm)
size = len(tracks)

altitude_9nm = np.zeros(size)
ground_speed_9nm = np.zeros(size)
lateral_dispersion_9nm = np.zeros(size)

for i in range(len(tracks)):
    altitude_9nm[i] = track_data[i, columns['baroaltitude']]
    ground_speed_9nm[i] = track_data[i, columns['velocity']]
    lateral_dispersion_9nm[i] = track_data[i, columns['lateral_dispersion']]

axes[0].hist(lateral_dispersion_9nm, bins=20, edgecolor='black')
axes[0].set_title('Lateral Dispersion')
axes[0].set_xlabel('Distance, nautical miles')
axes[0].set_ylabel('Frequency')

axes[1].hist(altitude_9nm, bins=25, edgecolor='black', color='green')
axes[1].set_title('Vertical Dispersion')
axes[1].set_xlabel('Baroaltitude, feet')
axes[1].set_ylabel('Frequency')

axes[2].hist(ground_speed_9nm, bins=30, edgecolor='black', color='purple')
axes[2].set_title('Speed Dispersion')
axes[2].set_xlabel('Ground Speed, knots')
axes[2].set_ylabel('Frequency')

plt.show()

# %%
title = "Measurements at 3-6-9 NM from the Runway Threshold\n"
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(20, 18), tight_layout=False)
fig.suptitle(title, weight='bold', fontsize=14)
fig.subplots_adjust(hspace=0.01, wspace=0.4)

axes[0, 0].hist(lateral_dispersion_3nm, bins=20, edgecolor='black')
axes[0, 0].set_title('Lateral Dispersion at 3 NM from the RT')
axes[0, 0].set_xlabel('Distance, nautical miles')
axes[0, 0].set_ylabel('Frequency')

axes[0, 1].hist(altitude_3nm, bins=25, edgecolor='black', color='green')
axes[0, 1].set_title('Vertical Dispersion at 3 NM from the RT')
axes[0, 1].set_xlabel('Baroaltitude, feet')
axes[0, 1].set_ylabel('Frequency')

axes[0, 2].hist(ground_speed_3nm, bins=30, edgecolor='black', color='purple')
axes[0, 2].set_title('Speed Dispersion at 3 NM from the RT')
axes[0, 2].set_xlabel('Ground Speed, knots')
axes[0, 2].set_ylabel('Frequency')

axes[1, 0].hist(lateral_dispersion_6nm, bins=20, edgecolor='black')
axes[1, 0].set_title('Lateral Dispersion at 6 NM from the RT')
axes[1, 0].set_xlabel('Distance, nautical miles')
```

```
axes[1, 0].set_ylabel('Frequency')

axes[1, 1].hist(altitude_6nm, bins=25, edgecolor='black', color='green')
axes[1, 1].set_title('Vertical Dispersion at 6 NM from the RT')
axes[1, 1].set_xlabel('Baroaltitude, feet')
axes[1, 1].set_ylabel('Frequency')

axes[1, 2].hist(ground_speed_6nm, bins=30, edgecolor='black', color='purple')
axes[1, 2].set_title('Speed Dispersion at 6 NM from the RT')
axes[1, 2].set_xlabel('Ground Speed, knots')
axes[1, 2].set_ylabel('Frequency')

axes[2, 0].hist(lateral_dispersion_9nm, bins=20, edgecolor='black')
axes[2, 0].set_title('Lateral Dispersion at 9 NM from the RT')
axes[2, 0].set_xlabel('Distance, nautical miles')
axes[2, 0].set_ylabel('Frequency')

axes[2, 1].hist(altitude_9nm, bins=25, edgecolor='black', color='green')
axes[2, 1].set_title('Vertical Dispersion at 9 NM from the RT')
axes[2, 1].set_xlabel('Baroaltitude, feet')
axes[2, 1].set_ylabel('Frequency')

axes[2, 2].hist(ground_speed_9nm, bins=30, edgecolor='black', color='purple')
axes[2, 2].set_title('Speed Dispersion at 9 NM from the RT')
axes[2, 2].set_xlabel('Ground Speed, knots')
axes[2, 2].set_ylabel('Frequency')

plt.show()

# %%
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(30,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)

for i, track in enumerate(tracks_filtered[0:500]):
    ax.plot(track[0:10, columns['lon']], track[0:10, columns['lat']])

    x = track[0, columns['lon']]
    y = track[0, columns['lat']]
    ax.text(x, y + 0.002, str(i), ha='center', fontsize=8, color='black', rotation=90)

ax.set_title('Flight Tracks Initial Position (first 10 points)')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')

# %%
lateral_dispersion_data = np.zeros(size)
altitude_data = np.zeros(size)
ground_speed_data = np.zeros(size)

nominal_lat = tracks[78][0, columns['lat']] # 78
nominal_lon = tracks[78][0, columns['lon']] # 78

for i, track in enumerate(tracks_filtered):
    lateral_dispersion = get_great_circle_distance(track[0, columns['lon']], track[0,
columns['lat']], nominal_lon, nominal_lat)
    lateral_dispersion_data[i] = lateral_dispersion
    altitude_data[i] = track[0, columns['baroaltitude']]
    ground_speed_data[i] = track[0, columns['velocity']]

# %%
title = "Measurements at the Origin"
fig, axes = plt.subplots(nrows=3, ncols=1, figsize=(6, 15), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

axes[0].hist(lateral_dispersion_data, bins=40, edgecolor='black')
```

```
axes[0].set_title('Lateral Dispersion')
axes[0].set_xlabel('Distance, nautical miles')
axes[0].set_ylabel('Frequency')

axes[1].hist(altitude_data, bins=50, edgecolor='black', color='green')
axes[1].set_title('Vertical Dispersion')
axes[1].set_xlabel('Baroaltitude, feet')
axes[1].set_ylabel('Frequency')

axes[2].hist(ground_speed_data, bins=30, edgecolor='black', color='purple')
axes[2].set_title('Speed Dispersion')
axes[2].set_xlabel('Ground Speed, knots')
axes[2].set_ylabel('Frequency')

plt.show()

# %%
for track in tracks_filtered:

    track[0, columns['course_change']] = False
    track[0, columns['deceleration']] = False
    track[0, columns['descend']] = False
    track[0, columns['speed_change']] = False
    track[0, columns['level_change']] = False
    track[0, columns['magnitude_course_change']] = 0
    track[0, columns['magnitude_deceleration']] = 0
    track[0, columns['magnitude_descend']] = 0
    track[0, columns['cumulative_course_change']] = 0
    track[0, columns['cumulative_deceleration']] = 0
    track[0, columns['cumulative_descend']] = 0

    for i in range(1, len(track)):

        # Course Changes:
        if track[i - 1, columns['heading']] - track[i, columns['heading']] > 5:
            track[i, columns['course_change']] = True
            track[i, columns['magnitude_course_change']] = track[i - 1, columns['heading']] -
track[i, columns['heading']]

            if track[i - 1, columns['magnitude_course_change']] != 0:
                track[i, columns['cumulative_course_change']] = track[i - 1,
columns['cumulative_course_change']] + track[i, columns['magnitude_course_change']]
            elif i > 2 and track[i - 2, columns['magnitude_course_change']] != 0 and track[i - 1,
columns['magnitude_course_change']] == 0:
                track[i - 1, columns['cumulative_course_change']] = track[i - 2,
columns['cumulative_course_change']]
                track[i, columns['cumulative_course_change']] = track[i - 2,
columns['cumulative_course_change']] + track[i, columns['magnitude_course_change']]
            else:
                track[i, columns['cumulative_course_change']] = track[i,
columns['magnitude_course_change']]
        else:
            track[i, columns['course_change']] = False
            track[i, columns['magnitude_course_change']] = 0
            track[i, columns['cumulative_course_change']] = 0

        # Constant Speed or Constant Deceleration Segment:
        if track[i - 1, columns['velocity']] - track[i, columns['velocity']] > 2:

            if track[i - 1, columns['deceleration']] == False:
                track[i, columns['speed_change']] = True
            else:
                track[i, columns['speed_change']] = False

            track[i, columns['deceleration']] = True
```

```
            track[i, columns['magnitude_deceleration']] = track[i - 1, columns['velocity']] -
track[i, columns['velocity']]

            if track[i - 1, columns['magnitude_deceleration']] != 0:
                track[i, columns['cumulative_deceleration']] = track[i - 1,
columns['cumulative_deceleration']] + track[i, columns['magnitude_deceleration']]
            elif i > 2 and track[i - 2, columns['magnitude_deceleration']] != 0 and track[i - 1,
columns['magnitude_deceleration']] == 0:
                track[i, columns['cumulative_deceleration']] = track[i - 2,
columns['cumulative_deceleration']] + track[i, columns['magnitude_deceleration']]
                track[i - 1, columns['cumulative_deceleration']] = track[i - 2,
columns['cumulative_deceleration']]
            else:
                track[i, columns['cumulative_deceleration']] = track[i,
columns['magnitude_deceleration']]
        else:
            track[i, columns['deceleration']] = False
            track[i, columns['speed_change']] = False
            track[i, columns['magnitude_deceleration']] = 0
            track[i, columns['cumulative_deceleration']] = 0

        # Level Flight or Descending Segment:
        if track[i - 1, columns['baroaltitude']] - track[i, columns['baroaltitude']] > 25:

            if track[i - 1, columns['descend']] == False:
                track[i, columns['level_change']] = True
            else:
                track[i, columns['level_change']] = False

            track[i, columns['descend']] = True
            track[i, columns['magnitude_descend']] = track[i - 1, columns['baroaltitude']] -
track[i, columns['baroaltitude']]

            if track[i - 1, columns['magnitude_descend']] != 0:
                track[i, columns['cumulative_descend']] = track[i - 1,
columns['cumulative_descend']] + track[i, columns['magnitude_descend']]
            elif i > 2 and track[i - 2, columns['magnitude_descend']] != 0 and track[i - 1,
columns['magnitude_descend']] == 0:
                track[i, columns['cumulative_descend']] = track[i - 2,
columns['cumulative_descend']] + track[i, columns['magnitude_descend']]
                track[i - 1, columns['cumulative_descend']] = track[i - 2,
columns['cumulative_descend']]
            else:
                track[i, columns['cumulative_descend']] = track[i, columns['magnitude_descend']]
        else:
            track[i, columns['descend']] = False
            track[i, columns['level_change']] = False
            track[i, columns['magnitude_descend']] = 0
            track[i, columns['cumulative_descend']] = 0

# %%
course_change_count_data = np.zeros(size)
course_change_magnitude_list = []

for idx, track in enumerate(tracks_filtered):

    course_change_count = 0
    course_change_magnitude = 0

    for i in range(1, len(track)):
        if track[i, columns['course_change']] == False and track[i - 1, columns['course_change']]
== True:
            course_change_count += 1
            course_change_magnitude = track[i - 1, columns['cumulative_course_change']]
            course_change_magnitude_list.append(course_change_magnitude)
```

```
    course_change_count_data[idx] = course_change_count

course_change_magnitude_data = np.array(course_change_magnitude_list)

# %%
title = "Course Change Segments"
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

axes[0].hist(course_change_count_data, bins=10, edgecolor='black')
axes[0].set_title('Number of Course Changes per Track')
axes[0].set_xlabel('Count')
axes[0].set_ylabel('Frequency')

axes[1].hist(course_change_magnitude_data, bins=50, edgecolor='black')
axes[1].set_title('Magnitude of Course Change Segments')
axes[1].set_xlabel('Degrees')
axes[1].set_ylabel('Frequency')

# %%
speed_change_count_data = np.zeros(size)
level_change_count_data = np.zeros(size)

for idx, track in enumerate(tracks_filtered):
    speed_change_count = 0
    level_change_count = 0

    for i in range(1, len(track)):
        if track[i, columns['speed_change']] == True:
            speed_change_count += 1

        if track[i, columns['level_change']] == True:
            level_change_count += 1

    speed_change_count_data[idx] = speed_change_count
    level_change_count_data[idx] = level_change_count

# %%
title = "Speed Change Segments"
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 5), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

ax.hist(speed_change_count_data, bins=25, edgecolor='black', color='purple')
ax.set_title('Number of Speed Changes per Track')
ax.set_xlabel('Count')
ax.set_ylabel('Frequency')

# %%
title = "Level Change Segments"
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 5), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

ax.hist(level_change_count_data, bins=25, edgecolor='black', color='green')
ax.set_title('Number of Level Changes per Track')
ax.set_xlabel('Count')
ax.set_ylabel('Frequency')

# %%
results_dict = {
    'along_track_distance': distances_at_rt,
    'altitude_rt': altitudes_at_rt,
    'ground_speed_rt': ground_speeds_at_rt,
    'lateral_dispersion_3nm': lateral_dispersion_3nm,
    'vertical_dispersion_3nm': altitude_3nm,
```

```
    'speed_dispersion_3nm': ground_speed_3nm,
    'lateral_dispersion_6nm': lateral_dispersion_6nm,
    'vertical_dispersion_6nm': altitude_6nm,
    'speed_dispersion_6nm': ground_speed_6nm,
    'lateral_dispersion_9nm': lateral_dispersion_9nm,
    'vertical_dispersion_9nm': altitude_9nm,
    'speed_dispersion_9nm': ground_speed_9nm,
    'course_change_count': course_change_count_data,
    'speed_change_count_data': speed_change_count_data,
    'level_change_count_data': level_change_count_data
}

# %%
# Prepare header: original metrics
metrics = [k for k, v in results_dict.items() if not isinstance(v, dict)]
header = ['flight_id'] + metrics

# Determine max length for arrays
max_len = max(len(v) for v in results_dict.values() if hasattr(v, '__len__') and not isinstance(v,
dict))

file_path = os.path.join(output_dir, 'Historical_Seed_Tracks_Results_North_Cluster.csv')
with open(file_path, mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(header)

    # Write rows: each row corresponds to an index across arrays
    for i in range(max_len):
        row = [tracks[i][0, columns['flight_id']]]  # Start with Flight ID

        # Add original metrics
        for key in metrics:
            value = results_dict[key]
            if hasattr(value, '__len__'):
                row.append(value[i] if i < len(value) else '')
            else:
                row.append(value)

        writer.writerow(row)

# %%
file_path = output_dir + f'RWY_19_Cluster_Northern_with_New_Parameters.csv'
file = open(file_path, 'w')

headers =
"position,flight_id,time,lat,lon,ground_speed,baroaltitude,vertrate,heading,distance_between_point
s,accumulated_distance,duration,departure_airport,arrival_airport,course_change,magnitude_course_c
hange,cumulative_course_change,deceleration,magnitude_deceleration,cumulative_deceleration,descend
,magnitude_descend,cumulative_descend,speed_change,level_change"
file.write(headers)

for track in tracks_filtered:
    for i in range(len(track)):
        line = f"\n{str(i)},{track[i, columns['flight_id']]},{str(track[i,
columns['time']])},{track[i, columns['lat']]},{track[i, columns['lon']]},{track[i,
columns['velocity']]},{track[i, columns['baroaltitude']]},\
            {track[i, columns['vertrate']]},{track[i, columns['heading']]},{track[i,
columns['distance_between_points']]},{track[i, columns['accumulated_distance']]},\
                {track[i, columns['duration']]},{track[i, columns['departure_airport']]},{track[i,
columns['arrival_airport']]},{track[i, columns['course_change']]},\
                    {track[i, columns['magnitude_course_change']]},{track[i,
columns['cumulative_course_change']]},{track[i, columns['deceleration']]},\
                        {track[i, columns['magnitude_deceleration']]},{track[i,
columns['cumulative_deceleration']]},{track[i, columns['descend']]},\
```

```
                            {track[i, columns['magnitude_descend']]},{track[i,
columns['cumulative_descend']]},{track[i, columns['speed_change']]},{track[i,
columns['level_change']]}"
        file.write(line)
file.close()
```

Waypoint Data:

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import radians, sin, cos, acos
from mpl_toolkits.mplot3d import Axes3D
import math
from datetime import datetime
from collections import defaultdict
import csv

# %%
output_dir = "C:\\Files\\"
file_path = os.path.join(output_dir, "RWY_19_Cluster_North.csv") # File name

# %%
data = pd.read_csv(file_path)

# %%
columns = {'time':0, 'icao24':1, 'lat':2, 'lon':3, 'velocity':4, 'heading':5, 'vertrate':6,
'onground':7, 'baroaltitude':8, 'flight_id':9, 'distance_between_points':10,
'accumulated_distance':11, 'departure_airport':12, 'arrival_airport':13, 'duration':14,
'lateral_dispersion':15, 'course_change':16, 'magnitude_course_change':17,
'cumulative_course_change':18, 'deceleration':19, 'magnitude_deceleration':20,
'cumulative_deceleration':21, 'descend':22, 'magnitude_descend':23, 'cumulative_descend':24,
'speed_change':25, 'level_change':26}

# %%
# Sort once globally by flight_id and time
data_sorted = data.sort_values(by=['flight_id', 'time']).reset_index(drop=True)

# Generate an array of segment indices
num_segments = 200
x = np.arange(num_segments)

# Group by flight_id
tracks = []
for _, group in data_sorted.groupby('flight_id'):
    if len(group) == num_segments:
        tracks.append(group.to_numpy())

# %%
# Get min and max dates
min_date = data['time'].min()
max_date = data['time'].max()

min_date_str = datetime.strptime(min_date, '%Y-%m-%d %H:%M:%S').strftime('%B %d, %Y')
max_date_str = datetime.strptime(max_date, '%Y-%m-%d %H:%M:%S').strftime('%B %d, %Y')

print("Min date:", min_date)
print("Max date:", max_date)

# %%
runway_treshold = {'lat':38.861193, 'lon':-77.038730}

# %%
title = f"Historical Flight Tracks arriving on Runway 19 from Northern Cluster at DCA
Airport\n{len(tracks)} Tracks\n{min_date_str} - {max_date_str}"
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(18,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)
```

```
for track in tracks:
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(x, track[:, columns['baroaltitude']])
    ax[2].plot(x, track[:, columns['velocity']])
ax[0].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY 19')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend(loc='lower right')

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %%
# Ensure all tracks end at runway threshold and have 200 points
tracks_filtered = []
num_segments = 200  # Desired length for all tracks

for track in tracks:
    # Adjust last point to runway threshold
    if track[-1, columns['lat']] <= runway_treshold['lat']:  # Overshoot
        for j in reversed(range(len(track) - 1)):
            if track[j, columns['lat']] > runway_treshold['lat']:
                new_point = track[j].copy()
                new_point[columns['lon']] = runway_treshold['lon']
                new_point[columns['lat']] = runway_treshold['lat']
                track = np.vstack((track[:j], new_point))
                break
    else:  # Undershoot
        track[-1, columns['lon']] = runway_treshold['lon']
        track[-1, columns['lat']] = runway_treshold['lat']

    # Interpolate or truncate to 200 points
    current_len = len(track)
    standardized_track = np.zeros((num_segments, track.shape[1]), dtype=object)

    # Keep first and last points
    standardized_track[0, :] = track[0, :]
    standardized_track[-1, :] = track[-1, :]

    # Interpolate intermediate points
    for i in range(1, num_segments - 1):
        # Compute proportional index in original track
        idx = int(i * (current_len - 1) / (num_segments - 1))
        standardized_track[i, :] = track[idx, :]

    tracks_filtered.append(standardized_track)

# %%
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)

ax[1].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY 19')

for track in tracks_filtered:
    ax[0].plot(track[0:, columns['lon']], track[0:, columns['lat']])
    ax[1].plot(track[190:, columns['lon']], track[190:, columns['lat']])
```

```
ax[0].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY 19')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend()

ax[1].set_title('Flight Tracks ending at the Runway Threshold (last 10 points)')
ax[1].set_xlabel('Longitude')
ax[1].set_ylabel('Latitude')
ax[1].legend()

# %%
def get_great_circle_distance(lon1, lat1, lon2, lat2):
    r = 6371.137 # The Earth's radius in km

    # Convert to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # Compute cosine of central angle
    cos_angle = sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) * cos(lon1 - lon2)

    # Clamp the value to avoid math domain error
    cos_angle = min(1, max(-1, cos_angle))

    # Great circle distance in km
    d = r * acos(cos_angle)

    # Convert km to nm:
    d = d * 0.5399568035

    return d

# %%
waypoints = {
    # SKILS FIVE ARRIVAL (RNAV)
    'BAL': {'lat':39.16673889, 'lon':-76.65018889, 'alt':10000, 'track':234, 'distance':16,
'speed':250},
    'EYESS': {'lat':38.96675278, 'lon':-76.88355833, 'alt':9000, 'track':284, 'distance':6},
    'EDDWD': {'lat':38.96682778, 'lon':-77.01684444, 'alt':8000, 'track':328, 'distance':5},
    'MEGGS': {'lat':39.03336944, 'lon':-77.08359167, 'alt':7000, 'track':327, 'distance':8},
    'NAYES': {'lat':39.13333889, 'lon':-77.21673889, 'alt':6000, 'track':327},
    # RNAV (GPS) Y RWY 19
    'DARIC': {'lat':38.95025, 'lon':-77.15005278, 'alt':2700, 'track':150, 'distance':2.5,
'speed':180}, # Initial Fix
    'SUNEY': {'lat':38.93334167, 'lon':-77.11669722, 'alt':2000, 'track':158, 'distance':1.7},
    'RORRK': {'lat':38.90016389, 'lon':-77.08359167, 'alt':1500, 'track':122, 'distance':1.6},
    'SLAKR': {'lat':38.90000833, 'lon':-77.06668333, 'alt':1020, 'track':149, 'distance':0.7}, #
Final Approach Fix
    'DALLA': {'lat':38.88347222, 'lon':-77.050125, 'alt':780, 'track':149, 'distance':1.2},
    'UDERE': {'lat':38.86683056, 'lon':-77.03345, 'alt':373, 'track':149, 'distance':0.9},
    'RWY19': {'lat':38.861193, 'lon':-77.038730, 'alt':0, 'track':149, 'distance':0}
}

# %%
title = f"Historical Flight Tracks with Waypoints Data"
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

# Plot tracks
for track in tracks_filtered:
    ax.plot(track[0:, columns['lon']], track[0:, columns['lat']])

# Plot waypoints
```

```
for label, data in waypoints.items():
    ax.plot(data['lon'], data['lat'], 'o',
            markeredgewidth=1,
            markeredgecolor='0',
            markersize=10,
            label=label)

ax.set_title('Lateral Position')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.legend()

plt.show()

# %%
flight_id_waypoints = []
waypoint_columns = {'waypoint': 0, 'index': 1, 'lat': 2, 'lon': 3, 'distance': 4, 'altitude' : 5,
'speed': 6}

# Build waypoint data for each flight
for track in tracks_filtered:
    track_waypoints = []
    for waypoint, waypoint_data in waypoints.items():
        target_lat = waypoint_data['lat']
        target_lon = waypoint_data['lon']

        lat_list = track[:, columns['lat']].tolist()
        lon_list = track[:, columns['lon']].tolist()

        # Calculate great-circle distances for all points in the track
        distances = [
            get_great_circle_distance(lon_list[k], lat_list[k], target_lon, target_lat)
            for k in range(len(track))
        ]

        # Find closest point
        closest_index = int(np.argmin(distances))
        closest_lat = lat_list[closest_index]
        closest_lon = lon_list[closest_index]
        closest_distance = distances[closest_index]
        closest_altitude = track[closest_index, columns['baroaltitude']]
        closest_speed = track[closest_index, columns['velocity']]

        # Store waypoint info
        array = np.array([waypoint, closest_index, closest_lat, closest_lon, closest_distance,
closest_altitude, closest_speed])
        track_waypoints.append(array)

    flight_id = track[0, columns['flight_id']]
    flight_id_waypoints.append([flight_id, track_waypoints])

# %%
title = "Lateral Dispersion at the Waypoints\n"
fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(18, 20), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize=14)

axes = axes.flatten()

# Dictionary to store results for later use
lateral_dispersion_waypoints_dict = {}

for i, waypoint in enumerate(waypoints.keys()):
    lateral_dispersion_waypoints = []

    # Collect distances for this waypoint across all flights
```

```
    for flight_data in flight_id_waypoints:
        for wp_data in flight_data[1]:
            if wp_data[waypoint_columns['waypoint']] == waypoint:
                lateral_dispersion_waypoints.append(float(wp_data[waypoint_columns['distance']]))

    # Save results in dictionary
    lateral_dispersion_waypoints_dict[waypoint] = {
        'data': lateral_dispersion_waypoints
    }

    # Plot histogram
    ax = axes[i]
    ax.hist(lateral_dispersion_waypoints, bins=25, edgecolor='black')
    ax.set_title(f'{waypoint}')
    ax.set_xlabel('Distance, nautical miles')
    ax.set_ylabel('Frequency')

plt.show()

# %%
%matplotlib widget
fig = plt.figure()
title = '3D View'
fig.suptitle(title, weight='bold', fontsize = 12)
ax = fig.add_subplot(111, projection='3d')

# Plot 3D tracks
for track in tracks_filtered:
    ax.plot(track[0:, columns['lon']], track[0:, columns['lat']], track[0:,
columns['baroaltitude']])

# Plot waypoints
for label, data in waypoints.items():
    ax.plot(data['lon'], data['lat'], data['alt'], 'o',
            markeredgewidth=1,
            markeredgecolor='0',
            label=label)

ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.set_zlabel('Altitude')
ax.view_init(elev=5, azim=30)
ax.legend(title='Waypoints', bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.show()

# %%
%matplotlib inline

# %%
title = "Vertical Dispersion at the Waypoints\n"
fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(18, 20), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize=14)

axes = axes.flatten()

# Dictionary to store results for later use
vertical_dispersion_waypoints_dict = {}

for i, waypoint in enumerate(waypoints.keys()):
    vertical_dispersion_waypoints = []

    # Collect altitudes for this waypoint across all flights
    for flight_data in flight_id_waypoints:
        for wp_data in flight_data[1]:  # wp_data is [waypoint, index, lat, lon, distance]
            if wp_data[waypoint_columns['waypoint']] == waypoint:
```

```
                vertical_dispersion_waypoints.append(float(wp_data[waypoint_columns['altitude']]))

    nominal_altitude = waypoints[waypoint].get('alt', 'N/A')

    # Save results in dictionary
    vertical_dispersion_waypoints_dict[waypoint] = {
        'nominal_altitude': nominal_altitude,
        'data': vertical_dispersion_waypoints
    }

    # Plot histogram
    ax = axes[i]
    ax.hist(vertical_dispersion_waypoints, bins=15, edgecolor='black', color='green')
    ax.set_title(f'{waypoint} ({nominal_altitude} ft)')
    ax.set_xlabel('Baroaltitude, feet')
    ax.set_ylabel('Frequency')

plt.show()

# %%
title = "Speed Dispersion at BAL and DARIC\n"
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 6), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize=14)

axes = axes.flatten()

# Define the waypoints of interest
selected_waypoints = ['BAL', 'DARIC']

for i, waypoint in enumerate(selected_waypoints):
    speed_dispersion_waypoints = []

    # Collect speeds for this waypoint across all flights
    for flight_data in flight_id_waypoints:
        for wp_data in flight_data[1]:  # wp_data = [waypoint, index, lat, lon, distance,
velocity, altitude]
            if wp_data[waypoint_columns['waypoint']] == waypoint:
                speed_dispersion_waypoints.append(float(wp_data[waypoint_columns['speed']]))

    # Get nominal altitude from waypoints dictionary
    nominal_altitude = waypoints[waypoint].get('speed', 'N/A')

    # Plot histogram
    ax = axes[i]
    ax.hist(speed_dispersion_waypoints, bins=25, edgecolor='black', color='purple')
    ax.set_title(f'{waypoint} ({nominal_altitude} ft)', fontsize=12)
    ax.set_xlabel('Ground Speed (knots)')
    ax.set_ylabel('Frequency')

plt.show()

# %%
title = "Speed Dispersion at the Waypoints\n"
fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(18, 20), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize=14)

axes = axes.flatten()

# Dictionary to store results for later use
speed_dispersion_waypoints_dict = {}

for i, waypoint in enumerate(waypoints.keys()):
    speed_dispersion_waypoints = []

    # Collect speeds for this waypoint across all flights
```

```
    for flight_data in flight_id_waypoints:
        for wp_data in flight_data[1]:  # wp_data is [waypoint, index, lat, lon, distance]
            if wp_data[waypoint_columns['waypoint']] == waypoint:
                speed_dispersion_waypoints.append(float(wp_data[waypoint_columns['speed']]))

    nominal_speed = waypoints[waypoint].get('speed', 'N/A')

    # Save results in dictionary
    speed_dispersion_waypoints_dict[waypoint] = {
        'nominal_speed': nominal_speed,
        'data': speed_dispersion_waypoints
    }

    # Plot histogram
    ax = axes[i]
    ax.hist(speed_dispersion_waypoints, bins=25, edgecolor='black', color='purple')
    ax.set_title(f'{waypoint}')
    ax.set_xlabel('Ground Speed, knots')
    ax.set_ylabel('Frequency')

plt.show()

# %%
rows = []
for flight_data in flight_id_waypoints:
    flight_id = flight_data[0]
    wp_list = flight_data[1]

    row = {'flight_id': flight_id}
    for wp_data in wp_list:
        waypoint_name = wp_data[waypoint_columns['waypoint']]
        distance = float(wp_data[waypoint_columns['distance']])
        altitude = float(wp_data[waypoint_columns['altitude']])
        speed = float(wp_data[waypoint_columns['speed']])

        # Add columns for distance, altitude, and speed
        row[f'{waypoint_name}_distance'] = distance
        row[f'{waypoint_name}_altitude'] = altitude
        row[f'{waypoint_name}_speed'] = speed

    rows.append(row)

# Create DataFrame
df = pd.DataFrame(rows)

# Save to CSV
file_path = os.path.join(output_dir, 'Historical_Data_Waypoints.csv')
df.to_csv(file_path, index=False)
```

Train and Seed Samples Selection:

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import radians, sin, cos, acos
from mpl_toolkits.mplot3d import Axes3D
import math
from datetime import datetime
from collections import defaultdict
from sklearn.metrics import pairwise_distances
import csv
from sklearn.cluster import DBSCAN

# %%
# Define the output directory where the file is stored
output_dir = "C:\\Files\\"

# Build the full file path by joining the directory and file name
file_path = os.path.join(output_dir, "RWY_19_Cluster_North_with_New_Parameters.csv") # File name
cluster = 'Northern'

# Read the CSV file into a pandas DataFrame
data = pd.read_csv(file_path)

# Sort once globally by flight_id and time
data_sorted = data.sort_values(by=['flight_id', 'time']).reset_index(drop=True)

# Number of segments to process for each flight track
num_segments = 200

# Generate an array of segment indices (0 to num_segments-1)
data_points = np.arange(num_segments)

# Group by flight_id
tracks = []
for _, group in data_sorted.groupby('flight_id'):
    if len(group) == num_segments:
        tracks.append(group.to_numpy())

# %%
# Dictionary mapping column names to their index positions in the dataset
columns = {
    'position': 0,                  # Position in segment
    'flight_id': 1,                 # Unique flight identifier
    'time': 2,                      # Timestamp
    'lat': 3,                       # Latitude
    'lon': 4,                       # Longitude
    'ground_speed': 5,              # Ground speed
    'baroaltitude': 6,              # Barometric altitude
    'vertrate': 7,                  # Vertical rate (climb/descent speed)
    'heading': 8,                   # Aircraft heading
    'distance_between_points':9,    # Distance traveled between points
    'accumulated_distance': 10,     # Cumulative Distance traveled
    'duration': 11,                 # Duration of the segment
    'departure_airport': 12,        # Departure airport code
    'arrival_airport': 13,          # Arrival airport code
    'course_change': 14,            # Change in course direction
    'magnitude_course_change': 15,  # Magnitude of course change
    'cumulative_course_change': 16, # Total course change accumulated
    'deceleration': 17,             # Deceleration segments
    'magnitude_deceleration': 18,   # Magnitude of deceleration
```

