A NOVEL 'SMART MICROCHIP PROPPANTS' TECHNOLOGY FOR PRECISION DIAGNOSTICS OF HYDRAULIC FRACTURE NETWORKS

Final Report
(for the period October 1, 2019 – August 31, 2024)
Prepared for:

U.S. Department of Energy
National Energy Technology Laboratory
3610 Collins Ferry Road
Morgantown, WV 26505-2353, USA
Cooperative Agreement No. DE-FE0031784

Prepared by:

Amirmasoud Kalantari Dahaghi, University of Kansas (Lead institution)

Aydin Babakhani, University of California at Los Angeles (UCLA) (Subawardee)

John Lovell, MicroSilicon Inc. (Subawardee)

Larry Britt, NSF Fracturing (Contractor)

Sherilyn Williams-Stroud, Confractus Inc (Contractor)

"EOG Resources Inc: Cost share provider"

University of Kansas Center for Research, Inc.

2385 Irving Hill Road, Lawrence, KS 66045-7568

November 2024

DISCLAIMER

This research report was prepared by the above-referenced project team as an account of work sponsored by the U.S. Department of Energy National Energy Technology Laboratory (DOE NETL). To the best of the project teams' knowledge and belief, this report is true, complete, and accurate; however, because of the research nature of the work performed, neither the University of Kansas Center for Research, Inc. nor the other participating entities, nor any of their directors, officers, or employees makes any warranty, express or implied, or assumes any legal liability or responsibility for the use of any information, apparatus, product, method, process, or similar item disclosed or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement or recommendation by the above-referenced project team.

ACKNOWLEDGMENT

This material is based upon work supported by DOE NETL under Award Number No. DE-FE0031784

DOE DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. EXECUTIVE SUMMARY

This project introduces innovative technology to improve subsurface characterization, visualization, and diagnostics of unconventional reservoirs (fossil resources). Through a collaborative effort involving the University of Kansas, UCLA, MicroSilicon Inc., and EOG Resources, the project aims to deliver precision diagnostics for hydraulic fractures using novel high-resolution imaging technology based on smart microchip proppants. Additionally, it seeks to enhance the accuracy and predictability of integrated numerical, and machine-learning modeling techniques for hydraulic fracture characterization and simulation.

This groundbreaking technology addresses significant gaps in understanding unconventional and tight reservoir behavior and optimizing well-completion strategies, enabling more cost-efficient recovery of unconventional resources. Figure 1 illustrates the proposed technology, showcasing areas of innovation. It envisions a closed-loop fracture diagnostic and modeling architecture designed to improve fracture design and optimize well spacing.

The project comprises both computational and experimental components. The computational component involves real-time fracture mapping, hydraulic fracture diagnostics, and simulations powered by physics-informed machine learning. The experimental component includes:

- Detailed geomechanical rock characterization.
- Laboratory testing of smart microchip sensors using 3D-printed synthetic and real core samples.
- Development and lab testing of downhole tools to power the microchip proppants and receive their signals.
- Field testing of the technology.

A key innovation is the battery-less, wireless, and fine-sized sensor technology, which offers real-time, cost-efficient, high-resolution, and "direct" fracture mapping. By employing microchip sizes tailored to various proppants, this technology provides better calibration and interpretation of other indirect diagnostic tools currently used for fracture characterization. Once injected into the formation, the MicroChip proppants generate real-time data, enabling a more accurate evaluation of small-scale hydraulic fracturing performance.

This project supports the Department of Energy's (DOE) objectives to enhance the economic and energy security of the United States. By providing advanced technology, it aims to maximize the recovery efficacy of unconventional resources while minimizing environmental impacts through optimized well spacing and improved completion designs. Furthermore, it ensures the United States maintains its technological leadership in advanced energy technologies for efficient and environmentally responsible fossil fuel production.

During this project, the team successfully completed key tasks, conducted comprehensive laboratory testing and measurements, and achieved successful testing outcomes. These accomplishments include:

- Completion of rock characterization and geomechanical studies for the selected site.

- Successful laboratory testing of Smart MicroChip proppants for functionality and transport.
- Verification of MicroChips' functionality at a high temperature of 250°C (482°F).
- Verification of MicroChips' functionality under high pressure, achieving 3,490 psi (24.06 MPa) without epoxy protection embedded in cement under compressive forces, and up to 10,000 psi with epoxy-type protection.
- Successful testing of the downhole tool in a lab environment, demonstrating its ability to self-power the smaller MicroChips.
- Development of the i-Geo Sensing web-based platform for physics-informed machine learning to process Smart MicroChip data and other diagnostic tools for efficient fracture mapping and simulation.
- Successful field trial conducted in an EOG-operated well in New Mexico.

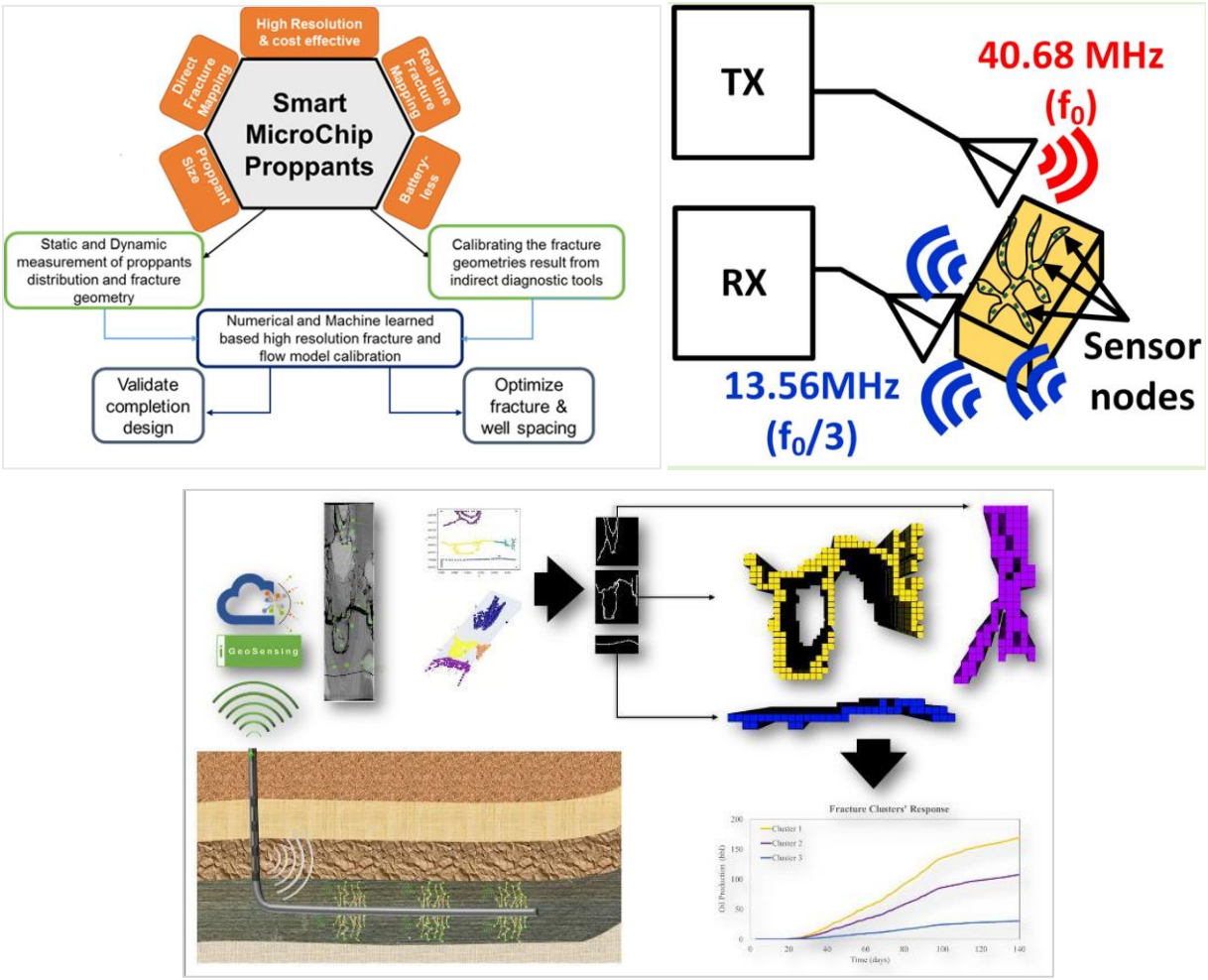


Figure 1a: Smart MicroChip Proppants technology: A closed-loop fracture diagnostic and modeling architecture

The following report is a summary of the work conducted and refers to a series of appendixes that contain more thorough and specific information about each section.

