The Thesis Committee for Previna Arumugam
Certifies that this is the approved version of the following Thesis:

Efficient Detection of Unintended Lateral Migration of CO_2: An Example from the Onshore Gulf Coast (Texas–Louisiana, USA)

APPROVED BY
SUPERVISING COMMITTEE:

Dr. Alexander Bump, Supervisor

Dr. Susan Hovorka, Co-Supervisor

Dr. Hailun Ni, Reader

Efficient Detection of Unintended Lateral Migration of CO_2: An Example from the Onshore Gulf Coast (Texas–Louisiana, USA)

by

Previna Arumugam

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Energy and Earth Resources

The University of Texas at Austin

August 2025

Dedication

To my mother, godparents, family, and friends, your strength and unwavering belief in me continue to push me farther than I ever thought possible.

To my colleagues at Hibiscus Petroleum — your encouragement helped me cultivate my passion for CCS and inspired me to chase bigger dreams.

To Yayasan Pahang and my home state, thank you for making this journey to the United States possible.

To Nur’ain Zalia and Dr. Hijaz Kamal Hasnan — your support opened the door to this incredible opportunity and made this Master’s journey a reality.

To Mark Hafle — your mentorship throughout my time in the U.S. has been a steady source of inspiration and guidance.

And to everyone who stood by me through thick and thin — this is for you.

Acknowledgements

I want to express my sincere gratitude to everyone who contributed to my academic, technical, and personal growth throughout this thesis journey.

Thank you to the Jackson School of Geosciences and the Energy and Earth Resources (EER) Department at The University of Texas at Austin for awarding the EER Fellowship and for fostering a supportive environment that encouraged interdisciplinary learning exploration. Special thanks to Dr. Fred Beach for his ongoing support. I am deeply grateful to Monica Reed, whose assistance made it possible for me to attend the IEAGHG CCS Summer School 2024, an experience that significantly broadened my knowledge and perspective on carbon capture and storage, and fueled the passion reflected in this thesis. I also wish to thank Shelby Escobar for her consistent guidance and encouragement throughout the program.

I am deeply thankful to the Gulf Coast Carbon Center (GCCC) and industrial sponsors for supporting this research. Special thanks to Dr. Alex Bump and Dr. Susan Hovorka for their invaluable mentorship, which greatly shaped my thinking and approach. I also thank German Chaves for laying important groundwork through his earlier thesis. I am grateful to Dr. Shuvajit Bhattacharya and Dr. Chao Li for their guidance in rock physics, and to Dr. Sergey Fomel and Rebecca Gao for their help with geophysical modeling. The energy and knowledge of the GCCC team were a constant source of inspiration. And also Dr. Andrey Bakulin to share the cost insights on the onshore seismic survey cost ranges.

Finally, I acknowledge the assistance of ChatGPT, Copilot, Notebook LLM, and other AI tools, which helped expedite my coding and problem-solving throughout this work.

Abstract

Efficient Detection of Unintended Lateral Migration of CO_2: An Example from the Onshore Gulf Coast (Texas–Louisiana, USA)

Previna Arumugam, MSEER

The University of Texas at Austin, 2025

Supervisor: Alexander Bump, Susan Hovorka

Carbon capture and storage (CCS) plays a pivotal role in reducing atmospheric CO_2, but its effectiveness hinges on reliable and cost-efficient monitoring, particularly in geologically complex regions like the Gulf Coast. Seismic monitoring, particularly time-lapse (4D) surveys, is often regarded as the industry standard, heavily influenced by pioneer projects like Sleipner. While seismic provides visually compelling and technically rich data, it is expensive and often impractical or unnecessary.

This thesis aims to develop a cost-effective, risk-based monitoring framework by characterizing model uncertainty, evaluating spatial and temporal risk zones, and aligning the monitoring strategy with actual containment risks based on CO_2 plumes. The methodology follows a streamlined logic: “model – map – monitor”. The study identifies *when* and *where* monitoring is most valuable using a case study of ensemble reservoir simulations—multiple realizations of subsurface behavior under uncertainty using spatial and temporal analysis on gas saturation outputs.

Synthetic seismic modeling supports the detection limits of seismic methods, showing that a 5% CO_2 saturation threshold defines the extent of detectable plumes without noise incorporation. Based on a cost analysis of various seismic monitoring methods, targeted 2D seismic surveys guided by spatial heatmaps and temporal windows demonstrate a potential reduction in monitoring costs compared to blanket 3D surveys, without compromising containment assurance.

The study recommends a shift in regulatory and operational practice from assumption-driven, one-size-fits-all requirements toward adaptive, risk-based, and site-specific monitoring strategies. This approach enhances economic viability and improves long-term storage security, thereby supporting the broader deployment of CCS technologies. Future work should incorporate pressure outputs from uncertainty models to generate pressure-based heatmaps, enabling a combined plume and pressure map to strengthen targeted risk-based monitoring.

Table of Contents

List of Tables 11

List of Figures 12

Chapter I: Introduction 15

1.1. Statement of Problem 16

1.2. Objectives 23

1.3. Relevance 23

1.4. Chapter Organization 26

Chapter II: Research Background 28

2.1. Regulatory Framework and the Need for Monitoring Plans 28

2.2. Geological complexity of the U.S. Gulf Coast 29

2.3. Geophysics for Efficient CO_2 Plume Detection 31

2.3.1. Seismic Forward Modeling 32

2.3.1.1. Rock Physics Modeling 32

2.3.1.2. Synthetic Seismic Generation 33

2.3.1.3. Detectability Threshold from Forward Modeling 34

2.3.2. Active Seismic Monitoring Methods 37

2.3.2.1. Surface Seismic Monitoring 38

2.3.2.2. Vertical Seismic Profiling (VSP) 39

2.3.2.3. Distributed Acoustic Sensing (DAS) 40

2.3.3. Integration with Regulatory Framework 41

2.4. Monitoring Strategy Development 43

2.4.1. Prior Work 44

2.4.1.1. Single flow-unit model 45
2.4.1.2. Full-field model 48
2.5. Gaps in Literature 50
Chapter III: Methodology 52
3.1. Input Reservoir Models....52
3.2. Targeted Risk-Based Monitoring Strategy 53
3.2.1. Expressing Model Uncertainty 53
3.2.2. Generating Heatmaps....55
3.2.2.1. Spatial Analysis of Plume Migration....55
3.2.2.2. Temporal Analysis of Plume Migration 56
3.2.3. Enabling Targeted Risk-Based Monitoring 57
3.3. Seismic Forward Modeling and Detectability Analysis 58
Chapter IV: Results & Analysis....63
4.1. Input Reservoir models....63
4.2. Targeted Risk-based Monitoring Strategy....65
4.2.1. Spatial Analysis 65
4.2.2. Temporal Analysis....69
4.3. Synthetic Seismic Detectability Analysis 72
Chapter V: Discussion 77
5.1. Spatial and Temporal Risk Zones....77
5.2. Seismic Detectability and Monitoring Limits....80
5.3. Seismic cost and practical consideration 83
Cost Model....85
5.4. Recommendation 90

Chapter VI: Conclusion 92

References..........95

Appendix (or Appendices)..........103

Appendix A: Computer Modeling Group (CMG) Simulation Input File (.sif) 103

Appendix B: Python Workflow for Spatial and Temporal Gas Saturation Analysis 104

Appendix C: Computer Modeling Group (CMG) Geostatistical software library (.GSLIB) 131

Appendix D: Seismic Forward Modeling Python Code (Gao et al., Unpublished)132

Appendix E: Seismic Forward Modeling Madagascar Code (Fomel, 2024; Fomel et al., 2013; Gao et al., Unpublished) 370

List of Tables

Table 1 Seismic Monitoring Comparison based on Literature Synthesis..........................41

Table 2 Cost estimate for a single onshore survey based on spatial analysis (cost ranges from Andrey Bakulin, personal communication, 2025)....................87

Table 3 Cost estimate based on temporal analysis..89

List of Figures

Figure 1 Schematic illustration of a deep CO_2 storage project showing regulatory protection of underground sources of drinking water (USDWs) (Pett-Ridge et al., 2023) 19

Figure 2 Simplified CO_2 plume migration with radial and asymmetrical unintended lateral migration across lease boundary which could pose a risk of interference to nearby aquifer, producing fields, and leakage to vertical path (fault and wells). 20

Figure 3 A realistic CCS project landscape, featuring overlapping developments, extensive faulting, and numerous legacy wells, underscores the need for advanced monitoring and risk management. 21

Figure 4 Forecasted modeled plume fingering south of the lease boundary. (*2023CH06676*, 2023) 25

Figure 5 Fluvial Channel Geometries Representing Subsurface Uncertainties. Four end-member fluvial channel configurations used in the single flow-unit model: (a) Continuous-Narrow, (b) Continuous-Wide, (c) Discontinuous-Narrow, and (d) Discontinuous-Wide. 46

Figure 6 Synthetic Sub-Seismic Fault Configurations and Transmissibility Values illustrating the fault orientation (0°, 45°, 90°, 135°) and transmissibility (from 0.0 to 1.0). 47

Figure 7 CO_2 plume saturation at Year 200 across model realizations. Quadrants vary by fluvial channel characteristics, columns by fault transmissibility, and rows by fault orientation. All models dip NW–SE. 48

Figure 8 Full-field facies model with faults mapped from 3-D seismic. The model spans approximately 71,500 ft × 79,500 ft (~21.8 km × 24.2 km) horizontally with a total vertical thickness of 2,800 ft (~0.85 km). 49

Figure 9 20 representative CO_2 realizations. The base case is highlighted with a square. 54

Figure 10 Facies model original grid 250 ft × 250 ft × 14 ft (left), regularized new grid (right) 250 ft × 250 ft × 5 ft. 60

Figure 11 Synthetic Seismic cube visualization in Petrel (Collaborators: Rebecca Gao and Dr Sergey Fomel) The cube spans approximately 72,000 ft × 78,500 ft (~21.95 km × 23.93 km) horizontally and 2,100 ft (~0.64 km) vertically, at a resolution of 250 ft × 250 ft × 5 ft. 61

Figure 12 Full-field model (base case) CO_2 plume saturation clipped to extent of the plume only. 64

Figure 13 Base case (top), Stacked plume (middle), and heatmaps (bottom). 68

Figure 14 Migration distance versus time for 200 years. Dashed line indicates end of injection at 30 years. 70

Figure 15 Migration distance versus time for 50 years. Dashed line indicates end of injection at 30 years. 71

Figure 16 Dynamic simulation of CO_2 plume saturation five years after injection (left), 30 years post-injection (right). 73

Figure 17 Seismic amplitude difference between the baseline (pre-injection) and two time-lapse snapshots: after five years of injection (left) and thirty years post-injection (right). 74

Figure 18 The red contour shows the CO_2 plume front; the yellow contour indicates the seismic detectability limit based on amplitude response. Analysis on travel times may offer higher sensitivity on the limit (Barnett et al., 2025). Scale is in feet. 76

Figure 19 CO_2 plume migration risk maps from the single flow-unit model. Left: Base case maximum gas saturation. Center: Stacked maximum saturation from all 20 realizations. Right: Areas where the ensemble diverges from the base case, showing where unacceptable plume spread is most likely (high-risk corridor). Model area is 71,500 ft × 94,500 ft (~21.8 km × 28.8 km) with 500 ft grid spacing. 86

Chapter I: Introduction

The imperative to combat climate change has spurred extensive efforts worldwide towards developing and implementing carbon capture and storage (CCS) solutions. To provide high levels of assurance that storage will be permanent requires accurate characterization and rigorous risk assessment of potential CO_2 storage units. In the United States, regulatory frameworks under the EPA's Class VI Underground Injection Control (UIC) Rule mandate that operators must predict, monitor, and demonstrate control over both the lateral and vertical migration of injected CO_2 to ensure protection of underground sources of drinking water (USDWs) (UIC, 2013a, 2013b). Conformance and containment represent critical aspects of commercial-scale geologic carbon storage, focusing on closing the gap between the reservoir model and the actual observation of CO_2 plume distribution underground, and ensuring long-term security and environmental protection through containment. Significant costs are associated with monitoring activities; however these measures are necessary to safeguard the environment and address economic considerations.

The U.S. Gulf Coast basin has been shaped by complex depositional and tectonic processes over the Cenozoic era, producing one of the world's most prolific hydrocarbon provinces. The early Miocene depositional framework highlights significant progradation episodes that were built across a submerged shelf platform (Galloway, 1989). This sequence is bounded by key stratigraphic markers, including the Anahuac and Amphistegina B shales, and consists of systems such as the Santa Cruz fluvial and North Padre delta systems. These systems were characterized by wave-dominated deltas and barrier/strandplain complexes, which facilitated extensive sediment deposition along the Texas Coastal Plain and offshore regions. Hydrocarbon exploration in these reservoirs has produced significant resources, primarily associated with growth faulting and salt dome provinces.

The U.S. Gulf Coast has rapidly emerged as a world-class opportunity for carbon capture and storage (CCS), with more commercial-scale projects in development due to the region's capacity (Meckel & Treviño, 2014). The area has recently experienced a surge in CCS activity, driven by aggressive decarbonization targets, favorable geology, and strong policy incentives, such as the 45Q tax credit. The Gulf Coast presents a uniquely complex and challenging environment, characterized by highly heterogeneous Miocene fluvial-deltaic reservoirs, extensive fault networks, over a million legacy wells, and in some cases tightly constrained, irregular lease boundaries. These geological and operational challenges are further compounded by competing land uses and overpopulated hydrocarbon infrastructure. This intersection of opportunity and risk makes the Gulf Coast a valable test case for advancing safe, reliable, and cost-effective monitoring at scale. While this research focuses on the Texas-Louisiana Gulf Coast as a representative example of subsurface, the targeted risk-based monitoring strategies developed here broadly apply to CCS projects in diverse geological settings worldwide.

1.1. Statement of Problem

Carbon capture and storage (CCS) is one of the few scalable technologies for reducing atmospheric CO_2 emissions and mitigating climate change (IPCC et al., 2023). CCS involves injecting the captured CO_2 deep underground into a secure geological formation, thereby preventing its release into the atmosphere. After decades of pilot projects demonstrating the feasibility of CO_2 injection it is transitioning to commercial-scale deployments worldwide to meet the need for large-scale emission reduction. The rapid growth of CCS projects along the U.S. Gulf Coast has increased the need for monitoring strategies that are both effective and affordable. This expansion is primarily driven by incentives like the 45Q tax credit, which supports projects that capture and store large amounts of CO_2. Unlike an upfront subsidy, 45Q rewards projects over time: companies can claim tax credits for 12 years, but only after their CCS facilities are up and running and actively storing CO_2. This means the financial benefit is gradual, providing steady motivation to keep CO_2 securely stored year after year (Victor & Nichols, 2022).

However, the economics of CCS have undergone significant changes. Instead of relying on direct government funding, operators must demonstrate that their projects are cost-effective and meet stringent regulatory requirements. Chief among these is the EPA's Class VI Underground Injection Control (UIC) Rule, which requires ongoing monitoring as a condition for obtaining and maintaining a permit. The UIC Rule requires operators to demonstrate their ability to control and contain the injected CO_2, with monitoring plan focused on predicting plume and pressure front movement within a defined Area of Review (AoR) and preparing corrective action plans if migration extend beyond modeled boundaries. While monitoring is required to detect any allowed movement and leakage, but the framework focuses on modeling, predicting and preparing for CO_2 movement rather than directly detecting the leaks if any. In other word the framework is preventive, not reactive ultimately protecting underground sources of drinking water (USDWs).

Early science-driven projects, such as Sleipner and Illinois Basin-Decatur, set a precedent for intensive and expensive monitoring programs that are unsustainable at scale. The challenge now is to maintain the same level of safety and regulatory assurance but with a targeted, risk-based, and economically viable monitoring approach.

The U.S. Gulf Coast presents a uniquely challenging subsurface environment. Its complex geology, including heterogeneous fluvial-deltaic systems, sub-seismic faults, salt structures, and over a million legacy wells, introduces significant uncertainty in predicting and tracking CO_2 plume migration. These uncertainties are compounded by tightly constrained and irregular project boundaries, where even minor lateral plume migration away from the planned area may result in regulatory non-compliance.

In order to safeguard USDWs, the EPA enforces strict standards through the Class VI UIC permit, as mandated by the Safe Drinking Water Act (SDWA) in 40 CFR Parts 146.84 and 146.90. These rules require operators to demonstrate that CO_2 injected deep underground will remain securely contained beneath the primary confining zone, preventing migration into protected aquifers (Figure 1). As a result, regulatory requirements directly influence initial site screening, favoring geologic settings with thick, laterally continuous confining units and minimal faulting, as well as the design of monitoring systems that can rapidly detect any pressure changes or plume migration near the confining zone. Ultimately, these regulations shape where storage projects can be sited and the data that must be collected, modeled, and reported throughout the project lifecycle.

The U.S. Environmental Protection Agency (EPA), through the Class VI Underground Injection Control (UIC) Program, requires operators to demonstrate both elevated pressure and CO_2 plume containment within the Area of Review (AoR) during both the injection and post-injection phases. While the EPA outlines required monitoring activities, such as mechanical integrity testing and pressure monitoring, the program remains non-prescriptive, mainly regarding specific methodologies, frequencies, and spatial coverage. EPA designed the non-prescriptive program to allow maturation of optimal monitoring programs and flexibility to accommodate site specific factors.

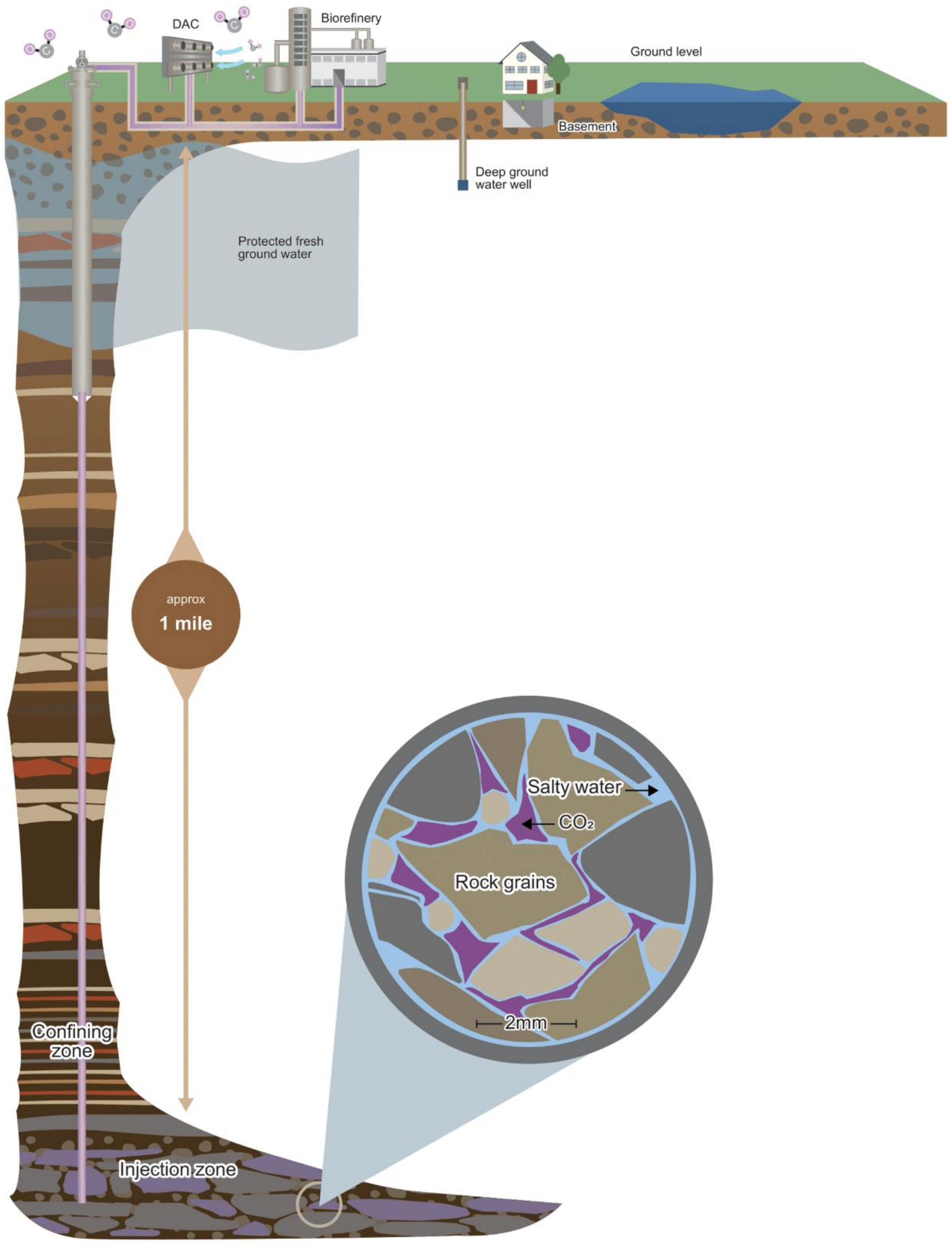


Figure 1 Schematic illustration of a deep CO_2 storage project showing regulatory protection of underground sources of drinking water (USDWs) (Pett-Ridge et al., 2023)

Most screening for CCS projects begins with the assumption of isotropic reservoirs and radial plume spread (Figure 2). However, geological heterogeneity and anisotropy often lead to asymmetric plume behavior, including unintended lateral migration beyond lease boundaries. Such deviations pose risks to underground drinking water sources (USDWs), nearby producing fields, and potential vertical leakage pathways such as faults and legacy wells that may be encountered by the unexpected plume migration. These risks challenge the integrity of containment and raise concerns about environmental safety, legal compliance, and public trust.

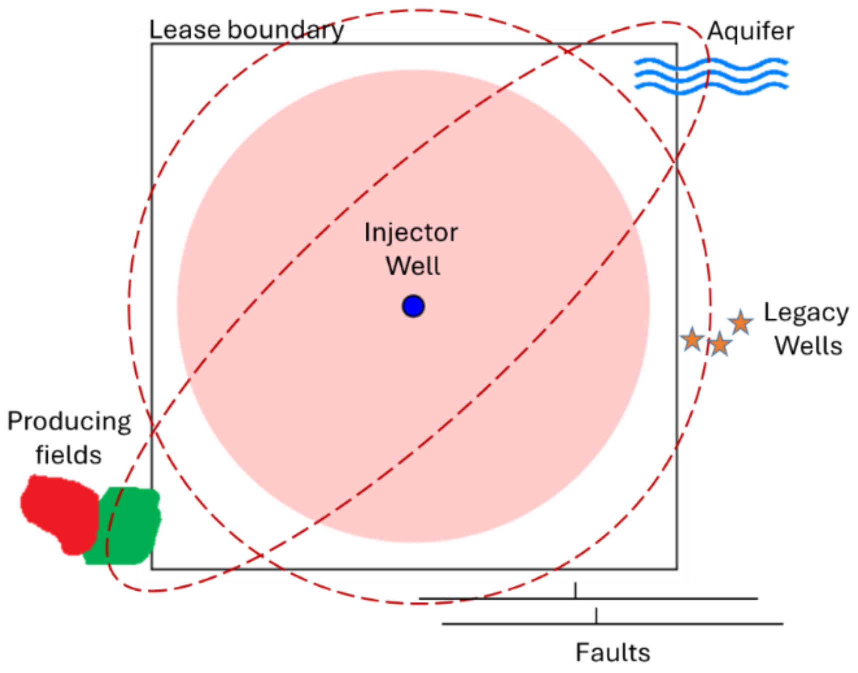


Figure 2 Simplified CO_2 plume migration with radial and asymmetrical unintended lateral migration across lease boundary which could pose a risk of interference to nearby aquifer, producing fields, and leakage to vertical path (fault and wells).

In practice, the subsurface landscape for CCS is complex (see Figure 3). Rather than a blank slate with uniform geology, the Gulf Coast is crowded with thousands of historic hydrocarbon fields, multiple injection projects, extensive fault networks, and tens of thousands of legacy wells. Individual storage leases often have irregular and fragmented boundaries, legacy wells and adjacent hydrocarbon fields, all of which limit acceptable CO_2 migration. In such settings, both plume and pressure must be closely monitored and managed per EPA requirements (UIC Class VI, 40 CFR 146.84 and 146.90) to ensure ongoing protection of USDWs and regulatory compliance.

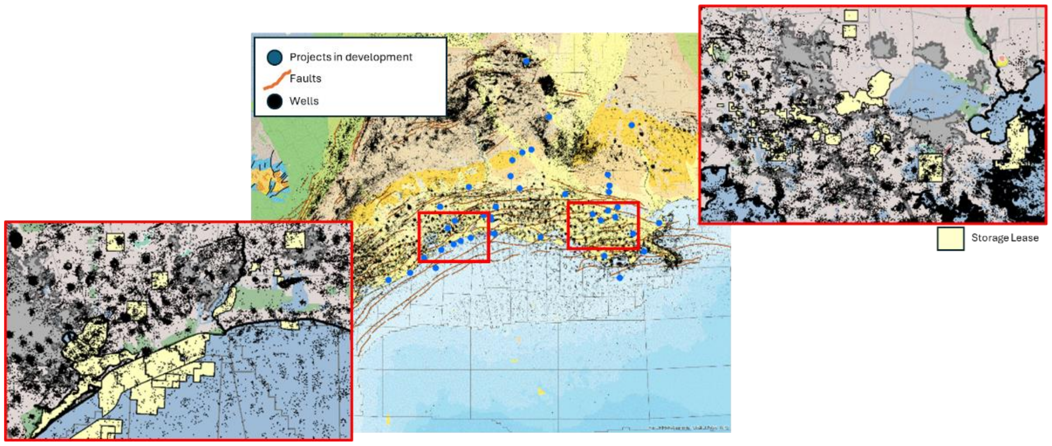


Figure 3 A realistic CCS project landscape, featuring overlapping developments, extensive faulting, and numerous legacy wells, underscores the need for advanced monitoring and risk management.

These complexities significantly increase the importance of reliable tracking of CO_2 plume migration and pressure buildup, thereby elevating the importance and challenge of the monitoring strategy. Traditional tools, such as time-lapse (4D) seismic monitoring, are widely regarded as a powerful tool for visualizing CO_2 plume evolution. It enables the detection of changes in subsurface elastic properties associated with CO_2 saturation. However, seismic surveys are expensive and typically conducted at multi-year intervals, which limits their temporal resolution. In most onshore settings, seismic detectability thresholds also limit ability to map the actual plume edge, instead providing a somewhat fuzzy edged detection of the areas of significant CO2 thickness and saturation. A generic cost estimate based on the IEAGHG Monitoring Selection Tools (IEAGHG, 2019), such as 2D and 3D seismic surveys, is classified as high-cost due to the sophisticated data acquisition and processing required. Well monitoring is moderately expensive, while they evaluate surface monitoring is the lowest-cost option. However, neither monitoring wells nor surface monitoring provides a map of the subsurface plume that meets UIC regulatory expectations. It is essential to note that these cost estimates are approximate and may vary significantly depending on project specifics, technological advancements, and market dynamics.

This thesis addresses the inefficiencies in monitoring strategies by evaluating the impact of subsurface uncertainties on CO_2 migration. It proposes a cost-effective, targeted risk-based monitoring framework for unintended lateral migration and supports it through the time-lapse detectability of CO_2 plume migration. Using the Miocene fluvial-deltaic reservoirs of the Texas-Louisiana Gulf Coast as a case study, this research integrates reservoir simulation and spatial-temporal risk analysis to identify high-risk zones and optimize monitoring well and survey placement, if necessary, as well as the frequency of subsurface monitoring. While the workflow is tailored to the Gulf Coast's complex geology and operational realities, the principles and methodologies are broadly applicable to CCS projects in diverse geological and regulatory environments worldwide. The overarching goal is to proactively detect unintended lateral migration of CO_2, ensure regulatory compliance, and support CCS's safe, reliable, and economically viable deployment at scale.

1.2. Objectives

The primary objective of this study is to develop a cost-effective, targeted risk-based monitoring framework for detecting unintended lateral migration of CO_2 plumes in geologically complex carbon storage reservoirs. The case study is on the Miocene fluvial-deltaic formations of the TX-LA Gulf Coast, building from reservoir and seismic models developed recently by collaborators.

To achieve this, the research is organized into three main parts

1. Characterize Model Uncertainty and CO_2 Plumes Evolution

Quantitatively assess the spatial and temporal variability of CO_2 plume migration using ensembles of reservoir simulations. Identify the range of possible plume extents and highlight high-risk zones where lateral migration could extend beyond the project boundaries.

2. Design and Evaluate Risk-Based Monitoring Strategy

Develop spatial and temporal heat maps from fluid-flow simulation results to identify an optimal monitoring plan, i.e., placement of monitoring wells and seismic survey frequency. Propose a targeted monitoring approach focusing resources on areas and times of highest risk, accounting for real-world operational constraints.

3. Identify and Quantify the Limitations of Seismic Detectability

Synthetic seismic response modeling (via Gassmann fluid substitution on gas saturation maps) generates amplitude maps. Demonstrate that the seismic amplitude anomaly is consistently smaller than the actual CO_2 plume extent and explicitly quantify this gap as the practical limitation in detecting plume migration.

1.3. Relevance

This research introduces a new outlook on carbon capture and storage (CCS) project monitoring strategies. Traditionally, monitoring design has relied heavily on stochastic or statistical sensitivity analyses, such as tornado plots to evaluate uncertainty. While useful, these methods often fail to capture the spatial and temporal complexity of plume migration in geologically heterogeneous reservoirs.

This study moves beyond conventional approaches by expressing model uncertainty through multiple geological realizations of a base-case reservoir model. Instead of treating uncertainty as abstract statistical variation, it is visualized as a range of possible plume behaviors. These variations generate spatial and temporal heatmaps to identify *where* and *when* monitoring should be focused. This approach provides a more intuitive and actionable understanding of plume migration risk.

Addressing this problem is a multifaceted and relevant issue, encompassing critical aspects such as environmental protection, reputation maintenance, and operational efficiency. Effective monitoring is crucial to demonstrating compliance with permit conditions. Where characterization is insufficient to rule out all undesirable outcomes, monitoring may also be a safeguard for underground sources of drinking water. Lastly, monitoring may offer reassurance to a skeptical public and even protection from lawsuits alleging harm from storage operations (Romanak et al., 2014). These goals must be balanced against cost. From an operational perspective, inefficient or overly conservative monitoring can increase costs and compromise the commercial viability of CCS projects, particularly under today's tax credit-driven incentives.

The first commercial CCS facility in the U.S., operated by Archer Daniels Midland Company (ADM) in Decatur, Illinois, a lawsuit against ADM, alleges trespass, nuisance, and unjust enrichment (Figure 4) due to possible unintended migration of CO_2 beyond the operator's storage lease. This underscores the critical need for effective monitoring to prevent such infringements and mitigate risks associated with carbon capture projects (*2023CH06676*, 2023). The primary evidence in the case is the defendant's reservoir model, which predicted trespass five years into the future. However, whether this prediction would materialize remains uncertain. Nonetheless, the ongoing lawsuit is a significant example of the potential repercussions of unintentional migration.

Another allegation against the same operator has been criticized following two CO_2 leaks near an underground drinking water source (USDW), raising concerns about transparency and regulatory oversight (Ramirez-Franco, 2024). The EPA identified corrosion-induced fluid migration in the deep monitoring well VW#2, prompting ADM to isolate the affected zones with bridge plugs and confirm that there was no impact on surface or groundwater (ADM, 2024). While ADM emphasized its commitment to safety and regulatory compliance, media coverage portrayed the incident in a negative light, fueling public concern. This contrast between technical containment and public perception underscores the crucial need for effective, transparent monitoring systems to identify and mitigate risks, preserve public trust, and counter misinformation.

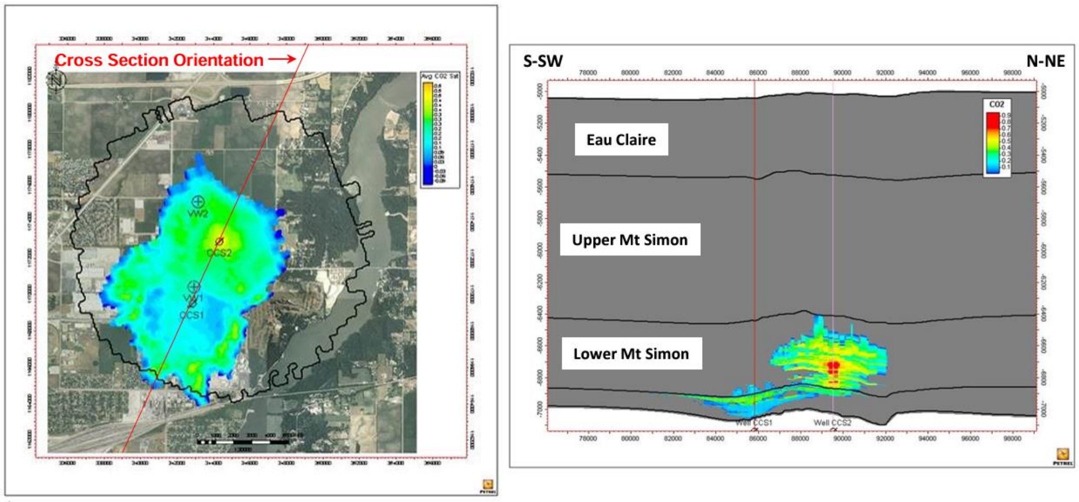


Figure 4 Forecasted modeled plume fingering south of the lease boundary. (*2023CH06676*, 2023)

The relevance of this work is underscored by real-world challenges, such as legal disputes over CO_2 trespass and public concerns about leaks near drinking water sources. These issues highlight the need for technically sound monitoring strategies, which are also transparent, cost-effective, and capable of early detection. By integrating reservoir modeling, targeted risk-based analysis, and seismic forward modeling, this study provides a practical framework for enhancing monitoring effectiveness and regulatory compliance in CCS projects, particularly in geologically complex regions such as the Gulf Coast. Ultimately, the goal is to nudge the EPA to shift away from conventional methods towards targeted, parsimonious monitoring where it makes a difference.

1.4. Chapter Organization

This thesis is structured to build context from the regional and technical background, through project-specific modeling, into the development and implications of a new monitoring strategy.

The chapters are organized as follows:

Chapter II establishes the regulatory framework for CCS monitoring on the Gulf Coast and provides a summary of the relevant geological complexity of the Gulf Coast explored and its relevance to the anonymized commercial project based on prior work (Chaves, 2024).

Chapter III details the methodology employed in this research to address the proposed problem, including ensemble reservoir simulations (Chaves, 2024), the targeted risk-based monitoring framework developed in this research, and the synthetic seismic modeling developed by Rebecca Gao and Dr. Sergey Fomel.

Chapter IV presents the primary results, including analysis of model-driven uncertainty, targeted risk-based monitoring, and the operational limits of detectability derived from synthetic seismic data.

Chapter V discusses broader implications of these results, emphasizing regulatory compliance, operational decision-making, and economic impact for CCS deployment.

Chapter VI summarizes key findings and recommendations for advancing practical, targeted risk-based monitoring strategies in geologically complex CCS settings.

Chapter II: Research Background

This chapter provides an overview of the role of monitoring in Carbon Capture and Storage (CCS). Integrating geological and geophysical understanding is important to support a safe, parsimonious monitoring strategy. The chapter aims to synthesize literature and regulatory monitoring requirements, subsurface complexity, seismic detectability based on seismic forward modeling, seismic monitoring technologies, risk-based strategy development, and an introduction to prior work.

2.1. Regulatory Framework and the Need for Monitoring Plans

The EPA's Class VI Underground Injection Control (UIC) Rule establishes a legally enforceable framework for monitoring and verifying the safe operation of geologic sequestration (GS) projects, with a particular emphasis on protecting underground sources of drinking water (USDWs). Operators are required to lease pore space for the occupation of CO_2. According to §146.84, owners or operators must predict the lateral and vertical migration of the carbon dioxide plume using computational modeling grounded in site-specific data, incorporating heterogeneities and uncertainties in the subsurface, and consider potential migration pathways such as faults, fractures, and legacy wells. Furthermore, the model must be updated at least every five years to support the re-evaluation of the Area of Review (AoR), ensuring ongoing containment and conformance throughout the entire operational and post-injection lifecycle (UIC, 2013a).

Under §146.90, Class VI projects are required to submit and follow a Testing and Monitoring Plan that confirms the project operates within permitted parameters and does not threaten USDWs. This enforceable plan must detail the strategies for monitoring the extent of the CO_2 plume and elevated pressure front throughout the project's lifespan. EPA guidance specifies at least one direct method (such as pressure sensors in the injection zone) and one indirect method (like seismic or electromagnetic surveys), unless site-specific geology renders these methods unsuitable. EPA guidance emphasizes the integration of monitoring data with numerical modeling to enhance the accuracy of predictions, particularly during AoR re-evaluations. Although the EPA adopts a non-prescriptive approach, this flexibility comes with a requirement for scientifically defensible monitoring strategies, customized to the site's complexity while still maintaining technical rigor (UIC, 2013b).

Despite the regulatory flexibility, most operators (e.g. Archer Daniels Midland Class VI Permit Application for Decatur Project) rely heavily on the EPA guidance, which suggest one direct and one indirect method may not be sufficient for approval of the Class VI permit (UIC, 2013b). Most operators appear to base their monitoring plans on an EPA sample template that includes spaces to fill the blanks for direct and indirect monitoring methods (U.S. Environmental Protection Agency, 2021). EPA guidance suggests that repeat 3D is the preferred indirect method, describing it as high resolution. To date, most, if not all Class VI permit applications on the Gulf Coast have taken that advice and offered repeat 3D seismic as their indirect method. While that approach may speed permit applications, it is expensive and logistically complex. A better approach would balance cost with systematic, site-specific analysis of project risks within the framework of regulatory requirements

2.2. Geological complexity of the U.S. Gulf Coast

The Cenozoic stratigraphy of the U.S. Gulf Coast consists of highly variable depositional systems, including fluvial-deltaic complexes, wave-dominated deltas, strandplain and barrier-lagoon systems, shelf-fed aprons, and submarine fans. (Galloway, 1989; Galloway et al., 2000). This diversity leads to complex and often discontinuous reservoir architectures that are difficult to accurately characterize with limited well logs (Bridge & Tye, 2000; Krishnamurthy et al., 2022; Larue et al., 2023). Heterogeneities, including stratigraphic pinch-outs and mudrock baffles, can disrupt lateral and vertical flow, reduce storage efficiency, and complicate predictions of plume migration. Studies emphasize that 3D sedimentary facies connectivity plays a critical role in governing both reservoir quality and plume behavior (Bump et al., 2023; Krishnamurthy et al., 2022; Meckel & Treviño, 2014).

Faulting is widespread in the U.S. Gulf Coast and plays a central role in fluid migration and trapping for many hydrocarbon fields and can be explicitly mapped and included in fluid flow models. However, minor faults that fall below seismic resolution (typically <100–200 m in trace length), are difficult to detect using conventional 3D seismic methods. These sub-seismic faults may not appear continuous, but their complexity, including slip surfaces and fault steps, influences vertical or lateral leakage and potentially have a strong impact on plume migration and stabilization (Chaves, 2024; Pickering et al., 1996).

Faulting is widespread in the U.S. Gulf Coast, where both seismic- and sub-seismic-scale faults influence fluid migration and plume behavior. While larger faults can often be mapped from seismic and incorporated into models, sub-seismic faults those with throws <30 m are below seismic resolution and difficult to detect, yet may still impact pressure dissipation and vertical containment (Chaves, 2024). German Chaves' work demonstrated that faults with transmissibility values above 0.1 generally do not significantly alter the Area of Review (AoR), but their orientation and interaction with high-permeability zones can influence plume shape.

The presence of historical and existing wells across the U.S. Gulf Coast further complicates site integrity. These wells particularly those that are poorly documented, improperly plugged, or degraded can act as potential leakage pathways if pressurized CO_2 or brine migrates toward them. Their variable construction quality and unknown subsurface condition make it difficult to ensure long-term containment, creating both environmental and financial risks. A robust monitoring strategy must include comprehensive wellbore inventories and integrity assessments, with corrective action plans for any wells located within the Area of Review (AoR). The AoR is the region surrounding the injection site where CO_2 and pressure changes could reasonably be expected to migrate the fluid during the project lifetime, as determined by numerical modeling. If plume stabilization behavior is not well constrained, the CO_2 could encounter old or unrecorded wells that were not prepared for exposure, increasing the risk of leakage and remediation costs. This study directly addresses those risks by evaluating how heterogeneity, fault transmissibility, and storage conditions affect plume stabilization over time.

The U.S. Gulf Coast is also characterized by complex pressure regimes, including widespread overpressure zones that can limit the storage window thickness and therefore storage capacity (Bump et al., 2023). Additionally, high-relief buoyant traps may allow CO_2 to accumulate and potentially overcome sealing thresholds, particularly if not adequately monitored (Bump et al., 2023; Finkbeiner et al., 2001).

In summary, the geological and structural complexities of the Gulf of Mexico introduce significant uncertainty into reservoir behavior, plume migration prediction, and pressure evolution prediction, making it a challenging yet essential testbed for developing risk-based, targeted monitoring strategies to reduce these uncertainties.

2.3. GEOPHYSICS FOR EFFICIENT CO_2 PLUME DETECTION

Monitoring geologic carbon storage requires geophysical tools that can detect and delineate CO_2 plumes with sufficient sensitivity, resolution, and spatial coverage to meet regulatory expectations to "track the plume front" as well as to derisk the operator's need to lease the pore space occupied by CO_2. Among the available methods, time-lapse (4D) seismic monitoring stands out as the most widely adopted technique for assessing conformance and containment. This section focuses on two key pillars: seismic forward modeling for plume detectability and parsimonious seismic data collection for field-scale implementation.

2.3.1. Seismic Forward Modeling

Seismic forward modeling determines whether the CO_2 plume can be detected under relevant subsurface conditions. The process integrates reservoir simulation outputs (e.g., saturation maps) with rock physics models to generate synthetic seismic responses. These datasets help evaluate the seismic detectability of plume-related changes in reservoir properties.

2.3.1.1. Rock Physics Modeling

Rock physics provides the theoretical foundation for seismic forward modeling by linking changes in fluid saturation to variations in the elastic properties of reservoir rocks. Among various approaches, the Gassmann equation is the most widely used, along with the Biot-Gassmann substitution and other empirical or heuristic models (Kazemeini et al., 2010; Smith et al., 2003; Vasco et al., 2019).

A common approach utilizes the Gassmann fluid substitution method to calculate changes in elastic moduli resulting from CO_2 replacing brine in pore spaces. These substitutions are performed under several key assumptions: (1) the shear modulus remains unchanged; (2) fluids are uniformly distributed at the seismic wavelength scale; and (3) the properties of the dry rock frame, fluid, and solid matrix are well-characterized (Arts et al., 2004; Smith et al., 2003).

The core equations are as follows.

Saturated Bulk Modulus:

$$K_{sat,new} = K_{dry} + \frac{\left(1 - \frac{K_{dry}}{K_s}\right)^2}{\frac{\phi}{K_f} + \frac{1-\phi}{K_s} - \frac{K_{dry}}{K_s^2}}$$

K_{dry}: dry rock frame bulk modulus
K_s: solid grain bulk modulus
K_f: fluid bulk modulus
ϕ: porosity

Bulk density:

$$\rho_{bulk,} = (1-\phi)\rho_s + \phi\rho_f$$

ρ_s: solid density
ρ_f: fluid density

P-wave and S-wave velocities:

$$V_p = \sqrt{\frac{K_{sat,new}+\frac{4}{3}G}{\rho_{bulk,}}}\; V_s = \sqrt{\frac{G}{\rho_{bulk,}}}$$

G: shear modulus (assumed constant)

2.3.1.2. Synthetic Seismic Generation

Once rock physics modeling yields updated values for Vp, Vs, and ρ, these parameters forms the basis for generating synthetic seismic responses. In a standard workflow, the elastic property volumes are used to compute acoustic impedance (Z), which is the product of ρ × Vp. These impedance volumes are then convolved with a representative seismic wavelet (typically a Ricker wavelet) to simulate synthetic seismograms (Kazemeini et al., 2010).

The reflectivity series R is derived from impedance contrasts at layer interfaces. For normal incidence, the reflection coefficient between two adjacent layers is:

$$R = \frac{Z_2 - Z_1}{Z_2 + Z_1} = \frac{\rho_2 V_{p2} - \rho_1 V_{p1}}{\rho_2 V_{p2} + \rho_1 V_{p1}}$$

By comparing pre-injection and post-injection states, seismic anomalies associated with the evolution of the CO_2 plume can be quantified. Studies have shown that such modeling significantly informs survey design and improves detection sensitivity, particularly in heterogeneous reservoirs (Barnett et al., 2025; Smith et al., 2003).

2.3.1.3.Detectability Threshold from Forward Modeling

Seismic forward modeling provides a predictive lens into the detectability of a CO_2 plume under real-world field conditions (i.e. presence or absence of the CO_2 plume), However, its utility depends on whether modeled anomalies exceed the detection threshold limits, which are influenced by fluid properties, reservoir conditions, and acquisition parameters. Detectability is typically assessed by comparing baseline (pre-injection) and monitor (post-injection) seismic datasets to identify measurable changes in the subsurface caused by CO_2 injection (Arts et al., 2004; White, 2011).

The primary diagnostic indicators of the presence of CO_2 include amplitude anomalies, often referred to as “bright spots” which arise from CO_2-induced contrasts in acoustic impedance, and travel-time shifts, commonly known as the “velocity push-down” effect, associated with reductions in P-wave velocity (Arts et al., 2004; Kazemeini et al., 2010; Smith et al., 2003). Time-shift analysis, which captures the delay in reflected seismic arrivals, is another commonly used method to infer plume tracking (Arts et al., 2004; Kazemeini et al., 2010). As a general guideline, a change in acoustic impedance of ~4% is considered sufficient to produce a detectable anomaly (Barnett et al., 2025).

The success of time-lapse seismic depends heavily on rock properties. Lumley (2013) notes that strong 4D seismic signals are most achievable in rocks with high porosity and high dry-frame compressibility, which increase fluid sensitivity. For example, at the Sleipner project, CO_2 injection into unconsolidated sands led to a P-wave velocity reduction of up to 60%, creating an apparent velocity push-down effect (Arts et al., 2004; Lumley, 2010). Likewise, the Ketzin site showed detectable seismic signatures for gaseous CO_2, though background noise and geological heterogeneity reduced repeatability (Kazemeini et al., 2010).

Detectability is controlled by plume size, CO_2 saturation, depth, and reservoir properties (Kazemeini et al., 2010). Larger, shallower plumes with higher saturation levels are more likely to be detected, while deeper plumes suffer from signal attenuation and reduced fluid compressibility (Gasperikova et al., 2020). Notably, when CO_2 rises above ~800 meters, it transitions from a supercritical to gaseous phase, enhancing seismic contrast due to increased volume and impedance effects (Gasperikova et al., 2020; Kazemeini et al., 2010; Lumley, 2010). In contrast, the presence of oil can dampen velocity changes and reduce seismic response (Arts et al., 2004; Lumley, 2010; Vasco et al., 2019). At Cranfield, low CO_2 saturation and hydrocarbons-obscured expected acoustic changes, limiting the reliability of time-lapse seismic in detecting emplaced CO_2 (Alfi & Hosseini, 2016).

A widely used metric for evaluating time-lapse detectability is the normalized root mean square (nRMS) difference between baseline and monitor surveys. Values below 0.4 are generally acceptable, and those below 0.2 are considered excellent (Isaenkov et al., 2021, 2022; Pevzner et al., 2021; Yurikov et al., 2022). For instance, the CO_2CRC Otway Project achieved plume detection with average nRMS values of 0.15 (Isaenkov et al., 2021; Lumley, 2010). Attaining such performance requires high signal-to-noise ratios (SNR) and precise acquisition design, including dense source-receiver spacing, optimized source frequency, and advanced processing techniques like migration and stacking (Gasperikova et al., 2020).

Acquisition geometry also plays a critical role. While denser shot-receiver arrays improve resolution and image clarity, they increase cost. Conversely, sparse geometries reduce cost but compromise sensitivity, particularly to small or deep plumes (Urosevic et al., 2011). Higher source frequencies can enhance vertical resolution but are typically impractical in deeper settings due to attenuation.

Benchmark studies such as Kimberlina-2 have been instrumental in evaluating detectability limits. This benchmark simulated 60 years of CO_2 injection and assessed the visibility of both primary and secondary plumes—those that migrate beyond the intended storage reservoir but remain within the storage complex. Results showed that 2D time-lapse seismic, paired with a 20 Hz wavelet and advanced processing (e.g., Least-Squares Reverse-Time Migration), could detect deep plumes under moderate noise conditions (SNR = 2–5). However, detectability declined sharply beyond 1,000 meters in low-SNR environments. Under ideal, noise-free conditions, 3D seismic with sparse acquisition and a 10% nRMS threshold was capable of detecting plumes as small as 20,000 tonnes (Gasperikova et al., 2020, 2022).

Wedge models are synthetic tools used to examine how seismic responses change with varying thicknesses of fluid-saturated zones. In this study, the wedge geometry acts as a proxy for a vertically resolved CO_2 plume, allowing us to assess detectability thresholds based on plume thickness, saturation, and acoustic impedance contrasts. While wedge models do not calculate plume thickness directly, they test whether a plume of a given thickness would generate a detectable seismic response. This is important because vertical plume resolution i.e., the ability to distinguish the top and bottom of the CO_2 plume depends on whether the wedge thickness exceeds the seismic tuning thickness. In this way, wedge models are an analogue for understanding detectability as a function of plume thickness, even though direct thickness measurements require other methods such as time-shift analysis or inversion using well log constraints. Recent studies (e.g., (Barnett et al., 2025; Li et al., 2024) have also used wedge models to examine the sensitivity of seismic responses to CO_2 saturation and diffusion effects. These investigations highlight the importance of optimizing SNR and accounting for patchy saturation, particularly in geologically complex formations like the Miocene fluvial-deltaic reservoirs of the Gulf Coast. According to Barnett et al (2025), time-shift analysis provides a more quantitative estimate of plume thickness by comparing seismic travel times between baseline and monitor surveys. The measured Δt can be converted to changes in velocity and, when combined with well or pseudo-well information, used to estimate plume extent.

In summary, while seismic forward modeling provides valuable insights into plume detectability, it remains constrained by plume characteristics, geologic heterogeneity, depth, and acquisition design. Detectability thresholds often defined through nRMS or impedance changes carry their own uncertainties, making seismic an inherently indirect tool. Consequently, low-saturation zones or noisy environments can lead to plume underestimation plume size, reinforcing the need for cautious interpretation and the integration of complementary monitoring techniques. Plume underestimation occurs when seismic monitoring fails to capture the full extent or volume of the CO_2 plume due to detection limits, geologic complexity, or weak seismic responses.

2.3.2. Active Seismic Monitoring Methods

Building on insights from seismic forward modeling, field-scale seismic monitoring provides the operational basis for tracking CO_2 plume evolution, verifying containment, and identifying potential leakage pathways in geologic carbon storage (GCS) projects. These methods translate modeled detectability thresholds into real-time or periodic subsurface observations. Among the most widely deployed techniques are surface seismic, Vertical Seismic Profiling (VSP), and Distributed Acoustic Sensing (DAS), each offering distinct advantages in spatial coverage, resolution, and cost.

2.3.2.1. Surface Seismic Monitoring

Time-lapse 3D surface seismic surveys are the most established method for imaging large subsurface volumes with high lateral resolution. Repeated acquisitions allow for visualization of CO_2 plume migration, and deviations from model predictions, The Sleipner project in Norway exemplifies this approach: since 1996, repeated 3D surveys have revealed strong amplitude anomalies and velocity push-down effects caused by CO_2 accumulation beneath thin intra-reservoir shales (Arts et al., 2004). These signals were detectable despite being below nominal resolution due to constructive tuning effects. In some setting pressure effects also can be detected through velocity changes. An increase in pore pressure reduces effective stress in the rock frame, which lowers P-wave and S-wave velocities, leading to travel-time delays (time shifts) and subtle changes in seismic amplitude. At the Snøhvit project, pressure was monitored through in-zone measurements, while the seismic response primarily reflected saturation changes (Alfi & Hosseini, 2016; Goudarzi et al., 2018; Hovorka et al., 2014).

At Cranfield (Mississippi), surface seismic captured amplitude changes consistent with CO_2 injection, although interpretation was complicated due to hydrocarbon interference (Vasco et al., 2019; Zhang et al., 2013). The initial Otway Project in Australia demonstrated excellent repeatability (nRMS ≈ 0.2), but the plume signals were subtle due to low elastic contrast where CO2 was injected into in a depleted gas reservoir (Urosevic et al., 2011). For instance, a 5-meter thick sand layer at 2,000 m depth failed to produce a clear reflection, as its thickness was only ~5% of the seismic wavelength (Gasperikova et al., 2020)

The Weyburn Field in Canada illustrates the interpretive complexity of 4D surface seismic. Here, a 12% decrease in acoustic impedance was observed near injection wells, attributed to both pore pressure buildup and CO_2 saturation. To distinguish between these effects, advanced tools such as converted-wave (PS) data and amplitude variation with offset (AVO) were used, demonstrating the value of multi-attribute analysis in a stratigraphically complex, thin, carbonate depleted hydrocarbon reservoirs (White, 2011).

Despite its strengths, surface seismic is constrained by high acquisition costs and limited repeat frequency and the logistical demands of repeated mobilization of sources and receivers. These activities can conflict with surface land use, disturb surface conditions, harm sensitive environments, or raise public concerns. Sparse geometries and near-surface variability can introduce aliasing and reduce data quality. These limitations highlight the need to incorporate forward modeling into survey design, particularly in optimizing source-receiver spacing, wavelet choice, and processing workflows to enhance signal-to-noise ratio (Gasperikova et al., 2020; Pevzner et al., 2021).

2.3.2.2. Vertical Seismic Profiling (VSP)

VSP offers higher vertical resolution and reduced sensitivity to near-surface noise compared to surface seismic. By placing geophones in boreholes, VSP captures wavefields closer to the injection zone, enhancing reliability. At both Cranfield and Otway, VSP detected plume migration more precisely than surface seismic, especially near the wellbore (Pevzner et al., 2021; Urosevic et al., 2011).