```
    'cumulative_deceleration': 19,   # Total deceleration accumulated
    'descend': 20,                   # Descending segments
    'magnitude_descend': 21,         # Magnitude of descent
    'cumulative_descend': 22,        # Total descent accumulated
    'speed_change': 23,
    'level_change': 24
}

# Define runway threshold coordinates
runway_treshold = {'lat': 38.861193, 'lon': -77.038730}

# %%
title = f"Historical Flight Tracks arriving on Runway 19 from {cluster} Cluster at DCA
Airport\n{len(tracks)} Tracks"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

for track in tracks:
    ax[0].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax[1].plot(data_points, track[:, columns['baroaltitude']])
    ax[2].plot(data_points, track[:, columns['ground_speed']])
ax[0].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY 19')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend(loc='lower right')

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

# %%
positions = [90, 100, 110, 120]

rows = len(tracks)
cols = len(positions)
position = np.zeros((rows, 2 * cols)) # Two columns required for lat and lon

# Cluster trajectories based on recorded position (lat, lon) for positions:
for row in range(rows):
    for col in range(cols):
        position[row, col*2:col*2+2] = tracks[row][positions[col],[columns['lat'],
columns['lon']]]

# Create DBSCAN model
model = DBSCAN(eps=0.01, min_samples=5)
model.fit(position)
cluster_labels = model.labels_
num_clusters = len(set(cluster_labels)) - (1 if -1 in cluster_labels else 0)
print("Number of clusters:", num_clusters)

# Plot
title = "DBSCAN Clusters"
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

ax.scatter(position[:, 1], position[:, 0], c=cluster_labels, cmap='rainbow')
ax.set_xlabel('Longitude')
```

```
ax.set_ylabel('Latitude')

# %%
# Put tracks into different clusters:
clusters = [[] for _ in range(num_clusters)];
for i in range(len(tracks)):
    clusters[cluster_labels[i]].append(tracks[i])

# Check how many tracks in each cluster:
num_tracks_cluster = [len(cluster) for cluster in clusters]
print(f'The number of tracks in each cluster is {num_tracks_cluster}. The total number is
{sum(num_tracks_cluster)}')

# %%
title = "Runway 19 North Cluster - Data Clusters\n\n\n"
fig, axes = plt.subplots(nrows=7, ncols=3, figsize=(18, 40), tight_layout=True)
fig.subplots_adjust(hspace=0.4)
fig.suptitle(title, weight='bold', fontsize=14)

# Flatten axes for easy indexing
axes = axes.flatten()

for i in range(len(clusters)):

    ax = axes[i]

    # Plot tracks
    for track in clusters[i]:
        ax.plot(track[0:, columns['lon']],track[0:, columns['lat']])
        # ax.plot(track[90, columns['lon']],track[90, columns['lat']], 'ko', markersize=1)
    ax.plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY
19')
    ax.set_title(f'Cluster #{i}\n{len(clusters[i])} tracks')
    ax.set_ylabel('Latitude')
    ax.set_xlabel('Longitude')
    ax.legend()

# %%
train_tracks = clusters[0]
test_tracks = []
for i in [1,2,3,4,6,7,8,9,10,11,12,13,14,15]:
    test_tracks += clusters[i]

title = f"Runway 19 from {cluster} Cluster at DCA Airport\n{len(train_tracks)} Train
Tracks\n{len(test_tracks)} Seed Tracks"
# fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(12,12), tight_layout=True)
fig, ax = plt.subplot_mosaic([['top', 'top'],['left', 'right']],figsize=(8,8),tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

for i, track in enumerate(train_tracks):
    label = f"Train" if i == 0 else None
    ax['top'].plot(track[:, columns['lon']], track[:, columns['lat']], color='gray', zorder=1,
label=label, alpha=0.3)
    ax['left'].plot(data_points, track[:, columns['baroaltitude']], color='gray', zorder=1,
label=label, alpha=0.3)
    ax['right'].plot(data_points, track[:, columns['ground_speed']], color='gray', zorder=1,
label=label, alpha=0.3)

for i, track in enumerate(test_tracks):
    label = f"Seed" if i == 0 else None
    ax['top'].plot(track[:, columns['lon']], track[:, columns['lat']], color='darkblue', zorder=2,
label=label, alpha=0.3)
    ax['left'].plot(data_points, track[:, columns['baroaltitude']], color='darkblue', zorder=2,
label=label, alpha=0.3)
```

```
    ax['right'].plot(data_points, track[:, columns['ground_speed']], color='darkblue', zorder=2,
label=label, alpha=0.3)

ax['top'].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY
19')

ax['top'].set_title('Flight Tracks')
ax['top'].set_xlabel('Longitude')
ax['top'].set_ylabel('Latitude')
ax['top'].legend(loc='lower right')

ax['left'].set_title('Barometric Altitude')
ax['left'].set_xlabel('Data Points')
ax['left'].set_ylabel('Baroaltitude, feet')
ax['left'].legend(loc='lower left')

ax['right'].set_title('Ground Speed')
ax['right'].set_xlabel('Data Points')
ax['right'].set_ylabel('Ground Speed, knots')
ax['right'].legend(loc='lower left')

# %%
n_selected_samples = 25

features = []

# remaining_tracks = [track for i, track in enumerate(tracks)
#                     if i not in train_selected]

for track in test_tracks:
    lon_mean = np.mean(track[:, columns['lon']])
    lat_mean = np.mean(track[:, columns['lat']])
    alt_mean = np.mean(track[:, columns['baroaltitude']])
    speed_mean = np.mean(track[:, columns['ground_speed']])
    features.append([lon_mean, lat_mean, alt_mean, speed_mean])

features = np.array(features)

# Compute pairwise distances
dist_matrix = pairwise_distances(features, metric='euclidean')

title = f"Runway 19 from {cluster} Cluster at DCA Airport\nSelected Seed Tracks"
fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(6,15), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

############
included_samples = {"JIA5678-2025-06-07", "JIA5678-2025-06-07", "RPA5657-2025-07-17", "RPA4337-
2025-08-17", "JBU955-2025-07-16", "RPA4546-2025-07-17"}
############

# Select 25 most different tracks using farthest-point sampling
np.random.seed(20)
selected = [np.random.randint(len(features))]  # start with a random track
for i in range(n_selected_samples - 1):
    # Find track farthest from current selection
    dists_to_selected = dist_matrix[selected].min(axis=0)
    next_idx = np.argmax(dists_to_selected)
    selected.append(next_idx)

    ###############
    if test_tracks[next_idx][0, columns['flight_id']] in included_samples:
    ###############
```

```
        ax[0].plot(test_tracks[next_idx][:, columns['lon']], test_tracks[next_idx][:,
columns['lat']])
        ax[1].plot(data_points, test_tracks[next_idx][:, columns['baroaltitude']])
        ax[2].plot(data_points, test_tracks[next_idx][:, columns['ground_speed']])

ax[0].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY 19')

ax[0].set_title('Flight Tracks')
ax[0].set_xlabel('Longitude')
ax[0].set_ylabel('Latitude')
ax[0].legend(loc='lower right')

ax[1].set_title('Barometric Altitude')
ax[1].set_xlabel('Data Points')
ax[1].set_ylabel('Baroaltitude, feet')

ax[2].set_title('Ground Speed')
ax[2].set_xlabel('Data Points')
ax[2].set_ylabel('Ground Speed, knots')

print("Selected track indices:", selected)
selected_test_tracks = [test_tracks[i] for i in selected]



# %%
file_path = output_dir + f'RWY_19_Northern_Cluster_Train_Tracks.csv'
file = open(file_path, 'w')

headers =
"position,flight_id,time,lat,lon,ground_speed,baroaltitude,vertrate,heading,distance_between_point
s,accumulated_distance,duration,departure_airport,arrival_airport,course_change,magnitude_course_c
hange,cumulative_course_change,deceleration,magnitude_deceleration,cumulative_deceleration,descend
,magnitude_descend,cumulative_descend,speed_change,level_change"
if os.path.exists(file_path) and os.path.getsize(file_path) == 0:
    file.write(headers)

for track in train_tracks:
    for i in range(len(track)):
        line = f"\n{str(i)},{track[i, columns['flight_id']]},{str(track[i,
columns['time']])},{track[i, columns['lat']]},{track[i, columns['lon']]},{track[i,
columns['ground_speed']]},{track[i, columns['baroaltitude']]},\
            {track[i, columns['vertrate']]},{track[i, columns['heading']]},{track[i,
columns['distance_between_points']]},{track[i, columns['accumulated_distance']]},\
                {track[i, columns['duration']]},{track[i, columns['departure_airport']]},{track[i,
columns['arrival_airport']]},{track[i, columns['course_change']]},\
                    {track[i, columns['magnitude_course_change']]},{track[i,
columns['cumulative_course_change']]},{track[i, columns['deceleration']]},\
                        {track[i, columns['magnitude_deceleration']]},{track[i,
columns['cumulative_deceleration']]},{track[i, columns['descend']]},\
                            {track[i, columns['magnitude_descend']]},{track[i,
columns['cumulative_descend']]},{track[i, columns['speed_change']]},{track[i,
columns['level_change']]}"
        file.write(line)
file.close()

# %%
file_path = output_dir + f'RWY_19_Northern_Cluster_Seed_Tracks_All.csv'
file = open(file_path, 'w')

headers =
"position,flight_id,time,lat,lon,ground_speed,baroaltitude,vertrate,heading,distance_between_point
s,accumulated_distance,duration,departure_airport,arrival_airport,course_change,magnitude_course_c
hange,cumulative_course_change,deceleration,magnitude_deceleration,cumulative_deceleration,descend
,magnitude_descend,cumulative_descend,speed_change,level_change"
if os.path.exists(file_path) and os.path.getsize(file_path) == 0:
```

```
    file.write(headers)

for track in test_tracks:
    for i in range(len(track)):
        line = f"\n{str(i)},{track[i, columns['flight_id']]},{str(track[i,
columns['time']])},{track[i, columns['lat']]},{track[i, columns['lon']]},{track[i,
columns['ground_speed']]},{track[i, columns['baroaltitude']]},\
            {track[i, columns['vertrate']]},{track[i, columns['heading']]},{track[i,
columns['distance_between_points']]},{track[i, columns['accumulated_distance']]},\
                {track[i, columns['duration']]},{track[i, columns['departure_airport']]},{track[i,
columns['arrival_airport']]},{track[i, columns['course_change']]},\
                    {track[i, columns['magnitude_course_change']]},{track[i,
columns['cumulative_course_change']]},{track[i, columns['deceleration']]},\
                        {track[i, columns['magnitude_deceleration']]},{track[i,
columns['cumulative_deceleration']]},{track[i, columns['descend']]},\
                            {track[i, columns['magnitude_descend']]},{track[i,
columns['cumulative_descend']]},{track[i, columns['speed_change']]},{track[i,
columns['level_change']]}"
        file.write(line)
file.close()

# %%
file_path = output_dir + f'RWY_19_Northern_Cluster_Seed_Tracks_25.csv'
file = open(file_path, 'w')

headers =
"position,flight_id,time,lat,lon,ground_speed,baroaltitude,vertrate,heading,distance_between_point
s,accumulated_distance,duration,departure_airport,arrival_airport,course_change,magnitude_course_c
hange,cumulative_course_change,deceleration,magnitude_deceleration,cumulative_deceleration,descend
,magnitude_descend,cumulative_descend,speed_change,level_change"
if os.path.exists(file_path) and os.path.getsize(file_path) == 0:
    file.write(headers)

for track in selected_test_tracks:
    for i in range(len(track)):
        line = f"\n{str(i)},{track[i, columns['flight_id']]},{str(track[i,
columns['time']])},{track[i, columns['lat']]},{track[i, columns['lon']]},{track[i,
columns['ground_speed']]},{track[i, columns['baroaltitude']]},\
            {track[i, columns['vertrate']]},{track[i, columns['heading']]},{track[i,
columns['distance_between_points']]},{track[i, columns['accumulated_distance']]},\
                {track[i, columns['duration']]},{track[i, columns['departure_airport']]},{track[i,
columns['arrival_airport']]},{track[i, columns['course_change']]},\
                    {track[i, columns['magnitude_course_change']]},{track[i,
columns['cumulative_course_change']]},{track[i, columns['deceleration']]},\
                        {track[i, columns['magnitude_deceleration']]},{track[i,
columns['cumulative_deceleration']]},{track[i, columns['descend']]},\
                            {track[i, columns['magnitude_descend']]},{track[i,
columns['cumulative_descend']]},{track[i, columns['speed_change']]},{track[i,
columns['level_change']]}"
        file.write(line)
file.close()
```

APPENDIX F – PYTHON SCRIPTS FOR GP MODELS

Model 1: Independent GPR Models with Single Response

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import radians, sin, cos, acos
from collections import defaultdict
from mpl_toolkits.mplot3d import Axes3D
import plotly.graph_objects as go
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, Kernel, Product, ConstantKernel,
ExpSineSquared, Matern, DotProduct, RationalQuadratic
from sklearn.metrics import  mean_absolute_error, mean_squared_error, r2_score,
explained_variance_score
from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler
from scipy.stats import norm
import joblib
import random
import math
import properscoring as ps

# %% [markdown]
# # Read Data

# %%
# Define the output directory where the file is stored
output_dir = "C:\\Files\\"

# Build the full file path by joining the directory and file name
file_path = os.path.join(output_dir, "RWY_19_Northern_Cluster_Train_Tracks.csv")
file_path_test_samples = os.path.join(output_dir, "RWY_19_Northern_Cluster_Seed_Tracks_6.csv")

# Read the CSV file into a pandas DataFrame
data = pd.read_csv(file_path)
data_test_samples = pd.read_csv(file_path_test_samples)

# Number of segments to process for each flight track
num_segments = 200

# Generate an array of segment indices (0 to num_segments-1)
data_points = np.arange(num_segments)

# Collect flight_ids used in test samples
test_flight_ids = set(data_test_samples['flight_id'].unique())

# Build test samples
test_samples = []
for _, group in data_test_samples.groupby('flight_id'):
    if len(group) == num_segments:
        test_samples.append(group.to_numpy())
```

```
# Build tracks excluding test flight_ids
tracks = []
for flight_id, group in data.groupby('flight_id'):
    if len(group) == num_segments and flight_id not in test_flight_ids:
        tracks.append(group.to_numpy())

# %%
# Dictionary mapping column names to their index positions in the dataset
columns = {
    'position': 0,                  # Position in segment
    'flight_id': 1,                 # Unique flight identifier
    'time': 2,                      # Timestamp
    'lat': 3,                       # Latitude
    'lon': 4,                       # Longitude
    'ground_speed': 5,              # Ground speed
    'baroaltitude': 6,              # Barometric altitude
    'vertrate': 7,                  # Vertical rate (climb/descent speed)
    'heading': 8,                   # Aircraft heading
    'distance_between_points':9,    # Distance traveled between points
    'accumulated_distance': 10,     # Cumulative Distance traveled
    'duration': 11,                 # Duration of the segment
    'departure_airport': 12,        # Departure airport code
    'arrival_airport': 13,          # Arrival airport code
    'course_change': 14,            # Change in course direction
    'magnitude_course_change': 15,  # Magnitude of course change
    'cumulative_course_change': 16, # Total course change accumulated
    'deceleration': 17,             # Deceleration segments
    'magnitude_deceleration': 18,   # Magnitude of deceleration
    'cumulative_deceleration': 19,  # Total deceleration accumulated
    'descend': 20,                  # Descending segments
    'magnitude_descend': 21,        # Magnitude of descent
    'cumulative_descend': 22,       # Total descent accumulated
    'lat_lon_morton_encoded': 23    # Latitide and Longitiitude encoded
}

# Add a new column to data
new_column = np.zeros((data.shape[0], 1))
data = np.hstack((data, new_column)) # Latitide and Longitiitude encoded

# Dictionary mapping keys to descriptive labels for visualization or reporting
keys = {
    'lat': 'Latitude',
    'lon': 'Longitude',
    'ground_speed': 'Ground Speed',
    'baroaltitude': 'Barometric Altitude',
    'heading': 'Heading',
    'course_change': 'Course Change',
    'magnitude_course_change': 'Magnitude of Course Change',
    'deceleration': 'Deceleration Segments',
    'magnitude_deceleration': 'Magnitude of Deceleration Segments',
    'descend': 'Descending Segments',
    'magnitude_descend': 'Magnitude of Descending Segments'
}

# Define runway threshold coordinates
runway_treshold = {'lat': 38.861193, 'lon': -77.038730}

# %% [markdown]
# # Gaussian Process Models

# %% [markdown]
# Declare a dictionary to save one or many response keys:

# %%
```

```
def get_dictionary(keys):

    inc = 0
    dictionary = {}
    for key in keys:
        dictionary[key] = inc
        inc += 1
    return dictionary

# %%
def evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples):

    # Normalize y
    scaler_errors = MinMaxScaler()
    y_test_norm = scaler_errors.fit_transform(y_test.reshape(-1, 1))
    y_pred_norm = scaler_errors.transform(y_pred.reshape(-1, 1))
    y_samples_norm = (y_samples - y_train.mean()) / y_train.std()

    # Compute metrics
    mae = mean_absolute_error(y_test_norm, y_pred_norm)
    mse = mean_squared_error(y_test_norm, y_pred_norm)
    rmse = np.sqrt(mse)
    crps = ps.crps_ensemble(y_test_norm.ravel(), y_samples_norm) * .1
    r2 = r2_score(y_test, y_pred)
    evs = explained_variance_score(y_test, y_pred)

    # Print results in CSV format
    print(f"{test_sample_id},{mae:.4f},{mse:.9f},{rmse:.4f},{np.mean(crps):.4f},{r2:.4f},{evs:.4f}
")

    return f"RMSE: {rmse:.4f}\nCRPS: {np.mean(crps):.4f}\nR2: {r2:.4f}"

# %% [markdown]
# ## Gaussian Process Regression

# %% [markdown]
# ### Series #1A and 1B: Position
# - Predictors: **position, latitude, longitude, heading, ground speed, baroaltitude**
# - Response: independent **delta latitude and delta longitude**
# - Model: GPR with Matern Kernel

# %% [markdown]
# Indicate training and test data sizes as well as a number of generated samples :

# %%
train_size = 50
test_size = len(test_samples)

# Randomly select tracks
random.seed(120)
tracks = random.sample(tracks, train_size)

for sample in test_samples:
    tracks.append(sample)

syn_size = 10000000
batch_size = 10

# %% [markdown]
# Extract and fill out predictors and responses arrays:

# %%
predictor_keys = get_dictionary(['position', 'lat', 'lon', 'heading', 'ground_speed',
'baroaltitude'])
response_keys = get_dictionary(['lat', 'lon']) # Use for Regression
```

```
predictors = np.zeros((len(tracks) * num_segments, len(predictor_keys)))
responses = np.zeros((len(tracks) * num_segments, len(response_keys)))

start_row = 0
for track in tracks:
    next_row = start_row + num_segments
    predictors[start_row:next_row,:] = track[:, [columns['position'], columns['lat'],
columns['lon'], columns['heading'], columns['ground_speed'], columns['baroaltitude']]]

    response_column = np.zeros((track.shape[0], len(response_keys)))
    for i in range(1, len(track)):

        for index, res_key in enumerate(response_keys):
            value1 = track[i - 1, [columns[res_key]]]
            value2 = track[i, [columns[res_key]]]
            delta = value2 - value1
            response_column[i, index] = delta

    responses[start_row:next_row,:] = response_column
    start_row = next_row

X = predictors

# %% [markdown]
# Normalize predictors data:

# %%
# Create and fit the scaler
scaler = StandardScaler()
X = scaler.fit_transform(predictors)

# %% [markdown]
# Define a Gaussian Process Regression model:

# %%
kernels = {}
def get_gpr_models(response_keys):

    gpr_models = {}

    for key, value in response_keys.items():
        kernel = 1.0 * Matern(length_scale=1.0, nu=0.5) + WhiteKernel(noise_level=0.5,
noise_level_bounds=(1e-20, 1e+10))
        kernels[key] = kernel
        gpr_model = GaussianProcessRegressor(kernel=kernel,
                                             n_restarts_optimizer=5,
                                             copy_X_train=True,
                                             normalize_y=False,
                                             random_state=100)
        gpr_models[key] = gpr_model

    return gpr_models

gpr_models = get_gpr_models(response_keys)

# %% [markdown]
# Train the Gaussian Process Regression model:

# %%
for res_key in response_keys:
    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments,]
    y_train = y[:train_size * num_segments]
```

```
    gpr_models[res_key].fit(X_train, y_train) # Regression
    print(f"\nModel Stats for: {res_key}")
    print(f"Kernel: {gpr_models[res_key].kernel_}")  # Optimized kernel after fitting
    print(f"Log Marginal Likelihood: {gpr_models[res_key].log_marginal_likelihood_value_:.6f}")  #
Model fit quality



# %%
# joblib.dump(gpr_models, os.path.join(output_dir, "PKL\GPR_Independent_Lat_Lon.pkl") )

# %%
# Load the model (as an alternative to a new GPR model)
# gpr_models = joblib.load(os.path.join(output_dir, "PKL\GPR_Independent_Lat_Lon.pkl"))

# %%
for res_key in response_keys:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Single Independent Δ Response\n\n", weight='bold',
fontsize = 12)

    start_row = train_size * num_segments

    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments,:]
    y_train = y[:train_size * num_segments]

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        # Mean prediction and standard deviation
        y_pred, y_pred_stdev = gpr_models[res_key].predict(X_test, return_std=True)

        # Generate samples from the posterior distribution:
        y_samples = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        # Plot training data
        start_train_row = 0
        for j in range(train_size):
            next_train_row = start_train_row + num_segments
            ax.scatter(data_points, y_train[start_train_row:next_train_row],
                       color='yellow', edgecolor='darkgoldenrod', marker=".", alpha=0.6, zorder=-
1,
                       label="Training Data" if j == 0 else "")
            start_train_row = next_train_row

        ax.plot(data_points, y_test, color='black', linewidth=2, zorder=3, label="Test Data")
        ax.plot(data_points, y_pred, color='black', linewidth=5, zorder=4) # Outline
        ax.plot(data_points, y_pred, color='darkorange', zorder=5, linewidth=4, label="Mean
Prediction")
        ax.fill_between(
            data_points.ravel(),
            y_pred - 1.96 * y_pred_stdev,
```

```
            y_pred + 1.96 * y_pred_stdev,
            alpha=0.4,
            zorder=1,
            label=r"95% confidence interval",
        )

        ax.set_title(f"Change in {keys[res_key]}\n{test_sample_id}", weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"Delta {keys[res_key]}")
        ax.legend(loc='lower left')

        # Add accuracy text inside the subplot
        ax.text(0.05, 0.85,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

    plt.show()

# %% [markdown]
# Response reconstruction and sampling:

# %%
# Initialize storage dictionary
results = {
    'y_test': {},
    'y_pred': {},
    'y_samples': {},
    'y_samples_mean': {},
    'y_samples_stdev': {}
}

for res_key in response_keys:
    # Initialize dictionaries for each res_key
    results['y_test'][res_key] = []
    results['y_pred'][res_key] = []
    results['y_samples'][res_key] = []
    results['y_samples_mean'][res_key] = []
    results['y_samples_stdev'][res_key] = []

    # Reset start_row for each res_key
    start_row = train_size * num_segments

    # Extract response for this key
    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments, :]
    y_train = y[:train_size * num_segments]

    # Create subplots
    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Single Independent Δ Response\n\n", weight='bold',
fontsize = 12)
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]
```

```
        # Mean prediction and standard deviation
        y_pred, y_pred_stdev = gpr_models[res_key].predict(X_test, return_std=True)

        # Inverse transform to get original values
        X_test_original = scaler.inverse_transform(X_test)

        # Backward Learning
        # Reconstruct response data given a single data point
        y_test_original = np.zeros(num_segments)
        y_pred_original = np.zeros(num_segments)

        # Initialize with the last original point
        y_test_original[-1] = X_test_original[-1, predictor_keys[res_key]]
        y_pred_original[-1] = X_test_original[-1, predictor_keys[res_key]]

        # Iterate backward
        for row in range(num_segments - 2, -1, -1):
            y_test_original[row] = y_test_original[row + 1] - y_test[row + 1]
            y_pred_original[row] = y_pred_original[row + 1] - y_pred[row + 1]

        # Generate samples from the posterior distribution:
        y_samples = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)

        # Reconstruct samples backward
        y_samples_original = np.zeros((y_samples.shape[0], y_samples.shape[1]))
        for j in range(y_samples.shape[1]):
            y_samples_original[-1, j] = X_test_original[-1, predictor_keys[res_key]]
            for row in range(num_segments - 2, -1, -1):
                y_samples_original[row, j] = y_samples_original[row + 1, j] - y_samples[row + 1,
j]

        # Compute mean and std of samples
        y_samples_mean = np.apply_over_axes(func=np.mean, a=y_samples_original, axes=1).squeeze()
        y_samples_stdev = np.apply_over_axes(func=np.std, a=y_samples_original, axes=1).squeeze()

        # Store copies of the results
        results['y_test'][res_key].append(y_test_original.copy())
        results['y_pred'][res_key].append(y_pred_original.copy())
        results['y_samples'][res_key].append(y_samples_original.copy())
        results['y_samples_mean'][res_key].append(y_samples_mean.copy())
        results['y_samples_stdev'][res_key].append(y_samples_stdev.copy())

        # Plot Data
        ax = axes[i]  # Select the correct subplot
        ax.plot(data_points, y_test_original, color='black', linewidth=3,  zorder=3, label="Test
Data")
        ax.plot(data_points, y_pred_original, color='black', linewidth=4, zorder=4) # Outline
        ax.plot(data_points, y_pred_original, color='darkorange',
linewidth=3,  zorder=5,label="Mean Prediction")
        ax.fill_between(
            data_points.ravel(),
            y_samples_mean - 1.96 * y_samples_stdev,
            y_samples_mean + 1.96 * y_samples_stdev,
            alpha=0.4,
            zorder=-1,
            label=r"95% confidence interval",
        )

        # Add accuracy text inside the subplot
        ax.text(0.05, 0.02,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
```

```
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

        ax.set_title(f"{keys[res_key]}\n{test_sample_id}", weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"{keys[res_key]}")
        ax.legend()

    plt.show()

# %% [markdown]
# Plot a combined latitude and longitude:

# %%
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
fig.subplots_adjust(hspace=0.4)
fig.suptitle("Gaussian Process Regression - Single Independent Δ Response\n\n", weight='bold',
fontsize = 12)
axes = axes.flatten()

syn_tracks = []  # Collect dictionaries for all test samples

for i in range(test_size):

    y_test_lat = results['y_test']['lat'][i]
    y_test_lon = results['y_test']['lon'][i]
    y_pred_lat = results['y_pred']['lat'][i]
    y_pred_lon = results['y_pred']['lon'][i]
    y_samples_lat = results['y_samples']['lat'][i]
    y_samples_lon = results['y_samples']['lon'][i]
    y_samples_mean_lat = results['y_samples_mean']['lat'][i]
    y_samples_stdev_lat = results['y_samples_stdev']['lat'][i]

    # Plot Data
    ax = axes[i]  # Select the correct subplot
    ax.plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, zorder=6,
label='RWY 19')
    ax.plot(y_test_lon, y_test_lat, color='black', linewidth=3,  zorder=3, label="Test Data")
    ax.plot(y_pred_lon, y_pred_lat, color='black', linewidth=4, zorder=4) # Outline
    ax.plot(y_pred_lon, y_pred_lat, color='darkorange', linewidth=3,  zorder=5,label="Mean
Prediction")

    test_sample_id = test_samples[i][0, columns['flight_id']]

    col = 0
    num_samples = y_samples_original.shape[1]

    for start in range(0, num_samples, batch_size):
        end = min(start + batch_size, num_samples)
        batch_lat = y_samples_lat[:, start:end]
        batch_lon = y_samples_lon[:, start:end]

        batch_mean_lat = batch_lat.mean(axis=1)
        batch_mean_lon = batch_lon.mean(axis=1)

        for row in range(batch_mean_lat.shape[0]):
            entry = {
                'test_sample': test_sample_id,
                'syn_track_index': col,
                'position': row,
                'lat': batch_mean_lat[row],
                'lon':batch_mean_lon[row]
            }
            syn_tracks.append(entry)
            col += 1
```

```
    ax.set_title(f"Flight Tracks\n{test_sample_id}", weight='bold')
    ax.set_xlabel('Longitude')
    ax.set_xlim(-77.5, -76.2)
    ax.set_ylabel("Latitude")
    ax.set_ylim(38.8, 39.6)
    ax.legend()

plt.show()

# %% [markdown]
# ### Series 1C: Altitude
# - Predictors: **position, ground speed, baroaltitude, descend, magnitude of descend**
# - Response: independent **delta altitude**

# %% [markdown]
# Extract and fill out predictors and responses arrays:

# %%
# predictor_keys = get_dictionary(['position', 'ground_speed', 'baroaltitude', 'descend',
'magnitude_descend'])
predictor_keys = get_dictionary(['position', 'ground_speed', 'baroaltitude', 'descend',
'cumulative_descend'])
response_keys = get_dictionary(['baroaltitude', ])

predictors = np.zeros((len(tracks) * num_segments, len(predictor_keys)))
responses = np.zeros((len(tracks) * num_segments, len(response_keys)))

start_row = 0
for track in tracks:
    next_row = start_row + num_segments
    # predictors[start_row:next_row,:] = track[:, [columns['position'], columns['ground_speed'],
columns['baroaltitude'], columns['descend'], columns['magnitude_descend']]]
    predictors[start_row:next_row,:] = track[:, [columns['position'], columns['ground_speed'],
columns['baroaltitude'], columns['descend'], columns['cumulative_descend']]]

    response_column = np.zeros((track.shape[0], len(response_keys)))
    for i in range(1, len(track)):

        for index, res_key in enumerate(response_keys):
            value1 = track[i - 1, [columns[res_key]]]
            value2 = track[i, [columns[res_key]]]
            delta = value2 - value1
            response_column[i, index] = delta

    responses[start_row:next_row,:] = response_column
    start_row = next_row

X = predictors

# %% [markdown]
# Normalize predictors data:

# %%
# Create and fit the scaler
scaler = StandardScaler()
X = scaler.fit_transform(predictors)

# %% [markdown]
# Define a Gaussian Process Regression model:

# %%
def get_gpr_models(response_keys):

    gpr_models = {}
```

```
    for key, value in response_keys.items():
        kernel = ConstantKernel(1.0, (1e-3, 1e3)) * Matern(length_scale=1.0,
length_scale_bounds=(1e-2, 1e5), nu=1.5) + WhiteKernel(noise_level=1)

        gpr_model = GaussianProcessRegressor(kernel=kernel,
                                             optimizer='fmin_l_bfgs_b', n_restarts_optimizer=5,
                                             copy_X_train=True,
                                             normalize_y=True,
                                             random_state=100)
        gpr_models[key] = gpr_model

    return gpr_models

gpr_models = get_gpr_models(response_keys)

# %%
for res_key in response_keys:
    # Generate prior samples from the kernel
    X_prior = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
    n_samples = 5
    prior_samples = gpr_models[res_key].sample_y(X_prior, n_samples=n_samples, random_state=100)

    # Plot prior
    fig, ax = plt.subplots(figsize=(8, 4))
    for i in range(n_samples):
        ax.plot(X_prior, prior_samples[:, i], lw=1, label=f"Sample {i}")
    ax.set_title(f"Samples from Prior Distribution for
{keys[res_key]}\n\n{gpr_models[res_key].kernel}", weight='bold')
    ax.set_xlabel("Input Value")
    ax.set_ylabel("Kernel Value")
    ax.legend()

# %% [markdown]
# Train the Gaussian Process Regression model:

# %%
for res_key in response_keys:
    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments,]
    y_train = y[:train_size * num_segments]

    gpr_models[res_key].fit(X_train, y_train) # Regression
    print(f"\nModel Stats for: {res_key}")
    print(f"Kernel: {gpr_models[res_key].kernel_}")  # Optimized kernel after fitting
    print(f"Log Marginal Likelihood: {gpr_models[res_key].log_marginal_likelihood_value_:.6f}")  #
Model fit quality

# %%
# joblib.dump(gpr_models, os.path.join(output_dir, "PKL\GPR_Independent_Baroaltitude.pkl"))

# %%
# Load the model (as an alternative to a new GPR model)
# gpr_models = joblib.load(os.path.join(output_dir, "PKL\GPR_Independent_Baroaltitude.pkl"))

# %%
for res_key in response_keys:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Single Independent Δ Response\n\n", weight='bold',
fontsize = 12)

    start_row = train_size * num_segments
```

```
    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments,:]
    y_train = y[:train_size * num_segments]

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        # Mean prediction and standard deviation
        y_pred, y_pred_stdev = gpr_models[res_key].predict(X_test, return_std=True)

        # Generate samples from the posterior distribution:
        y_samples = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        # Plot training data
        start_train_row = 0
        for j in range(train_size):
            next_train_row = start_train_row + num_segments
            ax.scatter(data_points, y_train[start_train_row:next_train_row],
                       color='yellow', edgecolor='darkgoldenrod', marker=".", alpha=0.6,
                       label="Training Data" if j == 0 else "")
            start_train_row = next_train_row

        ax.plot(data_points, y_test, color='black', linewidth=2, zorder=3, label="Test Data")
        ax.plot(data_points, y_pred, color='black', linewidth=5, zorder=4) # Outline
        ax.plot(data_points, y_pred, color='darkorange', zorder=5, linewidth=4, label="Mean
Prediction")
        ax.fill_between(
            data_points.ravel(),
            y_pred - 1.96 * y_pred_stdev,
            y_pred + 1.96 * y_pred_stdev,
            alpha=0.4,
            zorder=-1,
            label=r"95% confidence interval",
        )

        ax.set_title(f"Change in {keys[res_key]}\n{test_sample_id}", weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"Delta {keys[res_key]}")
        ax.legend()

        # Add accuracy text inside the subplot
        ax.text(0.05, 0.1,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

    plt.show()

# %% [markdown]
# Response reconstruction and sampling:
```

```
# %%
# Initialize storage dictionary
results = {
    'y_test': {},
    'y_pred': {},
    'y_samples': {},
    'y_samples_mean': {},
    'y_samples_stdev': {}
}

for res_key in response_keys:
    # Initialize dictionaries for each res_key
    results['y_test'][res_key] = []
    results['y_pred'][res_key] = []
    results['y_samples'][res_key] = []
    results['y_samples_mean'][res_key] = []
    results['y_samples_stdev'][res_key] = []

    # Reset start_row for each res_key
    start_row = train_size * num_segments

    # Extract response for this key
    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments, :]
    y_train = y[:train_size * num_segments]

    # Create subplots
    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Single Independent Δ Response\n\n", weight='bold',
fontsize = 12)
    axes = axes.flatten()

    syn_altitudes = []

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        # Mean prediction and standard deviation
        y_pred, y_pred_stdev = gpr_models[res_key].predict(X_test, return_std=True)

        # Inverse transform to get original values
        X_test_original = scaler.inverse_transform(X_test)

        # Backward Learning
        # Reconstruct response data given a single data point
        y_test_original = np.zeros(num_segments)
        y_pred_original = np.zeros(num_segments)

        # Initialize with the last original point
        y_test_original[-1] = X_test_original[-1, predictor_keys[res_key]]
        y_pred_original[-1] = X_test_original[-1, predictor_keys[res_key]]

        # Iterate backward
        for row in range(num_segments - 2, -1, -1):
            y_test_original[row] = y_test_original[row + 1] - y_test[row + 1]
            y_pred_original[row] = y_pred_original[row + 1] - y_pred[row + 1]

        # Generate samples from the posterior distribution:
        y_samples = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)
```

```
        # Reconstruct samples backward
        y_samples_original = np.zeros((y_samples.shape[0], y_samples.shape[1]))
        for j in range(y_samples.shape[1]):
            y_samples_original[-1, j] = X_test_original[-1, predictor_keys[res_key]]
            for row in range(num_segments - 2, -1, -1):
                y_samples_original[row, j] = y_samples_original[row + 1, j] - y_samples[row + 1,
j]

        # Compute mean and std of samples
        y_samples_mean = np.apply_over_axes(func=np.mean, a=y_samples_original, axes=1).squeeze()
        y_samples_stdev = np.apply_over_axes(func=np.std, a=y_samples_original, axes=1).squeeze()

        # Store copies of the results
        results['y_test'][res_key].append(y_test_original.copy())
        results['y_pred'][res_key].append(y_pred_original.copy())
        results['y_samples'][res_key].append(y_samples_original.copy())
        results['y_samples_mean'][res_key].append(y_samples_mean.copy())
        results['y_samples_stdev'][res_key].append(y_samples_stdev.copy())

        # Plot Data
        ax = axes[i]  # Select the correct subplot
        ax.plot(data_points, y_test_original, color='black', linewidth=3,  zorder=3, label="Test
Data")
        ax.plot(data_points, y_pred_original, color='black', linewidth=4, zorder=4) # Outline
        ax.plot(data_points, y_pred_original, color='darkorange',
linewidth=3,  zorder=5,label="Mean Prediction")

        col = 0
        num_samples = y_samples_original.shape[1]

        for start in range(0, num_samples, batch_size):
            end = min(start + batch_size, num_samples)
            batch = y_samples_original[:, start:end]

            batch_mean = batch.mean(axis=1)

            for row in range(batch_mean.shape[0]):
                entry = {
                    'test_sample': test_sample_id,
                    'syn_track_index': col,
                    'position': row,
                    'alt': batch_mean[row]
                }
                syn_altitudes.append(entry)
                col += 1

        ax.fill_between(
            data_points.ravel(),
            y_samples_mean - 1.96 * y_samples_stdev,
            y_samples_mean + 1.96 * y_samples_stdev,
            alpha=0.4,
            color='lightblue',
            zorder=-1,
            label=r"95% confidence interval",
        )

        # Add accuracy text inside the subplot
        ax.text(0.05, 0.1,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
```

```
        ax.set_title(f"{keys[res_key]}\n{test_sample_id}", weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"{keys[res_key]}")
        ax.legend()

    plt.show()

# %% [markdown]
# ### Series 1D: Ground Speed
# - Predictors: **position, ground speed, deceleration, magnitude of deceleration, baroaltitude**
# - Response: independent **delta ground speed**


# %% [markdown]
# Extract and fill out predictors and responses arrays:

# %%
predictor_keys = get_dictionary(['position', 'ground_speed', 'deceleration',
'cumulative_deceleration', 'baroaltitude'])
response_keys = get_dictionary(['ground_speed'])

predictors = np.zeros((len(tracks) * num_segments, len(predictor_keys)))
responses = np.zeros((len(tracks) * num_segments, len(response_keys)))

start_row = 0
for track in tracks:
    next_row = start_row + num_segments
    predictors[start_row:next_row,:] = track[:, [columns['position'], columns['ground_speed'],
columns['deceleration'], columns['cumulative_deceleration'],columns['baroaltitude']]]

    response_column = np.zeros((track.shape[0], len(response_keys)))
    for i in range(1, len(track)):

        for index, res_key in enumerate(response_keys):
            value1 = track[i - 1, [columns[res_key]]]
            value2 = track[i, [columns[res_key]]]
            delta = value2 - value1
            response_column[i, index] = delta

    responses[start_row:next_row,:] = response_column
    start_row = next_row

X = predictors

# %% [markdown]
# Normalize predictors data:

# %%
# Create and fit the scaler
scaler = StandardScaler()
X = scaler.fit_transform(predictors)

# %% [markdown]
# Define a Gaussian Process Regression model:

# %%
kernels = {}
def get_gpr_models(response_keys):

    gpr_models = {}

    for key, value in response_keys.items():
        kernel = ConstantKernel(1.0, (1e-3, 1e3)) * (Matern(length_scale=1.0,
length_scale_bounds=(1e-2, 1e3), nu=1.5)
```

```
                                                + RationalQuadratic(length_scale=1.0,
alpha=0.5, alpha_bounds=(1e-2, 1e10))
                                                ) + WhiteKernel(noise_level=1.0,
noise_level_bounds=(1e-20, 1e1))

        kernels[key] = kernel
        gpr_model = GaussianProcessRegressor(kernel=kernel,
                                             optimizer='fmin_l_bfgs_b', n_restarts_optimizer=5,
                                             alpha=0.001,
                                             copy_X_train=True,
                                             normalize_y=True,
                                             random_state=100)
        gpr_models[key] = gpr_model

    return gpr_models

gpr_models = get_gpr_models(response_keys)

# %%
for res_key in response_keys:
    # Generate prior samples from the kernel
    X_prior = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
    n_samples = 5
    prior_samples = gpr_models[res_key].sample_y(X_prior, n_samples=n_samples, random_state=100)

    # Plot prior
    fig, ax = plt.subplots(figsize=(8, 4))
    for i in range(n_samples):
        ax.plot(X_prior, prior_samples[:, i], lw=1, label=f"Sample {i}")
    ax.set_title(f"Samples from Prior Distribution for
{keys[res_key]}\n\n{gpr_models[res_key].kernel}", weight='bold')
    ax.set_xlabel("Input Value")
    ax.set_ylabel("Kernel Value")
    ax.legend()

# %% [markdown]
# Train the Gaussian Process Regression model:

# %%
for res_key in response_keys:
    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments,]
    y_train = y[:train_size * num_segments]

    gpr_models[res_key].fit(X_train, y_train) # Regression
    print(f"\nModel Stats for: {res_key}")
    print(f"Kernel: {gpr_models[res_key].kernel_}")  # Optimized kernel after fitting
    print(f"Log Marginal Likelihood: {gpr_models[res_key].log_marginal_likelihood_value_:.6f}")  #
Model fit quality

# %%
# joblib.dump(gpr_models, os.path.join(output_dir, "PKL\GPR_Independent_GroundSpeed.pkl") )

# %%
# Load the model (as an alternative to a new GPR model)
# gpr_models = joblib.load(os.path.join(output_dir, "PKL\GPR_Independent_GroundSpeed.pkl"))

# %%
for res_key in response_keys:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Single Independent Δ Response\n\n", weight='bold',
fontsize = 12)
```

```
    start_row = train_size * num_segments

    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments,:]
    y_train = y[:train_size * num_segments]

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        # Mean prediction and standard deviation
        y_pred, y_pred_stdev = gpr_models[res_key].predict(X_test, return_std=True)

        # Generate samples from the posterior distribution:
        y_samples = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        # Plot training data
        start_train_row = 0
        for j in range(train_size):
            next_train_row = start_train_row + num_segments
            ax.scatter(data_points, y_train[start_train_row:next_train_row],
                       color='yellow', edgecolor='darkgoldenrod', marker=".", alpha=0.6,
                       label="Training Data" if j == 0 else "")
            start_train_row = next_train_row

        ax.plot(data_points, y_test, color='black', linewidth=2, zorder=3, label="Test Data")
        ax.plot(data_points, y_pred, color='black', linewidth=5, zorder=4) # Outline
        ax.plot(data_points, y_pred, color='darkorange', zorder=5, linewidth=4, label="Mean
Prediction")
        ax.fill_between(
            data_points.ravel(),
            y_pred - 1.96 * y_pred_stdev,
            y_pred + 1.96 * y_pred_stdev,
            alpha=0.4,
            zorder=-1,
            label=r"95% confidence interval",
        )

        ax.set_title(f"Change in {keys[res_key]}\n{test_sample_id}", weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"Delta {keys[res_key]}")
        ax.legend()

        # Add accuracy text inside the subplot
        ax.text(0.05, 0.78,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

    plt.show()

# %% [markdown]
```

```
# Response reconstruction and sampling:

# %%
# Initialize storage dictionary
results = {
    'y_test': {},
    'y_pred': {},
    'y_samples': {},
    'y_samples_mean': {},
    'y_samples_stdev': {}
}

for res_key in response_keys:
    # Initialize dictionaries for each res_key
    results['y_test'][res_key] = []
    results['y_pred'][res_key] = []
    results['y_samples'][res_key] = []
    results['y_samples_mean'][res_key] = []
    results['y_samples_stdev'][res_key] = []

    # Reset start_row for each res_key
    start_row = train_size * num_segments

    # Extract response for this key
    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments, :]
    y_train = y[:train_size * num_segments]

    # Create subplots
    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Single Independent Δ Response\n\n", weight='bold',
fontsize = 12)
    axes = axes.flatten()

    syn_speeds = []

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        # Mean prediction and standard deviation
        y_pred, y_pred_stdev = gpr_models[res_key].predict(X_test, return_std=True)

        # Inverse transform to get original values
        X_test_original = scaler.inverse_transform(X_test)

        # Backward Learning
        # Reconstruct response data given a single data point
        y_test_original = np.zeros(num_segments)
        y_pred_original = np.zeros(num_segments)

        # Initialize with the last original point
        y_test_original[-1] = X_test_original[-1, predictor_keys[res_key]]
        y_pred_original[-1] = X_test_original[-1, predictor_keys[res_key]]

        # Iterate backward
        for row in range(num_segments - 2, -1, -1):
            y_test_original[row] = y_test_original[row + 1] - y_test[row + 1]
            y_pred_original[row] = y_pred_original[row + 1] - y_pred[row + 1]
```

```
        # Generate samples from the posterior distribution:
        y_samples = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)

        # Reconstruct samples backward
        y_samples_original = np.zeros((y_samples.shape[0], y_samples.shape[1]))
        for j in range(y_samples.shape[1]):
            y_samples_original[-1, j] = X_test_original[-1, predictor_keys[res_key]]
            for row in range(num_segments - 2, -1, -1):
                y_samples_original[row, j] = y_samples_original[row + 1, j] - y_samples[row + 1,
j]

        # Compute mean and std of samples
        y_samples_mean = np.apply_over_axes(func=np.mean, a=y_samples_original, axes=1).squeeze()
        y_samples_stdev = np.apply_over_axes(func=np.std, a=y_samples_original, axes=1).squeeze()

        # Store copies of the results
        results['y_test'][res_key].append(y_test_original.copy())
        results['y_pred'][res_key].append(y_pred_original.copy())
        results['y_samples'][res_key].append(y_samples_original.copy())
        results['y_samples_mean'][res_key].append(y_samples_mean.copy())
        results['y_samples_stdev'][res_key].append(y_samples_stdev.copy())

        # Plot Data
        ax = axes[i]  # Select the correct subplot
        ax.plot(data_points, y_test_original, color='black', linewidth=3,  zorder=3, label="Test
Data")
        ax.plot(data_points, y_pred_original, color='black', linewidth=4, zorder=4) # Outline
        ax.plot(data_points, y_pred_original, color='darkorange',
linewidth=3,  zorder=5,label="Mean Prediction")

        col = 0
        num_samples = y_samples_original.shape[1]

        for start in range(0, num_samples, batch_size):
            end = min(start + batch_size, num_samples)
            batch = y_samples_original[:, start:end]

            batch_mean = batch.mean(axis=1)

            # ax.plot(data_points, batch_mean,
            #     linewidth=0.5, alpha=0.8, color='gray', zorder=1,
            #     label="Batch Mean" if start == 0 else None
            # )

            for row in range(batch_mean.shape[0]):
                entry = {
                    'test_sample': test_sample_id,
                    'syn_track_index': col,
                    'position': row,
                    'speed': batch_mean[row]
                }
                syn_speeds.append(entry)
                col += 1

        ax.fill_between(
            data_points.ravel(),
            y_samples_mean - 1.96 * y_samples_stdev,
            y_samples_mean + 1.96 * y_samples_stdev,
            alpha=0.4,
            color='lightblue',
            zorder=-1,
            label=r"95% confidence interval",
        )

        # Add accuracy text inside the subplot
```

```
        ax.text(0.05, 0.05,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

        ax.set_title(f"{keys[res_key]}\n{test_sample_id}", weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"{keys[res_key]}")
        ax.legend()

    plt.show()

# %%
test_sample = 'JBU955-2025-07-16'

filtered_syn_tracks = [
    track for track in syn_tracks if track['test_sample'] == test_sample]

filtered_syn_altitudes = [
    alt for alt in syn_altitudes if alt['test_sample'] == test_sample]

filtered_syn_speeds = [
    speed for speed in syn_speeds if speed['test_sample'] == test_sample]

# %%
file_path = output_dir + 'SynTracks_SingleDeltaResponse_1SeedSamples_Batch10_New.csv'

with open(file_path, 'w') as file:
    # Write header
    headers = "test_sample_id,syn_track_index,position,lat,lon,alt,speed"
    file.write(headers)

    # Write rows
    for i in range(len(filtered_syn_tracks)):

            line = (
                f"\n{filtered_syn_tracks[i]['test_sample']},"
                f"{filtered_syn_tracks[i]['syn_track_index']},"
                f"{filtered_syn_tracks[i]['position']},"
                f"{filtered_syn_tracks[i]['lat']},"
                f"{filtered_syn_tracks[i]['lon']},"
                f"{filtered_syn_altitudes[i]['alt']},"
                f"{filtered_syn_speeds[i]['speed']}"
            )
            file.write(line)
```