2. INTRODUCTION

Unconventional oil and gas reservoir development requires a detailed understanding of the geometry and complexity of the generated hydraulic fractures. The current categories of fracture diagnostic tools include a variety of methods for near-wellbore fracture diagnostics (e.g., production and temperature logs, tracers, and borehole imaging) as well as a variety of far-field techniques (e.g., microseismic fracture mapping). These techniques provide indirect and interpreted fracture geometry; therefore, none of these methods consistently provides a fully detailed and accurate description of the character of created hydraulic fractures.

2.1 Current State-of-the-Art Technologies

Other methods of research and development studies that are currently underway to better understand the complex interaction of the rock and fracture systems in unconventional plays may provide important additional information but still without the precision required to fully understand the near-well processes associated with a stimulation treatment. Fracture diagnostics using electromagnetic induction have an accuracy of 30-50 feet, which is still far from what is needed to understand the detailed complexity of the near-wellbore zone. An additional drawback to electromagnetic induction methods is that large amounts of conductive proppants are needed, which significantly increases the completion cost. Another method that has been used is deep sub-surface resistivity imaging with or without conductive particles, but this method also suffers from limited resolution (~50 ft). Microseismic (MS) emitters may provide additional location information but are very expensive ($10-100 per proppant) and their interpretation is subjective. Microseismic mapping suffers from a lack of correlation with fracture conductivity and geometry.

Capturing the complexity within the near wellbore region where induced fractures originate and interact with the rock has proven to be one of the most elusive monitoring tasks. An important observation from most hydraulic fracturing treatments is that because of the mode of failure in the near-wellbore region, the microseismic signal may not be detected and as a result is not coincident with the perforations. The interaction of the rock around the borehole with induced and existing fractures requires a method of observation that is both high resolution (~1 ft) and cost-effective. Therefore, finding innovative ways to directly characterize the height, length, and orientation of hydraulic fractures will provide critical information to fill the gap between what can be determined with current observational methods and the point where failure initiates. Optimizing hydraulic fracture treatments is critically dependent on understanding the near-wellbore processes. The technology proposed in this work, which focuses on collecting information in this zone, is based on frequency-shifting electromagnetic smart proppants that solve multiple key issues:

i. Due to the frequency-shift nature of the proppants, the signal transmitted from the proppants can be easily separated from the electromagnetic reflections from the reservoir. Consequently, a very small amount of proppants (a few gallons) is needed to perform fracture mapping.
ii. This technology can achieve a resolution of better than 1ft, which is two orders of magnitude better than the conventional methods.
iii. The electromagnetic frequency-shifting proppants are based on silicon technology, which can be produced at almost no cost (a few cents per proppant) in large volumes.

iv. Precise near-wellbore fracture diagnostics using our proposed technology enables better calibration and understating of fracture treatment applications.

3. FIELD TESTING SITE SELECTION AND STATIC DATA COLLECTION

3.1 EOG Asset Screening

EOG Resources assets are screened to identify multiple locations for the field pilot testing of Smart MicroChip Proppants. One of the key design parameters of the Microchips is a suitable range of formation resistivity. Table 1 summarizes the list of different plays and reservoirs that EOG resources operate with their range of resistivity (Ohm-m). Based on our field screening, Woodford, Paddock, and Blineberry reservoirs in Permian were selected as the top candidates for the pilot test.

Table 1: Collected average resistivity values from multipole EOG reservoirs for pilot selection

Formation	Reservoir	Average Formation Resistivity (Ohm-m)
Permian	Bone Spring sands	4 to 30
Permian	Wolfcamp	< 49
Permian	Leonard	<35
Permian Basin in New Mexico	Paddock	430
Permian Basin in New Mexico	Blineberry	180
Permian Basin in New Mexico	Basal Abo	400
South Texas	Lower Eagle Ford	<15
South Texas	Upper Eagle Ford	<30
South Texas	Austin Chalk	60
Anadarko	Woodford	440
Anadarko	Meramec (Mississippian)	105
Rockies	Mowry	<30
Rockies	Niobrara	<40
Rockies	Bakken	< 40
Rockies	Turner	< 30
Rockies	Codell	<10

Based on our final field screening, Permian Basin, Yeso Field Reservoir in New Mexico is selected for the initial data collection and rock characterization and field trial. Accordingly, 6 ft of core and logs were obtained from the Boyd XState well that was used for geomechanical laboratory testing.

Figure 1b: Received core and log data and the CT scan of the full slab core.

3.2 GEOMECHANICAL EVALUATION, UN-PROPPED CRACK TEST, FLUID SENSITIVITY TEST, AND EMBEDMENT TEST OF THE PADDOCK FORMATION (BOYD STATE #15H EDDY COUNTY, NEW MEXICO)

Summary:

A laboratory investigation was conducted on the core from the Boyd State #15H in the Paddock Formation. Tests were conducted on mineralogy, grain density, and ultrasonic velocity to determine the geomechanical properties of the Paddock Formation. Triaxial compression tests to determine the static Young's Modulus, although planned, couldn't be performed due to the small sample size of the available core plugs.

In addition to the geomechanical testing an un-propped crack test, fluid sensitivity test, and embedment test were conducted to better understand and evaluate the fracturing characteristics and the brittleness/fracability of the Paddock Formation. In addition, these tests were used to assess the viability of various proppants (type, size, concentration) and fluids (treated water, linear gel, or cross-linked polymers) for their use in hydraulic fracturing.

Findings:

- The Paddock samples were primarily made up of dolomite constituents (an average of 82% dolomite).
- The Paddock samples tested had a range in dynamic Young's Modulus of 11.92 to 17.96 x 10^6 psi and an average of 14.51 x 10^6 psi.
- The dynamic Poisson's Ratio ranged from 0.18 to 0.33 and averaged 0.273 for the thirteen samples tested.
- Using the dynamic to static Young's Modulus correlation developed by
    - Britt Rock Mechanics Laboratory (SPE 125525)

- $E_{static} = 0.835 \times E_{dynamic} - 0.424$,
- The average static Young's Modulus was estimated to be 11.69 x 10^6 psi,

- Little shear anisotropy (< 5%) was identified in all but one of the samples tested and that sample was visibly fissured,
- Wet nitrogen could flow through an un-propped crack at 3000 psi confining pressure in the shale samples and retain extraordinary permeability.
- Little to no fluid sensitivity was noted in the Paddock Formation,
- Little proppant embedment (0.068 lbs/ft^2) was noted in the Paddock Formation.

Field Implications and Recommendations:

- The Paddock Formation has a high dynamic Young's Modulus which translates into a high static Young's Modulus.
- The Paddock Formation has little fluid sensitivity and proppant embedment making it a good fracturing application for a variety of materials (fluids and proppants).
- The Paddock Formation is a viable water frac candidate due to the ability to retain permeability and flow wet nitrogen through an unpropped fracture at confining pressure. All lab data indicates that the Paddock Formation is very brittle.

3.2.1 Introduction:

A laboratory investigation was conducted to evaluate the geomechanical rock properties of the Paddock Formation. Tri-axial compression testing couldn't be conducted as the core plug samples were too small to determine the static Young's Modulus. Ultrasonic velocity measurements were made and the dynamic Young's Modulus was estimated and evaluated with an established dynamic to static Young's Modulus correlation developed by Britt Rock Mechanics Laboratory (SPE 125525). In addition to the geomechanical testing, grain density, mineralogical, unpropped crack test, fluid sensitivity test, and an embedment test were performed to assess the viability of this formation for hydraulic fracturing. Table 2 summarizes the tests conducted on the core plugs from these Paddock Formation samples.

Table 2: Summary of the Core Tests Performed on The Paddock Formation

Well Name	ID	Depth, (feet)	FTIR	Grain Density	Ultrasonic Velocity	Crack Test	Fluid Sensitivity	Embedment Test
Boyd State # 15H	1	2507.50	X	X	X			
Boyd State # 15H	2	2507.75	X	X	X			

Boyd State # 15H	3	2508.25	X	X	X			
Boyd State # 15H	4	2508.50	X	X	X			
Boyd State # 15H	5	2508.75	X	X	X			
Boyd State # 15H	6	2509.50	X	X	X			
Boyd State # 15H	7	2509.75	X	X	X		X	X
Boyd State # 15H	8	2597.10	X	X	X			
Boyd State # 15H	9	2597.40	X	X	X			
Boyd State # 15H	10	2597.90	X	X	X			
Boyd State # 15H	11	2600.20	X	X	X	X		
Boyd State # 15H	12	2600.60	X	X	X			
Boyd State # 15H	13	2600.90	X	X	X			

As shown, the core plugs were utilized for mineralogy, rock mechanics, unpropped cracks, fluid sensitivity, and embedment tests. Thirteen ultrasonic velocity tests were conducted on the Paddock core samples along with an un-propped crack test, a fluid sensitivity test, and an embedment test.