VSP is particularly effective in distinguishing amplitude and travel-time changes, improving plume interpretability, and aiding calibration of surface seismic data. However, spatial coverage is limited to the vicinity of instrumented wells, and installation costs can be substantial, especially for permanent geophones or fiber-optic deployments.

2.3.2.3. Distributed Acoustic Sensing (DAS)

DAS is an emerging technology that transforms fiber-optic cables into dense seismic receiver arrays. It enables high-frequency, high-density seismic data acquisition with minimal surface disruption. At Otway, DAS achieved excellent repeatability (nRMS as low as 0.02) and detected plume-related changes within days of injection (Pevzner et al., 2021; Urosevic et al., 2011).

DAS offers several advantages, including real-time acquisition, low marginal costs, and the potential to retrofit existing wells. However, its sensitivity is limited to axial strain (single component), and it can be affected by thermal or seasonal noise. DAS is best suited for near-well monitoring and is most powerful when integrated with other seismic tools across multiple wells.

Both VSP and DAS present operational trade-offs. VSP is limited by borehole availability and high hardware costs, DAS by signal dimensionality and environmental factors. Still, when combined, they offer high-resolution subsurface imaging and robust time-lapse monitoring capabilities (Arts et al., 2004; Isaenkov et al., 2021, 2022; Lumley, 2010; Pevzner et al., 2021; Urosevic et al., 2011).

To contextualize these methods, Table 1 presents a comparative overview of resolution, relative cost, and EPA preference across various seismic monitoring techniques, as synthesized from key literature sources.

Table 1 Seismic Monitoring Comparison based on Literature Synthesis

Method	**Resolution**	**Relative Cost**	**EPA Preference (UIC, 2013b)**
Time-Lapse 3D Surface Seismic	High (large-area)	1 (Very High)	Most Preferred
Vertical Seismic Profiling (VSP)	Very High (near-well)	2–3 (Moderate)	Moderately Preferred
Crosswell Seismic	Highest (inter-well)	1–2 (Very High)	Less Preferred
2D Surface Seismic	Moderate (line-based)	4 (Low)	Least Preferred
DAS (with VSP or Walkaway)	Very High (fiber)	3 (Moderate)	Emerging/Flexible
Microseismic Profiling	Low (event-based)	3–4 (Moderate-Low)	Not Recommended
Focused Seismic Monitoring (Spotlight)	Targeted High	5 (Very Low)	Emerging
Multi-Component Seismic (AVO, PS)	High (fluid/pressure differentiation)	2–3 (Moderate)	Supplementary

In summary, while active seismic methods remain central to CCS monitoring, each method has inherent limitations in terms of resolution, sensitivity, spatial coverage, or cost. High-resolution tools, such as 3D seismic and VSP, offer excellent imaging but are often expensive or spatially constrained. Emerging technologies like DAS offer real-time insights but require integration with other methods for comprehensive monitoring. These trade-offs reinforce the notion that no single tool is universally sufficient, and that monitoring strategies must be tailored to balance technical capabilities, cost-effectiveness, and site-specific risks.

2.3.3. Integration with Regulatory Framework

Following the discussion of seismic monitoring technologies, it is essential to consider their alignment with regulatory expectations under the U.S. Environmental

Protection Agency’s Class VI Underground Injection Control (UIC) rule. This rule requires operators to monitor plume and pressure front migration using both direct and indirect methods. As described in the Class VI Well Testing and Monitoring Guidance, “…Resolution and spatial coverage can be high, and, under the right conditions, this method is ideal for imaging carbon dioxide in the subsurface…” Hence, seismic monitoring, as an indirect method, is strongly recommended, particularly 3D time-lapse surface seismic, due to its ability to provide wide-area, high-resolution imaging, as demonstrated in projects such as Sleipner (UIC, 2013b).

Current best practices one frequently discussed improvement is via hybrid monitoring frameworks as mentioned in the Class VI Well Testing and Monitoring Guidance “The most comprehensive understanding of plume and pressure-front behavior will follow from an integrated interpretation of information collected from a combination of these method.” These combine intermittent, high-resolution seismic methods with continuous, lower-cost tools to maximize efficiency without sacrificing containment assurance. For example, Distributed Acoustic Sensing (DAS) enables near-real-time monitoring along existing fiber-optic installations, allowing 3D seismic to be reserved for anomaly-driven investigations e.g. CO2CRC Otway Project in Australia (Pevzner et al., 2021). Similarly, focused or "spotlight" seismic techniques can reduce cost and deployment complexity while maintaining adequate detection thresholds. This method aims to predict optimal source and receiver locations to monitor specific "strategic areas" or "Spots" of interest, identified through reservoir engineer studies (Al Khatib et al., 2021).

Complementary geophysical methods such as Electrical Resistivity Tomography (ERT), gravity, and electromagnetic (EM) surveys can further enhance monitoring capability, especially when seismic sensitivity is limited by depth, saturation, or lithology. These tools could be particularly effective in high-saturation regions where seismic signals may plateau, and they provide independent lines of evidence for plume evolution and CO_2 mass balance (Gasperikova et al., 2020, 2022).

2.4. Monitoring Strategy Development

According to (Hovorka, 2017) the Assessment of Low Probability Material Impacts (ALPMI) is a structured, hypothesis-driven methodology designed to formally link risk assessment with monitoring design in geologic carbon storage (GCS) projects. Unlike conventional resource development strategies that optimize within an expected range of outcomes, ALPMI focuses on low-probability events that, if realized, would constitute "material impacts" — quantitatively defined events or trends (specified in terms of magnitude, location, timing, and certainty) that stakeholders agree are unacceptable — and therefore "unacceptable outcomes" (e.g., leakage beyond the storage complex or induced seismicity above agreed thresholds) that signify project failure.

The ALPMI framework involves several key steps. First, it requires the definition of quantitative and measurable success criteria. Second, it involves modeling potential material impacts to understand their magnitude, timing, and evolution. Third, the response of monitoring systems is forward modeled to determine whether these impacts can be reliably detected above background noise using available equipment at planned spatial and temporal frequencies. Fourth, monitoring is executed during project deployment. Finally, the collected data are used to evaluate and report a finding of project success (Hovorka, 2017).

This systematic approach enables thorough documentation of project performance, supports the development of cost-effective and site-specific monitoring programs, and provides transparent justification for monitoring decisions to regulators, financiers, and other stakeholders. While this research draws on the ALPMI framework for inspiration, it modifies its initial steps to introduce a new "model–map–monitor" method. Building on previous work by Chaves (2024), which defined and modeled a failure scenario (Step 1), this study primarily focuses on Step 2, assessing whether that scenario meets the success criteria and partially addresses Step 3 by simulating the monitoring response, although noise is not included.

2.4.1. Prior Work

In this study, a material impact is defined as an unacceptable outcome such as loss of containment that important stakeholders would consider project failure. One potential pathway to such an outcome is CO_2 plume migration beyond the lease boundary or prepared Area of Review (AoR), potentially triggering regulatory or operational consequences. This thesis builds directly on prior work by Chaves (2024), which modeled how subsurface uncertainty could lead to such triggering events, thereby causing material impacts, in the context of CO_2 plume migration and Area of Review (AoR) delineation in CCS projects.

The study systematically evaluated how sub-seismic faults and fluvial channel heterogeneities influence plume behavior, pressure buildup, and containment integrity. Using a combination of simplified box models, single flow-unit models, and a full-field model, prior work identified key risks, including lateral plume migration beyond lease boundaries, vertical leakage along faults or legacy wells, and loss of injectivity. While sub-seismic faults were confirmed as a significant uncertainty, their effect on the AoR was found to be minimal under realistic transmissibility values (>0.1), becoming consequential only in extreme scenarios. Notably, the study also demonstrated that low-permeability zones can act as pressure buffers, supporting the concept of composite confinement. These findings underscore the importance of high-resolution reservoir characterization and risk-based simulation in developing more efficient and defensible monitoring strategies.

2.4.1.1. Single flow-unit model

The single flow-unit model, originally developed by Chaves (2024) with support from Dr. David Hoffman, is based on a structural framework for a CCS project located on the Texas–Louisiana Gulf Coast. The project name and location remain confidential. The dip direction of the structure is NE-SW, which strongly controlled modeled plume migration. The input geological model incorporates seismic interpretations, mapped horizons, hundredths of well log data, and was discretized at high resolution (143 × 189 × 231 cells, with 500 ft × 500 ft lateral spacing and 19 ft vertical resolution). For dynamic simulation, a representative 200-ft-thick flow unit at a depth of about 5000 ft was extracted from the total 2800 ft thickness of the prospective reservoir and modeled using a 143 × 189 × 10 grid (270,270 cells), balancing computational efficiency with sufficient spatial fidelity.

The simulation assumes an injection period of 30 years, followed by 170 years of post-injection monitoring, for a total simulated duration of 200 years. During injection, 1 million tonnes of CO_2 per year (1 Mtpa) is injected into the selected interval. This results in a total of 30 million tonnes of CO_2 injected over the 30-year period. While the broader storage formation may include multiple flow zones, this study focuses on a representative 200 ft-thick single flow unit, chosen to evaluate performance, plume behavior, and monitoring needs in high resolution.

To systematically capture geological uncertainty, in Chaves's study, Dr. Dave Hoffman created four end-member fluvial channel geometries—Continuous Narrow, Continuous Wide, Discontinuous Narrow, and Discontinuous Wide—He derived variograms inputs defining the range of channel geometries based on published seismic amplitude data and variogram modeling (Figure 5). Based on the thesis, "published seismic amplitude data" primarily refers to seismic amplitude extractions, which are visual representations derived from 2D and 3D seismic data. These extractions are used to identify and characterize channel geometries such as their width, thickness, wavelength, and trends, serving as a key source of information for understanding sand distribution and depositional patterns within the geological models. Petrophysical property distributions, including sand fraction (Vsand), porosity, and permeability were generated using sequential Gaussian simulation (SGS) and co-kriging, with Vsand as the primary conditioning variable.

To further explore the potential for lateral migration and containment failure, Chaves introduced derived and systematically tested based on probabilistic predictions from actual data and correlations to generate synthetic sub-seismic faults in four orientations (0°, 45°, 90°, and 135°). Fault orientation refers to the direction the fault traces across the reservoir and assigned a range of transmissibility values (0 to 1) which represent how easily fluids can flow across the fault plane to simulate worst-case scenarios (Figure 6).

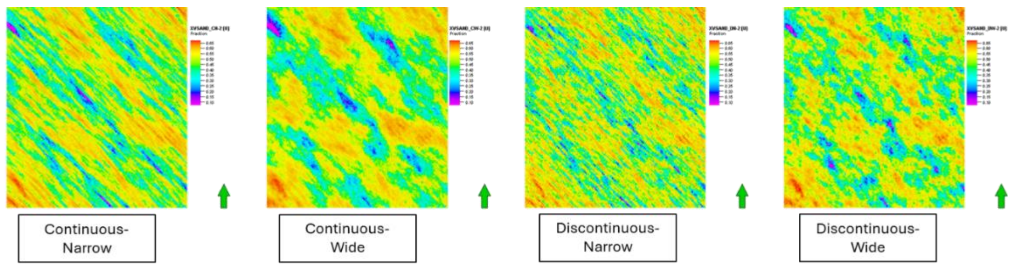


Figure 5 Fluvial Channel Geometries Representing Subsurface Uncertainties. Four end-member fluvial channel configurations used in the single flow-unit model: (a) Continuous-Narrow, (b) Continuous-Wide, (c) Discontinuous-Narrow, and (d) Discontinuous-Wide.

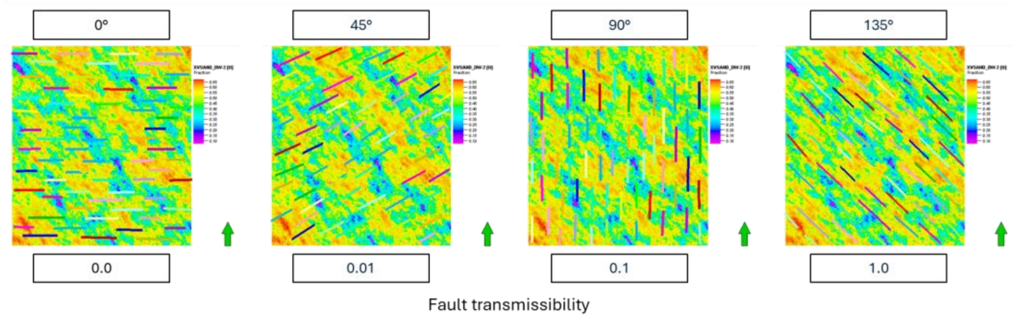


Figure 6 Synthetic Sub-Seismic Fault Configurations and Transmissibility Values illustrating the fault orientation (0°, 45°, 90°, 135°) and transmissibility (from 0.0 to 1.0).

Combining the four geological models with fault orientations and transmissibility values resulted in 64 unique simulation cases to assess CO_2 plume migration under geological uncertainty in CMG (Figure 7) Modeled CO_2 plume saturation at Year 200 across the ensemble of realizations. Each subplot represents a unique combination of geological parameters. Quadrants reflect variations in fluvial channel characteristics (e.g., width, continuity, orientation); columns vary by sub-seismic fault transmissibility; and rows vary by fault orientation. All models are oriented along a general NW–SE dipping direction. The figure illustrates how interactions between fluvial architecture and fault properties influence plume stabilization under uncertainty. 1.0 Mtpa of CO_2 injection from a single well. into a single-flow unit was simulated for 30 years of injection and 170 years post-injection, totaling 200 years of simulation.

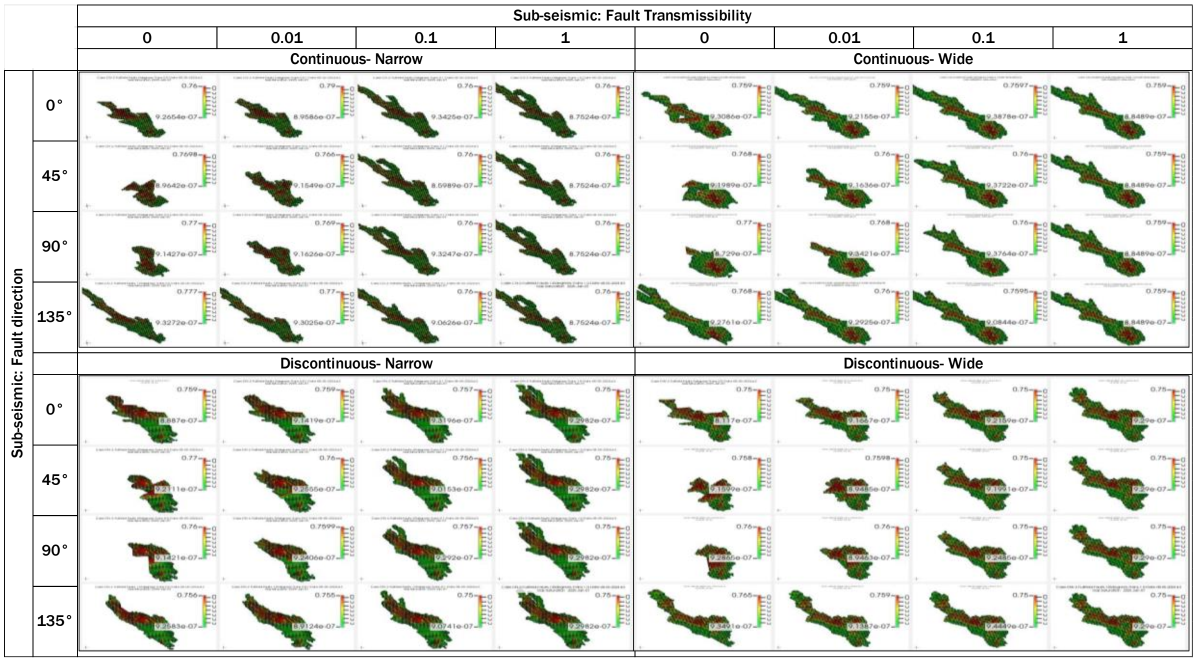


Figure 7 CO_2 plume saturation at Year 200 across model realizations. Quadrants vary by fluvial channel characteristics, columns by fault transmissibility, and rows by fault orientation. All models dip NW–SE.

2.4.1.2. Full-field model

The full-field model (Figure 8) provides a deterministic view of a single flow-unit incorporating the same operator-supplied structural data, faults mapped using 3-D seismic, and well information as the 64 models. The reservoir grid is defined at 286 × 318 × 200 cells (250 ft × 250 ft × 14 ft), with additional vertical refinement in injection zones. Where export artifacts required it, simulation faults replaced original structural faults to preserve continuity. The original framework faults caused gridding issues in the simulation model, so they were replaced with simplified "simulation faults" that preserved fault geometry but avoided artifacts during property distribution and flow simulation.

Petrophysical property fields were generated using Sequential Gaussian Simulation (SGS), conditioned by lithofacies from Sequential Indicator Simulation (SIS), and informed by regional depositional trends. A water injection step rate test (SRT) provides the observational data used to calibrate the model by adding and permeability multipliers to match observed pressures, resulting in a robust static model ready for dynamic simulation in CMG (Chaves, 2024).

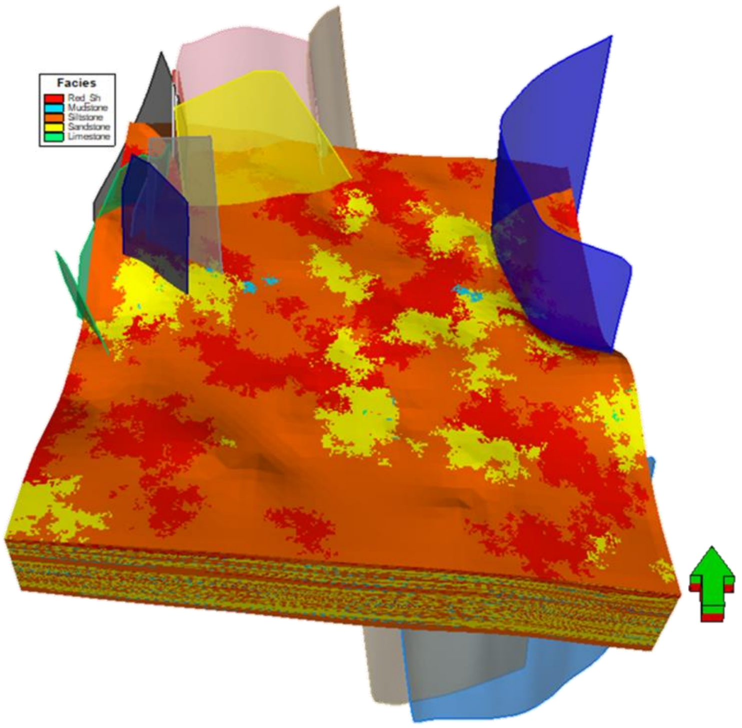


Figure 8 Full-field facies model with faults mapped from 3-D seismic. The model spans approximately 71,500 ft × 79,500 ft (~21.8 km × 24.2 km) horizontally with a total vertical thickness of 2,800 ft (~0.85 km).

Chaves created a full-field dynamic simulation model using a static model exported from Petrel (Figure 8) and imported into CMG software as rescue file format. The model retained the original grid dimensions and included three facies with calibrated relative permeability and capillary pressure curves. He adjusted these inputs using lab data and published correlations to match field conditions. Closed boundary conditions were applied using volume and transmissibility modifiers. The full-field model represents a confidential

CO_2 storage site and therefore the exact location is not disclosed. The model spans approximately 71,500 ft × 79,500 ft horizontally, with a total vertical thickness of 2,800 ft. It consists of 286 × 318 × 200 grid cells, with 250 ft horizontal and 14 ft vertical resolution. The model includes all Miocene injection flow units and the overlying sealing formation, with vertical refinement applied in the injection interval for improved simulation accuracy. The simulation included three injection wells operating over a 15-year injection period, followed by 50 years post-injection monitoring. CO_2 was injected into the Miocene formation under pressure constraints.

Additionally, sub-seismic faults with a 0° orientation were incorporated to assess their impact on plume migration and pressure distribution, with fault transmissibility evaluated as a sensitivity parameter. However, only the base case without geological or fault-related uncertainties was used for the current study, as the primary objective was to determine the detectability threshold rather than to analyze uncertainty.

2.5. Gaps in Literature

Although seismic methods remain a central component of geological carbon storage (GCS) monitoring, their practical limitations predominantly in deep or geologically complex reservoirs continue to challenge their site-specific and cost-effective application. Much of the existing literature focuses on improving the seismic toolset itself, such as refining forward modeling workflows or optimizing acquisition parameters. Commonly cited issues include signal masking in stiff formations and patchy saturation effects that reduce signal repeatability and detection accuracy. However, this body of work tends to assume that seismic will always be the default monitoring solution, rarely questioning whether it should be applied uniformly across all project areas.

Rather than advancing seismic modeling or maximizing tool sensitivity, this research takes a step back to reconsider the monitoring design process itself. It asks not *how* to monitor better everywhere, but *where* monitoring is actually needed—and *when* it provides value. Instead of deploying blanket surveys or making tool selections in isolation, this study uses ensemble-based reservoir simulations, grounded in existing geological models, to evaluate the spatial and temporal uncertainty of CO_2 plume behavior. From this, the ALPMI method of identifying when and where material impacts may occur are used to determine where monitoring should be focused.

This reservoir-model-guided monitoring approach lays the foundation for a model-informed, risk-based monitoring framework. It shifts the focus from tool-driven monitoring to risk-driven design, creating a pathway for selecting the most appropriate tools—seismic or otherwise—based on the risk profile of specific zones and timeframes. By doing so, it avoids the inefficiencies of one-size-fits-all strategies and offers a more defensible and cost-effective alternative aligned with regulatory expectations.

In summary, while previous studies have focused on improving the precision of monitoring tools or detecting minimum thresholds, this research redefines the monitoring challenge. It reverses the typical workflow starting not with the tools, but with the reservoir and builds a “model–map–monitor” framework to guide adaptive, site-specific monitoring based on actual plume migration risk, rather than technical capability alone.

Chapter III: Methodology

This chapter presents the workflow developed to assess and optimize monitoring strategies for detecting unintended and materially consequential lateral CO_2 plume migration. Building upon the static and dynamic reservoir models described in Chapter II. The methodology integrates two core components: (1) targeted risk-based monitoring strategy, and (2) seismic forward modeling for detectability assessment. The reservoir of interest is a Miocene fluvial-deltaic system located on the Texas-Louisiana Gulf Coast, characterized by significant geological complexity, including heterogeneous channel architectures, sub-seismic faults, salt domes, and legacy wells. All analyses utilize industry-standard platforms, including Petrel and CMG, with additional data preparation and processing conducted in Python and synthetic seismic generation in Madagascar. This integrated approach aims to deliver a cost-effective, new risk-based monitoring framework tailored to the challenges of complex subsurface environments.

3.1. Input Reservoir Models

The methodology presented in this chapter utilizes two primary reservoir models developed and described in prior work by (Chaves, 2024) as inputs for all subsequent analyses: a single flow-unit model and a full-field model. The single-flow-unit model is designed to probe key aspects of geological uncertainty at a spatial resolution. In contrast, the full-field model captures the broader reservoir context and dynamic behavior, which is used in the full-field synthetic seismic forward modeling. Both models incorporate static components (geological property modeling) and dynamic components (fluid flow simulations) to represent spatial heterogeneity and time-dependent plume migration accurately.

3.2. Targeted Risk-Based Monitoring Strategy

Conventionally, stochastic sensitivity analysis is used to inform monitoring design. Instead, this study generates spatial-temporal heatmaps from ensemble reservoir models to guide monitoring deployment. Rather than using model outputs for post hoc interpretation, this method actively informs monitoring decisions, integrating risk and cost within a practical framework that can be readily applied in real-world projects. This study develops a parsimonious, targeted risk-based monitoring approach using outputs from model simulations. Rather than relying on statistical summaries like tornado plots, each simulation represents a physically plausible outcome, ensuring the monitoring design reflects the full spatial and temporal uncertainty of the reservoir.

The core innovation of this study is visualizing uncertainty as spatial and temporal variations in plume behavior with heatmaps. By overlaying footprints of the 20 realizations, the heatmaps highlight "hot zones" where plume divergence is most significant. This enables a shift from assumption-based to data-driven surveillance, guiding monitoring where variability—and thus risk—is highest. The workflow integrates three components: (1) expressing model uncertainty, (2) generating heatmaps, and (3) enabling targeted monitoring that maximizes detection confidence with minimal cost.

3.2.1. Expressing Model Uncertainty

This study addresses model uncertainty by analyzing gas saturation results from single flow-unit model simulations originally developed by Chaves (2024), while the original analysis focused on both the CO_2 plume and the pressure front (Area of Review (AoR), this research examines only the CO_2 plume migration specifically, namely the spatial extent of CO_2 saturation over 200 years. In total, there were 64 unique models, as mentioned in Chapter 2, with different uncertainties tested, such as fluvial channel geometries and subsurface fault uncertainties due to orientation and transmissibility. To capture the most extreme behaviors, 20 representative (Figure 9) cases were selected, corresponding to fault transmissibility values of 0 (sealed) and 1 (fully open).

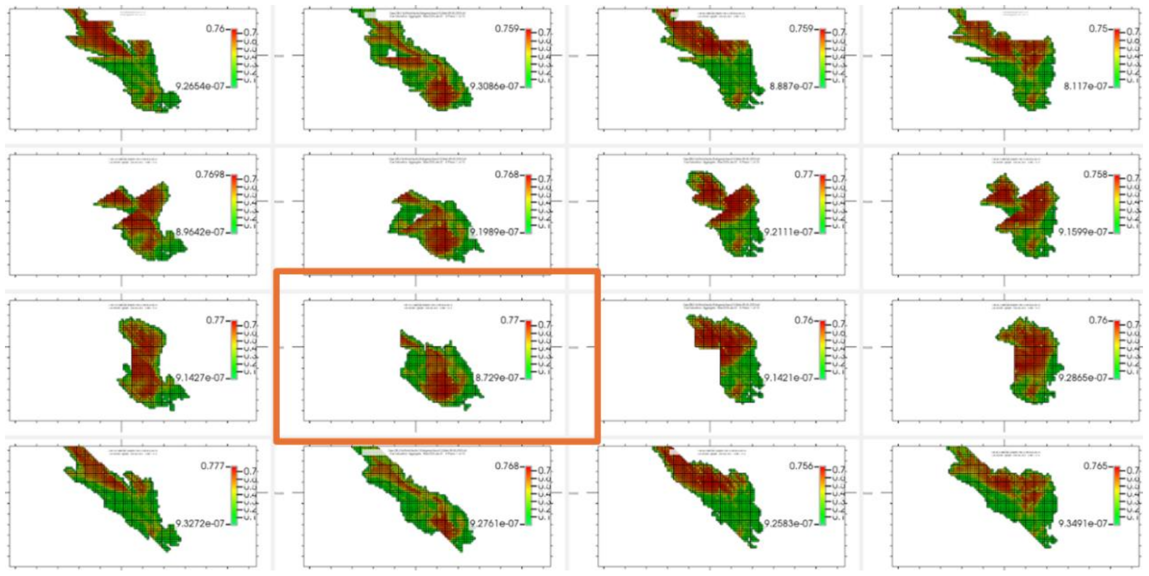


Figure 9 20 representative CO_2 realizations. The base case is highlighted with a square.

Simulation results were exported from CMG in (Simulation Input File, .sif) format (see Appendix A). The .sif format organizes data by property and time step, listing each cell's property value (such as gas saturation) in sequence, but does not include explicit spatial coordinates. This preserves site confidentiality and supports efficient parsing for analysis.

A Python workflow (see Appendix B) was developed to process these files, extracting gas saturation values for every cell and time step. The workflow reconstructs 2D grid maps by assigning each cell's gas saturation value according to its grid indices. For visualization and comparison, a "max aggregate" approach was employed: for each (I, J) grid cell, the maximum gas saturation value observed across all K layers (vertical cells) was assigned, resulting in a plan-view map of the plume's areal extent at each time step.

This approach enables the creation of gas saturation maps that closely match those produced in CMG, as shown in Figure 9, providing clear snapshots of plume growth and lateral migration for comparison and quality control.

3.2.2. Generating Heatmaps

3.2.2.1. Spatial Analysis of Plume Migration

The study employs a Python-based workflow (see Appendix B) to map and compare CO_2 plume movement across various simulation scenarios with the base case. For each case and time step, the gas saturation grid is converted into a binary map: grid cells with more than 1% CO_2 saturation are marked as 1 (indicating the presence of a plume), and the rest are marked as 0. These binary maps display the spatial footprint of the plume in a clear and consistent format.

The base case is processed in the same way to produce a binary reference map. Then, for the remaining 19 cases representing unintended lateral plume migration, the binary plume maps are aligned and stacked together. For each cell, the binary values from all cases are summed. The result is a single composite map where the value of each grid cell reflects the number of scenarios in which CO_2 was present at that location. A cell with a value of 0 indicates no plume presence in any case, while a value of 19 indicates that all cases resulted in plume presence at that location. This stacking procedure reveals the variability in plume migration across the ensemble of model realizations.

To identify deviations from the base case, this stacked heatmap is compared to the base case plume map by subtraction. The difference highlights areas where plume migration in the ensemble diverges from the base case, which are referred to as mismatch zones. However, simply subtracting the base case plume map from this stack produces a mismatch map that reflects both under- and over-prediction relative to the base case.

Crucially, this “mismatch” includes areas that were part of the base plume and are therefore not necessarily unexpected. To isolate the *additional migration* not seen in the base case, an “additions only” map is computed by identifying cells where CO_2 is present in the ensemble stack but absent in the base case. This reveals the plume expansion attributable to uncertainty, distinguishing risk-prone regions where plume behavior deviates from the base case.

This two-step binary approach (1) stacking ensemble plumes and (2) subtracting the base provides a more meaningful and spatially resolved comparison. It moves beyond basic anomaly detection and offers direct insight into where monitoring effort should be focused, relative to modeled expectations. Ultimately, this lays the foundation for a model-informed monitoring design.

Finally, the stacked values are visualized as a heatmap: “hot zones” corresponds to grid cells where the plume occurs in many cases, indicating a high probability of unintended CO_2 migration. These persistent areas become monitoring priorities. This layered approach, binary conversion, ensemble stacking, base comparison, and heatmap visualization, offers a targeted, risk-informed basis for model-guided monitoring design.

3.2.2.2. Temporal Analysis of Plume Migration

Based on spatial observations that plumes can migrate preferentially in various directions depending on geological and fault conditions that may not be defined deterministically but can be provided probabilistically, a temporal analysis can be conducted to evaluate plume migration dynamics over time along the preferential direction. A Python workflow (see Appendix B) was developed to track the maximum distance in grid cells that the CO_2 plume extends from the injection well for each simulation case each year. For every grid cell exceeding the detection threshold (gas saturation > 0.01), the

Euclidean (straight-line) distance from the well location was calculated using the square grid indices (I, J). Migration distances are reported in the number of grid cells, providing a flexible framework for relative comparison across scenarios; conversion to physical distance is straightforward by multiplying by the actual grid cell size. This study presents results in grid cell units, focusing on migration patterns and trends, with the option to convert to real-world distances in future analyses.

The Euclidean distance:

$$\text{Distance} = \sqrt{(I - I_{\text{well}})^2 + (J - J_{\text{well}})^2}$$

The farthest plume extent from the well was recorded for each simulation year, generating a time series of maximum migration distances. Scenarios were classified as “hot” or “cold” based on their maximum plume extent in the final simulation year: cases with a migration distance greater than 40 grid cells were designated as “hot,” while those below this threshold were considered “cold”. This threshold is determined based on observation. This classification and the base case were visualized using time-evolution plots, allowing for a direct comparison of plume migration trends, variability, and outlier behaviors across all realizations. This temporal analysis provides a robust tool for identifying riskier migration scenarios and informs the development of targeted monitoring strategies throughout the operational and post-injection phases of the project.

3.2.3. Enabling Targeted Risk-Based Monitoring

By integrating both spatial and temporal analyses of plume migration, this workflow enables the development of a truly targeted, risk-based monitoring strategy. The spatial heatmaps identify specific locations where the CO_2 plume is most likely to migrate or where the highest variability is observed across multiple scenarios, effectively highlighting persistent “hot zones” that warrant close surveillance. Temporal analysis complements this by revealing when plume migration is most active, allowing monitoring efforts to be concentrated not only in space but also during key time windows. Together, these insights eliminate guesswork and allow monitoring resources to be deployed precisely where and when they are most needed. This approach ensures that the monitoring program is both scientifically robust and cost-effective, directing investment to areas of highest risk, maximizing early detection of anomalous migration, and fulfilling regulatory and operational requirements with maximum efficiency. Ultimately, this risk-based methodology transforms monitoring from a broad, assumption-driven exercise into a focused, data-informed process tailored to the actual behavior of the subsurface system.

3.3. Seismic Forward Modeling and Detectability Analysis

The second part involves conducting seismic forward modeling and detectability analysis on the full-field model. With the full-field reservoir model established, the next stage was to systematically prepare input data for seismic forward modeling to evaluate the detectability of CO_2 plume migration. For this purpose, the full-field reservoir model served as the basis for all subsequent geophysical analysis.

The static reservoir facies model was discretized on a grid of 288 × 314 × 200 cells (totaling over 18 million cells), with an average vertical resolution of 10 feet and horizontal resolution of 250 feet exported from Petrel. Additionally, relevant well log data, specifically compressional and shear wave velocities, and bulk density, were compiled to provide the necessary elastic property inputs for the seismic modeling workflow. Time-lapse gas saturation outputs from dynamic reservoir simulations (spanning 15 years of injection and 50 years of post-injection, for a total of 65 years) were also exported as properties from CMG.

Both datasets were converted to Geostatistical Software Library (GSLIB) format (see Appendix C) to ensure compatibility with the Petrel and Madagascar software. The GSLIB format was chosen because it retains location information in the X, Y, and Z directions, which is crucial for accurate seismic modeling in Madagascar and analysis in coordinate-specific software, such as Petrel, as well as for ease of data transfer between software. The prepared datasets were then passed to collaborators Rebecca Gao and Dr. Sergey Fomel for seismic forward modeling. Most data preparation and transformation steps were conducted in Python (see Appendix D (Gao et al., Unpublished)), including filtering non-physical values, normalizing data ranges, and structuring all input variables for integration into the seismic modeling workflow.

The first significant step in the seismic workflow was the regularization of the original non-uniform grid, which involved interpolating missing data and populating elastic property parameters (see Figure 10). After regularization, the grid was refined to 288 × 314 × 420 cells (nearly 38 million cells), with the vertical resolution (Zinc) improved to 5 feet, while the X and Y dimensions remained unchanged. Variogram analysis, using available field log data, supported the spatial modeling of P-wave velocity, S-wave velocity, and bulk density, ensuring the resulting elastic property cubes reflected realistic geological trends.

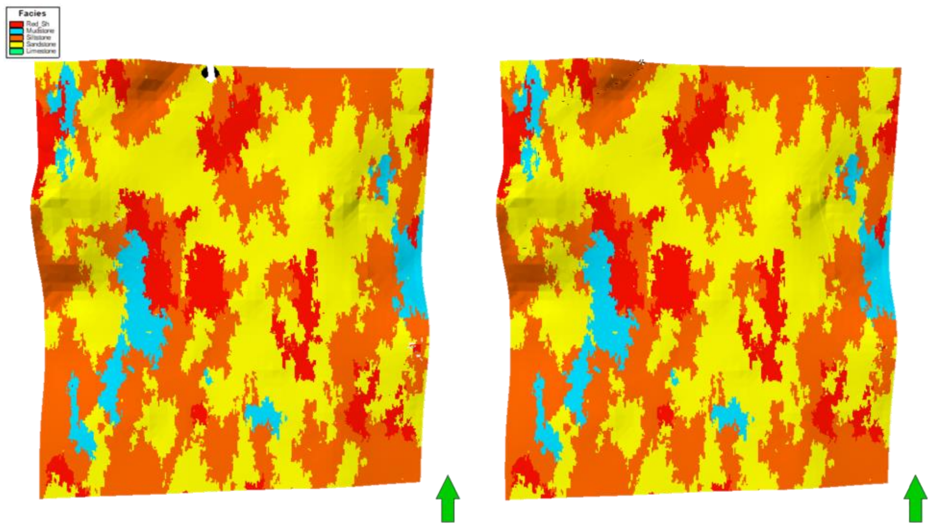


Figure 10 Facies model original grid 250 ft × 250 ft × 14 ft (left), regularized new grid (right) 250 ft × 250 ft × 5 ft.

The original facies model from Chaves (2024), shown on the left in Figure 10, was constructed on a non-uniform grid with numerous missing or null cells, resulting in gaps and discontinuities that hindered subsequent seismic forward modeling. The model was regularized and refined vertically to overcome these limitations by interpolating missing values and resampling the property data onto a uniform, high-resolution grid. This process, illustrated on the right in Figure 10, produced a continuous and fully populated facies model that preserves key geological features while ensuring compatibility with geophysical simulation workflows.

To simulate the seismic response of CO_2 injection, the Gassmann fluid substitution method was applied in Python (see Appendix D (Gao et al., Unpublished)). This approach estimates changes in elastic properties, most importantly, compressional wave velocity and bulk density, resulting from CO_2 replacing brine within the reservoir's pore space. Accurately capturing these changes is crucial for generating realistic synthetic seismic data that accurately represents evolving reservoir conditions. (Li et al., 2024; Smith et al., 2003).

Using these property cubes and time-dependent saturation data, full-stack synthetic seismic volumes were generated at a dominant frequency of 28 Hz using Madagascar (see Appendix E (Fomel, 2024; Fomel et al., 2013; Gao et al., Unpublished)). The modeling process included facies-to-property mapping, application of fluid substitution, calculation of acoustic impedance, depth-to-time conversion, and convolution with a representative seismic wavelet, resulting in synthetic seismic images in depth Figure 11.

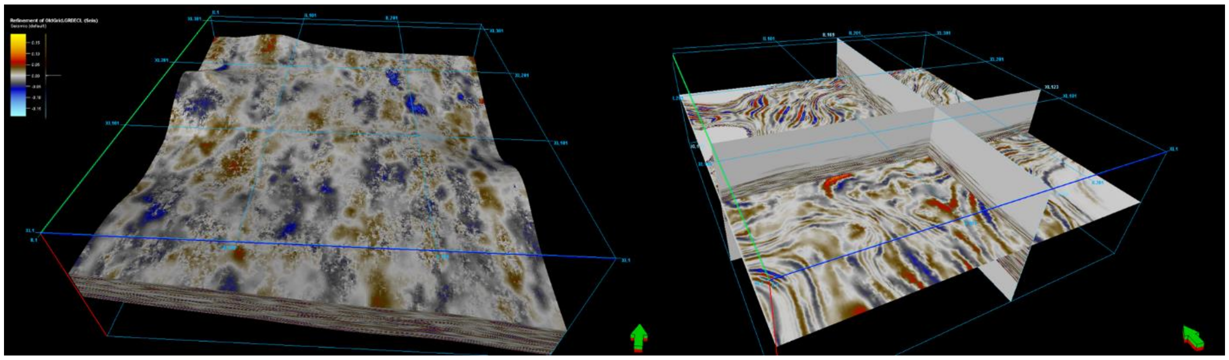


Figure 11 Synthetic Seismic cube visualization in Petrel (Collaborators: Rebecca Gao and Dr Sergey Fomel) The cube spans approximately 72,000 ft × 78,500 ft (~21.95 km × 23.93 km) horizontally and 2,100 ft (~0.64 km) vertically, at a resolution of 250 ft × 250 ft × 5 ft.

Once the synthetic seismic data were generated, the files were formatted and exported in GSLIB format and subsequently imported back into Petrel for amplitude-based detectability analysis. This final stage enabled quantitative evaluation of the seismic response to varying CO_2 saturation and the identification of plume detection thresholds, establishing the minimum conditions required for effective seismic monitoring in this geological setting.

In summary, this chapter presents a two-part methodological framework that combines spatial-temporal risk analysis and seismic forward modeling to inform the design of CO_2 plume monitoring. By leveraging ensemble reservoir simulations, heatmap-based uncertainty mapping, and synthetic seismic generation, the study presents a technically robust and cost-effective approach to developing monitoring strategies. The following chapter builds on this foundation to evaluate the detectability and monitoring performance across different scenarios, ultimately guiding more effective subsurface surveillance

.

Chapter IV: Results & Analysis

This chapter presents the results derived from the simulation workflows introduced in Chapter III, grounded in the geological context and model setup outlined in Chapter II. The results are organized according to the study's two-pronged methodology: (1) risk-based spatial and temporal analysis of CO_2 plume migration, and (2) seismic forward modeling for detectability assessment. Together, these findings support the development of a cost-effective, model-informed monitoring strategy for geologic carbon storage in complex subsurface environments.

4.1. INPUT RESERVOIR MODELS

All results in this chapter are derived from the two reservoir models previously described the single flow-unit model and the full-field model. While Chaves (2024) evaluated both the CO_2 plume and the pressure front to inform the Area of Review (AoR), the present analysis focuses solely on the plume to support detectability and monitoring design.

A single 200 ft thick flow-unit model was used to evaluate geological uncertainty across 64 realizations, varying fluvial channel architecture, fault orientation, and fault transmissibility over a 200-year simulation. As discussed in Chapter II, prior results demonstrated relatively minor variation in plume extent, suggesting robust plume containment under a wide range of conditions.

In contrast, the full-field model (2,800 ft thick, covering an area of approximately 71,500 ft × 79,500 ft, with three injection wells and large-scale faults represented by simulation faults) was used to generate a baseline plume saturation map (Figure 12) for seismic forward modeling. Sub-seismic faults uncertainty were omitted to isolate the detectability of the plume under conservative conditions, justified by earlier findings showing limited impact of sub-seismic faults on plume shape and extent (Chaves, 2024).

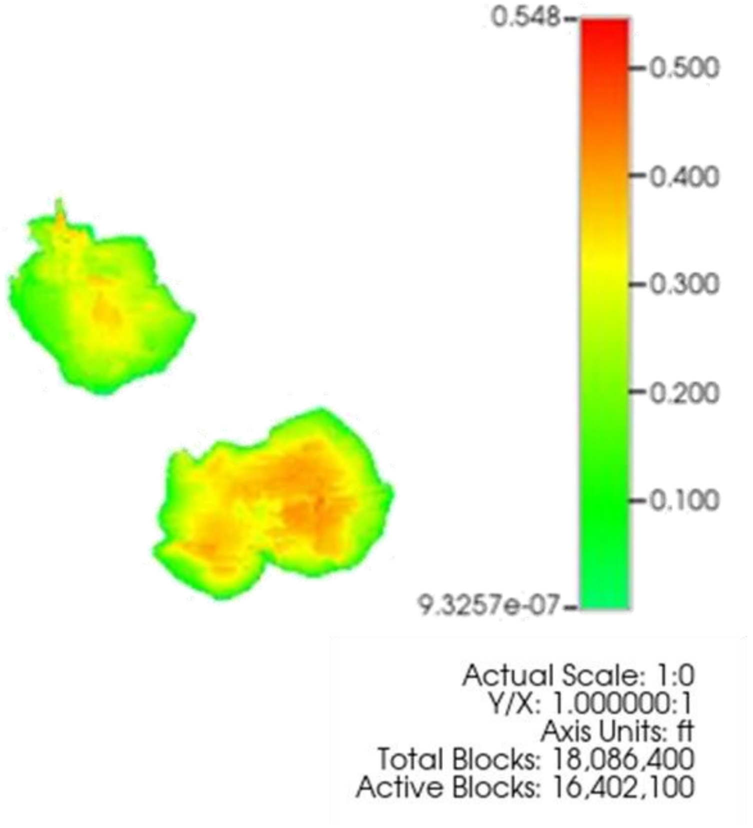


Figure 12 Full-field model (base case) CO_2 plume saturation clipped to extent of the plume only.

Although the full-field model was originally developed by Chaves (2024) for Area of Review (AoR) analysis, this study repurposes the same model to evaluate CO_2 plume detectability. The base case gas saturation map (Figure 12) was extracted directly from the original full-field simulation but is interpreted here through a different lens focusing on plume shape and detectability rather than pressure footprint. The rounded plume geometry observed in this base case served as a foundational reference for both the uncertainty analysis presented and the seismic detectability assessment in Chapter III.

4.2. TARGETED RISK-BASED MONITORING STRATEGY

This section presents a targeted monitoring strategy derived from spatial and temporal analyses of CO_2 plume migration under geological uncertainty. Departing from traditional sensitivity plots like tornado diagrams, this study adopts a spatially resolved, ensemble-based approach using outputs from the single flow-unit model. The smaller grid size of this model allowed for efficient simulation across 64 scenarios, capturing a wide range of uncertainty in fluvial architecture and sub-seismic fault behavior. As established by Chaves (2024), these uncertainties have minimal effect on AoR determination, but their influence on plume shape remains critical for monitoring design.

4.2.1. Spatial Analysis

Heatmaps were generated from 20 representative cases of the single flow unit model to visualize where and when to monitor CO_2 migration, providing spatial and temporal insights. These maps were produced by converting each case's gas saturation output into binary plume footprints using a threshold of 0.01. At each time step, these binary grids were aggregated across scenarios to generate stacked plume maps and heatmaps.

Figure 13 presents the results over five-time steps—Years 5, 10, 15, 30, and 200—across three rows:

i. The top row shows the evolution of CO_2 gas saturation of the base case plume.
ii. The middle row displays the stacked CO_2 gas saturation plume footprints from the 19 failure cases.

iii. The bottom row shows heatmaps (number of cases) of the differences between the stack and the base case.

In the top row, the plume expands steadily outward from the injection point (marked with a cross), growing in size and gas saturation over time. Its geometry remains relatively symmetric and rounded through Year 200, consistent with stable containment behavior which exhibit similar shape to the base case plume front in Figure 12. Most CO_2 gas saturation accumulated around the injection well with hot red and yellow colors, while the edges have lower concentrations of CO_2 gas saturation in green.

The middle row shows the maximum aggregated CO_2 gas saturation of the composite of the 19 failure cases. At early time steps (Years 5 and 10), the plume footprints are compact and similar to the base case. By Year 15, however, plume asymmetry emerges, and by Years 30 and 200, the plume becomes more elongated and biased toward the northwest. These footprints are visibly larger and more irregular, indicating divergence among the failure cases.

The bottom row presents the heatmap of deviations between the base case and the ensemble. The color scale reflects the number of scenarios in which CO_2 is present at each location. White indicates both the base case and no case presence; yellow to red shows increasing overlap. Early in the simulation, differences are minor and concentrated near the injection well. Over time, the plume spread becomes broader, with higher concentrations in the northwest quadrant, highlighting where the ensemble deviates most from the base case.

This sequence of images captures the temporal evolution of CO_2 plume behavior across scenarios, including where divergence begins and how it intensifies. Areas with consistently high overlap across cases (i.e., red/orange regions) represent persistent migration paths, while regions of low overlap indicate greater uncertainty. These heatmaps help identify potential monitoring zones based on where plume differences are most consistently observed.

In summary, the spatial analysis reveals a clear temporal trend in plume expansion and variability across scenarios. The base case shows gradual, symmetric growth, while the ensemble cases exhibit increasing lateral spread and geometric asymmetry over time. The stacked plume maps demonstrate a consistent northwest bias in plume migration, especially in later years. The deviation heatmaps highlight specific regions where plume presence differs most frequently from the base case, with differences becoming more pronounced after Year 15. These spatial patterns form the observational basis for identifying priority monitoring zones in areas of highest scenario overlap and plume variability.

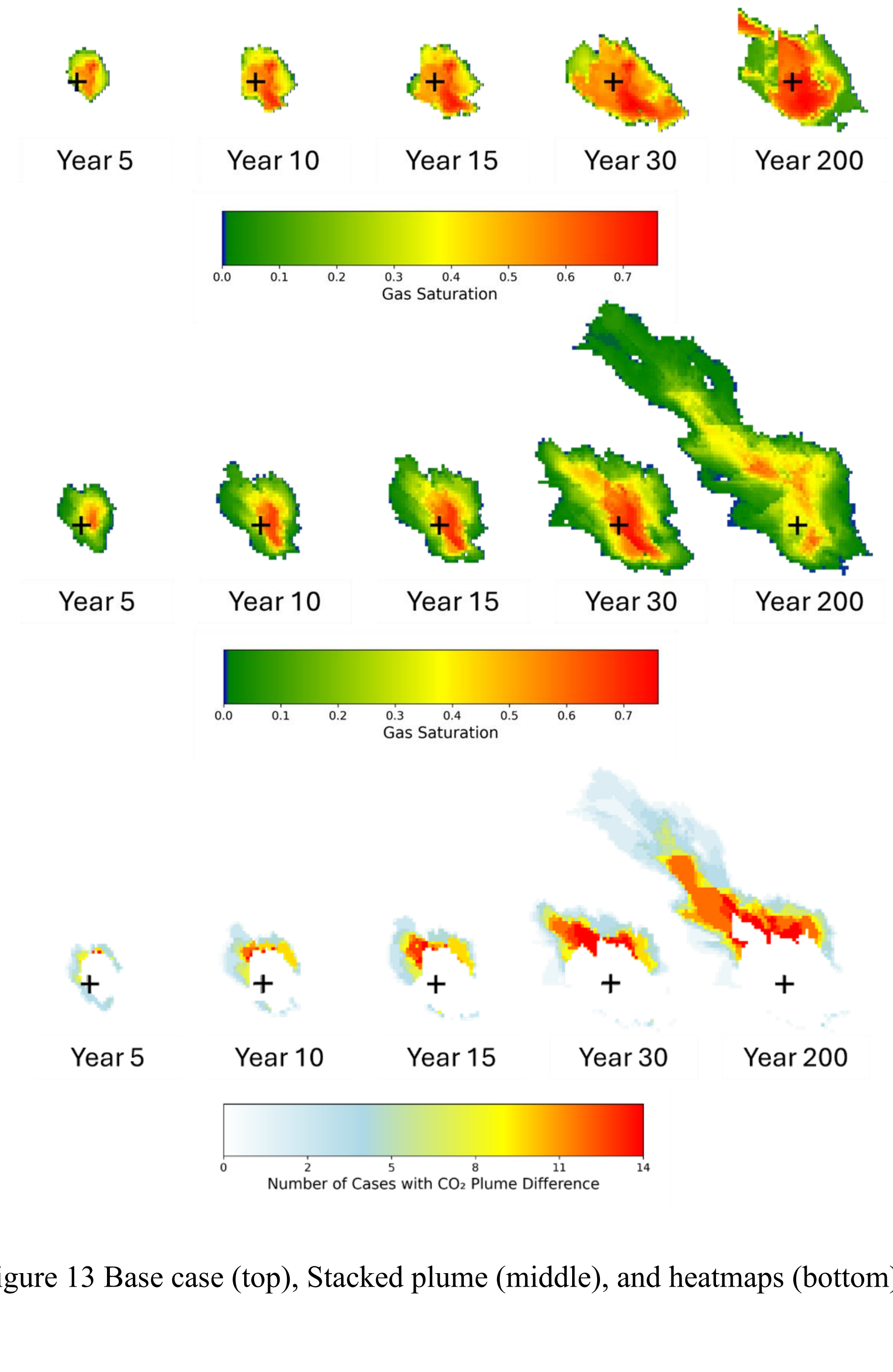


Figure 13 Base case (top), Stacked plume (middle), and heatmaps (bottom).

4.2.2. Temporal Analysis

Temporal analysis was conducted by tracking the maximum CO_2 plume migration distance from the injection well over time for all 20 scenarios, using the dominant migration path observed, which in this case trends updip toward the northwest. Migration distance was measured in grid cells, and results were plotted across the entire 200-year simulation period (Figure 14). By plotting migration distance versus time, two distinct clusters are observed: one group behaves similarly to the base case (black dotted line), and the other group migrates farther. For purposes of our study, this long migration path defined as unacceptable.

Figure 14 presents the full 200-year evolution of migration distances for all cases. The base case (black dashed line) demonstrates a steady and predictable plume migration pattern over the 200-year simulation period. During the early phase (0–10 years), the plume expands gradually, reaching approximately 10 grid cells, indicating stable containment. Between Years 10 and 50, migration continues at a moderate pace, ultimately plateauing just below 30 grid cells by the end of the simulation. This behavior reflects symmetrical plume growth. The consistent and bounded nature of the base case serves as a reference for acceptable plume migration, against which more aggressive or erratic behaviors in the failure scenarios can be evaluated.

In the early years (0–10), all scenarios show nearly identical behavior—the plume expands gradually and symmetrically, and the migration distance remains within a narrow range. This tight clustering reflects low initial uncertainty and predictable plume behavior during the injection phase.

After Year 25, the contrast between the two clusters sharpens. One group colored in cooler shades (blues) plateaus under 30 grid cells, showing good agreement with the base case. The other group colored in warmer hues (yellows to reds) migrates well beyond

40 grid cells, indicating larger lateral plume spread. The threshold 40 grid cells are based on interpreter observation. This divergence continues throughout the post-injection phase, with some scenarios reaching over 80 grid cells by Year 200.