Model 2: Independent GPR Models with Single Response – Predict Next n^{th} Position

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import radians, sin, cos, acos
from collections import defaultdict
from mpl_toolkits.mplot3d import Axes3D
import plotly.graph_objects as go
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, Kernel, Product, ConstantKernel,
ExpSineSquared, Matern, DotProduct, RationalQuadratic
from sklearn.metrics import  mean_absolute_error, mean_squared_error, r2_score,
explained_variance_score
from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler
from scipy.stats import norm
import GPy
import joblib
import random
import math
import properscoring as ps

# %% [markdown]
# # Read Data

# %%
# Define the output directory where the file is stored
output_dir = "C:\\Files\\"

# Build the full file path by joining the directory and file name
file_path = os.path.join(output_dir, "RWY_19_Northern_Cluster_Train_Tracks.csv")
file_path_test_samples = os.path.join(output_dir, "RWY_19_Northern_Cluster_Seed_Tracks_6.csv")

# Read the CSV file into a pandas DataFrame
data = pd.read_csv(file_path)
data_test_samples = pd.read_csv(file_path_test_samples)

# Number of segments to process for each flight track
num_segments = 200

# Generate an array of segment indices (0 to num_segments-1)
data_points = np.arange(num_segments)

# Collect flight_ids used in test samples
test_flight_ids = set(data_test_samples['flight_id'].unique())

# Build test samples
test_samples = []
for _, group in data_test_samples.groupby('flight_id'):
    if len(group) == num_segments:
        test_samples.append(group.to_numpy())

# Build tracks excluding test flight_ids
tracks = []
for flight_id, group in data.groupby('flight_id'):
    if len(group) == num_segments and flight_id not in test_flight_ids:
        tracks.append(group.to_numpy())

# %%
# Dictionary mapping column names to their index positions in the dataset
columns = {
    'position': 0,                  # Position in segment
```

```
    'flight_id': 1,                 # Unique flight identifier
    'time': 2,                      # Timestamp
    'lat': 3,                       # Latitude
    'lon': 4,                       # Longitude
    'ground_speed': 5,              # Ground speed
    'baroaltitude': 6,              # Barometric altitude
    'vertrate': 7,                  # Vertical rate (climb/descent speed)
    'heading': 8,                   # Aircraft heading
    'distance_between_points':9,    # Distance traveled between points
    'accumulated_distance': 10,     # Cumulative Distance traveled
    'duration': 11,                 # Duration of the segment
    'departure_airport': 12,        # Departure airport code
    'arrival_airport': 13,          # Arrival airport code
    'course_change': 14,            # Change in course direction
    'magnitude_course_change': 15,  # Magnitude of course change
    'cumulative_course_change': 16, # Total course change accumulated
    'deceleration': 17,             # Deceleration segments
    'magnitude_deceleration': 18,   # Magnitude of deceleration
    'cumulative_deceleration': 19,  # Total deceleration accumulated
    'descend': 20,                  # Descending segments
    'magnitude_descend': 21,        # Magnitude of descent
    'cumulative_descend': 22,       # Total descent accumulated
    'lat_lon_morton_encoded': 23    # Latitide and Longitiitude encoded
}

# Add a new column to data
new_column = np.zeros((data.shape[0], 1))
data = np.hstack((data, new_column)) # Latitide and Longitiitude encoded

# Dictionary mapping keys to descriptive labels for visualization or reporting
keys = {
    'lat': 'Latitude',
    'lon': 'Longitude',
    'ground_speed': 'Ground Speed',
    'baroaltitude': 'Barometric Altitude',
    'heading': 'Heading',
    'course_change': 'Course Change',
    'magnitude_course_change': 'Magnitude of Course Change',
    'deceleration': 'Deceleration Segments',
    'magnitude_deceleration': 'Magnitude of Deceleration Segments',
    'descend': 'Descending Segments',
    'magnitude_descend': 'Magnitude of Descending Segments'
}

# Define runway threshold coordinates
runway_treshold = {'lat': 38.861193, 'lon': -77.038730}

# %% [markdown]
# Declare a dictionary to save one or many response keys:

# %%
def get_dictionary(keys):

    inc = 0
    dictionary = {}
    for key in keys:
        dictionary[key] = inc
        inc += 1
    return dictionary

# %%
def evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples):

    # Normalize y
    scaler_errors = MinMaxScaler()
```

```
    y_test_norm = scaler_errors.fit_transform(y_test.reshape(-1, 1))
    y_pred_norm = scaler_errors.transform(y_pred.reshape(-1, 1))
    y_samples_norm = (y_samples - y_train.mean()) / y_train.std()

    # Compute metrics
    mae = mean_absolute_error(y_test_norm, y_pred_norm)
    mse = mean_squared_error(y_test_norm, y_pred_norm)
    rmse = np.sqrt(mse)
    crps = ps.crps_ensemble(y_test_norm.ravel(), y_samples_norm)
    r2 = r2_score(y_test, y_pred)
    evs = explained_variance_score(y_test, y_pred)

    # Print results in CSV format
    print(f"{test_sample_id},{mae:.4f},{mse:.9f},{rmse:.4f},{np.mean(crps):.4f},{r2:.4f},{evs:.4f}
")

    return f"RMSE: {rmse:.4f}\nCRPS: {np.mean(crps):.4f}\nR2: {r2:.4f}"

# %% [markdown]
# ## Gaussian Process Regression

# %% [markdown]
# Indicate training and test data sizes as well as a number of generated samples :

# %%
train_size = 50
test_size = len(test_samples)

# Randomly select tracks
random.seed(120)
tracks = random.sample(tracks, train_size)

for sample in test_samples:
    tracks.append(sample)

syn_size = 1000000

# %%
train_size = 20 # 20
test_size = len(test_samples)

# Randomly select tracks
random.seed(120)
tracks = random.sample(tracks, train_size)

for sample in test_samples:
    tracks.append(sample)

syn_size = 1000000

# %%
predictor_keys = get_dictionary(['position', 'lat', 'lon', 'ground_speed', 'baroaltitude'])
response_keys = get_dictionary(['lat', 'lon'])

predictors = np.zeros((len(tracks) * num_segments, len(predictor_keys)))
responses = np.zeros((len(tracks) * num_segments, len(response_keys)))

n = 5

start_row = 0
for track in tracks:
    next_row = start_row + num_segments
    predictors[start_row:next_row,:] = track[:, [columns['position'], columns['lat'],
columns['lon'], columns['ground_speed'], columns['baroaltitude']]]
```

```
    response_column = np.zeros((track.shape[0], len(response_keys)))

    for res_key in response_keys:
        for i in range(len(track) - n):
            response_column[i, response_keys[res_key]] = track[i + n, [columns[res_key]]]
        for i in range(n, len(track)):
            response_column[i, response_keys[res_key]] = track[i, [columns[res_key]]]

    responses[start_row:next_row,:] = response_column
    start_row = next_row

X = predictors

# %%
# Create and fit the scaler
scaler = StandardScaler()
X = scaler.fit_transform(predictors)

# %%
def get_gpr_models(response_keys):

    gpr_models = {}

    for key, value in response_keys.items():

        if key == 'lat':
            kernel = 1.0 * Matern(length_scale=1.0, nu=2.5) + WhiteKernel(noise_level=0.5,
noise_level_bounds=(1e-10, 1e+5))
            random_state = 22

        if key == 'lon':
            kernel = 1.0 * RBF(length_scale=20000.0) + WhiteKernel(noise_level=0.5,
noise_level_bounds=(1e-20, 1e+50))
            random_state = 10000

        gpr_model = GaussianProcessRegressor(kernel=kernel,
                                             n_restarts_optimizer=10,
                                             copy_X_train=True,
                                             normalize_y=False,
                                             random_state=random_state)
        gpr_models[key] = gpr_model

    return gpr_models

gpr_models = get_gpr_models(response_keys)

# %%
for res_key in response_keys:
    # Generate prior samples from the kernel
    X_prior = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
    n_samples = 5
    prior_samples = gpr_models[res_key].sample_y(X_prior, n_samples=n_samples, random_state=100)

    # Plot prior
    fig, ax = plt.subplots(figsize=(8, 4))
    for i in range(n_samples):
        ax.plot(X_prior, prior_samples[:, i], lw=1, label=f"Sample {i}")
    ax.set_title(f"Samples from Prior Distribution for
{keys[res_key]}\n\n{gpr_models[res_key].kernel}", weight='bold')
    ax.set_xlabel("Input Value")
    ax.set_ylabel("Kernel Value")
    ax.legend()

# %%
for res_key in response_keys:
```

```
    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments,]
    y_train = y[:train_size * num_segments]

    gpr_models[res_key].fit(X_train, y_train) # Regression
    print(f"\nModel Stats for: {res_key}")
    print(f"Kernel: {gpr_models[res_key].kernel_}")  # Optimized kernel after fitting
    print(f"Log Marginal Likelihood: {gpr_models[res_key].log_marginal_likelihood_value_:.6f}")  #
Model fit quality

# %%
# joblib.dump(gpr_models, os.path.join(output_dir, "PKL\GPR_Next_N_Pos.pkl") )

# %%
# Load the model (as an alternative to a new GPR model)
# gpr_models = joblib.load(os.path.join(output_dir, "PKL\GPR_Next_N_Pos.pkl"))

# %%
# Initialize storage dictionary
results = {
    'y_test': {},
    'y_pred': {},
    'y_samples': {},
    'y_samples_mean': {},
    'y_samples_stdev': {}
}

for res_key in response_keys:
    # Initialize dictionaries for each res_key
    results['y_test'][res_key] = []
    results['y_pred'][res_key] = []
    results['y_samples'][res_key] = []
    results['y_samples_mean'][res_key] = []
    results['y_samples_stdev'][res_key] = []

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Single Response in Absolute Coordinates\n",
weight='bold', fontsize = 12)

    start_row = train_size * num_segments

    y = responses[:, response_keys[res_key]]
    X_train = X[:train_size * num_segments,:]
    y_train = y[:train_size * num_segments]

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        # Mean prediction and standard deviation
        y_pred, y_pred_stdev = gpr_models[res_key].predict(X_test, return_std=True)

        # Generate samples from the posterior distribution:
        y_samples = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)

        # Compute mean and std of samples
        y_samples_mean = np.apply_over_axes(func=np.mean, a=y_samples, axes=1).squeeze()
        y_samples_stdev = np.apply_over_axes(func=np.std, a=y_samples, axes=1).squeeze()
```

```
        # Store copies of the results
        results['y_test'][res_key].append(y_test.copy())
        results['y_pred'][res_key].append(y_pred.copy())
        results['y_samples'][res_key].append(y_samples.copy())
        results['y_samples_mean'][res_key].append(y_samples_mean.copy())
        results['y_samples_stdev'][res_key].append(y_samples_stdev.copy())

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        # Plot training data
        start_train_row = 0
        for j in range(train_size):
            next_train_row = start_train_row + num_segments
            ax.plot(data_points, y_train[start_train_row:next_train_row],
                        color='purple', alpha=0.3,
                        label="Training Data" if j == 0 else "")
            start_train_row = next_train_row

        ax.plot(data_points, y_test, color='black', linewidth=2, zorder=3, label="Seed Track")
        ax.plot(data_points, y_pred, color='black', linewidth=5, zorder=4) # Outline
        ax.plot(data_points, y_pred, color='darkorange', zorder=5, linewidth=4, label="Mean
Prediction")
        ax.fill_between(
            data_points.ravel(),
            y_pred - 1.96 * y_pred_stdev,
            y_pred + 1.96 * y_pred_stdev,
            alpha=0.4,
            zorder=-1,
            label=r"95% confidence interval",
        )

        ax.set_title(f"{keys[res_key]}\n{tracks[train_size + i][0, columns['flight_id']]}",
weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"Delta {keys[res_key]}")
        ax.legend()

        # Evaluate performance:
        mae = mean_absolute_error(y_test, y_pred) # MAE measures average absolute difference
between predictions and actual values
        mse = mean_squared_error(y_test, y_pred) # MSE measures the average squared difference
between predicted and actual values
        r2 = r2_score(y_test, y_pred) # R-squared measures how well the model explains the
variance in the target variable
        evs = explained_variance_score(y_test, y_pred) # Measures how much of the variance in the
target variable is explained by the model

        print(f"{tracks[train_size + i][0, columns['flight_id']]} MAE: {mae:.4f}") # a good value
is smaller than the target's typical value
        print(f"{tracks[train_size + i][0, columns['flight_id']]} MSE: {mse:.9f}") # a good value
is closer to 0
        print(f"{tracks[train_size + i][0, columns['flight_id']]} R2 Score: {r2:.4f}") # a good
value is closer to 1
        print(f"{tracks[train_size + i][0, columns['flight_id']]} Explained Variance: {evs:.4f}")
# a good value is closer to 1
        print()

        # Add accuracy text inside the subplot
        ax.text(0.1, 0.03, f"MSE: {mse:.9f}\nR2 Score: {r2:.4f}\nExplain Var: {evs:.4f}",
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
```

```
    plt.show()

# %%
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
fig.subplots_adjust(hspace=0.4)
fig.suptitle("Gaussian Process Regression - Single Response\n Latitude & Longitude in Absolute
Coordinates\n\n", weight='bold', fontsize = 12)
axes = axes.flatten()

for i in range(test_size):

    y_test_lat = results['y_test']['lat'][i]
    y_test_lon = results['y_test']['lon'][i]
    y_pred_lat = results['y_pred']['lat'][i]
    y_pred_lon = results['y_pred']['lon'][i]
    y_samples_lat = results['y_samples']['lat'][i]
    y_samples_lon = results['y_samples']['lon'][i]
    y_samples_mean_lat = results['y_samples_mean']['lat'][i]
    y_samples_stdev_lat = results['y_samples_stdev']['lat'][i]