Table 3: Summary of the Mineralogy of The Paddock Formation

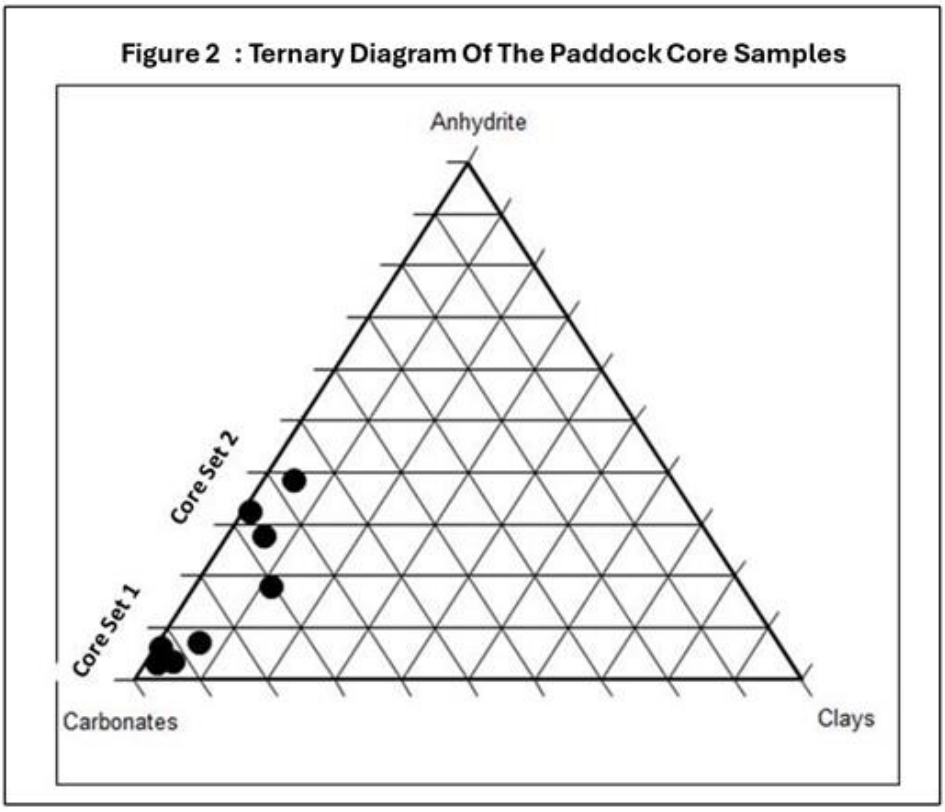
Well Name	*ID*	*Core Set*	*Depth, (feet)*	*Carbonate*	*Clay*	*Anhydrite*	*Feldspar*	*Quarts*
Boyd State #15H	***1***	***1***	***2507.50***	***94***	***0***	***2***	***4***	**0**
Boyd State #15H	***2***	***1***	***2507.75***	**92**	**1**	**6**	***1***	**0**
Boyd State #15H	***3***	***1***	***2508.25***	**95**	**0**	**4**	***1***	**0**
Boyd State #15H	***4***	***1***	***2508.50***	***94***	***2***	***3***	***1***	**0**
Boyd State #15H	***5***	***1***	***2508.75***	***94***	***0***	***3***	***3***	**0**
Boyd State #15H	***6***	***1***	***2509.50***	***92***	***0***	***6***	***2***	**0**
Boyd State #15H	***7***	***1***	***2509.75***	***95***	***0***	***5***	***0***	**0**
Boyd State #15H	***8***	***2***	***2597.10***	***68***	***12***	***17***	***2***	**2**
Boyd State #15H	***9***	***2***	***2597.40***	***66***	***1***	***32***	***0***	**0**
Boyd State #15H	***10***	***2***	***2597.90***	***90***	***4***	***3***	***2***	**0**
Boyd State #15H	***11***	***2***	***2600.20***	***66***	***6***	***27***	***0***	**0**
Boyd State #15H	***12***	***2***	***2600.60***	***85***	***6***	***7***	***2***	**0**
Boyd State #15H	***13***	***2***	***2600.90***	***56***	***5***	***38***	***1***	**0**

In addition, Fourier Transform Infrared Spectroscopy (FTIR) and grain density tests were conducted on all of the samples to better assess the carbonate and clay mineralogy (Appendix B).

3.2.2 Discussion:

First, the mineralogy of the samples was investigated. Fourier Transform Infrared Spectroscopy (FTIR) tests were conducted on all of the samples to assess the carbonate and clay mineralogy of the Paddock Formation. Analysis of the samples showed that the Paddock has a significant amount of dolomite with some anhydrite mineral constituents and some albeit little clay mineralogy. Table 2 summarizes the results of the core analysis of the Paddock Formation highlighting the mineral constituents of significance such as Carbonates (dolomite and calcite), Clays, Anhydrite, Feldspars, and Quartz.

Figure 2 : Ternary Diagram Of The Paddock Core Samples



As shown, the core samples from the core set one (2507.50 to 2509.75 ft) are predominately dolomite (> 89%) with small amounts of calcite (3-4%), anhydrite, feldspar, and clay. Samples from the core set two (2597.10 to 2600.90 ft) have significantly fewer dolomite constituents (56% to 90%) with no calcite and more anhydrite and clay. The sample from a depth of 2508.25 feet (Core Set 1-Sample ID 3), for example, had ninety-one percent dolomite, four percent anhydrite, and calcite, and only one percent feldspar and no clay constituents. Alternatively, the samples from 2600.20 feet (Core Set 2-Sample ID 11) were made up of sixty-six percent dolomite, twenty-seven percent anhydrite, and six percent clay, respectively.

From a mineralogical perspective, these core samples appear to have some differences given the variation in the primary mineral constituents of dolomite and anhydrite. Throughout the Permian and Delaware Basins, the abundant amounts of sulfate from evaporated seawater combined with the liberation of calcium during the dolomitization process resulted in significant amounts of anhydrite being deposited in the Clearfork and Paddock Formations. Figure 3 shows a Paddock sample from 2597.40 feet (Core Set 2 - Sample ID 9) which highlights an anhydrite nodule on the upper edge of the core sample.

With respect to the clay constituents, all of the core samples had few clay constituents with the samples having from 0 to 12 percent clays, respectively. These clays are primarily illite and smectite; however, given the core small amounts (average of only 2%) no extraordinary efforts should be utilized to minimize their effects in the completion and fracture stimulation process. Appendix B summarizes the quality control measures taken and details the mineralogy of each of the samples studied.

Figure 3: Anhydrite Nodule Within Dolomitic Paddock Formation
Core Set 2- Sample ID 9, Depth 2597.40 feet

Next, a series of ultrasonic measurements were made on each core plug. Each sample was subjected to a sonic frequency of 300 to 500 KHz in the lab and the compressional and shear velocities were measured. Table 4 summarizes the sonic (shear and compressional) travel times as a function of confining pressure for each sample.

Table 4: Summary of the Sonic Travel Times as a Function of the Confining Pressure										
ID	**Bulk r, gm/cc**	***1,000 psi Confinement***			***2,000 psi Confinement***			***3,000 psi Confinement***		
		P, ft/sec	**S1, ft/sec**	**S2, ft/sec**	**P, ft/sec**	**S1, ft/sec**	**S2, ft/sec**	**P, ft/sec**	**S1, ft/sec**	**S2, ft/sec**
1	**2.824**	**21201**	**12254**	**11512**	**22093**	**12648**	**11890**	**22513**	**12785**	**12070**
2	**2.842**	**21814**	**11772**	**12077**	**22730**	**11965**	**12316**	**22969**	**12270**	**12566**
3	**2.842**	**21430**	**11391**	**12248**	**22326**	**11578**	**12640**	**22441**	**11900**	**12781**
4	**2.803**	**21539**	**11368**	**12112**	**21850**	**11512**	**12325**	**21959**	**11693**	**12580**
5	**2.870**	**19642**	**10554**	**11443**	**20420**	**10919**	**11930**	**21486**	**11214**	**12014**
6	**2.848**	**22208**	**11998**	**12226**	**22818**	**12323**	**12603**	**23281**	**12477**	**12775**
7	**2.845**	**21352**	**11827**	**0**	**21919**	**12021**	**0**	**21988**	**12297**	**0**
8	**2.822**	**21133**	**13133**	**11808**	**21905**	**13409**	**11708**	**22170**	**13426**	**11910**
9	**2.855**	**21168**	**10833**	**11119**	**21995**	**11064**	**11301**	**22069**	**11098**	**11745**
10	**2.813**	**21535**	**12138**	**12216**	**21996**	**12540**	**12869**	**22114**	**12568**	**12923**
11	**2.830**	**21845**	**12225**	**12720**	**21969**	**12479**	**12898**	**22015**	**12650**	**12926**
12	**2.815**	**22050**	**12313**	**12512**	**22112**	**12578**	**12615**	**22305**	**12720**	**12949**
13	**2.851**	**22224**	**13273**	**13170**	**22552**	**14135**	**14284**	**23405**	**14966**	**14965**

It should be noted that two shear travel times (i.e., S1 and S2) are reported. The second shear travel time is measured perpendicular to the first. Such a comparison of shear travel times is a measure of shear anisotropy. Comparison of the shear velocities shows little evidence of significant anisotropy in these samples.