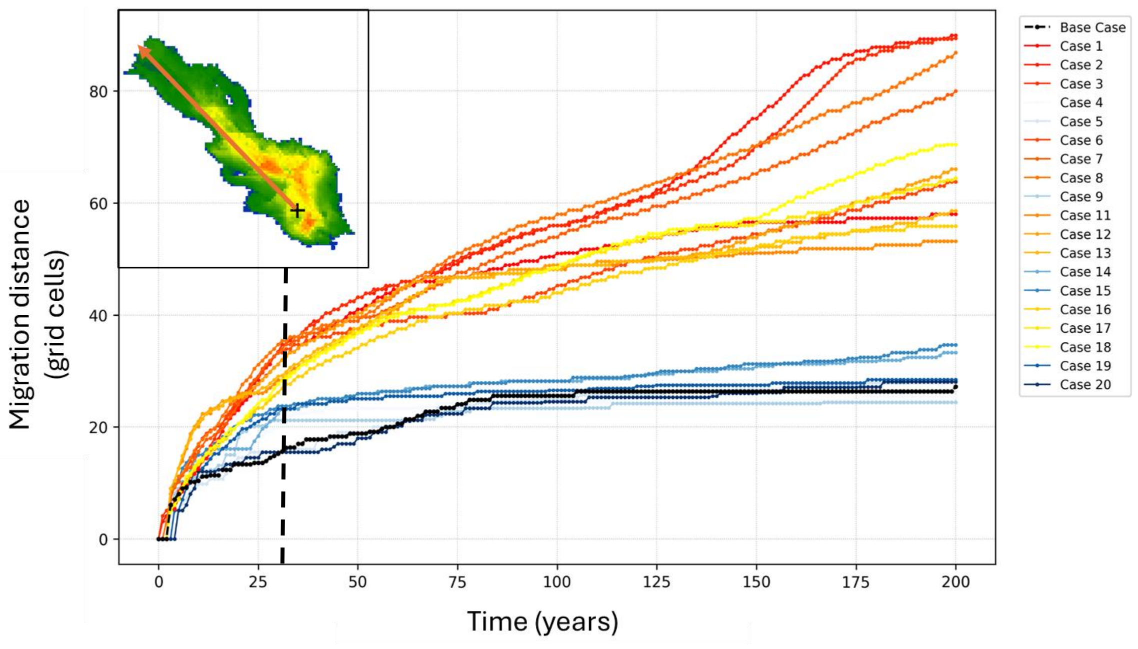


Figure 14 Migration distance versus time for 200 years. Dashed line indicates end of injection at 30 years.

Figure 15 zooms in on the first 50 years of plume migration to highlight the onset of divergence in finer detail. During the initial 5 years, migration distances across all scenarios are nearly indistinguishable. At Year 15, early signs of divergence begin to appear; however, the spread is still modest and many of the scenarios remain closely clustered around the base case. This suggests that within the first 15 years, there is still limited diagnostic value or overlaps in differentiating acceptable from unacceptable behavior. By Year 25, the onset of separation becomes more distinct. The two clusters one closely tracking the base case (cooler lines) and the other drifting upward (warmer lines) are now visibly differentiated. Nevertheless, the absolute difference in migration distance between these groups remains small at this point, with just a few grid cells separating them.

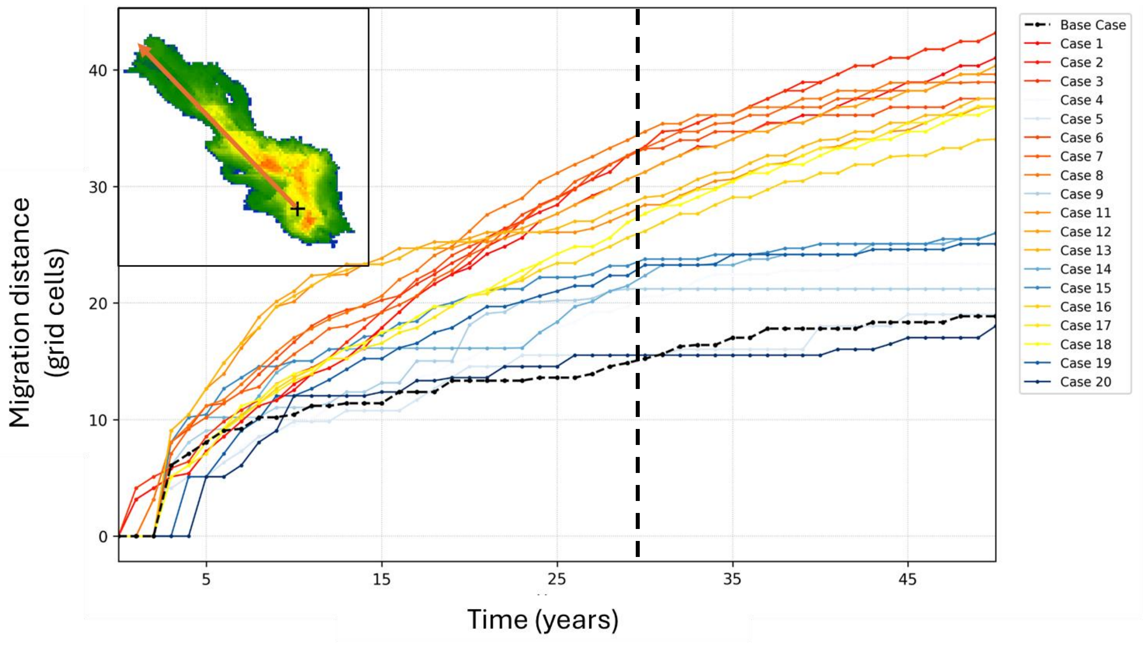


Figure 15 Migration distance versus time for 50 years. Dashed line indicates end of injection at 30 years.

This temporal pattern provides critical operational insight: the effective monitoring window opens as early as Year 15, when early detection of anomalous migration becomes possible. Early warning within this window allows for timely adjustment of injection strategies or implementation of mitigation measures before unacceptable migration occurs. At the same time, deferring intensive monitoring efforts until after Year 5 avoids unnecessary costs during a period of low diagnostic value. These findings support the design of a cost-effective, risk-informed monitoring strategy that is responsive to the evolving behavior of the plume.

The temporal analysis reveals a clear divergence in plume migration behavior over time, with two distinct clusters emerging from the ensemble of 20 scenarios. Initially, all cases exhibit nearly identical plume migration distances, indicating consistent early-time behavior and limited diagnostic value before Year 15. However, by Year 25, the ensemble begins to separate into two behavioral groups; one that remains closely aligned with the base case, plateauing at under 30 grid cells and another that steadily diverges, exceeding 40 grid cells and continuing to grow. This divergence becomes increasingly pronounced in the post-injection phase (after Year 30), with some failure scenarios surpassing 80 grid cells by Year 200.

4.3. Synthetic Seismic Detectability Analysis

A 4D time-lapse seismic model was developed by integrating the static geological framework with dynamic fluid saturation changes from the reservoir model. Instead of using a single-parameter input, the model was populated with elastic properties; P-wave velocity (Vp), S-wave velocity (Vs), and bulk density (ρ) were derived from well log analysis. Time-dependent saturation data from the dynamic model simulated evolving elastic properties. Gassmann fluid substitution was applied and synthetic seismic volumes generated at multiple time steps, forming the basis for detectability assessment.

Figure 16 presents two panels showing simulated CO_2 gas saturation from the dynamic reservoir model at two time points: after five years of injection (left) and after an additional thirty years of post-injection (right). In the left panel, three separate plumes are visible around the three injection wells, each with a distinct core of higher gas saturation (red and yellow) surrounded by lower-saturation regions (green and blue), and all remain relatively small at this early stage. In the right panel, the plumes have grown in size, with some merging to form larger, more elongated zones of elevated gas saturation; the highest saturation cores remain, while the lower-saturation regions have expanded, illustrating further migration and spreading of CO_2 over time. The color scale represents gas saturation values. These snapshots capture the temporal evolution of the plume and serve as dynamic inputs for seismic forward modeling, supporting the generation of synthetic time-lapse seismic volumes for each simulation year.

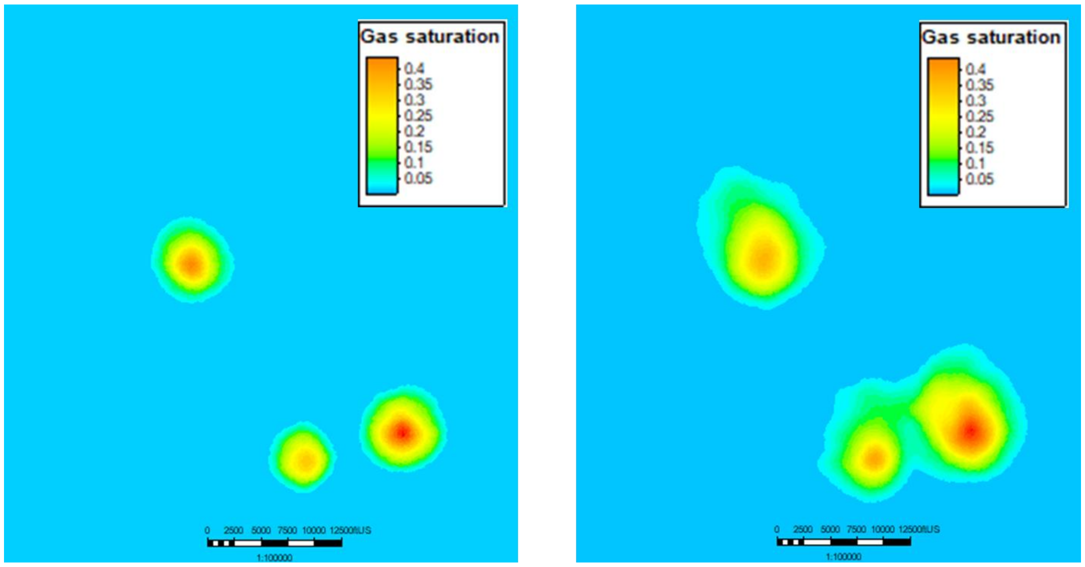


Figure 16 Dynamic simulation of CO_2 plume saturation five years after injection (left), 30 years post-injection (right).

To assess detectability, defined here as the ability to distinguish the presence or absence of CO_2 in the subsurface using seismic data the first step in this process involves generating a baseline synthetic seismic volume at year zero before injection using the static reservoir model with no CO_2 present. Subsequently, synthetic seismic volumes can be generated for each simulation year using saturation from fluid flow modeling, elastic properties from well log analysis, and Gasmann fluid substitution. By subtracting each time-lapse seismic volume from the baseline, amplitude difference volumes are calculated, highlighting changes in the seismic response due to CO_2 saturation.

Figure 17 shows synthetic seismic amplitude difference maps for the same model and time intervals as Figure 16, illustrating how CO_2 plume evolution is expressed in the seismic response. In both panels, amplitude differences are presented relative to the baseline, with the color scale ranging from blue (negative amplitude changes) through white to yellow (positive amplitude changes). The left panel corresponds to five years of injection, where amplitude anomalies are concentrated around three separate zones, each matching the locations of the high-saturation CO_2 plumes. The right panel shows results after an additional thirty years of post-injection, where the amplitude anomalies have grown in both magnitude and spatial extent; some have merged, producing larger, more continuous features. These synthetic seismic maps visually capture how the changing CO_2 saturation within the reservoir alters the seismic amplitude response over time.

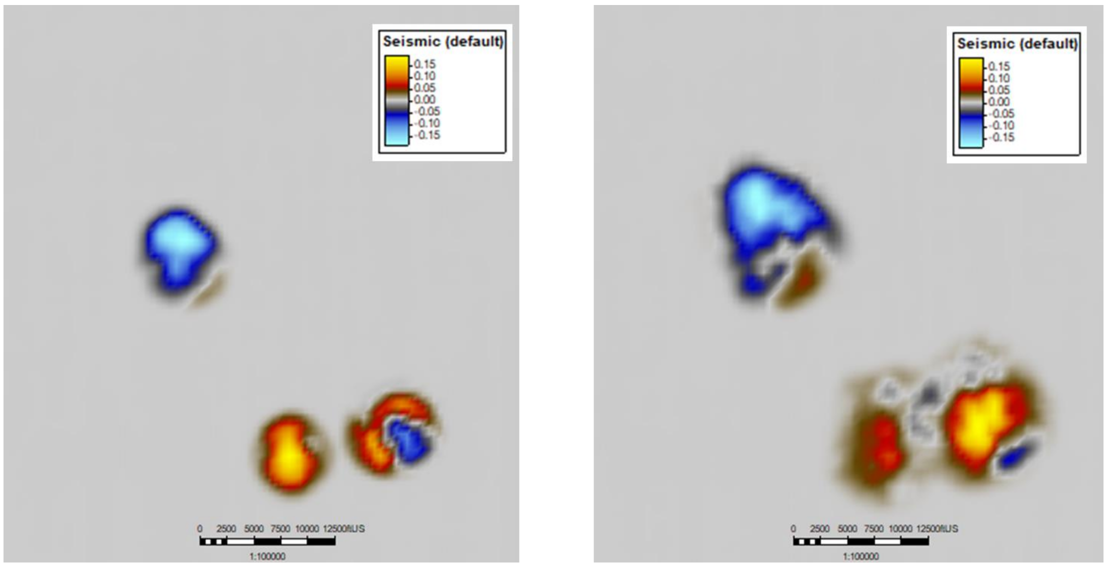


Figure 17 Seismic amplitude difference between the baseline (pre-injection) and two time-lapse snapshots: after five years of injection (left) and thirty years post-injection (right).

These changes are clearly reflected in the amplitude variations, confirming the sensitivity of the synthetic seismic response to plume evolution over time. Based on this, the detection of the CO_2 plume front can be evaluated, as shown in Figure 18, where the red contour represents the actual plume front derived from the fluid flow model. The yellow contour indicates the seismic amplitude anomaly limit, which is consistently smaller than the true plume extent. This difference defines the seismic detectability threshold.

Figure 18 displays a synthetic seismic amplitude difference map at a selected time, illustrating both the CO_2 plume front and the seismic detectability limit. Seismic amplitude represents changes relative to the baseline. The red contour outlines the true CO_2 plume front as derived from the reservoir simulation, while the yellow contour marks the seismic detectability limit based on amplitude response. This figure enables a direct visual comparison between the simulated plume extent and the area detectable using synthetic seismic. The analysis shows that the seismic response is unable to detect CO_2 saturation levels below 5% in this case. Therefore, the seismic detectability limit for this geological setting is defined as a minimum of 5% CO_2 saturation.

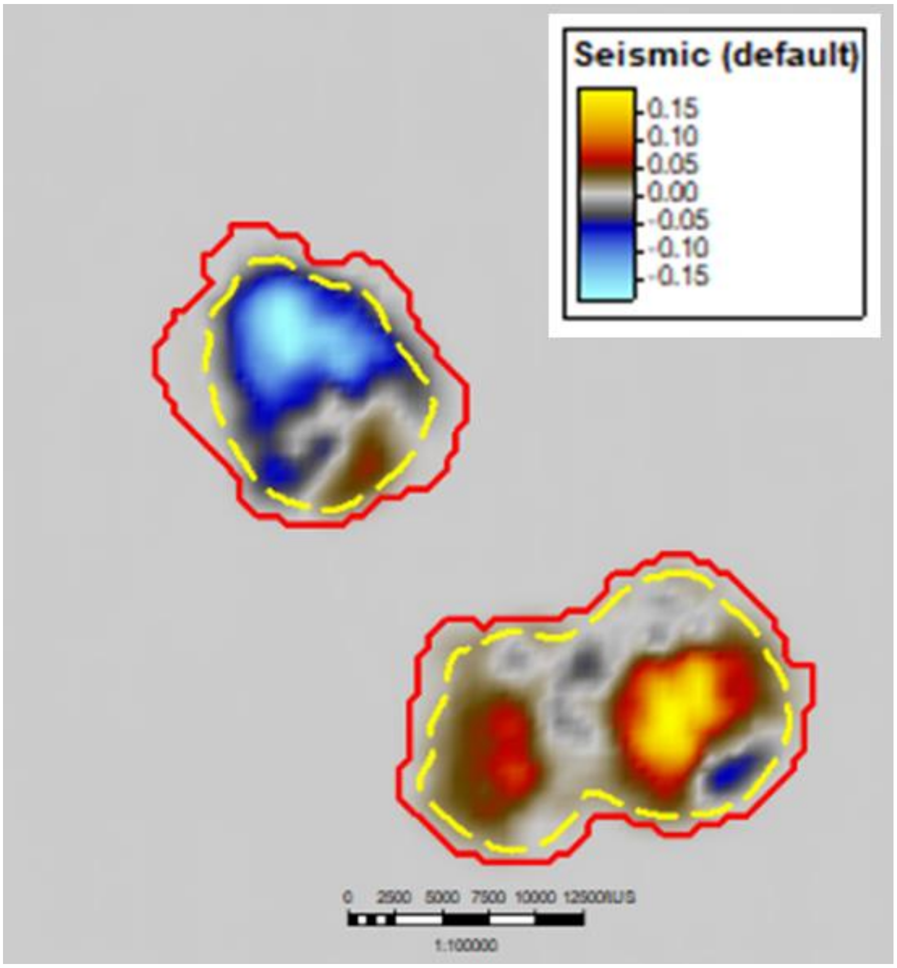


Figure 18 The red contour shows the CO_2 plume front; the yellow contour indicates the seismic detectability limit based on amplitude response. Analysis on travel times may offer higher sensitivity on the limit (Barnett et al., 2025). Scale is in feet.

Chapter V: Discussion

The management and verification of CO_2 plume containment remains a central challenge for carbon capture and storage (CCS) projects, particularly as regulatory expectations shift toward risk-based, site-specific monitoring. The U.S. EPA and international agencies increasingly emphasize that monitoring programs must be scientifically justified, cost-effective, and tailored to the specific risks of each site. This chapter discusses the implications of the results presented in Chapter IV, organized around three key themes: (1) spatial and temporal risk zones, (2) seismic detectability and monitoring limitations, and (3) cost and practical considerations. Each theme is discussed in relation to current literature, regulatory frameworks, and study findings.

5.1. Spatial and Temporal Risk Zones

Effective monitoring of CO_2 plume migration requires not only a robust understanding of plume behavior, but also an appreciation of the spatial and temporal uncertainty introduced by subsurface complexity. The Texas-Louisiana Gulf Coast, like many onshore U.S. storage settings, presents a geologically challenging environment: heterogeneous fluvial-deltaic stratigraphy, sub-seismic faulting, introduce uncertainty in how injected CO_2 migrates over time, particularly as plumes tend to remain symmetrical early on but become elongated during and after the post-injection phase. The presence of legacy wells does not directly impact plume shape but increases the importance of getting plume migration predictions right to avoid well encounters. These factors introduce directional variability in both lateral and vertical plume movement, complicating the design of reliable monitoring programs.

This study addresses that challenge through a "model–map–monitor" framework that operationalizes subsurface uncertainty into practical monitoring guidance. Building upon the full-field and uncertainty-rich single flow-unit models established by Chaves (2024), this research explicitly simulates uncertainty using a 20-member ensemble that captures end-member variations in channel architecture and sub-seismic fault orientation and transmissibility. The model outputs simulation over 200 years (30 years of injection followed by 170 years of post-injection migration) forms the foundation for both spatial heatmaps and temporal trend analysis.

The "map" stage of the workflow translates raw saturation data into high-resolution risk surfaces. Binary plume presence maps are generated for each realization and time step, then stacked to reveal the frequency with which CO_2 reaches specific locations. The resulting heatmaps (see Figure 13, Chapter IV) visualize spatial "hot zones" where CO_2 is most likely to appear across scenarios. In the base case, plume growth is relatively symmetrical and remains centered on the injection well. However, ensemble analysis reveals a different picture: as early as Year 15 of the 30 year injection period , the plume begins to show directional bias most notably northwest, corresponding with higher transmissibility zones and geologic pathways dictated by local heterogeneity and the influence of small dip. This migration pathway is preferential pathway for CO_2 plume to extent.

By Year 200, this divergence becomes stark. While some scenarios remain near the base case footprint, others exhibit significant lateral spread sometimes doubling the migration distance. These differences arise from subtle but critical variations in the geologic model, particularly the behavior of sub-seismic faults and channel connectivity. Importantly, areas with consistent plume overlap across scenarios highlighted in red/orange in the heatmaps mark the highest-risk zones, where CO_2 presence is not only likely but recurrent because the cases are stacked. In the single flow-unit model, the worst-case scenario plume migration reaches up to 80 grid cells, equivalent to 40,000 ft (~12.2

km). A monitoring array (e.g. 2D seismic line) of at least 10–12 km, oriented along the dominant plume migration direction, would be required to effectively cover the potential spread. These zones are the optimal targets for cost-effective, spatially focused monitoring strategies such as repeat 2D seismic or “spot” methods.

Complementing this spatial picture is a temporal analysis of maximum plume extent, which tracks the Euclidean migration distance of the CO_2 front over the entire 200-year period (Figure 14, Figure 15, Chapter IV). The results show that plume behavior is nearly identical across scenarios during the first 5–10 years, reflecting strong early containment and limited value in deploying intensive monitoring infrastructure during this phase. However, starting around Year 15, deviations begin to emerge but with multiple overlaps. By Year 25, two distinct behavioral clusters are visible: one tracks the base case closely, while the other migrates beyond 40 grid cells an operational threshold indicating unacceptable lateral movement. Although the gap between these clusters are small, but clear distinction between the cluster can be observed.

This subtle but critical window between Year 15 and Year 25 defines the earliest moment when adaptive monitoring becomes essential. Monitoring too early wastes resources, as plume behavior is still predictable. Monitoring too late risks missing the onset of migration that could exceed regulatory or operational containment boundaries. These findings support a time-phased surveillance strategy: low-intensity baseline monitoring during early injection, followed by intensified surveillance in key directions and timeframes as plume divergence emerges. Moreover, the good news is that the injection period is 30 years, and being able to differentiate acceptable from unacceptable CO_2 plume migration behavior as early as 15 years gives the operator enough time to adjust the injection strategy. Even better if another monitoring parameter, like pressure, can be incorporated.

Together, the spatial and temporal analyses affirm that monitoring strategies must be both site-specific and dynamically informed by multiple probabilistic model outputs that bound the site uncertainties. Rather than applying a uniform, one-size-fits-all approach, this study demonstrates that surveillance efforts should be concentrated in zones and time periods where unacceptable outlier migration responses can be separated from compliant and acceptable responses. This paradigm shift moves beyond static coverage or tool-based targeting to a probabilistic monitoring strategy—using ensembles not just to validate a single outcome, but to test whether unacceptable scenarios are emerging and require early detection.

Although these results are specific to the Gulf Coast site studied here, the workflow itself is transferable. Any CCS project with sufficient static and dynamic modeling data can apply this "model–map–monitor" methodology to identify priority monitoring zones, optimize technology selection, and minimize cost all while maintaining regulatory defensibility and public trust.

5.2. Seismic Detectability and Monitoring Limits

A persistent challenge in CO_2 plume monitoring is the inherent limitation of seismic physics: detection is not guaranteed by the presence of CO_2 alone but rather depends on whether changes in subsurface properties produce a measurable seismic response. Seismic imaging is widely adopted for its spatial coverage and ability to track plume evolution, yet its effectiveness is constrained by detectability thresholds, typically requiring at least a 4% change in acoustic impedance or nRMS below 0.4 (see Chapter II). These thresholds are highly sensitive to subsurface conditions, including reservoir depth, formation stiffness, saturation distribution and operation acquisition parameters.

This study addresses these limitations through seismic forward modeling, using the full-field reservoir model to simulate gas saturation changes over time (see Chapter IV, Figure 16, Figure 17, Figure 18). The resulting synthetic amplitude difference maps allow comparison between the true modeled plume front and the seismically visible anomaly. A consistent finding is that the seismic response underestimates the plume extent: the yellow amplitude anomaly never fully reaches the red contour representing the actual CO_2 front. This is not a modeling error, but a physical constraint on detectability. In this case, the seismic detection limit corresponds to approximately 5% CO_2 saturation, leaving lower-saturation margins undetectable even under ideal, noise-free conditions. Here, the underestimate ranges between roughly 200 and 2000ft with the worst mismatch on the northwest sides of the plumes (Figure 18). More generally, the mismatch between seismic amplitude and the actual plume depends on the rate of lateral change of saturation. Rapid lateral change in the saturation will result in relatively small mismatch whereas slow lateral change may result in much larger mismatch. The reservoir model can be used as calibration tool to predict the rate of change in saturation.

These findings reinforce the need for a "model–map–monitor" framework. Instead of assuming the base case model is "correct" and retrofitting a monitoring plan to it, this workflow accepts uncertainty by running multiple realizations. The result is a spatial-temporal map of risk highlighting where and when plume migration deviates from expectations, and guiding monitoring toward those areas. This shift avoids false certainty and enables adaptive surveillance based on actual geologic variability. Moreover, by knowing seismic detectability limits ahead of time, operators can reverse-engineer their monitoring plan: first identify where the plume may migrate, then assess whether it will be visible to seismic, and finally determine the most suitable tool and frequency. The detectability analysis is still useful because it provides the calibration between the reservoir model and seismic survey for AoR re-evaluation. Future work could build on these results to formally integrate detectability thresholds into dynamic AoR adjustments.

Importantly, there is a demand for site-specific, risk-based justification for monitoring plans. Ensemble modeling addresses this need more effectively than traditional sensitivity plots (e.g., tornado diagrams) by asking, “What if?” What if transmissibility is higher? What if sub-seismic faults connect unexpectedly? Exploring these questions through multiple model outputs offers deeper insight into plume behavior than perfecting input parameters ever could.

Seismic tools should be selected based on site-specific risk profiles. DAS and VSP provide high-resolution imaging near wells but have limited spatial reach. Surface seismic covers broader areas but at higher cost, and its resolution is often insufficient for thin or low-saturation plumes. Thus, seismic should not be used by default it should be deployed where and when it adds value, with a clear understanding that parts of the plume will likely remain invisible to this method alone.

It is worth noting that this study did not incorporate field noise, acquisition geometry, or AVO (Amplitude Versus Offset) effects. These factors are currently under investigation through ongoing collaborations and will be critical for refining seismic detectability in future work. As such, the results presented here represent an optimistic upper bound, and real-world performance and survey limitations.

Despite these limitations, seismic forward modeling in this thesis provides valuable operational insight: it sets realistic expectations and emphasizes that detection is a probabilistic outcome, not a binary one. It also underscores the importance of integrating seismic with complementary tools such as pressure sensors to improve plume visibility and containment assurance. Pressure monitoring, in particular, is expected to offer earlier detection of migration risks and is often more resilient to site-specific limitations.

Finally, beyond technical performance, seismic monitoring plays a critical role in public trust and regulatory transparency. As seen in recent CCS projects, including those with legal disputes, stakeholders demand verifiable evidence of plume containment. A monitoring strategy backed by transparent, model-informed reasoning is not only scientifically sound but also more defensible in regulatory and public domains.

In summary, this section affirms a central principle of this research: monitoring design must begin with the models. By simulating plume uncertainty and understanding seismic constraints, operators can allocate monitoring resources intelligently, maximizing detectability, reducing unnecessary costs, and reinforcing confidence in CO_2 containment.

5.3. SEISMIC COST AND PRACTICAL CONSIDERATION

Cost remains a dominant factor in the selection, design, and justification of seismic monitoring programs for CO_2 storage—a reality consistently echoed in both industry experience and academic literature. While 3D time-lapse (4D) seismic remains the "gold standard" due to its ability to provide full-field spatial coverage and resolution, it comes with a substantial financial burden. For onshore U.S. projects, costs typically range between $50,000 and $100,000 per square mile, excluding permitting, data processing, and repeat acquisition expenses. In contrast, 2D seismic surveys are significantly cheaper $5,000 to $20,000 per linear mile but offer reduced imaging capability and limited lateral coverage.

The findings from Chapter IV reinforce the need to balance resolution with cost. While 3D seismic can detect major plume movement, the results clearly show that migration risk is neither spatially uniform nor temporally constant. Instead, the ensemble simulations reveal localized "hot zones" of plume divergence and specific post-injection time windows primarily after Year 15 when migration becomes most uncertain. A blanket, high-cost 3D seismic over the entire project area and timeline is, therefore, both technically excessive and economically inefficient.

Instead, the study advocates a tiered, model-informed monitoring strategy. Full-field 3D seismic, while highly effective, is best reserved for initial site characterization or milestone verification due to its cost. In contrast, repeating 2D seismic along model-predicted risk corridors can achieve substantial cost savings often five to ten times lower than 3D seismic while still capturing critical plume dynamics. Emerging techniques like "spotlight" (*Spotlight Earth*, n.d.) seismic surveys, represent an even more affordable alternative, offering localized imaging with minimal acquisition footprint (Al Khatib et al., 2021) add company webiste. This hybrid approach is supported in the literature (see also Kazemeini et al., 2010) and strongly aligns with EPA Class VI guidance, which requires at least one direct and one indirect monitoring method but allows operators the flexibility to develop scientifically defensible, risk-prioritized strategies tailored to site conditions.

Seismic monitoring tools should be matched to risk tier and detection need. Vertical Seismic Profiling (VSP) and Distributed Acoustic Sensing (DAS) offer high-resolution detection near injection wells, but their spatial coverage is limited. Conversely, surface seismic can image broader areas but suffers from higher cost, lower repeatability, and site-specific noise challenges. The optimal solution is not to choose one tool but to combine them strategically deploying higher-cost methods when and where warranted and augmenting them with lower-cost options like DAS for continuous surveillance or early-warning triggers. This principle of tool integration also increases resilience to non-detection risks and enhances cross-validation across different monitoring technologies.

Crucially, as demonstrated in Chapter IV, even the most advanced seismic tools cannot guarantee full detectability of CO_2 plumes unless the modeling parameters are carefully refined and replicated under real operational conditions. This requires that acquisition parameters such as the source signal, receiver sensitivities, and background noise precisely match those used in the model. If not properly aligned, the monitoring process can become unstable or misleading, much like a circular reference error in Excel, where outputs feed back into inputs without resolution. Due to physics and site limitations, seismic anomalies often lag behind actual plume edges.

The "model–map–monitor" workflow developed here is explicitly designed to support this integration. By simulating uncertainty across multiple realizations, it identifies both where the plume might go and when monitoring is most needed. This enables operators to design scientifically justified, cost-efficient, and regulatorily compliant programs while avoiding the common mistake of anchoring plans to a single reservoir model. This also empowers decision-makers to rationalize budget allocations and optimize monitoring schedules in alignment with regulatory milestones and performance-based closure requirements.

In summary, seismic monitoring is valuable, but most cost effective when used selectively and strategically. By leveraging ensemble modeling and spatial risk mapping, operators can reduce costs without compromising containment assurance delivering credible, transparent, and adaptable monitoring solutions, especially in complex regions like the Gulf Coast.

Cost Model

The realistic single-flow-unit model in Figure 19 illustrates a visual progression of plume-based risk mapping. The model spans approximately 71,500 ft × 94,500 ft (~21.8 km × 28.8 km) with a grid of 143 × 189 × 10 cells and a resolution of 500 ft × 500 ft × 20 ft. The left panel displays the maximum gas saturation for the base case. The center panel represents the stacked maximum gas saturation across all 20 realizations, highlighting recurrent plume presence in the central corridor trending northwest. To isolate high-risk zones—those reflecting unacceptable deviation—the right panel subtracts the base case from the stacked ensemble. This leaves behind areas where plume behavior diverged beyond acceptable limits, revealing a narrow, repeatable migration path.

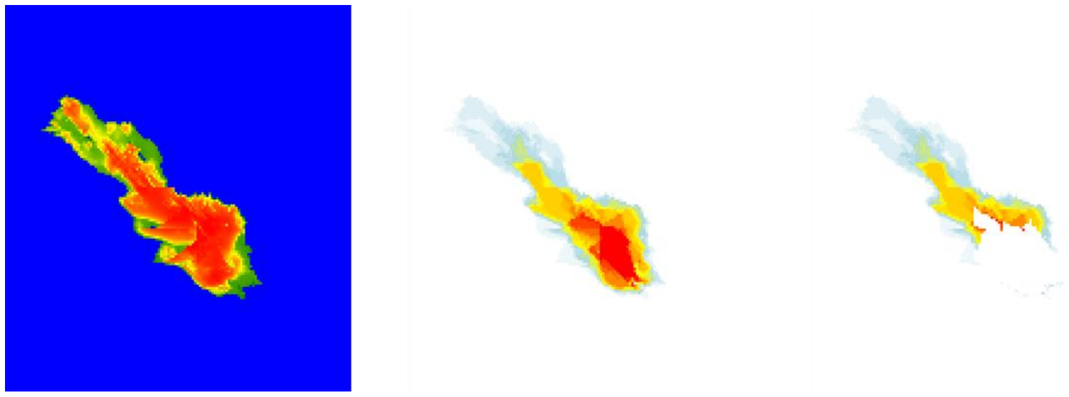


Figure 19 CO_2 plume migration risk maps from the single flow-unit model. Left: Base case maximum gas saturation. Center: Stacked maximum saturation from all 20 realizations. Right: Areas where the ensemble diverges from the base case, showing where unacceptable plume spread is most likely (high-risk corridor). Model area is 71,500 ft × 94,500 ft (~21.8 km × 28.8 km) with 500 ft grid spacing.

This ensemble-based plume divergence mapping provides the foundation for cost-efficient monitoring design. As shown in Figure 19, the majority of plume presence is concentrated within an elliptical, northwest-trending zone occupying a fraction of the full model area. Therefore, to reflect a more realistic upper-bound cost scenario, the "full 3D seismic" assumption is revised to cover ~29.08 mi^2, or approximately 12% of the full model domain (242.4 mi^2) covers the area of plume extent. Meanwhile, based on qualitative analysis of the rightmost heatmap, the highest-risk zone—representing unacceptable plume divergence—is assumed to occupy ~24.24 mi^2, or roughly 10% of the model area (high-risk zone or hot-zone). This spatial concentration enables a more focused seismic monitoring footprint, avoiding unnecessary coverage across low-risk regions. The high-risk corridor in this study is defined as the spatial zone where ensemble simulations show recurrent plume presence and the greatest divergence from the base-case scenario. For cost modeling purposes, three spatial strategies were considered to calculate baseline survey estimates under different coverage scenarios, using the low and high cost ranges discussed above:

- Full-field 3D seismic: revised to cover ~29.1 mi^2 (12% of the model area)
- Targeted 3D seismic: focused on the high-risk corridor (Figure 19) (~24.2 mi^2 or 10% of the model area)
- 2D seismic line: 80 grid cells × 500 ft = 40,000 ft (7.6 miles), representing the worst-case lateral plume migration path through the risk zone

Table 2 Cost estimate for a single onshore survey based on spatial analysis summarizes the estimated costs for a single seismic survey under three spatial monitoring strategies. A full-field 3D seismic survey, now redefined to cover approximately 12% of the model area (~29.1 mi^2 or 75.3 km^2), represents the highest-cost option, with estimated expenses ranging from \$1.45 million to \$7.53 million. A targeted 3D survey, focused on the high-risk corridor (~24.2 mi^2 or 62.8 km^2, or 10% of the model area), offers substantial cost reductions, with costs ranging from \$1.21 million to \$6.28 million. The lowest-cost approach is a 2D seismic line spanning the worst-case plume migration distance (7.58 mi or 12.2 km), with estimated costs between \$37,900 and \$392,400.

Table 2 Cost estimate for a single onshore survey based on spatial analysis (cost ranges from Andrey Bakulin, personal communication, 2025).

Monitoring Strategy	**Spatial Coverage**	**Cost per Survey (M\$)**
3D seismic	29.08 mi^2 (75.3 km^2)	1.45 – 7.53
Targeted 3D seismic	24.24 mi^2 (62.8 km^2)	1.21 – 6.28

2D seismic line	7.58 mi (12.2 km)	0.038 – 0.392

The cost analysis assumes a total monitoring timeline of 80 years, consisting of a 30-year CO_2 injection period followed by 50 years of post-injection monitoring, in accordance with EPA Class VI guidance. Two temporal monitoring strategies were evaluated: (1) a high-frequency approach, with seismic surveys conducted every 5 years, resulting in 17 total surveys over the monitoring period (Years 0 through 80); and (2) a time-targeted approach, with surveys conducted every 15 years, totaling six surveys at Years 0, 15, 30, 45, 60, and 75. These temporal strategies were applied to both full-field (12%) and targeted (10%) spatial coverage scenarios to compare cumulative monitoring costs under different spatial and temporal configurations.

Table 3 presents a temporal cost comparison of seismic monitoring assuming either a 5-year survey interval (17 total surveys) or a 15-year interval (6 total surveys), including baseline. The estimates are stratified across three levels of spatial monitoring: full-field 3D, targeted 3D, and 2D seismic.

- Full-field 3D seismic (revised to cover 12% of the model area, ~29.1 mi^2) is the most expensive option, with per-survey costs ranging from \$1.45M to \$7.53M. Over 17 surveys, this strategy could cost \$24.7M to \$128.1M, while a 15-year interval results in a total cost of \$8.73M to \$45.2M.
- Targeted 3D seismic (focused on the highest-risk 10% zone, ~24.2 mi^2) offers moderate savings, with per-survey costs ranging from \$1.21M to \$6.28M. Over 17 surveys, total costs range from \$20.6M to \$106.7M, and for the 15-year interval, \$7.27M to \$37.66M. This reflects a consistent cost reduction of approximately 16.7% compared to full-field 3D seismic under both temporal strategies.

- 2D seismic lines, used along the worst-case plume migration path (~7.6 mi), are the most affordable approach. With per-survey costs between $37.9K and $392.4K, total costs range from $643.9K to $6.67M for 17 surveys, and $227.3K to $2.35M for the 15-year interval. This represents a cost reduction of 95–97% compared to full-field 3D seismic under equivalent temporal conditions.

Table 3 Cost estimate based on temporal analysis

Monitoring Strategy	**Cost per Survey (M$)**	**5-Year Interval (17×) (M$)**	**15-Year Interval (6×) (M$)**
3D seismic	1.45 – 7.53	24.72 – 128.06	8.73 – 45.20
Targeted 3D seismic	1.21 – 6.28	20.60 – 106.71	7.27 – 37.66
2D seismic line	0.038 – 0.392	0.644 – 6.67	0.227 – 2.35

Overall, for onshore U.S. projects, 3D seismic survey costs typically range between $50,000 and $100,000 per square mile, excluding permitting, data processing, and repeat acquisition expenses. In contrast, 2D seismic surveys are significantly cheaper, typically ranging from $5,000 to $20,000 per linear mile, though they offer reduced imaging capability and more limited spatial coverage. Based on the cost estimates presented, 2D seismic is approximately 95–97% less expensive than full-field 3D seismic. The cost tables also highlight that spatial targeting (e.g., focusing on high-risk corridors) and temporal optimization (e.g., surveys at 15-year intervals instead of every 5 years) yield substantial cumulative savings. However, cost reductions from targeted 3D seismic alone are more modest, approximately 17% lower than full-field 3D under equivalent survey frequencies. These findings underscore the value of adaptive, model-informed monitoring, where both spatial coverage and timing are optimized to balance cost efficiency with monitoring effectiveness.

5.4. RECOMMENDATION

This study supports a shift in CCS monitoring practices from default, tool-driven approaches to flexible, model-informed, risk-based frameworks. The spatial and temporal analysis presented demonstrates that CO_2 plume migration is neither uniform nor static. Instead, risk evolves over time and space, with distinct "hot zones" and critical windows emerging under different geological scenarios. These findings align with U.S. EPA Class VI guidance and broader regulatory trends, which increasingly emphasize that monitoring plans must be site-specific, scientifically justified, and cost-effective.

In response to these evolving expectations, this research advocates for a model–map–monitor strategy that leverages ensemble-based reservoir simulations to identify where and when proactive monitoring is most valuable. Rather than relying on a single "best" model or applying blanket surveillance, operators should embrace uncertainty, use modeling to map risk, and allocate monitoring resources accordingly.

Given the limitations of seismic alone, especially in detecting low-saturation zones or operating within complex, noisy settings this study recommends a tiered, integrated monitoring framework:

i. Deploy high-cost, full-field 3D seismic selectively, such as during initial site characterization, or in response to specific anomalies.
ii. Prioritize repeat 2D and spotlight surveys over model-identified high-risk zones during periods of greatest uncertainty. These approaches save cost compared to blanket 3D seismic with minimal compromise in monitoring effectiveness. The current cost model, developed using simplified area

assumptions from a single-flow-unit framework, demonstrated cost reductions of approximately 17% for targeted 3D and up to 97% for 2D seismic, depending on the spatial and temporal strategy applied. However, due to limitations in the current Python implementation, dynamic area calculations across ensemble realizations and full economic modeling were not conducted. Future work should extend the cost model to integrate ensemble-based plume footprint analysis, enabling more granular and economically optimized monitoring design at field scale.

iii. Integrate complementary tools, especially pressure monitoring, to enhance detectability and provide earlier warnings in marginal zones. Pressure signals often precede seismic anomalies and are more resilient to site-specific noise. Future efforts should expand modeling to predict pressure responses and integrate them into monitoring plans.

In summary, the "model–map–monitor" approach demonstrated here provides a defensible, adaptable, and cost-efficient pathway for CCS monitoring design. As the industry matures and expectations rise, success will depend not on maximizing data, but on smartly targeting surveillance based on modeled risk. This framework empowers operators and regulators to achieve containment assurance while avoiding unnecessary cost paving the way for responsible, scalable CCS deployment.

Chapter VI: Conclusion

This thesis addressed a central challenge in geologic carbon storage: how to design CO_2 plume monitoring programs that are both scientifically robust and economically viable in the face of geological complexity and regulatory uncertainty. Building on a targeted review of current monitoring frameworks, regulatory guidance, and CCS literature (Chapter 2), this work developed and applied a workflow that integrates reservoir modeling, scenario-based uncertainty analysis, and synthetic seismic forward modeling (Chapters 3 and 4). This combination enabled a detailed assessment of both the technical limits and practical opportunities for risk-based, targeted monitoring of CO_2 storage projects.

The results show that CO_2 plume migration is neither spatially nor temporally uniform; "hot zones" of persistent migration and critical monitoring windows can be systematically identified by leveraging ensemble-based reservoir modeling. Heatmaps and temporal analyses demonstrated that these zones emerge only under specific geological scenarios and timeframes, supporting the adoption of adaptive, risk-prioritized monitoring strategies over uniform, one-size-fits-all approaches. This methodology also offers a shift away from traditional tornado plots and single best-case models, reframing uncertainty as a planning tool rather than a drawback.

Synthetic seismic modeling confirmed that while seismic remains a powerful tool for detecting CO_2, its sensitivity is limited by both physical thresholds and acquisition constraints. In this study, the seismic anomaly consistently lagged behind the true plume front, especially in low-saturation (noise free forward modeling). These findings emphasize the need for complementary lines of evidence, such as pressure monitoring, which may offer earlier and more consistent detection in marginal zones.

A major practical finding is that targeted monitoring using repeat 2D seismic or spotlight surveys in identified risk zones can reduce surveillance costs by a factor of five to ten compared to conventional 4D seismic, without sacrificing detection confidence when guided by reservoir modeling. While 2D and spotlight methods may offer narrower spatial coverage, when deployed along model-predicted risk corridors and during periods of greatest uncertainty, they can achieve comparable detection confidence with significantly less financial burden. Importantly, this cost benefit is enhanced when monitoring is not continuous but scheduled strategically. Thus, while the per-survey cost savings are already substantial, the total lifecycle savings can be even greater when both space and time are optimized in tandem.

The current cost model, based on simplified area assumptions from a single-flow-unit framework, demonstrated cost reductions of approximately 17% for targeted 3D seismic and up to 97% for 2D seismic, depending on the spatial and temporal monitoring strategy employed. However, due to limitations in the Python implementation, full economic modeling—particularly dynamic area quantification across all ensemble realizations—was not performed. Beyond refining spatial footprint estimates, future work should prioritize the full integration of both spatial and temporal targeting into the cost framework. This involves identifying not only where plume divergence is most likely to occur, but also when monitoring efforts are most critical based on evolving uncertainty. Such dual targeting would allow operators to allocate resources with greater precision, reduce redundant or low-value surveys, and strengthen the scientific defensibility of monitoring strategies—particularly for complex, large-scale CO_2 storage projects. This aligns with emerging regulatory trends and recent literature advocating for flexible, model-justified monitoring frameworks. While the application here is site-specific to a Gulf Coast reservoir, the general workflow is transferable and can be adapted to other CCS sites with similar uncertainty profiles.

However, the thesis also acknowledges limitations: seismic noise and AVO effects were not included, pressure data was not integrated, and seismic cost estimates remain site-specific and subject to market variability. Future work will expand upon this workflow by explicitly incorporating field seismic noise and Amplitude Versus Offset (AVO) effects into the detectability analysis, ensuring that model predictions more closely match real-world monitoring conditions. Other future work involves integrating pressure data from fluid flow simulation pressure heatmaps both spatially and temporally.

Simply, this research offers a timely and actionable roadmap for operators and regulators facing mounting pressure to reduce costs without compromising safety. The path forward is clear: ditch the default, blanket seismic and adopt smart, risk-informed monitoring. By doing so, the industry can maintain public trust, meet regulatory requirements, and accelerate the deployment of CCS at scale. In conclusion, this research provides both a technical foundation and a clear policy nudge for the CCS industry and regulators, especially the U.S. EPA, to move beyond blanket monitoring requirements. It advocates for risk-based, site-specific, and adaptive monitoring strategies that maximize both technical assurance and cost-effectiveness, supporting the sustainable and scalable deployment of carbon storage in the energy transition.

References

2023CH06676 (The Circuit Court of Cook County 2023).

ADM. (2024, August 24). *Response to EPA Notice*.

Al Khatib, H., Boubaker, Y., & Morgan, E. (2021). Breaking the seismic 4D 'image' paradigm of seismic monitoring. *First Break*, *39*(9), 85–91. https://doi.org/10.3997/1365-2397.fb2021072

Alfi, M., & Hosseini, S. A. (2016). Integration of reservoir simulation, history matching, and 4D seismic for CO2-EOR and storage at Cranfield, Mississippi, USA. *Fuel*, *175*, 116–128. https://doi.org/10.1016/j.fuel.2016.02.032

Arts, R., Eiken, O., Chadwick, A., Zweigel, P., Van Der Meer, L., & Zinszner, B. (2004). Monitoring of CO2 injected at Sleipner using time-lapse seismic data. *Energy*, *29*(9–10), 1383–1392. https://doi.org/10.1016/j.energy.2004.03.072

Barnett, H. G., Ireland, M. T., Dunham, C. K., & van der Land, C. (2025). *Low Computational Cost Stochastic Gassmann Fluid Substitution Modelling of Hydrogen and Carbon Dioxide in Clastic Storage Reservoirs*.

Bridge, J. S., & Tye, R. S. (2000). Interpreting the Dimensions of Ancient Fluvial Channel Bars, Channels, and Channel Belts from Wireline-Logs and Cores. *AAPG Bulletin*, *84*(8), 1205–1228. https://doi.org/10.1306/A9673C84-1738-11D7-8645000102C1865D

Bump, A. P., Bakhshian, S., Ni, H., Hovorka, S. D., Olariu, M. I., Dunlap, D., Hosseini, S. A., & Meckel, T. A. (2023). Composite confining systems: Rethinking geologic seals for permanent CO2 sequestration. *International Journal of*

Greenhouse Gas Control, *126*, 103908. https://doi.org/10.1016/j.ijggc.2023.103908

Chaves, G. (2024). *Effect of Sub-seismic Reservoir Heterogeneity on CO_2 Plume Migration, Onshore Gulf of Mexico (Texas, USA)*. University of Texas at Austin.

Finkbeiner, T., Zoback, M., Flemings, P., & Stump, B. (2001). Stress, pore pressure, and dynamically constrained hydrocarbon columns in the South Eugene Island 330 field, northern Gulf of Mexico. *AAPG Bulletin*, *85*(6). https://doi.org/10.1306/8626CA55-173B-11D7-8645000102C1865D

Fomel, S. (2024). *Madagascar* [C, Python]. University of Texas at Austin. https://ahay.org/wiki/Main_Page

Fomel, S., Sava, P., Vlad, I., Liu, Y., & Bashkardin, V. (2013). Madagascar: Open-source software project for multidimensional data analysis and reproducible computational experiments. *Journal of Open Research Software*, *1*(1), e8. https://doi.org/10.5334/jors.ag

Galloway, W. E. (1989). Depositional framework and hydrocarbon resources of the early Miocene (Fleming) episode, northwest Gulf Coast Basin. *Marine Geology*, *90*(1–2), 19–29. https://doi.org/10.1016/0025-3227(89)90110-2

Galloway, W. E., Ganey-Curry, P. E., Li, X., & Buffler, R. T. (2000). Cenozoic depositional history of the Gulf of Mexico basin. *AAPG Bulletin*, *84*(11), 1743–1774. https://doi.org/10.1306/8626C37F-173B-11D7-8645000102C1865D

Gao, R., Previna, A., Fomel, S., & Chen, Y. (Unpublished). *Seismic Forward Modeling* [Python]. University of Texas at Austin.

Gasperikova, E., Appriou, D., Bonneville, A., Feng, Z., Huang, L., Gao, K., Yang, X., & Daley, T. (2022). Sensitivity of geophysical techniques for monitoring secondary CO2 storage plumes. *International Journal of Greenhouse Gas Control*, *114*, 103585. https://doi.org/10.1016/j.ijggc.2022.103585

Gasperikova, E., Daley, T., Appriou, D., Bonneville, A., Feng, Z., Huang, L., Yang, X., Wang, Z., Dilmore, R., & Gao, K. (2020). *Detection Thresholds and Sensitivities of Geophysical Techniques for CO2 Plume Monitoring* (NRAP-TRS--I-001-2020, DOE/NETL--2021/2638, 1735331; p. NRAP-TRS--I-001-2020, DOE/NETL--2021/2638, 1735331). https://doi.org/10.2172/1735331

Goudarzi, A., Hosseini, S. A., Sava, D., & Nicot, J. (2018). Simulation and 4D seismic studies of pressure management and CO $_2$ plume control by means of brine extraction and monitoring at the Devine Test Site, South Texas, USA. *Greenhouse Gases: Science and Technology*, *8*(1), 185–204. https://doi.org/10.1002/ghg.1731

Hovorka, S. (2017). Assessment of Low Probability Material Impacts. *Energy Procedia*, *114*, 5311–5315. https://doi.org/10.1016/j.egypro.2017.03.1648

Hovorka, S., Nicot, J.-P., Zeidouni, M., Sun, A., Yang, C., Sava, D., Mickler, P., & Remington, R. L. (2014). *Expert-Based Development of a Standard in CO2 Sequestration Monitoring Technology*.

IEAGHG. (2019). *Monitoring Selection Tool*. Ieaghg. https://ieaghg.org/ccs-resources/monitoring-selection-tool

IPCC, Core Writing Team, H. L., & Romero, J. (2023). *Climate Change 2023: Synthesis Report. Longer Report* [Report]. IPCC. https://doi.org/10.59327/IPCC/AR6-9789291691647

Isaenkov, R., Pevzner, R., Glubokovskikh, S., Yavuz, S., Shashkin, P., Yurikov, A., Tertyshnikov, K., Gurevich, B., Correa, J., Wood, T., Freifeld, B., & Barraclough, P. (2022). Advanced time-lapse processing of continuous DAS VSP data for plume evolution monitoring: Stage 3 of the CO2CRC Otway project case study. *International Journal of Greenhouse Gas Control*, *119*, 103716. https://doi.org/10.1016/j.ijggc.2022.103716

Isaenkov, R., Pevzner, R., Glubokovskikh, S., Yavuz, S., Yurikov, A., Tertyshnikov, K., Gurevich, B., Correa, J., Wood, T., Freifeld, B., Mondanos, M., Nikolov, S., & Barraclough, P. (2021). An automated system for continuous monitoring of CO2 geosequestration using multi-well offset VSP with permanent seismic sources and receivers: Stage 3 of the CO2CRC Otway Project. *International Journal of Greenhouse Gas Control*, *108*, 103317. https://doi.org/10.1016/j.ijggc.2021.103317

Kazemeini, S. H., Juhlin, C., & Fomel, S. (2010). Monitoring CO2 response on surface seismic data; a rock physics and seismic modeling feasibility study at the CO2 sequestration site, Ketzin, Germany. *Journal of Applied Geophysics*, *71*(4), 109–124. https://doi.org/10.1016/j.jappgeo.2010.05.004

Krishnamurthy, P. G., DiCarlo, D., & Meckel, T. (2022). Geologic Heterogeneity Controls on Trapping and Migration of CO_2. *Geophysical Research Letters*, *49*(16), e2022GL099104. https://doi.org/10.1029/2022GL099104

Larue, D. K., Allen, J., Beeson, D., & Robbins, J. (2023). Fluvial reservoir architecture, directional heterogeneity and continuity, recognizing incised valley fills, and the case for nodal avulsion on a distributive fluvial system: Kern River field, California. *AAPG Bulletin*, *107*(3), 477–513. https://doi.org/10.1306/09232220163

Li, C., Bhattacharya, S., Alhotan, M. M., & Delshad, M. (2024). *Time-lapse geophysical responses of hydrogen-saturated rock: Implications on subsurface monitoring*. https://doi.org/10.31223/X52985

Lumley, D. (2010). 4D seismic monitoring of CO2 sequestration. *The Leading Edge*, *29*(2), 150–155. https://doi.org/10.1190/1.3304817

Meckel, T., & Treviño, R. H. (2014). *Gulf of Mexico Miocene CO_2 Site Characterization Mega Transect Final Scientific/Technical Report (Revised)* (DE-FE0001941). Bureau of Economic Geology, The University of Texas at Austin. https://www.netl.doe.gov/projects/files/FE0001941_FinalReport_122014.pdf

Pett-Ridge, J., Kuebbing, S., Mayer, A., Hovorka, S., Pilorgé, H., Baker, S., Pang, S., Scown, C., Mayfield, K., Wong, A., Aines, R., Ammar, H., Aui, A., Ashton, M., Basso, B., Bradford, M., Bump, A., Busch, I., Calzado, E., … Zhang, Y. (2023). *Roads to Removal: Options for Carbon Dioxide Removal in the United States*

(LLNL--TR-852901, 2301853, 1080440; p. LLNL--TR-852901, 2301853, 1080440). https://doi.org/10.2172/2301853

Pevzner, R., Isaenkov, R., Yavuz, S., Yurikov, A., Tertyshnikov, K., Shashkin, P., Gurevich, B., Correa, J., Glubokovskikh, S., Wood, T., Freifeld, B., & Barraclough, P. (2021). Seismic monitoring of a small CO2 injection using a multi-well DAS array: Operations and initial results of Stage 3 of the CO2CRC Otway project. *International Journal of Greenhouse Gas Control*, *110*, 103437. https://doi.org/10.1016/j.ijggc.2021.103437

Pickering, G., Bull, J. M., & Sanderson, D. J. (1996). Scaling of fault displacements and implications for the estimation of sub-seismic strain. *Geological Society, London, Special Publications*, *99*(1), 11–26. https://doi.org/10.1144/GSL.SP.1996.099.01.03

Ramirez-Franco, J. (2024). *First commercial CCS plant is in Illinois. It leaks. | Grist*.