    # Plot Data
    ax = axes[i]  # Select the correct subplot
    ax.plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, zorder=6,
label='RWY 19')
    ax.plot(y_test_lon, y_test_lat, color='black', linewidth=3,  zorder=3, label="Seed Track")
    ax.plot(y_pred_lon, y_pred_lat, color='black', linewidth=4, zorder=4) # Outline
    ax.plot(y_pred_lon, y_pred_lat, color='darkorange', linewidth=3,  zorder=5,label="Mean
Prediction")
    # Plot samples
    for j in range(y_samples.shape[1]):
        label = f"{y_samples.shape[1]} Samples" if j == 0 else None
        label_interval = r"95% confidence interval" if j == 0 else None
        ax.plot(y_samples_lon[:,j], y_samples_lat[:,j], color='gray', linewidth=0.5, zorder=1,
label=label)
        ax.fill_between(
            y_samples_lon[:,j].ravel(),
            y_samples_mean_lat - 1.96 * y_samples_stdev_lat,
            y_samples_mean_lat + 1.96 * y_samples_stdev_lat,
            color='lightblue',
            alpha=0.4,
            zorder=-1,
            label=label_interval,
        )

    ax.set_title(f"Synthetic Flight Tracks for {tracks[train_size + i][0, columns['flight_id']]}")
    ax.set_xlabel('Longitude')
    ax.set_xlim(-77.4, -76.2)
    ax.set_ylabel("Latitude")
    ax.set_ylim(38.8, 39.7)
    ax.legend()

plt.show()
```

Model 3: GPR with Encoded Multi-Response

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import radians, sin, cos, acos
from collections import defaultdict
from mpl_toolkits.mplot3d import Axes3D
import plotly.graph_objects as go
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, Kernel, Product, ConstantKernel,
ExpSineSquared, Matern, DotProduct, RationalQuadratic
from sklearn.metrics import  mean_absolute_error, mean_squared_error, r2_score,
explained_variance_score
from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler
from scipy.stats import norm
import GPy
import joblib
import random
import math
import properscoring as ps

# %% [markdown]
# # Read Data

# %%
# Define the output directory where the file is stored
output_dir = "C:\\Files\\"

# Build the full file path by joining the directory and file name
file_path = os.path.join(output_dir, "RWY_19_Northern_Cluster_Train_Tracks.csv")
file_path_test_samples = os.path.join(output_dir, "RWY_19_Northern_Cluster_Seed_Tracks_6.csv")

# Read the CSV file into a pandas DataFrame
data = pd.read_csv(file_path)
data_test_samples = pd.read_csv(file_path_test_samples)

# Number of segments to process for each flight track
num_segments = 200

# Generate an array of segment indices (0 to num_segments-1)
data_points = np.arange(num_segments)

# Collect flight_ids used in test samples
test_flight_ids = set(data_test_samples['flight_id'].unique())

# Build test samples
test_samples = []
for _, group in data_test_samples.groupby('flight_id'):
    if len(group) == num_segments:
        test_samples.append(group.to_numpy())

# Build tracks excluding test flight_ids
tracks = []
for flight_id, group in data.groupby('flight_id'):
    if len(group) == num_segments and flight_id not in test_flight_ids:
        tracks.append(group.to_numpy())

# %%
# Dictionary mapping column names to their index positions in the dataset
columns = {
    'position': 0,                  # Position in segment
```

```
    'flight_id': 1,                 # Unique flight identifier
    'time': 2,                      # Timestamp
    'lat': 3,                       # Latitude
    'lon': 4,                       # Longitude
    'ground_speed': 5,              # Ground speed
    'baroaltitude': 6,              # Barometric altitude
    'vertrate': 7,                  # Vertical rate (climb/descent speed)
    'heading': 8,                   # Aircraft heading
    'distance_between_points':9,    # Distance traveled between points
    'accumulated_distance': 10,     # Cumulative Distance traveled
    'duration': 11,                 # Duration of the segment
    'departure_airport': 12,        # Departure airport code
    'arrival_airport': 13,          # Arrival airport code
    'course_change': 14,            # Change in course direction
    'magnitude_course_change': 15,  # Magnitude of course change
    'cumulative_course_change': 16, # Total course change accumulated
    'deceleration': 17,             # Deceleration segments
    'magnitude_deceleration': 18,   # Magnitude of deceleration
    'cumulative_deceleration': 19,  # Total deceleration accumulated
    'descend': 20,                  # Descending segments
    'magnitude_descend': 21,        # Magnitude of descent
    'cumulative_descend': 22,       # Total descent accumulated
    'lat_lon_morton_encoded': 23    # Latitide and Longitiitude encoded
}

# Add a new column to data
new_column = np.zeros((data.shape[0], 1))
data = np.hstack((data, new_column)) # Latitide and Longitiitude encoded

# Dictionary mapping keys to descriptive labels for visualization or reporting
keys = {
    'lat': 'Latitude',
    'lon': 'Longitude',
    'ground_speed': 'Ground Speed',
    'baroaltitude': 'Barometric Altitude',
    'heading': 'Heading',
    'course_change': 'Course Change',
    'magnitude_course_change': 'Magnitude of Course Change',
    'deceleration': 'Deceleration Segments',
    'magnitude_deceleration': 'Magnitude of Deceleration Segments',
    'descend': 'Descending Segments',
    'magnitude_descend': 'Magnitude of Descending Segments'
}

# Define runway threshold coordinates
runway_treshold = {'lat': 38.861193, 'lon': -77.038730}

# %% [markdown]
# Declare a dictionary to save one or many response keys:

# %%
def get_dictionary(keys):

    inc = 0
    dictionary = {}
    for key in keys:
        dictionary[key] = inc
        inc += 1
    return dictionary

# %%
def evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples):

    # Normalize y
    scaler_errors = MinMaxScaler()
```

```
    y_test_norm = scaler_errors.fit_transform(y_test.reshape(-1, 1))
    y_pred_norm = scaler_errors.transform(y_pred.reshape(-1, 1))
    y_samples_norm = (y_samples - y_train.mean()) / y_train.std()

    # Compute metrics
    mae = mean_absolute_error(y_test_norm, y_pred_norm)
    mse = mean_squared_error(y_test_norm, y_pred_norm)
    rmse = np.sqrt(mse)
    crps = ps.crps_ensemble(y_test_norm.ravel(), y_samples_norm)
    r2 = r2_score(y_test, y_pred)
    evs = explained_variance_score(y_test, y_pred)

    # Print results in CSV format
    print(f"{test_sample_id},{mae:.4f},{mse:.9f},{rmse:.4f},{np.mean(crps):.4f},{r2:.4f},{evs:.4f}
")

    return f"RMSE: {rmse:.4f}\nCRPS: {np.mean(crps):.4f}\nR2: {r2:.4f}"

# %% [markdown]
# ## Gaussian Process Regression

# %%
train_size = 50
test_size = len(test_samples)

# Randomly select tracks
random.seed(120)
tracks = random.sample(tracks, train_size)

for i in range(len(test_samples)):
    tracks.append(test_samples[i])

syn_size = 1000000

# %%
def encode(heading, distance):
    encoded = heading * 1000 + distance * 1000
    return int(encoded)

def decode(encoded):
    heading = int(encoded // 1000)
    distance = (encoded - heading * 1000) / 1000.0
    return heading, distance

# %%
class HammingKernel(Kernel):
    def __init__(self, length_scale=1.0):
        self.length_scale = length_scale

    def __call__(self, X, Y=None, eval_gradient=False):
        if Y is None:
            Y = X
        # Convert to integers
        X_int = X.astype(np.uint64)
        Y_int = Y.astype(np.uint64)

        # Compute Hamming distance
        K = np.zeros((X_int.shape[0], Y_int.shape[0]))
        for i in range(X_int.shape[0]):
            for j in range(Y_int.shape[0]):
                xor_val = X_int[i, 0] ^ Y_int[j, 0]
                hamming_dist = bin(xor_val).count("1")
                K[i, j] = np.exp(-hamming_dist / self.length_scale)

        if eval_gradient:
```

```
            # Gradient is not implemented for custom kernel
            return K, np.zeros((X.shape[0], Y.shape[0], 1))
        return K

    def diag(self, X):
        return np.ones(X.shape[0])

    def is_stationary(self):
        return False

@property
def hyperparameter_length_scale(self):
    return ("length_scale", "numeric", (1e-5, 1e5))

@property
def theta(self):
    return np.log([self.length_scale])

@property
def bounds(self):
    return np.log([[1e-5, 1e5]])

def clone_with_theta(self, theta):
    return HammingKernel(length_scale=np.exp(theta[0]))

# %%
predictor_keys = get_dictionary(['position', 'lat', 'lon', 'course_change',
'accumulated_distance'])
response_keys = get_dictionary(['encoded'])

predictors = np.zeros((len(tracks) * num_segments, len(predictor_keys)))
responses = np.zeros((len(tracks) * num_segments, 7))

start_row = 0
for track in tracks:
    next_row = start_row + num_segments
    predictors[start_row:next_row,:] = track[:, [columns['position'], columns['lat'],
columns['lon'], columns['course_change'], columns['accumulated_distance']]]

    response_column = np.zeros((track.shape[0], 7))

    for res_key in response_keys:
        for i in range(len(track)):

            response_column[i, 0] = track[i, columns['lat']]
            response_column[i, 1] = track[i, columns['lon']]
            response_column[i, 2] = track[i, columns['heading']]
            response_column[i, 3] = track[i, columns['distance_between_points']]

            heading = float(int(track[i, columns['heading']]))  # scalar
            distance = float(track[i, columns['distance_between_points']])  # scalar

            # encoded = morton_encode_heading_distance(heading, distance)
            # heading_decoded, distance_decoded = morton_decode_heading_distance(encoded)

            encoded = encode(heading, distance)
            heading_decoded, distance_decoded = decode(encoded)

            response_column[i, 4] = encoded
            response_column[i, 5] = heading_decoded
            response_column[i, 6] = distance_decoded

    responses[start_row:next_row,:] = response_column
    start_row = next_row
```

```
response = responses[:, 4]

# %%
# Create and fit the scaler
scaler = StandardScaler()
X = scaler.fit_transform(predictors)

# %%
def get_gpr_models(response_keys):

    gpr_models = {}

    for key, value in response_keys.items():
        kernel = HammingKernel(length_scale=10.0) + Matern(length_scale=1.0, nu=1.5) +
WhiteKernel(noise_level=1.0)
        gpr_model = GaussianProcessRegressor(kernel=kernel,
                                             # optimizer='fmin_l_bfgs_b', n_restarts_optimizer=10,
                                             optimizer=None,
                                             copy_X_train=True,
                                             normalize_y=True,
                                             random_state=100)
        gpr_models[key] = gpr_model

    return gpr_models

gpr_models = get_gpr_models(response_keys)

# %%
for res_key in response_keys:
    y = response
    X_train = X[:train_size * num_segments,]
    y_train = y[:train_size * num_segments]

    y_scaler = MinMaxScaler()
    y_train_scaled = y_scaler.fit_transform(y_train.reshape(-1, 1))

    gpr_models[res_key].fit(X_train, y_train_scaled) # Regression
    print(f"\nModel Stats for: {res_key}")
    print(f"Kernel: {gpr_models[res_key].kernel_}")  # Optimized kernel after fitting
    print(f"Log Marginal Likelihood: {gpr_models[res_key].log_marginal_likelihood_value_:.6f}")  #
Model fit quality

# %%
# joblib.dump(gpr_models, os.path.join(output_dir,
"PKL\GPR_CombinedMultiResponse_HeadingDistance.pkl") )

# %%
# Load the model (as an alternative to a new GPR model)
# gpr_models = joblib.load(os.path.join(output_dir,
"PKL\GPR_CombinedMultiResponse_HeadingDistance.pkl"))

# %%
for res_key in response_keys:
    gpr = gpr_models[res_key]

    # Optimized kernel
    print(gpr.kernel_)

    # Kernel hyperparameters
    print(gpr.kernel_.get_params())

    # Log marginal likelihood at fitted hyperparameters
    print(gpr.log_marginal_likelihood_value_)

    # Or recompute it
```

```
    print(gpr.log_marginal_likelihood())

# %%
for res_key in response_keys:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Encoded Multi-Response\n\n", weight='bold',
fontsize = 12)

    start_row = train_size * num_segments

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        # Mean prediction and standard deviation
        y_pred_scaled, y_pred_stdev_scaled = gpr_models[res_key].predict(X_test, return_std=True)

        # Invert scaled predictions and standard deviations back to original scale
        y_pred = y_scaler.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()
        y_pred_stdev = y_pred_stdev_scaled * y_scaler.data_range_[0] # MinMaxScaler

        # Generate samples from posterior distribution
        y_samples_scaled = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)
        # Rescale posterior samples to original scale
        y_samples = y_scaler.inverse_transform(y_samples_scaled)
        # Compute mean across all samples
        samples_mean = np.mean(y_samples, axis=1)

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        # Plot training data
        start_train_row = 0
        for j in range(train_size):
            next_train_row = start_train_row + num_segments
            ax.scatter(data_points, y_train[start_train_row:next_train_row],
                       color='yellow', edgecolor='darkgoldenrod', marker=".", alpha=0.6,
                       label="Training Data" if j == 0 else "")
            start_train_row = next_train_row

        ax.plot(data_points, y_test, color='black', linewidth=2, zorder=3, label="Seed")
        ax.plot(data_points, y_pred, color='black', linewidth=5, zorder=4) # Outline
        ax.plot(data_points, y_pred, color='darkorange', zorder=5, linewidth=4, label="Mean
Prediction")
        ax.fill_between(
            data_points.ravel(),
            y_pred - 1.96 * y_pred_stdev,
            y_pred + 1.96 * y_pred_stdev,
            alpha=0.4,
            zorder=-1,
            label=r"95% confidence interval",
        )

        # Add accuracy text inside the subplot
        ax.text(0.05, 0.85,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
```

```
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

        ax.set_title(f"Encoded Values\n{tracks[train_size + i][0, columns['flight_id']]}",
weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"{res_key}")
        ax.legend()

# %%
def compute_decoded_ci(y_pred, y_pred_stdev, decode_func, confidence=0.95):
    """
    Compute decoded confidence intervals for heading and distance.
    """
    # Compute z-score for confidence level
    z = norm.ppf((1 + confidence) / 2)  # e.g., 1.96 for 95%

    # Confidence interval bounds in original units
    lower_bound = y_pred - z * y_pred_stdev
    upper_bound = y_pred + z * y_pred_stdev

    # Decode bounds
    heading_ci = np.zeros((len(y_pred), 2))
    distance_ci = np.zeros((len(y_pred), 2))

    for i in range(len(y_pred)):
        heading_ci[i, 0], distance_ci[i, 0] = decode_func(lower_bound[i])
        heading_ci[i, 1], distance_ci[i, 1] = decode_func(upper_bound[i])

    return heading_ci, distance_ci

# %%
for res_key in ['heading', 'distance']:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Encoded Multi-Response\n\n", weight='bold',
fontsize = 12)

    start_row = train_size * num_segments

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        # Mean prediction and standard deviation
        y_pred_scaled, y_pred_stdev_scaled = gpr_models['encoded'].predict(X_test,
return_std=True)

        # Invert scaled predictions and standard deviations back to original scale
        y_pred = y_scaler.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()
        y_pred_stdev = y_pred_stdev_scaled * y_scaler.data_range_[0] # MinMaxScaler

        # Generate samples from posterior distribution
        y_samples_scaled = gpr_models['encoded'].sample_y(X_test, n_samples=syn_size)
```

```
        # Rescale posterior samples to original scale
        y_samples = y_scaler.inverse_transform(y_samples_scaled)
        # Compute mean across all samples
        samples_mean = np.mean(y_samples, axis=1)

        # Inverse transform to get original values
        X_test_original = scaler.inverse_transform(X_test)

        heading_decoded = np.zeros((y_pred.shape[0], 2))
        distance_decoded = np.zeros((y_pred.shape[0], 2))

        for j in range(len(y_pred)):
            heading_decoded[j, 0], distance_decoded[j, 0] = decode(y_test[j])
            heading_decoded[j, 1], distance_decoded[j, 1] = decode(y_pred[j])

        # Confidence intervals
        heading_ci, distance_ci = compute_decoded_ci(y_pred, y_pred_stdev, decode,
confidence=0.95)

        # Evaluate performance:
        scaler_errors = MinMaxScaler()

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        if res_key == 'heading':

            # Decode heading for each sample
            heading_samples = np.zeros((y_samples.shape[0], y_samples.shape[1]))
            for sample in range(y_samples.shape[1]):
                for row in range(y_samples.shape[0]):
                    heading_samples[row, sample], _ = decode(y_samples[row, sample])  # Only
heading

            # Compute mean heading across samples
            heading_samples_mean = np.mean(heading_samples, axis=1)

            ax.plot(data_points, heading_decoded[:, 0], color='black', linewidth=2, label="Seed")
            ax.plot(data_points, heading_decoded[:, 1], color='black', linewidth=5, zorder=4) #
Outline
            ax.plot(data_points, heading_decoded[:, 1], color='darkorange', linewidth=4, zorder=5,
label="Mean Prediction")
            ax.set_title(f"Heading\n{tracks[train_size + i][0, columns['flight_id']]}",
weight='bold')
            ax.fill_between(
                data_points.ravel(),
                heading_ci[:,0],
                heading_ci[:,1],
                alpha=0.4,
                zorder=-1,
                label=r"95% confidence interval",
            )

            # Add accuracy text inside the subplot
            ax.text(0.05, 0.02,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

        if res_key == 'distance':
            ax.plot(data_points, distance_decoded[:, 0], color='black', linewidth=2, label="Seed")
```

```
            ax.plot(data_points, distance_decoded[:, 1], color='black', linewidth=3, zorder=4) #
Outline
            ax.plot(data_points, distance_decoded[:, 1], color='darkorange', linewidth=2,
zorder=5, label="Mean Prediction")
            ax.set_title(f"Distance Between Points\n{tracks[train_size + i][0,
columns['flight_id']]}", weight='bold')
            ax.fill_between(
                data_points.ravel(),
                distance_ci[:,0],
                distance_ci[:,1],
                alpha=0.4,
                zorder=-1,
                label=r"95% confidence interval",
            )

            # Add accuracy text inside the subplot
            ax.text(0.05, -0.02,
                evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"{res_key}")
        ax.legend()

# %%
def next_lat_lon(lat0, lon0, heading, distance_nm):
    """
    Calculate next latitude and longitude given:
    - lat0, lon0: starting point in degrees
    - heading: in degrees (0° = North, clockwise)
    - distance_nm: distance in nautical miles
    """
    R = 6371000  # Earth radius in meters
    distance_m = distance_nm * 1852  # Convert nautical miles to meters

    lat0_rad = math.radians(lat0)
    lon0_rad = math.radians(lon0)
    heading_rad = math.radians(heading)

    lat2 = math.asin(math.sin(lat0_rad) * math.cos(distance_m / R) +
                     math.cos(lat0_rad) * math.sin(distance_m / R) * math.cos(heading_rad))

    lon2 = lon0_rad + math.atan2(math.sin(heading_rad) * math.sin(distance_m / R) *
math.cos(lat0_rad),
                                 math.cos(distance_m / R) - math.sin(lat0_rad) * math.sin(lat2))

    return math.degrees(lat2), math.degrees(lon2)

# %%
for res_key in ['position']:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Encoded Multi-Response\n\n", weight='bold',
fontsize = 12)

    start_row = train_size * num_segments

    # Flatten axes for easy indexing
    axes = axes.flatten()
```

```
    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        # Mean prediction and standard deviation
        y_pred_scaled, y_pred_stdev_scaled = gpr_models['encoded'].predict(X_test,
return_std=True)

        # Invert scaled predictions and standard deviations back to original scale
        y_pred = y_scaler.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()
        y_pred_stdev = y_pred_stdev_scaled * y_scaler.data_range_[0] # MinMaxScaler

        # Inverse transform to get original values
        X_test_original = scaler.inverse_transform(X_test)

        heading_decoded = np.zeros((y_pred.shape[0], 2))
        distance_decoded = np.zeros((y_pred.shape[0], 2))

        for j in range(len(y_pred)):
            heading_decoded[j, 0], distance_decoded[j, 0] = decode(y_test[j]) # test
            heading_decoded[j, 1], distance_decoded[j, 1] = decode(y_pred[j]) # pred

        # Confidence intervals
        heading_ci, distance_ci = compute_decoded_ci(y_pred, y_pred_stdev, decode,
confidence=0.95)

        latitudes = np.zeros((y_pred.shape[0], 2))
        longitudes = np.zeros((y_pred.shape[0], 2))
        lat_ci = np.zeros((y_pred.shape[0], 2)) # [lower bound, upper bound]
        lon_ci = np.zeros((y_pred.shape[0], 2))
        lat0 = X_test_original[0, 1] # Set reference point
        lon0 = X_test_original[0, 2]

        for j in range(len(y_pred)):
            heading_decoded[j, 0], distance_decoded[j, 0] = decode(y_test[j])
            heading_decoded[j, 1], distance_decoded[j, 1] = decode(y_pred[j])

            if j == 0:
                latitudes[0, 0], longitudes[0, 0] = next_lat_lon(lat0, lon0, heading_decoded[j,
0], distance_decoded[j, 0]) # test
                latitudes[0, 1], longitudes[0, 1] = next_lat_lon(lat0, lon0, heading_decoded[j,
1], distance_decoded[j, 1]) # pred
                # Confidence interval bounds for predicted lat/lon
                lat_ci[j, 0], lon_ci[j, 0] = next_lat_lon(lat0, lon0, heading_ci[j, 0],
distance_ci[j, 0])  # lower bound
                lat_ci[j, 1], lon_ci[j, 1] = next_lat_lon(lat0, lon0, heading_ci[j, 1],
distance_ci[j, 1])  # upper bound
            else:
                latitudes[j, 0], longitudes[j, 0] = next_lat_lon(latitudes[j - 1, 0], longitudes[j
- 1, 0], heading_decoded[j, 0], distance_decoded[j, 0]) # test
                latitudes[j, 1], longitudes[j, 1] = next_lat_lon(latitudes[j - 1, 1], longitudes[j
- 1, 1], heading_decoded[j, 1], distance_decoded[j, 1]) # pred
                # Confidence interval bounds for predicted lat/lon
                lat_ci[j, 0], lon_ci[j, 0] = next_lat_lon(latitudes[j-1, 1], longitudes[j-1, 1],
heading_ci[j, 0], distance_ci[j, 0])
                lat_ci[j, 1], lon_ci[j, 1] = next_lat_lon(latitudes[j-1, 1], longitudes[j-1, 1],
heading_ci[j, 1], distance_ci[j, 1])

        # Plot Data
        ax = axes[i]  # Select the correct subplot
```

```
        ax.plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, zorder=6,
label='RWY 19')
        ax.plot(X_test_original[:, 2], X_test_original[:, 1], color='blue', linewidth=2,
label="Actual Seed Track")
        ax.plot(longitudes[:, 0], latitudes[:, 0], color='black', linewidth=2,
label="Reconstructed Seed Track")

        # Generate posterior samples
        y_samples_scaled = gpr_models['encoded'].sample_y(X_test, n_samples=syn_size)
        sample_latitudes = np.zeros((y_samples_scaled.shape[0], y_samples_scaled.shape[1]))
        sample_longitudes = np.zeros((y_samples_scaled.shape[0], y_samples_scaled.shape[1]))

        for sample in range(y_samples_scaled.shape[1]):
            y_sample = y_scaler.inverse_transform(y_samples_scaled[:, sample].reshape(-1,
1)).flatten()
            heading_sample = np.zeros(len(y_sample))
            distance_sample = np.zeros(len(y_sample))
            for row in range(len(y_sample)):
                heading_sample[row], distance_sample[row] = decode(y_sample[row])

            heading_decoded_sample = np.zeros(y_sample.shape[0])
            distance_decoded_sample = np.zeros(y_sample.shape[0])
            latitude_sample = np.zeros(len(y_sample))
            longitude_sample = np.zeros(len(y_sample))
            # Decode and compute lat/lon path for this sample
            for row in range(y_sample.shape[0]):
                heading_decoded_sample[row], distance_decoded_sample[row] = decode(y_sample[row])
                if row == 0:
                    latitude_sample[row], longitude_sample[row] = next_lat_lon(lat0, lon0,
heading_decoded_sample[row], distance_decoded_sample[row])
                else:
                    latitude_sample[row], longitude_sample[row] = next_lat_lon(latitude_sample[row
- 1], longitude_sample[row - 1], heading_decoded_sample[row], distance_decoded_sample[row])

            sample_latitudes[:, sample] = latitude_sample
            sample_longitudes[:, sample] = longitude_sample

        # Compute batch means
        batch_size = 100
        num_batches = sample_latitudes.shape[1] // batch_size
        batch_mean_lat = []
        batch_mean_lon = []
        for b in range(num_batches):
            start_idx = b * batch_size
            end_idx = start_idx + batch_size
            batch_mean_lat.append(np.mean(sample_latitudes[:, start_idx:end_idx], axis=1))
            batch_mean_lon.append(np.mean(sample_longitudes[:, start_idx:end_idx], axis=1))

        batch_mean_lat = np.array(batch_mean_lat).T
        batch_mean_lon = np.array(batch_mean_lon).T

        # Compute overall mean and confidence band
        mean_lat = np.mean(sample_latitudes, axis=1)
        mean_lon = np.mean(sample_longitudes, axis=1)
        z = norm.ppf(0.95)
        lat_std = np.std(sample_latitudes, axis=1)
        lon_std = np.std(sample_longitudes, axis=1)
        lat_lower = mean_lat - z * lat_std
        lat_upper = mean_lat + z * lat_std
        lon_lower = mean_lon - z * lon_std
        lon_upper = mean_lon + z * lon_std

        # Plot posterior mean and confidence band
        ax.plot(mean_lon, mean_lat, color='black', linewidth=4, zorder=6)
        ax.plot(mean_lon, mean_lat, color='red', linewidth=3, zorder=6, label="Posterior Mean")
```

```
        ax.set_title(f"Flight Track\n{tracks[train_size + i][0, columns['flight_id']]}",
weight='bold')
        ax.set_xlabel('Longitude')
        ax.set_ylabel("Latitude")
        if i == 0:
            ax.legend()

    plt.show()

# %%
def next_lat_lon_opposite(lat0, lon0, heading, distance_nm):
    """
    Calculate next latitude and longitude in the opposite direction given:
    - lat0, lon0: starting point in degrees
    - heading: in degrees (0° = North, clockwise)
    - distance_nm: distance in nautical miles
    """

    R = 6371000  # Earth radius in meters
    distance_m = distance_nm * 1852  # Convert nautical miles to meters

    # Reverse heading by 180 degrees
    opposite_heading = (heading + 180) % 360

    # Convert to radians
    lat0_rad = math.radians(lat0)
    lon0_rad = math.radians(lon0)
    heading_rad = math.radians(opposite_heading)

    # Compute new latitude
    lat2 = math.asin(math.sin(lat0_rad) * math.cos(distance_m / R) +
                     math.cos(lat0_rad) * math.sin(distance_m / R) * math.cos(heading_rad))

    # Compute new longitude
    lon2 = lon0_rad + math.atan2(math.sin(heading_rad) * math.sin(distance_m / R) *
math.cos(lat0_rad),
                                 math.cos(distance_m / R) - math.sin(lat0_rad) * math.sin(lat2))

    return math.degrees(lat2), math.degrees(lon2)

# %%
syn_size = 1000

# %%
for res_key in ['position']:

    # Create figure and axes
    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Encoded Multi-Response\n\n", weight='bold',
fontsize = 12)

    start_row = train_size * num_segments
    axes = axes.flatten()

    syn_tracks = []  # Collect dictionaries for all test samples

    for i in range(test_size):
        # Prepare test data
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row
```

```
        # GP mean and std
        y_pred_scaled, y_pred_stdev_scaled = gpr_models['encoded'].predict(X_test,
return_std=True)
        y_pred = y_scaler.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()
        y_pred_stdev = y_pred_stdev_scaled * y_scaler.data_range_[0]  # MinMaxScaler

        # Original track
        X_test_original = scaler.inverse_transform(X_test)

        # Decode headings/distances
        heading_decoded = np.zeros((len(y_pred), 2))
        distance_decoded = np.zeros((len(y_pred), 2))
        for j in range(len(y_pred)):
            heading_decoded[j, 0], distance_decoded[j, 0] = decode(y_test[j])
            heading_decoded[j, 1], distance_decoded[j, 1] = decode(y_pred[j])

        # Confidence intervals
        heading_ci, distance_ci = compute_decoded_ci(y_pred, y_pred_stdev, decode,
confidence=0.95)

        # Runway threshold reference
        lat0 = X_test_original[-1, 1]
        lon0 = X_test_original[-1, 2]

        # Arrays for reconstructed lat/lon
        latitudes = np.zeros((len(y_pred), 2))
        longitudes = np.zeros((len(y_pred), 2))
        lat_ci = np.zeros((y_pred.shape[0], 2)) # [lower bound, upper bound]
        lon_ci = np.zeros((y_pred.shape[0], 2))

        for j in range(len(y_pred) - 1, -1, -1):
            if j == len(y_pred) - 1:
                # Start from runway threshold
                latitudes[j, 0], longitudes[j, 0] = next_lat_lon_opposite(lat0, lon0,
heading_decoded[j, 0], distance_decoded[j, 0])
                latitudes[j, 1], longitudes[j, 1] = next_lat_lon_opposite(lat0, lon0,
heading_decoded[j, 1], distance_decoded[j, 1])
                lat_ci[j, 0], lon_ci[j, 0] = next_lat_lon_opposite(lat0, lon0, heading_ci[j, 0],
distance_ci[j, 0])
                lat_ci[j, 1], lon_ci[j, 1] = next_lat_lon_opposite(lat0, lon0, heading_ci[j, 1],
distance_ci[j, 1])
            else:
                # Continue moving backwards
                latitudes[j, 0], longitudes[j, 0] = next_lat_lon_opposite(latitudes[j + 1, 0],
longitudes[j + 1, 0], heading_decoded[j, 0], distance_decoded[j, 0])
                latitudes[j, 1], longitudes[j, 1] = next_lat_lon_opposite(latitudes[j + 1, 1],
longitudes[j + 1, 1], heading_decoded[j, 1], distance_decoded[j, 1])
                lat_ci[j, 0], lon_ci[j, 0] = next_lat_lon_opposite(latitudes[j + 1, 1],
longitudes[j + 1, 1], heading_ci[j, 0], distance_ci[j, 0])
                lat_ci[j, 1], lon_ci[j, 1] = next_lat_lon_opposite(latitudes[j + 1, 1],
longitudes[j + 1, 1], heading_ci[j, 1], distance_ci[j, 1])

        # Plot actual and reconstructed tracks
        ax = axes[i]
        ax.plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, zorder=7,
label='RWY 19')
        ax.plot(X_test_original[:, 2], X_test_original[:, 1], color='blue', linewidth=2,
label="Actual Seed Track")
        ax.plot(longitudes[:, 0], latitudes[:, 0], color='black', linewidth=2,
label="Reconstructed Seed Track")
        # ax.plot(longitudes[:, 1], latitudes[:, 1], color='darkorange', linewidth=2, zorder=5,
label="Prediction Mean")

        # Generate posterior samples
        y_samples_scaled = gpr_models['encoded'].sample_y(X_test, n_samples=syn_size)
```

```
        sample_latitudes = np.zeros((y_samples_scaled.shape[0], y_samples_scaled.shape[1]))
        sample_longitudes = np.zeros((y_samples_scaled.shape[0], y_samples_scaled.shape[1]))

        for sample in range(y_samples_scaled.shape[1]):
            y_sample = y_scaler.inverse_transform(y_samples_scaled[:, sample].reshape(-1,
1)).flatten()
            heading_sample = np.zeros(len(y_sample))
            distance_sample = np.zeros(len(y_sample))
            for row in range(len(y_sample)):
                heading_sample[row], distance_sample[row] = decode(y_sample[row])
                distance_sample[row] = distance_sample[row]

            latitude_sample = np.zeros(len(y_sample))
            longitude_sample = np.zeros(len(y_sample))
            for row in range(len(y_sample) - 1, -1, -1):
                if row == len(y_sample) - 1:
                    latitude_sample[row], longitude_sample[row] = next_lat_lon_opposite(lat0,
lon0, heading_sample[row], distance_sample[row])
                else:
                    latitude_sample[row], longitude_sample[row] =
next_lat_lon_opposite(latitude_sample[row + 1], longitude_sample[row + 1], heading_sample[row],
distance_sample[row])

            sample_latitudes[:, sample] = latitude_sample
            sample_longitudes[:, sample] = longitude_sample

        # Compute overall mean and confidence band
        mean_lat = np.mean(sample_latitudes, axis=1)
        mean_lon = np.mean(sample_longitudes, axis=1)
        z = norm.ppf(0.95)
        lat_std = np.std(sample_latitudes, axis=1)
        lon_std = np.std(sample_longitudes, axis=1)
        lat_lower = mean_lat - z * lat_std
        lat_upper = mean_lat + z * lat_std
        lon_lower = mean_lon - z * lon_std
        lon_upper = mean_lon + z * lon_std

        # Plot posterior mean and confidence band
        ax.plot(mean_lon, mean_lat, color='black', linewidth=4, zorder=6)
        ax.plot(mean_lon, mean_lat, color='red', linewidth=3, zorder=6, label="Posterior Mean")

        # Compute batch means
        batch_size = 100
        num_batches = sample_latitudes.shape[1] // batch_size
        batch_mean_lat = []
        batch_mean_lon = []
        for b in range(num_batches):
            start_idx = b * batch_size
            end_idx = start_idx + batch_size
            batch_mean_lat.append(np.mean(sample_latitudes[:, start_idx:end_idx], axis=1))
            batch_mean_lon.append(np.mean(sample_longitudes[:, start_idx:end_idx], axis=1))

        batch_mean_lat = np.array(batch_mean_lat).T
        batch_mean_lon = np.array(batch_mean_lon).T

        # Plot batch means
        for idx in range(batch_mean_lat.shape[1]):
            label = f"{batch_mean_lat.shape[1]} Samples (Batch = {batch_size})" if idx == 0 else
None
            ax.plot(batch_mean_lon[:, idx], batch_mean_lat[:, idx], color='gray', linewidth=1,
alpha=1.0, label=label)

        test_sample_id = test_samples[i][0, columns['flight_id']]

        for col in range(batch_mean_lat.shape[1]):
```

```
            for row in range(batch_mean_lat.shape[0]):
                entry = {
                    'test_sample': test_sample_id,
                    'syn_track_index': col,
                    'position': row,
                    'lat': batch_mean_lat[row, col],
                    'lon': batch_mean_lon[row, col]
                }
                syn_tracks.append(entry)

        # Labels
        ax.set_title(f"Flight Track\n{tracks[train_size + i][0, columns['flight_id']]}",
weight='bold')
        ax.set_xlabel('Longitude')
        ax.set_ylabel("Latitude")
        if i == 0:
            ax.legend()

    plt.show()

# %%
file_path = output_dir + 'SynTracks_CombinedMultiResponse_25TestSamples_Batch100.csv'

with open(file_path, 'w') as file:
    # Write header
    headers = "test_sample_id,syn_track_index,position,lat,lon,alt,speed"
    file.write(headers)

    # Write rows
    for i in range(len(syn_tracks)):
        line = (
            f"\n{syn_tracks[i]['test_sample']},"
            f"{syn_tracks[i]['syn_track_index']},"
            f"{syn_tracks[i]['position']},"
            f"{syn_tracks[i]['lat']},"
            f"{syn_tracks[i]['lon']},"
        )
        file.write(line)
```