Paddock samples as the shear anisotropy recorded were less than 5% on seven of the samples, less than 10 percent on another five samples, and only one sample (Core Set-1, ID-7) showed significant anisotropy and that was a sample with a visible crack. The average shear anisotropy of the twelve unflawed samples was 4.83%.

Figure 4 shows the Paddock core sample from 2509.75 feet (Core Set 1-Sample ID 7) which highlights a fissure that extends nearly throughout the sample. Such a flaw dramatically affects the shear anisotropy.

Figure 4: Paddock Core Sample Showing A Fissure or. /Flaw In The Sample Core Set 1- Sample ID 7, Depth 2509.75 feet

Next, the shear and compressional velocities and the bulk density were then utilized to calculate dynamic rock properties for the core samples. The dynamic rock properties calculated from these laboratory measurements are summarized in Table 5. As shown, the Poisson's Ratio varies from 0.18 to 0.33 and the dynamic Young's Modulus varies from 11.92 to 17.96 x 10^6 psi at 2000 psi confining pressure.

Table 5: Summary of the Dynamic Rock Properties For The Paddock Formation

ID	*1,000 psi Confinement*			*2,000 psi Confinement*			*3,000 psi Confinement*		
	n	**Edynamic, MMpsi**		n	**E_dynamic, MMpsi**	**E_static* MMpsi**	n	**Edynamic, MMpsi**	
1	***0.25***	***14.20***	**-**	***0.26***	***15.21***	**12.3**	***0.26***	***15.61***	**-**
2	***0.29***	***13.66***	**-**	***0.31***	***14.27***	**11.5**	***0.30***	***14.91***	**-**
3	***0.30***	***12.88***	**-**	***0.32***	***13.44***	**10.8**	***0.30***	***14.07***	**-**
4	***0.31***	***12.69***	**-**	***0.31***	***13.02***	**10.4**	***0.30***	***13.37***	**-**
5	***0.30***	***11.11***	**-**	***0.30***	***11.92***	**9.5**	***0.31***	***12.70***	**-**
6	***0.29***	***14.22***	**-**	***0.29***	***15.00***	**12.1**	***0.30***	***15.43***	**-**

7	***0.28***	***13.64***	**-**	***0.28***	***14.16***	**11.4**	***0.27***	***14.67***	**-**
8	***0.19***	***15.46***	**-**	***0.20***	***16.32***	**13.2**	***0.21***	***16.50***	**-**
9	***0.32***	***11.88***		***0.33***	***12.46***	**10.0**	***0.33***	***12.54***	
10	***0.27***	***14.07***		***0.26***	***14.93***	**12.0**	***0.26***	***15.02***	
11	***0.27***	***14.42***		***0.26***	***14.90***	**12.0**	***0.25***	***15.21***	
12	***0.27***	***14.57***		***0.26***	***15.05***	**12.1**	***0.26***	***15.37***	
13	***0.22***	***16.46***		***0.18***	***17.96***	**14.6**	***0.15***	***19.75***	

Also shown for a 2000-psi confining pressure is the calculated static Young's Modulus derived from the Britt Rock Mechanics Laboratory Correlation first published in 2009. This correlation, shown as Equation 1, has a correlation:

$$E_{static} = 0.835 \times E_{dynamic} - 0.424 \quad (1)$$

coefficient, R^2, of 0.714 and includes hundreds of samples of sands, shales, and carbonates from around the world.

Figure 5 shows a plot comparing the Dynamic Young's Modulus from the ultrasonic tests to the Static Young's Modulus derived from the above correlation at 2000 psi confining pressure for all samples. Core Set 1 samples are in green and Core Set 2 samples are in red. As shown, the dynamic Young's Modulus and the correlated static Young's Modulus are quite high; however, they are consistent with other dolomites that have been tested in the Clearfork of the Permian Basin and the Potosi and Bonne Terre Formation dolomites of Southwestern Missouri. Appendix C details the testing procedures, ultrasonic velocity tests, and interpretation.

In addition to geomechanical testing, a series of tests were conducted to evaluate whether an unpropped hydraulic fracture could retain conductivity in the Paddock Formation. Residual fracture width has been observed both in the laboratory and in field experiments. Surface asperities or roughness at the fracture face may account for this residual width. A laboratory study by Schlumberger investigated this aspect of treated water fracture stimulations. Their results showed that an un-propped induced fracture under effective confining pressure conditions in excess of what is anticipated in the Paddock Formation could be expected to have some fracture conductivity. Their work further showed that fracture displacement and surface asperities were required to provide adequate fracture conductivity in the absence of proppants and suggested that high-strength proppants and higher, more conventional, concentrations of proppant were required to mitigate the need for the fracture displacement and surface asperities effect on fracture conductivity.

Figure 5: Static Young's Modulus From Dynamic To Static Correlation
Britt Rock Mechanics Laboratory, SPE 125525

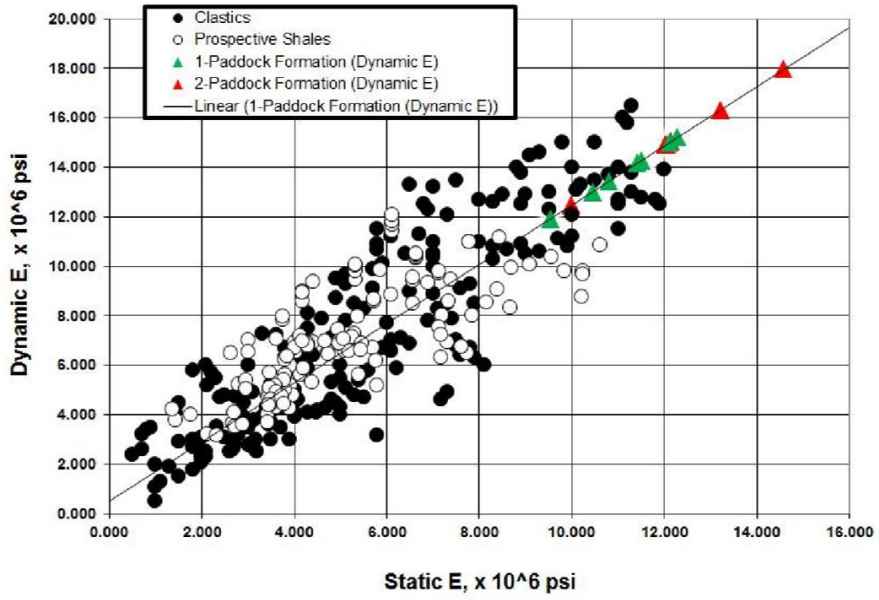


Figure 6: Britt Rock Mechanics Laboratory Test Apparatus

Figure 7: Test Cell Head with Piston and Inlet Ports

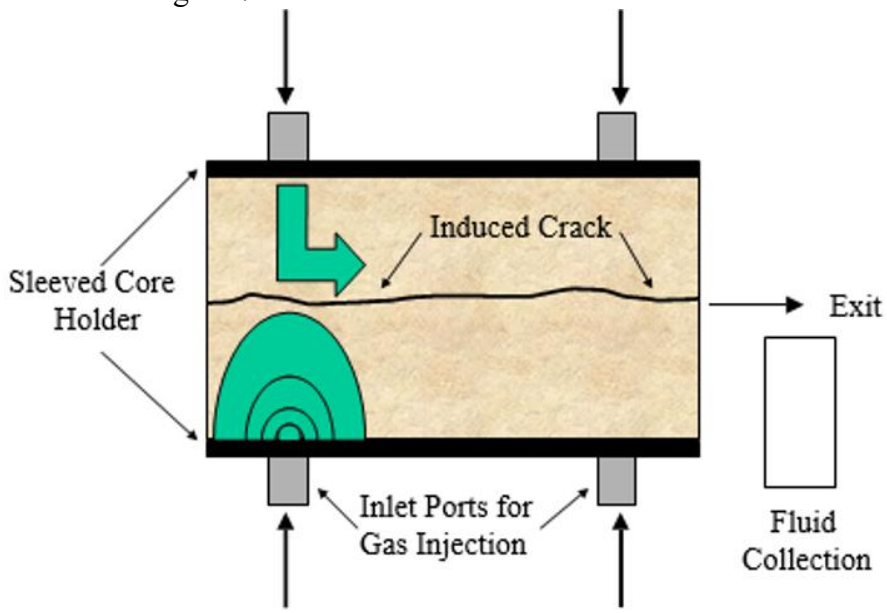


Figure 8: Test Cell Head with Piston and Inlet Ports

One un-propped crack test (Core Set 2, ID 11-2600.20 feet) from the Paddock Formation was conducted. To conduct this work, a core plug was cracked lengthwise with a masonry rock splitter to emulate a hydraulically created fracture. Sample ID 11 was chosen for this test because it had a nearly through-going natural fissure and split easily. Next, the cracked core plug was placed in Britt Rock Mechanics Laboratory "built for purpose" laboratory equipment, and confining pressure was applied. Figure 6 shows a picture of the equipment that was built and used in this study. This equipment consists of a test cell that can handle confining pressures from 0 to 10,000

psi and temperatures up to 300°F. A piston head is used to apply confining pressure and includes inlet electronic and flow ports. In the induced crack testing, the piston is actuated to apply confining pressure and emulate flow and shut-in conditions. While confined at pressure and temperature, flow is established through the inlet ports, and the permeability of the core and induced crack is measured. Figure 7 displays the head of the test cell. Shown in this figure is the piston head for applying confining pressure and inlet electronic and flow ports. During this study, effective confining pressures up to 1250 psi were applied to emulate the likely possible field conditions of the Formation.