Romanak, K. D., Wolaver, B., Yang, C., Sherk, G. W., Dale, J., Dobeck, L. M., & Spangler, L. H. (2014). Process-based soil gas leakage assessment at the Kerr Farm: Comparison of results to leakage proxies at ZERT and Mt. Etna. *International Journal of Greenhouse Gas Control*, *30*, 42–57. https://doi.org/10.1016/j.ijggc.2014.08.008

Smith, T. M., Sondergeld, C. H., & Rai, C. S. (2003). Gassmann fluid substitutions: A tutorial. *GEOPHYSICS*, *68*(2), 430–440. https://doi.org/10.1190/1.1567211

Spotlight Earth. (n.d.). Spotlight Earth. Retrieved August 8, 2025, from https://spotlight-earth.com/

UIC. (2013a). *Underground Injection Control (UIC) Program Class VI Well Area of Review Evaluation and Corrective Action Guidance*.

UIC. (2013b). *Underground Injection Control (UIC) Program Class VI Well Testing and Monitoring Guidance*.

Urosevic, M., Pevzner, R., Shulakova, V., Kepic, A., Caspari, E., & Sharma, S. (2011). Seismic monitoring of CO2 injection into a depleted gas reservoir–Otway Basin Pilot Project, Australia. *Energy Procedia*, *4*, 3550–3557. https://doi.org/10.1016/j.egypro.2011.02.283

U.S. Environmental Protection Agency. (2021). *Attachment C: Testing and Monitoring Plan* (Permit Attachment IL-115-6A-0001; Archer Daniels Midland CCS#2 Class VI Permit Application). U.S. Environmental Protection Agency, Region 5.

Vasco, D. W., Alfi, M., Hosseini, S. A., Zhang, R., Daley, T., Ajo-Franklin, J. B., & Hovorka, S. D. (2019). The Seismic Response to Injected Carbon Dioxide: Comparing Observations to Estimates Based Upon Fluid Flow Modeling. *Journal of Geophysical Research: Solid Earth*, *124*(7), 6880–6907. https://doi.org/10.1029/2018JB016429

Victor, N., & Nichols, C. (2022). CCUS deployment under the U.S. 45Q tax credit and adaptation by other North American Governments: MARKAL modeling results. *Computers & Industrial Engineering*, *169*, 108269. https://doi.org/10.1016/j.cie.2022.108269

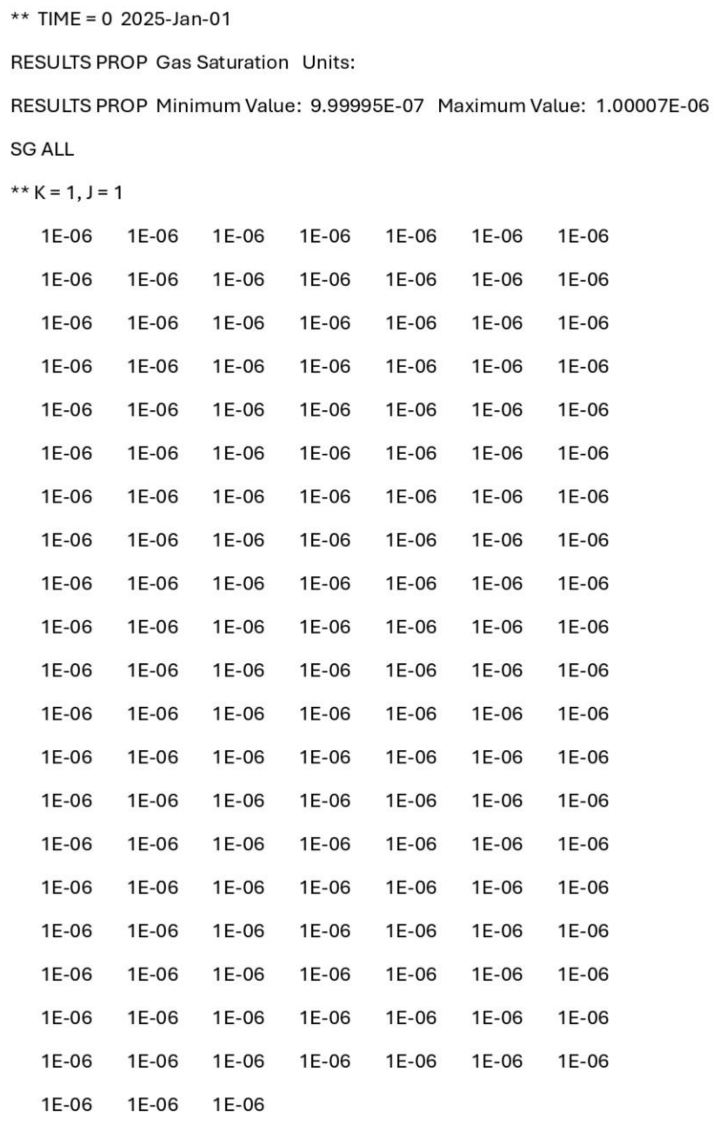
White, D. J. (2011). Geophysical monitoring of the Weyburn CO2 flood: Results during 10 years of injection. *Energy Procedia*, *4*, 3628–3635. https://doi.org/10.1016/j.egypro.2011.02.293

Yurikov, A., Tertyshnikov, K., Yavuz, S., Shashkin, P., Isaenkov, R., Sidenko, E., Glubokovskikh, S., Barraclough, P., & Pevzner, R. (2022). Seismic monitoring of CO2 geosequestration using multi-well 4D DAS VSP: Stage 3 of the CO2CRC Otway project. *International Journal of Greenhouse Gas Control*, *119*, 103726. https://doi.org/10.1016/j.ijggc.2022.103726

Zhang, R., Song, X., Fomel, S., Sen, M. K., & Srinivasan, S. (2013). Time-lapse seismic data registration and inversion for CO2 sequestration study at Cranfield. *GEOPHYSICS*, *78*(6), B329–B338. https://doi.org/10.1190/geo2012-0386.1

Appendix (or Appendices)

APPENDIX A: COMPUTER MODELING GROUP (CMG) SIMULATION INPUT FILE (.SIF)



```
** TIME = 0  2025-Jan-01
RESULTS PROP  Gas Saturation   Units:
RESULTS PROP  Minimum Value:  9.99995E-07   Maximum Value:  1.00007E-06
SG ALL
** K = 1, J = 1
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06    1E-06
    1E-06    1E-06    1E-06
```

APPENDIX B: PYTHON WORKFLOW FOR SPATIAL AND TEMPORAL GAS SATURATION ANALYSIS

```
+*In[ ]:*+

[source, ipython3]

----

import os

import pandas as pd

from tqdm import tqdm  # For progress bar

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

from matplotlib.colors import LinearSegmentedColormap

import re

import datetime

from matplotlib.cm import get_cmap

import csv

----




+*In[ ]:*+

[source, ipython3]

----

def find_case_files(data_dir, pattern="Gas Saturation.txt"):

    return sorted([f for f in os.listdir(data_dir) if pattern in f])


def parse_and_max_gas_map(file_path):
    data = []
    with open(file_path, 'r') as file:
        year, k_index, j_index, i_index = None, None, None, None
        for line in file:
            if line.startswith("**  TIME ="):
                year = line.split()[4].split("-")[0]
            elif line.startswith("** K =") and "J =" in line:
                k_index = int(line.split("K = ")[1].split(",")[0].strip())
                j_index = int(line.split("J = ")[1].strip())
                i_index = 1
            elif line.strip() and not line.startswith("**") and not line.startswith("RESULTS"):
                try:
                    for val in map(float, line.split()):
                        data.append((year, k_index, j_index, i_index, val))
                        i_index += 1
                except ValueError:
                    continue
    df = pd.DataFrame(data, columns=["Year", "K", "J", "I", "Gas Saturation"])
    df_max = df.groupby(["Year", "J", "I"])["Gas Saturation"].max().reset_index()
    return df_max

def save_case_gas_map(df, label, output_dir):
    out_path = os.path.join(output_dir, f"{label}.feather")
    df.to_feather(out_path)
```

```
def batch_process_gas_maps(data_dir, output_dir):
    os.makedirs(output_dir, exist_ok=True)
    files = find_case_files(data_dir)

    # Define your base case pattern (edit if you want a stricter/looser match)
    base_case_pattern = "Case-CW-2-Fullfield-Faults-90degrees-Trans-0.0-Date-08-05-2024 Gas Saturation.txt"
    mapping = []

    for i, fname in enumerate(tqdm(files, desc="Processing cases")):
        if fname == base_case_pattern:
            label = "Base_Case"
        else:
            label = f"Case_{i+1}"
        # Save mapping info
        mapping.append({"Case_Label": label, "Filename": fname})

        # Process and save
        df = parse_and_max_gas_map(os.path.join(data_dir, fname))
        save_case_gas_map(df, label, output_dir)

    # Save mapping as CSV
    mapping_path = os.path.join(output_dir, "case_file_mapping.csv")
    pd.DataFrame(mapping).to_csv(mapping_path, index=False)
    print(f"✅ All max gas maps saved to {output_dir}")
    print(f"✅ Mapping saved to {mapping_path}")


# --- Usage ---
data_dir = "./"  # Path to your text files
output_dir = "processed/gas_maps"
batch_process_gas_maps(data_dir, output_dir)
----



+*In[ ]:*+
[source, ipython3]
----
# Colormap for gas saturation (blue-green-yellow-red)
cmg_cmap = LinearSegmentedColormap.from_list("cmg", [
    (0.0, "blue"), (0.01, "green"), (0.5, "yellow"), (1.0, "red")
])
# Colormap for count maps (hot scale)
hot_cmap = LinearSegmentedColormap.from_list("hot_thresholded", [
    (0.0, "#ffffff"), (0.25, "#add8e6"), (0.5, "#ffff00"),
    (0.75, "#ffa500"), (1.0, "#ff0000")
])
----
```

+*In[ ]:*+

[source, ipython3]

```
----
def natural_sort_key(s):
    return [int(text) if text.isdigit() else text.lower() for text in re.split(r'(\d+)', s)]
----
```

+*In[ ]:*+

[source, ipython3]

```
----
# Directory with your processed .feather files
gas_map_dir = "processed/gas_maps"

# Get all feather files, sorted
case_files = [f for f in os.listdir(gas_map_dir) if f.endswith('.feather')]
case_labels = [f.replace('.feather', '').replace('_', ' ') for f in case_files]
case_labels_sorted = sorted(case_labels, key=natural_sort_key)

# Identify base case (by label containing 'base') and other cases
base_case_label = [label for label in case_labels_sorted if 'base' in label.lower()][0]
other_case_labels = [label for label in case_labels_sorted if label != base_case_label]

# Load all case data into a dictionary
```

```
case_data = {}
for label in case_labels_sorted:
    path = os.path.join(gas_map_dir, f"{label.replace(' ', '_')}.feather")
    case_data[label] = pd.read_feather(path)
----
```

```
+*In[ ]:*+
[source, ipython3]
----
def get_gas_grid(df, year):
    """
    Returns a 2D grid (J, I) of gas saturation for a given year.
    """
    filtered = df[df["Year"] == year]
    if filtered.empty:
        return None
    i_max = filtered["I"].max()
    j_max = filtered["J"].max()
    grid = np.full((j_max, i_max), np.nan)
    for _, row in filtered.iterrows():
        grid[int(row["J"]) - 1, int(row["I"]) - 1] = row["Gas Saturation"]
    return grid

def get_binary_grid(gas_grid, threshold=0.01):
    """
    Converts gas grid to binary: 1 if value > threshold, else 0.
    """
    if gas_grid is None:
        return None
    return (gas_grid > threshold).astype(np.uint8)
----
```

+*In[ ]:*+

[source, ipython3]

```
----
def plot_all_panels(case_data, base_case_label, other_case_labels, years, well_i,
well_j, threshold=0.01):
    n_panels = 5
    fig, axs = plt.subplots(
        n_panels, len(years),
        figsize=(len(years)*3.5, n_panels*2.8),
        constrained_layout=True
    )
    vmax_gas = 0.76  # Can adjust if you want full [0,1] scale
    vmax_count = len(other_case_labels)

    for col_idx, year in enumerate(years):
        # --- Base Case Gas Saturation ---
```

```
base_gas = get_gas_grid(case_data[base_case_label], year)
if base_gas is None:
    for row in range(n_panels):
        axs[row, col_idx].axis("off")
    continue

# --- Stacked Gas Saturation (max across all cases, per cell) ---
stacked_gas = np.copy(base_gas)
for label in other_case_labels:
    grid = get_gas_grid(case_data[label], year)
    if grid is not None:
        stacked_gas = np.maximum(stacked_gas, np.nan_to_num(grid, nan=0))

# --- Base Binary ---
base_bin = get_binary_grid(base_gas, threshold)

# --- Stacked Binary: Number of cases with plume ---
stack_bin = np.zeros_like(base_bin)
for label in other_case_labels:
    grid = get_gas_grid(case_data[label], year)
    if grid is not None:
        stack_bin += get_binary_grid(grid, threshold)

# --- Additions Only: plume appears in cases but not base ---
additions = np.where((stack_bin > 0) & (base_bin == 0), stack_bin, 0)
```

```
        # --- Panel 1: Base Case Gas Saturation ---
        im0 = axs[0, col_idx].imshow(base_gas, cmap=cmg_cmap, origin="lower",
vmin=0, vmax=vmax_gas)
        axs[0, col_idx].plot(well_i-1, well_j-1, marker='+', color='black',
markersize=9, markeredgewidth=2)
        axs[0, col_idx].set_title(f"Base Gas ({year})", fontsize=9)
        axs[0, col_idx].set_xticks([]); axs[0, col_idx].set_yticks([])

        # --- Panel 2: Stacked Gas Saturation ---
        im1 = axs[1, col_idx].imshow(stacked_gas, cmap=cmg_cmap,
origin="lower", vmin=0, vmax=vmax_gas)
        axs[1, col_idx].plot(well_i-1, well_j-1, marker='+', color='black',
markersize=9, markeredgewidth=2)
        axs[1, col_idx].set_title(f"Stacked Max Gas ({year})", fontsize=9)
        axs[1, col_idx].set_xticks([]); axs[1, col_idx].set_yticks([])

        # --- Panel 3: Base Binary (black & white) ---
        im2 = axs[2, col_idx].imshow(base_bin, cmap='gray', origin="lower",
vmin=0, vmax=1)
        axs[2, col_idx].plot(well_i-1, well_j-1, marker='+', color='red', markersize=9,
markeredgewidth=2)
        axs[2, col_idx].set_title(f"Base Binary ({year})", fontsize=9)
        axs[2, col_idx].set_xticks([]); axs[2, col_idx].set_yticks([])
        # --- Panel 4: Stacked Binary (number of cases) ---
        im3 = axs[3, col_idx].imshow(stack_bin, cmap=hot_cmap, origin="lower", vmin=0, vmax=vmax_count)
        axs[3, col_idx].plot(well_i-1, well_j-1, marker='+', color='black', markersize=9, markeredgewidth=2)
        axs[3, col_idx].set_title(f"#Cases Plume ({year})", fontsize=9)
        axs[3, col_idx].set_xticks([]); axs[3, col_idx].set_yticks([])

        # --- Panel 5: Additions Only ---
        im4 = axs[4, col_idx].imshow(additions, cmap=hot_cmap, origin="lower", vmin=0, vmax=vmax_count)
        axs[4, col_idx].plot(well_i-1, well_j-1, marker='+', color='black', markersize=9, markeredgewidth=2)
        axs[4, col_idx].set_title(f"Additions ({year})", fontsize=9)
        axs[4, col_idx].set_xticks([]); axs[4, col_idx].set_yticks([])

    # --- Row labels ---
    row_titles = [
        "Base Case Gas", "Stacked Max Gas",
        "Base Binary (>1%)", "#Cases with Plume", "Additions Only"
    ]
    for row_idx, label in enumerate(row_titles):
        axs[row_idx, 0].set_ylabel(label, fontsize=11)

    # --- Colorbars for each row (right side) ---
    fig.colorbar(im0, ax=axs[0, :], orientation='vertical', fraction=0.03, pad=0.02, label="Gas Saturation")
    fig.colorbar(im1, ax=axs[1, :], orientation='vertical', fraction=0.03, pad=0.02, label="Gas Saturation")
    fig.colorbar(im2, ax=axs[2, :], orientation='vertical', fraction=0.03, pad=0.02, label="Binary")
    fig.colorbar(im3, ax=axs[3, :], orientation='vertical', fraction=0.03, pad=0.02, label="# Cases >1%")
    fig.colorbar(im4, ax=axs[4, :], orientation='vertical', fraction=0.03, pad=0.02, label="Additions Only")

    plt.show()
----



+*In[ ]:*+
[source, ipython3]
----
# Example target years and well location
#target_years = [2030, 2035, 2040, 2055, 2225]
target_years = ["2030", "2035", "2040", "2055", "2225"]
well_i, well_j = 84, 54  # Adjust if your well location is different
----
```

+*In[ ]:*+

[source, ipython3]

```
----
start_time = datetime.datetime.now()
print(f"⏱ Plotting started at: {start_time.strftime('%Y-%m-%d %H:%M:%S')}")

plot_all_panels(
    case_data=case_data,
    base_case_label=base_case_label,
    other_case_labels=other_case_labels,
    years=target_years,
    well_i=well_i,
    well_j=well_j,
    threshold=0.01
)
end_time = datetime.datetime.now()
print(f"✅ Plotting ended at:   {end_time.strftime('%Y-%m-%d %H:%M:%S')}")

----
```

+*In[ ]:*+

[source, ipython3]

```
----
#from tqdm import tqdm  # For progress bar
```

```
def save_all_panel_maps(
    case_data, base_case_label, other_case_labels, years,
    well_i, well_j, output_dir="all_panel_maps_pngs", threshold=0.01,
    cmap_gas=None, cmap_hot=None
):
    os.makedirs(output_dir, exist_ok=True)
    panel_types = [
        "base_gas", "stacked_max_gas",
        "base_binary", "stacked_binary", "additions_only"
    ]
    if cmap_gas is None:
        from matplotlib.colors import LinearSegmentedColormap
        cmap_gas = LinearSegmentedColormap.from_list("cmg", [
            (0.0, "blue"), (0.01, "green"), (0.5, "yellow"), (1.0, "red")
        ])
    if cmap_hot is None:
        from matplotlib.colors import LinearSegmentedColormap
        cmap_hot = LinearSegmentedColormap.from_list("hot_thresholded", [
            (0.0, "#ffffff"), (0.25, "#add8e6"), (0.5, "#ffff00"),
            (0.75, "#ffa500"), (1.0, "#ff0000")
        ])
    n_cases = len(other_case_labels)
    vmax_gas = 0.76
    vmax_count = n_cases
```

```
def get_gas_grid(df, year):
    filtered = df[df["Year"] == year]
    if filtered.empty:
        return None
    i_max = filtered["I"].max()
    j_max = filtered["J"].max()
    grid = np.full((j_max, i_max), np.nan)
    for _, row in filtered.iterrows():
        grid[int(row["J"]) - 1, int(row["I"]) - 1] = row["Gas Saturation"]
    return grid


def get_binary_grid(gas_grid, threshold=0.01):
    if gas_grid is None:
        return None
    return (gas_grid > threshold).astype(np.uint8)


progress = tqdm(years, desc="Saving all maps")
for year in progress:
    # --- Base Gas ---
    base_gas = get_gas_grid(case_data[base_case_label], year)
    if base_gas is None:
        continue


    # --- Stacked Max Gas ---
```

```
        stacked_gas = None
        for label in other_case_labels:
            grid = get_gas_grid(case_data[label], year)
            if grid is not None:
                if stacked_gas is None:
                    stacked_gas = np.copy(grid)
                else:
                    stacked_gas = np.maximum(stacked_gas, np.nan_to_num(grid,
nan=0))

        # --- Base Binary ---
        base_bin = get_binary_grid(base_gas, threshold)

        # --- Stacked Binary ---
        stack_bin = np.zeros_like(base_bin)
        for label in other_case_labels:
            grid = get_gas_grid(case_data[label], year)
            if grid is not None:
                stack_bin += get_binary_grid(grid, threshold)

        # --- Additions Only ---
        additions = np.where((stack_bin > 0) & (base_bin == 0), stack_bin, 0)

        # --- Save all maps ---
        maps_to_save = {
```

```
        "base_gas":   (base_gas,      cmap_gas, (0, vmax_gas)),
        "stacked_max_gas": (stacked_gas,   cmap_gas, (0, vmax_gas)),
        "base_binary": (base_bin,      "gray",   (0, 1)),
        "stacked_binary": (stack_bin,   cmap_hot, (0, vmax_count)),
        "additions_only": (additions,   cmap_hot, (0, vmax_count))
      }
      for panel_type, (data_map, cmap, (vmin, vmax)) in maps_to_save.items():
        fig, ax = plt.subplots(figsize=(4, 4))
        im = ax.imshow(data_map, cmap=cmap, origin='lower', vmin=vmin,
vmax=vmax)
        ax.axis('off')
        plt.subplots_adjust(left=0, right=1, top=1, bottom=0)
        filename = f"{output_dir}/{panel_type}_{year}.png"
        plt.savefig(filename, dpi=300, bbox_inches='tight', pad_inches=0)
        plt.close(fig)

    print(f"✅ All maps saved to {output_dir}/ (per year and map type)")
```

+*In[ ]:*+

[source, ipython3]

```
save_all_panel_maps(
    case_data=case_data,
    base_case_label=base_case_label,
    other_case_labels=other_case_labels,
    years=target_years,
    well_i=well_i,
    well_j=well_j,
    output_dir="all_panel_maps_pngs",  # Output folder
    threshold=0.01
)
----



+*In[ ]:*+
[source, ipython3]
----
# --- Data Loading ---
gas_map_dir = "processed/gas_maps"
case_files = [f for f in os.listdir(gas_map_dir) if f.endswith('.feather')]
case_labels = [f.replace('.feather', '').replace('_', ' ') for f in case_files]
case_labels_sorted = sorted(
    case_labels,
    key=lambda s: [int(text) if text.isdigit() else text.lower() for text in
re.split(r'(\d+)', s)]
)
```

```
case_data = {}
for label in case_labels_sorted:
    path = os.path.join(gas_map_dir, f"{label.replace(' ', '_')}.feather")
    case_data[label] = pd.read_feather(path)

# --- Plume Migration Calculation (NW direction only) ---
def calculate_plume_migration(df, well_i, well_j, threshold):
    """
    Calculates the maximum migration distance of the plume
    strictly in the NW (upper-left) direction from the injection well.
    """
    df = df.copy()
    df["Year"] = df["Year"].astype(int)
    migration = []
    for year in sorted(df["Year"].unique()):
        year_df = df[df["Year"] == year]
        plume = year_df[year_df["Gas Saturation"] > threshold].copy()
        if not plume.empty:
            plume["Distance"] = np.sqrt((plume["I"] - well_i)**2 + (plume["J"] -
well_j)**2)
            nw = plume[(plume["I"] < well_i) & (plume["J"] > well_j)]
            max_dist = nw["Distance"].max() if not nw.empty else 0
        else:
            max_dist = 0
        migration.append({"Year": year, "Max Distance": max_dist})
    return pd.DataFrame(migration)

# --- Plotting Function ---
def analyze_and_plot_plume_migration(
    case_data, all_case_labels, well_i, well_j,
    threshold=0.01, target_year=2225, output_dir="visualizations",
    zoom_first_n_years=None, show_since_start=True
):
    # 1. Calculate migration for all cases
    case_migration = {}
    for case in all_case_labels:
        case_migration[case] = calculate_plume_migration(case_data[case], well_i,
well_j, threshold)

    # 2. Separate hot/cold cases
    hot_cases, cold_cases = [], []
    for case, dist_df in case_migration.items():
        final_row = dist_df[dist_df["Year"] == target_year]
        if final_row.empty or case.lower().startswith("base"):
            continue
        if final_row["Max Distance"].values[0] > 40:
            hot_cases.append(case)
        else:
            cold_cases.append(case)
    plt.figure(figsize=(13, 7))
    cold_cmap                                                                    =
matplotlib.colormaps.get_cmap("Blues").resampled(len(cold_cases))#cold_cmap      =
get_cmap("Blues", len(cold_cases))
    hot_cmap                                                                     =
matplotlib.colormaps.get_cmap("autumn").resampled(len(hot_cases))#hot_cmap       =
get_cmap("autumn", len(hot_cases))

    # 3. Find injection start year for x-axis
    all_years = []
    for case in all_case_labels:
        years = case_data[case]["Year"].astype(int).unique()
        all_years.extend(years)
    min_year = min(all_years)

    def extract_case_number(label):
        if label.lower().startswith("base"):
            return -1
        match = re.search(r'\d+', label)
        return int(match.group()) if match else float('inf')

    # 4. Plot base case
    if any("base" in case.lower() for case in case_migration):
        for base_case in [case for case in case_migration if "base" in case.lower()]:
            base_df = case_migration[base_case]
            x = base_df["Year"] - min_year if show_since_start else base_df["Year"]
            plt.plot(
                x, base_df["Max Distance"],
                linestyle="--", color="black", linewidth=1.5, marker='o', markersize=2,
                label=base_case, zorder=5
            )

    # 5. Plot cold and hot cases
    for i, case in enumerate(sorted(cold_cases, key=extract_case_number)):
        df = case_migration[case]
        x = df["Year"] - min_year if show_since_start else df["Year"]
        plt.plot(
            x, df["Max Distance"],
            color=cold_cmap(i), linewidth=1, marker='o', markersize=1.5,
            label=case
        )
    for i, case in enumerate(sorted(hot_cases, key=extract_case_number)):
        df = case_migration[case]
        x = df["Year"] - min_year if show_since_start else df["Year"]
        plt.plot(
            x, df["Max Distance"],
            color=hot_cmap(i), linewidth=1, marker='o', markersize=1.5,
            label=case
        )
```

```
# 6. Set axis limits: always x=0, y=0; y-max is local max for zoomed-in, else
auto
if zoom_first_n_years is not None:
    plt.xlim(0, zoom_first_n_years)
    # Find max y in the zoomed x-range
    y_vals = []
    for case in all_case_labels:
        df = case_migration[case]
        x = df["Year"] - min_year if show_since_start else df["Year"]
        mask = (x >= 0) & (x <= zoom_first_n_years)
        vals = df.loc[mask, "Max Distance"].values
        y_vals.extend(vals)
    if y_vals:
        plt.ylim(0, max(y_vals) * 1.05)
    else:
        plt.ylim(0, 1)
else:
    xmax = max([df["Year"].max() - min_year if show_since_start else
df["Year"].max()
            for df in case_migration.values()])
    plt.xlim(0, xmax)
    plt.ylim(bottom=0)


# 7. Labels and title
```

```
        plt.xlabel("Year Since Injection Start" if show_since_start else "Year",
fontsize=13)
        plt.ylabel("Maximum Plume Migration Distance (NW Direction)", fontsize=13)
        plt.title("NW Plume Migration Distance from Well Over Time", fontsize=15)
        plt.grid(True, linestyle=':', linewidth=0.5)

        handles, labels = plt.gca().get_legend_handles_labels()
        sorted_items = sorted(zip(labels, handles), key=lambda x:
extract_case_number(x[0]))
        sorted_labels, sorted_handles = zip(*sorted_items)
        plt.legend(sorted_handles, sorted_labels, loc='upper left',
bbox_to_anchor=(1.02, 1), fontsize="medium")

        plt.minorticks_on()
        plt.tight_layout()
        os.makedirs(output_dir, exist_ok=True)
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
        plot_path = os.path.join(output_dir,
f"plume_migration_hot_cold_{timestamp}.png")
        plt.savefig(plot_path, dpi=300, bbox_inches="tight")
        print(f"🖼 Plot saved to: {plot_path}")
        plt.show()
```

+*In[ ]:*+

[source, ipython3]

```
# Plot full years (e.g., 200 years)
analyze_and_plot_plume_migration(
    case_data=case_data,
    all_case_labels=case_labels_sorted,
    well_i=84, well_j=54,
    threshold=0.01,
    target_year=2225,
    output_dir="visualizations",
    zoom_first_n_years=None,  # auto set xlim to full range
    show_since_start=True
)
```

+*In[ ]:*+

[source, ipython3]

```
# --- Example usages ---
# Plot first 25 years
analyze_and_plot_plume_migration(
    case_data=case_data,
    all_case_labels=case_labels_sorted,
    well_i=84, well_j=54,
    threshold=0.01,
    target_year=2225,
    output_dir="visualizations",
    zoom_first_n_years=50,
    show_since_start=True
)
```

+*In[ ]:*+
[source, ipython3]

```
# --- Example usages ---

# Plot first 25 years
analyze_and_plot_plume_migration(
    case_data=case_data,
    all_case_labels=case_labels_sorted,
    well_i=84, well_j=54,
    threshold=0.01,
    target_year=2225,
    output_dir="visualizations",
    zoom_first_n_years=25,
    show_since_start=True
)

----



+*In[ ]:*+
[source, ipython3]
----
import types

# List of all user-defined function names
functions = [name for name, obj in globals().items()
             if isinstance(obj, types.FunctionType) and obj.__module__ == '__main__']
print("Functions:", functions)
print("Total functions:", len(functions))

----
```

+*In[ ]:*+

[source, ipython3]

```
import types

# List of all user-defined variable names (exclude functions, modules, and built-
ins)
variables = [name for name, obj in globals().items()
             if not name.startswith("__") and
               not isinstance(obj, types.FunctionType) and
               not isinstance(obj, types.ModuleType)]

print("Variables:", variables)
print("Total variables:", len(variables))
```

APPENDIX C: COMPUTER MODELING GROUP (CMG) GEOSTATISTICAL SOFTWARE LIBRARY (.GSLIB)

```
7
i_index
j_index
k_index
x_coord ft
y_coord ft
z_coord ft
Gas_Saturation
1 1 200 [redacted] 7 5776.73 1e-06
2 1 200 7 5784.61 1e-06
3 1 200 7 5792.51 1e-06
4 1 200 7 5800.4 1e-06
5 1 200 7 5808.29 1e-06
6 1 200 7 5816.19 1e-06
7 1 200 7 5824.24 1e-06
8 1 200 7 5832.61 1e-06
9 1 200 7 5841.11 1e-06
10 1 20 07 5849.57 1e-06
11 1 20 07 5858.02 1e-06
12 1 20 07 5866.47 1e-06
13 1 20 07 5874.93 1e-06
14 1 20 07 5883.46 1e-06
15 1 20 07 5892.09 1e-06
16 1 20 07 5900.71 1e-06
17 1 20 07 5909.33 1e-06
18 1 20 07 5917.96 1e-06
```

APPENDIX D: SEISMIC FORWARD MODELING PYTHON CODE (GAO ET AL., UNPUBLISHED)

```
+*In[ ]:*+

[source, ipython3]

----

# === Core Python & Data Handling ===

import os

import time

import glob

import random

from collections import Counter, defaultdict

from typing import Tuple, List


# === Numerical Computing ===

import numpy as np

import pandas as pd

import cupy as cp

from scipy import stats

from scipy.interpolate import griddata

from scipy.spatial import cKDTree

from scipy.ndimage import gaussian_filter

from scipy.stats import pearsonr


# === Visualization ===

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap
```

```
import seaborn as sns
from matplotlib.patches import Patch
from mpl_toolkits.mplot3d import Axes3D

# === Geophysical / Domain-Specific ===
import lasio
import gstools as gs

# === Machine Learning ===
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.pipeline import make_pipeline

# === Deep Learning (PyTorch) ===
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, ConcatDataset, random_split,
Subset

# === Utilities ===
from tqdm import tqdm
from numba import njit

----



+*In[ ]:*+
[source, ipython3]
----
def plot_well_logs(las_file, log_names, well_name):
    """
    Plots the specified well logs vs depth as a 1×n vertical subplot, including facies
tracks.

    Parameters:
    - las_file (str): Path to the LAS file.
    - log_names (list of str): List of log names to plot.

    Returns:
    - Displays a multi-panel plot of well logs vs. depth with facies track.
    """

    # Load LAS file
    las = lasio.read(las_file)
    df = las.df().reset_index()  # Convert to DataFrame and reset index to make depth
a column

    # Extract log units from LAS file
    log_units = {curve.mnemonic: curve.unit for curve in las.curves}

    # Identify facies track(s)
    facies_cols = [col for col in df.columns if col.startswith("FACIES_")]

    # Ensure requested logs exist in the file
    available_logs = [log for log in log_names if log in df.columns]
    if not available_logs:
        raise ValueError(f"None of the requested logs exist in {las_file}. Available
logs: {list(df.columns)}")

    # Define number of subplots (logs + facies track)
```

```
            n_logs = len(available_logs) + len(facies_cols)

            # Set up figure
            fig, axes = plt.subplots(1, n_logs, figsize=(n_logs * 3, 20), sharey=True,
gridspec_kw={"wspace": 0.1})  # Adjust spacing

            # Ensure axes is iterable when n_logs=1
            if n_logs == 1:
               axes = [axes]

            # fig.suptitle(f"Well Logs from {las.well['WELL'].value}", fontsize=16,
fontweight='bold')
            fig.suptitle(f"Well Logs from {well_name}", fontsize=16, fontweight='bold')  #
hide well name

            # Get depth range
            depth_min, depth_max = df["DEPT"].min(), df["DEPT"].max()

            # Plot each log
            for i, log in enumerate(available_logs):
               ax = axes[i]  # Select the correct axis
               unit = log_units.get(log, "")  # Get unit, default to empty string if not found

               if unit == "ohm.m":  # Apply logarithmic scale for resistivity
                  ax.set_xscale("log")
```

```
                    ax.plot(df[log], df["DEPT"], label=f"{log} ({unit})", color="b")
                    ax.set_xlabel(f"{log} ({unit})", fontsize=14)  # Increase axis label font size
                    ax.set_ylabel("Depth (m)", fontsize=14) if i == 0 else None  # Set y-label only
on first subplot
                    ax.invert_yaxis()  # Depth increases downward
                    ax.grid(True, linestyle="--", alpha=0.5)
                    ax.legend(fontsize=12)  # Increase legend font size

                    # Format depth tick labels to avoid excessive decimal places, hide this
                    #  ax.set_yticks(np.linspace(depth_min,  depth_max,  num=10))    #  Set  10
evenly spaced ticks
                    # ax.set_yticklabels([f"{tick:.0f}" for tick in ax.get_yticks()], fontsize=14)  #
Show rounded depths
            # turn off y-ticks everywhere except the first subplot
                # for ax in axes:
                    ax.tick_params(axis='y', which='both', left=False, labelleft=False)

                    # axes[0].tick_params(axis='y', which='both', left=True, labelleft=True)

                # ✅ Add facies track if available
                for j, facies_col in enumerate(facies_cols):
                    ax = axes[len(available_logs) + j]  # Select subplot

                    # Drop NaN facies values before mapping colors
```

```
facies_values = df[facies_col].dropna().unique()
facies_cmap = sns.color_palette("tab10", len(facies_values))
facies_dict = {val: facies_cmap[i] for i, val in enumerate(facies_values)}

# Convert facies values to categorical numerical representation
df["Facies_Num"] = df[facies_col].map({val: i for i, val in enumerate(facies_values)})

facies_cmap = ListedColormap([facies_dict[val] for val in facies_values])
ax.imshow(df["Facies_Num"].values[:, np.newaxis], aspect="auto", cmap=facies_cmap, extent=[0, 1, depth_max, depth_min])

ax.set_xlabel(facies_col, fontsize=14)
ax.set_xlim(0, 1)  # Keep facies track aligned
ax.set_xticks([])  # Remove x-ticks
# ax.set_yticks(np.linspace(depth_min, depth_max, num=10))  # ✅ Add depth ticks to facies track
# ax.set_yticklabels([f"{tick:.0f}" for tick in ax.get_yticks()], fontsize=14)  # ✅ Format depth labels
ax.set_title(f"Facies Track", fontsize=14, fontweight="bold")

# Create a legend
handles = [plt.Line2D([0], [0], color=facies_dict[val], lw=4, label=f"Facies {int(val)}") for val in facies_values]
ax.legend(handles=handles, title=facies_col, fontsize=12, loc="upper right")
```

```
    # plt.tight_layout(rect=[0, 0, 1, 0.96])  # Adjust layout to fit title
    plt.show()

----




+*In[ ]:*+
[source, ipython3]
----
def plot_two_layers(
    layer1_records, layer2_records, plt_title,
    layer1_title="Layer 1", layer2_title="Layer 2",
    property_1="facies", property_2="facies", cmap="viridis", lowper=5,
hiper=95,syn=True
):
    """
    Plot two layers side by side with the specified property.

    Parameters:
        layer1_records (DataFrame): Data for the first layer.
        layer2_records (DataFrame): Data for the second layer.
        layer1_title (str): Title for the first layer plot.
        layer2_title (str): Title for the second layer plot.
        property_name (str): The column name of the property to plot.
        cmap (str): Colormap for the plots.
    """
    # Create pivot tables for both layers
    layer1_pivot = layer1_records.pivot(index="j_index", columns="i_index", values=property_1)
    layer2_pivot = layer2_records.pivot(index="j_index", columns="i_index", values=property_2)
    # print(f'layer1_pivot')

    vs_vals = layer1_records[property_1].dropna()
    vmin1 = np.percentile(   vs_vals, lowper)
    vmax1 = np.percentile(   vs_vals, hiper)
    if syn==True:
        print(f"vmin1 : {vmin1}, vmax1: {vmax1}")

    vs_vals = layer2_records[property_2].dropna()
    vmin2 = np.percentile(vs_vals, lowper)
    vmax2 = np.percentile(vs_vals, hiper)

    # vmin2 = np.percentile(    pd.concat([ layer2_records[property_2]]), 1)
    # vmax2 = np.percentile(    pd.concat([ layer2_records[property_2]]), 99)
    print(f"vmin2 : {vmin2}, vmax2: {vmax2}")
    vmin = np.min([vmin1, vmin2])
```

```
        vmax = np.max([vmax1, vmax2])
        print(f"vmin : {vmin}, vmax: {vmax}")

        # Set up the figure and axes for subplots
        fig, axes = plt.subplots(1, 2, figsize=(16, 8))

        # Plot the first layer
        if syn:
            vmin1=vmin
            vmin2=vmin
            vmax1=vmax
            vmax2=vmax

        im1 = axes[0].imshow(
            layer1_pivot,
            cmap=cmap,
            origin="lower",
            extent=[
                layer1_pivot.columns.min(), layer1_pivot.columns.max(),
                layer1_pivot.index.min(),            layer1_pivot.index.max()],vmin=vmin1,
vmax=vmax1
        )
        axes[0].set_title(layer1_title)
        axes[0].set_xlabel("i_index")
        axes[0].set_ylabel("j_index")
```

```
    fig.colorbar(im1, ax=axes[0], label=property_1.capitalize())

    # Plot the second layer
    im2 = axes[1].imshow(
        layer2_pivot,
        cmap=cmap,
        origin="lower",
        extent=[
            layer2_pivot.columns.min(), layer2_pivot.columns.max(),
            layer2_pivot.index.min(), layer2_pivot.index.max()],
         vmin=vmin2, vmax=vmax2
    )
    axes[1].set_title(layer2_title)
    axes[1].set_xlabel("i_index")
    axes[1].set_ylabel("j_index")
    fig.colorbar(im2, ax=axes[1], label=property_2.capitalize())

    # Show the plots
    fig.suptitle(plt_title, fontsize=16, y=0.95)
    plt.tight_layout()
    plt.show()
```

+*In[ ]:*+

[source, ipython3]

```
----
def read_las(file, printLogs=False):
    las = lasio.read(file)
    if printLogs:
        for curve in las.curves:
            print(f"  {curve.mnemonic:10} | {curve.unit:6} | {curve.descr}")

    df = las.df().reset_index()
    df["WELL"] = file  # Track well source

    # Identify facies columns
    facies_cols = [col for col in df.columns if col.startswith(strFaciesPrefix)]
    if not facies_cols:
        print(f"⚠ No facies columns found in {file}, skipping.")
        return None

    # Choose facies column with the most non-null values
    best_facies_col = max(facies_cols, key=lambda col: df[col].count())
    df["FACIES_SELECTED"] = df[best_facies_col]

    print(f"✅ Using facies column {best_facies_col} in {file}")
    return df
```

+*In[ ]:*+

[source, ipython3]

----

def predictLog(target_log, strFaciesPrefix, zone_log, las_files, predictors, plotit=True, normalize=True, method='RF', poly_degree=1):

"""

Predicts a given log using available well logs, facies, and depth, grouped by zones, using Random Forest or Polynomial Regression.

Parameters:

- target_log (str or list): Log(s) to predict (e.g., 'VP', 'XVCL', 'XRHOB')

- strFaciesPrefix (str): Facies column prefix (e.g., 'FACIES_')

- zone_log (str): The log that defines geological zones (e.g., 'ZONELOG')

- las_files (list): List of LAS file paths

- predictors (list): List of predictor log names (e.g., ['XPORT', 'XRESD', 'XSP', 'XVCL'])

- plotit (bool): Whether to plot the results (default: True)

- normalize (bool): Whether to normalize the predictors before using RandomForest or Polynomial Regression.

```
    - method (str): 'RF' for Random Forest or 'PN' for Polynomial Regression.
    - poly_degree (int): Degree of the polynomial model (default: 1).

    Returns:
    - las_df (pd.DataFrame): DataFrame with predicted log added.
    """
    # -------------------------------------------
    # 1. LOG IMPORT
    # -------------------------------------------
    dfs, used_wells = [], []

    for f in las_files:
        df = read_las(f)
        if df is not None and any(col in df.columns for col in (target_log if
isinstance(target_log, list) else [target_log])):
            dfs.append(df)
            used_wells.append(f)

    if not dfs:
        raise ValueError(f"No LAS files contain {target_log}. Check available logs.")

    print(f"✅ Used LAS files: {used_wells}")

    las_df = pd.concat(dfs, ignore_index=True)
```

```
# -------------------------------------------
# 2. DETERMINE COMMON LOGS ACROSS USED WELLS
# -------------------------------------------
available_logs = list(set.intersection(*[set(df.columns) for df in dfs]))
common_logs = [log for log in predictors if log in available_logs]

print(f"✅ Common logs across used wells: {common_logs}")

# Ensure target_log exists
selected_log = target_log if isinstance(target_log, str) else target_log[0]
print(f"✅ Using target log: {selected_log}")

# -------------------------------------------
# 3. APPLY LOG TRANSFORMATION TO RESISTIVITY AND Vp
# -------------------------------------------
if 'XRESD' in common_logs:
    las_df['XRESD'] = np.log(las_df['XRESD'])  # Apply logarithm
    print("✅ Applied log transformation to XRESD (Resistivity)")

if selected_log == 'Vp':
    obj_log = 'Vp_log'
    las_df[obj_log] = np.log(las_df[selected_log])  # Log-transform Vp
else:
    obj_log = selected_log
```

```
# ------------------------------------------
# 4. FILTER DATA & CHECK MISSING VALUES
# ------------------------------------------
relevant_logs = list(set([obj_log, "FACIES_SELECTED", zone_log, 'DEPT'] +
common_logs))
# relevant_logs = list(set([obj_log, "FACIES_SELECTED",  'DEPT'] +
common_logs))
print("Missing values per column before dropna():")
print(las_df[relevant_logs].isna().sum())

required_cols = [obj_log, "FACIES_SELECTED", "DEPT"] + common_logs  #
Columns that must NOT contain NaNs
las_df = las_df[relevant_logs].dropna(subset=required_cols)

print(f'✅ las_df after dropna: {las_df.shape}')

if len(las_df) < 50:
   print(f"⚠ Warning: Only {len(las_df)} valid data points available!")

facies_labels = las_df["FACIES_SELECTED"].unique()

if plotit:
   plt.figure(figsize=(12, 6))
   sns.histplot(data=las_df, x=selected_log, hue="FACIES_SELECTED",
kde=True, bins=30, palette="tab10")
```

```
        plt.xlabel(selected_log)
        plt.ylabel("Frequency")
        plt.title(f"Histogram of {selected_log} by Facies (Original Data)")
        plt.legend(title="Facies", labels=[f"Facies {int(f)}" for f in facies_labels])
        plt.show()

        selected_zones = [1, 3, 5, 7, 9, 11]
        fig, axes = plt.subplots(2, 3, figsize=(12, 6), sharex=True, sharey=True)

        for i, zone in enumerate(selected_zones):
            row, col = divmod(i, 3)  # Determine subplot position
            zone_data = las_df[las_df[zone_log] == zone]

            sns.histplot(data=zone_data,                              x=selected_log,
hue="FACIES_SELECTED", kde=True, bins=30, palette="tab10", ax=axes[row, col])
            axes[row, col].set_xlabel(selected_log)
            axes[row, col].set_ylabel("Frequency")
            axes[row, col].set_title(f"Zone {zone}")

            # Add a legend inside each subplot
            handles  =  [Patch(facecolor=sns.color_palette("tab10")[i],  label=f"Facies
{int(f)}") for i, f in enumerate(facies_labels)]
            # Set the legend with facies labels
            axes[row, col].legend(handles=handles, title="Facies", loc="upper right")
```

```
    plt.tight_layout()
    plt.show()

# -------------------------------------------
# 5. TRAIN & PREDICT `target_log` USING RANDOM FOREST OR
POLYNOMIAL REGRESSION (GROUPED BY ZONE)
# -------------------------------------------
las_df[f"{obj_log}_Predicted"] = np.nan

for zone in las_df[zone_log].unique():
    zone_data = las_df[las_df[zone_log] == zone].dropna(subset=[obj_log] +
common_logs)
    if len(zone_data) < 10:
        print(f"⚠ Skipping zone {zone} (Too few data points: {len(zone_data)})")
        continue

    X = zone_data[common_logs].copy()
    y = zone_data[obj_log].copy()

    # Apply normalization if enabled
    if normalize:
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)
    else:
        X_scaled = X
```

```
            if method == 'RF':  # Random Forest
                rf = RandomForestRegressor(n_estimators=100, max_depth=None,
random_state=42, n_jobs=-1)
                rf.fit(X_scaled, y)
                las_df.loc[zone_data.index, f"{obj_log}_Predicted"] =
rf.predict(X_scaled)
            elif method == 'PN':  # Polynomial Regression
                poly_model = make_pipeline(PolynomialFeatures(degree=poly_degree),
LinearRegression())
                poly_model.fit(X_scaled, y)
                las_df.loc[zone_data.index, f"{obj_log}_Predicted"] =
poly_model.predict(X_scaled)

        if selected_log == 'Vp':
            las_df[f"{selected_log}_Predicted"] =
np.exp(las_df[f"{obj_log}_Predicted"])  # Convert back from log

        # ------------------------------------------
        # 6. COMPARE ORIGINAL VS. PREDICTED `target_log` PER FACIES
        # ------------------------------------------
        original_std = las_df.groupby("FACIES_SELECTED")[selected_log].std()
        predicted_std =
las_df.groupby("FACIES_SELECTED")[f"{selected_log}_Predicted"].std()
```

```
            std_comparison = pd.DataFrame({"Original Std": original_std, "Predicted Std":
predicted_std})
            print(f"\n📊 Standard Deviation of {selected_log} by Facies:")
            print(std_comparison)

            if plotit:
               plt.figure(figsize=(12, 6))
               sns.histplot(data=las_df,                              x=f"{selected_log}_Predicted",
hue="FACIES_SELECTED", kde=True, bins=30, palette="tab10")
               plt.xlabel(f"{selected_log}_Predicted")
               plt.ylabel("Frequency")
               plt.title(f"Histogram of {selected_log} by Facies (processed)")
               plt.legend(title="Facies", labels=[f"Facies {int(f)}" for f in facies_labels])
               plt.show()

               selected_zones = [1, 3, 5, 7, 9, 11]
               fig, axes = plt.subplots(2, 3, figsize=(12, 6), sharex=True, sharey=True)

               for i, zone in enumerate(selected_zones):
                  row, col = divmod(i, 3)  # Determine subplot position
                  zone_data = las_df[las_df[zone_log] == zone]
                  sns.histplot(data=zone_data,                     x=f"{selected_log}_Predicted",
hue="FACIES_SELECTED", kde=True, bins=30, palette="tab10", ax=axes[row, col])
                  axes[row, col].set_xlabel(f"Predicted {selected_log}")
                  axes[row, col].set_ylabel("Frequency")
            axes[row, col].set_title(f"Zone {zone}")
            # Add a legend inside each subplot
            handles = [Patch(facecolor=sns.color_palette("tab10")[i], label=f"Facies {int(f)}") for i, f in enumerate(facies_labels)]
            # Set the legend with facies labels
            axes[row, col].legend(handles=handles, title="Facies", loc="upper right")
        plt.tight_layout()
        plt.show()

    # -------------------------------------------
    # 7. COMPUTE MEAN & STD PER FACIES
    # -------------------------------------------
    predicted_mean = las_df.groupby("FACIES_SELECTED")[f"{selected_log}_Predicted"].mean()
    predicted_std = las_df.groupby("FACIES_SELECTED")[f"{selected_log}_Predicted"].std()

    mean_std_df = pd.DataFrame({
        "Mean": predicted_mean,
        "Std": predicted_std
    })
    print(f"\n📊 Mean & Standard Deviation of {selected_log} by Facies:")
    print(mean_std_df)

    return las_df, mean_std_df
```

+*In[ ]:*+

[source, ipython3]

```
def nan_gaussian_filter_corrected(arr, sigma=1.5):
    mask = ~np.isnan(arr)
    arr_filled = np.where(mask, arr, 0)
    smoothed = gaussian_filter(arr_filled, sigma=sigma)
    weight = gaussian_filter(mask.astype(np.float32), sigma=sigma)
    with np.errstate(invalid='ignore', divide='ignore'):
        result = smoothed / weight
        result[weight < 1e-3] = np.nan
    return result
```

+*In[ ]:*+

[source, ipython3]

```
n1=288  # size of x (ft)

n2=314  # size of y (ft)

n3=200  # size of z (depth)

# 18086400 grids


# read in facies distribution file, exported from Petrel.

input_file = "FID.gslib"

# read the facies data

with open(input_file, "r") as infile:

    lines = infile.readlines()

# Skip the first 9 lines (format explanation text)

data_lines = lines[9:]


i_index, j_index, k_index = [], [], []

x_coord, y_coord, z_coord, facies = [], [], [], []


# Process each data line
```

```
for line in data_lines:
    columns = line.split()
    if len(columns) == 7:  # Ensure there are exactly 7 columns
        i_index.append(int(columns[0]))
        j_index.append(int(columns[1]))
        k_index.append(int(columns[2]))
        x_coord.append(float(columns[3]))
        y_coord.append(float(columns[4]))
        z_coord.append(float(columns[5]))
        facies.append(float(columns[6]))
    else:
        print(f"Skipping invalid line: {line.strip()}")
# read in Sg distribution file, exported from Petrel.
df = pd.DataFrame({
    "i_index": i_index,
    "j_index": j_index,
    "k_index": k_index,
    "x_coord": x_coord,
    "y_coord": y_coord,
    "z_coord": z_coord,
    "facies": facies
})
df["facies"] = pd.to_numeric(df["facies"], errors="coerce")
df["facies"] = df["facies"].replace(-99, np.nan)
```

```
df[["x_coord", "y_coord", "z_coord"]] = df[["x_coord", "y_coord", "z_coord"]].replace(-99, np.nan)

# Function to load a single .gslib file
def load_gslib(file_path):
    with open(file_path, "r") as infile:
        lines = infile.readlines()
    # Skip header lines
    data_lines = lines[3:]  # Skip the first two lines (header)
    # Convert the remaining lines into floats
    data = [float(line.strip()) for line in data_lines if line.strip()]
    return data

gslib_files = {
    "Sg2024.gslib": "Sg2024",
    "Sg2030.gslib": "Sg2030",
    "Sg2040.gslib": "Sg2040",
    "Sg2050.gslib": "Sg2050",
    "Sg2060.gslib": "Sg2060",
    "Sg2070.gslib": "Sg2070",
    "Sg2080.gslib": "Sg2080",
}

# Create a dictionary to store the properties
propertiesSg = {}
```

```
# Load each .gslib file and store its values in the dictionary
for file_path, column_name in gslib_files.items():
    propertiesSg[column_name] = load_gslib(file_path)

# Combine the properties with the existing DataFrame
for column_name, values in propertiesSg.items():
    df[column_name] = values
    df[column_name] = pd.to_numeric(df[column_name], errors="coerce")
    df[column_name] = df[column_name].replace(-99, np.nan)

# Check the resulting DataFrame
print(df.head())
----
```

```
+*In[ ]:*+
[source, ipython3]
----
ik1 = 1  # First layer index
ik2 = 1  # Second layer index

# Filter records for each layer
```

```
layer1_property = "y_coord"
layer2_property = "facies"

plot_two_layers(
   layer1_records=df[df["k_index"] == ik1],
   layer2_records=df[df["k_index"] == ik2],
   plt_title = 'Petrel data: df',
   layer1_title=f"{layer1_property} for k_index = {ik1}",
   layer2_title=f"{layer2_property} for k_index = {ik2}",
   property_1=layer1_property,property_2=layer2_property, syn=False
)
```

+*In[ ]:*+

[source, ipython3]

```
ik1 = 16 # First layer index
ik2 = 16  # Second layer index

layer1_property = "facies"
```

```
layer2_property = "Sg2030"

plot_two_layers(
    layer1_records=df[df["k_index"] == ik1],
    layer2_records=df[df["k_index"] == ik2],
    plt_title = 'Petrel data: df',
    layer1_title=f"{layer1_property} for k_index = {ik1}",
    layer2_title=f"{layer2_property} for k_index = {ik2}",
    property_1=layer1_property,property_2=layer2_property, syn=False
)
----
```