Model 4: Deep GP with Encoded Multi-Response

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from math import radians, sin, cos, acos
from collections import defaultdict
from mpl_toolkits.mplot3d import Axes3D
import plotly.graph_objects as go
from sklearn.gaussian_process import GaussianProcessClassifier, GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, Kernel, Product, ConstantKernel,
ExpSineSquared, Matern, DotProduct, RationalQuadratic
from sklearn.metrics import  mean_absolute_error, mean_squared_error, r2_score,
explained_variance_score
from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix,
ConfusionMatrixDisplay
import deepgp
import GPy
from GPy.core.parameterization import transformations
import joblib
import random
import math
import properscoring as ps

# %% [markdown]
# # Read Data

# %%
# Define the output directory where the file is stored
output_dir = "C:\\Files\\"

# Build the full file path by joining the directory and file name
file_path = os.path.join(output_dir, "RWY_19_Northern_Cluster_Train_Tracks.csv")
file_path_test_samples = os.path.join(output_dir, "RWY_19_Northern_Cluster_Seed_Tracks_6.csv")

# Read the CSV file into a pandas DataFrame
data = pd.read_csv(file_path)
data_test_samples = pd.read_csv(file_path_test_samples)

# Number of segments to process for each flight track
num_segments = 200

# Generate an array of segment indices (0 to num_segments-1)
data_points = np.arange(num_segments)

# Collect flight_ids used in test samples
test_flight_ids = set(data_test_samples['flight_id'].unique())

# Build test samples
test_samples = []
for _, group in data_test_samples.groupby('flight_id'):
    if len(group) == num_segments:
        test_samples.append(group.to_numpy())

# Build tracks excluding test flight_ids
tracks = []
for flight_id, group in data.groupby('flight_id'):
    if len(group) == num_segments and flight_id not in test_flight_ids:
        tracks.append(group.to_numpy())
```

```
# %%
# Dictionary mapping column names to their index positions in the dataset
columns = {
    'position': 0,                  # Position in segment
    'flight_id': 1,                 # Unique flight identifier
    'time': 2,                      # Timestamp
    'lat': 3,                       # Latitude
    'lon': 4,                       # Longitude
    'ground_speed': 5,              # Ground speed
    'baroaltitude': 6,              # Barometric altitude
    'vertrate': 7,                  # Vertical rate (climb/descent speed)
    'heading': 8,                   # Aircraft heading
    'distance_between_points':9,    # Distance traveled between points
    'accumulated_distance': 10,     # Cumulative Distance traveled
    'duration': 11,                 # Duration of the segment
    'departure_airport': 12,        # Departure airport code
    'arrival_airport': 13,          # Arrival airport code
    'course_change': 14,            # Change in course direction
    'magnitude_course_change': 15,  # Magnitude of course change
    'cumulative_course_change': 16, # Total course change accumulated
    'deceleration': 17,             # Deceleration segments
    'magnitude_deceleration': 18,   # Magnitude of deceleration
    'cumulative_deceleration': 19,  # Total deceleration accumulated
    'descend': 20,                  # Descending segments
    'magnitude_descend': 21,        # Magnitude of descent
    'cumulative_descend': 22,       # Total descent accumulated
}

# Dictionary mapping keys to descriptive labels for visualization or reporting
keys = {
    'lat': 'Latitude',
    'lon': 'Longitude',
    'ground_speed': 'Ground Speed',
    'baroaltitude': 'Barometric Altitude',
    'heading': 'Heading',
    'course_change': 'Course Change',
    'magnitude_course_change': 'Magnitude of Course Change',
    'deceleration': 'Deceleration Segments',
    'magnitude_deceleration': 'Magnitude of Deceleration Segments',
    'descend': 'Descending Segments',
    'magnitude_descend': 'Magnitude of Descending Segments'
}

# Define runway threshold coordinates
runway_treshold = {'lat': 38.861193, 'lon': -77.038730}

# %% [markdown]
# Declare a dictionary to save one or many response keys:

# %%
def get_dictionary(keys):

    inc = 0
    dictionary = {}
    for key in keys:
        dictionary[key] = inc
        inc += 1
    return dictionary

# %%
def evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples):

    # Normalize y
    scaler_errors = MinMaxScaler()
    y_test_norm = scaler_errors.fit_transform(y_test.reshape(-1, 1))
```

```
    y_pred_norm = scaler_errors.transform(y_pred.reshape(-1, 1))
    y_samples_norm = (y_samples - y_train.mean()) / y_train.std()

    # Compute metrics
    mae = mean_absolute_error(y_test_norm, y_pred_norm)
    mse = mean_squared_error(y_test_norm, y_pred_norm)
    rmse = np.sqrt(mse)
    crps = ps.crps_ensemble(y_test_norm.ravel(), y_samples_norm)
    r2 = r2_score(y_test, y_pred)
    evs = explained_variance_score(y_test, y_pred)

    # Print results in CSV format
    print(f"{test_sample_id},{mae:.4f},{mse:.9f},{rmse:.4f},{np.mean(crps):.4f},{r2:.4f},{evs:.4f}
")

    return f"RMSE: {rmse:.4f}\nCRPS: {np.mean(crps):.4f}\nR2: {r2:.4f}"

# %% [markdown]
# ## Deep Gaussian Process Regression

# %%
train_size = 50
test_size = len(test_samples)

# Randomly select tracks
random.seed(120)
tracks = random.sample(tracks, train_size)

for i in range(len(test_samples)):
    tracks.append(test_samples[i])

syn_size = 1000000

# %%
def encode(heading, distance):
    encoded = heading * 1000 + distance * 1000
    return int(encoded)

def decode(encoded):
    heading = int(encoded // 1000)
    distance = (encoded - heading * 1000) / 1000.0
    return heading, distance

# %%
predictor_keys = get_dictionary(['position', 'lat', 'lon', 'course_change',
'accumulated_distance'])
response_keys = get_dictionary(['encoded'])

predictors = np.zeros((len(tracks) * num_segments, len(predictor_keys)))
responses = np.zeros((len(tracks) * num_segments, 7))

start_row = 0
for track in tracks:
    next_row = start_row + num_segments
    predictors[start_row:next_row,:] = track[:, [columns['position'], columns['lat'],
columns['lon'], columns['course_change'], columns['accumulated_distance']]]

    response_column = np.zeros((track.shape[0], 7))

    for res_key in response_keys:
        for i in range(len(track)):

            response_column[i, 0] = track[i, columns['lat']]
            response_column[i, 1] = track[i, columns['lon']]
            response_column[i, 2] = track[i, columns['heading']]
```

```
            response_column[i, 3] = track[i, columns['distance_between_points']]

            heading = float(int(track[i, columns['heading']]))  # scalar
            distance = float(track[i, columns['distance_between_points']])  # scalar

            # encoded = morton_encode_heading_distance(heading, distance)
            # heading_decoded, distance_decoded = morton_decode_heading_distance(encoded)

            encoded = encode(heading, distance)
            heading_decoded, distance_decoded = decode(encoded)

            response_column[i, 4] = encoded
            response_column[i, 5] = heading_decoded
            response_column[i, 6] = distance_decoded

    responses[start_row:next_row,:] = response_column
    start_row = next_row

response = responses[:, 4]

# %%
# Create and fit the scaler
scaler = StandardScaler()
X = scaler.fit_transform(predictors)

# %%
def get_deep_gp_models(reponse_keys):

    deep_gp_models = {}

    for key, value in reponse_keys.items():
        deep_gp_model = None
        deep_gp_models[key] = deep_gp_model

    return deep_gp_models

deep_gp_models = get_deep_gp_models(response_keys)

# %%
def fit_deep_gp_model(X_train,y_train):

    # Deep GP Model Construction
    Q = 3  # number of latent dimensions
    y_train_shape = np.zeros((y_train.shape[0], 1))
    y_train_shape[:, 0] = y_train

    layers = [y_train_shape.shape[1], Q, 1, X_train.shape[1]]  # [1, 1, 3]
    inits = ['PCA'] * (len(layers) - 1)
    num_inducing = 100
    back_constraint = False

    # Composite kernel for input layer
    composite_kernel = (
        GPy.kern.Matern32(input_dim=layers[-1], ARD=True)
        # + GPy.kern.Linear(input_dim=layers[-1])
        + GPy.kern.Bias(input_dim=layers[-1])
    )

    kernels = [
        GPy.kern.Matern32(input_dim=layers[1], ARD=True),   # hidden layer 1
        GPy.kern.Matern32(input_dim=layers[2], ARD=True),  # hidden layer 2
        composite_kernel                                   # input layer
    ]

    # Build Deep GP
```

```
    m = deepgp.DeepGP(layers,
                      Y=y_train_shape,
                      X=X_train,
                      inits=inits,
                      kernels=kernels,
                      num_inducing=num_inducing,
                      back_constraint=back_constraint)

    # Optimization:
    # Make sure initial noise variance gives a reasonable signal to noise ratio.
    # Fix to that value for a few iterations to avoid early local minima
    for i in range(len(m.layers)):
        output_var = m.layers[i].Y.var() if i==0 else m.layers[i].Y.mean.var()
        m.layers[i].Gaussian_noise.variance = output_var*0.01
        m.layers[i].Gaussian_noise.variance.fix()
        m.layers[i].likelihood.variance.fix(warning=False)
    m.optimize(max_iters=200, messages=True)

    for i in range(len(m.layers)):
        m.layers[i].likelihood.variance.constrain_positive(warning=False)
    m.optimize(max_iters=200, messages=True)

    # Unfix noise variance now that we have initialized the model:
    for i in range(len(m.layers)):
        m.layers[i].Gaussian_noise.variance.unfix()
    m.optimize(max_iters=1000, messages=True)

    return m

# %%
for res_key in response_keys:
    y = response
    X_train = X[:train_size * num_segments,]
    y_train = y[:train_size * num_segments]


    y_scaler = MinMaxScaler()
    y_train_scaled = y_scaler.fit_transform(y_train.reshape(-1, 1)).flatten()


    deep_gp_models[res_key] = fit_deep_gp_model(X_train, y_train_scaled)

# %%
# joblib.dump(deep_gp_models, os.path.join(output_dir, "PKL\DeepGP_Heading_Distance.pkl") )

# %%
# Load the model (as an alternative to a new GPR model)
# deep_gp_models = joblib.load(os.path.join(output_dir, "PKL\DeepGP_Heading_Distance.pkl"))

# %%
for res_key in response_keys:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Deep GP - Encoded Multi-Response\n", weight='bold', fontsize = 12)

    start_row = train_size * num_segments

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
```

```
        start_row = next_row

        y_pred_scaled, _ = deep_gp_models[res_key].predict(X_test)
        y_pred = y_scaler.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        # Plot training data
        start_train_row = 0
        for j in range(train_size):
            next_train_row = start_train_row + num_segments
            ax.scatter(data_points, y_train[start_train_row:next_train_row],
                       color='yellow', edgecolor='darkgoldenrod', marker=".", alpha=0.6,
                       label="Training Data" if j == 0 else "")
            start_train_row = next_train_row

        ax.plot(data_points, y_test, color='black', linewidth=2, zorder=3, label="Seed Sample")
        ax.plot(data_points, y_pred, color='black', linewidth=5, zorder=4) # Outline
        ax.plot(data_points, y_pred, color='darkorange', zorder=5, linewidth=4, label="Mean
Prediction")
        ax.set_title(f"Encoded Values\n{tracks[train_size + i][0, columns['flight_id']]}",
weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"{res_key}")
        ax.legend()

        # Evaluate performance:
        scaler_errors = MinMaxScaler()
        y_test_norm = scaler_errors.fit_transform(y_test.reshape(-1, 1))
        y_pred_norm = scaler_errors.transform(y_pred.reshape(-1, 1))

            # Compute metrics
        mse = mean_squared_error(y_test_norm, y_pred_norm)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_test, y_pred)
        print(f",{rmse:.4f},{r2:.4f}")


# %%
for res_key in ['heading', 'distance']:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Deep GP - Encoded Multi-Response\n", weight='bold', fontsize = 12)

    # Reset start_row for each res_key
    start_row = train_size * num_segments

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        heading_original = responses[start_row:next_row, 2]
        distance_original = responses[start_row:next_row, 3]
        start_row = next_row

        # Mean prediction and standard deviation
        y_pred_scaled, _ = deep_gp_models['encoded'].predict(X_test)
        y_pred = y_scaler.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()

        n_samples = 10
```

```
        # Generate samples from the last layer
        y_mean, y_var = deep_gp_models['encoded'].layers[-1].predict(X_test)
        y_samples =
np.random.randn(n_samples,X_test.shape[0],y_mean.shape[1])*np.sqrt(y_var)+y_mean

        # Propagate samples backward through hidden layers
        for layer in deep_gp_models['encoded'].layers[-2::-1]:
            # Flatten samples for prediction
            y_mean, y_var = layer.predict(y_samples.reshape(n_samples * X_test.shape[0], -1))
            y_samples = np.random.randn(*y_var.shape) * np.sqrt(y_var) + y_mean

        # Reshape samples: (num_test_points, num_samples)
        y_hat_samples = np.zeros((X_test.shape[0], n_samples))
        for s in range(n_samples):
            for j in range(X_test.shape[0]):
                y_hat_samples[j, s] = y_samples[j + X_test.shape[0] * s][0]

        # Inverse transform samples back to original scale
        y_hat_samples_original = y_scaler.inverse_transform(y_hat_samples)

        # Inverse transform to get original values
        X_test_original = scaler.inverse_transform(X_test)

        heading_decoded = np.zeros((y_pred.shape[0], 2))
        distance_decoded = np.zeros((y_pred.shape[0], 2))
        heading_samples = np.zeros((y_hat_samples_original.shape[0],
y_hat_samples_original.shape[1]))
        distance_samples = np.zeros((y_hat_samples_original.shape[0],
y_hat_samples_original.shape[1]))

        for j in range(len(y_pred)):
            heading_decoded[j, 0], distance_decoded[j, 0] = decode(y_test[j])
            heading_decoded[j, 1], distance_decoded[j, 1] = decode(y_pred[j])

        for row in range(y_hat_samples_original.shape[0]):
            for col in range(y_hat_samples_original.shape[1]):
                heading_samples[row, col], distance_samples[row, col] =
decode(y_hat_samples_original[row, col])

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        if res_key == 'heading':
            ax.plot(data_points, heading_decoded[:, 0], color='black', linewidth=2, label="Seed
Sample")
            ax.plot(data_points, heading_decoded[:, 1], color='black', linewidth=5, zorder=4) #
Outline
            ax.plot(data_points, heading_decoded[:, 1], color='darkorange', linewidth=4, zorder=5,
label="Mean Prediction")

            # Confidence interval
            heading_lower = np.percentile(heading_samples, 2.5, axis=1)
            heading_upper = np.percentile(heading_samples, 97.5, axis=1)

            ax.fill_between(
                data_points,
                heading_lower,
                heading_upper,
                alpha=0.25,
                zorder=2,
                label=r"95% confidence interval"
                )

            ax.set_title(f"Heading\n{tracks[train_size + i][0, columns['flight_id']]}",
weight='bold')
```

```
        if res_key == 'distance':
            ax.plot(data_points, distance_decoded[:, 0], color='black', linewidth=2, label="Seed
Sample")
            ax.plot(data_points, distance_decoded[:, 1], color='black', linewidth=5, zorder=4) #
Outline
            ax.plot(data_points, distance_decoded[:, 1], color='darkorange', linewidth=4,
zorder=5, label="Mean Prediction")

            # Confidence interval
            distance_lower = np.percentile(distance_samples, 2.5, axis=1)
            distance_upper = np.percentile(distance_samples, 97.5, axis=1)

            ax.fill_between(
                data_points,
                distance_lower,
                distance_upper,
                alpha=0.25,
                zorder=2,
                label=r"95% confidence interval"
                )

            ax.set_title(f"Distance Between Points\n{tracks[train_size + i][0,
columns['flight_id']]}", weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"{res_key}")
        ax.legend()

    plt.show()

# %%
def next_lat_lon_opposite(lat0, lon0, heading, distance_nm):
    """
    Calculate next latitude and longitude in the opposite direction given:
    - lat0, lon0: starting point in degrees
    - heading: in degrees (0° = North, clockwise)
    - distance_nm: distance in nautical miles
    """

    R = 6371000  # Earth radius in meters
    distance_m = distance_nm * 1852  # Convert nautical miles to meters

    # Reverse heading by 180 degrees
    opposite_heading = (heading + 180) % 360

    # Convert to radians
    lat0_rad = math.radians(lat0)
    lon0_rad = math.radians(lon0)
    heading_rad = math.radians(opposite_heading)

    # Compute new latitude
    lat2 = math.asin(math.sin(lat0_rad) * math.cos(distance_m / R) +
                     math.cos(lat0_rad) * math.sin(distance_m / R) * math.cos(heading_rad))

    # Compute new longitude
    lon2 = lon0_rad + math.atan2(math.sin(heading_rad) * math.sin(distance_m / R) *
math.cos(lat0_rad),
                                 math.cos(distance_m / R) - math.sin(lat0_rad) * math.sin(lat2))

    return math.degrees(lat2), math.degrees(lon2)

# %%
for res_key in ['position']:

    fig, axes = plt.subplots(
```

```
        nrows=2,
        ncols=3,
        figsize=(15, 10),
        tight_layout=True
    )

    fig.subplots_adjust(hspace=0.4)

    fig.suptitle(
        "Deep GP - Encoded Multi-Response\n",
        weight='bold',
        fontsize=12
    )

    # Reset start_row for each res_key
    start_row = train_size * num_segments

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):

        next_row = start_row + num_segments

        X_test = X[start_row:next_row, ]
        y_test = y[start_row:next_row]

        heading_original = responses[start_row:next_row, 2]
        distance_original = responses[start_row:next_row, 3]

        start_row = next_row

        # Mean prediction
        y_pred_scaled, _ = deep_gp_models['encoded'].predict(X_test)

        y_pred = y_scaler.inverse_transform(
            y_pred_scaled.reshape(-1, 1)
        ).flatten()

        n_samples = 100

        # Generate samples from the last layer
        y_mean, y_var = deep_gp_models['encoded'].layers[-1].predict(X_test)

        y_samples = (
            np.random.randn(
                n_samples,
                X_test.shape[0],
                y_mean.shape[1]
            ) * np.sqrt(y_var) + y_mean
        )

        # Propagate samples backward through hidden layers
        for layer in deep_gp_models['encoded'].layers[-2::-1]:

            y_mean, y_var = layer.predict(
                y_samples.reshape(
                    n_samples * X_test.shape[0],
                    -1
                )
            )

            y_samples = (
                np.random.randn(*y_var.shape)
                * np.sqrt(y_var)
```

```
            + y_mean
        )

    # Reshape samples
    y_hat_samples = np.zeros((X_test.shape[0], n_samples))

    for s in range(n_samples):
        for j in range(X_test.shape[0]):
            y_hat_samples[j, s] = (
                y_samples[j + X_test.shape[0] * s][0]
            )

    # Inverse transform samples
    y_hat_samples_original = y_scaler.inverse_transform(
        y_hat_samples
    )

    # Inverse transform to original values
    X_test_original = scaler.inverse_transform(X_test)

    heading_decoded = np.zeros((y_pred.shape[0], 2))
    distance_decoded = np.zeros((y_pred.shape[0], 2))

    latitudes = np.zeros((y_pred.shape[0], 2))
    longitudes = np.zeros((y_pred.shape[0], 2))

    heading_samples = np.zeros(
        (
            y_hat_samples_original.shape[0],
            y_hat_samples_original.shape[1]
        )
    )

    distance_samples = np.zeros(
        (
            y_hat_samples_original.shape[0],
            y_hat_samples_original.shape[1]
        )
    )

    # Decode actual and predicted values
    for j in range(len(y_pred)):

        heading_decoded[j, 0], distance_decoded[j, 0] = decode(y_test[j])

        heading_decoded[j, 1], distance_decoded[j, 1] = decode(y_pred[j])

        distance_decoded[j, 1] = distance_decoded[j, 1]

    # Runway threshold anchor
    lat_rw = runway_treshold['lat']
    lon_rw = runway_treshold['lon']

    # Backward reconstruction
    for j in range(len(y_pred) - 1, -1, -1):

        if j == len(y_pred) - 1:

            latitudes[j, 0], longitudes[j, 0] = (
                next_lat_lon_opposite(
                    lat_rw,
                    lon_rw,
                    heading_decoded[j, 0],
                    distance_decoded[j, 0]
                )
```

```
            )

            latitudes[j, 1], longitudes[j, 1] = (
                next_lat_lon_opposite(
                    lat_rw,
                    lon_rw,
                    heading_decoded[j, 1],
                    distance_decoded[j, 1]
                )
            )

        else:

            latitudes[j, 0], longitudes[j, 0] = (
                next_lat_lon_opposite(
                    latitudes[j + 1, 0],
                    longitudes[j + 1, 0],
                    heading_decoded[j, 0],
                    distance_decoded[j, 0]
                )
            )

            latitudes[j, 1], longitudes[j, 1] = (
                next_lat_lon_opposite(
                    latitudes[j + 1, 1],
                    longitudes[j + 1, 1],
                    heading_decoded[j, 1],
                    distance_decoded[j, 1]
                )
            )

    # Decode posterior samples
    for row in range(y_hat_samples_original.shape[0]):
        for col in range(y_hat_samples_original.shape[1]):

            heading_samples[row, col], distance_samples[row, col] = (
                decode(y_hat_samples_original[row, col])
            )

            distance_samples[row, col] = distance_samples[row, col]

    # ======================================================
    # Reconstruct posterior sample trajectories
    # ======================================================

    lat_samples = np.zeros((X_test.shape[0], n_samples))
    lon_samples = np.zeros((X_test.shape[0], n_samples))

    for col in range(n_samples):

        for row in range(X_test.shape[0] - 1, -1, -1):

            if row == X_test.shape[0] - 1:

                lat_samples[row, col], lon_samples[row, col] = (
                    next_lat_lon_opposite(
                        lat_rw,
                        lon_rw,
                        heading_samples[row, col],
                        distance_samples[row, col]
                    )
                )

            else:
```

```
                lat_samples[row, col], lon_samples[row, col] = (
                    next_lat_lon_opposite(
                        lat_samples[row + 1, col],
                        lon_samples[row + 1, col],
                        heading_samples[row, col],
                        distance_samples[row, col]
                    )
                )

        # 95% confidence interval
        lat_lower = np.percentile(lat_samples, 2.5, axis=1)
        lat_upper = np.percentile(lat_samples, 97.5, axis=1)

        # ======================================================
        # Plot
        # ======================================================

        ax = axes[i]

        ax.plot(
            runway_treshold['lon'],
            runway_treshold['lat'],
            'ko',
            markersize=10,
            zorder=6,
            label='RWY 19'
        )

        ax.plot(
            X_test_original[:, 2],
            X_test_original[:, 1],
            color='blue',
            linewidth=2,
            label="Actual Data"
        )

        ax.plot(
            longitudes[:, 0],
            latitudes[:, 0],
            color='black',
            linewidth=2,
            label="Seed Sample"
        )


        # 95% confidence interval
        ax.fill_between(
            longitudes[:, 1],
            lat_lower,
            lat_upper,
            alpha=0.25,
            zorder=2,
            label='95% CI'
        )

        # Posterior trajectories
        for s in range(n_samples):
            ax.plot(
                lon_samples[:, s],
                lat_samples[:, s],
                color='gray',
                alpha=0.5,
                linewidth=0.5,
                zorder=1
            )
```

```
        ax.plot(
            longitudes[:, 1],
            latitudes[:, 1],
            color='black',
            linewidth=3,
            zorder=4
        )

        ax.plot(
            longitudes[:, 1],
            latitudes[:, 1],
            color='darkorange',
            linewidth=2,
            zorder=5,
            label="Mean Prediction"
        )

        ax.set_title(
            f"Flight Track\n{tracks[train_size + i][0, columns['flight_id']]}",
            weight='bold'
        )

        ax.set_xlabel('Longitude')
        ax.set_ylabel("Latitude")
        ax.legend()