Outlet flow ports exist both in the core direction perpendicular to the core as shown in Figure 8. In this study, we attempted to measure the retained permeability of the core in the direction of the induced crack as a function of confining pressure. A retained experiment was performed with the core plug prepared as described in the previous paragraphs. The test was designed to determine the retained permeability of the core while flowing wet nitrogen as the confining pressure was increased to 1250 psi and maintained for a couple of days. Results of the testing (Figure 9) show that the retained permeability measured at 100 psi (Step 1) was 62.1 md while the retained permeability as the confining pressure was increased 1.89-fold while being confined at 1250 psi for two days.

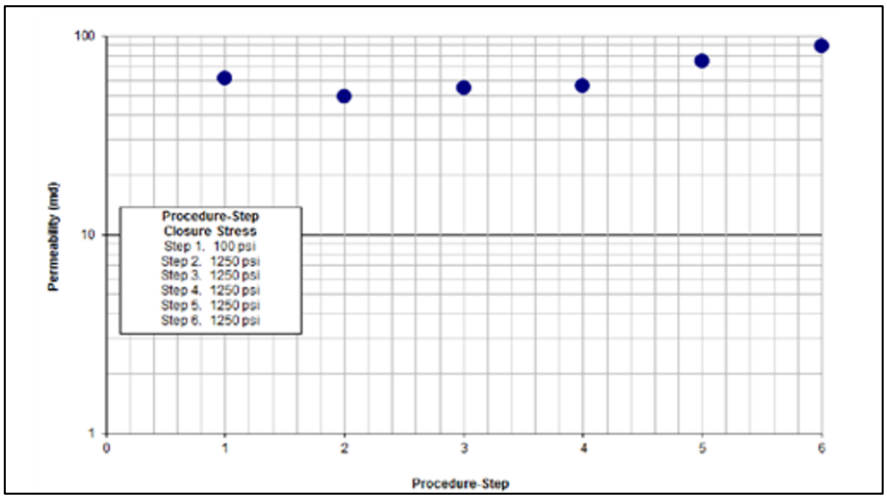


Figure 9: The Effect of Stress on Wet Nitrogen Injection (2156.23 meters)

Any formation can be a viable water frac target, but some lend themselves to the use of treated water as a fracturing fluid better than others. Low permeability naturally fissured formations, for example, make excellent water-frac candidates. This is because water-frac stimulations differ from more conventional gelled fracture stimulations as water is a cleaner fluid than conventional gels and does little damage to any natural fissures present and water is a poor fluid for transport of the proppant within the fracture. This latter issue (poor transport fluid) is the basis for our water-frac designs. The resulting fracture after a water-frac has two distinct components as shown in Figure 29. The first part of the fracture is the bottom of the fracture where all of the proppant settles during the treatment. The second part of the fracture is the upper part which has little or no proppant as it has all settled to the bottom of the fracture. Figure 10 shows a schematic of a water-frac where

forty percent of the fracture is filled with proppant and sixty percent of the fracture is unpropped. Reservoir simulations of such a fracture provide the basis of design for a water-frac treatment.

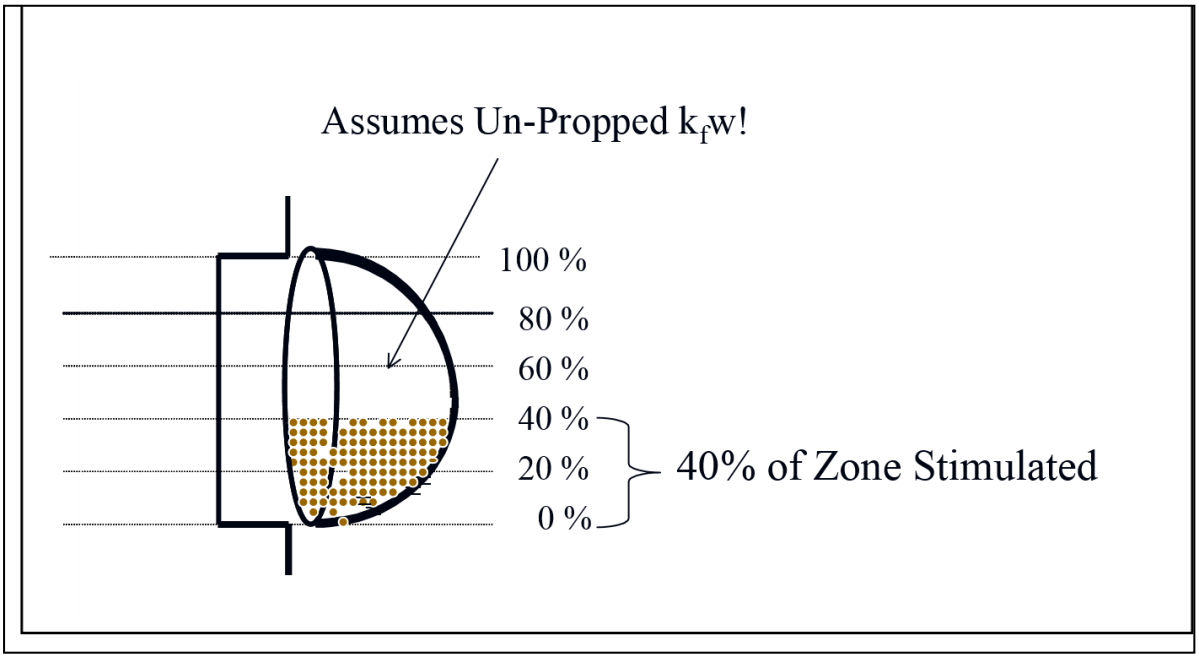


Figure 10: Schematic of a water-frac

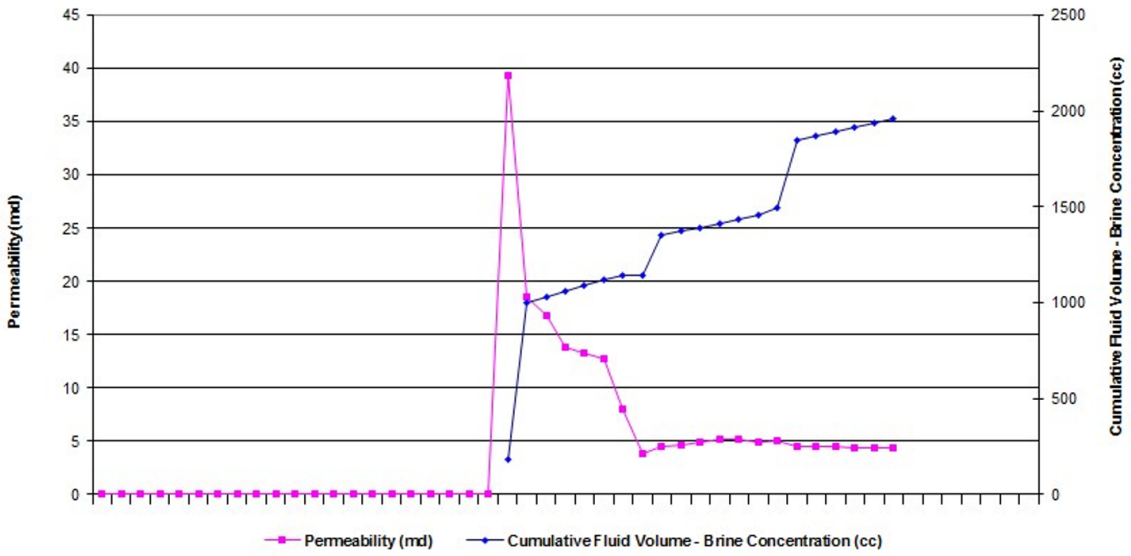
These simulations showed that a water-frac stimulation will perform as a fully propped fracture provided the ratio of the un-propped conductivity of the fracture to the product of the reservoir permeability and the height of the un-propped part of the fracture is in excess of 2. If this ratio is less than a value of 2 it indicates that the retained conductivity of the un-propped part of the fracture is too small for the reservoir permeability and un-propped fracture height. In this scenario, the only thing that can be done to improve the situation is to reduce the un-propped fracture height by pumping more proppant and filling up more of the fracture. Results of the crack testing indicate that the un-propped conductivity of the fracture in the Paddock Formation is significant and that the Paddock Formation can support tens of feet of un-propped fracture (assuming the reservoir permeability is less than 0.10 md for leak-off considerations).