+*In[ ]:*+

[source, ipython3]

```
----
####################################
########## impute x, y ,z coordinates
####################################

def interpolate_coordinates_2d(known_records, missing_records):
    # Create a copy to store interpolated values
    interpolated_records = missing_records.copy()
```

```
# Iterate over unique k_index (layers)
i=0
for k in known_records["k_index"].unique():
    # Filter known and missing records for this layer
    known_layer = known_records[known_records["k_index"] == k]
    missing_layer = missing_records[missing_records["k_index"] == k]

    if known_layer.empty or missing_layer.empty:
        continue  # Skip if there are no records for this layer

    # Prepare input points and values for interpolation
    known_points = known_layer[["i_index", "j_index"]].values
    x_values = known_layer["x_coord"].values
    y_values = known_layer["y_coord"].values
    z_values = known_layer["z_coord"].values

    # Points where values are missing
    missing_points = missing_layer[["i_index", "j_index"]].values

    # Perform linear interpolation for x_coord
    interpolated_x = griddata(
        points=known_points,
        values=x_values,
        xi=missing_points,
            method="linear"
        )

        # Perform linear interpolation for y_coord
        interpolated_y = griddata(
            points=known_points,
            values=y_values,
            xi=missing_points,
            method="linear"
        )

        # Perform linear interpolation for z_coord
        interpolated_z = griddata(
            points=known_points,
            values=z_values,
            xi=missing_points,
            method="linear"
        )

        # Store interpolated values in the missing records
        interpolated_records.loc[interpolated_records["k_index"] == k, "x_coord"] =
interpolated_x
        interpolated_records.loc[interpolated_records["k_index"] == k, "y_coord"] =
interpolated_y
```

```
            interpolated_records.loc[interpolated_records["k_index"] == k, "z_coord"] =
interpolated_z
            print(f'layer {k} fills {len(interpolated_x)} values')
            i=i+1

        return interpolated_records, known_points, known_layer, interpolated_x

    known_records = df[df[["x_coord", "y_coord", "z_coord"]].notna().all(axis=1)]
    known_coords = known_records[["i_index", "j_index", "k_index"]].values
    known_values = known_records[["x_coord", "y_coord", "z_coord"]].values
    missing_records = df[df[["x_coord", "y_coord", "z_coord"]].isna().any(axis=1)]
    missing_coords = missing_records[["i_index", "j_index", "k_index"]].values
    print("Shape of all records:", df.shape)  # (18065200, 7)
    print("Shape of known records:", known_records.shape)  # (18065200, 7)
    print("Shape of missing records:", missing_records.shape)  # (21200, 7)
    interpolated_missing_records,known_points, known_layer, interpolated_x =
interpolate_coordinates_2d(known_records, missing_records)

    # Combine known and interpolated records if needed
    complete_xyzrecords = pd.concat([known_records,
interpolated_missing_records]).sort_values(by=["k_index", "j_index", "i_index"])
    ----
```

+*In[ ]:*+

```
[source, ipython3]
----
####################################
############## impute facies #######
####################################

def fill_missing_values_layerwise(
    known_records, missing_records, property_name, method="linear"
):
    """
    Fill missing values for a given property in a layer-wise manner using 2D
interpolation.

    Parameters:
        known_records (DataFrame): DataFrame containing known values for the
property.
        missing_records (DataFrame): DataFrame containing missing values for the
property.
        property_name (str): The name of the property column to fill (e.g., "x_coord",
"y_coord", "z_coord", "facies").
        method (str): Interpolation method ("linear" or "nearest"). Default is "linear".

    Returns:
        DataFrame: The missing_records DataFrame with the interpolated property
values filled in.
```

```
"""
filled_values = []

for k in known_records["k_index"].unique():
    # Filter known and missing records for this layer
    known_layer = known_records[known_records["k_index"] == k]
    missing_layer = missing_records[missing_records["k_index"] == k]

    if known_layer.empty or missing_layer.empty:
        continue  # Skip if there are no records for this layer

    # Prepare input points and values for interpolation
    known_points = known_layer[["i_index", "j_index"]].values
    property_values = known_layer[property_name].values

    # Points where values are missing
    missing_points = missing_layer[["i_index", "j_index"]].values

    # Perform interpolation
    interpolated_values = griddata(
        points=known_points,
        values=property_values,
        xi=missing_points,
        method=method
    )
```

```
        # Append results as a DataFrame for this layer
        filled_layer = missing_layer.copy()
        filled_layer[property_name] = interpolated_values
        filled_values.append(filled_layer)

    # Combine all filled layers into a single DataFrame
    filled_records = pd.concat(filled_values, ignore_index=True)
    return filled_records

known_records = df[df[["facies"]].notna().all(axis=1)]
known_coords = known_records[["i_index", "j_index", "k_index"]].values
known_values = known_records[["facies"]].values
missing_records = df[df[["facies"]].isna().any(axis=1)]
missing_coords = missing_records[["i_index", "j_index", "k_index"]].values
print("Shape of all records:", df.shape)  # (18065200, 7)
print("Shape of known records:", known_records.shape)  # (18065200, 7)
print("Shape of missing records:", missing_records.shape)  # (21200, 7)
filled_missing_facies = fill_missing_values_layerwise(
    known_records=known_records,
    missing_records=missing_records,
    property_name="facies",
    method="nearest"
)
```

```
complete_faciesrecords = pd.concat([known_records, filled_missing_facies]).sort_values(by=["k_index", "j_index", "i_index"])
----



+*In[ ]:*+
[source, ipython3]
----
####################################
############## combine to get imputed facies
####################################

# Merge the two DataFrames, prioritizing facies from `complete_faciesrecords`
complete_data = pd.merge(
    complete_xyzrecords,
    complete_faciesrecords[["i_index", "j_index", "k_index", "facies"]],
    on=["i_index", "j_index", "k_index"],
    how="left"
)

# Fill missing facies values (from facies_x) with facies_y
complete_data["facies"] = complete_data["facies_x"].combine_first(complete_data["facies_y"])

# Drop the temporary columns facies_x and facies_y
```

```
complete_data.drop(columns=["facies_x", "facies_y"], inplace=True)

# Check for remaining NaN values in the `facies` column
nan_facies_count = complete_data["facies"].isna().sum()
print(f"Number of rows with missing facies after merge: {nan_facies_count}")

----



+*In[ ]:*+
[source, ipython3]
----
#####################################
############## impute Sg #######
#####################################

properties = [ "Sg2024", "Sg2030", "Sg2040", "Sg2050","Sg2060","Sg2070"]
for property_name in properties:
    known_records = df[df[[property_name]].notna().all(axis=1)]
    missing_records = df[df[[property_name]].isna().any(axis=1)]

    method = "nearest"
    filled_missing_values = fill_missing_values_layerwise(
        known_records=known_records,
        missing_records=missing_records,
            property_name=property_name,
            method=method
        )

        # Combine known and interpolated records for the current property
        complete_property_records                =                pd.concat([known_records,
filled_missing_values]).sort_values(
            by=["k_index", "j_index", "i_index"]
        )

        complete_data = pd.merge(
            complete_data,
            complete_property_records[["i_index",            "j_index",            "k_index",
property_name]],
            on=["i_index", "j_index", "k_index"],
            how="left"
        )
    ----




    +*In[ ]:*+
    [source, ipython3]
    ----
```

```
old_zone_intervals = {
    "L_Miocene_Shale - L_Miocene_A": (1, 10),
    "L_Miocene_A - AMPB_SAND": (11, 30),
    "AMPB_SAND - L_Miocene_B": (31,40),
    "L_Miocene_B - L_Miocene_C": (41, 60),
    "L_Miocene_C - TOP_LMIO_INJ_ZONE": (61,70),
    "TOP_LMIO_INJ_ZONE - LMIO_INJ_5": (71, 90),
    "LMIO_INJ_5 - LMIO_INJ_4": (91,110),
    "LMIO_INJ_4 - LMIO_INJ_3": (111,130),
    "LMIO_INJ_3 - LMIO_INJ_6": (131,150),
    "LMIO_INJ_6 - LMIO_INJ_2": (151,170),
    "LMIO_INJ_2 - Anahuac": (171, 190),
    "Anahuac - Anahuac_Sand_Top": (191, 200)
}
----
```

+*In[ ]:*+

[source, ipython3]

```
----
# to find out z ranges for last zone:
k1Lastzone = old_zone_intervals["Anahuac - Anahuac_Sand_Top"][0]
k2Lastzone = old_zone_intervals["Anahuac - Anahuac_Sand_Top"][1]
z1 = df[df["k_index"] == k1Lastzone]["z_coord"].agg(["min", "max"])
print(f"z_coord range for k_index = {k1Lastzone}:", z1.to_dict())
```

```
z2 = df[df["k_index"] == k2Lastzone]["z_coord"].agg(["min", "max"])
print(f"z_coord range for k_index = {k2Lastzone}:", z2.to_dict())

----



+*In[ ]:*+
[source, ipython3]
----
klast=420
zdfss = dfss[dfss["k_index"] == klast]["z_coord"].agg(["min", "max"])
print(f"z_coord range for k_index = {klast}:", zdfss.to_dict())
----




+*In[ ]:*+
[source, ipython3]
----
# Define LAS files
las_files = ["welldata/well0.las", "welldata/well1.las", "welldata/well2.las"]
# facies: SP, GR, GR
strPor = ['XPORT']     # Porosity
strRt = ['XRESD']      # Resistivity
```

```
strSP = ['XSP']        # SP
strVclay = ['XVCL']    # VClay
strVp = ['VP']         # Velocity
strVs = ['VS']
strFaciesPrefix = "FACIES_"  # Facies log starts with this prefix
strRhob = ['XRHOB']

dfs = [read_las(file) for file in las_files]

# Merge all data into a single dataframe
las_df = pd.concat(dfs, ignore_index=False)

print(las_df.shape)
# Find common logs across all wells
common_logs = set(dfs[0].columns)
for dfi in dfs[1:]:
   common_logs &= set(dfi.columns)
common_logs.discard("DEPT")  # Keep depth separate

print("Common Logs Across Wells:", common_logs)

predictors = strPor + strRt + strSP + strVclay  # Predictor logs
print(predictors)
----
```

+*In[ ]:*+

[source, ipython3]

```
----
print(dfs[0]['DEPT'].min())
print(dfs[0]['DEPT'].max())
print(dfs[1]['DEPT'].min())
print(dfs[1]['DEPT'].max())
print(dfs[2]['DEPT'].min())
print(dfs[2]['DEPT'].max())
----
```

+*In[ ]:*+

[source, ipython3]

```
----
las_file = las_files[1]
log_names = ['XRHOB', 'XPORT', 'XRESD', 'XSP', 'XVCL','XVLIME','XVSAND','XDT','XDTS','XPEF', 'XNPHIL','XTHOR', 'FACIES_SELECTED', 'ZONELOG','VP','VS']  # Specify logs to plot
plot_well_logs(las_files[0], log_names,'well0 (depth hided)')
plot_well_logs(las_files[1], log_names,'well1 (depth hided)')
```

+*In[ ]:*+

[source, ipython3]

----

plot_well_logs(las_files[2], log_names,'well2 (depth hided)')

+*In[ ]:*+

[source, ipython3]

----

zone_log = "ZONELOG" # Name of the geological zone log

+*In[ ]:*+

[source, ipython3]

----

zone_log = "Zonelog" # Name of the geological zone log

```
predictors = [ 'XPORT', 'XRESD', 'XVCL','XVLIME','XDT','XDTS','XPEF',
'XNPHIL','XTHOR']

lasfile = las_files[2]
df = read_las(lasfile)
zone=0
selected_log=strRhob
zone_data = df[df['ZONELOG'] == zone].dropna(subset=['XRHOB'] + predictors)

X = zone_data[predictors].copy(deep=True)
y = zone_data['XRHOB'].copy(deep=True)

rf = RandomForestRegressor(
    n_estimators=100,   # Number of trees in the forest
    max_depth=None,      # No maximum depth (fully grown trees)
    min_samples_split=2, # Minimum samples to split an internal node
    min_samples_leaf=1,  # Minimum samples per leaf
    random_state=42,     # Set a random seed for reproducibility
    n_jobs=-1           # Use all CPU cores for training
)

rf = RandomForestRegressor(n_estimators=50, max_depth=10,
min_samples_split=4, min_samples_leaf=2,random_state=42, n_jobs=-1)

rf.fit(X,y)
```

```
feature_importances = rf.feature_importances_
feature_names = X.columns  # Features used in training

# Sort features by importance
sorted_idx = np.argsort(feature_importances)
plt.figure(figsize=(10, 5))
plt.barh(range(len(sorted_idx)), feature_importances[sorted_idx], align="center")
plt.yticks(range(len(sorted_idx)), np.array(feature_names)[sorted_idx])
plt.xlabel("Feature Importance")
plt.title("Feature Importance in Predicting XRHOB")
plt.show()

----



+*In[ ]:*+
[source, ipython3]
----
predictors = ['XPORT',  'XVCL', 'XVLIME']
# updatedRhobLas = predictLog(strRhob, "FACIES_", zone_log, las_files, predictors, plotit=False, normalize=True, method='RF')
[updatedRhobLas, mean_std_Rhob] = predictLog(strRhob, "FACIES_", zone_log, las_files, predictors, plotit=True, normalize=True, method='PN',  poly_degree=1)
----
```

+*In[ ]:*+

[source, ipython3]

```
zone_log = "Zonelog"  # Name of the geological zone log
# # predictors = strPor + strRt +  strVclay  # Predictor logs
predictors = [  'XPORT', 'XRESD', 'XVCL','XVLIME','XRHOB']

lasfile = las_files[2]
df = read_las(lasfile)
zone=0
selected_log=strRhob
zone_data = df[df['ZONELOG'] == zone].dropna(subset=['VP'] + predictors)
# zone_data = las_df[las_df[zone_log] == zone].dropna(subset=[''] + common_logs)
X = zone_data[predictors].copy(deep=True)
y = zone_data['VP'].copy(deep=True)

rf = RandomForestRegressor(
    n_estimators=100,   # Number of trees in the forest
    max_depth=None,      # No maximum depth (fully grown trees)
    min_samples_split=2, # Minimum samples to split an internal node
    min_samples_leaf=1,  # Minimum samples per leaf
    random_state=42,     # Set a random seed for reproducibility
    n_jobs=-1            # Use all CPU cores for training
)

rf = RandomForestRegressor(n_estimators=50, max_depth=10,
min_samples_split=4, min_samples_leaf=2,random_state=42, n_jobs=-1)

rf.fit(X,y)

feature_importances = rf.feature_importances_
feature_names = X.columns  # Features used in training

# Sort features by importance
sorted_idx = np.argsort(feature_importances)
plt.figure(figsize=(10, 5))
plt.barh(range(len(sorted_idx)), feature_importances[sorted_idx], align="center")
plt.yticks(range(len(sorted_idx)), np.array(feature_names)[sorted_idx])
plt.xlabel("Feature Importance")
plt.title("Feature Importance in Predicting VP")
plt.show()

----
```

```
+*In[ ]:*+
[source, ipython3]
----
zone_log = "ZONELOG"  # Name of the geological zone log
predictors = [ 'XPORT', 'XRESD', 'XVCL','XVLIME','XRHOB']
[updatedVpLas, mean_std_Vp] = predictLog(strVp, "FACIES_", zone_log, las_files, predictors, plotit=True, normalize=True, method='PN',  poly_degree=1)

----




+*In[ ]:*+
[source, ipython3]
----
zone_log = "Zonelog"  # Name of the geological zone log
# # predictors = strPor + strRt +  strVclay  # Predictor logs
predictors = ['XPORT', 'XRESD', 'XVCL', 'XVLIME', 'XPEF', 'XNPHIL', 'XTHOR','XGR','XPOTA','XRHOB','XURA','XVSAND']

lasfile = las_files[2]
df = read_las(lasfile)
```

```
zone=0
selected_log=strRhob
zone_data = df[df['ZONELOG'] == zone].dropna(subset=['VS'] + predictors)

X = zone_data[predictors].copy(deep=True)
y = zone_data['VS'].copy(deep=True)

rf = RandomForestRegressor(
    n_estimators=100,   # Number of trees in the forest
    max_depth=None,      # No maximum depth (fully grown trees)
    min_samples_split=2, # Minimum samples to split an internal node
    min_samples_leaf=1,  # Minimum samples per leaf
    random_state=42,     # Set a random seed for reproducibility
    n_jobs=-1           # Use all CPU cores for training
)

rf = RandomForestRegressor(n_estimators=50, max_depth=10,
min_samples_split=4, min_samples_leaf=2,random_state=42, n_jobs=-1)

rf.fit(X,y)

feature_importances = rf.feature_importances_
feature_names = X.columns  # Features used in training

# Sort features by importance
```

```
sorted_idx = np.argsort(feature_importances)
plt.figure(figsize=(10, 5))
plt.barh(range(len(sorted_idx)), feature_importances[sorted_idx], align="center")
plt.yticks(range(len(sorted_idx)), np.array(feature_names)[sorted_idx])
plt.xlabel("Feature Importance")
plt.title("Feature Importance in Predicting VS")
plt.show()
```

+*In[ ]:*+

[source, ipython3]

```
zone_log = "ZONELOG"  # Name of the geological zone log
strVs = "VS"
predictors = ['XPORT', 'XRESD', 'XVCL', 'XVLIME', 'XPEF', 'XNPHIL', 'XTHOR','XGR','XPOTA','XRHOB','XURA','XVSAND']
[updatedVsLas, mean_std_Vs] = predictLog(strVs, "FACIES_", zone_log, las_files, predictors, plotit=True, normalize=True, method='PN',  poly_degree=1)
```

```
+*In[ ]:*+
[source, ipython3]
----
####################################
###### Fit a polynomial for each facies, removing extreme values (outliers)
####################################
facies_col = [col for col in las_df.columns if col.startswith("FACIES_")][0]
facies_list = las_df[facies_col].unique()

facies_fit = {}

for facies_val in facies_list:

    subset = las_df[(las_df[facies_col] == facies_val) & (las_df['DEPT']<10000)].copy()
    print(f"{facies_val},{len(subset)}")

    # Skip if too few points
    if len(subset) < 10:
        continue

    x = subset["DEPT"].values
    y = subset["VP"].values
```

```
        # Remove top and bottom 1% of velocity values to exclude outliers
        lower_bound = np.percentile(y, 1)  # Bottom 0.5%
        upper_bound = np.percentile(y, 99) # Top 99.5%
        mask = (y >= lower_bound) & (y <= upper_bound)

        x_filtered = x[mask]
        y_filtered = y[mask]

        if len(x_filtered) < 10:
            print(f"⚠ Facies {facies_val} has too few data points after filtering,
skipping.")
            continue

        coefs = np.polyfit(x_filtered, y_filtered, deg=1)
        poly_func = np.poly1d(coefs)

        # Compute residuals -> measure scatter around the fitted curve
        y_pred = poly_func(x_filtered)
        residuals = y_filtered - y_pred
        residual_std = np.std(residuals)

        facies_fit[facies_val] = (coefs, residual_std)

    print("✅ Polynomial fits updated after filtering extreme values.")
```

+*In[ ]:*+

[source, ipython3]

```
colors = sns.color_palette("tab10", n_colors=len(facies_list))
plt.figure(figsize=(8, 6))

for i, facies_val in enumerate(facies_list):
    subset = las_df[las_df[facies_col] == facies_val]
    if facies_val not in facies_fit:
        continue  # Skip if not enough points

    coefs, residual_std = facies_fit[facies_val]
    poly_func = np.poly1d(coefs)

    # Plot scatter for this facies
    plt.scatter(subset["DEPT"],      subset["VP"],      alpha=0.2,      color=colors[i],
label=f"Facies {facies_val} data")

    # Plot polynomial trend line
    depth_range = np.linspace(subset["DEPT"].min(), subset["DEPT"].max(), 100)
    vp_fit = poly_func(depth_range)
```

```
            plt.plot(depth_range,    vp_fit,    color=colors[i],    linewidth=2,    label=f"Facies
{facies_val} fit")

            # ✅ Improved Error Bar Visibility
            if facies_val in mean_std_Vp.index:
                vp_mean = mean_std_Vp.loc[facies_val, "Mean"]
                vp_std = mean_std_Vp.loc[facies_val, "Std"]

                # Find depth where poly trend ≈ vp_mean
                depth_best = depth_range[np.abs(vp_fit - vp_mean).argmin()]

                # Plot error bar with strong visibility
                plt.errorbar(depth_best, vp_mean, yerr=vp_std, fmt='s', color="black",
                             markersize=10, capsize=5, capthick=3, elinewidth=2,
                             markeredgecolor="white",     markeredgewidth=2,     label=f"Facies
{facies_val} Mean ± Std")

        plt.gca().invert_yaxis()  # if depth increases downward
        plt.xlabel("Depth (m)")
        plt.ylabel("Vp (m/s)")
        plt.title("Vp vs. Depth Polynomial Fit per Facies with Mean & Std")
        plt.legend()
        plt.show()

        ----
```

+*In[ ]:*+

[source, ipython3]

```
----
######################################
############### use this ############
######################################

facies_fitVs = {}
poly_order = 1  # Polynomial degree
las_Vs = las_files[2]
las_dfVs = lasio.read(las_Vs).df().reset_index()
las_dfVs = las_dfVs.dropna(subset=["VS"])
facies_col = [col for col in las_dfVs.columns if col.startswith("FACIES_")][0]
facies_list = las_dfVs[facies_col].unique()

for facies_val in facies_list:

    subset = las_dfVs[las_dfVs[facies_col] == facies_val].copy()

    # Skip if too few points
    if len(subset) < 10:
        continue
    print(f"Facies {facies_val} has {len(subset)} points")

    x = subset["DEPT"].values
    y = subset["VS"].values

    x_filtered = x
    y_filtered = y

    if len(x_filtered) < 10:
        print(f"⚠️ Facies {facies_val} has too few data points after filtering, skipping.")
        continue

    # Fit a polynomial of chosen order
    coefs = np.polyfit(x_filtered, y_filtered, deg=poly_order)
    poly_func = np.poly1d(coefs)

    # Compute residuals -> measure scatter around the fitted curve
    y_pred = poly_func(x_filtered)
    residuals = y_filtered - y_pred
    residual_std = np.std(residuals)

    # Store in dictionary
    facies_fitVs[facies_val] = (coefs, residual_std)
```

```
# print("✅ Polynomial fits updated after filtering extreme values.")
facies_fitVs
----



+*In[ ]:*+
[source, ipython3]
----
colors = sns.color_palette("tab10", n_colors=len(facies_list))
plt.figure(figsize=(8, 6))

for i, facies_val in enumerate(facies_list):
    subset = las_dfVs[las_dfVs[facies_col] == facies_val]
    if facies_val not in facies_fitVs:
        continue  # Skip if not enough points

    coefs, residual_std = facies_fitVs[facies_val]
    poly_func = np.poly1d(coefs)

    # Plot scatter for this facies
    plt.scatter(subset["DEPT"],    subset["VS"],    alpha=0.2,    color=colors[i],
label=f"Facies {facies_val} data")
```

```
        # Plot polynomial trend line
        depth_range = np.linspace(subset["DEPT"].min(), subset["DEPT"].max(), 100)
        vp_fit = poly_func(depth_range)
        plt.plot(depth_range, vp_fit, color=colors[i], linewidth=2, label=f"Facies
{facies_val} fit")

        # ✅ Improved Error Bar Visibility
        if facies_val in mean_std_Vs.index:
            vp_mean = mean_std_Vs.loc[facies_val, "Mean"]
            vp_std = mean_std_Vs.loc[facies_val, "Std"]

            # Find depth where poly trend ≈ vp_mean
            depth_best = depth_range[np.abs(vp_fit - vp_mean).argmin()]

            # Plot error bar with strong visibility
            plt.errorbar(depth_best, vp_mean, yerr=vp_std, fmt='s', color="black",
                         markersize=10, capsize=5, capthick=3, elinewidth=2,
                         markeredgecolor="white", markeredgewidth=2, label=f"Facies
{facies_val} Mean ± Std")

    plt.gca().invert_yaxis()  # if depth increases downward
    plt.xlabel("Depth (m)")
    plt.ylabel("Vs (m/s)")
    plt.title("Vs vs. Depth Polynomial Fit per Facies with Mean & Std")
    # plt.legend()
```

```
plt.show()
----




+*In[ ]:*+
[source, ipython3]
----
####################################
##############  predict Vp
####################################

# Suppose facies_fit is from the polynomial fitting:
#   facies_fit[facies_val] = (poly_coefs, residual_std)
# Where:
#   poly_coefs -> array of polynomial coefficients for that facies
#   residual_std -> float standard deviation of residual for that facies

# 1. Create a dictionary of polynomial functions for quick evaluation
poly_dict = {}
std_dict = {}
for f, (coefs, std) in facies_fit.items():
    poly_dict[f] = np.poly1d(coefs)  # polynomial function
    std_dict[f] = std                # store std separately
    print(coefs)
    print(std)

# 2. Compute DepthPos if needed (assuming negative z => positive depth)
df["DepthPos"] = -df["z_coord"]  # only if z_coord is negative downward

# 3. Evaluate the polynomial trend for each cell: VpTrend=Vp(depth, facies)
df["VpTrend"] = np.nan
for f in poly_dict.keys():
    mask = df["facies"] == f
    df.loc[mask, "VpTrend"] = poly_dict[f](df.loc[mask, "DepthPos"])

# 4. Generate random Vp around that trend
#    For each facies, add random noise ~ N(0, std_fac)
df["VpRandom"] = np.nan
for f in poly_dict.keys():
    mask = df["facies"] == f
    noise = np.random.normal(loc=0.0, scale=std_dict[f], size=mask.sum())
    df.loc[mask, "VpRandom"] = df.loc[mask, "VpTrend"] + noise
```

+*In[ ]:*+

```
[source, ipython3]
----
df.head()
----




+*In[ ]:*+
[source, ipython3]
----
####################################
############## predict Vs
####################################

# Suppose facies_fit is from your polynomial fitting:
#   facies_fit[facies_val] = (poly_coefs, residual_std)
# Where:
#   poly_coefs -> array of polynomial coefficients for that facies
#   residual_std -> float standard deviation of residual for that facies

# 1. Create a dictionary of polynomial functions for quick evaluation
poly_dict = {}
std_dict = {}
for f, (coefs, std) in facies_fitVs.items():
    poly_dict[f] = np.poly1d(coefs)  # polynomial function
    std_dict[f] = std               # store std separately
    print(coefs)
    print(std)

# 2. Compute DepthPos if needed (assuming negative z => positive depth)
df["DepthPos"] = -df["z_coord"]  # only if z_coord is negative downward

# 3. Evaluate the polynomial trend for each cell: VpTrend=Vp(depth, facies)
df["VsTrend"] = np.nan
for f in poly_dict.keys():
    mask = df["facies"] == f
    df.loc[mask, "VsTrend"] = poly_dict[f](df.loc[mask, "DepthPos"])

# 4. Generate random Vp around that trend
#    For each facies, add random noise ~ N(0, std_fac)
df["VsRandom"] = np.nan
for f in poly_dict.keys():
    mask = df["facies"] == f
    # noise = np.random.normal(loc=0.0, scale=std_dict[f], size=mask.sum())
    noise = np.random.normal(loc=0.0, scale=std_dict[f] * 0.3, size=mask.sum())  #
or 0.4, 0.3, etc.

    df.loc[mask, "VsRandom"] = df.loc[mask, "VsTrend"] + noise
```

+*In[ ]:*+

[source, ipython3]

```
####################################
############## use this ########### predict Rhob
####################################
# Initialize RbRandom column
df["RbRandom"] = np.nan

# Loop through each facies and generate random density values
for f in mean_std_Rhob.index:  # Iterate through facies in mean_std_Rhob
    if f in mean_std_Rhob.index:  # Ensure facies exists in the dataset
        rb_mean = mean_std_Rhob.loc[f, "Mean"]
        rb_std = mean_std_Rhob.loc[f, "Std"]

        mask = df["facies"] == f  # Select only this facies
        noise = np.random.normal(loc=0.0, scale=rb_std, size=mask.sum())      #
Generate random noise
        df.loc[mask, "RbRandom"] = rb_mean + noise  # Assign random values
```

print("✅ RbRandom generated for each facies.")

+*In[ ]:*+

[source, ipython3]

----

df.head()

+*In[ ]:*+

[source, ipython3]

----

complete_data.head()

+*In[ ]:*+

[source, ipython3]

```
----

ik1 = 1  # First layer index
ik2 = 1  # Second layer index
known_records = df[df[["facies"]].notna().all(axis=1)]
known_coords = known_records[["i_index", "j_index", "k_index"]].values
known_values = known_records[["facies"]].values
# Filter records for each layer
layer1_records = known_records[known_records["k_index"] == ik1]
layer2_records = known_records[known_records["k_index"] == ik2]
layer1_property = "y_coord"
layer2_property = "facies"
# Plot facies for the two layers
plot_two_layers(
   layer1_records=layer1_records,
   layer2_records=layer2_records,
   plt_title='Petrel data',
   layer1_title=f"{layer1_property} for k_index = {ik1}",
   layer2_title=f"{layer2_property} for k_index = {ik2}",
   property_1=layer1_property,property_2=layer2_property
)

# ik1 = 2  # First layer index
# ik2 = 2  # Second layer index
layer1_records = complete_data[complete_data["k_index"] == ik1]
```

```
layer2_records = complete_data[complete_data["k_index"] == ik2]
plot_two_layers(
    layer1_records=layer1_records,
    layer2_records=layer2_records,
    plt_title='complete_data',
    layer1_title=f"{layer1_property} for k_index = {ik1}",
    layer2_title=f"{layer2_property} for k_index = {ik2}",
    property_1=layer1_property,property_2=layer2_property
)
```

+*In[ ]:*+

[source, ipython3]

```
# Filter the points at the upper-left and lower-left corners
upper_left = complete_data[(complete_data["i_index"] == 1) & (complete_data["j_index"] == 1)]
lower_left = complete_data[(complete_data["i_index"] == 1) & (complete_data["j_index"] == 314)]

# Filter the points at the upper-right and lower-right corners
```

```
upper_right = complete_data[(complete_data["i_index"] == complete_data["i_index"].max()) & (complete_data["j_index"] == 1)]
lower_right = complete_data[(complete_data["i_index"] == complete_data["i_index"].max()) & (complete_data["j_index"] == 314)]

# Extract x_coord values
x_upper_left = upper_left["x_coord"].values[0] if not upper_left.empty else None
x_lower_left = lower_left["x_coord"].values[0] if not lower_left.empty else None

x_upper_right = upper_right["x_coord"].values[0] if not upper_right.empty else None
x_lower_right = lower_right["x_coord"].values[0] if not lower_right.empty else None

# Calculate differences
diff_left = None if x_upper_left is None or x_lower_left is None else abs(x_upper_left - x_lower_left)
diff_right = None if x_upper_right is None or x_lower_right is None else abs(x_upper_right - x_lower_right)

# Output results
print(f"Upper-Left x_coord: {x_upper_left}, Lower-Left x_coord: {x_lower_left}, Difference: {diff_left}")
print(f"Upper-Right x_coord: {x_upper_right}, Lower-Right x_coord: {x_lower_right}, Difference: {diff_right}")
```

+*In[ ]:*+

[source, ipython3]

```
def check_rhomboid(data, dx, dy):
    """
    Check if the grid forms a perfect rectangle or a rhomboid.

    Parameters:
        data (DataFrame): A DataFrame containing x_coord and y_coord columns.
        dx (float): The regular grid spacing in the x direction.
        dy (float): The regular grid spacing in the y direction.

    Returns:
        bool: True if the grid forms a rhomboid, False if it forms a rectangle.
    """
    # Extract unique x and y coordinates
    x_coords = np.sort(data["x_coord"].unique())
    y_coords = np.sort(data["y_coord"].unique())

    # Calculate the range and ensure it's divisible by dx/dy
    x_range = np.ptp(x_coords)  # Peak-to-peak range of x coordinates
    y_range = np.ptp(y_coords)  # Peak-to-peak range of y coordinates

    # Check if the ranges are divisible by dx and dy
    is_rhomboid_x = not np.isclose(x_range % dx, 0, atol=1e-6)
    is_rhomboid_y = not np.isclose(y_range % dy, 0, atol=1e-6)

    if is_rhomboid_x or is_rhomboid_y:
        print("The grid forms a rhomboid due to irregular spacing.")
        return True
    else:
        print("The grid forms a perfect rectangle.")
        return False

check_rhomboid(complete_data,dx=250, dy=250)

----
```

+*In[ ]:*+

[source, ipython3]

```
        # Define fixed grid spacing
        dx, dy, dz = 250, 250, 5
        # Get the bounding box of the data
        x_min, x_max = complete_data['x_coord'].min(), complete_data['x_coord'].max()
        y_min, y_max = complete_data['y_coord'].min(), complete_data['y_coord'].max()
        z_min, z_max = complete_data['z_coord'].min(), complete_data['z_coord'].max()

        # Create a 3D meshgrid for the regular grid
        # Generate regular x, y grid
        x_regular = np.arange(x_min, x_max + dx, dx)
        y_regular = np.arange(y_min, y_max + dy, dy)

        # Create meshgrid for the horizontal plane
        x_grid, y_grid = np.meshgrid(x_regular, y_regular, indexing='ij')
        # Interpolate z_surface to the regular grid

        # Calculate the surface data
        surface_data                    =                    complete_data.groupby(['x_coord',
'y_coord'])['z_coord'].max().reset_index()
        surface_data.rename(columns={'z_coord': 'z_surface'}, inplace=True)
        btm_data                    =                    complete_data.groupby(['x_coord',
'y_coord'])['z_coord'].min().reset_index()
        btm_data.rename(columns={'z_coord': 'z_bottom'}, inplace=True)
        print(btm_data.shape)
        # 1. Calculate thickness map
```

```
    surface_btm_data = pd.merge(surface_data, btm_data, on=['x_coord', 'y_coord'])
    surface_btm_data['thickness_map']    =    surface_btm_data['z_surface']    -
surface_btm_data['z_bottom']
    print(surface_btm_data.head())  # Display a few rows of the combined data
    print(surface_btm_data['thickness_map'].describe())  # Show thickness statistics

    # Interpolate z_surface to the regular grid
    z_surface_grid = griddata(
        points=surface_data[['x_coord', 'y_coord']].values,
        values=surface_data['z_surface'].values,
        xi=(x_grid, y_grid),
        method='linear'  # Interpolate to regular x, y grid
    )
    # 2. Determine the thickest point and compute nz
    max_thickness = surface_btm_data['thickness_map'].max()
    nz = int(np.ceil(max_thickness / dz))  # Number of layers required
    print(f"Maximum Thickness: {max_thickness}, Number of Layers: {nz}")

    # 3. Generate 3D grid
    z_layers = np.arange(0, nz * dz, dz)  # Vertical grid levels
    z_3d_grid = z_surface_grid[:, :, np.newaxis] - z_layers[np.newaxis, np.newaxis, :]
# Extend depth-wise
    print(z_3d_grid.shape)

    nz = z_3d_grid.shape[2]  # Number of vertical layers
```

```
# Create 3D grids for x and y by broadcasting
x_3d_grid = x_grid[:, :, np.newaxis].repeat(nz, axis=2)  # Extend along z-axis
y_3d_grid = y_grid[:, :, np.newaxis].repeat(nz, axis=2)

# Combine grids into a DataFrame
grid_points = pd.DataFrame({
    'x_coord': x_3d_grid.ravel(),
    'y_coord': y_3d_grid.ravel(),
    'z_coord': z_3d_grid.ravel()
})

# Print details for confirmation
print(f"Regular grid created with shape: {x_3d_grid.shape}")
print(f"Number of grid points: {grid_points.shape[0]}")

# Optional: Save the grid points as a CSV file
grid_points.to_csv("regular_grid_with_surface.csv", index=False)

# Output summary
grid_points.head()
----
```

+*In[ ]:*+

```
[source, ipython3]
----
# make sure it is constant dx, dy, dz

def check_constant_spacing(x_grid, y_grid, z_3d_grid):
    """
    Check if the grid has constant spacing in x, y, and z directions.

    Parameters:
        x_grid (ndarray): 2D array of x-coordinates for the grid.
        y_grid (ndarray): 2D array of y-coordinates for the grid.
        z_3d_grid (ndarray): 3D array of z-coordinates for the grid.

    Returns:
        bool: True if dx, dy, and dz are constant; False otherwise.
        dict: Contains the values of dx, dy, dz, and flags for each direction.
    """
    # Calculate spacings in x, y, and z directions
    dx_values = np.diff(x_grid[:, 0], axis=0)  # Differences along x-axis
    dy_values = np.diff(y_grid[0, :], axis=0)  # Differences along y-axis
    dz_values = np.diff(z_3d_grid[0, 0, :], axis=0)  # Differences along z-axis
(vertical)

    # Check if the spacings are constant
    is_dx_constant = np.allclose(dx_values, dx_values[0])
    is_dy_constant = np.allclose(dy_values, dy_values[0])
    is_dz_constant = np.allclose(dz_values, dz_values[0])

    # Collect the results
    result = {
        "dx": dx_values[0] if is_dx_constant else "Variable",
        "dy": dy_values[0] if is_dy_constant else "Variable",
        "dz": dz_values[0] if is_dz_constant else "Variable",
        "is_dx_constant": is_dx_constant,
        "is_dy_constant": is_dy_constant,
        "is_dz_constant": is_dz_constant,
    }

    # Print detailed results
    print("Spacing Results:")
    print(f"dx: {result['dx']} (Constant: {result['is_dx_constant']})")
    print(f"dy: {result['dy']} (Constant: {result['is_dy_constant']})")
    print(f"dz: {result['dz']} (Constant: {result['is_dz_constant']})")

    # Return overall result
    return is_dx_constant and is_dy_constant and is_dz_constant, result

# Example Usage
is_constant, results = check_constant_spacing(x_grid, y_grid, z_3d_grid)
if is_constant:
    print("The grid has constant dx, dy, and dz.")
else:
    print("The grid does not have constant spacing.")

----
```

+*In[ ]:*+

[source, ipython3]

```
----
def plot_original_complete_data(complete_data):
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection="3d")

    # Scatter plot of the original data points
    sc = ax.scatter(
        complete_data["x_coord"],
        complete_data["y_coord"],
        complete_data["z_coord"],
        c=complete_data["facies"],
        cmap="viridis",
        alpha=0.7,
        s=5,  # Size of the points
        edgecolor="none"
    )

    # Add colorbar to represent facies values
    cbar = plt.colorbar(sc, ax=ax, shrink=0.5, pad=0.1)
    cbar.set_label("Facies")

    # Set axis labels
    ax.set_xlabel("X (ft)")
    ax.set_ylabel("Y (ft)")
    ax.set_zlabel("Z (ft)")

    ax.set_title("Facies on original grid completed data")

    plt.tight_layout()
    plt.show()

# Call the function to plot the data
plot_original_complete_data(complete_data)
```

+*In[ ]:*+

[source, ipython3]

```
# Save the 3D visualization as an image file for inspection
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Plot each vertical "slice" in the x direction
# Loop through layers of z_3d_grid for the vertical slices
for k in range(0, z_3d_grid.shape[2], 10):  # Adjust step for clarity
    ax.plot_surface(
        x_grid, y_grid, z_3d_grid[:, :, k],
        alpha=0.4, cmap='viridis', edgecolor='none'
    )



# Plot the surface
ax.plot_surface(
    x_grid, y_grid, z_surface_grid,
    alpha=0.7, cmap='terrain', edgecolor='k', rstride=5, cstride=5
)

# Set axis labels
ax.set_xlabel('X (m)')
ax.set_ylabel('Y (m)')
ax.set_zlabel('Z (m)')
ax.set_title("3D Grid with Topography (Vertical X, Y, and Horizontal Z)")
plt.show()
# Save the figure
file_path = "3D_Grid_with_Topography.png"
plt.savefig(file_path)
plt.close(fig)
```

+*In[ ]:*+

[source, ipython3]

```
def assign_layers_to_new_grid_layerwise_gpu(x_3d_grid, y_3d_grid, z_3d_grid,
complete_data, dz, debug=False):
    """
    Assign each layer in the new grid to a corresponding old layer based on depth
using GPU.
```

```
    Parameters:
        x_3d_grid, y_3d_grid, z_3d_grid (array): Regularized 3D grid coordinates
(NumPy arrays).
        complete_data (DataFrame): Original data with `x_coord`, `y_coord`,
`z_coord`, and `k_index`.
        dz (float): Vertical spacing in the new grid.
        debug (bool): Whether to print debug information.

    Returns:
        assigned_layers (array): 3D array where each layer is assigned the most
frequent old `k_index`.
    """
    # Convert grids to CuPy arrays for GPU processing
    x_3d_grid_gpu = cp.asarray(x_3d_grid)
    y_3d_grid_gpu = cp.asarray(y_3d_grid)
    z_3d_grid_gpu = cp.asarray(z_3d_grid)try

    # Convert complete_data to CuPy arrays
    x_coord_gpu = cp.asarray(complete_data["x_coord"].values)
    y_coord_gpu = cp.asarray(complete_data["y_coord"].values)
    z_coord_gpu = cp.asarray(complete_data["z_coord"].values)
    k_index_gpu = cp.asarray(complete_data["k_index"].values)

    # Initialize assigned layers
    assigned_layers = cp.full(z_3d_grid.shape[2], fill_value=-1, dtype=cp.int32)
```

```
zup_gpu = z_3d_grid_gpu + 0.5 * dz
zbtm_gpu = z_3d_grid_gpu - 0.5 * dz

for k in tqdm(range(z_3d_grid.shape[2]), desc="Processing layers"):
    start_time = time.time()  # Start timing for this layer
    layer_assignments = []try

    if debug:
        print(f'Processing new grid layer: {k}')

    # Perform GPU filtering for all points in parallel
    for i in range(z_3d_grid.shape[0]):  # x-dimension
        for j in range(z_3d_grid.shape[1]):  # y-dimension
            z_value = z_3d_grid_gpu[i, j, k]

            # Select vertical points for the current (i, j)
            mask = (x_coord_gpu == x_3d_grid_gpu[i, j, 0]) & \
                   (y_coord_gpu == y_3d_grid_gpu[i, j, 0]) & \
                   (z_coord_gpu >= zbtm_gpu[i, j, k]) & \
                   (z_coord_gpu <= zup_gpu[i, j, k])

            matching_k_gpu = k_index_gpu[mask]

            if matching_k_gpu.size > 0:
                # Find the closest layer to z_value
                closest_idx = cp.argmin(cp.abs(z_coord_gpu[mask] - z_value))
                layer_assignments.append(matching_k_gpu[closest_idx].item())

            # Assign the most common layer
            if layer_assignments:
                most_common_layer                                        =
Counter(layer_assignments).most_common(1)[0][0]
                assigned_layers[k] = most_common_layer

            end_time = time.time()  # End timing for this layer
            if debug:
                elapsed_time = end_time - start_time
                print(f"Layer {k}: Assigned to old layer {most_common_layer} | Time
taken: {elapsed_time:.2f} seconds")
            # if k>3:
            #     break

        # Convert back to NumPy array
        return cp.asnumpy(assigned_layers)

    assigned_layers = assign_layers_to_new_grid_layerwise_gpu(
        x_3d_grid, y_3d_grid, z_3d_grid, complete_data, dz=5, debug=True
    )
```

```
print("Assigned Layers:", assigned_layers)
assigned_layers0=assigned_layers.copy()
----
```

+*In[ ]:*+

[source, ipython3]

```
----
with open("assignedLayersOld2NewGrid.txt", "r") as file:
    content = file.read()

# Replace commas with spaces and split into a list of numbers
layermappings = np.array(content.replace(",", " ").split(), dtype=int)
assigned_layers = layermappings
plt.plot(assigned_layers)
plt.xlabel('new layers')
plt.ylabel('old layers')
----
```

+*In[ ]:*+

[source, ipython3]

```
# the new layer mapped from the old one need be smoothed

def smooth_and_enforce_non_decreasing(assigned_layers, window_size=10):
    """
    Smooth assigned layers using moving average and enforce non-decreasing integers.

    Parameters:
        assigned_layers (array): Original layer assignments.
        window_size (int): Size of the moving average window.

    Returns:
        smoothed_layers (array): Smoothed and non-decreasing layer assignments.
    """
    # Step 1: Smooth using moving average
    smoothed = np.convolve(
        assigned_layers, np.ones(window_size) / window_size, mode="same"
    )

    # Step 2: Round to nearest integers
    smoothed = np.round(smoothed).astype(int)

    # Step 3: Enforce non-decreasing values
    for i in range(1, len(smoothed)):
        smoothed[i] = max(smoothed[i], smoothed[i - 1])
    return smoothed

# Example Usage
window_size = 10  # Set the window size for smoothing
smoothed_layers = smooth_and_enforce_non_decreasing(assigned_layers, window_size=window_size)

# Plotting
plt.figure(figsize=(5, 3))
# plt.plot(range(len(assigned_layers)), assigned_layers, label="Original Assigned Layers", alpha=0.6, linewidth=2)
plt.plot(range(len(smoothed_layers)), smoothed_layers, label=f"Smoothed Layers (Window = {window_size})", linewidth=2, color="red")
plt.title("Smoothing and Enforcing Non-Decreasing Assigned Layers")
plt.xlabel("New Grid Layer (k)")
plt.ylabel("Old Grid Layer")
# plt.legend()
plt.show()

----


+*In[ ]:*+
[source, ipython3]
```

```
----
# Create a dictionary: old_layer -> list of new layers
old_to_new_layers = defaultdict(list)

for new_layer, old_layer in enumerate(smoothed_layers, start=1):
    old_to_new_layers[old_layer].append(new_layer)


----
```

+*In[ ]:*+

[source, ipython3]

```
----

# New zone mapping
new_layer_zone = {}

for zone_name, (old_start, old_end) in old_zone_intervals.items():
    # Collect all new layers that correspond to any old layer in this range
    new_layers = []
    for old_layer in range(old_start, old_end + 1):
        if old_layer in old_to_new_layers:
            new_layers.extend(old_to_new_layers[old_layer])  # Add mapped new
layers

        # Store min/max new layers for the zone
        if new_layers:
            new_layer_zone[zone_name] = (min(new_layers), max(new_layers))

    # Convert to DataFrame
    zone_df = pd.DataFrame([
        {"zone": zone, "new_layer_start": start, "new_layer_end": end}
        for zone, (start, end) in new_layer_zone.items()
    ])

    print(zone_df)

    ----



    +*In[ ]:*+
    [source, ipython3]
    ----
    zonelayers_dfss = {zone: end - start + 1 for zone, (start, end) in
new_layer_zone.items()}

    print(zonelayers_dfss)
```

```
layer_counts = np.array([end - start + 1 for start, end in new_layer_zone.values()])
print(layer_counts)
----
```

+*In[ ]:*+

[source, ipython3]

```
----
def interpolate_properties_layer_by_layer_with_assignment(
    x_3d_grid, y_3d_grid, z_3d_grid, assigned_layers, complete_data, properties
):
    """
    Interpolate multiple property values onto the new grid using layer assignments.

    Parameters:
        x_3d_grid, y_3d_grid, z_3d_grid (array): Regularized grid coordinates.
        assigned_layers (array): Original layer assignments for each point in the new
grid.
        complete_data (DataFrame): Original data with properties (`facies`, `Rhob`,
`Vp`, etc.).
        properties (list of str): List of property names to interpolate.

    Returns:
        property_grids (dict): Dictionary where keys are property names and values
are 3D grids.
    """
    # Initialize a dictionary to store the 3D grids for each property
    property_grids = {prop: np.full_like(z_3d_grid, fill_value=np.nan) for prop in
properties}

    # Iterate over unique assigned layers (original `k_index`)
    for original_layer in np.unique(assigned_layers):
        if original_layer == -1:
            continue  # Skip unassigned points

        # Extract points in the current original layer
        layer_data = complete_data[complete_data["k_index"] == original_layer]
        print(f'Original layer: {original_layer}. Layer data: {layer_data.shape}')

        # Prepare points for interpolation
        known_points = layer_data[["x_coord", "y_coord"]].values

        if len(known_points) == 0:
            continue  # Skip if no data for this layer

        # Interpolate for all points in the new grid assigned to this layer
        for k, layer_assignment in enumerate(assigned_layers):
            if layer_assignment != original_layer:
```

```
            continue  # Skip if the current new grid layer is not assigned to the current
old layer

        # Extract the 2D slice for this new grid layer
        new_points = np.column_stack([x_3d_grid[:, :, k].ravel(), y_3d_grid[:, :,
k].ravel()])

        # Interpolate each property
        for prop in properties:
            if prop=='facies':
                method = 'nearest'
            else:
                method = 'linear'
            known_values = layer_data[prop].values
            if len(known_values) == 0:
                continue  # Skip if no data for this property

            # Perform nearest-neighbor interpolation
            interpolated_values = griddata(
                points=known_points,
                values=known_values,
                xi=new_points,
                method=method
            ).reshape(x_3d_grid.shape[:2])
            # Assign interpolated values to the 3D grid for the property
            property_grids[prop][:, :, k] = interpolated_values

    return property_grids



# Define the properties to interpolate
# properties = ["facies", "Rhob", "Vp", "Vs", "Sg_final"]
properties = ["facies","RbRandom","VpRandom", "VsRandom"]
# properties = ["RbRandom"]
# Perform the interpolation
property_grids = interpolate_properties_layer_by_layer_with_assignment(
    x_3d_grid, y_3d_grid, z_3d_grid, smoothed_layers, complete_data, properties
)

# Access the individual grids
facies_3d_grid = property_grids["facies"]
rhob_3d_grid = property_grids["RbRandom"]
vp_3d_grid = property_grids["VpRandom"]
vs_3d_grid = property_grids["VsRandom"]
```

+*In[ ]:*+

```
[source, ipython3]
----
propertiesSg = ["Sg_final","Sg2024","Sg2030","Sg2040", "Sg2050", "Sg2060","Sg2070" ]
# Perform the interpolation
Sg_grids = interpolate_properties_layer_by_layer_with_assignment(
  x_3d_grid, y_3d_grid, z_3d_grid, smoothed_layers, complete_data, propertiesSg
)

# Access the individual grids
sg_final_grid = Sg_grids["Sg_final"]
Sg2024_grid = Sg_grids["Sg2024"]
Sg2030_grid = Sg_grids["Sg2030"]
Sg2040_grid = Sg_grids["Sg2040"]
Sg2050_grid = Sg_grids["Sg2050"]
Sg2060_grid = Sg_grids["Sg2060"]
Sg2070_grid = Sg_grids["Sg2070"]
----


+*In[ ]:*+
[source, ipython3]
----
def merge_dicts_concat_arrays(dict1, dict2, axis=0):
    merged_dict = {}
    for key in dict1:
        if key in dict2:
            # Concatenate arrays along the specified axis
            merged_dict[key] = np.concatenate((dict1[key], dict2[key]), axis=axis)
        else:
            raise KeyError(f"Key '{key}' found in dict1 but not in dict2.")
    return merged_dict

# Merge the dictionaries along the first axis (axis=0)
merged_prop = property_grids | Sg_grids

# Example: Shape of arrays after merging
for key, array in merged_prop.items():
    print(f"{key}: {array.shape}")
----



+*In[ ]:*+
[source, ipython3]
----
def generate_dataframe_from_grids(x_3d_grid, y_3d_grid, z_3d_grid, property_grids):
    """
    Generate a DataFrame from 3D grids and property values.
    Parameters:
      x_3d_grid, y_3d_grid, z_3d_grid (array): Regularized 3D grids for coordinates.
      property_grids (dict): Dictionary of 3D property grids.

    Returns:
      DataFrame: Combined DataFrame with `i_index`, `j_index`, `k_index`, coordinates, and properties.
    """
    # Get grid dimensions
    ni, nj, nk = x_3d_grid.shape

    # Flatten the grids
    i_index = np.repeat(np.arange(1, ni + 1), nj * nk)
    j_index = np.tile(np.repeat(np.arange(1, nj + 1), nk), ni)
    k_index = np.tile(np.arange(1, nk + 1), ni * nj)

    x_coords = x_3d_grid.ravel()
    y_coords = y_3d_grid.ravel()
    z_coords = z_3d_grid.ravel()

    # Create a dictionary for DataFrame
    data_dict = {
      "i_index": i_index,
            "j_index": j_index,
            "k_index": k_index,
            "x_coord": x_coords,
            "y_coord": y_coords,
            "z_coord": z_coords,
        }

        # Add properties to the dictionary
        for prop, grid in property_grids.items():
            data_dict[prop] = grid.ravel()

        # Create and return the DataFrame
        return pd.DataFrame(data_dict)

    # Generate the DataFrame
    combined_data    =    generate_dataframe_from_grids(x_3d_grid,    y_3d_grid,
z_3d_grid, merged_prop)

    # Preview the DataFrame
    print(combined_data.head())

    ----
```

+*In[ ]:*+

[source, ipython3]

```
----
df_sorted = combined_data.sort_values(by=["k_index", "j_index", "i_index"], ascending=[False, True, True])
df_sorted.head()
----
```

+*In[ ]:*+

[source, ipython3]

```
----
df_cropped = df_sorted[(df_sorted["k_index"] >= 1) & (df_sorted["k_index"] <= 420)]
df_cropped.head()
----
```

+*In[ ]:*+

[source, ipython3]

```
----
# we have almost every facies, but only 5 core facies. Below is to clean the facies
dfss = df_cropped.copy()
```

```
# Define the mapping function
def map_facies(facies_value):
    if facies_value in [2, 4, 12, 18, 30]:
        return facies_value
    elif facies_value in [0, 1]:
        return 2
    elif facies_value == 3:
        return np.random.choice([2, 4])
    elif facies_value in [5, 6, 7]:
        return 4
    elif facies_value == 8:
        return np.random.choice([4, 12])
    elif facies_value in [9, 10, 11]:
        return 12
    elif facies_value in [13, 14, 15, 16, 17]:
        return 12
    elif facies_value in [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]:
        return 18
    else:
        return facies_value  # Default case, though all should be covered

# Apply the mapping function to create a new column
dfss['faciesCorr'] = dfss['facies'].apply(map_facies)

unique_facies = dfss["faciesCorr"].unique()
```

```
print(unique_facies)

----




+*In[ ]:*+
[source, ipython3]
----
# save for Petrel
propertiesList = ["faciesCorr",
                 "Rb0", "Rb2030","Rb2050","Rb2070",
                 "Vp0", "Vp2030","Vp2050","Vp2070",
                "Vs0", "Vs2030","Vs2050","Vs2070"]

with open("MyProperties.gslib", "w") as f:
   # GSLIB-style header
   f.write("MyPropertiesFile\n")
   f.write("13\n")              # number of variables below

f.write("faciesCorr\nRb0\nRb2030\nRb2050\nRb2070\nVp0\nVp2030\nVp2050\nVp207
0\nVs0\nVs2030\nVs2050\nVs2070\n")
   # Now write one row per cell
   for row in dfss[propertiesList].values:
        f.write(" ".join(map(str, row)) + "\n")
----



+*In[ ]:*+
[source, ipython3]
----
propertiesList                          =                          ["Rb0","RbRandom",
"Rb2030","Rb2030Random","Rb2050","Rb2050Random","Rb2070","Rb2070Random",
                  "Vp0","VpRandom",
"Vp2030","Vp2030Random","Vp2050","Vp2050Random","Vp2070","Vp2070Random",
                  "Vs0","VsRandom",
"Vs2030","Vs2030Random","Vs2050","Vs2050Random","Vs2070","Vs2070Random"]

# Save each property into a separate file
for prop in propertiesList:
    filename = f"{prop}.txt"
    with open(filename, "w") as f:
        for row in dfss.itertuples(index=False):    # df_cropped
            f.write(f"{row.i_index} {row.j_index} {row.k_index} "
                    f"{row.x_coord:.8f} {row.y_coord:.8f} {row.z_coord:.8f} "
                    f"{getattr(row, prop):.6f}\n")


----
```