    plt.show()
```

Model 5: GPR with Principal Components

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import radians, sin, cos, acos
from collections import defaultdict
from mpl_toolkits.mplot3d import Axes3D
import plotly.graph_objects as go
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.multioutput import MultiOutputRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel, Kernel, Product, ConstantKernel,
ExpSineSquared, Matern, DotProduct, RationalQuadratic
from sklearn.metrics import  mean_absolute_error, mean_squared_error, r2_score,
explained_variance_score
from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler
from scipy.stats import norm
import GPy
import joblib
import random
import math
import properscoring as ps
from sklearn.decomposition import PCA

# %% [markdown]
# # Read Data

# %%
# Define the output directory where the file is stored
output_dir = "C:\\Files\\"

# Build the full file path by joining the directory and file name
file_path = os.path.join(output_dir, "RWY_19_Northern_Cluster_Train_Tracks.csv")
file_path_test_samples = os.path.join(output_dir, "RWY_19_Northern_Cluster_Seed_Tracks_6.csv")

# Read the CSV file into a pandas DataFrame
data = pd.read_csv(file_path)
data_test_samples = pd.read_csv(file_path_test_samples)

# Number of segments to process for each flight track
num_segments = 200

# Generate an array of segment indices (0 to num_segments-1)
data_points = np.arange(num_segments)

# Collect flight_ids used in test samples
test_flight_ids = set(data_test_samples['flight_id'].unique())

# Build test samples
test_samples = []
for _, group in data_test_samples.groupby('flight_id'):
    if len(group) == num_segments:
        test_samples.append(group.to_numpy())

# Build tracks excluding test flight_ids
tracks = []
for flight_id, group in data.groupby('flight_id'):
    if len(group) == num_segments and flight_id not in test_flight_ids:
        tracks.append(group.to_numpy())

# %%
# Dictionary mapping column names to their index positions in the dataset
```

```
columns = {
    'position': 0,                  # Position in segment
    'flight_id': 1,                 # Unique flight identifier
    'time': 2,                      # Timestamp
    'lat': 3,                       # Latitude
    'lon': 4,                       # Longitude
    'ground_speed': 5,              # Ground speed
    'baroaltitude': 6,              # Barometric altitude
    'vertrate': 7,                  # Vertical rate (climb/descent speed)
    'heading': 8,                   # Aircraft heading
    'distance_between_points':9,    # Distance traveled between points
    'accumulated_distance': 10,     # Cumulative Distance traveled
    'duration': 11,                 # Duration of the segment
    'departure_airport': 12,        # Departure airport code
    'arrival_airport': 13,          # Arrival airport code
    'course_change': 14,            # Change in course direction
    'magnitude_course_change': 15,  # Magnitude of course change
    'cumulative_course_change': 16, # Total course change accumulated
    'deceleration': 17,             # Deceleration segments
    'magnitude_deceleration': 18,   # Magnitude of deceleration
    'cumulative_deceleration': 19,  # Total deceleration accumulated
    'descend': 20,                  # Descending segments
    'magnitude_descend': 21,        # Magnitude of descent
    'cumulative_descend': 22,       # Total descent accumulated
}

# Dictionary mapping keys to descriptive labels for visualization or reporting
keys = {
    'lat': 'Latitude',
    'lon': 'Longitude',
    'ground_speed': 'Ground Speed',
    'baroaltitude': 'Barometric Altitude',
    'heading': 'Heading',
    'course_change': 'Course Change',
    'magnitude_course_change': 'Magnitude of Course Change',
    'deceleration': 'Deceleration Segments',
    'magnitude_deceleration': 'Magnitude of Deceleration Segments',
    'descend': 'Descending Segments',
    'magnitude_descend': 'Magnitude of Descending Segments'
}

# Define runway threshold coordinates
runway_treshold = {'lat': 38.861193, 'lon': -77.038730}

# %% [markdown]
# # Gaussian Process Models

# %% [markdown]
# Declare a dictionary to save one or many response keys:

# %%
def get_dictionary(keys):

    inc = 0
    dictionary = {}
    for key in keys:
        dictionary[key] = inc
        inc += 1
    return dictionary

# %%
def evaluate_model_performance(test_sample_id, y_train, y_test, y_pred, y_samples):

    # Normalize y
    scaler_errors = MinMaxScaler()
```

```
    y_test_norm = scaler_errors.fit_transform(y_test.reshape(-1, 1))
    y_pred_norm = scaler_errors.transform(y_pred.reshape(-1, 1))
    y_samples_norm = (y_samples - y_train.mean()) / y_train.std()

    # Compute metrics
    mae = mean_absolute_error(y_test_norm, y_pred_norm)
    mse = mean_squared_error(y_test_norm, y_pred_norm)
    rmse = np.sqrt(mse)
    crps = ps.crps_ensemble(y_test_norm.ravel(), y_samples_norm)
    r2 = r2_score(y_test, y_pred)
    evs = explained_variance_score(y_test, y_pred)

    # Print results in CSV format
    print(f"{test_sample_id},{mae:.4f},{mse:.9f},{rmse:.4f},{np.mean(crps):.4f},{r2:.4f},{evs:.4f}
")

    return f"RMSE: {rmse:.4f}\nCRPS: {np.mean(crps):.4f}\nR2: {r2:.4f}"

# %% [markdown]
# Indicate training and test data sizes as well as a number of generated samples :

# %%
train_size = 50
test_size = len(test_samples)

# Randomly select tracks
random.seed(120)
tracks = random.sample(tracks, train_size)

for i in range(len(test_samples)):
    tracks.append(test_samples[i])

syn_size = 10000000

# %%
encoded_response_keys = get_dictionary(['lat', 'lon', 'baroaltitude', 'ground_speed'])

num_tracks = len(tracks)
deltas = np.zeros((num_tracks * num_segments, len(encoded_response_keys)))

row = 0
for track in tracks:
    for i in range(track.shape[0]):
        if i == 0:
            delta_lat = 0.0
            delta_lon = 0.0
            delta_alt = 0.0
            delta_speed = 0.0
        else:
            delta_lat = track[i, columns['lat']] - track[i-1, columns['lat']]
            delta_lon = track[i, columns['lon']] - track[i-1, columns['lon']]
            delta_alt = track[i, columns['baroaltitude']] - track[i-1, columns['baroaltitude']]
            delta_speed = track[i, columns['ground_speed']] - track[i-1, columns['ground_speed']]

        deltas[row] = [delta_lat, delta_lon, delta_alt, delta_speed]
        row += 1

# %%
scaler_pca = StandardScaler()
deltas_scaled = scaler_pca.fit_transform(deltas)

pca = PCA(n_components=1)
pca.fit(deltas_scaled)

# %%
```

```
def encode_pca(delta_lat, delta_lon, delta_alt, delta_speed):

    # Original 4D delta vector
    pair = np.array([[delta_lat, delta_lon, delta_alt, delta_speed]])

    # Scale the 4D vector
    pair_scaled = scaler_pca.transform(pair)

    # Project onto PC1 (single component)
    pc1 = pca.transform(pair_scaled)[0, 0]

    # Reconstruct back to 4D using only PC1
    recon_scaled = pca.inverse_transform([[pc1]])

    # Compute 4D residual in scaled space
    residual_scaled = pair_scaled - recon_scaled

    # Encoded vector = PC1 + 4 residuals
    return np.array([
        pc1,
        residual_scaled[0, 0],   # residual_lat
        residual_scaled[0, 1],   # residual_lon
        residual_scaled[0, 2],   # residual_alt
        residual_scaled[0, 3]    # residual_speed
    ])

# %%
def decode_pca(encoded):

    pc1 = encoded[0]
    r_lat = encoded[1]
    r_lon = encoded[2]
    r_alt = encoded[3]
    r_speed = encoded[4]

    # Reconstruct PC1-only version in scaled space
    recon_scaled = pca.inverse_transform([[pc1]])

    # Add 4D residual
    original_scaled = recon_scaled + np.array([[r_lat, r_lon, r_alt, r_speed]])

    # Unscale the 4D vector
    original = scaler_pca.inverse_transform(original_scaled)[0]

    delta_lat = original[0]
    delta_lon = original[1]
    delta_alt = original[2]
    delta_speed = original[3]

    return delta_lat, delta_lon, delta_alt, delta_speed

# %%
predictor_keys = get_dictionary(['position', 'lat', 'lon', 'heading', 'course_change',
'baroaltitude', 'descend', 'ground_speed', 'deceleration'])
encoded_response_keys = get_dictionary(['lat', 'lon', 'baroaltitude', 'ground_speed'])
response_keys = get_dictionary(['encoded'])

predictors = np.zeros((num_tracks * num_segments, len(predictor_keys)))
responses  = np.zeros((num_tracks * num_segments, len(encoded_response_keys) + 1)) # [pc1,
residual_lat, residual_lon, residual_alt, residual_speed]

row = 0
for track in tracks:
    for i in range(track.shape[0]):
```

```
        # Fill predictors
        predictors[row, :] = track[i, [
            columns['position'], columns['lat'], columns['lon'],
            columns['heading'], columns['course_change'],
            columns['baroaltitude'], columns['descend'],
            columns['ground_speed'], columns['deceleration']
        ]]

        # Compute deltas
        if i == 0:
            delta_lat = 0.0
            delta_lon = 0.0
            delta_alt = 0.0
            delta_speed = 0.0
        else:
            delta_lat = track[i, columns['lat']] - track[i-1, columns['lat']]
            delta_lon = track[i, columns['lon']] - track[i-1, columns['lon']]
            delta_alt = track[i, columns['baroaltitude']] - track[i-1, columns['baroaltitude']]
            delta_speed = track[i, columns['ground_speed']] - track[i-1, columns['ground_speed']]

        # Encode
        encoded = encode_pca(delta_lat, delta_lon, delta_alt, delta_speed)

        responses[row] = encoded  # [pc1, residual_lat, residual_lon, residual_alt,
residual_speed]

        row += 1

# %%
# Create and fit the scaler
scaler = StandardScaler()
X = scaler.fit_transform(predictors)

# %%
kernels = {}
def get_gpr_models(response_keys):

    gpr_models = {}

    for key, value in response_keys.items():
        kernel = (
            ConstantKernel(1.0, (1e-3, 1e3)) * (Matern(length_scale=1.0, length_scale_bounds=(1e-
2, 1e3), nu=1.5) + RationalQuadratic(length_scale=1.0, alpha=1.0, alpha_bounds=(1e1, 1e12)))
            + WhiteKernel(noise_level=1.0, noise_level_bounds=(1e-10, 1e3))
        )
        kernels[key] = kernel
        gpr_model = GaussianProcessRegressor(kernel=kernel,
                                             n_restarts_optimizer=5,
                                             copy_X_train=True,
                                             normalize_y=False,
                                             random_state=100)
        gpr_models[key] = gpr_model

    return gpr_models

gpr_models = get_gpr_models(response_keys)

# %%
for res_key in response_keys:
    y = responses
    X_train = X[:train_size * num_segments,]
    y_train = y[:train_size * num_segments]

    gpr_models[res_key].fit(X_train, y_train) # Regression
    print(f"\nModel Stats for: {res_key}")
```

```
    print(f"Kernel: {gpr_models[res_key].kernel_}")
    print(f"Log Marginal Likelihood: {gpr_models[res_key].log_marginal_likelihood_value_:.6f}")  #
Model fit quality

# %%
# joblib.dump(gpr_models, os.path.join(output_dir, "PKL\GPR_PCA.pkl") )

# %%
# Load the model (as an alternative to a new GPR model)
# gpr_models = joblib.load(os.path.join(output_dir, "PKL\GPR_PCA.pkl"))

# %%
for res_key in response_keys:
    gpr = gpr_models[res_key]

    # Optimized kernel
    print(gpr.kernel_)

    # Kernel hyperparameters
    print(gpr.kernel_.get_params())

    # Log marginal likelihood at fitted hyperparameters
    print(gpr.log_marginal_likelihood_value_)

    # Or recompute it
    print(gpr.log_marginal_likelihood())

# %%
for res_key in response_keys:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Combined Multi-Response\n\n", weight='bold',
fontsize = 12)

    start_row = train_size * num_segments

    # Flatten axes for easy indexing
    axes = axes.flatten()

    for i in range(test_size):
        next_row = start_row + num_segments
        X_test = X[start_row:next_row,]
        y_test = y[start_row:next_row]
        start_row = next_row

        test_sample_id = test_samples[i][0, columns['flight_id']]

        y_pred, y_pred_stdev = gpr_models[res_key].predict(X_test, return_std=True)

        # Generate samples from posterior distribution
        y_samples = gpr_models[res_key].sample_y(X_test, n_samples=syn_size)

        # Plot Data
        ax = axes[i]  # Select the correct subplot

        # Plot training data
        start_train_row = 0
        for j in range(train_size):
            next_train_row = start_train_row + num_segments
            ax.scatter(data_points, y_train[start_train_row:next_train_row, 0],
                       color='yellow', edgecolor='darkgoldenrod', marker=".", alpha=0.6,
                       label="Training Data" if j == 0 else "")
            start_train_row = next_train_row
```

```
        ax.plot(data_points, y_test[:, 0], color='black', linewidth=2, zorder=3, label="Test
Sample")
        ax.plot(data_points, y_pred[:, 0], color='black', linewidth=5, zorder=4) # Outline
        ax.plot(data_points, y_pred[:, 0], color='darkorange', zorder=5, linewidth=4,
label="Prediction Mean")
        ax.fill_between(
            data_points.ravel(),
            y_pred[:, 0] - 1.96 * y_pred_stdev[:, 0],
            y_pred[:, 0] + 1.96 * y_pred_stdev[:, 0],
            alpha=0.4,
            zorder=-1,
            label=r"95% confidence interval",
        )

        # Add accuracy text inside the subplot
        ax.text(0.05, 0.85,
                evaluate_model_performance(test_sample_id, y_train[:, 0], y_test[:, 0], y_pred[:,
0], y_samples[:, 0]),
                transform=ax.transAxes,
                fontsize=10,
                color='darkgreen',
                ha='left', va='bottom',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

        ax.set_title(f"Encoded Values\n{tracks[train_size + i][0, columns['flight_id']]}",
weight='bold')
        ax.set_xlabel('Data Points')
        ax.set_ylabel(f"{res_key}")
        if i == 0:
            ax.legend()

# %%
syn_latitudes = []
syn_longitudes = []
syn_altitudes = []
syn_speeds = []

# Initialize storage dictionary
results = {
    'y_test': {},
    'y_pred': {},
    'y_samples': {},
    'y_samples_mean': {},
    'y_samples_stdev': {}
}

for res_key in encoded_response_keys:

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
    fig.subplots_adjust(hspace=0.4)
    fig.suptitle("Gaussian Process Regression - Combined Multi-Response\n\n", weight='bold',
fontsize = 12)
    axes = axes.flatten()

    # Initialize dictionaries for each res_key
    results['y_test'][res_key] = []
    results['y_pred'][res_key] = []
    results['y_samples'][res_key] = []
    results['y_samples_mean'][res_key] = []
    results['y_samples_stdev'][res_key] = []

    start_row = train_size * num_segments

    for i in range(test_size):
```

```
        ax = axes[i]
        next_row = start_row + num_segments
        X_test = X[start_row:next_row, ]
        y_test = y[start_row:next_row]
        start_row = next_row

        X_test_original = scaler.inverse_transform(X_test)
        test_sample_id = test_samples[i][0, columns['flight_id']]

        # ------------------------------------------------------
        # PREDICT PCA OUTPUT (PC1 + residuals)
        # ------------------------------------------------------
        y_pred_full, y_pred_stdev_full =
gpr_models['encoded'].predict(X_test,  return_std=True)  # shape = (T, 3)

        # ------------------------------------------------------
        # DECODE TO DELTAS
        # ------------------------------------------------------
        decoded_lat_test = np.zeros(num_segments)
        decoded_lat_pred = np.zeros(num_segments)
        decoded_lon_test = np.zeros(num_segments)
        decoded_lon_pred = np.zeros(num_segments)
        decoded_alt_test = np.zeros(num_segments)
        decoded_alt_pred = np.zeros(num_segments)
        decoded_speed_test = np.zeros(num_segments)
        decoded_speed_pred = np.zeros(num_segments)

        for t in range(num_segments):
            decoded_lat_test[t], decoded_lon_test[t], decoded_alt_test[t], decoded_speed_test[t] =
decode_pca(y_test[t])
            decoded_lat_pred[t], decoded_lon_pred[t], decoded_alt_pred[t], decoded_speed_pred[t] =
decode_pca(y_pred_full[t])

        # ------------------------------------------------------
        # RECONSTRUCT THE FULL TIME-SERIES
        # ------------------------------------------------------
        series_pred = np.zeros(num_segments)
        series_test = np.zeros(num_segments)

        # initialize from last known point
        series_pred[-1] = X_test_original[-1, predictor_keys[res_key]]
        series_test[-1] = X_test_original[-1, predictor_keys[res_key]]

        for k in range(num_segments - 2, -1, -1):

            if res_key == 'lat':
                series_test[k] = series_test[k+1] - decoded_lat_test[k+1]
                series_pred[k] = series_pred[k+1] - decoded_lat_pred[k+1]
            elif res_key == 'lon':
                series_test[k] = series_test[k+1] - decoded_lon_test[k+1]
                series_pred[k] = series_pred[k+1] - decoded_lon_pred[k+1]
            elif res_key == 'baroaltitude':
                series_test[k] = series_test[k+1] - decoded_alt_test[k+1]
                series_pred[k] = series_pred[k+1] - decoded_alt_pred[k+1]
            elif res_key == 'ground_speed':
                series_test[k] = series_test[k+1] - decoded_speed_test[k+1]
                series_pred[k] = series_pred[k+1] - decoded_speed_pred[k+1]

        # ------------------------------------------------------
        # PLOT RECONSTRUCTED SERIES
        # ------------------------------------------------------
        ax.plot(data_points, series_test, color='black', linewidth=2, zorder=7, label="Seed
Sample")
        ax.plot(data_points, series_pred, color='darkorange', linewidth=3, zorder=5,
label="Prediction Mean")
```

```
# ------------------------------------------------------
# SAMPLE GENERATION (y_samples shape = (T, 3, syn_size))
# ------------------------------------------------------

T = num_segments
S = syn_size

# Generate encoded samples from posterior
y_samples = gpr_models['encoded'].sample_y(X_test, n_samples=S)

# Storage for decoded delta samples
delta_lat_samples = np.zeros((T, S))
delta_lon_samples = np.zeros((T, S))
delta_alt_samples = np.zeros((T, S))
delta_speed_samples = np.zeros((T, S))

# Decode each sample
for t in range(T):
    for j in range(S):
        encoded_vec = np.array([
            y_samples[t, 0, j],   # PC1 sample
            y_samples[t, 1, j],   # residual_lat
            y_samples[t, 2, j],   # residual_lon
            y_samples[t, 3, j],   # residual_alt
            y_samples[t, 4, j]    # residual_speed
        ])
        d_lat, d_lon, d_alt, d_speed = decode_pca(encoded_vec)
        delta_lat_samples[t, j] = d_lat
        delta_lon_samples[t, j] = d_lon
        delta_alt_samples[t, j] = d_alt
        delta_speed_samples[t, j] = d_speed

# Storage for reconstructed original sample series
y_samples_original = np.zeros((T, S))

# Reconstruct each sample backward
for j in range(S):

    # initialize final point from test data
    y_samples_original[-1, j] = X_test_original[-1, predictor_keys[res_key]]

    for row in range(T - 2, -1, -1):

        if res_key == 'lat':
            y_samples_original[row, j] = (
                y_samples_original[row + 1, j] - delta_lat_samples[row + 1, j]
            )
        elif res_key == 'lon':
            y_samples_original[row, j] = (
                y_samples_original[row + 1, j] - delta_lon_samples[row + 1, j]
            )
        elif res_key == 'baroaltitude':
            y_samples_original[row, j] = (
                y_samples_original[row + 1, j] - delta_alt_samples[row + 1, j]
            )
        elif res_key == 'ground_speed':
            y_samples_original[row, j] = (
                y_samples_original[row + 1, j] - delta_speed_samples[row + 1, j]
            )

# ------------------------------------------------------
# Sample mean & standard deviation
# ------------------------------------------------------
y_samples_mean  = y_samples_original.mean(axis=1)
```

```
        y_samples_stdev = y_samples_original.std(axis=1)

        # Plot posterior mean and confidence band
        ax.plot(data_points, y_samples_mean, color='black', linewidth=4, zorder=6)
        ax.plot(data_points, y_samples_mean, color='red', linewidth=3, zorder=6, label="Posterior
Mean")
        ax.fill_between(
            data_points,
            y_samples_mean - 1.96 * y_samples_stdev,
            y_samples_mean + 1.96 * y_samples_stdev,
            color='gray',
            alpha=0.4,
            zorder=5,
            label="Posterior 95% CI"
        )

        # ------------------------------------------------------
        # RESULTS STORING
        # ------------------------------------------------------

        # Store copies of the results
        results['y_test'][res_key].append(series_test.copy())
        results['y_pred'][res_key].append(series_pred.copy())
        results['y_samples'][res_key].append(y_samples_original.copy())
        results['y_samples_mean'][res_key].append(y_samples_mean.copy())
        results['y_samples_stdev'][res_key].append(y_samples_stdev.copy())

        # ------------------------------------------------------
        # BATCHING & STORING
        # ------------------------------------------------------
        batch_size = 25
        col = 0
        num_samples = y_samples_original.shape[1]

        for start in range(0, num_samples, batch_size):
            end = min(start + batch_size, num_samples)
            batch = y_samples_original[:, start:end]

            batch_mean = batch.mean(axis=1)

            ax.plot(data_points, batch_mean,
                linewidth=1.0, alpha=1.0, color='gray', zorder=5,
                label=f"Batch = {batch_size} Samples" if start == 0 else None
            )

            # Save syn results
            if res_key == 'lat':
                for row in range(batch_mean.shape[0]):
                    entry = {
                        'test_sample': test_sample_id,
                        'syn_track_index': col,
                        'position': row,
                        'lat': batch_mean[row]
                    }
                    syn_latitudes.append(entry)
            elif res_key == 'lon':
                for row in range(batch_mean.shape[0]):
                    entry = {
                        'test_sample': test_sample_id,
                        'syn_track_index': col,
                        'position': row,
                        'lon': batch_mean[row]
                    }
                    syn_longitudes.append(entry)
            elif res_key == 'baroaltitude':
```

```
                for row in range(batch_mean.shape[0]):
                    entry = {
                        'test_sample': test_sample_id,
                        'syn_track_index': col,
                        'position': row,
                        'alt': batch_mean[row]
                    }
                    syn_altitudes.append(entry)
            elif res_key == 'ground_speed':
                for row in range(batch_mean.shape[0]):
                    entry = {
                        'test_sample': test_sample_id,
                        'syn_track_index': col,
                        'position': row,
                        'speed': batch_mean[row]
                    }
                    syn_speeds.append(entry)
            col += 1

        ax.set_title(f"{keys[res_key]}\n{test_sample_id}", weight='bold')
        ax.set_xlabel("Data Points")
        ax.set_ylabel(keys[res_key])

        if i == 0:
            ax.legend(loc="upper right")

    plt.show()

# %%
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10), tight_layout=True)
fig.subplots_adjust(hspace=0.4)
fig.suptitle("Gaussian Process Regression - Combined Multi-Response\n\n", weight='bold',
fontsize=12)
axes = axes.flatten()

for i in range(test_size):

    ax = axes[i]
    test_sample_id = test_samples[i][0, columns['flight_id']]

    # ------------------------------------------------------
    # LOAD STORED RESULTS FROM results DICTIONARY
    # ------------------------------------------------------
    lat_test = results['y_test']['lat'][i]
    lat_pred = results['y_pred']['lat'][i]
    y_samples_original_lat = results['y_samples']['lat'][i]
    y_samples_mean_lat = results['y_samples_mean']['lat'][i]
    y_samples_stdev_lat = results['y_samples_stdev']['lat'][i]

    lon_test = results['y_test']['lon'][i]
    lon_pred = results['y_pred']['lon'][i]
    y_samples_original_lon = results['y_samples']['lon'][i]
    y_samples_mean_lon = results['y_samples_mean']['lon'][i]
    y_samples_stdev_lon = results['y_samples_stdev']['lon'][i]

    # ------------------------------------------------------
    # PLOT TEST + PREDICTION
    # ------------------------------------------------------
    ax.plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, zorder=6,
label='RWY 19')
    ax.plot(lon_test, lat_test, color='black', linewidth=2, zorder=7, label="Seed Sample")
    ax.plot(lon_pred, lat_pred, color='darkorange', linewidth=3, zorder=5, label="Prediction
Mean")

    # posterior mean (with black outline)
```

```
    ax.plot(y_samples_mean_lon, y_samples_mean_lat, color='black', linewidth=4, zorder=4)
    ax.plot(y_samples_mean_lon, y_samples_mean_lat, color='red', linewidth=3, zorder=5,
label="Posterior Mean")

    # -----------------------------------------------------
    # BATCHING
    # -----------------------------------------------------
    col = 0
    num_samples = y_samples_original_lat.shape[1]

    for start in range(0, num_samples, batch_size):
        end = min(start + batch_size, num_samples)

        batch_lat = y_samples_original_lat[:, start:end]
        batch_lon = y_samples_original_lon[:, start:end]

        batch_mean_lat = batch_lat.mean(axis=1)
        batch_mean_lon = batch_lon.mean(axis=1)

        ax.plot(batch_mean_lon, batch_mean_lat,
                linewidth=1.0, alpha=1.0, color='gray', zorder=3,
                label=f"Batch = {batch_size} Samples" if start == 0 else None
        )

    ax.set_title(f"Flight Tracks\n{test_sample_id}", weight='bold')
    ax.set_xlabel('Longitude')
    ax.set_ylabel('Latitude')

    if i == 0:
        ax.legend(loc="upper left")

plt.show()

# %%
test_sample = 'JBU955-2025-07-16'

filtered_syn_latitudes = [
    track for track in syn_latitudes if track['test_sample'] == test_sample
]

filtered_syn_longitudes = [
    track for track in syn_longitudes if track['test_sample'] == test_sample
]

filtered_syn_altitudes = [
    alt for alt in syn_altitudes if alt['test_sample'] == test_sample]

filtered_syn_speeds = [
    speed for speed in syn_speeds if speed['test_sample'] == test_sample]

# %%
file_path = output_dir + f'SynTracks_CombinedMultiDeltaResponse_Batch{batch_size}.csv'

with open(file_path, 'w') as file:
    # Write header
    headers = "test_sample_id,syn_track_index,position,lat,lon,alt,speed"
    file.write(headers)

    # Write rows
    for i in range(len(filtered_syn_latitudes)):

            line = (
                f"\n{filtered_syn_latitudes[i]['test_sample']},"
                f"{filtered_syn_latitudes[i]['syn_track_index']},"
                f"{filtered_syn_latitudes[i]['position']},"
```

```
                f"{filtered_syn_latitudes[i]['lat']},"
                f"{filtered_syn_longitudes[i]['lon']},"
                f"{filtered_syn_altitudes[i]['alt']},"
                f"{filtered_syn_speeds[i]['speed']}"
            )
            file.write(line)
```