In addition to the geomechanical testing, a fluid sensitivity test was conducted to evaluate whether fluid sensitivity or damage is important to the hydraulic fracturing of the Paddock Formation. The test included conducting a KCl sensitivity test utilizing BRML's built-for-purpose equipment. This laboratory investigation consisted of loading a test cell with a core sample and then saturating the core sample with 6% KCl, 4% KCl, and 2% KCl while measuring the permeability and the cumulative fluid injected. The confining pressure of the cell during the testing was maintained at 1250 psi and the retained permeability was measured until stabilized for each KCl concentration. The test consisted of using a fissured core sample from 2,509.75 feet injected and decreased to 7.3 md as the core was saturated with 6% KCl. The retained permeability for the 4% KCl and 2% KCl saturated core decreased to 5.1 md and 4.4 md, respectively. Such a small reduction in permeability is insignificant damage given the fluid dynamics of the test and the fact that the permeability was

being measured through a crack in the core. Appendix E details the testing procedures, raw data, and interpretive plots.

In addition to the geomechanical testing, a test was conducted to evaluate whether embedment is (Core Set 1, ID-7) to determine the potential for fluid damage in the Paddock Formation. Figure 11 shows a plot of the retained permeability as a function of KCl concentration for the core sample. Analysis of this figure shows that the initial permeability was 39.1 md as 6% KCl was important to the hydraulic fracturing of the Paddock Formation.

Figure 11: Fluid Sensitivity Testing w/ KCl (CS-1, ID-7, 2,509.75 feet)



The test included conducting an embedment test utilizing BRML's built-for-purpose equipment. Figure 12 shows a schematic of the embedment testing procedure. This laboratory investigation consisted of loading a test cell with the formation core cut in half and filled with a quarter inch of 40/70 mesh (d = 0.01462 inches) Carbo-Ceramic proppant. The confining pressure of the cell was then increased incrementally from 100 to 3,000 psi with the confining pressure stabilized at each pressure.

After each increase in confining pressure, the sample/cell dimensions were measured and documented. These dimensions were then evaluated to establish a relationship between embedment and proppant stress. The test consisted of using the sample from 2,509.75 meters (Core Set 1, ID 7) to determine the potential for proppant embedment in this shale formation.

Figure 12: Fines Migration and Embedment Testing Schematic

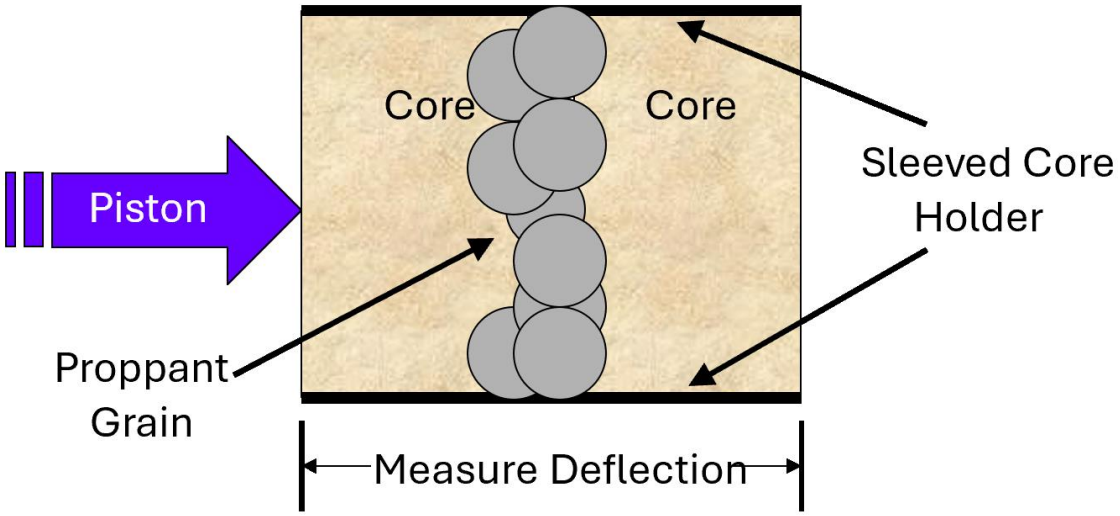


Figure 13: Embedment Testing (Core Set 1, ID 7, Depth 2,509.75 ft)

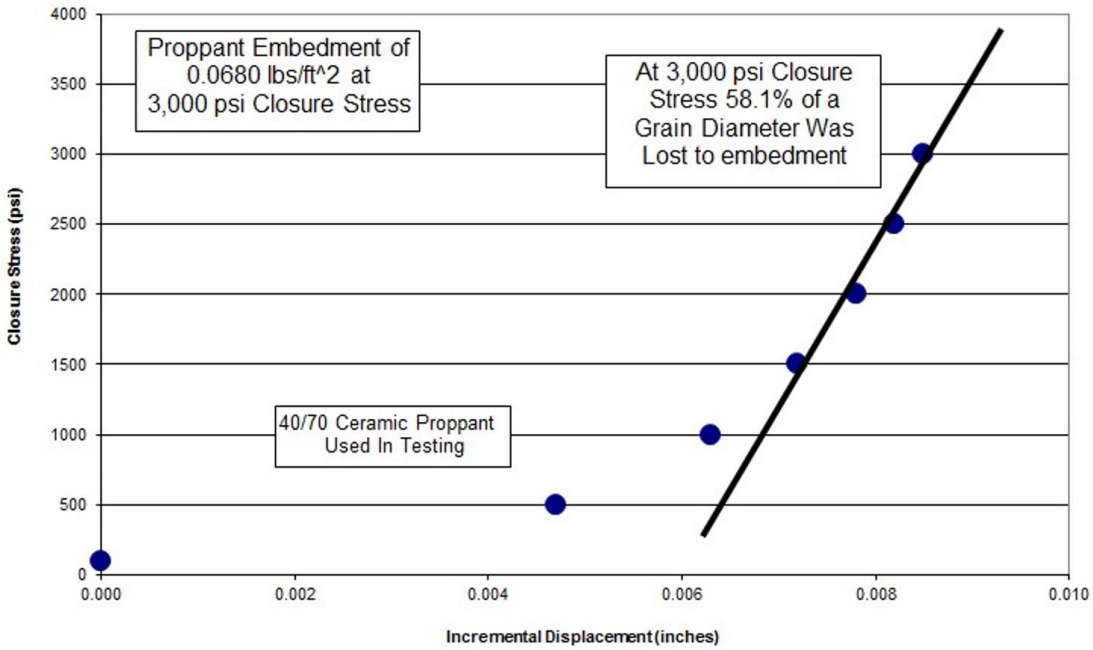


Figure 12 shows a plot of the embedment versus proppant stress data generated for the core sample and the 40/70 mesh Carbo-Lite proppant. As shown, the embedment increased nearly linearly with proppant stress from 1500 to 3000 psi at which point the test ended. Based on this analysis, an estimate of the proppant embedment, 0.068 lbs/ft² (58.1% of a grain diameter) at 3000 psi was determined. Further analysis of the data from Figure 32 shows a best-fit line through the data resulting in a correlation coefficient, R^2, of 0.9757. The correlation is shown in Equation 2:

$$y = 10^6 x - 6741.4 \quad (2)$$

To determine the maximum stress on the proppant, utilize Equation 3.

$$\sigma_{\text{proppant}}, \text{psi} = \frac{n(\text{POB} - \text{Pres}) + \text{Pres} + T - \text{Pf}}{1 - n} \quad (3)$$

As shown, the stress the proppant sees depends on the overburden stress, P_{OB}, the reservoir pressure, P_{res}, the bottomhole pressure in the fracture, P_f, the ability of the vertical stress to be transmitted in the horizontal direction (related to Poisson's Ratio, n), and the tectonic stress, T. A few observations can be made by reviewing this equation.

First, as the reservoir pressure is depleted, the stress on the proppant decreases. Secondly, as the well is drawn down, the stress on the proppant will increase. Finally, since the pressure in the fracture increases away from the wellbore (i.e. assumes finite conductivity fracture), for conventional fracturing the maximum stress on the proppant will be seen early in the well life at the wellbore.

In fact, it can be shown that the maximum stress on the proppant will occur at hydrocarbon breakthrough during the well cleanup unless the flowing bottom-hole pressure is rigorously controlled.