+*In[ ]:*+

[source, ipython3]

```
# Plotting histogram of facies in `complete_data` and `facies_3d_grid`

ik1 = 1
facies_original = complete_data[complete_data["k_index"] == ik1]["facies"].values
ik2 = 1
facies_regularized = facies_3d_grid[:, :, ik2].ravel()

fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Original facies histogram
axes[0].hist(facies_original, bins=30, color='blue', alpha=0.7, edgecolor='black')
axes[0].set_title(f"Histogram of Facies in Original Data (Layer {ik1})")
axes[0].set_xlabel("Facies Value")
axes[0].set_ylabel("Frequency")

# Regularized facies histogram
axes[1].hist(facies_regularized, bins=30, color='green', alpha=0.7, edgecolor='black')
```

```
axes[1].set_title(f"Histogram of Facies in Regularized Data (Layer {ik2})")
axes[1].set_xlabel("Facies Value")
axes[1].set_ylabel("Frequency")


plt.tight_layout()
plt.show()

----



+*In[ ]:*+
[source, ipython3]
----
# Plotting histogram of facies in `complete_data` and `facies_3d_grid`

# Extract facies data from complete_data
facies_original = complete_data["facies"]

# Flatten the facies_3d_grid for histogram comparison
facies_regularized = facies_3d_grid.ravel()

# Plot histograms
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
```

```
# Original facies histogram
axes[0].hist(facies_original, bins=30, color='blue', alpha=0.7, edgecolor='black')
axes[0].set_title("Histogram of Facies in Original Data")
axes[0].set_xlabel("Facies Value")
axes[0].set_ylabel("Frequency")

# Regularized facies histogram
axes[1].hist(facies_regularized, bins=30, color='green', alpha=0.7,
edgecolor='black')
axes[1].set_title("Histogram of Facies in Regularized Data")
axes[1].set_xlabel("Facies Value")
axes[1].set_ylabel("Frequency")

# Adjust layout and show
plt.tight_layout()
plt.show()
----
```

```
+*In[ ]:*+
[source, ipython3]
----
```

```
phi_c = 0.36
rho_overburden = 1600  # kg/m3
g = 9.8

# Mineral properties
K_quartz, K_clay = 39, 21  # GPa
mu_quartz, mu_clay = 45, 6.85  # GPa
rho_quartz, rho_clay = 2.65, 2.60  # g/cm3

# Fluid properties
K_brine, K_co2 = 2.2, 0.1  # GPa
rho_brine, rho_co2 = 1.03, 0.65  # g/cm3

# Mixing rule exponent (Brie model)
brie_exp = 3

# Facies to Vsh and porosity mappings
facies_vsh = {2.0: 0.65, 4.0: 0.4, 12.0: 0.25, 18.0: 0.1, 30.0: 0.0}  # Limestone is
clean
facies_phi = {2.0: 0.05, 4.0: 0.217, 12.0: 0.30, 18.0: 0.24, 30.0: 0.11}

# --- Functions ---
def vrh_average(vsh):
    K_vrh = 0.5 * ((1 - vsh) * K_quartz + vsh * K_clay + 1 / ((1 - vsh) / K_quartz +
vsh / K_clay))
```

```
    mu_vrh = 0.5 * ((1 - vsh) * mu_quartz + vsh * mu_clay + 1 / ((1 - vsh) / mu_quartz + vsh / mu_clay))
    return K_vrh, mu_vrh

def poisson_ratio(K, mu):
    return (3 * K - 2 * mu) / (6 * K + 2 * mu)

def effective_pressure(depth):
    return rho_overburden * g * depth / 1e9  # GPa

def hertz_mindlin(K_vrh, mu_vrh, nu, depth):
    p_eff = effective_pressure(depth)
    C = 2.8 / phi_c
    Kc = ((C**2 * (1 - phi_c)**2 * mu_vrh**2 * p_eff) / (18 * np.pi**2 * (1 - nu)**2))**(1/3)
    muc = ((5 - 4 * nu) / (10 - 5 * nu)) * ((3 * C**2 * (1 - phi_c)**2 * mu_vrh**2 * p_eff) / (2 * np.pi**2 * (1 - nu)**2))**(1/3)
    return Kc, muc

def dry_frame_K(phi, Kc, muc, K_vrh):
    xi = muc / 6 * (9 * Kc + 8 * muc) / (Kc + 2 * muc)
    term1 = phi / phi_c / (Kc + 4/3 * muc)
    term2 = (1 - phi / phi_c) / (K_vrh + 4/3 * muc)
    return 1 / (term1 + term2) - 4/3 * muc
```

```
def K_fluid(Sg):
    return (K_brine - K_co2) * (1 - Sg)**brie_exp + K_co2


def rho_matrix(vsh):
    return (1 - vsh) * rho_quartz + vsh * rho_clay


def rho_fluid(Sg):
    return Sg * rho_co2 + (1 - Sg) * rho_brine


def rho_sat(phi, rho_m, rho_f):
    return (1 - phi) * rho_m + phi * rho_f


def K_sat(K_dry, K_vrh, Kf, phi):
    num = (1 - K_dry / K_vrh)**2
    denom = phi / Kf + (1 - phi) / K_vrh - (K_dry / K_vrh**2)
    return K_dry + num / denom


def compute_velocities(Ksat, mu, rho):
    Vp = np.sqrt((Ksat + 4/3 * mu) / rho) * 3280.84  # ft/s
    Vs = np.sqrt(mu / rho) * 3280.84  # ft/s
    return Vp, Vs


# Apply randomization with noise based on standard deviations
Vp0 = np.full(len(dfss), np.nan)
Vs0 = np.full(len(dfss), np.nan)
```

```
Rb0 = np.full(len(dfss), np.nan)

Vp = np.full(len(dfss), np.nan)
Vs0 = np.full(len(dfss), np.nan)
Rb0 = np.full(len(dfss), np.nan)

Vp2030 = np.full(len(dfss), np.nan)
Vs2030 = np.full(len(dfss), np.nan)
Rb2030 = np.full(len(dfss), np.nan)

Vp2030Random = np.full(len(dfss), np.nan)
Vs2030Random = np.full(len(dfss), np.nan)
Rb2030Random = np.full(len(dfss), np.nan)

Vp2050 = np.full(len(dfss), np.nan)
Vs2050 = np.full(len(dfss), np.nan)
Rb2050 = np.full(len(dfss), np.nan)
Vp2050Random = np.full(len(dfss), np.nan)
Vs2050Random = np.full(len(dfss), np.nan)
Rb2050Random = np.full(len(dfss), np.nan)

Vp2070 = np.full(len(dfss), np.nan)
Vs2070 = np.full(len(dfss), np.nan)
Rb2070 = np.full(len(dfss), np.nan)
Vp2070Random = np.full(len(dfss), np.nan)
```

```
Vs2070Random = np.full(len(dfss), np.nan)
Rb2070Random = np.full(len(dfss), np.nan)

for f in faciesModelList:
    mask = dfss["faciesCorr"] == f
    depth = -dfss.loc[mask, "z_coord"].values
    vsh = facies_vsh[f]
    phi = facies_phi[f]

    K_vrh, mu_vrh = vrh_average(vsh)
    nu = poisson_ratio(K_vrh, mu_vrh)
    Kc, muc = hertz_mindlin(K_vrh, mu_vrh, nu, depth)
    Kdry = dry_frame_K(phi, Kc, muc, K_vrh)

    # Before injection (all brine)
    Kf0 = np.full_like(phi, K_brine)

    rho_f0 = np.full_like(phi, rho_brine)
    rho_m = rho_matrix(vsh)  # scalar
    rho_0 = rho_sat(phi, rho_m, rho_f0)
    Ksat0 = K_sat(Kdry, K_vrh, Kf0, phi)
    vp0, vs0 = compute_velocities(Ksat0, muc, rho_0)
    Rb0[mask] = rho_0
    Vp0[mask] = vp0
    Vs0[mask] = vs0
```

```
vp_std = facies_fit[f][1]
vs_std = facies_fitVs[f][1]
rb_std = mean_std_Rhob.loc[f, "Std"]

dvp = np.random.normal(0.0, vp_std, size=vp0.shape)
dvs = np.random.normal(0.0, vs_std, size=vs0.shape)
drb = np.random.normal(0.0, rb_std, size=rho_0.shape)

dfss.loc[mask, "VpRandom"] = vp0 + dvp
dfss.loc[mask, "VsRandom"] = vs0 + dvs
dfss.loc[mask, "RbRandom"] = rho_0 + drb

# After injection
Sg2030 = dfss.loc[mask, "Sg2030"].values
Kf = K_fluid(Sg2030)
rho_f = rho_fluid(Sg2030)
rho_2030 = rho_sat(phi, rho_m, rho_f)
Ksat = K_sat(Kdry, K_vrh, Kf, phi)
Vp2030[mask], Vs2030[mask] = compute_velocities(Ksat, muc, rho_2030)
Rb2030[mask] = rho_2030
Rb2030Random[mask] = Rb2030[mask] + drb
Vp2030Random[mask] = Vp2030[mask]+ dvp
Vs2030Random[mask] = Vs2030[mask]+ dvs
```

```
    Sg2050 = dfss.loc[mask, "Sg2050"].values
    Kf = K_fluid(Sg2050)
    rho_f = rho_fluid(Sg2050)
    rho_2050 = rho_sat(phi, rho_m, rho_f)
    Ksat = K_sat(Kdry, K_vrh, Kf, phi)
    Vp2050[mask], Vs2050[mask] = compute_velocities(Ksat, muc, rho_2050)
    Rb2050[mask] = rho_2050
    Rb2050Random[mask] = Rb2050[mask] + drb
    Vp2050Random[mask] = Vp2050[mask]+ dvp
    Vs2050Random[mask] = Vs2050[mask]+ dvs

    Sg2070 = dfss.loc[mask, "Sg2070"].values
    Kf = K_fluid(Sg2070)
    rho_f = rho_fluid(Sg2070)
    rho_2070 = rho_sat(phi, rho_m, rho_f)
    Ksat = K_sat(Kdry, K_vrh, Kf, phi)
    Vp2070[mask], Vs2070[mask] = compute_velocities(Ksat, muc, rho_2070)
    Rb2070[mask] = rho_2070
    Rb2070Random[mask] = Rb2070[mask] + drb
    Vp2070Random[mask] = Vp2070[mask]+ dvp
    Vs2070Random[mask] = Vs2070[mask]+ dvs


dfss["Vp0"] = Vp0
dfss["Vs0"] = Vs0
```

```
dfss["Rb0"] = Rb0

dfss["Vp2030"] = Vp2030
dfss["Vs2030"] = Vs2030
dfss["Rb2030"] = Rb2030
dfss["Vp2030Random"] = Vp2030Random
dfss["Vs2030Random"] = Vs2030Random
dfss["Rb2030Random"] = Rb2030Random

dfss["Vp2050"] = Vp2050
dfss["Vs2050"] = Vs2050
dfss["Rb2050"] = Rb2050
dfss["Vp2050Random"] = Vp2050Random
dfss["Vs2050Random"] = Vs2050Random
dfss["Rb2050Random"] = Rb2050Random
dfss["Vp2070"] = Vp2070
dfss["Vs2070"] = Vs2070
dfss["Rb2070"] = Rb2070
dfss["Vp2070Random"] = Vp2070Random
dfss["Vs2070Random"] = Vs2070Random
dfss["Rb2070Random"] = Rb2070Random
# Display sample stats
dfss[["Vp0",        "VpRandom",        "Vp2030","Vp2030Random","Vs0",
"VsRandom","Vs2030", "Vs2030Random",
        "Rb0",
"RbRandom","Rb2030","Rb2030Random","Vp2050","Vp2050Random"    ]].describe()
```

+*In[ ]:*+

[source, ipython3]

```
def plot_volume_slices(dfss, layer_index_top=100, layer_index_side=100, propertyname="faciesCorr"):
    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    import numpy as np

    fig = plt.figure(figsize=(14, 10))
    ax = fig.add_subplot(111, projection="3d")

    # Full scatter plot
    sc = ax.scatter(
        dfss["x_coord"],
        dfss["y_coord"],
        dfss["z_coord"],
        c=dfss[propertyname],
        cmap="viridis",
        alpha=0.1,
        s=2,
        edgecolor="none"
    )

    # === TOP SLICE (fixed k_index === layer_index_top) ===
    top = dfss[dfss["k_index"] == layer_index_top]
    ax.scatter(
        top["x_coord"],
        top["y_coord"],
        top["z_coord"],
        c=top[propertyname],
        cmap="viridis",
        s=15,
        edgecolor="none"
    )

    # === SIDE SLICE (e.g., fixed i_index === layer_index_side) ===
    side = dfss[dfss["i_index"] == layer_index_side]
    ax.scatter(
        side["x_coord"],
        side["y_coord"],
        side["z_coord"],
        c=side[propertyname],
        cmap="viridis",
        s=15,
        edgecolor="none"
    )

    # Add colorbar
    cbar = plt.colorbar(sc, ax=ax, shrink=0.5, pad=0.1)
    cbar.set_label(propertyname)

    ax.set_xlabel("X (m)")
    ax.set_ylabel("Y (m)")
    ax.set_zlabel("Z (m)")
    ax.set_title(f"CO2 Saturation in 2030 with Top Layer {layer_index_top} and
Side Layer {layer_index_side}")

    plt.tight_layout()
    plt.show()

plot_volume_slices(dfss, layer_index_top=363, layer_index_side=150,
propertyname="Sg2030")
```

+*In[ ]:*+

[source, ipython3]

```
plot_volume_slices(dfss, layer_index_top=363, layer_index_side=150, propertyname="Vp2050")
```

+*In[ ]:*+

[source, ipython3]

```
def plot_saturation_slice(df, layer_k=300):
    import matplotlib.pyplot as plt

    slice_df = df[df["k_index"] == layer_k]

    fig, ax = plt.subplots(figsize=(10, 8))
    sc = ax.scatter(
        slice_df["x_coord"],
        slice_df["y_coord"],
        c=slice_df["Sg2030"],
        cmap="viridis",
        s=10,
        alpha=1.0,
        edgecolors="none"
    )

    ax.set_title(f"CO₂ Saturation at k_index = {layer_k}")
    ax.set_xlabel("X (m)")
    ax.set_ylabel("Y (m)")
    cbar = plt.colorbar(sc, ax=ax)
    cbar.set_label("Sg2030")
    plt.grid(True)
    plt.tight_layout()
    plt.show()
def plot_vertical_slice(df, i_target=150):
    slice_df = df[df["i_index"] == i_target]

    fig, ax = plt.subplots(figsize=(10, 8))
    sc = ax.scatter(
        slice_df["j_index"],
        slice_df["z_coord"],
        c=slice_df["Sg2030"],
        cmap="viridis",
        s=10,
        alpha=1.0,
        edgecolors="none"
        )

        ax.set_title(f"Vertical Cross Section at i_index = {i_target}")
        ax.set_xlabel("j_index")
        ax.set_ylabel("Depth (Z, m)")
        ax.invert_yaxis()
        cbar = plt.colorbar(sc, ax=ax)
        cbar.set_label("Sg2030")
        plt.grid(True)
        plt.tight_layout()
        plt.show()

    def plot_3d_layer(df, k_target=300, thickness=5):
        from mpl_toolkits.mplot3d import Axes3D
        fig = plt.figure(figsize=(10, 8))
        ax = fig.add_subplot(111, projection='3d')

        mask = (df["k_index"] >= k_target - thickness) & (df["k_index"] <= k_target +
thickness)
        subset = df[mask]
        subset = subset.sort_values(by="Sg2030")

        sc = ax.scatter(
            subset["x_coord"],
            subset["y_coord"],
        subset["z_coord"],
        c=subset["Sg2030"],
        cmap="viridis",
        vmin=0,vmax=0.5,
        alpha=0.8,
        s=5
    )

    ax.set_title(f"3D CO₂ Plume at k_index ≈ {k_target}")
    ax.set_xlabel("X (m)")
    ax.set_ylabel("Y (m)")
    ax.set_zlabel("Z (m)")
    ax.view_init(elev=20, azim=-120)  # adjust view
    cbar = plt.colorbar(sc, ax=ax, shrink=0.5, pad=0.1)
    cbar.set_label("Sg2030")
    plt.tight_layout()
    plt.show()


plot_3d_layer(dfss, k_target=363, thickness=5)
```

+*In[ ]:*+

[source, ipython3]

```
def plot_co2_plume(df, sg_col="Sg2030", vmin=0, vmax=0.5):
    # Only keep points with nonzero saturation
    plume = df[df[sg_col] > 0.01].copy()
    print(plume.shape)

    # Sort so high-Sg is not hidden under low-Sg
    plume = plume.sort_values(by=sg_col)

    fig = plt.figure(figsize=(12, 9))
    ax = fig.add_subplot(111, projection='3d')

    sc = ax.scatter(
        plume["x_coord"],
        plume["y_coord"],
        plume["z_coord"],
        c=plume[sg_col],
        cmap="viridis",
        vmin=vmin,
        vmax=vmax,
        alpha=0.9,
        s=5
    )
    # Axes and colorbar
    cbar = plt.colorbar(sc, ax=ax, pad=0.1, shrink=0.5)
    cbar.set_label(sg_col)

    ax.set_title(f"3D CO₂ Plume Visualization ({sg_col})")
    ax.set_xlabel("X (m)")
    ax.set_ylabel("Y (m)")
    ax.set_zlabel("Z (m)")
    ax.view_init(elev=20, azim=120)  # Adjust camera angle if needed

    plt.tight_layout()
    plt.show()

plot_co2_plume(dfss, sg_col="Sg2030", vmin=0, vmax=0.4)
plot_co2_plume(dfss, sg_col="Sg2050", vmin=0, vmax=0.4)
plot_co2_plume(dfss, sg_col="Sg2070", vmin=0, vmax=0.4)
----
```

```
+*In[ ]:*+
[source, ipython3]
----
datadir = '../processing'
filenames = ['Seis.bin', 'Seis2030.bin', 'Seis2070.bin']
titles = ['Baseline (Seis)', '2030 (Seis2030)', '2070 (Seis2070)']

nz, nx, ny = 420, 288, 314
iz = 353  # index for the depth slice (Z-axis)

def save_to_gslib(df, filename, property_name="property"):
    with open(filename, 'w') as f:
        f.write("GSLIB format file\n")
        f.write("4\n")
        f.write("i_index\n")
        f.write("j_index\n")
        f.write("k_index\n")
        f.write(f"{property_name}\n")
        df.to_csv(f, sep=' ', header=False, index=False, float_format='%.6f')

# Create i, j, k indices
i_index, j_index, k_index = np.meshgrid(
    np.arange(1, nx + 1),      # i: 1 to 288
    np.arange(1, ny + 1),      # j: 1 to 314
    np.arange(nz, 0, -1),      # k: 420 to 1 (descending)
    indexing='ij'
)

datasets = []
for filename in filenames:
    datafile = os.path.join(datadir, filename)
    data = np.fromfile(datafile, dtype=np.float32)
    # Fix reshape for correct order
    data_reshaped = data.reshape((ny, nx, nz))        # (ny, nx, nz)
    data_reshaped = np.transpose(data_reshaped, (1, 0, 2))  # (nx, ny, nz)

    df = pd.DataFrame({
    "i_index": i_index.flatten(order='F'),
    "j_index": j_index.flatten(order='F'),
    "k_index": k_index.flatten(order='F'),
    "property": data_reshaped.flatten(order='F')
    })
    name = os.path.splitext(filename)[0]
    print(name)

    save_to_gslib(df, f"{name}.gslib", property_name=name)*
```

+*In[ ]:*+

[source, ipython3]

```
filenames = ['AVO_d10.bin', 'AVO_d25.bin', 'AVO_d55.bin']
nz, nx, ny = 420, 288, 314
# iz = 353  # index for the depth slice (Z-axis)
ix=100
dfss1=dfss[dfss['i_index']==ix]
dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
ymin = np.min(dfss1_sorted['y_coord'])
ymax = np.max(dfss1_sorted['y_coord'])
# Reshape to (ny, nz)
z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
zmin = np.min(z_matrix)
zmax = np.max(z_matrix)
top_z = z_matrix[:, 0]
bottom_z = z_matrix[:, -1]
dz = 5
# First, compute the required number of samples to pad above and below
start_gaps = np.round((zmax - top_z) / dz).astype(int)     # shape (ny,)
```

```
stop_gaps = np.round((bottom_z - zmin) / dz).astype(int)   # shape (ny,)

# Total padded depth is always: top padding + seismic + bottom padding
nzfill = int(np.max(start_gaps + nz + stop_gaps))  # max across all traces
fig, axes = plt.subplots(1, 3, figsize=(20, 6))
datasets = []
for i, fname in enumerate(filenames):
    # Load binary file
    filepath = os.path.join(datadir, fname)
    data = np.fromfile(filepath, dtype=np.float32)
    data_reshaped = data.reshape((ny, nx, nz)).transpose(1, 0, 2)  # to shape (nx, ny,
nz)
    avostr = fname.split('.')[0]

    # Extract vertical slice along y-z at fixed x=ix
    seismic_slice = data_reshaped[ix, :, :]
    filled_slice = np.full((ny, nzfill), -99.0, dtype=np.float32)

# Fill the values into the padded array
    for j in range(ny):
        start_idx = start_gaps[j]
        end_idx = start_idx + nz
        if end_idx > nzfill:
            end_idx = nzfill
            seismic_len = nzfill - start_idx
```

```
                filled_slice[j, start_idx:end_idx] = seismic_slice[j, :seismic_len]
            else:
                filled_slice[j, start_idx:end_idx] = seismic_slice[j, :]
        datasets.append(filled_slice)
        # Plot
        im = axes[i].imshow(filled_slice.T,
        extent=[ymin, ymax, zmax, zmin],
        cmap='seismic',
        aspect=15,
        vmin=np.nanmin(filled_slice[filled_slice > -99]),   # exclude -99 from color
scaling
        vmax=np.nanmax(filled_slice[filled_slice > -99]))
        axes[i].set_title(avostr)
        axes[i].set_xlabel('Y index')
        axes[i].set_ylabel('Z index (depth)')
        plt.colorbar(im, ax=axes[i], shrink=0.6)

    plt.tight_layout()
    plt.show()
    ----



    +*In[ ]:*+
    [source, ipython3]
    ----
```

```
ix=200
dfss1=dfss[dfss['i_index']==ix]
dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
ymin = np.min(dfss1_sorted['y_coord'])
ymax = np.max(dfss1_sorted['y_coord'])
# Reshape to (ny, nz)
z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
zmin = np.min(z_matrix)
zmax = np.max(z_matrix)
top_z = z_matrix[:, 0]
bottom_z = z_matrix[:, -1]
dz = 5
# First, compute the required number of samples to pad above and below
start_gaps = np.round((zmax - top_z) / dz).astype(int)     # shape (ny,)
stop_gaps = np.round((bottom_z - zmin) / dz).astype(int)   # shape (ny,)

# Total padded depth is always: top padding + seismic + bottom padding
nzfill = int(np.max(start_gaps + nz + stop_gaps))  # max across all traces
fig, axes = plt.subplots(1, 3, figsize=(20, 6))
datasets = []
for i, fname in enumerate(filenames):
    # Load binary file
    filepath = os.path.join(datadir, fname)
    data = np.fromfile(filepath, dtype=np.float32)
```

```
        data_reshaped = data.reshape((ny, nx, nz)).transpose(1, 0, 2)  # to shape (nx, ny,
nz)
        avostr = fname.split('.')[0]

        # Extract vertical slice along y-z at fixed x=ix
        seismic_slice = data_reshaped[ix, :, :]
        filled_slice = np.full((ny, nzfill), -99.0, dtype=np.float32)

      # Fill the values into the padded array
        for j in range(ny):
          start_idx = start_gaps[j]
          end_idx = start_idx + nz
          if end_idx > nzfill:
            end_idx = nzfill
            seismic_len = nzfill - start_idx
            filled_slice[j, start_idx:end_idx] = seismic_slice[j, :seismic_len]
          else:
            filled_slice[j, start_idx:end_idx] = seismic_slice[j, :]
        datasets.append(filled_slice)
        # Plot
        im = axes[i].imshow(filled_slice.T,
        extent=[ymin, ymax, zmax, zmin],
        cmap='seismic',
        aspect=15,
    vmin=np.nanmin(filled_slice[filled_slice > -99]),  # exclude -99 from color
scaling
    vmax=np.nanmax(filled_slice[filled_slice > -99]))
    axes[i].set_title(avostr)
    axes[i].set_xlabel('Y index')
    axes[i].set_ylabel('Z index (depth)')
    plt.colorbar(im, ax=axes[i], shrink=0.6)

plt.tight_layout()
plt.show()
----


+*In[ ]:*+
[source, ipython3]
----
filenames1 = ['AVO2070_d10.bin', 'AVO2070_d25.bin', 'AVO2070_d55.bin']
fig, axes = plt.subplots(1, 3, figsize=(20, 6))

for i, fname in enumerate(filenames1):
    # Load binary file
    filepath = os.path.join(datadir, fname)
    data = np.fromfile(filepath, dtype=np.float32)
    data_reshaped = data.reshape((ny, nx, nz)).transpose(1, 0, 2)  # to shape (nx, ny,
nz)
```

```
avostr = fname.split('.')[0]
# Extract vertical slice along y-z at fixed x=ix
seismic_slice = data_reshaped[ix, :, :]
filled_slice = np.full((ny, nzfill), -99.0, dtype=np.float32)
for j in range(ny):
    start_idx = start_gaps[j]
    end_idx = start_idx + nz
    if end_idx > nzfill:
        end_idx = nzfill
        seismic_len = nzfill - start_idx
        filled_slice[j, start_idx:end_idx] = seismic_slice[j, :seismic_len]
    else:
        filled_slice[j, start_idx:end_idx] = seismic_slice[j, :]
datasets.append(filled_slice)
im = axes[i].imshow(filled_slice.T,
extent=[ymin, ymax, zmax, zmin],
cmap='seismic',
aspect=15,
vmin=np.nanmin(filled_slice[filled_slice > -99]),   # exclude -99 from color
scaling
vmax=np.nanmax(filled_slice[filled_slice > -99]))
axes[i].set_title(avostr)
axes[i].set_xlabel('Y index')
axes[i].set_ylabel('Z index (depth)')
plt.colorbar(im, ax=axes[i], shrink=0.6)
plt.tight_layout()
plt.show()
----


+*In[ ]:*+
[source, ipython3]
----

np.random.seed(0)

# Prepare the required slices
AVO_d10 = datasets[0]
AVO_d25_diff = datasets[1] - datasets[0]
AVO_d55_diff = datasets[2] - datasets[0]
AVO2070_d10 = datasets[3]- datasets[0]
AVO2070_d25_diff = datasets[4] - datasets[1]
AVO2070_d55_diff = datasets[5] - datasets[2]

titles = [
    "AVO_d10 (baseline)",
    "AVO_d25 - AVO_d10",
    "AVO_d55 - AVO_d10",
    "AVO2070_d10 - AVO_d10",
        "AVO2070_d25 - AVO_d25",
        "AVO2070_d55 - AVO_d55"
    ]

    images = [
        AVO_d10,
        AVO_d25_diff,
        AVO_d55_diff,
        AVO2070_d10,
        AVO2070_d25_diff,
        AVO2070_d55_diff
    ]



    fig, axes = plt.subplots(2, 3, figsize=(18, 8))
    for i, ax in enumerate(axes.flat):
        image=images[i]
        im = ax.imshow(image.T, cmap='seismic', aspect=15, 
vmin=np.nanmin(image[image > -99]),vmax=np.nanmax(image[image > -
99]),extent=[ymin, ymax, zmax, zmin] )
        ax.set_title(titles[i])
        ax.set_xlabel('Y index')
        ax.set_ylabel('Z index')
        plt.colorbar(im, ax=ax, shrink=0.7)
    plt.tight_layout()
    plt.show()

    ----
```

```
+*In[ ]:*+
[source, ipython3]
----
fig, axes = plt.subplots(1, 3, figsize=(18, 6))
for ax, data, label in zip(axes, [Vp_low, Vs_low, Rb_low], ['Vp_low', 'Vs_low', 'Density_low']):
    image=data
    im = ax.imshow(image.T, cmap='seismic', aspect=15, vmin=np.nanmin(image),vmax=np.nanmax(image),extent=[ymin, ymax, zmax, zmin] )
    ax.set_title(label)
    fig.colorbar(im, ax=ax, shrink=0.7)

plt.tight_layout()
plt.show()
----
```

+*In[ ]:*+

[source, ipython3]

```
----

# Modified HCTNet2D with added dropout
class HCTNet2D(nn.Module):
    def __init__(self, in_channels=3, hidden_dim=64, dropout_rate=0.1):
        super(HCTNet2D, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(in_channels, hidden_dim, kernel_size=3, padding=1),
            nn.BatchNorm2d(hidden_dim),
            nn.ReLU(),
            nn.Dropout2d(dropout_rate),
            nn.Conv2d(hidden_dim, hidden_dim*2, kernel_size=3, padding=1),
            nn.BatchNorm2d(hidden_dim*2),
            nn.ReLU(),
            nn.Dropout2d(dropout_rate)
        )

        self.shared_rep = nn.Sequential(
            nn.Conv2d(hidden_dim*2, hidden_dim*4, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(hidden_dim*4, hidden_dim*2, kernel_size=3, padding=1),
            nn.ReLU()
        )

        def head_block():
            return nn.Sequential(
                nn.Conv2d(hidden_dim*2, 32, kernel_size=3, padding=1),
                nn.ReLU(),
                nn.Conv2d(32, 1, kernel_size=1)
            )

        self.head_vp = head_block()
        self.head_vs = head_block()
        self.head_rho = head_block()

    def forward(self, x):
        x = self.encoder(x)
        x = self.shared_rep(x)
        return self.head_vp(x), self.head_vs(x), self.head_rho(x)

# Updated training loop with early stopping and loss plotting
def train_model(model, train_loader, val_loader, epochs=100, lr=1e-3, patience_limit=5, device='cuda' if torch.cuda.is_available() else 'cpu'):
    model = model.to(device)
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr)

    train_losses, val_losses = [], []
    best_val_loss = float('inf')
    patience = 0
    best_model_state = None

    for epoch in range(epochs):
        model.train()
        total_train_loss = 0
        for batch in train_loader:
            x = batch['seismic'].to(device)
            vp = batch['vp'].to(device)
            vs = batch['vs'].to(device)
            rho = batch['rho'].to(device)
            mask = batch['mask'].to(device)

            pred_vp, pred_vs, pred_rho = model(x)
            loss = masked_mse(pred_vp, vp, mask) + \
                   masked_mse(pred_vs, vs, mask) + \
                   masked_mse(pred_rho, rho, mask)


            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()

        avg_train_loss = total_train_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        model.eval()
        total_val_loss = 0
        with torch.no_grad():
            for batch in val_loader:
                x = batch['seismic'].to(device)
                vp = batch['vp'].to(device)
                vs = batch['vs'].to(device)
                rho = batch['rho'].to(device)
                mask = batch['mask'].to(device)
                mask_single = mask[:, :1]
                pred_vp, pred_vs, pred_rho = model(x)
                loss = masked_mse(pred_vp, vp, mask_single) + \
                masked_mse(pred_vs, vs, mask_single) + \
                masked_mse(pred_rho, rho, mask_single)
            total_val_loss += loss.item()

        avg_val_loss = total_val_loss / len(val_loader)
        val_losses.append(avg_val_loss)

        print(f"Epoch {epoch+1} - Train Loss: {avg_train_loss:.4f} - Val Loss:
{avg_val_loss:.4f}")

        # Early stopping
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            patience = 0
            best_model_state = model.state_dict()
        else:
            patience += 1
            if patience >= patience_limit:
                print("Early stopping triggered.")
                break

    # Restore best model
    if best_model_state:
        model.load_state_dict(best_model_state)
    # Plot loss curves
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Val Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss Over Epochs')
    plt.legend()
    plt.grid(True)
    plt.show()

    return model
```

+*In[ ]:*+
[source, ipython3]

```
class SeismicElasticPatchDataset(Dataset):
    def __init__(self,
                 seismic_volume,   # (H, W, 3)
```

```
        vp, vp_low,       # (H, W)
        vs, vs_low,       # (H, W)
        rho, rho_low,     # (H, W)
        patch_size=(50, 100),
        stride=(25, 50),
        nan_threshold=0.1,
        slice_id=None):
    """
    seismic_volume: numpy array H×W×3
    vp, vp_low, vs, vs_low, rho, rho_low: numpy arrays H×W
    """
    self.seismic = seismic_volume
    self.vp      = vp
    self.vp_low  = vp_low
    self.vs      = vs
    self.vs_low  = vs_low
    self.rho     = rho
    self.rho_low = rho_low

    self.ph, self.pw = patch_size
    self.sh, self.sw = stride
    self.nan_threshold = nan_threshold

    self.indices = self._compute_valid_patch_indices()
    self.slice_id = slice_id
```

```
        # print(f"[DATASET] init: seismic={seismic_volume.shape}  vp={vp.shape}
")
        # [DATASET] init: seismic=(314, 698, 3)  vp=(314, 698)

    def _compute_valid_patch_indices(self):
        H, W = self.seismic.shape[:2]
        inds = []
        for i in range(0, H - self.ph + 1, self.sh):
            for j in range(0, W - self.pw + 1, self.sw):
                patch = self.seismic[i:i+self.ph, j:j+self.pw, :]
                valid_fraction = np.count_nonzero(~np.isnan(patch)) / patch.size
                if valid_fraction >= 1.0 - self.nan_threshold:
                    inds.append((i, j))
        return inds

    def __len__(self):
        return len(self.indices)

    def __getitem__(self, idx):
        i, j = self.indices[idx]

        # --- seismic patch & mask ---
        patch_seis_np = self.seismic[i:i+self.ph, j:j+self.pw, :]      # shape (ph, pw, 3)
```

```
# build a single-channel mask from channel 0 (they're all NaN in the same spots)
mask2d = ~np.isnan(patch_seis_np[..., 0])                    # shape (ph, pw)
# replace NaNs with zero before sending to the network
patch_seis_np = np.nan_to_num(patch_seis_np).astype(np.float32)

# to torch: seismic (3, ph, pw), mask (1, ph, pw)
patch_seis = torch.from_numpy(patch_seis_np).permute(2, 0, 1)    # (3, ph, pw)
mask     = torch.from_numpy(mask2d.astype(np.float32)).unsqueeze(0)  # (1, ph, pw)

# --- elastic patches (all already float32, no channel dimension) ---
patch_vp     = np.nan_to_num(self.vp[i:i+self.ph, j:j+self.pw]).astype(np.float32)
patch_vp_low = np.nan_to_num(self.vp_low[i:i+self.ph, j:j+self.pw]).astype(np.float32)
patch_vs     = np.nan_to_num(self.vs[i:i+self.ph, j:j+self.pw]).astype(np.float32)
patch_vs_low = np.nan_to_num(self.vs_low[i:i+self.ph, j:j+self.pw]).astype(np.float32)
patch_rho    = np.nan_to_num(self.rho[i:i+self.ph, j:j+self.pw]).astype(np.float32)
patch_rho_low= np.nan_to_num(self.rho_low[i:i+self.ph, j:j+self.pw]).astype(np.float32)
```

```
        # to torch: each becomes (1, ph, pw)
        vp      = torch.from_numpy(patch_vp).unsqueeze(0)
        vp_low  = torch.from_numpy(patch_vp_low).unsqueeze(0)
        vs      = torch.from_numpy(patch_vs).unsqueeze(0)
        vs_low  = torch.from_numpy(patch_vs_low).unsqueeze(0)
        rho     = torch.from_numpy(patch_rho).unsqueeze(0)
        rho_low = torch.from_numpy(patch_rho_low).unsqueeze(0)

        # print(f"[DATASET] seismic={patch_seis.shape}  mask={mask.shape}  "
        #       f"vp={vp.shape}  vs={vs.shape}  rho={rho.shape}")
        # [DATASET] seismic=torch.Size([3, 50, 100])  mask=torch.Size([1, 50, 100]) vp=torch.Size([1, 50, 100]) vs=torch.Size([1, 50, 100]) rho=torch.Size([1, 50, 100])


        return {
            "seismic": patch_seis,   # (3, ph, pw)
            "mask":    mask,         # (1, ph, pw)
            "vp":      vp,           # (1, ph, pw)
            "vp_low":  vp_low,       # (1, ph, pw)
            "vs":      vs,           # (1, ph, pw)
            "vs_low":  vs_low,       # (1, ph, pw)
            "rho":     rho,          # (1, ph, pw)
            "rho_low": rho_low,      # (1, ph, pw)
            "origin":  (i, j),
            "slice_id": torch.tensor(self.slice_id, dtype=torch.long)
        }
```

+*In[ ]:*+
[source, ipython3]

```
def evaluate_model_on_test_dynamic(model,
                                   test_loader,
                                   slice_metadata,
                                   patch_size=(50,100),
                                   device='cuda'):
    model.eval()
    model.to(device)
    ph, pw = patch_size

    # 1) Build one canvas per slice
    canvases = {}
    for md in slice_metadata:
```

```
    sid = md['slice_id']
    ny, nzf = md['shape']
    canvases[sid] = {
        "vp_true":  np.zeros((ny, nzf), dtype=np.float32),
        "vs_true":  np.zeros((ny, nzf), dtype=np.float32),
        "rho_true": np.zeros((ny, nzf), dtype=np.float32),
        "vp_pred":  np.zeros((ny, nzf), dtype=np.float32),
        "vs_pred":  np.zeros((ny, nzf), dtype=np.float32),
        "rho_pred": np.zeros((ny, nzf), dtype=np.float32),
        "weight":   np.zeros((ny, nzf), dtype=np.int32),
    }

# 2) Accumulate patch-by-patch
for batch in test_loader:
    # **UNPACK EVERYTHING YOU RETURNED IN __getitem__**
    seismic    = batch["seismic"].to(device)     # (B,3,ph,pw)
    vp_t       = batch["vp"].squeeze(1).cpu().numpy()     # (B,ph,pw)
    vs_t       = batch["vs"].squeeze(1).cpu().numpy()
    rho_t      = batch["rho"].squeeze(1).cpu().numpy()
    vp_p, vs_p, rho_p = model(seismic)         # forward pass
    vp_p       = vp_p.squeeze(1).detach().cpu().numpy()
    vs_p       = vs_p.squeeze(1).detach().cpu().numpy()
    rho_p      = rho_p.squeeze(1).detach().cpu().numpy()
    mask_b     = batch["mask"].cpu().numpy()   # (B,1,ph,pw)
    origins    = batch["origin"]              # tuple of (i0 tensor, j0 tensor)
```

```
        for b in range(vp_t.shape[0]):
    # Get patch position
            sid = slice_metadata[b]['slice_id']
            i0, j0 = slice_metadata[b]["shape"]

            # Get the canvas for this slice
            c = canvases[sid]

            # Create mask for valid indices
            rows = np.arange(patch_size[0])
            cols = np.arange(patch_size[1])

            # Calculate valid indices that don't exceed canvas boundaries
            valid_rows = rows[rows + i0 < c["vp_true"].shape[0]]
            valid_cols = cols[cols + j0 < c["vp_true"].shape[1]]

            # Only use valid indices for adding to canvas
            if len(valid_rows) > 0 and len(valid_cols) > 0:
                c["vp_true"][i0 + valid_rows[:, None], j0 + valid_cols] +=
vp_t[b][valid_rows[:, None], valid_cols]
                c["vp_pred"][i0 + valid_rows[:, None], j0 + valid_cols] +=
vp_p[b][valid_rows[:, None], valid_cols]
                c["vs_true"][i0 + valid_rows[:, None], j0 + valid_cols] +=
vs_t[b][valid_rows[:, None], valid_cols]
```

```
                # Add similar lines for

    # 3) Divide out the weights and mask out zeros → NaN
    for md in slice_metadata:
        sid = md['slice_id']
        ny, nzfill = md["shape"]
        c = canvases[sid]
        w = c["weight"]
        zero = (w == 0)
        for key in ("vp_true","vs_true","rho_true","vp_pred","vs_pred","rho_pred"):
            c[key] = np.divide(
                c[key],
                w,
                out=np.zeros_like(c[key],dtype=np.float32),
                where=w>0
            )
            c[key][zero] = np.nan

    return canvases
```

+*In[ ]:*+

[source, ipython3]

```
----
def evaluate_model_on_test_dynamic(model, test_loader, slice_metadata,
patch_size=(50,100), device='cuda'):
    model.eval()
    model.to(device)
    ph, pw = patch_size

    # 1) Build canvases exactly as you had it:
    canvases = { md['slice_id']: {
            "vp_true":  np.zeros(md['shape'],dtype=np.float32),
            "vs_true":  np.zeros(md['shape'],dtype=np.float32),
            "rho_true": np.zeros(md['shape'],dtype=np.float32),
            "vp_pred":  np.zeros(md['shape'],dtype=np.float32),
            "vs_pred":  np.zeros(md['shape'],dtype=np.float32),
            "rho_pred": np.zeros(md['shape'],dtype=np.float32),
            "weight":   np.zeros(md['shape'],dtype=np.int32),
          }
        for md in slice_metadata }

    # 2) Accumulate patch-by-patch
    for batch in test_loader:
        seismic = batch["seismic"].to(device)          # (B,3,ph,pw)
        vp_t   = batch["vp"].squeeze(1).cpu().numpy()   # (B,ph,pw)
        vs_t   = batch["vs"].squeeze(1).cpu().numpy()
        rho_t  = batch["rho"].squeeze(1).cpu().numpy()
```

```
# forward
with torch.no_grad():
    vp_p, vs_p, rho_p = model(seismic)
    vp_p = vp_p.squeeze(1).detach().cpu().numpy()
    vs_p = vs_p.squeeze(1).detach().cpu().numpy()
    rho_p= rho_p.squeeze(1).detach().cpu().numpy()

mask_b    = batch["mask"].cpu().numpy()[:,0]      # (B,ph,pw)
origins   = batch["origin"]                       # tuple of (i0 tensor, j0 tensor)
slice_ids = batch["slice_id"]                     # (B,)

B = vp_t.shape[0]
for b in range(B):
    sid = slice_ids[b].item()          # <— pull the right canvas
    i0, j0 = origins[0][b].item(), origins[1][b].item()
    m2d = mask_b[b] > 0.5
    rows, cols = np.nonzero(m2d)
    c = canvases[sid]

    # accumulate only valid cells
    c["vp_true"][ i0+rows, j0+cols ] += vp_t[b][ rows, cols ]
    c["vp_pred"][ i0+rows, j0+cols ] += vp_p[b][ rows, cols ]

    c["vs_true"][ i0+rows, j0+cols ] += vs_t[b][ rows, cols ]
```

```
            c["vs_pred"][ i0+rows, j0+cols ] += vs_p[b][ rows, cols ]


            c["rho_true"][ i0+rows, j0+cols ] += rho_t[b][ rows, cols ]
            c["rho_pred"][ i0+rows, j0+cols ] += rho_p[b][ rows, cols ]


            c["weight"][ i0+rows, j0+cols ] += 1


    # 3) normalize & mask
    for md in slice_metadata:
        sid = md['slice_id']
        c   = canvases[sid]
        w   = c["weight"]
        zero = (w == 0)
        for key in ("vp_true","vs_true","rho_true","vp_pred","vs_pred","rho_pred"):
            c[key] = np.divide(
                c[key], w,
                out=np.zeros_like(c[key],dtype=np.float32),
                where=w>0
            )
            c[key][zero] = np.nan


    return canvases
```

+*In[ ]:*+

[source, ipython3]

```
def train_model(model, train_loader, val_loader, epochs=100, lr=1e-3, patience_limit=5, device='cuda' if torch.cuda.is_available() else 'cpu'):
    model = model.to(device)
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr)

    train_losses, val_losses = [], []
    best_val_loss = float('inf')
    patience = 0
    best_model_state = None

    for epoch in range(epochs):
        model.train()
        total_train_loss = 0
        for batch in train_loader:
            x = batch['seismic'].to(device)
            vp = batch['vp'].to(device)
            vs = batch['vs'].to(device)
            rho = batch['rho'].to(device)
            mask = batch['mask'].to(device)
            pred_vp, pred_vs, pred_rho = model(x)
            #      print(f"[TRAIN]      x={x.shape}            pred_vp={pred_vp.shape}
vp={vp.shape}  mask={mask.shape}")
            # [TRAIN] x=torch.Size([16, 3, 50, 100])  pred_vp=torch.Size([16, 1, 50,
100])  vp=torch.Size([16, 1, 50, 100])  mask=torch.Size([16, 1, 50, 100])

            loss = masked_mse(pred_vp, vp, mask) + \
                   masked_mse(pred_vs, vs, mask) + \
                   masked_mse(pred_rho, rho, mask)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()

        avg_train_loss = total_train_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        model.eval()
        total_val_loss = 0
        with torch.no_grad():
            for batch in val_loader:
                x = batch['seismic'].to(device)
                vp = batch['vp'].to(device)
                vs = batch['vs'].to(device)
                rho = batch['rho'].to(device)
                mask = batch['mask'].to(device)
                mask_single = mask[:, :1]
                pred_vp, pred_vs, pred_rho = model(x)
                loss = masked_mse(pred_vp, vp, mask_single) + \
                       masked_mse(pred_vs, vs, mask_single) + \
                       masked_mse(pred_rho, rho, mask_single)
                total_val_loss += loss.item()

        avg_val_loss = total_val_loss / len(val_loader)
        val_losses.append(avg_val_loss)

        print(f"Epoch {epoch+1} - Train Loss: {avg_train_loss:.4f} - Val Loss:
{avg_val_loss:.4f}")

        # Early stopping
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            patience = 0
            best_model_state = model.state_dict()
        else:
            patience += 1
            if patience >= patience_limit:
                print("Early stopping triggered.")
                break
```

```
    # Restore best model
    if best_model_state:
        model.load_state_dict(best_model_state)

    # Plot loss curves
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Val Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss Over Epochs')
    plt.legend()
    plt.grid(True)
    plt.show()

    return model
```

+*In[ ]:*+

```
[source, ipython3]
----
random_ix = sorted(random.sample(range(10, nx - 10), num_slices))
random_ix
----
```

```
+*In[ ]:*+
[source, ipython3]
----

nz, nx, ny = 420, 288, 314
num_slices = 20
# random_ix = sorted(random.sample(range(10, nx - 10), num_slices))
filenames = ['AVO_d10.bin', 'AVO_d25.bin', 'AVO_d55.bin']
datadir = '../processing'
dz = 5

slice_dataset = []
slice_metadata = []
Vp_maxlist = np.zeros(num_slices, dtype=np.float32)
Vs_maxlist = np.zeros(num_slices, dtype=np.float32)
Rb_maxlist = np.zeros(num_slices, dtype=np.float32)
```

```
z_maxlist = np.zeros(num_slices, dtype=np.float32)
z_minlist = np.zeros(num_slices, dtype=np.float32)
# ymin, ymax is the same for all ix
ymin, ymax = np.min(dfss['y_coord']), np.max(dfss['y_coord'])

for idx, ix in enumerate(random_ix):
    dfss1 = dfss[dfss['i_index'] == ix]
    dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
    z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
    zmin, zmax = np.min(z_matrix), np.max(z_matrix)
    z_minlist[idx], z_maxlist[idx] = zmin, zmax
    top_z = z_matrix[:, 0]
    bottom_z = z_matrix[:, -1]
    start_gaps = np.round((zmax - top_z) / dz).astype(int)
    stop_gaps = np.round((bottom_z - zmin) / dz).astype(int)
    nzfill = int(np.max(start_gaps + nz + stop_gaps))
    slice_metadata.append({
        "slice_id": idx,
        "inline":   ix,
        "shape":    (ny, nzfill)
    })
    seismic_stack = np.full((ny, nzfill, 3), np.nan, dtype=np.float32)
    Vp_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
    Vs_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
    Rb_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
```

```
        for i, fname in enumerate(filenames):
            filepath = os.path.join(datadir, fname)
            data = np.fromfile(filepath, dtype=np.float32)
            data_reshaped = data.reshape((ny, nx, nz)).transpose(1, 0, 2)
            seismic_slice = data_reshaped[ix, :, :]
            for j in range(ny):
                start_idx = start_gaps[j]
                end_idx = start_idx + nz
                if end_idx > nzfill:
                    length = nzfill - start_idx
                    seismic_stack[j, start_idx:end_idx, i] = seismic_slice[j, :length]
                    Vp_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Vp0'].values.reshape(ny, nz)[j, :length]
                    Vs_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Vs0'].values.reshape(ny, nz)[j, :length]
                    Rb_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Rb0'].values.reshape(ny, nz)[j, :length]
                else:
                    seismic_stack[j, start_idx:end_idx, i] = seismic_slice[j, :]
                    Vp_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Vp0'].values.reshape(ny, nz)[j, :]
                    Vs_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Vs0'].values.reshape(ny, nz)[j, :]
```

```
            Rb_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Rb0'].values.reshape(ny, nz)[j, :]

        # Apply normalization + smoothing
        seismic_stack = seismic_stack / np.nanmax(np.abs(seismic_stack))
        Vp_low = nan_gaussian_filter_corrected(Vp_padded, sigma=5)
        Vs_low = nan_gaussian_filter_corrected(Vs_padded, sigma=5)
        Rb_low = nan_gaussian_filter_corrected(Rb_padded, sigma=5)
        # Normalize properties
        Vp_maxlist[idx] = np.nanmax(np.abs(Vp_padded))
        Vp_norm = Vp_padded / Vp_maxlist[idx]
        # print(Vp_maxlist[idx])

        Vs_maxlist[idx] = np.nanmax(np.abs(Vs_padded))
        Vs_norm = Vs_padded / Vs_maxlist[idx]
        # print(Vs_maxlist[idx])

        Rb_maxlist[idx] = np.nanmax(np.abs(Rb_padded))
        Rb_norm = Rb_padded / Rb_maxlist[idx]
        # print(Rb_maxlist[idx])

        Vp_low_norm = Vp_low / Vp_maxlist[idx]
        Vs_low_norm = Vs_low / Vs_maxlist[idx]
        Rb_low_norm = Rb_low / Rb_maxlist[idx]
        slice_dataset.append((seismic_stack, Vp_norm, Vp_low_norm, Vs_norm, Vs_low_norm, Rb_norm, Rb_low_norm))


    ishow=0
    fig, axes = plt.subplots(1, 3, figsize=(18, 6))
    for ax, data, label,maxv in zip(axes, [slice_dataset[ishow][1], slice_dataset[ishow][3], slice_dataset[ishow][5]], ['Vp', 'Vs', 'Density'],[Vp_maxlist[ishow],Vs_maxlist[ishow],Rb_maxlist[ishow]]):
        # print(maxv)
        image=data.copy()*maxv
        im = ax.imshow(image.T, cmap='seismic', aspect=15, vmin=np.nanmin(image),vmax=np.nanmax(image),extent=[ymin, ymax, z_maxlist[ishow], z_minlist[ishow]] )
        ax.set_title(label)
        fig.colorbar(im, ax=ax, shrink=0.7)
    plt.tight_layout()
    plt.show()

    fig, axes = plt.subplots(1, 3, figsize=(18, 6))
    for ax, data, label,maxv in zip(axes, [slice_dataset[ishow][2], slice_dataset[ishow][4], slice_dataset[ishow][6]], ['Vplow', 'Vslow', 'Densitylow'],[Vp_maxlist[ishow],Vs_maxlist[ishow],Rb_maxlist[ishow]]):
        # print(maxv)
        image=data.copy()*maxv
        im = ax.imshow(image.T, cmap='seismic', aspect=15,
vmin=np.nanmin(image),vmax=np.nanmax(image),extent=[ymin, ymax,
z_maxlist[ishow], z_minlist[ishow]] )
        ax.set_title(label)
        fig.colorbar(im, ax=ax, shrink=0.7)
    plt.tight_layout()
    plt.show()



----



+*In[ ]:*+
[source, ipython3]
----
all_datasets = []
for slice_idx, s in enumerate(slice_dataset):  # <-- Now slice_idx is defined
    dataset = SeismicElasticPatchDataset(
        seismic_volume=s[0],   # (ny, nzfill, 3)
        vp=s[1], vp_low=s[2],
        vs=s[3], vs_low=s[4],
        rho=s[5], rho_low=s[6],
            patch_size=(50, 100),
            stride=(10, 25),
            nan_threshold=0.15,
            slice_id = slice_idx
        )
        # dataset.slice_id = slice_idx  # <-- This is now valid
        all_datasets.append(dataset)



    full_dataset = ConcatDataset(all_datasets)


    n = len(full_dataset)
    print(n)
    train_size = int(0.7 * n)
    val_size = int(0.15 * n)
    test_size = n - train_size - val_size


    train_set, val_set, test_set = random_split(full_dataset, [train_size, val_size,
test_size])



    ----
```