Validation:

```
# %%
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from math import radians, sin, cos, acos
from collections import defaultdict
from mpl_toolkits.mplot3d import Axes3D
import plotly.graph_objects as go
import random
import math
from scipy.stats import wasserstein_distance, ks_2samp
from scipy.spatial.distance import jensenshannon
from scipy.stats import gaussian_kde
from sklearn.preprocessing import MinMaxScaler

# %%
# Define the output directory where the file is stored
output_dir = "C:\\Files\\"

# Build the full file path by joining the directory and file name
file_path = os.path.join(output_dir, "Historical_Seed_Tracks_Results_North_Cluster.csv") # File
name

# Load the combined results file
df = pd.read_csv(file_path)

# Convert entire DataFrame to NumPy array
hist_data = df.to_numpy()

# Create dictionary: {index: header_name}
headers = {i: col for i, col in enumerate(df.columns)}
hist_columns = {v: k for k, v in headers.items()}

# %%
# Build the full file path by joining the directory and file name
# file_path = os.path.join(output_dir, "SynTracks_CombinedMultiDeltaResponse_Batch25.csv") # File
name
# file_path = os.path.join(output_dir, "SynTracks_SingleDeltaResponse_1SeedSamples_Batch10.csv") #
File name
# file_path = output_dir + f'SynTracks_CombinedMultiDeltaResponse_Batch10.csv'
# file_path = output_dir + f'SynTracks_CombinedMultiDeltaResponse_Batch5.csv'
file_path = output_dir + f'SynTracks_CombinedMultiDeltaResponse_Batch25.csv'

# Read the CSV file into a pandas DataFrame
syn_data = pd.read_csv(file_path)
data_array = syn_data.to_numpy()

# Number of segments to process for each flight track
num_segments = 200

# Group by test_sample and syn_track_index
tracks = [data_array[i:i + num_segments] for i in range(0, len(data_array), num_segments)]

# Add column for new parameters
for idx in range(len(tracks)):
    rows, cols = tracks[idx].shape
    new_cols = np.full((rows, 20), np.zeros)
    tracks[idx] = np.hstack([tracks[idx], new_cols])

# %%
```

```
# Dictionary mapping column names to their index positions in the dataset
columns = {
    'test_sample_id': 0,
    'syn_track_index': 1,
    'position': 2,
    'lat': 3,
    'lon': 4,
    'alt': 5,
    'speed': 6,
    'accumulated_distance': 7,
    'heading': 8,
    'course_change': 9,
    'magnitude_course_change': 10,
    'cumulative_course_change': 11,
    'speed_change': 12,
    'deceleration': 12,
    'magnitude_deceleration': 14,
    'cumulative_deceleration': 15,
    'level_change': 16,
    'descend': 17,
    'magnitude_descend': 18,
    'cumulative_descend': 19,

    'lateral_dispersion_3nm': 20,
    'lateral_dispersion_6nm': 21,
    'lateral_dispersion_9nm': 22
}

# Generate an array of segment indices (0 to num_segments-1)
data_points = np.arange(num_segments)

runway_treshold = {'lat':38.861193, 'lon':-77.038730}

# %%
title = f"Synthetic Flight Tracks"
fig, ax = plt.subplot_mosaic([['top', '.'],['left', 'right']],figsize=(8,8),tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

dummy_alt_value = 1000
dummy_speed_value = 250

for track in tracks:
    # Check if 'alt' column exists
    if 'alt' not in columns:
        # Add dummy altitude column
        alt_data = np.full(track.shape[0], dummy_alt_value)
        track = np.column_stack((track, alt_data))
        columns['alt'] = track.shape[1] - 1  # Update columns index

    # Check if 'speed' column exists
    if 'speed' not in columns:
        # Add dummy speed column
        speed_data = np.full(track.shape[0], dummy_speed_value)
        track = np.column_stack((track, speed_data))
        columns['speed'] = track.shape[1] - 1  # Update columns index

    # Plot lon vs lat
    ax['top'].plot(track[:, columns['lon']], track[:, columns['lat']])
    ax['left'].plot(data_points, track[:, columns['alt']])
    ax['right'].plot(data_points, track[:, columns['speed']])

ax['top'].plot(runway_treshold['lon'], runway_treshold['lat'], 'ko', markersize=10, label='RWY
19')
```

```
ax['top'].set_title('Flight Tracks')
ax['top'].set_xlabel('Longitude')
ax['top'].set_ylabel('Latitude')
ax['top'].legend(loc='lower right')

ax['left'].set_title('Barometric Altitude')
ax['left'].set_xlabel('Data Points')
ax['left'].set_ylabel('Baroaltitude, feet')

ax['right'].set_title('Ground Speed')
ax['right'].set_xlabel('Data Points')
ax['right'].set_ylabel('Ground Speed, knots')

# %%
def get_great_circle_distance(lon1, lat1, lon2, lat2):
    r = 6371.137 # The Earth's radius in km

    # Convert to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # Compute cosine of central angle
    cos_angle = sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) * cos(lon1 - lon2)

    # Clamp the value to avoid math domain error
    cos_angle = min(1, max(-1, cos_angle))

    # Great circle distance in km
    d = r * acos(cos_angle)

    # Convert km to nm:
    d = d * 0.5399568035

    return d

# %%
# Function to compute distribution comparison metrics
def compute_distribution_metrics(hist_data, syn_data, bins):
    hist_data = np.array(hist_data).reshape(-1, 1)
    syn_data = np.array(syn_data).reshape(-1, 1)

    # Normalize both datasets to [0, 1] range
    scaler = MinMaxScaler()
    combined = np.vstack((hist_data, syn_data))
    scaler.fit(combined)
    hist_data = scaler.transform(hist_data).flatten()
    syn_data = scaler.transform(syn_data).flatten()

    # Wasserstein Distance
    wd = wasserstein_distance(hist_data, syn_data)

    # Kolmogorov-Smirnov Test
    ks_stat, ks_pvalue = ks_2samp(hist_data, syn_data)

    # Jensen-Shannon Divergence (requires normalized histograms)
    hist1, bin_edges = np.histogram(hist_data, bins=bins, density=True)
    hist2, _ = np.histogram(syn_data, bins=bin_edges, density=True)

    # Avoid zero probabilities for JSD
    hist1 += 1e-12
    hist2 += 1e-12

    jsd = jensenshannon(hist1, hist2)

    print(f"Wasserstein Distance,{wd:.4f}") # Lower values = more similar distributions
```

```
    print(f"KS Statistic,{ks_stat:.4f},p-value,{ks_pvalue:.4e}") # KS statistic closer to 0 =
distributions are similar, p-value > 0.05 → cannot reject similarity
    print(f"Jensen-Shannon Divergence,{jsd:.4f}") # 0 = identical distributions, closer to 1 =
very different
    print()

    return {
        "WD": wd,
        "KSS": ks_stat,
        "KSP": ks_pvalue,
        "JSD": jsd
    }

# %%
for track in tracks:
    track_length = 0
    track[0, columns['accumulated_distance']] = 0
    for i in range(1, len(track)):
        lon1 = track[i-1, columns['lon']]
        lat1 = track[i-1, columns['lat']]
        lon2 = track[i, columns['lon']]
        lat2 = track[i, columns['lat']]
        d = get_great_circle_distance(lon1, lat1, lon2, lat2)
        track_length += d
        track[i, columns['accumulated_distance']] = track_length # Cumulative distance

# %%
syn_distances_at_rt = np.zeros(len(tracks))
syn_altitudes_at_rt = np.zeros(len(tracks))
syn_ground_speeds_at_rt = np.zeros(len(tracks))

for i in range(len(tracks)):
    syn_distances_at_rt[i] = tracks[i][-1, columns['accumulated_distance']]
    syn_altitudes_at_rt[i] = tracks[i][-1, columns['alt']]
    syn_ground_speeds_at_rt[i] = tracks[i][-1, columns['speed']]

# %%
title = "Measurements at the Runway Threshold"
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 5), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize=14)

metrics = compute_distribution_metrics(hist_data[:, hist_columns['along_track_distance']],
syn_distances_at_rt, bins=20)
ax.text(0.45, 0.97,
    f"Wasserstein Distance = {metrics['WD']:.4f}\nKS Statistic = {metrics['KSS']:.4f}\np-value =
{metrics['KSP']:.4e}\nJensen-Shannon Divergence = {metrics['JSD']:.4f}",
    transform=ax.transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
ax.hist(hist_data[:, hist_columns['along_track_distance']], bins=20,  color='gray', alpha=1.0,
label='Historical', density=True)
ax.hist(syn_distances_at_rt, bins=20, alpha=0.7, label='Synthetic', density=True)
ax.set_title('Along Track Distance')
ax.set_xlabel('Distance, nautical miles')
ax.set_ylabel('Probability Density')
ax.legend(loc='lower right')

plt.show()

# %%
def get_treshold_point(lat, lon, heading, distance):
    r = 6371.137 # The Earth's radius in km
```

```
    # Convert to radians
    lat1 = math.radians(lat)
    lon1 = math.radians(lon)
    bearing = math.radians(heading + 180)

    # Angular distance
    d_r = (distance * 1.852) / r # Convert nm to km

    # Destination latitude
    lat2 = math.asin(math.sin(lat1) * math.cos(d_r) +
                     math.cos(lat1) * math.sin(d_r) * math.cos(bearing))

    # Destination longitude
    lon2 = lon1 + math.atan2(math.sin(bearing) * math.sin(d_r) * math.cos(lat1),
                             math.cos(d_r) - math.sin(lat1) * math.sin(lat2))

    return math.degrees(lat2), math.degrees(lon2)

# %%
def process_tracks(tracks, distances_info):
    results = {}

    for distance_nm, nominal_index, dispersion_col in distances_info:
        positions = []
        track_data = []

        # Generate circle positions
        for i in range(360):
            lat_nm, lon_nm = get_treshold_point(runway_treshold['lat'], runway_treshold['lon'], i,
distance_nm)
            positions.append([i, lat_nm, lon_nm])
            if i == nominal_index:
                nominal_lat = lat_nm
                nominal_lon = lon_nm

        # Process each track
        for track in tracks:
            for i in reversed(range(len(track) - 2)):
                d1 = get_great_circle_distance(track[i, columns['lon']], track[i, columns['lat']],
                                               runway_treshold['lon'], runway_treshold['lat'])
                d2 = get_great_circle_distance(track[i - 1, columns['lon']], track[i - 1,
columns['lat']],
                                               runway_treshold['lon'], runway_treshold['lat'])

                if d1 <= distance_nm and d2 > distance_nm:
                    # Find closest point on the circle
                    closest_point = min(positions, key=lambda pos: get_great_circle_distance(
                        track[i - 1, columns['lon']], track[i - 1, columns['lat']], pos[2],
pos[1]))
                    distance = get_great_circle_distance(track[i - 1, columns['lon']], track[i -
1, columns['lat']],
                                                         closest_point[2], closest_point[1])

                    # Create threshold value array
                    treshold_value = np.zeros((track.shape[1]))
                    treshold_value[columns['lat']] = closest_point[1]
                    treshold_value[columns['lon']] = closest_point[2]
                    treshold_value[columns['alt']] = track[i, columns['alt']]
                    treshold_value[columns['speed']] = track[i, columns['speed']]
                    treshold_value[columns['accumulated_distance']] += distance
                    treshold_value[columns[dispersion_col]] = get_great_circle_distance(
                        track[i - 1, columns['lon']], track[i - 1, columns['lat']], nominal_lon,
nominal_lat)

                    track_data.append(treshold_value)
```

```
                    break

        results[distance_nm] = {'positions': positions, 'track_data': track_data}

    return results


# Call the function for 3NM, 6NM, 9NM
distances_info = [
    (3, 147, 'lateral_dispersion_3nm'),
    (6, 140, 'lateral_dispersion_6nm'),
    (9, 136, 'lateral_dispersion_9nm')
]

combined_results = process_tracks(tracks, distances_info)

# Access results:
syn_track_data_at_3nm = combined_results[3]['track_data']
syn_track_data_at_6nm = combined_results[6]['track_data']
syn_track_data_at_9nm = combined_results[9]['track_data']

# %%
# Convert data to NumPy arrays
syn_track_data_at_3nm = np.array(syn_track_data_at_3nm)
syn_track_data_at_6nm = np.array(syn_track_data_at_6nm)
syn_track_data_at_9nm = np.array(syn_track_data_at_9nm)

# Extract synthetic data
syn_lateral_dispersion_3nm = syn_track_data_at_3nm[:, columns['lateral_dispersion_3nm']]
syn_vertical_dispersion_3nm = syn_track_data_at_3nm[:, columns['alt']]
syn_speed_dispersion_3nm = syn_track_data_at_3nm[:, columns['speed']]
syn_lateral_dispersion_6nm = syn_track_data_at_6nm[:, columns['lateral_dispersion_6nm']]
syn_vertical_dispersion_6nm = syn_track_data_at_6nm[:, columns['alt']]
syn_speed_dispersion_6nm = syn_track_data_at_6nm[:, columns['speed']]
syn_lateral_dispersion_9nm = syn_track_data_at_9nm[:, columns['lateral_dispersion_9nm']]
syn_vertical_dispersion_9nm = syn_track_data_at_9nm[:, columns['alt']]
syn_speed_dispersion_9nm = syn_track_data_at_9nm[:, columns['speed']]

# %%
title = "Measurements at 3-6-9 NM from the Runway Threshold\n"
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(18, 16), tight_layout=False)
fig.suptitle(title, weight='bold', fontsize=14)
fig.subplots_adjust(hspace=0.01, wspace=0.4)

subtitle = 'Lateral Dispersion at 3 NM from the RT'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['lateral_dispersion_3nm']],
syn_lateral_dispersion_3nm, bins=20)
axes[0, 0].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[0, 0].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[0, 0].hist(hist_data[:, hist_columns['lateral_dispersion_3nm']], bins=20,  color='gray',
alpha=1.0, label='Historical', density=True)
axes[0, 0].hist(syn_lateral_dispersion_3nm, bins=20, alpha=0.7, label='Synthetic', density=True)
axes[0, 0].set_title(subtitle)
axes[0, 0].set_xlabel('Distance, nautical miles')
axes[0, 0].set_ylabel('Probability Density')
ax.legend(loc='lower right')

subtitle = 'Vertical Dispersion at 3 NM from the RT'
```

```
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['vertical_dispersion_3nm']],
syn_vertical_dispersion_3nm, bins=25)
axes[0, 1].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[0, 1].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[0, 1].hist(hist_data[:, hist_columns['vertical_dispersion_3nm']], bins=25,  color='gray',
alpha=1.0, label='Historical', density=True)
axes[0, 1].hist(syn_vertical_dispersion_3nm, bins=25, alpha=0.7, color='green', label='Synthetic',
density=True)
axes[0, 1].set_title(subtitle)
axes[0, 1].set_xlabel('Baroaltitude, feet')
axes[0, 1].set_ylabel('Probability Density')
axes[0, 1].legend(loc='lower right')

subtitle = 'Speed Dispersion at 3 NM from the RT'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['speed_dispersion_3nm']],
syn_speed_dispersion_3nm, bins=30)
axes[0, 2].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[0, 2].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[0, 2].hist(hist_data[:, hist_columns['speed_dispersion_3nm']], bins=30,  color='gray',
alpha=1.0, label='Historical', density=True)
axes[0, 2].hist(syn_speed_dispersion_3nm, bins=30, alpha=0.7, color='purple', label='Synthetic',
density=True)
axes[0, 2].set_title(subtitle)
axes[0, 2].set_xlabel('Ground Speed, knots')
axes[0, 2].set_ylabel('Probability Density')
axes[0, 2].legend(loc='lower right')

subtitle = 'Lateral Dispersion at 6 NM from the RT'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['lateral_dispersion_6nm']],
syn_lateral_dispersion_6nm, bins=20)
axes[1, 0].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[1, 0].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[1, 0].hist(hist_data[:, hist_columns['lateral_dispersion_6nm']], bins=20, color='gray',
alpha=1.0, label='Historical', density=True)
axes[1, 0].hist(syn_lateral_dispersion_6nm, bins=20, alpha=0.7, label='Synthetic', density=True)
axes[1, 0].set_title(subtitle)
axes[1, 0].set_xlabel('Distance, nautical miles')
axes[1, 0].set_ylabel('Probability Density')
axes[1, 0].legend(loc='lower right')

subtitle = 'Vertical Dispersion at 6 NM from the RT'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['vertical_dispersion_6nm']],
syn_vertical_dispersion_6nm, bins=25)
```

```
axes[1, 1].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[1, 1].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[1, 1].hist(hist_data[:, hist_columns['vertical_dispersion_6nm']], bins=25,  color='gray',
alpha=1.0, label='Historical', density=True)
axes[1, 1].hist(syn_vertical_dispersion_6nm, bins=25, alpha=0.7, color='green', label='Synthetic',
density=True)
axes[1, 1].set_title(subtitle)
axes[1, 1].set_xlabel('Baroaltitude, feet')
axes[1, 1].set_ylabel('Probability Density')
axes[1, 1].legend(loc='lower right')

subtitle = 'Speed Dispersion at 6 NM from the RT'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['speed_dispersion_6nm']],
syn_speed_dispersion_6nm, bins=30)
axes[1, 2].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[1, 2].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[1, 2].hist(hist_data[:, hist_columns['speed_dispersion_6nm']], bins=30, color='gray',
alpha=1.0, label='Historical', density=True)
axes[1, 2].hist(syn_speed_dispersion_6nm, bins=30, alpha=0.7, color='purple', label='Synthetic',
density=True)
axes[1, 2].set_title(subtitle)
axes[1, 2].set_xlabel('Ground Speed, knots')
axes[1, 2].set_ylabel('Probability Density')
axes[1, 2].legend(loc='lower right')

subtitle = 'Lateral Dispersion at 9 NM from the RT'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['lateral_dispersion_9nm']],
syn_lateral_dispersion_9nm, bins=20)
axes[2, 0].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[2, 0].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[2, 0].hist(hist_data[:, hist_columns['lateral_dispersion_9nm']], bins=20,  color='gray',
alpha=1.0, label='Historical', density=True)
axes[2, 0].hist(syn_lateral_dispersion_9nm, bins=20, alpha=0.7, label='Synthetic', density=True)
axes[2, 0].set_title(subtitle)
axes[2, 0].set_xlabel('Distance, nautical miles')
axes[2, 0].set_ylabel('Probability Density')
axes[2, 0].legend(loc='lower right')

subtitle = 'Vertical Dispersion at 9 NM from the RT'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['vertical_dispersion_9nm']],
syn_vertical_dispersion_9nm, bins=20)
axes[2, 1].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
```

```
    transform=axes[2, 1].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[2, 1].hist(hist_data[:, hist_columns['vertical_dispersion_9nm']], bins=25,  color='gray',
alpha=1.0, label='Historical', density=True)
axes[2, 1].hist(syn_vertical_dispersion_9nm, bins=25, alpha=0.7, color='green', label='Synthetic',
density=True)
axes[2, 1].set_title(subtitle)
axes[2, 1].set_xlabel('Baroaltitude, feet')
axes[2, 1].set_ylabel('Probability Density')
axes[2, 1].legend(loc='lower right')

subtitle = 'Speed Dispersion at 9 NM from the RT'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['speed_dispersion_9nm']],
syn_speed_dispersion_9nm, bins=30)
axes[2, 2].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[2, 2].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[2, 2].hist(hist_data[:, hist_columns['speed_dispersion_9nm']], bins=30, color='gray',
alpha=1.0, label='Historical', density=True)
axes[2, 2].hist(syn_speed_dispersion_9nm, bins=30, alpha=0.7, color='purple', label='Synthetic',
density=True)
axes[2, 2].set_title(subtitle)
axes[2, 2].set_xlabel('Ground Speed, knots')
axes[2, 2].set_ylabel('Probability Density')
axes[2, 2].legend(loc='lower right')

plt.show()

# %%
def calculate_heading(lat1, lon1, lat2, lon2):
    # Convert to radians
    lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])

    dlon = lon2 - lon1
    x = np.sin(dlon) * np.cos(lat2)
    y = np.cos(lat1) * np.sin(lat2) - np.sin(lat1) * np.cos(lat2) * np.cos(dlon)

    heading = np.degrees(np.arctan2(x, y))
    return (heading + 360) % 360  # Normalize to 0-360

# %%
for track in tracks:

    track[0, columns['heading']] = 0
    track[0, columns['course_change']] = False
    track[0, columns['deceleration']] = False
    track[0, columns['descend']] = False
    track[0, columns['speed_change']] = False
    track[0, columns['level_change']] = False
    track[0, columns['magnitude_course_change']] = 0
    track[0, columns['magnitude_deceleration']] = 0
    track[0, columns['magnitude_descend']] = 0
    track[0, columns['cumulative_course_change']] = 0
    track[0, columns['cumulative_deceleration']] = 0
    track[0, columns['cumulative_descend']] = 0
```

```
    for i in range(1, len(track)):

        lat1, lon1 = track[i - 1, columns['lat']], track[i - 1, columns['lon']]
        lat2, lon2 = track[i, columns['lat']], track[i, columns['lon']]

        track[i, columns['heading']] = calculate_heading(lat1, lon1, lat2, lon2)

        # Course Changes:
        if i > 3:
            if track[i - 1, columns['heading']] - track[i, columns['heading']] > 5:
                track[i, columns['course_change']] = True
                track[i, columns['magnitude_course_change']] = track[i - 1, columns['heading']] -
track[i, columns['heading']]

                if track[i - 1, columns['magnitude_course_change']] != 0:
                    track[i, columns['cumulative_course_change']] = track[i - 1,
columns['cumulative_course_change']] + track[i, columns['magnitude_course_change']]
                elif i > 2 and track[i - 2, columns['magnitude_course_change']] != 0 and track[i -
1, columns['magnitude_course_change']] == 0:
                    track[i - 1, columns['cumulative_course_change']] = track[i - 2,
columns['cumulative_course_change']]
                    track[i, columns['cumulative_course_change']] = track[i - 2,
columns['cumulative_course_change']] + track[i, columns['magnitude_course_change']]
                else:
                    track[i, columns['cumulative_course_change']] = track[i,
columns['magnitude_course_change']]
            else:
                track[i, columns['course_change']] = False
                track[i, columns['magnitude_course_change']] = 0
                track[i, columns['cumulative_course_change']] = 0
        else:
            track[i, columns['course_change']] = False
            track[i, columns['magnitude_course_change']] = 0
            track[i, columns['cumulative_course_change']] = 0

        # Constant Speed or Constant Deceleration Segment:
        if track[i - 1, columns['speed']] - track[i, columns['speed']] > 2:

            if track[i - 1, columns['deceleration']] == False:
                track[i, columns['speed_change']] = True
            else:
                track[i, columns['speed_change']] = False

            track[i, columns['deceleration']] = True
            track[i, columns['magnitude_deceleration']] = track[i - 1, columns['speed']] -
track[i, columns['speed']]

            if track[i - 1, columns['magnitude_deceleration']] != 0:
                track[i, columns['cumulative_deceleration']] = track[i - 1,
columns['cumulative_deceleration']] + track[i, columns['magnitude_deceleration']]
            elif i > 2 and track[i - 2, columns['magnitude_deceleration']] != 0 and track[i - 1,
columns['magnitude_deceleration']] == 0:
                track[i, columns['cumulative_deceleration']] = track[i - 2,
columns['cumulative_deceleration']] + track[i, columns['magnitude_deceleration']]
                track[i - 1, columns['cumulative_deceleration']] = track[i - 2,
columns['cumulative_deceleration']]
            else:
                track[i, columns['cumulative_deceleration']] = track[i,
columns['magnitude_deceleration']]
        else:
            track[i, columns['deceleration']] = False
            track[i, columns['speed_change']] = False
            track[i, columns['magnitude_deceleration']] = 0
            track[i, columns['cumulative_deceleration']] = 0
```

```
        # Level Flight or Descending Segment:
        if track[i - 1, columns['alt']] - track[i, columns['alt']] > 25:

            if track[i - 1, columns['descend']] == False:
                track[i, columns['level_change']] = True
            else:
                track[i, columns['level_change']] = False

            track[i, columns['descend']] = True
            track[i, columns['magnitude_descend']] = track[i - 1, columns['alt']] - track[i,
columns['alt']]

            if track[i - 1, columns['magnitude_descend']] != 0:
                track[i, columns['cumulative_descend']] = track[i - 1,
columns['cumulative_descend']] + track[i, columns['magnitude_descend']]
            elif i > 2 and track[i - 2, columns['magnitude_descend']] != 0 and track[i - 1,
columns['magnitude_descend']] == 0:
                track[i, columns['cumulative_descend']] = track[i - 2,
columns['cumulative_descend']] + track[i, columns['magnitude_descend']]
                track[i - 1, columns['cumulative_descend']] = track[i - 2,
columns['cumulative_descend']]
            else:
                track[i, columns['cumulative_descend']] = track[i, columns['magnitude_descend']]
        else:
            track[i, columns['descend']] = False
            track[i, columns['level_change']] = False
            track[i, columns['magnitude_descend']] = 0
            track[i, columns['cumulative_descend']] = 0

# %%
course_change_count_data = np.zeros(len(tracks))
course_change_magnitude_list = []

for idx, track in enumerate(tracks):

    course_change_count = 0
    course_change_magnitude = 0

    for i in range(1, len(track)):
        if track[i, columns['course_change']] == False and track[i - 1, columns['course_change']]
== True:
            course_change_count += 1
            course_change_magnitude = track[i - 1, columns['cumulative_course_change']]
            course_change_magnitude_list.append(course_change_magnitude)

    course_change_count_data[idx] = course_change_count

course_change_magnitude_data = np.array(course_change_magnitude_list)

# %%
speed_change_count_data = np.zeros(len(tracks))
level_change_count_data = np.zeros(len(tracks))

for idx, track in enumerate(tracks):
    speed_change_count = 0
    level_change_count = 0

    for i in range(1, len(track)):
        if track[i, columns['speed_change']] == True:
            speed_change_count += 1

        if track[i, columns['level_change']] == True:
            level_change_count += 1

    speed_change_count_data[idx] = speed_change_count
```

```
    level_change_count_data[idx] = level_change_count

# %%
title = "Number of Course, Speed, and Level Change Segments per Track\n"
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 6), tight_layout=False)
fig.suptitle(title, weight='bold', fontsize=14)
fig.subplots_adjust(hspace=0.01, wspace=0.4)

subtitle = 'Course Changes'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['course_change_count']],
course_change_count_data, bins=10)
axes[0].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[0].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[0].hist(hist_data[:, hist_columns['course_change_count']], bins=10,  color='gray', alpha=1.0,
label='Historical', density=True)
axes[0].hist(course_change_count_data, bins=10, alpha=0.7, label='Synthetic', density=True)
axes[0].set_title(subtitle)
axes[0].set_xlabel('Count')
axes[0].set_ylabel('Probability Density')
ax.legend(loc='lower right')

subtitle = 'Level Changes'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['level_change_count_data']],
level_change_count_data, bins=20)
axes[1].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[1].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[1].hist(hist_data[:, hist_columns['level_change_count_data']], bins=20,  color='gray',
alpha=1.0, label='Historical', density=True)
axes[1].hist(level_change_count_data, bins=20, alpha=0.7, label='Synthetic', color='green',
density=True)
axes[1].set_title(subtitle)
axes[1].set_xlabel('Count')
axes[1].set_ylabel('Probability Density')
ax.legend(loc='lower right')

subtitle = 'Speed Changes'
print(subtitle)
metrics = compute_distribution_metrics(hist_data[:, hist_columns['speed_change_count_data']],
speed_change_count_data, bins=25)
axes[2].text(0.75, 0.97,
    f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
    transform=axes[2].transAxes,
    fontsize=10,
    color='darkgreen',
    ha='left', va='top',
    bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))
axes[2].hist(hist_data[:, hist_columns['speed_change_count_data']], bins=25,  color='gray',
alpha=1.0, label='Historical', density=True)
axes[2].hist(speed_change_count_data, bins=25, alpha=0.7, label='Synthetic', color='purple',
density=True)
```

```
axes[2].set_title(subtitle)
axes[2].set_xlabel('Count')
axes[2].set_ylabel('Probability Density')
axes[2].legend(loc='upper left')

# %%
title = "Course Change Segments"
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize = 14)

axes[0].hist(course_change_count_data, bins=10, alpha=0.7, label='Synthetic')
axes[0].set_title('Number of Course Changes per Track')
axes[0].set_xlabel('Count')
axes[0].set_ylabel('Frequency')
axes[0].legend(loc='upper right')

axes[1].hist(course_change_magnitude_data, bins=25, alpha=0.7, label='Synthetic')
axes[1].set_title('Magnitude of Course Change Segments')
axes[1].set_xlabel('Degrees')
axes[1].set_ylabel('Frequency')
axes[1].legend(loc='upper right')

# %%
# Define the output directory where the file is stored
output_dir = "C:\\Files\\"

# Build the full file path by joining the directory and file name
file_path = os.path.join(output_dir, "Historical_Data_Waypoints.csv") # File name

# Load the combined results file
df = pd.read_csv(file_path)

# Convert entire DataFrame to NumPy array
hist_waypoints = df.to_numpy()

# Create dictionary: {index: header_name}
headers_waypoints = {i: col for i, col in enumerate(df.columns)}
hist_waypoints_columns = {v: k for k, v in headers_waypoints.items()}

# %%
waypoints = {
    # SKILS FIVE ARRIVAL (RNAV)
    'BAL': {'lat':39.16673889, 'lon':-76.65018889, 'alt':10000, 'track':234, 'distance':16,
'speed':250},
    'EYESS': {'lat':38.96675278, 'lon':-76.88355833, 'alt':9000, 'track':284, 'distance':6},
    'EDDWD': {'lat':38.96682778, 'lon':-77.01684444, 'alt':8000, 'track':328, 'distance':5},
    'MEGGS': {'lat':39.03336944, 'lon':-77.08359167, 'alt':7000, 'track':327, 'distance':8},
    'NAYES': {'lat':39.13333889, 'lon':-77.21673889, 'alt':6000, 'track':327},
    # RNAV (GPS) Y RWY 19
    'DARIC': {'lat':38.95025, 'lon':-77.15005278, 'alt':2700, 'track':150, 'distance':2.5,
'speed':180}, # Initial Fix
    'SUNEY': {'lat':38.93334167, 'lon':-77.11669722, 'alt':2000, 'track':158, 'distance':1.7},
    'RORRK': {'lat':38.90016389, 'lon':-77.08359167, 'alt':1500, 'track':122, 'distance':1.6},
    'SLAKR': {'lat':38.90000833, 'lon':-77.06668333, 'alt':1020, 'track':149, 'distance':0.7}, #
Final Approach Fix
    'DALLA': {'lat':38.88347222, 'lon':-77.050125, 'alt':780, 'track':149, 'distance':1.2},
    'UDERE': {'lat':38.86683056, 'lon':-77.03345, 'alt':373, 'track':149, 'distance':0.9},
    'RWY19': {'lat':38.861193, 'lon':-77.038730, 'alt':0, 'track':149, 'distance':0}
}

# %%
# %matplotlib widget

# %%
title = f"Synthetic Flight Tracks with Waypoints"
```

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6,6), tight_layout=True)
fig.subplots_adjust(hspace=0.2, wspace=0.2)
fig.suptitle(title, weight='bold', fontsize = 12)

# Plot tracks
for track in tracks:
    ax.plot(track[0:, columns['lon']], track[0:, columns['lat']])

# Plot waypoints
for label, data in waypoints.items():
    ax.plot(data['lon'], data['lat'], 'o',
            markeredgewidth=1,
            markeredgecolor='0',
            markersize=10,
            label=label)

ax.set_title('Lateral Position')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.legend()

plt.show()

# %%
flight_id_waypoints = []
waypoint_columns = {'waypoint': 0, 'index': 1, 'lat': 2, 'lon': 3, 'distance': 4, 'altitude' : 5,
'speed': 6}

# Build waypoint data for each flight
for track in tracks:
    track_waypoints = []
    for waypoint, waypoint_data in waypoints.items():
        target_lat = waypoint_data['lat']
        target_lon = waypoint_data['lon']

        lat_list = track[:, columns['lat']].tolist()
        lon_list = track[:, columns['lon']].tolist()

        # Calculate great-circle distances for all points in the track
        distances = [
            get_great_circle_distance(lon_list[k], lat_list[k], target_lon, target_lat)
            for k in range(len(track))
        ]

        # Find closest point
        closest_index = int(np.argmin(distances))
        closest_lat = lat_list[closest_index]
        closest_lon = lon_list[closest_index]
        closest_distance = distances[closest_index]
        closest_altitude = track[closest_index, columns['alt']]
        closest_speed = track[closest_index, columns['speed']]

        # Store waypoint info
        array = np.array([waypoint, closest_index, closest_lat, closest_lon, closest_distance,
closest_altitude, closest_speed])
        track_waypoints.append(array)

    flight_id = track[0, columns['syn_track_index']]
    flight_id_waypoints.append([flight_id, track_waypoints])

# %%
# Extract waypoint names from CSV columns
waypoint_names = [col.replace('_distance', '') for col in df.columns if '_distance' in col]

title = "Lateral Dispersion at the Waypoints\n"
```

```
fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(18, 20), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize=14)

axes = axes.flatten()

for i, waypoint in enumerate(waypoints.keys()):
    ax = axes[i]

    # Synthetic data from flight_id_waypoints
    lateral_dispersion_waypoints = []
    for flight_data in flight_id_waypoints:
        for wp_data in flight_data[1]:
            if wp_data[waypoint_columns['waypoint']] == waypoint:
                lateral_dispersion_waypoints.append(float(wp_data[waypoint_columns['distance']]))

    # Data from CSV (Historical)
    if waypoint in waypoint_names:
        hist_distances = df[f'{waypoint}_distance'].dropna()
    else:
        hist_distances = []

    # Plot both histograms
    ax.hist(hist_distances, bins=25, alpha=1.0, color='gray', label='Historical', density=True)
    ax.hist(lateral_dispersion_waypoints, bins=25, alpha=0.6, label='Syntehtic', density=True)

    subtitle = f'{waypoint}'
    print(subtitle)
    metrics = compute_distribution_metrics(hist_distances, lateral_dispersion_waypoints, bins=25)
    ax.text(0.7, 0.97,
        f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
        transform=ax.transAxes,
        fontsize=10,
        color='darkgreen',
        ha='left', va='top',
        bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

    ax.set_title(subtitle)
    ax.set_xlabel('Distance, nautical miles')
    ax.set_ylabel('Probability Density')
    ax.legend(loc='lower right')

plt.show()

# %%
title = "Vertical Dispersion at the Waypoints\n"
fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(18, 20), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize=14)

axes = axes.flatten()

for i, waypoint in enumerate(waypoints.keys()):
    ax = axes[i]

    # Synthetic data from flight_id_waypoints
    vertical_dispersion_waypoints = []
    for flight_data in flight_id_waypoints:
        for wp_data in flight_data[1]:
            if wp_data[waypoint_columns['waypoint']] == waypoint:
                vertical_dispersion_waypoints.append(float(wp_data[waypoint_columns['altitude']]))

    # Data from CSV (Historical)
    if waypoint in waypoint_names:
        hist_altitudes = df[f'{waypoint}_altitude'].dropna()
    else:
```

```
        hist_altitudes = []

    # Plot both histograms
    ax.hist(hist_altitudes, bins=15, alpha=1.0, color='gray', label='Historical', density=True)
    ax.hist(vertical_dispersion_waypoints, bins=15, alpha=0.6, color='green', label='Synthetic',
density=True)

    # If synthetic data is constant, add a vertical line for clarity
    if len(set(vertical_dispersion_waypoints)) == 1:
        ax.axvline(vertical_dispersion_waypoints[0], color='green', alpha=0.6, linewidth=19)
        ax.set_ylim(0, 0.02)

    subtitle = f'{waypoint}'
    print(subtitle)
    metrics = compute_distribution_metrics(hist_altitudes, vertical_dispersion_waypoints, bins=15)
    ax.text(0.03, 0.97,
        f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
        transform=ax.transAxes,
        fontsize=10,
        color='darkgreen',
        ha='left', va='top',
        bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

    ax.set_title(subtitle)
    ax.set_xlabel('Baroaltitude, feet')
    ax.set_ylabel('Probability Density')
    ax.legend(loc='lower left')

plt.show()

# %%
title = "Speed Dispersion at the Waypoints\n"
fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(18, 20), tight_layout=True)
fig.suptitle(title, weight='bold', fontsize=14)

axes = axes.flatten()

for i, waypoint in enumerate(waypoints.keys()):
    ax = axes[i]

    # Synthetic data from flight_id_waypoints
    speed_dispersion_waypoints = []
    for flight_data in flight_id_waypoints:
        for wp_data in flight_data[1]:
            if wp_data[waypoint_columns['waypoint']] == waypoint:
                speed_dispersion_waypoints.append(float(wp_data[waypoint_columns['speed']]))

    # Data from CSV (Historical)
    if waypoint in waypoint_names:
        hist_speeds = df[f'{waypoint}_speed'].dropna()
    else:
        hist_speeds = []

    # Plot both histograms
    ax.hist(hist_speeds, bins=25, alpha=1.0, color='gray', label='Historical', density=True)
    ax.hist(speed_dispersion_waypoints, bins=25, alpha=0.6, color='purple', label='Synthetic',
density=True)