However, proppant embedment is not an issue in the Paddock Formation as this dolomite only has a total embedment at 3,000 psi which is only one-third that seen for the average formation (approximately 0.200 lbs/ft^2). Obviously, this formation is brittle enough to support hydraulic fracturing with treated water.

4. SMART MICROCHIP PROPPANT: DESIGN, DEVELOPMENT, LAB TESTING, AND VERIFICATION FOR THE FIELD TRIAL

4.1 Summary

Fracture mapping is essential in hydraulic fracturing and finds applications in the oil and gas industry. Mapping of fractures using traditional radioactive or micro-seismic methods involves contamination and is prone to being inaccurate, thereby reducing the yield in fracturing applications. Therefore, wireless sensor networks (WSNs) with each sensing node having a small form factor and low power consumption are currently being investigated for use in such applications. This article presents a fully battery-less system of coherent sensing nodes using wireless energy harvesting. These nodes are capable of mapping fractures reliably at temperatures up to 250°C and pressures up to 24 MPa.

Each node comprises a microchip having dimensions of 1.1 mm × 0.56 mm, two coils of 8 mm diameter each, and resonating capacitors. The microchip was fabricated in the Taiwan Semiconductor Manufacturing Company (TSMC) 0.18 μm process. The node receives a 40.68 MHz radio frequency (RF) signal in the industrial, scientific, and medical applications (ISM) band and transmits back a locked subharmonic 13.56 MHz ISM band RF signal.

The subharmonic signal is generated on-chip using a digital divide-by-3 circuit, drastically reducing the microchip power consumption compared with injection-locked oscillators or phase-locked loops (PLLs). The sensor nodes used in the system have a form factor of 17 mm × 12 mm × 0.2 mm and a minimum average power consumption of 1.5 μW.

Index Terms – Battery-less, coherent, complementary metal–oxide–semiconductor (CMOS), energy harvesting, fracture mapping, high temperature, hydraulic fracturing, sensing nodes, wireless, wireless sensor network (WSN).

4.2 Introduction

IRELESS sensor networks (WSNs) have become increasingly popular in recent years for use in various applications [1], [2], [3]. These networks include many spatially scattered sensor nodes, each of which senses some information from its environment, processes it, and transmits the processed data back to another node or a base station. WSNs can be battery-powered or wirelessly powered. Since wirelessly powered WSNs harvest energy from the environment, they do not have to encounter issues regarding the replacement or recharging of batteries.

Therefore, they provide significant advantages for sensing in regions that are remote or inaccessible. In addition, WSNs have the advantage of scalability, which enables the convenient addition or removal of nodes to/from the network. Since WSNs are mostly used in applications where powering them is inconvenient, the major constraint affecting the design of these nodes is low power consumption.

Among the many applications where WSNs can be used, hydraulic fracturing is essential for its widespread use in the energy industry, especially the oil and gas industry. Hydraulic fracturing

has been used to increase the flow rate of oil and natural gas in oilfields [4]. Extensive knowledge about the location and orientation of the fractures is required to increase production efficiency inside these oilfields [5]. Traditionally, fractures have been mapped using techniques, such as radionuclide monitoring, where radioactive tracers are injected into the fractures along with the propellant fluid [6]. However, these techniques are prone to high contamination risks if parameters, such as the amount, toxicity, and half-life of the tracers, are not strictly controlled.

More advanced applications use microseismic mapping, which includes monitoring the seismic activities along the fractures and is similar to seismology [7], [8], [9]. However, there is not enough understanding of the seismological processes inside the fractures. Moreover, they suffer from coherent noise within the band of seismic recordings. These shortcomings prompt us to look into wirelessly powered WSNs having a small form factor and low power consumption as a cleaner, newer, and more power-efficient technique for fracture mapping.

Fig. 14 shows a conceptual representation of fracture mapping using WSNs. Advancements in semiconductor technology have made it possible to reduce the size of microchips rapidly, therefore reducing the size of these sensing nodes. Currently, the size of these nodes is dominated by the size of the passive components, which include the off-chip resistors, capacitors, and coils or antennas. In applications, such as fracture mapping in oil and gas fields, the signals received and transmitted by the node should be able to penetrate the intervening medium between the transmitter (TX) and the node.

The requirements for penetration depth restrict the frequencies of signals that can be used below hundreds of megahertz. At these frequencies, reducing the form factor to a few millimeters entails the usage of an electrically small antenna or inductive coupling between coils in which the coil on the node is much smaller than the TX or receiver (RX) coil. These constraints reduce the efficiency of wireless powering links and, therefore, the harvested power for the microchip [10], [11], ultimately leading to a constraint on the power consumption of the microchip while still using it for extremely complex sensing and wireless communication applications.

Previous systems for fracture mapping have primarily focused on using nanoparticles and their responses to electromagnetic or acoustic waves inside fractures. Aderibigbe et al. [12], [13] use paramagnetic nanoparticles for the detection of fractures using susceptibility measurements at different locations. Sun [14] used a nanofluid and the convection and diffusion processes that affect it, while Liu et al. [15] measured the magnetic anomaly responses due to the injection of magnetic proppant.

However, for small fractures, the changes in magnetic susceptibility and fields are weak in the presence and absence of fractures, reducing the accuracy of fracture mapping. To the best of our knowledge, Al-Shehri et al. [16], [17] are the only works in the literature to date that use WSNs for fracture mapping through a technology called FracBots (Fracture Robots). These FracBots use off-the-shelf components and near-field communication and are placed inside fractures to receive and transmit signals. However, these nodes have a large size, large coils for the RX and TX, and use milliwatt-level power.

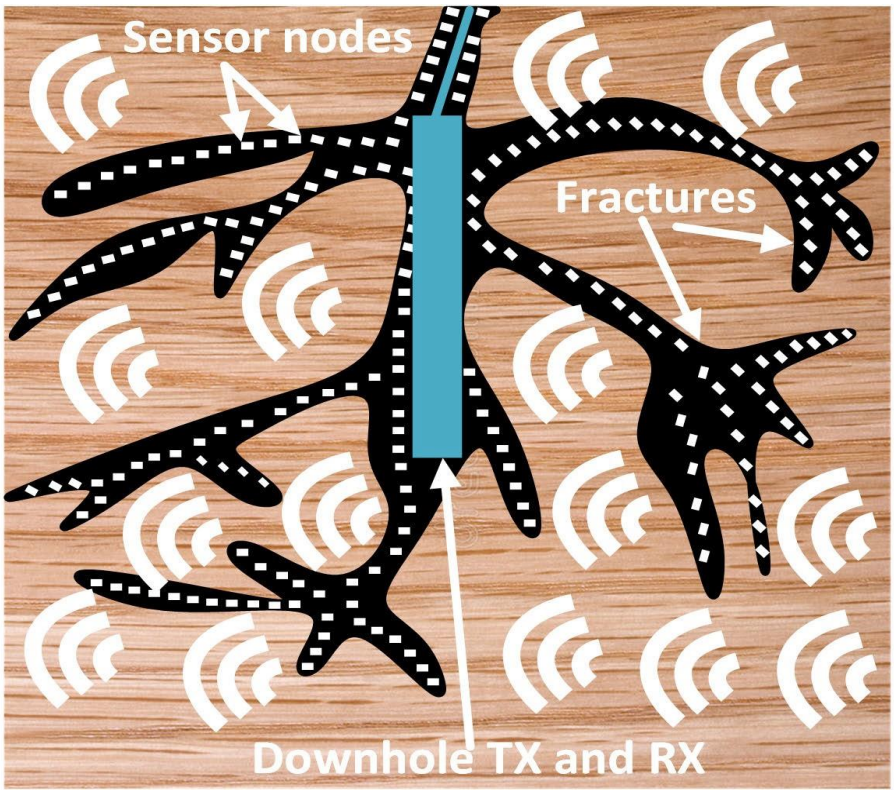


Figure 14: Conceptual representation of fracture mapping using WSNs.

In recent years, hydraulic fracturing has been undertaken in high-temperature rock formations for applications, such as obtaining natural gas from unconventional reservoirs [18].

These reservoirs are often located at large depths inside the surface at temperatures close to 250^{o}C and pressures close to 7 MPa. Ensuring the functionality of the system at high temperatures and pressures is, therefore, an essential part of using WSNs for fracture mapping. Mobility variation, threshold voltage reduction, and junction leakage currents are the major factors affecting the use of standard silicon complementary metal–oxide–semiconductor (CMOS) processes at high temperatures.

Therefore, silicon carbide (SiC), silicon-on-insulator (SOI), and other III–V semiconductors were conventionally considered the materials of choice for designing microchips at such temperatures [19], [20], [21]. Scaling of CMOS process nodes has resulted in an increase in doping concentration, resulting in lower junction leakage current at high temperatures. However, scaling also results in a lower value of threshold voltage, leading to more channel leakage. Since the junction leakage dominates at high temperatures, standard silicon CMOS processes have been recently used for high-temperature applications in fields, such as aerospace, automobile, and deep-well drilling [22], [23], [24].