+*In[ ]:*+

[source, ipython3]

```
slice_id_map = []
for sid, ds in enumerate(full_dataset.datasets):
    slice_id_map += [sid] * len(ds)
slice_id_map = np.array(slice_id_map, dtype=int)

# 2) Grab the subset indices from the torch.utils.data.Subset objects
train_idx = np.array(train_set.indices, dtype=int)
val_idx   = np.array(val_set.indices, dtype=int)
test_idx  = np.array(test_set.indices, dtype=int)

# 3) Tally up
records = []
for sid, inline in enumerate(random_ix):
    total = np.sum(slice_id_map == sid)
    train = np.sum(slice_id_map[train_idx] == sid)
    val   = np.sum(slice_id_map[val_idx] == sid)
    test  = np.sum(slice_id_map[test_idx] == sid)
    records.append({
        "slice_id":      sid,
        "inline_number": inline,
        "total_patches": total,
        "train":        train,
        "val":          val,
        "test":         test,
        "test_frac":    test/total if total else np.nan
    })

df_counts = pd.DataFrame(records)
print(df_counts.to_string(index=False))
# df_counts is  DataFrame of per-slice patch counts

# Locate the row with the largest “test” count
best_row = df_counts.loc[df_counts['test'].idxmax()]

# Extract its slice_id and inline number
best_sid    = int(best_row['slice_id'])
best_inline = best_row['inline_number']
best_n_test = best_row['test']

print(f"Slice ID with most test patches: {best_sid}")
print(f"inline number {best_inline} has {best_n_test} test patches")

# find the slice_id for the inline you're interested in
sid = random_ix.index(best_inline)    # e.g. inline k=150
print(sid)
# get the dataset and its shape
```

```
ny_nzf = full_dataset.datasets[sid].seismic.shape[:2]


# initialize coverage
coverage = np.zeros((ny_nzf[0], ny_nzf[1]), dtype=bool)


# mark each test patch
for idx in test_set.indices:  # train_set test_set
    if slice_id_map[idx] != sid:
        continue
    # convert global idx → (row_in_slice_dataset) by subtracting cumulative lengths
    offset = idx - sum(len(ds) for ds in full_dataset.datasets[:sid])
    i0, j0 = full_dataset.datasets[sid].indices[offset]  # origin of that patch
    coverage[i0:i0+50, j0:j0+100] = True


plt.figure(figsize=(6,8))
plt.imshow(coverage.T, origin='lower', aspect='auto', cmap='gray_r')
plt.title(f"Test-patch coverage on inline {best_inline}")
plt.xlabel("Crossline index")
plt.ylabel("Depth index")
plt.show()
----
```

```
+*In[ ]:*+
[source, ipython3]
----
train_loader = DataLoader(train_set, batch_size=16, shuffle=True, num_workers=4)
val_loader = DataLoader(val_set, batch_size=16, shuffle=False, num_workers=2)
test_loader = DataLoader(test_set, batch_size=16, shuffle=False, num_workers=2)
model = HCTNet2D(in_channels=3, hidden_dim=64, dropout_rate=0.1)
trained_model = train_model(model, train_loader, val_loader, epochs=50)


----




+*In[ ]:*+
[source, ipython3]
----



slice_id_map = []
for sid, ds in enumerate(full_dataset.datasets):
    slice_id_map += [sid] * len(ds)
slice_id_map = np.array(slice_id_map, dtype=int)
```

```
# 2) Grab the subset indices from the torch.utils.data.Subset objects
train_idx = np.array(train_set.indices, dtype=int)
val_idx   = np.array(val_set.indices, dtype=int)
test_idx  = np.array(test_set.indices, dtype=int)

# 3) Tally up
records = []
for sid, inline in enumerate(random_ix):
    total = np.sum(slice_id_map == sid)
    train = np.sum(slice_id_map[train_idx] == sid)
    val   = np.sum(slice_id_map[val_idx] == sid)
    test  = np.sum(slice_id_map[test_idx] == sid)
    records.append({
        "slice_id":      sid,
        "inline_number": inline,
        "total_patches": total,
        "train":         train,
        "val":           val,
        "test":          test,
        "test_frac":     test/total if total else np.nan
    })

df_counts = pd.DataFrame(records)
print(df_counts.to_string(index=False))
```

```
# df_counts is  DataFrame of per-slice patch counts

# Locate the row with the largest "test" count
best_row = df_counts.loc[df_counts['test'].idxmax()]

# Extract its slice_id and inline number
best_sid    = int(best_row['slice_id'])
best_inline = best_row['inline_number']
best_n_test = best_row['test']

print(f"Slice ID with most test patches: {best_sid}")
print(f"inline number {best_inline} has {best_n_test} test patches")

# find the slice_id for the inline you're interested in
sid = random_ix.index(best_inline)    # e.g. inline k=150
print(sid)
# get the dataset and its shape
ny_nzf = full_dataset.datasets[sid].seismic.shape[:2]

# 1) figure out which slice_ids actually appear in test_set
test_slice_ids = set()
for idx in test_set.indices:
    # full_dataset[idx] returns a dict with 'slice_id': torch.Tensor(...)
    sid = full_dataset[idx]['slice_id']
    # it might be a tensor, so:
```

```
    test_slice_ids.add(int(sid))

# 2) filter your slice_metadata down to only those
test_slice_metadata = [ md for md in slice_metadata
                        if md['slice_id'] in test_slice_ids ]

# 3) now build only those canvases
canvases = evaluate_model_on_test_dynamic(
    trained_model,
    test_loader,
    test_slice_metadata,    # <-- pass *this*, not the full list
    patch_size=(50,100),
    device='cuda'
)

# 4) now `canvases[best_sid]` will only exist if best_sid ∈ test_slice_ids,
#    and since you picked best_sid as the inline with the most test patches,
#    it will have nonzero weight.
c = canvases[best_sid]
print("weight sum:", c['weight'].sum())

zmin, zmax = z_minlist[best_sid], z_maxlist[best_sid]

# 3) pull the *input* low-frequency fields from slice_dataset
```

```
    _, Vp_norm, Vp_low_norm, Vs_norm, Vs_low_norm, Rb_norm, Rb_low_norm =
slice_dataset[best_sid]

    # and de-normalize them using your per-slice max lists
    Vp_low  = Vp_low_norm * Vp_maxlist[best_sid]
    Vs_low  = Vs_low_norm * Vs_maxlist[best_sid]
    Rb_low  = Rb_low_norm * Rb_maxlist[best_sid]

    # 4) pull the *true* and *predicted* canvases that you built in
evaluate_model_on_test_dynamic
    c = canvases[best_sid]
    Vp_true = Vp_norm * Vp_maxlist[best_sid]
    Vs_true = Vs_norm * Vs_maxlist[best_sid]
    Rb_true = Rb_norm * Rb_maxlist[best_sid]

    Vp_pred = c['vp_pred'] * Vp_maxlist[best_sid]
    Vs_pred = c['vs_pred'] * Vs_maxlist[best_sid]
    Rb_pred = c['rho_pred'] * Rb_maxlist[best_sid]



    Vp_pred_filled = np.where(np.isnan(Vp_pred), Vp_low, Vp_pred)
    Vs_pred_filled = np.where(np.isnan(Vs_pred), Vs_low, Vs_pred)
    Rb_pred_filled = np.where(np.isnan(Rb_pred), Rb_low, Rb_pred)

    Vp_pred_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_pred_filled)
```

```
Vs_pred_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_pred_filled)
Rb_pred_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_pred_filled)

# unpack your three rows
vp_rows   = [Vp_low,    Vp_pred_filled,    Vp_true]
vs_rows   = [Vs_low,    Vs_pred_filled,    Vs_true]
rho_rows  = [Rb_low,    Rb_pred_filled,    Rb_true]

# compute shared scales
vmin_vp, vmax_vp   = np.nanmin(vp_rows),   np.nanmax(vp_rows)
vmin_vs, vmax_vs   = np.nanmin(vs_rows),   np.nanmax(vs_rows)
vmin_rho, vmax_rho = np.nanmin(rho_rows), np.nanmax(rho_rows)

# 5) now plot a 3×3 grid: rows = [Input, Pred, True], cols = [Vp, Vs, ρ]
fig, axes = plt.subplots(3, 3, figsize=(15, 12), sharex=True, sharey=True)

row_data = [
   ([Vp_low,    Vs_low,    Rb_low],    "Input"),
   ([Vp_pred_crop,   Vs_pred_crop,   Rb_pred_crop],   "Predicted"),
   ([Vp_true,   Vs_true,   Rb_true],   "True")
]


props = ["Vp", "Vs", "Density"]
```

```
for i, (data_row, row_label) in enumerate(row_data):
    for j, (img, prop) in enumerate(zip(data_row, props)):
        ax = axes[i, j]
        if j == 0:   # first column is Vp
            vmn, vmx = vmin_vp, vmax_vp
        elif j == 1: # second is Vs
            vmn, vmx = vmin_vs, vmax_vs
        else:        # third is ρ
            vmn, vmx = vmin_rho, vmax_rho

        im = ax.imshow(
            img.T,
            origin='upper',
            extent=[ymin, ymax, zmax, zmin],
            vmin=vmn, vmax=vmx,
            aspect='auto',
            cmap='viridis'
        )
        if i == 0:
            ax.set_title(prop, fontsize=14)
        if j == 0:
            ax.set_ylabel(row_label, fontsize=14)
        fig.colorbar(im, ax=ax, shrink=0.75)

fig.supxlabel("Crossline (Y)")
fig.supylabel("Depth (Z)")
plt.tight_layout()
plt.show()

----
```

```
+*In[ ]:*+
[source, ipython3]
----
random_ix2070 = random_ix
filenames = ['AVO2070_d10.bin', 'AVO2070_d25.bin', 'AVO2070_d55.bin']
datadir = '../processing'
dz = 5

slice_dataset2070 = []
slice_metadata2070 = []
Vp_maxlist = np.zeros(num_slices, dtype=np.float32)
Vs_maxlist = np.zeros(num_slices, dtype=np.float32)
Rb_maxlist = np.zeros(num_slices, dtype=np.float32)
z_maxlist = np.zeros(num_slices, dtype=np.float32)
z_minlist = np.zeros(num_slices, dtype=np.float32)
# ymin, ymax is the same for all ix
```

```
ymin, ymax = np.min(dfss['y_coord']), np.max(dfss['y_coord'])

for idx, ix in enumerate(random_ix2070):
    dfss1 = dfss[dfss['i_index'] == ix]
    dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
    z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
    zmin, zmax = np.min(z_matrix), np.max(z_matrix)
    z_minlist[idx], z_maxlist[idx] = zmin, zmax
    top_z = z_matrix[:, 0]
    bottom_z = z_matrix[:, -1]
    start_gaps = np.round((zmax - top_z) / dz).astype(int)
    stop_gaps = np.round((bottom_z - zmin) / dz).astype(int)
    nzfill = int(np.max(start_gaps + nz + stop_gaps))
    slice_metadata2070.append({
        "slice_id": idx,
        "inline":   ix,
        "shape":    (ny, nzfill)
    })
    seismic_stack = np.full((ny, nzfill, 3), np.nan, dtype=np.float32)
    Vp_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
    Vs_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
    Rb_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)

    for i, fname in enumerate(filenames):
        filepath = os.path.join(datadir, fname)
```

```
          data = np.fromfile(filepath, dtype=np.float32)
          data_reshaped = data.reshape((ny, nx, nz)).transpose(1, 0, 2)
          seismic_slice = data_reshaped[ix, :, :]
          for j in range(ny):
            start_idx = start_gaps[j]
            end_idx = start_idx + nz
            if end_idx > nzfill:
              length = nzfill - start_idx
              seismic_stack[j, start_idx:end_idx, i] = seismic_slice[j, :length]
              Vp_padded[j,           start_idx:end_idx]           =
dfss1_sorted['Vp2070'].values.reshape(ny, nz)[j, :length]
              Vs_padded[j,           start_idx:end_idx]           =
dfss1_sorted['Vs2070'].values.reshape(ny, nz)[j, :length]
              Rb_padded[j,           start_idx:end_idx]           =
dfss1_sorted['Rb2070'].values.reshape(ny, nz)[j, :length]
            else:
              seismic_stack[j, start_idx:end_idx, i] = seismic_slice[j, :]
              Vp_padded[j,           start_idx:end_idx]           =
dfss1_sorted['Vp2070'].values.reshape(ny, nz)[j, :]
              Vs_padded[j,           start_idx:end_idx]           =
dfss1_sorted['Vs2070'].values.reshape(ny, nz)[j, :]
              Rb_padded[j,           start_idx:end_idx]           =
dfss1_sorted['Rb2070'].values.reshape(ny, nz)[j, :]

        # Apply normalization + smoothing
        seismic_stack = seismic_stack / np.nanmax(np.abs(seismic_stack))
        Vp_low = nan_gaussian_filter_corrected(Vp_padded, sigma=5)
        Vs_low = nan_gaussian_filter_corrected(Vs_padded, sigma=5)
        Rb_low = nan_gaussian_filter_corrected(Rb_padded, sigma=5)
        # Normalize properties
        Vp_maxlist[idx] = np.nanmax(np.abs(Vp_padded))
        Vp_norm = Vp_padded / Vp_maxlist[idx]

        Vs_maxlist[idx] = np.nanmax(np.abs(Vs_padded))
        Vs_norm = Vs_padded / Vs_maxlist[idx]

        Rb_maxlist[idx] = np.nanmax(np.abs(Rb_padded))
        Rb_norm = Rb_padded / Rb_maxlist[idx]

        Vp_low_norm = Vp_low / Vp_maxlist[idx]
        Vs_low_norm = Vs_low / Vs_maxlist[idx]
        Rb_low_norm = Rb_low / Rb_maxlist[idx]
        slice_dataset2070.append((seismic_stack, Vp_norm, Vp_low_norm, Vs_norm,
Vs_low_norm, Rb_norm, Rb_low_norm))

    all_datasets = []
    for slice_idx, s in enumerate(slice_dataset2070):  # <-- Now slice_idx is defined
        dataset = SeismicElasticPatchDataset(
            seismic_volume=s[0],   # (ny, nzfill, 3)
            vp=s[1], vp_low=s[2],
            vs=s[3], vs_low=s[4],
            rho=s[5], rho_low=s[6],
            patch_size=(50, 100),
            stride=(10, 25),
            nan_threshold=0.15,
            slice_id = slice_idx
        )
        # dataset.slice_id = slice_idx  # <-- This is now valid
        all_datasets.append(dataset)



    full_dataset2070 = ConcatDataset(all_datasets)


    n = len(full_dataset2070)
    print(n)
    train_size = int(0.7 * n)
    val_size = int(0.15 * n)
    test_size = n - train_size - val_size


    train_set2070, val_set2070, test_set2070 = random_split(full_dataset2070,
[train_size, val_size, test_size])
```

```
train_loader2070 = DataLoader(train_set2070, batch_size=16, shuffle=True, num_workers=4)
val_loader2070 = DataLoader(val_set2070, batch_size=16, shuffle=False, num_workers=2)
test_loader2070 = DataLoader(test_set2070, batch_size=16, shuffle=False, num_workers=2)
model = HCTNet2D(in_channels=3, hidden_dim=64, dropout_rate=0.1)

trained_model2070 = train_model(model, train_loader2070, val_loader2070, epochs=50)
----



+*In[ ]:*+
[source, ipython3]
----
z_maxlist = np.zeros(num_slices, dtype=np.float32)
z_minlist = np.zeros(num_slices, dtype=np.float32)
# ymin, ymax is the same for all ix
ymin, ymax = np.min(dfss['y_coord']), np.max(dfss['y_coord'])

for idx, ix in enumerate(random_ix2070):
   dfss1 = dfss[dfss['i_index'] == ix]
   dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
   z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
```

```
    zmin, zmax = np.min(z_matrix), np.max(z_matrix)
    z_minlist[idx], z_maxlist[idx] = zmin, zmax



slice_id_map = []
for sid, ds in enumerate(full_dataset2070.datasets):
    slice_id_map += [sid] * len(ds)
slice_id_map = np.array(slice_id_map, dtype=int)

# 2) Grab the subset indices from the torch.utils.data.Subset objects
train_idx = np.array(train_set2070.indices, dtype=int)
val_idx   = np.array(val_set2070.indices, dtype=int)
test_idx  = np.array(test_set2070.indices, dtype=int)

# 3) Tally up
records = []
for sid, inline in enumerate(random_ix2070):
    total = np.sum(slice_id_map == sid)
    train = np.sum(slice_id_map[train_idx] == sid)
    val   = np.sum(slice_id_map[val_idx] == sid)
    test  = np.sum(slice_id_map[test_idx] == sid)
    records.append({
        "slice_id":      sid,
        "inline_number": inline,
        "total_patches": total,
        "train":        train,
        "val":          val,
        "test":         test,
        "test_frac":    test/total if total else np.nan
    })

df_counts = pd.DataFrame(records)
print(df_counts.to_string(index=False))
# df_counts is  DataFrame of per-slice patch counts

# Locate the row with the largest “test” count
best_row = df_counts.loc[df_counts['test'].idxmax()]

# Extract its slice_id and inline number
best_sid    = int(best_row['slice_id'])
best_inline = best_row['inline_number']
best_n_test = best_row['test']

print(f"Slice ID with most test patches: {best_sid}")
print(f"inline number {best_inline} has {best_n_test} test patches")

# find the slice_id for the inline you're interested in
sid = random_ix2070.index(best_inline)    # e.g. inline k=150
print(sid)
# get the dataset and its shape
```

```
ny_nzf = full_dataset2070.datasets[sid].seismic.shape[:2]

# initialize coverage
coverage = np.zeros((ny_nzf[0], ny_nzf[1]), dtype=bool)

# mark each test patch
for idx in test_set2070.indices:  # train_set test_set
    if slice_id_map[idx] != sid:
        continue
    # convert global idx → (row_in_slice_dataset) by subtracting cumulative lengths
    offset = idx - sum(len(ds) for ds in full_dataset2070.datasets[:sid])
    i0, j0 = full_dataset2070.datasets[sid].indices[offset]  # origin of that patch
    coverage[i0:i0+50, j0:j0+100] = True

plt.figure(figsize=(6,8))
plt.imshow(coverage.T, origin='lower', aspect='auto', cmap='gray_r')
plt.title(f"Test-patch coverage on inline {best_inline}")
plt.xlabel("Crossline index")
plt.ylabel("Depth index")
plt.show()



test_slice_ids = set()
for idx in test_set2070.indices:
    # full_dataset[idx] returns a dict with 'slice_id': torch.Tensor(...)
    sid = full_dataset2070[idx]['slice_id']
    # it might be a tensor, so:
    test_slice_ids.add(int(sid))

# 2) filter your slice_metadata down to only those
test_slice_metadata2070 = [ md for md in slice_metadata2070
                            if md['slice_id'] in test_slice_ids ]

# 3) now build only those canvases
canvases2070 = evaluate_model_on_test_dynamic(
    trained_model2070,
    test_loader2070,
    test_slice_metadata2070,    # <-- pass *this*, not the full list
    patch_size=(50,100),
    device='cuda'
)

best_sid = 6
# 4) now `canvases[best_sid]` will only exist if best_sid ∈ test_slice_ids,
#    and since you picked best_sid as the inline with the most test patches,
#    it will have nonzero weight.
c = canvases2070[best_sid]

print("weight sum:", c['weight'].sum())
```

```
best_sid = 6
zmin, zmax = z_minlist[best_sid], z_maxlist[best_sid]

# 3) pull the *input* low-frequency fields from slice_dataset
_, Vp_norm, Vp_low_norm, Vs_norm, Vs_low_norm, Rb_norm, Rb_low_norm = slice_dataset2070[best_sid]

# and de-normalize them using your per-slice max lists
Vp_low  = Vp_low_norm * Vp_maxlist[best_sid]
Vs_low  = Vs_low_norm * Vs_maxlist[best_sid]
Rb_low  = Rb_low_norm * Rb_maxlist[best_sid]


# 4) pull the *true* and *predicted* canvases that you built in evaluate_model_on_test_dynamic
Vp_true = Vp_norm * Vp_maxlist[best_sid]
Vs_true = Vs_norm * Vs_maxlist[best_sid]
Rb_true = Rb_norm * Rb_maxlist[best_sid]

Vp_pred = c['vp_pred'] * Vp_maxlist[best_sid]
Vs_pred = c['vs_pred'] * Vs_maxlist[best_sid]
Rb_pred = c['rho_pred'] * Rb_maxlist[best_sid]

Vp_pred_filled = np.where(np.isnan(Vp_pred), Vp_low, Vp_pred)
Vs_pred_filled = np.where(np.isnan(Vs_pred), Vs_low, Vs_pred)
```

```
Rb_pred_filled = np.where(np.isnan(Rb_pred), Rb_low, Rb_pred)

Vp_pred_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_pred_filled)
Vs_pred_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_pred_filled)
Rb_pred_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_pred_filled)

Vp_low_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_low)
Vs_low_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_low)
Rb_low_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_low)

# unpack your three rows
vp_rows   = [Vp_low,    Vp_pred_filled,    Vp_true]
vs_rows   = [Vs_low,    Vs_pred_filled,    Vs_true]
rho_rows  = [Rb_low,    Rb_pred_filled,    Rb_true]
# compute shared scales
vmin_vp, vmax_vp   = np.nanmin(vp_rows),   np.nanmax(vp_rows)
vmin_vs, vmax_vs   = np.nanmin(vs_rows),   np.nanmax(vs_rows)
vmin_rho, vmax_rho = np.nanmin(rho_rows), np.nanmax(rho_rows)

# 5) now plot a 3×3 grid: rows = [Input, Pred, True], cols = [Vp, Vs, ρ]
fig, axes = plt.subplots(3, 3, figsize=(15, 12), sharex=True, sharey=True)

row_data = [
   ([Vp_low_crop,    Vs_low_crop,    Rb_low_crop],    "Input"),
   ([Vp_pred_crop,   Vs_pred_crop,   Rb_pred_crop],   "Predicted"),
    ([Vp_true,   Vs_true,   Rb_true],   "True")
]

props = ["Vp", "Vs", "Density"]

for i, (data_row, row_label) in enumerate(row_data):
    for j, (img, prop) in enumerate(zip(data_row, props)):
        ax = axes[i, j]
        if j == 0:   # first column is Vp
            vmn, vmx = vmin_vp, vmax_vp
        elif j == 1: # second is Vs
            vmn, vmx = vmin_vs, vmax_vs
        else:        # third is ρ
            vmn, vmx = vmin_rho, vmax_rho

        im = ax.imshow(
            img.T,
            origin='upper',
            extent=[ymin, ymax, zmax, zmin],
            vmin=vmn, vmax=vmx,
            aspect='auto',
            cmap='viridis'
        )
        if i == 0:
            ax.set_title(prop, fontsize=14)
        if j == 0:
            ax.set_ylabel(row_label, fontsize=14)
        fig.colorbar(im, ax=ax, shrink=0.75)

fig.supxlabel("Crossline (Y)")
fig.supylabel("Depth (Z)")
plt.tight_layout()
plt.show()
```

+*In[ ]:*+

[source, ipython3]

```
nz, nx, ny = 420, 288, 314
num_slices = 20
random_ix2030 = random_ix
filenames = ['AVO2030_d10.bin', 'AVO2030_d25.bin', 'AVO2030_d55.bin']
datadir = '../processing'
dz = 5
```

```
slice_dataset2030 = []
slice_metadata2030 = []
Vp_maxlist = np.zeros(num_slices, dtype=np.float32)
Vs_maxlist = np.zeros(num_slices, dtype=np.float32)
Rb_maxlist = np.zeros(num_slices, dtype=np.float32)
z_maxlist = np.zeros(num_slices, dtype=np.float32)
z_minlist = np.zeros(num_slices, dtype=np.float32)
# ymin, ymax is the same for all ix
ymin, ymax = np.min(dfss['y_coord']), np.max(dfss['y_coord'])

for idx, ix in enumerate(random_ix2030):
    dfss1 = dfss[dfss['i_index'] == ix]
    dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
    z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
    zmin, zmax = np.min(z_matrix), np.max(z_matrix)
    z_minlist[idx], z_maxlist[idx] = zmin, zmax
    top_z = z_matrix[:, 0]
    bottom_z = z_matrix[:, -1]
    start_gaps = np.round((zmax - top_z) / dz).astype(int)
    stop_gaps = np.round((bottom_z - zmin) / dz).astype(int)
    nzfill = int(np.max(start_gaps + nz + stop_gaps))
    slice_metadata2030.append({
        "slice_id": idx,
        "inline":   ix,
        "shape":    (ny, nzfill)
        })
        seismic_stack = np.full((ny, nzfill, 3), np.nan, dtype=np.float32)
        Vp_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
        Vs_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
        Rb_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)

        for i, fname in enumerate(filenames):
            filepath = os.path.join(datadir, fname)
            data = np.fromfile(filepath, dtype=np.float32)
            data_reshaped = data.reshape((ny, nx, nz)).transpose(1, 0, 2)
            seismic_slice = data_reshaped[ix, :, :]
            for j in range(ny):
                start_idx = start_gaps[j]
                end_idx = start_idx + nz
                if end_idx > nzfill:
                    length = nzfill - start_idx
                    seismic_stack[j, start_idx:end_idx, i] = seismic_slice[j, :length]
                    Vp_padded[j, start_idx:end_idx] =
dfss1_sorted['Vp2030'].values.reshape(ny, nz)[j, :length]
                    Vs_padded[j, start_idx:end_idx] =
dfss1_sorted['Vs2030'].values.reshape(ny, nz)[j, :length]
                    Rb_padded[j, start_idx:end_idx] =
dfss1_sorted['Rb2030'].values.reshape(ny, nz)[j, :length]
                else:
                    seismic_stack[j, start_idx:end_idx, i] = seismic_slice[j, :]
                    Vp_padded[j,                        start_idx:end_idx]                          =
dfss1_sorted['Vp2030'].values.reshape(ny, nz)[j, :]
                    Vs_padded[j,                        start_idx:end_idx]                          =
dfss1_sorted['Vs2030'].values.reshape(ny, nz)[j, :]
                    Rb_padded[j,                        start_idx:end_idx]                          =
dfss1_sorted['Rb2030'].values.reshape(ny, nz)[j, :]

            # Apply normalization + smoothing
            seismic_stack = seismic_stack / np.nanmax(np.abs(seismic_stack))
            Vp_low = nan_gaussian_filter_corrected(Vp_padded, sigma=5)
            Vs_low = nan_gaussian_filter_corrected(Vs_padded, sigma=5)
            Rb_low = nan_gaussian_filter_corrected(Rb_padded, sigma=5)
            # Normalize properties
            Vp_maxlist[idx] = np.nanmax(np.abs(Vp_padded))
            Vp_norm = Vp_padded / Vp_maxlist[idx]
            # print(Vp_maxlist[idx])

            Vs_maxlist[idx] = np.nanmax(np.abs(Vs_padded))
            Vs_norm = Vs_padded / Vs_maxlist[idx]
            # print(Vs_maxlist[idx])

            Rb_maxlist[idx] = np.nanmax(np.abs(Rb_padded))
            Rb_norm = Rb_padded / Rb_maxlist[idx]
            # print(Rb_maxlist[idx])
```

```
        Vp_low_norm = Vp_low / Vp_maxlist[idx]
        Vs_low_norm = Vs_low / Vs_maxlist[idx]
        Rb_low_norm = Rb_low / Rb_maxlist[idx]
        slice_dataset2030.append((seismic_stack, Vp_norm, Vp_low_norm, Vs_norm,
Vs_low_norm, Rb_norm, Rb_low_norm))

    all_datasets = []
    for slice_idx, s in enumerate(slice_dataset2030):  # <-- Now slice_idx is defined
        dataset = SeismicElasticPatchDataset(
            seismic_volume=s[0],   # (ny, nzfill, 3)
            vp=s[1], vp_low=s[2],
            vs=s[3], vs_low=s[4],
            rho=s[5], rho_low=s[6],
            patch_size=(50, 100),
            stride=(10, 25),
            nan_threshold=0.15,
            slice_id = slice_idx
        )
        # dataset.slice_id = slice_idx  # <-- This is now valid
        all_datasets.append(dataset)


    full_dataset2030 = ConcatDataset(all_datasets)

    n = len(full_dataset2030)
```

```
print(n)
train_size = int(0.7 * n)
val_size = int(0.15 * n)
test_size = n - train_size - val_size

train_set2030, val_set2030, test_set2030 = random_split(full_dataset2030, [train_size, val_size, test_size])

train_loader2030 = DataLoader(train_set2030, batch_size=16, shuffle=True, num_workers=4)
val_loader2030 = DataLoader(val_set2030, batch_size=16, shuffle=False, num_workers=2)
test_loader2030 = DataLoader(test_set2030, batch_size=16, shuffle=False, num_workers=2)
model = HCTNet2D(in_channels=3, hidden_dim=64, dropout_rate=0.1)

trained_model2030 = train_model(model, train_loader2030, val_loader2030, epochs=50)

slice_id_map = []
for sid, ds in enumerate(full_dataset2030.datasets):
   slice_id_map += [sid] * len(ds)
slice_id_map = np.array(slice_id_map, dtype=int)

# 2) Grab the subset indices from the torch.utils.data.Subset objects
```

```
train_idx = np.array(train_set2030.indices, dtype=int)
val_idx   = np.array(val_set2030.indices, dtype=int)
test_idx  = np.array(test_set2030.indices, dtype=int)

# 3) Tally up
records = []
for sid, inline in enumerate(random_ix2030):
    total = np.sum(slice_id_map == sid)
    train = np.sum(slice_id_map[train_idx] == sid)
    val   = np.sum(slice_id_map[val_idx] == sid)
    test  = np.sum(slice_id_map[test_idx] == sid)
    records.append({
        "slice_id":      sid,
        "inline_number": inline,
        "total_patches": total,
        "train":         train,
        "val":           val,
        "test":          test,
        "test_frac":     test/total if total else np.nan
    })

df_counts = pd.DataFrame(records)
print(df_counts.to_string(index=False))
# df_counts is  DataFrame of per-slice patch counts
# Locate the row with the largest “test” count
best_row = df_counts.loc[df_counts['test'].idxmax()]

# Extract its slice_id and inline number
best_sid    = int(best_row['slice_id'])
best_inline = best_row['inline_number']
best_n_test = best_row['test']

print(f"Slice ID with most test patches: {best_sid}")
print(f"inline number {best_inline} has {best_n_test} test patches")

# find the slice_id for the inline you're interested in
sid = random_ix2030.index(best_inline)    # e.g. inline k=150
print(sid)
# get the dataset and its shape
ny_nzf = full_dataset2030.datasets[sid].seismic.shape[:2]

# initialize coverage
coverage = np.zeros((ny_nzf[0], ny_nzf[1]), dtype=bool)

# mark each test patch
for idx in test_set2030.indices:  # train_set test_set
    if slice_id_map[idx] != sid:
        continue
    # convert global idx → (row_in_slice_dataset) by subtracting cumulative lengths
    offset = idx - sum(len(ds) for ds in full_dataset2030.datasets[:sid])
    i0, j0 = full_dataset2030.datasets[sid].indices[offset]  # origin of that patch
    coverage[i0:i0+50, j0:j0+100] = True

plt.figure(figsize=(6,8))
plt.imshow(coverage.T, origin='lower', aspect='auto', cmap='gray_r')
plt.title(f"Test-patch coverage on inline {best_inline}")
plt.xlabel("Crossline index")
plt.ylabel("Depth index")
plt.show()



test_slice_ids = set()
for idx in test_set2030.indices:
    # full_dataset[idx] returns a dict with 'slice_id': torch.Tensor(...)
    sid = full_dataset2030[idx]['slice_id']
    # it might be a tensor, so:
    test_slice_ids.add(int(sid))

# 2) filter your slice_metadata down to only those
test_slice_metadata2030 = [ md for md in slice_metadata2030
                        if md['slice_id'] in test_slice_ids ]

# 3) now build only those canvases
canvases2030 = evaluate_model_on_test_dynamic(
    trained_model2030,
    test_loader2030,
    test_slice_metadata2030,    # <-- pass *this*, not the full list
    patch_size=(50,100),
    device='cuda'
)

# 4) now `canvases[best_sid]` will only exist if best_sid ∈ test_slice_ids,
#    and since you picked best_sid as the inline with the most test patches,
#    it will have nonzero weight.
c = canvases2030[best_sid]

print("weight sum:", c['weight'].sum())



zmin, zmax = z_minlist[best_sid], z_maxlist[best_sid]

# 3) pull the *input* low-frequency fields from slice_dataset
_, Vp_norm, Vp_low_norm, Vs_norm, Vs_low_norm, Rb_norm, Rb_low_norm =
slice_dataset2030[best_sid]

# and de-normalize them using your per-slice max lists
Vp_low  = Vp_low_norm * Vp_maxlist[best_sid]
Vs_low  = Vs_low_norm * Vs_maxlist[best_sid]
Rb_low  = Rb_low_norm * Rb_maxlist[best_sid]
```

```
    # 4) pull the *true* and *predicted* canvases that you built in
evaluate_model_on_test_dynamic

    Vp_true = Vp_norm * Vp_maxlist[best_sid]
    Vs_true = Vs_norm * Vs_maxlist[best_sid]
    Rb_true = Rb_norm * Rb_maxlist[best_sid]

    Vp_pred = c['vp_pred'] * Vp_maxlist[best_sid]
    Vs_pred = c['vs_pred'] * Vs_maxlist[best_sid]
    Rb_pred = c['rho_pred'] * Rb_maxlist[best_sid]

    Vp_pred_filled = np.where(np.isnan(Vp_pred), Vp_low, Vp_pred)
    Vs_pred_filled = np.where(np.isnan(Vs_pred), Vs_low, Vs_pred)
    Rb_pred_filled = np.where(np.isnan(Rb_pred), Rb_low, Rb_pred)

    Vp_pred_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_pred_filled)
    Vs_pred_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_pred_filled)
    Rb_pred_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_pred_filled)

    Vp_low_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_low)
    Vs_low_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_low)
    Rb_low_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_low)

    # unpack your three rows
```

```
vp_rows   = [Vp_low,    Vp_pred_filled,    Vp_true]
vs_rows   = [Vs_low,    Vs_pred_filled,    Vs_true]
rho_rows  = [Rb_low,    Rb_pred_filled,    Rb_true]
# compute shared scales
vmin_vp, vmax_vp   = np.nanmin(vp_rows),   np.nanmax(vp_rows)
vmin_vs, vmax_vs   = np.nanmin(vs_rows),   np.nanmax(vs_rows)
vmin_rho, vmax_rho = np.nanmin(rho_rows), np.nanmax(rho_rows)

# 5) now plot a 3×3 grid: rows = [Input, Pred, True], cols = [Vp, Vs, ρ]
fig, axes = plt.subplots(3, 3, figsize=(15, 12), sharex=True, sharey=True)

row_data = [
    ([Vp_low_crop,    Vs_low_crop,    Rb_low_crop],    "Input"),
    ([Vp_pred_crop,   Vs_pred_crop,   Rb_pred_crop],   "Predicted"),
    ([Vp_true,   Vs_true,   Rb_true],   "True")
]

props = ["Vp", "Vs", "Density"]

for i, (data_row, row_label) in enumerate(row_data):
    for j, (img, prop) in enumerate(zip(data_row, props)):
        ax = axes[i, j]
        if j == 0:   # first column is Vp
            vmn, vmx = vmin_vp, vmax_vp
        elif j == 1: # second is Vs
```

```
            vmn, vmx = vmin_vs, vmax_vs
        else:        # third is ρ
            vmn, vmx = vmin_rho, vmax_rho

        im = ax.imshow(
            img.T,
            origin='upper',
            extent=[ymin, ymax, zmax, zmin],
            vmin=vmn, vmax=vmx,
            aspect='auto',
            cmap='viridis'
        )
        if i == 0:
            ax.set_title(prop, fontsize=14)
        if j == 0:
            ax.set_ylabel(row_label, fontsize=14)
        fig.colorbar(im, ax=ax, shrink=0.75)

fig.supxlabel("Crossline (Y)")
fig.supylabel("Depth (Z)")
plt.tight_layout()
plt.show()
```

```
+*In[ ]:*+

[source, ipython3]

----



----




+*In[ ]:*+

[source, ipython3]

----

z_maxlist = np.zeros(num_slices, dtype=np.float32)

z_minlist = np.zeros(num_slices, dtype=np.float32)

# ymin, ymax is the same for all ix

ymin, ymax = np.min(dfss['y_coord']), np.max(dfss['y_coord'])


for idx, ix in enumerate(random_ix2030):

    dfss1 = dfss[dfss['i_index'] == ix]

    dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])

    z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)

    zmin, zmax = np.min(z_matrix), np.max(z_matrix)

    z_minlist[idx], z_maxlist[idx] = zmin, zmax


slice_id_map = []
```

```
for sid, ds in enumerate(full_dataset2030.datasets):
    slice_id_map += [sid] * len(ds)
slice_id_map = np.array(slice_id_map, dtype=int)

# 2) Grab the subset indices from the torch.utils.data.Subset objects
train_idx = np.array(train_set2030.indices, dtype=int)
val_idx   = np.array(val_set2030.indices, dtype=int)
test_idx  = np.array(test_set2030.indices, dtype=int)

# 3) Tally up
records = []
for sid, inline in enumerate(random_ix2030):
    total = np.sum(slice_id_map == sid)
    train = np.sum(slice_id_map[train_idx] == sid)
    val   = np.sum(slice_id_map[val_idx] == sid)
    test  = np.sum(slice_id_map[test_idx] == sid)
    records.append({
        "slice_id":      sid,
        "inline_number": inline,
        "total_patches": total,
        "train":         train,
        "val":           val,
        "test":          test,
        "test_frac":     test/total if total else np.nan
    })
```

```
df_counts = pd.DataFrame(records)
print(df_counts.to_string(index=False))
# df_counts is  DataFrame of per-slice patch counts

# Locate the row with the largest "test" count
best_row = df_counts.loc[df_counts['test'].idxmax()]

# Extract its slice_id and inline number
best_sid    = int(best_row['slice_id'])
best_inline = best_row['inline_number']
best_n_test = best_row['test']

print(f"Slice ID with most test patches: {best_sid}")
print(f"inline number {best_inline} has {best_n_test} test patches")

# find the slice_id for the inline you're interested in
sid = random_ix2030.index(best_inline)    # e.g. inline k=150
print(sid)
# get the dataset and its shape
ny_nzf = full_dataset2030.datasets[sid].seismic.shape[:2]

# initialize coverage
coverage = np.zeros((ny_nzf[0], ny_nzf[1]), dtype=bool)
```

```
# mark each test patch
for idx in test_set2030.indices:  # train_set test_set
    if slice_id_map[idx] != sid:
        continue
    # convert global idx → (row_in_slice_dataset) by subtracting cumulative lengths
    offset = idx - sum(len(ds) for ds in full_dataset2030.datasets[:sid])
    i0, j0 = full_dataset2030.datasets[sid].indices[offset]  # origin of that patch
    coverage[i0:i0+50, j0:j0+100] = True

plt.figure(figsize=(6,8))
plt.imshow(coverage.T, origin='lower', aspect='auto', cmap='gray_r')
plt.title(f"Test-patch coverage on inline {best_inline}")
plt.xlabel("Crossline index")
plt.ylabel("Depth index")
plt.show()



test_slice_ids = set()
for idx in test_set2030.indices:
    # full_dataset[idx] returns a dict with 'slice_id': torch.Tensor(...)
    sid = full_dataset2030[idx]['slice_id']
    # it might be a tensor, so:
    test_slice_ids.add(int(sid))

# 2) filter your slice_metadata down to only those
```

```
test_slice_metadata2030 = [ md for md in slice_metadata2030
                          if md['slice_id'] in test_slice_ids ]

# 3) now build only those canvases
canvases2030 = evaluate_model_on_test_dynamic(
    trained_model2030,
    test_loader2030,
    test_slice_metadata2030,    # <-- pass *this*, not the full list
    patch_size=(50,100),
    device='cuda'
)
best_sid = 7
# 4) now `canvases[best_sid]` will only exist if best_sid ∈ test_slice_ids,
#    and since you picked best_sid as the inline with the most test patches,
#    it will have nonzero weight.
c = canvases2030[best_sid]

print("weight sum:", c['weight'].sum())



zmin, zmax = z_minlist[best_sid], z_maxlist[best_sid]

# 3) pull the *input* low-frequency fields from slice_dataset
_, Vp_norm, Vp_low_norm, Vs_norm, Vs_low_norm, Rb_norm, Rb_low_norm =
slice_dataset2030[best_sid]
```

```
# and de-normalize them using your per-slice max lists
Vp_low  = Vp_low_norm * Vp_maxlist[best_sid]
Vs_low  = Vs_low_norm * Vs_maxlist[best_sid]
Rb_low  = Rb_low_norm * Rb_maxlist[best_sid]



# 4) pull the *true* and *predicted* canvases that you built in evaluate_model_on_test_dynamic

Vp_true = Vp_norm * Vp_maxlist[best_sid]
Vs_true = Vs_norm * Vs_maxlist[best_sid]
Rb_true = Rb_norm * Rb_maxlist[best_sid]

Vp_pred = c['vp_pred'] * Vp_maxlist[best_sid]
Vs_pred = c['vs_pred'] * Vs_maxlist[best_sid]
Rb_pred = c['rho_pred'] * Rb_maxlist[best_sid]


Vp_pred_filled = np.where(np.isnan(Vp_pred), Vp_low, Vp_pred)
Vs_pred_filled = np.where(np.isnan(Vs_pred), Vs_low, Vs_pred)
Rb_pred_filled = np.where(np.isnan(Rb_pred), Rb_low, Rb_pred)

Vp_pred_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_pred_filled)
```

```
Vs_pred_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_pred_filled)
Rb_pred_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_pred_filled)

Vp_low_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_low)
Vs_low_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_low)
Rb_low_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_low)

# unpack your three rows
vp_rows   = [Vp_low,    Vp_pred_filled,    Vp_true]
vs_rows   = [Vs_low,    Vs_pred_filled,    Vs_true]
rho_rows  = [Rb_low,    Rb_pred_filled,    Rb_true]
# compute shared scales
vmin_vp, vmax_vp   = np.nanmin(vp_rows),   np.nanmax(vp_rows)
vmin_vs, vmax_vs   = np.nanmin(vs_rows),   np.nanmax(vs_rows)
vmin_rho, vmax_rho = np.nanmin(rho_rows), np.nanmax(rho_rows)

# 5) now plot a 3×3 grid: rows = [Input, Pred, True], cols = [Vp, Vs, ρ]
fig, axes = plt.subplots(3, 3, figsize=(15, 12), sharex=True, sharey=True)

row_data = [
   ([Vp_low_crop,    Vs_low_crop,    Rb_low_crop],    "Input"),
   ([Vp_pred_crop,   Vs_pred_crop,   Rb_pred_crop],   "Predicted"),
   ([Vp_true,   Vs_true,   Rb_true],   "True")
]
```

```
props = ["Vp", "Vs", "Density"]

for i, (data_row, row_label) in enumerate(row_data):
    for j, (img, prop) in enumerate(zip(data_row, props)):
        ax = axes[i, j]
        if j == 0:   # first column is Vp
            vmn, vmx = vmin_vp, vmax_vp
        elif j == 1: # second is Vs
            vmn, vmx = vmin_vs, vmax_vs
        else:        # third is ρ
            vmn, vmx = vmin_rho, vmax_rho

        im = ax.imshow(
            img.T,
            origin='upper',
            extent=[ymin, ymax, zmax, zmin],
            vmin=vmn, vmax=vmx,
            aspect='auto',
            cmap='viridis'
        )
        if i == 0:
            ax.set_title(prop, fontsize=14)
        if j == 0:
            ax.set_ylabel(row_label, fontsize=14)
        fig.colorbar(im, ax=ax, shrink=0.75)
fig.supxlabel("Crossline (Y)")
fig.supylabel("Depth (Z)")
plt.tight_layout()
plt.show()
----
```

+*In[ ]:*+

[source, ipython3]

```
----

nz, nx, ny = 420, 288, 314
num_slices = 20
# random_ix2050 = sorted(random.sample(range(10, nx - 10), num_slices))
random_ix2050 = random_ix
filenames = ['AVO2050_d10.bin', 'AVO2050_d25.bin', 'AVO2050_d55.bin']
datadir = '../processing'
dz = 5

slice_dataset2050 = []
slice_metadata2050 = []
Vp_maxlist = np.zeros(num_slices, dtype=np.float32)
```

```
Vs_maxlist = np.zeros(num_slices, dtype=np.float32)
Rb_maxlist = np.zeros(num_slices, dtype=np.float32)
z_maxlist = np.zeros(num_slices, dtype=np.float32)
z_minlist = np.zeros(num_slices, dtype=np.float32)
# ymin, ymax is the same for all ix
ymin, ymax = np.min(dfss['y_coord']), np.max(dfss['y_coord'])

for idx, ix in enumerate(random_ix2050):
    dfss1 = dfss[dfss['i_index'] == ix]
    dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
    z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
    zmin, zmax = np.min(z_matrix), np.max(z_matrix)
    z_minlist[idx], z_maxlist[idx] = zmin, zmax
    top_z = z_matrix[:, 0]
    bottom_z = z_matrix[:, -1]
    start_gaps = np.round((zmax - top_z) / dz).astype(int)
    stop_gaps = np.round((bottom_z - zmin) / dz).astype(int)
    nzfill = int(np.max(start_gaps + nz + stop_gaps))
    slice_metadata2050.append({
        "slice_id": idx,
        "inline":   ix,
        "shape":    (ny, nzfill)
    })
    seismic_stack = np.full((ny, nzfill, 3), np.nan, dtype=np.float32)
    Vp_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
```

```
            Vs_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
            Rb_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)

            for i, fname in enumerate(filenames):
                filepath = os.path.join(datadir, fname)
                data = np.fromfile(filepath, dtype=np.float32)
                data_reshaped = data.reshape((ny, nx, nz)).transpose(1, 0, 2)
                seismic_slice = data_reshaped[ix, :, :]
                for j in range(ny):
                    start_idx = start_gaps[j]
                    end_idx = start_idx + nz
                    if end_idx > nzfill:
                        length = nzfill - start_idx
                        seismic_stack[j, start_idx:end_idx, i] = seismic_slice[j, :length]
                        Vp_padded[j,                    start_idx:end_idx]              =
dfss1_sorted['Vp2050'].values.reshape(ny, nz)[j, :length]
                        Vs_padded[j,                    start_idx:end_idx]              =
dfss1_sorted['Vs2050'].values.reshape(ny, nz)[j, :length]
                        Rb_padded[j,                    start_idx:end_idx]              =
dfss1_sorted['Rb2050'].values.reshape(ny, nz)[j, :length]
                    else:
                        seismic_stack[j, start_idx:end_idx, i] = seismic_slice[j, :]
                        Vp_padded[j,                    start_idx:end_idx]              =
dfss1_sorted['Vp2050'].values.reshape(ny, nz)[j, :]
```

```
            Vs_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Vs2050'].values.reshape(ny, nz)[j, :]
            Rb_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Rb2050'].values.reshape(ny, nz)[j, :]

        # Apply normalization + smoothing
        seismic_stack = seismic_stack / np.nanmax(np.abs(seismic_stack))
        Vp_low = nan_gaussian_filter_corrected(Vp_padded, sigma=5)
        Vs_low = nan_gaussian_filter_corrected(Vs_padded, sigma=5)
        Rb_low = nan_gaussian_filter_corrected(Rb_padded, sigma=5)
        # Normalize properties
        Vp_maxlist[idx] = np.nanmax(np.abs(Vp_padded))
        Vp_norm = Vp_padded / Vp_maxlist[idx]
        # print(Vp_maxlist[idx])

        Vs_maxlist[idx] = np.nanmax(np.abs(Vs_padded))
        Vs_norm = Vs_padded / Vs_maxlist[idx]
        # print(Vs_maxlist[idx])

        Rb_maxlist[idx] = np.nanmax(np.abs(Rb_padded))
        Rb_norm = Rb_padded / Rb_maxlist[idx]
        # print(Rb_maxlist[idx])

        Vp_low_norm = Vp_low / Vp_maxlist[idx]
        Vs_low_norm = Vs_low / Vs_maxlist[idx]
        Rb_low_norm = Rb_low / Rb_maxlist[idx]
        slice_dataset2050.append((seismic_stack, Vp_norm, Vp_low_norm, Vs_norm,
Vs_low_norm, Rb_norm, Rb_low_norm))

    all_datasets = []
    for slice_idx, s in enumerate(slice_dataset2050):  # <-- Now slice_idx is defined
        dataset = SeismicElasticPatchDataset(
            seismic_volume=s[0],   # (ny, nzfill, 3)
            vp=s[1], vp_low=s[2],
            vs=s[3], vs_low=s[4],
            rho=s[5], rho_low=s[6],
            patch_size=(50, 100),
            stride=(10, 25),
            nan_threshold=0.15,
            slice_id = slice_idx
        )
        # dataset.slice_id = slice_idx  # <-- This is now valid
        all_datasets.append(dataset)


    full_dataset2050 = ConcatDataset(all_datasets)

    n = len(full_dataset2050)
    print(n)
    train_size = int(0.7 * n)
```

```
val_size = int(0.15 * n)
test_size = n - train_size - val_size

train_set2050, val_set2050, test_set2050 = random_split(full_dataset2050, [train_size, val_size, test_size])



train_loader2050 = DataLoader(train_set2050, batch_size=16, shuffle=True, num_workers=4)
val_loader2050 = DataLoader(val_set2050, batch_size=16, shuffle=False, num_workers=2)
test_loader2050 = DataLoader(test_set2050, batch_size=16, shuffle=False, num_workers=2)
model = HCTNet2D(in_channels=3, hidden_dim=64, dropout_rate=0.1)

trained_model2050 = train_model(model, train_loader2050, val_loader2050, epochs=50)



slice_id_map = []
for sid, ds in enumerate(full_dataset2050.datasets):
   slice_id_map += [sid] * len(ds)
slice_id_map = np.array(slice_id_map, dtype=int)
```

```
# 2) Grab the subset indices from the torch.utils.data.Subset objects
train_idx = np.array(train_set2050.indices, dtype=int)
val_idx   = np.array(val_set2050.indices, dtype=int)
test_idx  = np.array(test_set2050.indices, dtype=int)

# 3) Tally up
records = []
for sid, inline in enumerate(random_ix2050):
    total = np.sum(slice_id_map == sid)
    train = np.sum(slice_id_map[train_idx] == sid)
    val   = np.sum(slice_id_map[val_idx] == sid)
    test  = np.sum(slice_id_map[test_idx] == sid)
    records.append({
        "slice_id":      sid,
        "inline_number": inline,
        "total_patches": total,
        "train":         train,
        "val":           val,
        "test":          test,
        "test_frac":     test/total if total else np.nan
    })

df_counts = pd.DataFrame(records)
print(df_counts.to_string(index=False))
```

```
# df_counts is  DataFrame of per-slice patch counts

# Locate the row with the largest “test” count
best_row = df_counts.loc[df_counts['test'].idxmax()]

# Extract its slice_id and inline number
best_sid    = int(best_row['slice_id'])
best_inline = best_row['inline_number']
best_n_test = best_row['test']

print(f"Slice ID with most test patches: {best_sid}")
print(f"inline number {best_inline} has {best_n_test} test patches")

# find the slice_id for the inline you're interested in
sid = random_ix2050.index(best_inline)    # e.g. inline k=150
print(sid)
# get the dataset and its shape
ny_nzf = full_dataset2050.datasets[sid].seismic.shape[:2]

# initialize coverage
coverage = np.zeros((ny_nzf[0], ny_nzf[1]), dtype=bool)

# mark each test patch
for idx in test_set2050.indices:  # train_set test_set
    if slice_id_map[idx] != sid:
        continue
    # convert global idx → (row_in_slice_dataset) by subtracting cumulative lengths
    offset = idx - sum(len(ds) for ds in full_dataset2050.datasets[:sid])
    i0, j0 = full_dataset2050.datasets[sid].indices[offset]  # origin of that patch
    coverage[i0:i0+50, j0:j0+100] = True

plt.figure(figsize=(6,8))
plt.imshow(coverage.T, origin='lower', aspect='auto', cmap='gray_r')
plt.title(f"Test-patch coverage on inline {best_inline}")
plt.xlabel("Crossline index")
plt.ylabel("Depth index")
plt.show()



test_slice_ids = set()
for idx in test_set2050.indices:
    # full_dataset[idx] returns a dict with 'slice_id': torch.Tensor(...)
    sid = full_dataset2050[idx]['slice_id']
    # it might be a tensor, so:
    test_slice_ids.add(int(sid))

# 2) filter your slice_metadata down to only those
test_slice_metadata2050 = [ md for md in slice_metadata2050
                    if md['slice_id'] in test_slice_ids ]
```

```
# 3) now build only those canvases
canvases2050 = evaluate_model_on_test_dynamic(
    trained_model2050,
    test_loader2050,
    test_slice_metadata2050,    # <-- pass *this*, not the full list
    patch_size=(50,100),
    device='cuda'
)

# 4) now `canvases[best_sid]` will only exist if best_sid ∈ test_slice_ids,
#    and since you picked best_sid as the inline with the most test patches,
#    it will have nonzero weight.
c = canvases2050[best_sid]

print("weight sum:", c['weight'].sum())



zmin, zmax = z_minlist[best_sid], z_maxlist[best_sid]

# 3) pull the *input* low-frequency fields from slice_dataset
_, Vp_norm, Vp_low_norm, Vs_norm, Vs_low_norm, Rb_norm, Rb_low_norm =
slice_dataset2050[best_sid]

# and de-normalize them using your per-slice max lists
Vp_low  = Vp_low_norm * Vp_maxlist[best_sid]
```

```
Vs_low  = Vs_low_norm * Vs_maxlist[best_sid]
Rb_low  = Rb_low_norm * Rb_maxlist[best_sid]


# 4) pull the *true* and *predicted* canvases that you built in evaluate_model_on_test_dynamic

Vp_true = Vp_norm * Vp_maxlist[best_sid]
Vs_true = Vs_norm * Vs_maxlist[best_sid]
Rb_true = Rb_norm * Rb_maxlist[best_sid]

Vp_pred = c['vp_pred'] * Vp_maxlist[best_sid]
Vs_pred = c['vs_pred'] * Vs_maxlist[best_sid]
Rb_pred = c['rho_pred'] * Rb_maxlist[best_sid]


Vp_pred_filled = np.where(np.isnan(Vp_pred), Vp_low, Vp_pred)
Vs_pred_filled = np.where(np.isnan(Vs_pred), Vs_low, Vs_pred)
Rb_pred_filled = np.where(np.isnan(Rb_pred), Rb_low, Rb_pred)

Vp_pred_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_pred_filled)
Vs_pred_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_pred_filled)
Rb_pred_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_pred_filled)
```

```
Vp_low_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_low)
Vs_low_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_low)
Rb_low_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_low)

# unpack your three rows
vp_rows   = [Vp_low,    Vp_pred_filled,    Vp_true]
vs_rows   = [Vs_low,    Vs_pred_filled,    Vs_true]
rho_rows  = [Rb_low,    Rb_pred_filled,    Rb_true]
# compute shared scales
vmin_vp, vmax_vp   = np.nanmin(vp_rows),   np.nanmax(vp_rows)
vmin_vs, vmax_vs   = np.nanmin(vs_rows),   np.nanmax(vs_rows)
vmin_rho, vmax_rho = np.nanmin(rho_rows), np.nanmax(rho_rows)

# 5) now plot a 3×3 grid: rows = [Input, Pred, True], cols = [Vp, Vs, ρ]
fig, axes = plt.subplots(3, 3, figsize=(15, 12), sharex=True, sharey=True)

row_data = [
   ([Vp_low_crop,    Vs_low_crop,    Rb_low_crop],    "Input"),
   ([Vp_pred_crop,   Vs_pred_crop,   Rb_pred_crop],   "Predicted"),
   ([Vp_true,   Vs_true,   Rb_true],   "True")
]

props = ["Vp", "Vs", "Density"]

for i, (data_row, row_label) in enumerate(row_data):
        for j, (img, prop) in enumerate(zip(data_row, props)):
            ax = axes[i, j]
            if j == 0:   # first column is Vp
                vmn, vmx = vmin_vp, vmax_vp
            elif j == 1: # second is Vs
                vmn, vmx = vmin_vs, vmax_vs
            else:        # third is ρ
                vmn, vmx = vmin_rho, vmax_rho

            im = ax.imshow(
                img.T,
                origin='upper',
                extent=[ymin, ymax, zmax, zmin],
                vmin=vmn, vmax=vmx,
                aspect='auto',
                cmap='viridis'
            )
            if i == 0:
                ax.set_title(prop, fontsize=14)
            if j == 0:
                ax.set_ylabel(row_label, fontsize=14)
            fig.colorbar(im, ax=ax, shrink=0.75)

    fig.supxlabel("Crossline (Y)")
    fig.supylabel("Depth (Z)")
```

```
plt.tight_layout()
plt.show()

----



+*In[ ]:*+
[source, ipython3]
----
z_maxlist = np.zeros(num_slices, dtype=np.float32)
z_minlist = np.zeros(num_slices, dtype=np.float32)
# ymin, ymax is the same for all ix
ymin, ymax = np.min(dfss['y_coord']), np.max(dfss['y_coord'])

for idx, ix in enumerate(random_ix2050):
    dfss1 = dfss[dfss['i_index'] == ix]
    dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
    z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
    zmin, zmax = np.min(z_matrix), np.max(z_matrix)
    z_minlist[idx], z_maxlist[idx] = zmin, zmax

----



+*In[ ]:*+
```