    # If synthetic data is constant, add a vertical line for clarity
    if len(set(speed_dispersion_waypoints)) == 1:
        ax.axvline(speed_dispersion_waypoints[0], color='purple', alpha=0.6, linewidth=12)
        ax.set_ylim(0, 0.1)

    subtitle = f'{waypoint}'
```

```
    print(subtitle)
    metrics = compute_distribution_metrics(hist_speeds, speed_dispersion_waypoints, bins=25)
    ax.text(0.03, 0.77,
        f"WD = {metrics['WD']:.4f}\nKS = {metrics['KSS']:.4f}\np = {metrics['KSP']:.4e}\nJSD =
{metrics['JSD']:.4f}",
        transform=ax.transAxes,
        fontsize=10,
        color='darkgreen',
        ha='left', va='top',
        bbox=dict(facecolor='white', alpha=0.6, edgecolor='none'))

    ax.set_xlabel('Ground Speed, knots')
    ax.set_ylabel('Probability Density')
    ax.legend(loc='upper left')

plt.show()
```

REFERENCES

[1] R. Weibel and R. J. Hansman, "Safety Considerations for Operation of Different Classes of UAVs in the NAS," in *AIAA 4th Aviation Technology, Integration and Operations (ATIO) Forum*, Chicago, Illinois: American Institute of Aeronautics and Astronautics, Sep. 2004. doi: 10.2514/6.2004-6244.

[2] "National Airspace System | Federal Aviation Administration." [Online]. Available: https://www.faa.gov/air_traffic/nas

[3] Federal Aviation Administration, Airman Testing Standards Branch, "Risk Management Handbook (FAA-H-8083-2A)," 2022, [Online]. Available: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.faa.gov/sites/faa.gov/files/2022-06/risk_management_handbook_2A.pdf

[4] INCOSE, *INCOSE Systems Engineering Handbook*. John Wiley & Sons, 2023.

[5] Federal Aviation Administration, "Building your Operational Risk Assessment," presented at the FAA UAS Symposium, [Online]. Available: https://www.faa.gov/sites/faa.gov/files/uas/resources/events_calendar/archive/Building_Your_Operational_Risk_Assessment.pdf

[6] X. Lin, N. Fulton, and M. Westcott, "Target level of safety measures in air transportation-Review, validation and recommendations," in *Proceedings of the IASTED International Conference*, 2009, p. 222. [Online]. Available: https://www.researchgate.net/profile/Xun-Lin-2/publication/265251293_Target_Level_of_Safety_Measures_in_Air_Transportation_-_Review_Validation_and_Recommendations/links/540675710cf23d9765a7f69d/Target-Level-of-Safety-Measures-in-Air-Transportation-Review-Validation-and-Recommendations.pdf

[7] B. Figuet, R. Monstein, M. Waltert, and J. Morio, "Data-driven mid-air collision risk modelling using extreme-value theory," *Aerosp. Sci. Technol.*, vol. 142, p. 108646, Nov. 2023, doi: 10.1016/j.ast.2023.108646.

[8] Andy Quinn, "New Safety Model for the Commercial Human Spaceflight Industry," ResearchGate. [Online]. Available: https://www.researchgate.net/publication/258643697_New_Safety_Model_for_the_Commercial_Human_Spaceflight_Industry

[9] *Manual on implementation of a 300 m (1 000 ft) vertical separation minimum between FL 290 and FL 410 inclusive*, 3. ed. in Doc / International Civil Aviation Organization [Englische Ausgabe], no. 9574. Montreal: Internat. Civil Aviation Organization, 2012.

[10] R. L. Newman and D. C. Foyle, "Test scenarios for rare events," in *Proceedings of the 12th International Symposium on Aviation Psychology*, 2003, pp. 873–882. [Online]. Available: https://humanfactors.arc.nasa.gov/groups/hcsl/publications/Newman_AvPsyc03.pdf

[11] A. N. Kane, A. S. Jadhav, S. Sahamate, V. Kokare, and K. B. Jadhav, “Review of Aircraft Crash Reports and Engine Pod Location,” in *2021 IEEE Pune Section International Conference (PuneCon)*, Dec. 2021, pp. 1–9. doi: 10.1109/PuneCon52575.2021.9686505.
[12] J. T. Luxhoj and D. W. Coit, “Modeling low probability/high consequence events: an aviation safety risk model,” in *RAMS’06. Annual Reliability and Maintainability Symposium, 2006.*, IEEE, 2006, pp. 215–221. Accessed: Mar. 14, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1677377/
[13] International Civil Aviation Organization (ICAO), “Manual on the Use of the Collision Risk Model (CRM) for ILS Operations.” ICAO, 1980. [Online]. Available: https://www.icao.int/isbn/Lists/Publications/DispForm.aspx?ID=113
[14] “Accident Data.” [Online]. Available: https://www.ntsb.gov/safety/data/Pages/Data_Stats.aspx
[15] Federal Aviation Administration, “Order 8000.369 Safety Management System,” Order 8040.4C, Sep. 2023. [Online]. Available: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.faa.gov/documentLibrary/media/Order/FAA_Order_8040.4C.pdf
[16] Airport Cooperative Research Program, Transportation Research Board, and National Academies of Sciences, Engineering, and Medicine, *Risk Assessment Method to Support Modification of Airfield Separation Standards*. Washington, D.C.: Transportation Research Board, 2011, p. 14501. doi: 10.17226/14501.
[17] “Resources - Library Contents - FAA - FAASTeam - FAASafety.gov.” [Online]. Available: https://www.faasafety.gov/gslac/alc/libview_normal.aspx?id=6851
[18] R. E. Machol, “Thirty Years of Modeling Midair Collisions,” *Interfaces*, vol. 25, no. 5, pp. 151–172, Oct. 1995, doi: 10.1287/inte.25.5.151.
[19] A. K. Thapa, J. Shortle, and L. Sherry, “Air-to-Air Collision Risk Models (CRM) in the Terminal Airspace,” in *2023 Integrated Communication, Navigation and Surveillance Conference (ICNS)*, Herndon, VA, USA: IEEE, Apr. 2023, pp. 1–9. doi: 10.1109/ICNS58246.2023.10124323.
[20] Federal Aviation Administration, “Safety Study of Closely Spaced Parallel Operations Utilizing Paired Approach,” Flight Technologies and Procedures Division, AFS-400, Washington, DC 20591, Technical Report DOT/FAA/AFS400/2019/R/25, Apr. 2019. [Online]. Available: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.faa.gov/sites/faa.gov/files/about/office_org/headquarters_offices/avs/DOT-FAA-AFS400-2019-R-25.pdf
[21] Q. Cai, H. Jie ANG, and S. Alam, “Collision risk assessment of reduced aircraft separation minima in procedural airspace using advanced communication and navigation,” *Chin. J. Aeronaut.*, vol. 36, no. 4, pp. 315–337, Apr. 2023, doi: 10.1016/j.cja.2022.11.016.
[22] Y. Zhang, J. Shortle, and L. Sherry, “Methodology for collision risk assessment of flow corridor concept,” in *2013 Integrated Communications, Navigation and Surveillance Conference (ICNS)*, Apr. 2013, pp. 1–8. doi: 10.1109/ICNSurv.2013.6548552.
[23] M. Fujita, “Frequency of rare event occurrences,” in *ENRI international workshop for ATM/CNS, Tokyo, Japan*, 2009. Accessed: Mar. 14, 2026. [Online]. Available: https://www.enri.go.jp/jp/event/int_workshop/doc/eiwac2009/FrequencyofRareEventOccurrences(Proceeding).pdf

[24] P. T. W. Hudson and G. C. Van Der Graaf, “The Rule of Three: Situation awareness in hazardous situations,” in *SPE International Conference and Exhibition on Health, Safety, Environment, and Sustainability?*, SPE, 1998, p. SPE-46765. [Online]. Available: https://onepetro.org/SPEHSE/proceedings-abstract/98HSE/All-98HSE/SPE-46765-MS/190069
[25] “DFW International Airport | Official Website.” Accessed: Mar. 16, 2025. [Online]. Available: https://www.dfwairport.com/business/about/stats/
[26] D. Ziakkas and K. Pechlivanis, “Artificial intelligence applications in aviation accident classification: A preliminary exploratory study,” *Decis. Anal. J.*, vol. 9, p. 100358, Dec. 2023, doi: 10.1016/j.dajour.2023.100358.
[27] M. Soori, B. Arezoo, and R. Dastres, “Artificial intelligence, machine learning and deep learning in advanced robotics, a review,” *Cogn. Robot.*, vol. 3, pp. 54–70, 2023, doi: 10.1016/j.cogr.2023.04.001.
[28] J. Sprockhoff, S. Gupta, U. Durak, and T. Krueger, “Scenario-Based Synthetic Data Generation for an AI-based System Using a Flight Simulator,” in *AIAA SCITECH 2024 Forum*, American Institute of Aeronautics and Astronautics. doi: 10.2514/6.2024-1462.
[29] A. Nanyonga, H. Wasswa, K. Joiner, U. Turhan, and G. Wild, “Explainable Supervised Learning Models for Aviation Predictions in Australia,” *Aerospace*, vol. 12, no. 3, p. 223, Mar. 2025, doi: 10.3390/aerospace12030223.
[30] C. Torens, U. Durak, and J. C. Dauer, “Guidelines and Regulatory Framework for Machine Learning in Aviation,” in *AIAA SCITECH 2022 Forum*, San Diego, CA & Virtual: American Institute of Aeronautics and Astronautics, Jan. 2022. doi: 10.2514/6.2022-1132.
[31] H. Bello *et al.*, “Towards certifiable AI in aviation: landscape, challenges, and opportunities,” Sep. 13, 2024, *arXiv*: arXiv:2409.08666. doi: 10.48550/arXiv.2409.08666.
[32] L. Sherry *et al.*, “Review of Current State of Artificial Intelligence/Machine Learning and Other Advanced Techniques Related to Air-to-Air Collision Risk Models (CRM) in the Terminal Airspace,” 2023, [Online]. Available: https://rosap.ntl.bts.gov/view/dot/67526
[33] E. Solak, R. Murray-Smith, W. E. Leithead, D. Leith, and C. Rasmussen, “Derivative observations in Gaussian process models of dynamic systems,” *Adv. Neural Inf. Process. Syst.*, vol. 15, 2002, [Online]. Available: https://proceedings.neurips.cc/paper/2002/hash/5b8e4fd39d9786228649a8a8bec4e008-Abstract.html
[34] E. Uğurel, S. Huang, and C. Chen, “Learning to generate synthetic human mobility data: A physics-regularized Gaussian process approach based on multiple kernel learning,” *Transp. Res. Part B Methodol.*, vol. 189, p. 103064, Nov. 2024, doi: 10.1016/j.trb.2024.103064.
[35] J. Chen, W. Mu, Y. Li, and D. Li, “On the identifiability and interpretability of Gaussian process models,” *Adv. Neural Inf. Process. Syst.*, vol. 36, pp. 70267–70278, 2023.
[36] International Civil Aviation Organization, Ed., *Performance-based navigation (PBN) manual*, 3rd. ed. in Doc / Englische Ausgabe, no. 9613- AN/937,3. Montreal: ICAO, 2008.
[37] “FAA-H-8083-15B, Instrument Flying Handbook”.
[38] J. Zhou, S. Cafieri, D. Delahaye, and M. Sbihi, “Optimization of arrival and departure routes in terminal maneuvering area,” in *ICRAT 2014, 6th International Conference on Research in Air Transportation*, 2014, p. pp-xxxx. [Online]. Available: https://enac.hal.science/hal-01002807/

[39] S. Noh, “Methodology for Collision Risk Assessment of Aircraft with Diverse Collision Avoidance Capabilities,” 2020, [Online]. Available: https://hdl.handle.net/1920/12517
[40] P. G. Reich, “Analysis of Long-Range Air Traffic Systems: Separation Standards—I,” *J. Navig.*, vol. 19, no. 1, pp. 88–98, Jan. 1966, doi: 10.1017/S037346330004056X.
[41] M. Xue, V. Kuo, J. I. De Alvear Cardenas, and P. Pradeep, “A Method of Compliance for Achieving Target Collision Risk in UTM Operations,” in *AIAA AVIATION FORUM AND ASCEND 2024*, Las Vegas, Nevada: American Institute of Aeronautics and Astronautics, Jul. 2024. doi: 10.2514/6.2024-4457.
[42] P. G. Reich, “Separation standards—II,” *J. Navig.*, vol. 19, no. 2, pp. 169–186, 1966.
[43] P. G. Reich, “Analysis of long-range air traffic systems: separation standards—III,” *J. Navig.*, vol. 19, no. 3, pp. 331–347, 1966.
[44] J. F. Shortle, Y. Xie, C. H. Chen, and G. L. Donohue, “Simulating Collision Probabilities of Landing Airplanes at Nontowered Airports,” *SIMULATION*, vol. 80, no. 1, pp. 21–31, Jan. 2004, doi: 10.1177/0037549704042028.
[45] E. J. Garcia Gonzalez, “Development of a 3-Dimensional Mathematical Collision Risk Model Based on Recorded Aircraft Trajectories to Estimate the Safety Level in High Density En-Route Airspaces,” PhD Thesis, Universidad Politécnica de Madrid, 2013. doi: 10.20868/UPM.thesis.21884.
[46] R. Melnyk, D. Schrage, V. Volovoi, and H. Jimenez, “Sense and Avoid Requirements for Unmanned Aircraft Systems Using a Target Level of Safety Approach,” *Risk Anal.*, vol. 34, no. 10, pp. 1894–1906, Oct. 2014, doi: 10.1111/risa.12200.
[47] X. Olive and L. Basora, “Detection and identification of significant events in historical aircraft trajectory data,” *Transp. Res. Part C Emerg. Technol.*, vol. 119, p. 102737, Oct. 2020, doi: 10.1016/j.trc.2020.102737.
[48] S. Noh and J. F. Shortle, “Sensitivity analysis of event sequence diagrams for aircraft accident scenarios,” in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, IEEE, 2015, pp. 3E2-1. Accessed: Sep. 19, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7311408/?casa_token=a52QBlt2tPUAAAAA:eQSeC-oduKV0MsLhlpO8ee2lbw5e4ZvQTTVgCCXlrRX5DfCYQcaqci7O_KWCpFOcGQnfUN9u19M
[49] O. Snisarevska, L. Sherry, and J. Shortle, “Model-Based Simulation for Design of nn Airport Autonomous Approach and Landing System,” *INCOSE Int. Symp.*, vol. 28, no. 1, pp. 924–939, Jul. 2018, doi: 10.1002/j.2334-5837.2018.00525.x.
[50] L. Sherry, J. Shortle, and O. Snisarevska, “Design of a Control Law for an Autonomous Approach and Landing Spacing System,” in *2018 Aviation Technology, Integration, and Operations Conference*, Atlanta, Georgia: American Institute of Aeronautics and Astronautics, Jun. 2018. doi: 10.2514/6.2018-3983.
[51] O. Snisarevska, L. Sherry, J. Shortle, and G. Donohue, “Balancing throughput and safety: An autonomous approach and landing system (AALS),” in *2018 Integrated Communications, Navigation, Surveillance Conference (ICNS)*, IEEE, 2018, pp. 3B1-1. Accessed: Sep. 19, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8384854/?casa_token=mZNyWgLMxUYAAAAA:kM0kBAp3cPCn9WIKpPIsd0mYirFm9IP8ljV6Px8Z-XwdIHAHGC1HcW7uaOVPlewYo222szFVxnI

[52] H. Wang, P. Xu, and F. Zhong, “Modeling and Feature Analysis of Air Traffic Complexity Propagation,” *Sustainability*, vol. 14, no. 18, Sep. 2022, doi: 10.3390/su141811157.
[53] A. Bauranov and J. Rakas, “Bayesian network model of aviation safety: Impact of new communication technologies on mid-air collisions,” *Reliab. Eng. Syst. Saf.*, vol. 243, p. 109905, Mar. 2024, doi: 10.1016/j.ress.2023.109905.
[54] M. Daigle *et al.*, “Real-Time Prediction of Safety Margins in the National Airspace,” in *17th AIAA Aviation Technology, Integration, and Operations Conference*, in AIAA AVIATION Forum. , American Institute of Aeronautics and Astronautics, 2017. doi: 10.2514/6.2017-4388.
[55] X. Zhang and S. Mahadevan, “Bayesian neural networks for flight trajectory prediction and safety assessment,” *Decis. Support Syst.*, vol. 131, p. 113246, Apr. 2020, doi: 10.1016/j.dss.2020.113246.
[56] W. Glover and J. Lygeros, “A Stochastic Hybrid Model for Air Traffic Control Simulation,” in *Hybrid Systems: Computation and Control*, R. Alur and G. J. Pappas, Eds., Berlin, Heidelberg: Springer, 2004, pp. 372–386. doi: 10.1007/978-3-540-24743-2_25.
[57] E. Bonabeau, “Agent-based modeling: Methods and techniques for simulating human systems,” *Proc. Natl. Acad. Sci.*, vol. 99, no. suppl_3, pp. 7280–7287, May 2002, doi: 10.1073/pnas.082080899.
[58] “New Algorithms for Aircraft Intent Inference and Trajectory Prediction | Request PDF,” ResearchGate. [Online]. Available: https://www.researchgate.net/publication/245433456_New_Algorithms_for_Aircraft_Intent_Inference_and_Trajectory_Prediction
[59] “What is Data Augmentation? - Data Augmentation Techniques Explained - AWS,” Amazon Web Services, Inc. Accessed: Feb. 27, 2025. [Online]. Available: https://aws.amazon.com/what-is/data-augmentation/
[60] R. Ji, J. Shortle, and L. Sherry, “Synthetic Flight Track Generation using Wasserstein-based Generative Network Modeling”, [Online]. Available: https://catsr.vse.gmu.edu/pubs/Synthetic_Track_Generation_Wasserstein.pdf
[61] T. Krauth, J. Morio, X. Olive, B. Figuet, and R. Monstein, “Synthetic Aircraft Trajectories Generated with Multivariate Density Models,” *Eng. Proc.*, vol. 13, no. 1, p. 7, 2021, doi: 10.3390/engproc2021013007.
[62] T. Krauth, A. Lafage, J. Morio, X. Olive, and M. Waltert, “Deep generative modelling of aircraft trajectories in terminal maneuvering areas,” *Mach. Learn. Appl.*, vol. 11, p. 100446, Mar. 2023, doi: 10.1016/j.mlwa.2022.100446.
[63] T. Krauth, J. Morio, X. Olive, and B. Figuet, “Advanced collision risk estimation in terminal manoeuvring areas using a disentangled variational autoencoder for uncertainty quantification,” *Eng. Appl. Artif. Intell.*, vol. 133, p. 108137, 2024.
[64] S. Aref, J. Shortle, and L. Sherry, “Generating Synthetic Flight Tracks for Collision Risk Safety Analysis: Variational Autoencoders with a Single Seed Track,” *2024 Integr. Commun. Navig. Surveill. Conf. ICNS*, pp. 1–9, Apr. 2024, doi: 10.1109/ICNS60906.2024.10550543.
[65] S. Aref, J. Shortle, and L. Sherry, “Using Variational Autoencoders to Generate 4D Synthetic Flight Tracks for Collision Risk Safety Analysis,” in *2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC)*, IEEE, 2024, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10749485/

[66] S. Aref, J. Shortle, and L. Sherry, “A Methodology for Multi-Seed Generation of 4-D Synthetic Flight Tracks for Air-to-Air Collision Risk Modelling using Variational Autoencoder,” *Submitt. J. Artif. Inetlligence*, [Online]. Available: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://catsr.vse.gmu.edu/pubs/Multi_seed_paper__Engineering_applications_of_AI_.pdf
[67] S. Wijnands, A. Sharpanskykh, and K. Aly, “Generation of Synthetic Aircraft Landing Trajectories Using Generative Adversarial Networks”.
[68] Y. Yin, S. Zhang, Y. Zhang, Y. Zhang, and S. Xiang, “Context-aware Aircraft Trajectory Prediction with Diffusion Models,” in *2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC)*, Sep. 2023, pp. 5312–5317. doi: 10.1109/ITSC57777.2023.10422124.
[69] S. Yang, L. Liu, B. Chen, S. Cheng, Z. Shi, and Z. Zou, “GooDFlight: Goal-Oriented Diffusion Model for Flight Trajectory Prediction,” *IEEE Trans. Aerosp. Electron. Syst.*, 2025, [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10858371/?casa_token=Q2AvcrYS-ZgAAAAA:LsIV2zx8dU7BSrGSh3JFuNGRL2mK5cUde0ZWq47O_wPnwPlF6Ptt_VBMbS4aXKik8OqZlRplcxA
[70] J. Ma, P. Xiang, Q. Yao, Z. Jiang, J. Huang, and H. Li, “Optimizing Aircraft Route Planning Based on Data-Driven and Physics-Informed Wind Field Predictions,” *Mathematics*, vol. 13, no. 3, 2025, doi: 10.3390/math13030367.
[71] W. J. Eerland, S. Box, and A. Sóbester, “Modeling the Dispersion of Aircraft Trajectories Using Gaussian Processes,” *J. Guid. Control Dyn.*, vol. 39, no. 12, pp. 2661–2672, Dec. 2016, doi: 10.2514/1.G000537.
[72] J. Xu, L. Sherry, and J. Shortle, “Generating Synthetic Flight Tracks for Collision Risk Safety Analysis using Gaussian Process for Dynamic Systems”, [Online]. Available: https://catsr.vse.gmu.edu/pubs/GPR_SyntheticTrack1.pdf
[73] O. Donnelly, J. Xu, J. Shortle, and L. Sherry, “AI in the Air: Application of AI for Generating Flight Tracks for Collision Risk Analysis”.
[74] “Kingma: Auto-encoding variational bayes - Google Scholar.” [Online]. Available: https://scholar.google.com/scholar_lookup?title=Auto-encoding%20variational%20bayes&publication_year=2013&author=D.P.%20Kingma&author=M.%20Welling
[75] C. Doersch, “Tutorial on Variational Autoencoders,” Jan. 03, 2021, *arXiv*: arXiv:1606.05908. doi: 10.48550/arXiv.1606.05908.
[76] X. Hou, L. Shen, K. Sun, and G. Qiu, “Deep feature consistent variational autoencoder,” in *2017 IEEE winter conference on applications of computer vision (WACV)*, IEEE, 2017, pp. 1133–1141. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7926714/
[77] D. Bank, N. Koenigstein, and R. Giryes, “Autoencoders,” in *Machine Learning for Data Science Handbook*, L. Rokach, O. Maimon, and E. Shmueli, Eds., Cham: Springer International Publishing, 2023, pp. 353–374. doi: 10.1007/978-3-031-24628-9_16.
[78] S. Kullback, “Kullback-leibler divergence.” Tech. Rep, 1951.
[79] C. Song and R. Kawai, “Monte Carlo and variance reduction methods for structural reliability analysis: A comprehensive review,” *Probabilistic Eng. Mech.*, vol. 73, p. 103479, Jul. 2023, doi: 10.1016/j.probengmech.2023.103479.

[80] A. T. Cemgil, S. Ghaisas, K. Dvijotham, S. Gowal, and P. Kohli, “Autoencoding Variational Autoencoder,” Dec. 07, 2020, *arXiv*: arXiv:2012.03715. doi: 10.48550/arXiv.2012.03715.
[81] C. Luo, “Understanding diffusion models: A unified perspective,” *ArXiv Prepr. ArXiv220811970*, 2022, [Online]. Available: https://papers.baulab.info/papers/also/Luo-2022.pdf
[82] I. J. Goodfellow *et al.*, “Generative adversarial nets,” *Adv. Neural Inf. Process. Syst.*, vol. 27, 2014, [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2014/hash/f033ed80deb0234979a61f95710dbe25-Abstract.html
[83] L. Rüschendorf, “The Wasserstein distance and approximation theorems,” *Probab. Theory Relat. Fields*, vol. 70, no. 1, pp. 117–129, Mar. 1985, doi: 10.1007/BF00532240.
[84] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli, “Deep unsupervised learning using nonequilibrium thermodynamics,” in *International conference on machine learning*, pmlr, 2015, pp. 2256–2265. [Online]. Available: http://proceedings.mlr.press/v37/sohl-dickstein15.html
[85] J. Ho, A. Jain, and P. Abbeel, “Denoising diffusion probabilistic models,” *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 6840–6851, 2020.
[86] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations,” Nov. 28, 2017, *arXiv*: arXiv:1711.10561. doi: 10.48550/arXiv.1711.10561.
[87] Y. Wang *et al.*, “PredRNN: A Recurrent Neural Network for Spatiotemporal Predictive Learning,” Apr. 09, 2022, *arXiv*: arXiv:2103.09504. doi: 10.48550/arXiv.2103.09504.
[88] S. K. Sahu, “16 - Hierarchical Bayesian Models for Space–Time Air Pollution Data,” in *Handbook of Statistics*, vol. 30, T. Subba Rao, S. Subba Rao, and C. R. Rao, Eds., in Time Series Analysis: Methods and Applications, vol. 30. , Elsevier, 2012, pp. 477–495. doi: 10.1016/B978-0-444-53858-1.00016-8.
[89] M. Xu, W. Ding, J. Zhu, Z. Liu, B. Chen, and D. Zhao, “Task-agnostic online reinforcement learning with an infinite mixture of gaussian processes,” *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 6429–6440, 2020.
[90] E. Salaun, M. Gariel, A. E. Vela, and E. Feron, “Aircraft Proximity Maps Based on Data-Driven Flow Modeling,” *J. Guid. Control Dyn.*, vol. 35, no. 2, pp. 563–577, Mar. 2012, doi: 10.2514/1.53859.
[91] O. Snisarevska, L. Sherry, J. Xu, and J. Shortle, “Balancing Robustness and Fidelity in Synthetic Flight Tracks Generated via Gaussian Process Methods for Collision Risk Modeling,” in *2026 Integrated Communications, Navigation and Surveillance Conference (ICNS)*, IEEE, 2026, pp. 1–11. Accessed: Sep. 13, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11570267/?casa_token=1mFC-yLwbScAAAAA:TdncTZm-lvfcJYuZaiofkWrqFE5WaybIxwxMar24btvUoMHaiPXbVeWUXIKYoMx93x244w8-AIA
[92] A. S. Booth, A. Cooper, and R. B. Gramacy, “Nonstationary Gaussian Process Surrogates,” Dec. 02, 2024, *arXiv*: arXiv:2305.19242. doi: 10.48550/arXiv.2305.19242.
[93] W. Bruinsma, E. Perim, W. Tebbutt, S. Hosking, A. Solin, and R. Turner, “Scalable exact inference in multi-output Gaussian processes,” in *International Conference on Machine Learning*, PMLR, 2020, pp. 1190–1201. [Online]. Available: http://proceedings.mlr.press/v119/bruinsma20a.html

[94] C. E. Rasmussen and C. K. I. Williams, *Gaussian processes for machine learning*, 3. print. in Adaptive computation and machine learning. Cambridge, Mass.: MIT Press, 2008.
[95] S. Theodoridis, “Chapter 2 - Probability and Stochastic Processes,” in *Machine Learning (Second Edition)*, S. Theodoridis, Ed., Academic Press, 2020, pp. 19–65. doi: 10.1016/B978-0-12-818803-3.00011-8.
[96] T. J. Santner, B. J. Williams, and W. I. Notz, *The Design and Analysis of Computer Experiments*. in Springer Series in Statistics. New York, NY: Springer New York, 2018. doi: 10.1007/978-1-4939-8847-1.
[97] M. Heinonen, H. Mannerström, J. Rousu, S. Kaski, and H. Lähdesmäki, “Non-Stationary Gaussian Process Regression with Hamiltonian Monte Carlo,” in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, PMLR, May 2016, pp. 732–740. [Online]. Available: https://proceedings.mlr.press/v51/heinonen16.html
[98] D. Duvenaud, “Automatic model construction with Gaussian processes,” PhD Thesis, 2014. [Online]. Available: https://www.repository.cam.ac.uk/items/a54f0711-f777-4ee1-a685-3b76dae5ad91
[99] C. Paciorek and M. Schervish, “Nonstationary Covariance Functions for Gaussian Process Regression,” in *Advances in Neural Information Processing Systems*, MIT Press, 2003. [Online]. Available: https://papers.nips.cc/paper_files/paper/2003/hash/326a8c055c0d04f5b06544665d8bb3ea-Abstract.html
[100] D. Maclaurin, D. Duvenaud, and R. Adams, “Gradient-based hyperparameter optimization through reversible learning,” in *International conference on machine learning*, PMLR, 2015, pp. 2113–2122. [Online]. Available: http://proceedings.mlr.press/v37/maclaurin15.html
[101] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, no. 85, pp. 2825–2830, 2011.
[102] Y. Pan, X. Zeng, H. Xu, Y. Sun, D. Wang, and J. Wu, “Evaluation of Gaussian process regression kernel functions for improving groundwater prediction,” *J. Hydrol.*, vol. 603, p. 126960, Dec. 2021, doi: 10.1016/j.jhydrol.2021.126960.
[103] J. Joyce, “Bayes’ theorem,” 2003, [Online]. Available: http://seop.illc.uva.nl/entries/bayes-theorem/
[104] A. Damianou and N. D. Lawrence, “Deep gaussian processes,” in *Artificial intelligence and statistics*, PMLR, 2013, pp. 207–215. [Online]. Available: http://proceedings.mlr.press/v31/damianou13a
[105] K. Jakkala, “Deep Gaussian Processes: A Survey,” Jun. 21, 2021, *arXiv*: arXiv:2106.12135. doi: 10.48550/arXiv.2106.12135.
[106] D. H. Svendsen, P. Morales-Álvarez, A. B. Ruescas, R. Molina, and G. Camps-Valls, “Deep Gaussian processes for biogeophysical parameter retrieval and model inversion,” *ISPRS J. Photogramm. Remote Sens.*, vol. 166, pp. 68–81, Aug. 2020, doi: 10.1016/j.isprsjprs.2020.04.014.
[107] V. Dutordoir, J. Hensman, M. van der Wilk, C. H. Ek, Z. Ghahramani, and N. Durrande, “Deep neural networks as point estimates for deep Gaussian processes,” *Adv. Neural Inf. Process. Syst.*, vol. 34, pp. 9443–9455, 2021.
[108] H. Salimbeni and M. Deisenroth, “Doubly stochastic variational inference for deep Gaussian processes,” *Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, [Online]. Available:

https://proceedings.neurips.cc/paper/2017/hash/8208974663db80265e9bfe7b222dcb18-Abstract.html
[109] I. H. Laradji, M. Schmidt, V. Pavlovic, and M. Kim, "Efficient Deep Gaussian Process Models for Variable-Sized Inputs," in *2019 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2019, pp. 1–7. doi: 10.1109/IJCNN.2019.8851768.
[110] S. Reece, R. Garnett, M. Osborne, and S. Roberts, "Anomaly Detection and Removal Using Non-Stationary Gaussian Processes," Jul. 02, 2015, *arXiv*: arXiv:1507.00566. doi: 10.48550/arXiv.1507.00566.
[111] R. B. Gramacy and H. K. H. Lee, "Bayesian treed Gaussian process models with an application to computer modeling," Mar. 17, 2009, *arXiv*: arXiv:0710.4536. [Online]. Available: http://arxiv.org/abs/0710.4536
[112] A. Garriga-Alonso, C. E. Rasmussen, and L. Aitchison, "Deep Convolutional Networks as shallow Gaussian Processes," May 05, 2019, *arXiv*: arXiv:1808.05587. doi: 10.48550/arXiv.1808.05587.
[113] M. Van der Wilk, C. E. Rasmussen, and J. Hensman, "Convolutional gaussian processes," *Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, Accessed: Aug. 14, 2025. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/1c54985e4f95b7819ca0357c0cb9a09f-Abstract.html
[114] B.-J. Lee, J. Lee, and K.-E. Kim, "Hierarchically-partitioned Gaussian process approximation," in *Artificial Intelligence and Statistics*, PMLR, 2017, pp. 822–831. Accessed: Aug. 14, 2025. [Online]. Available: http://proceedings.mlr.press/v54/lee17a.html
[115] M. Van der Wilk, C. E. Rasmussen, and J. Hensman, "Convolutional gaussian processes," *Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/1c54985e4f95b7819ca0357c0cb9a09f-Abstract.html
[116] M. A. Alvarez, L. Rosasco, and N. D. Lawrence, "Kernels for vector-valued functions: A review," *Found. Trends® Mach. Learn.*, vol. 4, no. 3, pp. 195–266, 2012.
[117] K. J. Koh and F. Cirak, "Stochastic PDE representation of random fields for large-scale Gaussian process regression and statistical finite element analysis," Sep. 05, 2023, *arXiv*: arXiv:2305.13879. doi: 10.48550/arXiv.2305.13879.
[118] "Online Survey Software | Qualtrics Survey Solutions." Accessed: Aug. 29, 2026. [Online]. Available: https://dotocio.gov1.qualtrics.com/jfe/form/SV_1NdO5jpfKbPKNrT?Q_CHL=si&Q_CanScreenCapture=1
[119] C. Carter, "Great circle distances," *SiRF White Pap.*, 2002, Accessed: Sep. 07, 2026. [Online]. Available: https://inventeksys.com/wp-content/uploads/2011/11/GPS_Facts_Great_Circle_Distances.pdf
[120] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *kdd*, 1996, pp. 226–231. Accessed: Aug. 29, 2026. [Online]. Available: http://cdn.aaai.org/KDD/1996/KDD96-037.pdf
[121] R. Scitovski and K. Sabo, "DBSCAN-like clustering method for various data densities," *Pattern Anal. Appl.*, vol. 23, no. 2, pp. 541–554, 2020.
[122] H. Nickisch and C. E. Rasmussen, "Approximations for binary Gaussian process classification," *J. Mach. Learn. Res.*, vol. 9, no. 10, pp. 2035–2078, 2008.

[123] T. O. Hodson, “Root mean square error (RMSE) or mean absolute error (MAE): When to use them or not,” *Geosci. Model Dev. Discuss.*, vol. 2022, pp. 1–10, 2022.
[124] N. J. Nagelkerke, “A note on a general definition of the coefficient of determination,” *biometrika*, 1991, Accessed: Sep. 07, 2026. [Online]. Available: https://www.academia.edu/download/84326379/6b6e9ba4050a20f2980bea1de35d23882b51.pdf
[125] S. Arnold, E.-M. Walz, J. Ziegel, and T. Gneiting, “Decompositions of the mean continuous ranked probability score,” *Electron. J. Stat.*, vol. 18, no. 2, pp. 4992–5044, 2024.
[126] L. Rüschendorf, “The Wasserstein distance and approximation theorems,” *Probab. Theory Relat. Fields*, vol. 70, no. 1, pp. 117–129, Mar. 1985, doi: 10.1007/BF00532240.
[127] J. H. Drew, A. G. Glen, and L. M. Leemis, “Computing the cumulative distribution function of the Kolmogorov–Smirnov statistic,” *Comput. Stat. Data Anal.*, vol. 34, no. 1, pp. 1–15, 2000.
[128] M. L. Menéndez, J. A. Pardo, L. Pardo, and M. del C. Pardo, “The jensen-shannon divergence,” *J. Frankl. Inst.*, vol. 334, no. 2, pp. 307–318, 1997.
[129] M. Schäfer, M. Strohmeier, V. Lenders, I. Martinovic, and M. Wilhelm, “Bringing up OpenSky: A large-scale ADS-B sensor network for research,” in *IPSN-14 Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, Apr. 2014, pp. 83–94. doi: 10.1109/IPSN.2014.6846743.
[130] T. Li, B. Wang, F. Shang, J. Tian, and K. Cao, “Threat model and construction strategy on ADS-B attack data,” *IET Inf. Secur.*, vol. 14, no. 5, pp. 542–552, Sep. 2020, doi: 10.1049/iet-ifs.2018.5635.
[131] “AirNav: KDCA - Ronald Reagan Washington National Airport.” Accessed: Sep. 01, 2026. [Online]. Available: https://www.airnav.com/airport/KDCA

BIOGRAPHY

Oleksandra Snisarevska received her Bachelor of Science in System Engineering from National Technical University of Ukraine 'Igor Sikorsky Kyiv Polytechnic Institute' in 2015, then went on to George Mason University to receive her Master of Science in Systems Engineering in 2018. After receiving her Doctor of Philosophy in Systems Engineering and Operations Research from George Mason University in 2026, she will continue to work at Bell Textron Inc. in Fort Worth, Texas, as a Principal Engineer.