[source, ipython3]

```
slice_id_map = []
for sid, ds in enumerate(full_dataset2050.datasets):
    slice_id_map += [sid] * len(ds)
slice_id_map = np.array(slice_id_map, dtype=int)

# 2) Grab the subset indices from the torch.utils.data.Subset objects
train_idx = np.array(train_set2050.indices, dtype=int)
val_idx   = np.array(val_set2050.indices, dtype=int)
test_idx  = np.array(test_set2050.indices, dtype=int)

# 3) Tally up
records = []
for sid, inline in enumerate(random_ix2050):
    total = np.sum(slice_id_map == sid)
    train = np.sum(slice_id_map[train_idx] == sid)
    val   = np.sum(slice_id_map[val_idx] == sid)
    test  = np.sum(slice_id_map[test_idx] == sid)
    records.append({
        "slice_id":      sid,
        "inline_number": inline,
        "total_patches": total,
        "train":         train,
        "val":           val,
        "test":          test,
        "test_frac":     test/total if total else np.nan
    })

df_counts = pd.DataFrame(records)
print(df_counts.to_string(index=False))
# df_counts is  DataFrame of per-slice patch counts

# Locate the row with the largest “test” count
best_row = df_counts.loc[df_counts['test'].idxmax()]

# Extract its slice_id and inline number
best_sid    = int(best_row['slice_id'])
best_inline = best_row['inline_number']
best_n_test = best_row['test']

print(f"Slice ID with most test patches: {best_sid}")
print(f"inline number {best_inline} has {best_n_test} test patches")

# find the slice_id for the inline you're interested in
sid = random_ix2050.index(best_inline)    # e.g. inline k=150
print(sid)
# get the dataset and its shape
ny_nzf = full_dataset2050.datasets[sid].seismic.shape[:2]
```

```
# initialize coverage
coverage = np.zeros((ny_nzf[0], ny_nzf[1]), dtype=bool)

# mark each test patch
for idx in test_set2050.indices:  # train_set test_set
    if slice_id_map[idx] != sid:
        continue
    # convert global idx → (row_in_slice_dataset) by subtracting cumulative lengths
    offset = idx - sum(len(ds) for ds in full_dataset2050.datasets[:sid])
    i0, j0 = full_dataset2050.datasets[sid].indices[offset]  # origin of that patch
    coverage[i0:i0+50, j0:j0+100] = True

plt.figure(figsize=(6,8))
plt.imshow(coverage.T, origin='lower', aspect='auto', cmap='gray_r')
plt.title(f"Test-patch coverage on inline {best_inline}")
plt.xlabel("Crossline index")
plt.ylabel("Depth index")
plt.show()



test_slice_ids = set()
for idx in test_set2050.indices:
    # full_dataset[idx] returns a dict with 'slice_id': torch.Tensor(...)
    sid = full_dataset2050[idx]['slice_id']
    # it might be a tensor, so:
    test_slice_ids.add(int(sid))

# 2) filter your slice_metadata down to only those
test_slice_metadata2050 = [ md for md in slice_metadata2050
                        if md['slice_id'] in test_slice_ids ]

# 3) now build only those canvases
canvases2050 = evaluate_model_on_test_dynamic(
    trained_model2050,
    test_loader2050,
    test_slice_metadata2050,    # <-- pass *this*, not the full list
    patch_size=(50,100),
    device='cuda'
)
best_sid = 9
# 4) now `canvases[best_sid]` will only exist if best_sid ∈ test_slice_ids,
#    and since you picked best_sid as the inline with the most test patches,
#    it will have nonzero weight.
c = canvases2050[best_sid]

print("weight sum:", c['weight'].sum())



zmin, zmax = z_minlist[best_sid], z_maxlist[best_sid]
```

```
# 3) pull the *input* low-frequency fields from slice_dataset
_, Vp_norm, Vp_low_norm, Vs_norm, Vs_low_norm, Rb_norm, Rb_low_norm = slice_dataset2050[best_sid]

# and de-normalize them using your per-slice max lists
Vp_low  = Vp_low_norm * Vp_maxlist[best_sid]
Vs_low  = Vs_low_norm * Vs_maxlist[best_sid]
Rb_low  = Rb_low_norm * Rb_maxlist[best_sid]



# 4) pull the *true* and *predicted* canvases that you built in evaluate_model_on_test_dynamic
c = canvases2050[best_sid]
Vp_true = Vp_norm * Vp_maxlist[best_sid]
Vs_true = Vs_norm * Vs_maxlist[best_sid]
Rb_true = Rb_norm * Rb_maxlist[best_sid]

Vp_pred = c['vp_pred'] * Vp_maxlist[best_sid]
Vs_pred = c['vs_pred'] * Vs_maxlist[best_sid]
Rb_pred = c['rho_pred'] * Rb_maxlist[best_sid]


Vp_pred_filled = np.where(np.isnan(Vp_pred), Vp_low, Vp_pred)
```

```
Vs_pred_filled = np.where(np.isnan(Vs_pred), Vs_low, Vs_pred)
Rb_pred_filled = np.where(np.isnan(Rb_pred), Rb_low, Rb_pred)

Vp_pred_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_pred_filled)
Vs_pred_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_pred_filled)
Rb_pred_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_pred_filled)

Vp_low_crop = np.where(np.isnan(Vp_true), Vp_true, Vp_low)
Vs_low_crop = np.where(np.isnan(Vs_true), Vs_true, Vs_low)
Rb_low_crop = np.where(np.isnan(Rb_true), Rb_true, Rb_low)

# unpack your three rows
vp_rows  = [Vp_low,   Vp_pred_filled,   Vp_true]
vs_rows  = [Vs_low,   Vs_pred_filled,   Vs_true]
rho_rows = [Rb_low,   Rb_pred_filled,   Rb_true]
# compute shared scales
vmin_vp, vmax_vp   = np.nanmin(vp_rows),   np.nanmax(vp_rows)
vmin_vs, vmax_vs   = np.nanmin(vs_rows),   np.nanmax(vs_rows)
vmin_rho, vmax_rho = np.nanmin(rho_rows), np.nanmax(rho_rows)

# 5) now plot a 3×3 grid: rows = [Input, Pred, True], cols = [Vp, Vs, ρ]
fig, axes = plt.subplots(3, 3, figsize=(15, 12), sharex=True, sharey=True)

row_data = [
   ([Vp_low_crop,   Vs_low_crop,   Rb_low_crop],   "Input"),
    ([Vp_pred_crop,   Vs_pred_crop,   Rb_pred_crop],   "Predicted"),
    ([Vp_true,   Vs_true,   Rb_true],   "True")
]

props = ["Vp", "Vs", "Density"]

for i, (data_row, row_label) in enumerate(row_data):
    for j, (img, prop) in enumerate(zip(data_row, props)):
        ax = axes[i, j]
        if j == 0:   # first column is Vp
            vmn, vmx = vmin_vp, vmax_vp
        elif j == 1: # second is Vs
            vmn, vmx = vmin_vs, vmax_vs
        else:        # third is ρ
            vmn, vmx = vmin_rho, vmax_rho

        im = ax.imshow(
            img.T,
            origin='upper',
            extent=[ymin, ymax, zmax, zmin],
            vmin=vmn, vmax=vmx,
            aspect='auto',
            cmap='viridis'
        )
        if i == 0:
            ax.set_title(prop, fontsize=14)
        if j == 0:
            ax.set_ylabel(row_label, fontsize=14)
        fig.colorbar(im, ax=ax, shrink=0.75)

fig.supxlabel("Crossline (Y)")
fig.supylabel("Depth (Z)")
plt.tight_layout()
plt.show()
----
```

+*In[ ]:*+

[source, ipython3]

```
def compute_delta_maps(canvases_ref, canvases_target, max_list, prop="vp"):
    """
    Compute delta maps (difference from reference year) for a given property across
slices.
    Args:
        canvases_ref: dict of canvases (reference year)
        canvases_target: dict of canvases (target year)
        max_list: normalization constants per slice
        prop: one of 'vp', 'vs', or 'rho'
    Returns:
        delta_maps: list of 2D numpy arrays (delta per slice)
    """
    delta_maps = []
    for sid in range(len(canvases_ref)):
        #       print(f'sid:{sid}       ref:{canvases_ref[sid][f"{prop}_pred"].shape},
tgt:{canvases_target[sid][f"{prop}_pred"].shape} ')

        ref = canvases_ref[sid][f"{prop}_pred"] * max_list[sid]
        tgt = canvases_target[sid][f"{prop}_pred"] * max_list[sid]
        delta = tgt - ref
        delta_maps.append(delta)
    return delta_maps


def plot_delta_vs_co2scatter(delta_maps, sg_maps, prop_name, title):
    """
    Plot scatter plots of ΔProperty vs CO₂ saturation.
    Args:
        delta_maps: list of ΔProperty (2D arrays)
        sg_maps: list of CO₂ saturation maps (2D arrays)
        prop_name: 'Vp', 'Vs', or 'Density'
    """
    plt.figure(figsize=(7, 5))
    for delta, sg in zip(delta_maps, sg_maps):
        mask = (~np.isnan(delta)) & (delta!=0) & (~np.isnan(sg)) & (sg>1e-4)
        plt.scatter(sg[mask], delta[mask], s=1, alpha=0.3)

    plt.xlabel("CO₂ Saturation")
    plt.ylabel(f"Δ{prop_name} ({title} - 2024)")
    plt.title(f"Sensitivity of {prop_name} to CO₂ in year {title}")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

def plot_property_change(delta, title, vmin=None, vmax=None):
    plt.figure(figsize=(8, 6))
    im = plt.imshow(delta.T, cmap='bwr', origin='upper',
                    extent=[ymin, ymax, zmax, zmin],
                    aspect='auto', vmin=vmin, vmax=vmax)
    plt.colorbar(im)
    plt.title(f"Δ {title} (2030 - 2024)")
    plt.xlabel("Crossline (Y)")
    plt.ylabel("Depth (Z)")
    plt.tight_layout()
    plt.show()

----
```

+*In[ ]:*+

[source, ipython3]

```
----
delta_vp_2030 = compute_delta_maps(canvases, canvases2030, Vp_maxlist,
prop="vp")
delta_vp_2050 = compute_delta_maps(canvases, canvases2050, Vp_maxlist,
prop="vp")
delta_vp_2070 = compute_delta_maps(canvases, canvases2070, Vp_maxlist,
prop="vp")
delta_vs_2030 = compute_delta_maps(canvases, canvases2030, Vs_maxlist,
prop="vs")
delta_vs_2050 = compute_delta_maps(canvases, canvases2050, Vs_maxlist,
prop="vs")
delta_vs_2070 = compute_delta_maps(canvases, canvases2070, Vs_maxlist,
prop="vs")
delta_rho_2030 = compute_delta_maps(canvases, canvases2030, Rb_maxlist,
prop="rho")
delta_rho_2050 = compute_delta_maps(canvases, canvases2050, Rb_maxlist,
prop="rho")
delta_rho_2070 = compute_delta_maps(canvases, canvases2070, Rb_maxlist,
prop="rho")
```

+*In[ ]:*+

[source, ipython3]

```
slice_Sg2030 = []
slice_Sg2050 = []
slice_Sg2070 = []

for idx, ix in enumerate(random_ix):
    dfss1 = dfss[dfss['i_index'] == ix]
    dfss1_sorted = dfss1.sort_values(by=['j_index', 'k_index'])
    z_matrix = dfss1_sorted['z_coord'].values.reshape(ny, nz)
    zmin, zmax = np.min(z_matrix), np.max(z_matrix)
    top_z = z_matrix[:, 0]
    bottom_z = z_matrix[:, -1]
    start_gaps = np.round((zmax - top_z) / dz).astype(int)
    stop_gaps = np.round((bottom_z - zmin) / dz).astype(int)
    nzfill = int(np.max(start_gaps + nz + stop_gaps))
    Sg2030_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
    Sg2050_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
    Sg2070_padded = np.full((ny, nzfill), np.nan, dtype=np.float32)
    for j in range(ny):
        start_idx = start_gaps[j]
        end_idx = start_idx + nz
        if end_idx > nzfill:
          length = nzfill - start_idx
          Sg2030_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Sg2030'].values.reshape(ny, nz)[j, :length]
          Sg2030_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Sg2050'].values.reshape(ny, nz)[j, :length]
          Sg2030_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Sg2070'].values.reshape(ny, nz)[j, :length]
        else:
          Sg2030_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Sg2030'].values.reshape(ny, nz)[j, :]
          Sg2050_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Sg2050'].values.reshape(ny, nz)[j, :]
          Sg2070_padded[j,                    start_idx:end_idx]                    =
dfss1_sorted['Sg2070'].values.reshape(ny, nz)[j, :]

      slice_Sg2030.append(Sg2030_padded)
      slice_Sg2050.append(Sg2050_padded)
      slice_Sg2070.append(Sg2070_padded)


    ----
```

+*In[ ]:*+

[source, ipython3]

```
----

plot_delta_vs_co2scatter(delta_vp_2030, slice_Sg2030, "Vp","2030")
plot_delta_vs_co2scatter(delta_vp_2070, slice_Sg2070, "Vp","2070")
----
```

+*In[ ]:*+

[source, ipython3]

```
----
def flatten_clean(data_list):
    return np.concatenate([d[~np.isnan(d)].flatten() for d in data_list])
def compute_sensitivity_range(sg_vals, delta_vals, bins=10):
    bin_edges = np.linspace(0, 0.5, bins+1)
    ranges = []
    for i in range(bins):
        mask = (sg_vals >= bin_edges[i]) & (sg_vals < bin_edges[i+1])
        vals_in_bin = delta_vals[mask]
        if len(vals_in_bin) > 0:
            mean = np.mean(vals_in_bin)
            std = np.std(vals_in_bin)
            ranges.append((mean - std, mean + std))
        else:
            ranges.append((0, 0))
    return np.array(ranges), bin_edges
sg_vals2030 = flatten_clean(slice_Sg2030)  # (2637600,)
vp_vals2030 = flatten_clean(delta_vp_2030) # (1876278,)
vs_vals2030 = flatten_clean(delta_vs_2030)
rho_vals2030 = flatten_clean(delta_rho_2030)
----
```

+*In[ ]:*+

[source, ipython3]

```
----
sg_vals2030, vp_vals2030 = flatten_clean_pair(slice_Sg2030, delta_vp_2030)
_, vs_vals2030 = flatten_clean_pair(slice_Sg2030, delta_vs_2030)
_, rho_vals2030 = flatten_clean_pair(slice_Sg2030, delta_rho_2030)

----
```

+*In[ ]:*+

[source, ipython3]

```
----
def flatten_clean_pair(sg_slices, delta_slices):
    """Takes two lists of 2D arrays and returns flat arrays of matched non-NaN values."""
    sg_all = []
    delta_all = []

    for sg, delta in zip(sg_slices, delta_slices):
        if sg.shape != delta.shape:
            print("Shape mismatch:", sg.shape, delta.shape)
            continue  # skip mismatched slices
        mask = (~np.isnan(sg)) & (~np.isnan(delta))
        sg_all.append(sg[mask])
        delta_all.append(delta[mask])

    sg_flat = np.concatenate(sg_all)
    delta_flat = np.concatenate(delta_all)
    return sg_flat, delta_flat


# Apply to your data
sg_vals2030, vp_vals2030 = flatten_clean_pair(slice_Sg2030, delta_vp_2030)
_, vs_vals2030 = flatten_clean_pair(slice_Sg2030, delta_vs_2030)
_, rho_vals2030 = flatten_clean_pair(slice_Sg2030, delta_rho_2030)

# Then use your plotting code
vp_range, edges = compute_sensitivity_range(sg_vals2030, vp_vals2030)
```

```
vs_range, _     = compute_sensitivity_range(sg_vals2030, vs_vals2030)
rho_range, _    = compute_sensitivity_range(sg_vals2030, rho_vals2030)

labels = [f"{edges[i]:.2f}-{edges[i+1]:.2f}" for i in range(len(edges)-1)]
bar_width = 0.25
x = np.arange(len(labels))

fig, ax = plt.subplots(figsize=(12, 6))
ax.barh(x - bar_width, vp_range[:,1] - vp_range[:,0], bar_width, left=vp_range[:,0], label='Vp')
ax.barh(x, vs_range[:,1] - vs_range[:,0], bar_width, left=vs_range[:,0], label='Vs')
# ax.barh(x + bar_width, rho_range[:,1] - rho_range[:,0], bar_width, left=rho_range[:,0], label='Density')

ax.set_yticks(x)
ax.set_yticklabels(labels)
ax.set_xlabel("ΔProperty Value Range")
ax.set_title("Sensitivity of Properties to CO₂ Saturation (by bins)")
ax.legend()
ax.grid(True)

plt.tight_layout()
plt.show()
----
```

```
+*In[ ]:*+
[source, ipython3]
----
max(s[2].shape[1] for s in slice_dataset)
----
```

```
+*In[ ]:*+
[source, ipython3]
----
# ny = max(s[0].shape[0] for s in slice_dataset)
newnz = max(s[0].shape[1] for s in slice_dataset)
def pad_to_shape(arr, target_shape):
    pad_y = target_shape[0] - arr.shape[0]
    pad_z = target_shape[1] - arr.shape[1]
    return np.pad(arr, ((0, pad_y), (0, pad_z)), mode='constant', constant_values=np.nan)

AVO2024_d10 = [pad_to_shape(s[0][:, :, 0], (ny, newnz)) for s in slice_dataset]
AVO2024_d25 = [pad_to_shape(s[0][:, :, 1], (ny, newnz)) for s in slice_dataset]
AVO2024_d55 = [pad_to_shape(s[0][:, :, 2], (ny, newnz)) for s in slice_dataset]
```

```
AVO2030_d10 = [pad_to_shape(s[0][:, :, 0], (ny, newnz)) for s in slice_dataset2030]
AVO2030_d25 = [pad_to_shape(s[0][:, :, 1], (ny, newnz)) for s in slice_dataset2030]
AVO2030_d55 = [pad_to_shape(s[0][:, :, 2], (ny, newnz)) for s in slice_dataset2030]

AVO2024_d10 = np.stack(AVO2024_d10, axis=0)
AVO2024_d25 = np.stack(AVO2024_d25, axis=0)
AVO2024_d55 = np.stack(AVO2024_d55, axis=0)

AVO2030_d10 = np.stack(AVO2030_d10, axis=0)
AVO2030_d25 = np.stack(AVO2030_d25, axis=0)
AVO2030_d55 = np.stack(AVO2030_d55, axis=0)

delta_AVO2030_d10 = AVO2030_d10 - AVO2024_d10
delta_AVO2030_d25 = AVO2030_d25 - AVO2024_d25
delta_AVO2030_d55 = AVO2030_d55 - AVO2024_d55

slice_Sg2030_padto = [pad_to_shape(s, (ny, newnz)) for s in slice_Sg2030]
slice_Sg2030_padto = np.stack(slice_Sg2030_padto, axis=0)  # shape: (20, ny, nz)
----
```

+*In[ ]:*+

```
[source, ipython3]
----
slice_Sg2030_padto = [pad_to_shape(s, (ny, newnz)) for s in slice_Sg2030]
slice_Sg2030_padto = np.stack(slice_Sg2030_padto, axis=0)  # shape: (20, ny, nz)

----



+*In[ ]:*+
[source, ipython3]
----
def flatten_clean_pair(slice_list_base, slice_list_target):
    base_all = []
    target_all = []
    for i, (b, t) in enumerate(zip(slice_list_base, slice_list_target)):
        b = np.array(b)
        t = np.array(t)
        if b.shape != t.shape:
            print(f"[WARN] Skipping index {i} due to shape mismatch: base={b.shape}, target={t.shape}")
            continue
        mask = (~np.isnan(b)) & (~np.isnan(t))
        base_all.append(b[mask])
        target_all.append(t[mask])
    return np.concatenate(base_all), np.concatenate(target_all)
```

```
sg_vals2030, avo10_vals = flatten_clean_pair(slice_Sg2030_padto, delta_AVO2030_d10)
_, avo25_vals = flatten_clean_pair(slice_Sg2030_padto, delta_AVO2030_d25)
_, avo55_vals = flatten_clean_pair(slice_Sg2030_padto, delta_AVO2030_d55)


----


+*In[ ]:*+
[source, ipython3]
----
# flatten and clean



def compute_sensitivity_range(base, delta_vals, bins=10):
    bin_edges = np.linspace(np.min(base), np.max(base), bins+1)
    bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])
    value_ranges = []

    for i in range(bins):
        mask = (base >= bin_edges[i]) & (base < bin_edges[i+1])
```

```
            vals = delta_vals[mask]
            if len(vals) > 0:
                value_range = np.percentile(vals, 95) - np.percentile(vals, 5)  # Robust
range
            else:
                value_range = 0
            value_ranges.append(value_range)

        return bin_centers, value_ranges
    bins = 10
    bin_centers, range_10 = compute_sensitivity_range(sg_vals2030, avo10_vals,
bins=bins)
    _, range_25 = compute_sensitivity_range(sg_vals2030, avo25_vals, bins=bins)
    _, range_55 = compute_sensitivity_range(sg_vals2030, avo55_vals, bins=bins)



    y_labels = [f"{bin_centers[i]:.2f}-{bin_centers[i+1]:.2f}" if i+1 < len(bin_centers)
else f"{bin_centers[i]:.2f}+" for i in range(bins)]

    fig, ax = plt.subplots(figsize=(10, 6))
    width = 0.2
    y = np.arange(len(y_labels))

    ax.barh(y - width, range_10, height=width, label='AVO @ 10°')
    ax.barh(y,       range_25, height=width, label='AVO @ 25°')
    ax.barh(y + width, range_55, height=width, label='AVO @ 55°')

    ax.set_yticks(y)
    ax.set_yticklabels(y_labels)
    ax.set_xlabel("ΔAVO Reflectivity Range")
    ax.set_title("Sensitivity of AVO Angles to CO₂ Saturation (2030)")
    ax.legend()
    # ax.invert_yaxis()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

+*In[ ]:*+

[source, ipython3]

APPENDIX E: SEISMIC FORWARD MODELING MADAGASCAR CODE (FOMEL, 2024; FOMEL ET AL., 2013; GAO ET AL., UNPUBLISHED)

```
from rsf.proj import *


#Flow('FID','FID.txt','asc2rsf')
#Flow('x_coor','x_coord.txt','dd form=native')


#git add data
#mv ~/DOwnload/FID.gslib data



# work on facies


Flow('FIDdata','faciesCorr.txt',
    '''
    echo n1=7 n2=37981440 data_format=ascii_float
    in=$SOURCE
    key1=i key2=j key3=k key4=x key5=y key6=z key7=fid
    | dd form=native
    ''', stdin=0)


# sfheaderattr < FID.rsf segy=n
Flow('ijk','FIDdata','window n1=3 | dd type=int')


Flow('FID','FIDdata ijk',
    '''
```

```
        window n1=1 f1=6 squeeze=n |
        intbin3 head=${SOURCES[1]} xkey=0 ykey=1 zkey=2 |
        window |
        put d1=0.250 d2=0.250 label1=X label2=Y o1=1137.825 o2=10801.125
        ''')
    Result('FID',
         '''
         byte gainpanel=all bar=bar.rsf |
         transp plane=12 | transp plane=13 |
         grey3 color=j frame1=0 frame2=150 frame3=200 point1=0.25 point2=0.70
         flat=n  title=FID  scalebar=y   unit1=ft  label1=Z  label2=X  unit2="x1000  ft"
label3=Y unit3="x1000 ft"
         ''')

    Flow('xy','FIDdata ijk',
        '''
        window n1=2 f1=3 squeeze=n |
        intbin3 head=${SOURCES[1]} xkey=0 ykey=1 zkey=2 |
        dd type=complex | window
        ''')
    #    n1=288        d1=1          o1=1
    #    n2=314        d2=1          o2=1
    #    n3=494        d3=1          o3=1
    #    n4=1          d4=1          o4=3
```

```
Result('xy',
    '''
    window n3=1 |
    graph symbol=x title=Coordinates
    label1=X label2=Y plotcol=6
    min1=1.137825e6 max1=1.209575e6
    min2=1.080112e7 max2=1.087938e7
    ''')
# work on vp
Vpfiles=['Vp0.txt','Vp2030.txt','Vp2050.txt','Vp2070.txt']
for Vpfile in Vpfiles:
   strs = Vpfile.split(".")
   Vpname = strs[0]
   if Vpname=='Vp0':
      Vpname='Vp'
      yearname=''
   else:
      yearname=Vpname[2:]
   Vpdataname = 'Vpdata'+yearname
   print(Vpdataname)
   Flow(Vpdataname,Vpfile,
       '''
       echo n1=7 n2=37981440 data_format=ascii_float
       in=$SOURCE
       key1=i key2=j key3=k key4=x key5=y key6=z key7=Vp
```

```
            | dd form=native
            ''', stdin=0)

        Flow(Vpname,[Vpdataname, 'ijk'],
            '''
            window n1=1 f1=6 squeeze=n |
            intbin3 head=${SOURCES[1]} xkey=0 ykey=1 zkey=2 |
            window |
            transp plane=12 | transp plane=13 |
            put   d2=0.25   d3=0.25   label1=Z   label2=X   label3=Y   o2=1137.825
o3=10801.125
            ''')  # | put d1=-5 o1=-4295.10
        Result(Vpname,
            '''
            byte gainpanel=all mean=y bar=bar.rsf |
            grey3   color=virdis   frame3=80   frame2=100   frame1=353    point1=0.5
point2=0.70
            flat=n title="Vp (ft/s)" scalebar=y  unit1=ft  unit2="x1000 ft" unit3="x1000
ft"
            ''')

    Flow('Vp2030_suf1','Vp2030','window n1=1 f1=353 squeeze=y')


    # work on vs
```

```
Vsfiles=['Vs0.txt','Vs2030.txt','Vs2050.txt','Vs2070.txt']
for Vsfile in Vsfiles:
    strs = Vsfile.split(".")
    Vsname = strs[0]
    if Vsname=='Vs0':
        Vsname='Vs'
        yearname=''
    else:
        yearname=Vsname[2:]
    Vsdataname = 'Vsdata'+yearname
    print(Vsdataname)
    Flow(Vsdataname,Vsfile,
         '''
         echo n1=7 n2=37981440 data_format=ascii_float
         in=$SOURCE
         key1=i key2=j key3=k key4=x key5=y key6=z key7=Vs
         | dd form=native
         ''', stdin=0)

    Flow(Vsname,[Vsdataname, 'ijk'],
         '''
         window n1=1 f1=6 squeeze=n |
         intbin3 head=${SOURCES[1]} xkey=0 ykey=1 zkey=2 |
         window |
         transp plane=12 | transp plane=13 |
            put    d2=0.25    d3=0.25    label1=Z    label2=X    label3=Y    o2=1137.825
o3=10801.125
            ''')  # | put d1=-5 o1=-4295.10
        Result(Vsname,
            '''
            byte gainpanel=all mean=y bar=bar.rsf |
            grey3    color=virdis    frame3=80    frame2=100    frame1=353     point1=0.5
point2=0.70
            flat=n title="Vs (ft/s)" scalebar=y  unit1=ft  unit2="x1000 ft" unit3="x1000
ft"
            ''')

    # work on Rb
    Rbfiles=['Rb0.txt','Rb2030.txt','Rb2050.txt','Rb2070.txt']
    for Rbfile in Rbfiles:
        strs = Rbfile.split(".")
        Rbname = strs[0]
        if Rbname=='Rb0':
            Rbname='Rb'
            yearname=''
        else:
            yearname=Rbname[2:]
        Rbdataname = 'Rbdata'+yearname
        print(Rbdataname)
        Flow(Rbdataname,Rbfile,
            '''
            echo n1=7 n2=37981440 data_format=ascii_float
            in=$SOURCE
            key1=i key2=j key3=k key4=x key5=y key6=z key7=Rb
            | dd form=native
            ''', stdin=0)

        Flow(Rbname,[Rbdataname, 'ijk'],
            '''
            window n1=1 f1=6 squeeze=n |
            intbin3 head=${SOURCES[1]} xkey=0 ykey=1 zkey=2 |
            window |
            transp plane=12 | transp plane=13 |
            put d2=0.25 d3=0.25 label1=Z label2=X label3=Y o2=1137.825
o3=10801.125
            ''')  # | put d1=-5 o1=-4295.10
        Result(Rbname,
            '''
            byte gainpanel=all mean=y bar=bar.rsf |
            grey3 color=virdis frame3=80 frame2=100 frame1=353  point1=0.5
point2=0.70
            flat=n title="Rhob (g/cm3)" scalebar=y  unit1=ft  unit2="x1000 ft"
unit3="x1000 ft"
            ''')
```

```
Flow('Imp','Vp Rb',
    '''
    put label1=Z unit1=ft label2=x unit2=ft label3=y unit3=ft
    mul ${SOURCES[1]}
    ''')

Result('Imp',
      '''
      byte gainpanel=all mean=y bar=bar.rsf |
      grey3 flat=n frame1=0 frame2=150 frame3=200 color=seismic
      point1=0.3 point2=0.7 title="Acoustic Impedance" scalebar=y
      ''')

# convert from depth to time
Flow('Impt','Imp Vp',
    '''
    depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
    smooth rect2=5 rect3=5
    ''')   # dt=0.001 nt=401

Result('Impt',
      '''
      byte gainpanel=all mean=y bar=bar.rsf |
      grey3 flat=n frame3=250 frame2=100 frame1=0 color=viridis scalebar=y
```

```
        point1=0.3 point2=0.7 title="Acoustic Impedance" label1=Time unit1=s
        ''')
    # convolution modeling
    Flow('Seist','Impt','ai2refl  ricker1 frequency=28')

    Result('Seist',
        '''
        byte gainpanel=all  bar=bar.rsf |
        grey3 flat=n frame3=250 frame2=100 frame1=0
        point1=0.3  point2=0.7  title="Seismic  Image  in  Time"  label1=Time  unit1=s
scalebar=y color=seismic
        ''')
    # convert from time to depth
    #Flow('Seis','Seist  Vp','time2depth  velocity=${SOURCES[1]}  |  put  d1=-5  o1=-
4295.10 ')
    Flow('Seis','Seist  Vp','time2depth  velocity=${SOURCES[1]}  |  put  d1=-5  o1=-
4295.10')
    Flow('Seis.bin', 'Seis', 'rsf2bin bfile=$TARGET')

    Result('Seis',
        '''
        byte gainpanel=all  bar=bar.rsf |
        grey3 flat=n frame3=250 frame2=100 frame1=353 scalebar=y color=seismic
        point1=0.5 point2=0.7 title="Seismic Image in Depth" label1=Depth unit1=ft
minval=-0.3 maxval=0.3
```

```
     ''')
#   Seis.rsf
#   n1=420        d1=-5          o1=-4295.1    label1="Z" unit1="ft"
#   n2=288        d2=0.25        o2=1137.82    label2="x" unit2="ft"
#   n3=314        d3=0.25        o3=10801.1    label3="y" unit3="ft"
#   n4=1          d4=1           o4=6

######################### 2030

Flow('Sgdata2030','Sg2030.txt',
    '''
    echo n1=7 n2=37981440 data_format=ascii_float
    in=$SOURCE
    key1=i key2=j key3=k key4=x key5=y key6=z key7=Sg
    | dd form=native
    ''', stdin=0)


Flow('Sg2030','Sgdata2030 ijk',
    '''
    window n1=1 f1=6 squeeze=n |
    intbin3 head=${SOURCES[1]} xkey=0 ykey=1 zkey=2 |
    window |
    transp plane=12 | transp plane=13 |
        put d2=0.25 d3=0.25 label1=Z label2=X label3=Y o2=1137.825 o3=10801.125
d1=-5 o1=-4295.10
        ''')

    Plot('Sg2030',
         '''
         byte gainpanel=all mean=y bar=bar.rsf |
         grey3   color=virdis   frame3=80   frame2=150   frame1=353     point1=0.5
point2=0.70
         flat=n title="Sg2030" scalebar=y  unit1=ft  unit2="x1000 ft" unit3="x1000 ft"
         ''')



    Flow('Imp2030','Vp2030 Rb2030',
        '''
        put label1=Z unit1=ft label2=x unit2=ft label3=y unit3=ft
        mul ${SOURCES[1]}
        ''')

    Result('Imp2030',
         '''
         byte gainpanel=all mean=y bar=bar.rsf |
         grey3 flat=n frame1=0 frame2=150 frame3=200 color=seismic
         point1=0.3 point2=0.7 title="Acoustic Impedance" scalebar=y
```

```
      ''')

# convert from depth to time
Flow('Impt2030','Imp2030 Vp2030',
    '''
    depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
    smooth rect2=5 rect3=5
    ''')

Result('Impt2030',
      '''
      byte gainpanel=all mean=y bar=bar.rsf |
      grey3 flat=n frame3=250 frame2=100 frame1=0 color=viridis
      point1=0.3 point2=0.7 title="Acoustic Impedance" label1=Time unit1=s
      ''')
# convolution modeling
Flow('Seist2030','Impt2030','ai2refl  ricker1 frequency=28')

Result('Seist2030',
      '''
      byte gainpanel=all  bar=bar.rsf |
      grey3 flat=n frame3=250 frame2=100 frame1=0
      point1=0.3 point2=0.7 title=Seismic Image in Time label1=Time unit1=s
      ''')
# convert from time to depth
```

```
        Flow('Seis2030','Seist2030 Vp2030','time2depth velocity=${SOURCES[1]} | put
d1=-5 o1=-4295.10')
        Flow('Seis2030.bin', 'Seis2030', 'rsf2bin bfile=$TARGET')
        Plot('Seis2030',
             '''
             byte gainpanel=all  bar=bar.rsf |
             grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
             point1=0.5 point2=0.7 title="Seismic Image in 2030" label1=Depth unit1=ft
minval=-0.3 maxval=0.3
             ''')
        Result('Seis2030','Seis2030','SideBySideIso')
        Flow('diffSeis30','Seis2030 Seis','math s2=${SOURCES[0]} s1=${SOURCES[1]}
output="s2-s1"')
        Plot('diffSeis30',
             '''
             byte gainpanel=all  bar=bar.rsf |
             grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
             point1=0.5 point2=0.7 title="Seismic Difference in 2030" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
             ''')
        ########################## 2070
        Flow('Sgdata2070','Sg2070.txt',
            '''
            echo n1=7 n2=37981440 data_format=ascii_float
            in=$SOURCE
            key1=i key2=j key3=k key4=x key5=y key6=z key7=Sg
            | dd form=native
            ''', stdin=0)



        Flow('Sg2070','Sgdata2070 ijk',
            '''
            window n1=1 f1=6 squeeze=n |
            intbin3 head=${SOURCES[1]} xkey=0 ykey=1 zkey=2 |
            window |
            transp plane=12 | transp plane=13 |
            put d2=0.25 d3=0.25 label1=Z label2=X label3=Y o2=1137.825 o3=10801.125
d1=-5 o1=-4295.10
            ''')

        Plot('Sg2070',
             '''
             byte gainpanel=all mean=y bar=bar.rsf |
             grey3   color=virdis   frame3=80   frame2=150   frame1=353      point1=0.5
point2=0.70
             flat=n title="Sg2070" scalebar=y  unit1=ft  unit2="x1000 ft" unit3="x1000 ft"
             ''')
        Result('Sg2070','Sg2070','SideBySideIso')

        Flow('Imp2070','Vp2070 Rb2070',
            '''
            put label1=Z unit1=ft label2=x unit2=ft label3=y unit3=ft
            mul ${SOURCES[1]}
            ''')

        Result('Imp2070',
             '''
             byte gainpanel=all mean=y bar=bar.rsf |
             grey3 flat=n frame1=0 frame2=150 frame3=200 color=seismic
             point1=0.3 point2=0.7 title="Acoustic Impedance" scalebar=y
             ''')

        # convert from depth to time
        Flow('Impt2070','Imp2070 Vp2070',
            '''
            depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
            smooth rect2=5 rect3=5
            ''')

        Result('Impt2070',
             '''
             byte gainpanel=all mean=y bar=bar.rsf |
             grey3 flat=n frame3=250 frame2=100 frame1=0 color=viridis
             point1=0.3 point2=0.7 title="Acoustic Impedance" label1=Time unit1=s
             ''')
```

```
        # convolution modeling
        Flow('Seist2070','Impt2070','ai2refl  ricker1 frequency=28')

        Result('Seist2070',
              '''
              byte gainpanel=all  bar=bar.rsf |
              grey3 flat=n frame3=250 frame2=100 frame1=0
              point1=0.3 point2=0.7 title=Seismic Image in Time label1=Time unit1=s
              ''')
        # convert from time to depth
        Flow('Seis2070','Seist2070 Vp2070','time2depth velocity=${SOURCES[1]} | put
d1=-5 o1=-4295.10')
        Flow('Seis2070.bin', 'Seis2070', 'rsf2bin bfile=$TARGET')


        Plot('Seis2070',
              '''
              byte gainpanel=all  bar=bar.rsf |
              grey3 flat=n frame3=80 frame2=100 frame1=353 scalebar=y color=seismic
              point1=0.5 point2=0.7 title="Seismic Image in 2070" label1=Depth unit1=ft
minval=-0.3 maxval=0.3
              ''')
        Result('Seis2070','Seis2070','SideBySideIso')
        Flow('diffSeis70','Seis2070 Seis','math s2=${SOURCES[0]} s1=${SOURCES[1]}
output="s2-s1"')
```

```
        Plot('diffSeis70',
               '''
               byte gainpanel=all  bar=bar.rsf |
               grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
               point1=0.5  point2=0.7  title="Seismic  Difference  in  2070"  label1=Depth
unit1=ft minval=-0.3 maxval=0.3
               ''')

        Result('SeisDiff3070','Seis2030 Seis2070 diffSeis30 diffSeis70','TwoRows')
        Result('Sg3070','Sg2030 Sg2070','TwoColumns')

        Result('SeisDiff','diffSeis30 diffSeis70','SideBySideIso')
        ################################################## AVO
        Flow('Vpt','Vp',
             '''
             depth2time velocity=${SOURCES[0]} dt=0.0005 nt=201  |
             smooth rect2=5 rect3=5
             ''')

        Flow('Vst','Vs Vp',
             '''
             depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
             smooth rect2=5 rect3=5
             ''')
```

```
Flow('Rhobt','Rb Vp',
    '''
    depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
    smooth rect2=5 rect3=5
    ''')
Flow('AVOt','Vpt         Vst         Rhobt','zoeppritz2         vs=${SOURCES[1]}
rho=${SOURCES[2]} a0=10 da=15 na=5 | sftransp | sfricker1 frequency=50 | sftransp' )  #
angel: 10, 25, 40, 55, 70 deg

angels=[10,25,40,55,70]
for i in range(5):

   strAVOtangel='AVOt_d'+str(angels[i])
   print(strAVOtangel)
   Flow(strAVOtangel, 'AVOt','window n1=1 f1=%d' %i)
   strAVOangel='AVO_d'+str(angels[i])
   Flow(strAVOangel,[strAVOtangel,                                'Vp'],'time2depth
velocity=${SOURCES[1]} ' )
   Flow(strAVOangel+'.bin', strAVOangel, 'rsf2bin bfile=$TARGET')

#Flow('AVOt1','AVOt','window n1=1 f1=0')    # n1=401
Flow('AVO1','AVOt1 Vp','time2depth velocity=${SOURCES[1]} ' )
Plot('AVO1',
     '''
```

```
          byte gainpanel=all  bar=bar.rsf |
          grey3 flat=n frame3=80 frame2=100 frame1=353 scalebar=y color=seismic
          point1=0.5 point2=0.7 title="AVO angle=10" label1=Depth unit1=ft
          ''')

     #Flow('AVOt2','AVOt','window n1=1 f1=4')
     Flow('AVO2','AVOt2 Vp','time2depth velocity=${SOURCES[1]} ' )
     Plot('AVO2',
          '''
          byte gainpanel=all  bar=bar.rsf |
          grey3 flat=n frame3=80 frame2=100 frame1=353 scalebar=y color=seismic
          point1=0.5 point2=0.7 title="AVO angle=70" label1=Depth unit1=ft
          ''')
     Result('AVO12','AVO1 AVO2','TwoColumns')



     #sfzoeppritz2 < Vp.rsf  vs=Vs.rsf  rho=Rhob.rsf  a0=10  da=5  na=5  |  sftransp  |
sfricker1 frequency=10 | sftransp > avo.rsf
     #< avo.rsf sfwindow n4=1 f4=150 > avoY1.rsf
     #< avoY1.rsf sftransp plane=12 | sftransp plane=23  | sfgrey | sfpen



     ################################################## AVO 2030
```

```
Flow('Vpt2030','Vp2030',
    '''
    depth2time velocity=${SOURCES[0]} dt=0.0005 nt=201  |
    smooth rect2=5 rect3=5
    ''')

Flow('Vst2030','Vs2030 Vp2030',
    '''
    depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
    smooth rect2=5 rect3=5
    ''')


Flow('Rhobt2030','Rb2030 Vp2030',
    '''
    depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
    smooth rect2=5 rect3=5
    ''')
Flow('AVOt2030','Vpt2030 Vst2030 Rhobt2030','zoeppritz2 vs=${SOURCES[1]}
rho=${SOURCES[2]} a0=10 da=15 na=5 | sftransp | sfricker1 frequency=50 | sftransp' ) #
angel: 10, 25, 40, 55, 70 deg

angels=[10,25,40,55,70]
```

```
for i in range(5):

    strAVOtangel='AVOt2030_d'+str(angels[i])
    print(strAVOtangel)
    Flow(strAVOtangel, 'AVOt2030','window n1=1 f1=%d' %i)
    strAVOangel='AVO2030_d'+str(angels[i])
    Flow(strAVOangel,[strAVOtangel,                    'Vp2030'],'time2depth
velocity=${SOURCES[1]} ' )
    Flow(strAVOangel+'.bin', strAVOangel, 'rsf2bin bfile=$TARGET')

#Flow('AVOt1','AVOt','window n1=1 f1=0')    # n1=401

Flow('diffAVO30_d10','AVO2030_d10   AVO_d10','math   s2=${SOURCES[0]}
s1=${SOURCES[1]}  output="s2-s1"')
Result('AVO2030_d10',
     '''
     byte gainpanel=all  bar=bar.rsf |
     grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
     point1=0.5 point2=0.7 title="AVO in 2030 at 10deg" label1=Depth unit1=ft
minval=-0.3 maxval=0.3
     ''')
Result('AVO2030_d40',
     '''
     byte gainpanel=all  bar=bar.rsf |
     grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
```

```
            point1=0.5 point2=0.7 title="AVO in 2030 at 40deg" label1=Depth unit1=ft
minval=-0.3 maxval=0.3
            ''')
        Result('diffAVO30_d10',
            '''
            byte gainpanel=all  bar=bar.rsf |
            grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
            point1=0.5 point2=0.7 title="AVO Difference in 2030 at 10deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
            ''')
        Flow('diffAVO30_d40','AVO2030_d40   AVO_d40','math   s2=${SOURCES[0]}
s1=${SOURCES[1]}  output="s2-s1"')
        Result('diffAVO30_d40',
            '''
            byte gainpanel=all  bar=bar.rsf |
            grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
            point1=0.5 point2=0.7 title="AVO Difference in 2030 at 40deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
            ''')
        Flow('diffAVO30_d55','AVO2030_d55   AVO_d55','math   s2=${SOURCES[0]}
s1=${SOURCES[1]}  output="s2-s1"')
        Result('diffAVO30_d55',
            '''
            byte gainpanel=all  bar=bar.rsf |
            grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
            point1=0.5 point2=0.7 title="AVO Difference in 2030 at 55deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
            ''')

    ################################################# AVO 2050


    Flow('Vpt2050','Vp2050',
        '''
        depth2time velocity=${SOURCES[0]} dt=0.0005 nt=201  |
        smooth rect2=5 rect3=5
        ''')

    Flow('Vst2050','Vs2050 Vp2050',
        '''
        depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
        smooth rect2=5 rect3=5
        ''')


    Flow('Rhobt2050','Rb2050 Vp2050',
        '''
        depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
        smooth rect2=5 rect3=5
        ''')
    Flow('AVOt2050','Vpt2050 Vst2050 Rhobt2050','zoeppritz2 vs=${SOURCES[1]}
rho=${SOURCES[2]} a0=10 da=15 na=5 | sftransp | sfricker1 frequency=50 | sftransp' )  #
angel: 10, 25, 40, 55, 70 deg

    angels=[10,25,40,55,70]
    for i in range(5):

       strAVOtangel='AVOt2050_d'+str(angels[i])
       print(strAVOtangel)
       Flow(strAVOtangel, 'AVOt2050','window n1=1 f1=%d' %i)
       strAVOangel='AVO2050_d'+str(angels[i])
       Flow(strAVOangel,[strAVOtangel,                              'Vp2050'],'time2depth
velocity=${SOURCES[1]} ' )
       Flow(strAVOangel+'.bin', strAVOangel, 'rsf2bin bfile=$TARGET')

    #Flow('AVOt1','AVOt','window n1=1 f1=0')    # n1=401
    Flow('diffAVO50_d10','AVO2050_d10   AVO_d10','math   s2=${SOURCES[0]}
s1=${SOURCES[1]}  output="s2-s1"')
    Plot('diffAVO50_d10',
            '''
            byte gainpanel=all  bar=bar.rsf |
            grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
            point1=0.5 point2=0.7 title="AVO Difference in 2050 at 10deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
            ''')
```

```
Flow('diffAVO50_d40','AVO2050_d40   AVO_d40','math   s2=${SOURCES[0]}
s1=${SOURCES[1]}  output="s2-s1"')
Plot('diffAVO50_d40',
     '''
     byte gainpanel=all  bar=bar.rsf |
     grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
     point1=0.5 point2=0.7 title="AVO Difference in 2050 at 40deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
     ''' )

################################################## AVO 2070


Flow('Vpt2070','Vp2070',
    '''
    depth2time velocity=${SOURCES[0]} dt=0.0005 nt=201  |
    smooth rect2=5 rect3=5
    ''')

Flow('Vst2070','Vs2070 Vp2070',
    '''
    depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
    smooth rect2=5 rect3=5
    ''')
```

```
        Flow('Rhobt2070','Rb2070 Vp2070',
            '''
            depth2time velocity=${SOURCES[1]} dt=0.0005 nt=201  |
            smooth rect2=5 rect3=5
            ''')
        Flow('AVOt2070','Vpt2070 Vst2070 Rhobt2070','zoeppritz2 vs=${SOURCES[1]}
rho=${SOURCES[2]} a0=10 da=15 na=5 | sftransp | sfricker1 frequency=50 | sftransp' )  #
angel: 10, 25, 40, 55, 70 deg

        angels=[10,25,40,55,70]
        for i in range(5):

           strAVOtangel='AVOt2070_d'+str(angels[i])
           print(strAVOtangel)
           Flow(strAVOtangel, 'AVOt2070','window n1=1 f1=%d' %i)
           strAVOangel='AVO2070_d'+str(angels[i])
           Flow(strAVOangel,[strAVOtangel,                              'Vp2070'],'time2depth
velocity=${SOURCES[1]} ' )
           Flow(strAVOangel+'.bin', strAVOangel, 'rsf2bin bfile=$TARGET')

        #Flow('AVOt1','AVOt','window n1=1 f1=0')    # n1=401
        Plot('AVO2070_d10',
             '''
             byte gainpanel=all  bar=bar.rsf |
             grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
             point1=0.5 point2=0.7 title="AVO in 2070 at 10deg" label1=Depth unit1=ft
minval=-0.3 maxval=0.3
             ''')
        Plot('AVO2070_d25',
             '''
             byte gainpanel=all  bar=bar.rsf |
             grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
             point1=0.5 point2=0.7 title="AVO in 2070 at 25deg" label1=Depth unit1=ft
minval=-0.3 maxval=0.3
             ''')
        Plot('AVO2070_d55',
             '''
             byte gainpanel=all  bar=bar.rsf |
             grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
             point1=0.5 point2=0.7 title="AVO in 2070 at 55deg" label1=Depth unit1=ft
minval=-0.3 maxval=0.3
             ''')
        Result('AVO2070_d102555','AVO2070_d10                    AVO2070_d25
AVO2070_d55','SideBySideIso')

        Flow('diffAVO70_d10','AVO2070_d10   AVO_d10','math   s2=${SOURCES[0]}
s1=${SOURCES[1]}  output="s2-s1"')
        Plot('diffAVO70_d10',
             '''
```

```
            byte gainpanel=all  bar=bar.rsf |
            grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
            point1=0.5 point2=0.7 title="AVO Difference in 2070 at 10deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
            ''')

        Flow('diffAVO70_d40','AVO2070_d40   AVO_d40','math   s2=${SOURCES[0]}
s1=${SOURCES[1]}  output="s2-s1"')
        Plot('diffAVO70_d40',
            '''
            byte gainpanel=all  bar=bar.rsf |
            grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
            point1=0.5 point2=0.7 title="AVO Difference in 2070 at 40deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
            ''')

        Flow('diffAVO70_d55','AVO2070_d55   AVO_d55','math   s2=${SOURCES[0]}
s1=${SOURCES[1]}  output="s2-s1"')
        Plot('diffAVO70_d55',
            '''
            byte gainpanel=all  bar=bar.rsf |
            grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
            point1=0.5 point2=0.7 title="AVO Difference in 2070 at 55deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
            ''')
```

```
    Flow('diffAVO70_d55d10','AVO2070_d55                    AVO2070_d10','math
s2=${SOURCES[0]} s1=${SOURCES[1]}  output="s2-s1"')
    Result('diffAVO70_d55d10',
        '''
        byte gainpanel=all  bar=bar.rsf |
        grey3 flat=n frame3=80 frame2=150 frame1=353 scalebar=y color=seismic
        point1=0.5 point2=0.7 title="AVO Difference in 2070 at 55deg" label1=Depth
unit1=ft minval=-0.3 maxval=0.3
        ''')

    Result('AVODiff3070','Seis2030 Seis2070 diffSeis30 diffSeis70','TwoRows')
    Result('AVODiff_d40','diffAVO30_d40 diffAVO70_d40','SideBySideIso')
    Result('AVODiff_d10','diffAVO30_d10 diffAVO70_d10','SideBySideIso')
    Result('SeisAvoDiff3070_d10','diffSeis30        diffSeis70        diffAVO30_d10
diffAVO70_d10','TwoRows')
    Result('SeisAvoDiff3070_d40','diffSeis30        diffSeis70        diffAVO30_d40
diffAVO70_d40','TwoRows')
    Result('SeisAvoDiff3070_d55','diffSeis30        diffSeis70        diffAVO30_d55
diffAVO70_d55','TwoRows')
    Result('AvoDiff3070_d104055','diffAVO30_d10                    diffAVO30_d40
diffAVO30_d55 diffAVO70_d10 diffAVO70_d40 diffAVO70_d55','TwoRows')
    Result('AvoDiff30_d104055','diffAVO30_d10                      diffAVO30_d40
diffAVO30_d55','SideBySideIso')
```

```
        Result('AvoDiff70_d104055','diffAVO70_d10                    diffAVO70_d40
diffAVO70_d55','SideBySideIso')

        End()
```