This work presents a wirelessly powered system of coherently transmitting sensor nodes having a small form factor, low measurement latency, and ultralow power consumption. These nodes use the received 40.68 MHz radio frequency (RF) signal to transmit back a subharmonic 13.56 MHz

RF signal, therefore aggressively minimizing power consumption in comparison with the traditionally used transmitting techniques, such as oscillators or phase-locked loops (PLLs).

The amplitude of the received signal can be used to detect the location of the node. These nodes are experimentally verified to transmit power coherently, enabling their use in WSNs. The nodes were also used for fracture mapping applications using a prototype in both one and two dimensions. Moreover, these nodes are verified to transmit back power at temperatures up to 250 $^{\circ}$C and pressures up to 24 MPa. Therefore, they are viable to be used for fracture mapping at high temperatures and pressures in oil and gas fields for applications, such as hydraulic fracturing.

4.3 Smart Microchips Proppants Technical Specifications and System Details

The proposed system for fracture mapping includes several printed circuit boards (PCBs) fabricated on FR4 or flexible polyimide substrates acting as nodes for receiving and transmitting RF signals. These PCBs are hereafter referred to as localizers in the rest of this article. Fig. 15 illustrates the localizer and annotates its dimensions.

Each localizer consists of a microchip and two identical coils. The RF power transmitted by the TX coil is received by the on-PCB downlink (DL) coil, while the RF power is transmitted from the microchip using the on-PCB uplink (UL) coil, which is received by the RX coil. Capacitors of values 16.8 and 160 pF are connected in parallel to the DL and UL coils, respectively. This causes the DL and UL coils to resonate at 40.68 and 13.56 MHz, respectively, improving the received and transmitted power at these desired frequencies.

The diameter of the DL and UL coils is limited to 8 mm to minimize the form factor of the entire system. Since the inductance of coils of such dimensions is very low: 1) the trace widths of these coils are minimized; 2) the number of turns is maximized; and 3) the thickness of the substrate is reduced to obtain the maximum possible value of inductance. A larger inductance improves the link efficiency for wireless power transfer (WPT) by increasing the mutual inductance between two coils. Therefore, these coils are fabricated on a 0.2 mm-thick two-layer FR4 substrate with six turns on each layer.

Since these coils have a small size, the parasitic inductance of connectors is comparable to the coil inductance, making it unfeasible to measure their S-parameters or inductance using commercially available equipment, such as a vector network analyzer (VNA) [25], [26].

Figure 16 (a) – (c) shows the simulated real part of Z_{11}, inductance, and quality factor of these coils using Ansys HFSS, respectively. It can be observed that these coils have a self-resonant frequency (SRF) of 100.74 MHz, an inductance of 0.96 μH, and quality factors of 63.77 and 48.59 at our desired frequencies of 40.68 and 13.56 MHz, respectively. Since these coils are connected to high-impedance nodes, they are not matched to 50 Q for maximum power transfer.

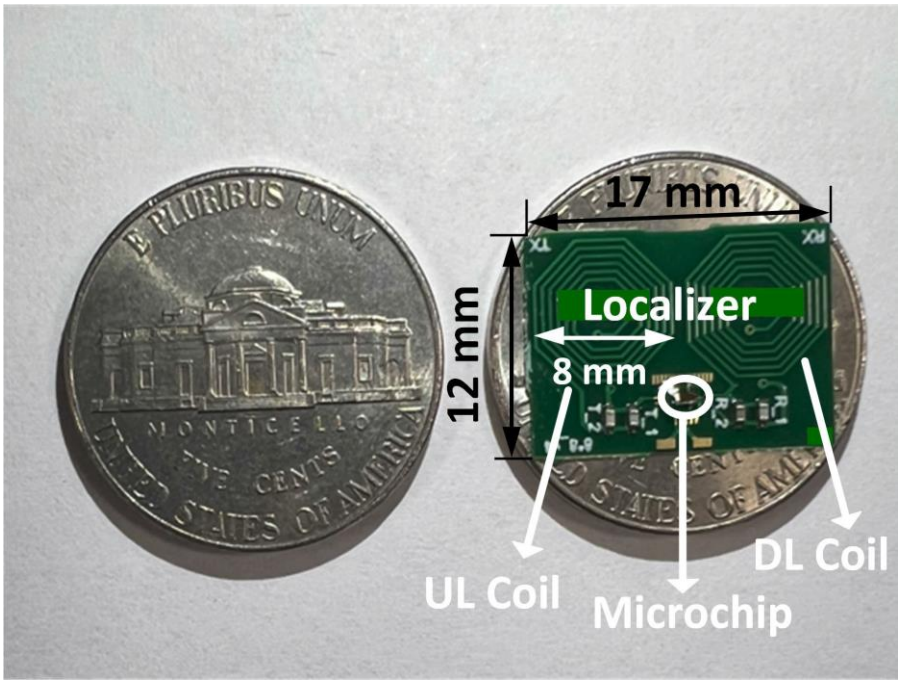


Figure 15: Localizer image with size compared with a nickel.

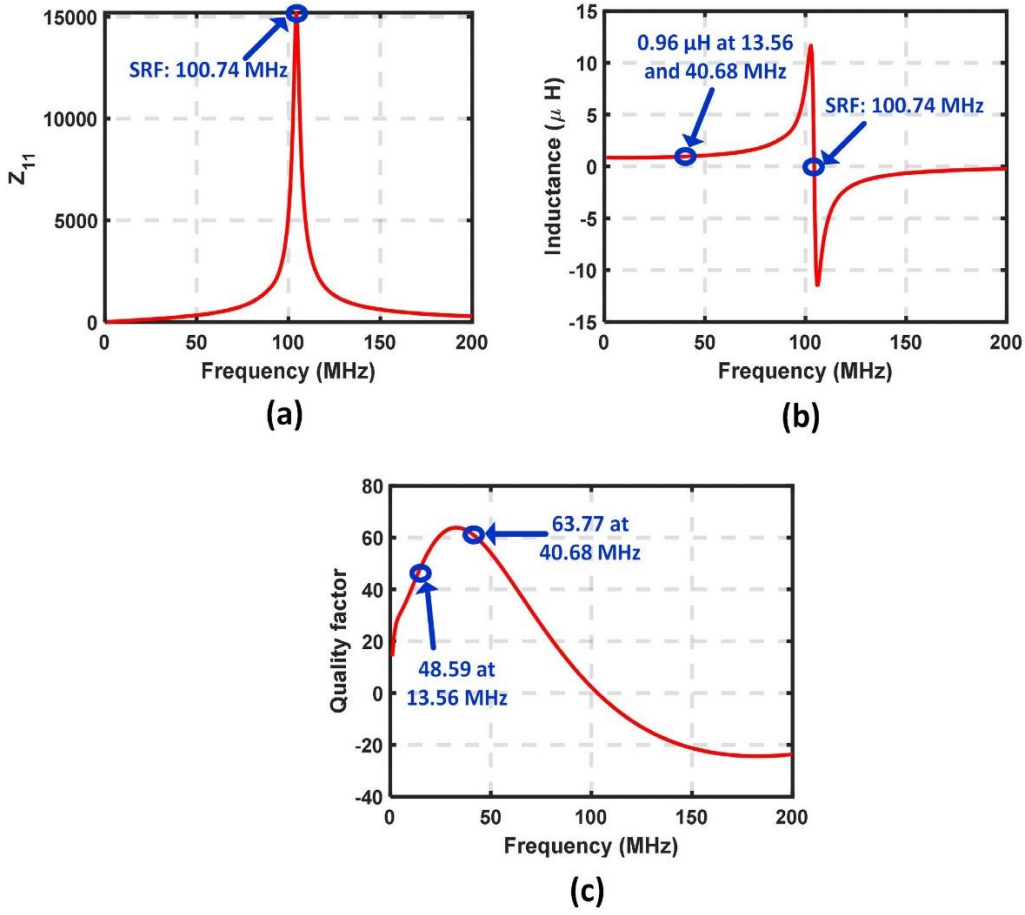


Figure 16: Simulated (a) re(Z_{11}), (b) inductance, and (c) quality factor of the localizer coil.

The TX and RX coils are designed to be large enough to transmit and receive a sufficient amount of power to and from the localizer, respectively. Therefore, the TX coil has a diameter of 3.5 cm, while the RX coil has a diameter of 4.5 cm. Both these coils are fabricated on a 1.6 mm-thick two-layer FR4 substrate. The TX coil has three turns, while the RX coil has six turns on each layer. The performance of these coils was measured using a Keysight N5230C PNA-L Network Analyzer. Figs. 17 (a)–(c) and 18 (a)–(c) show the measured real part of Z_{11}, inductance, and quality factor of the TX and RX coils, respectively.

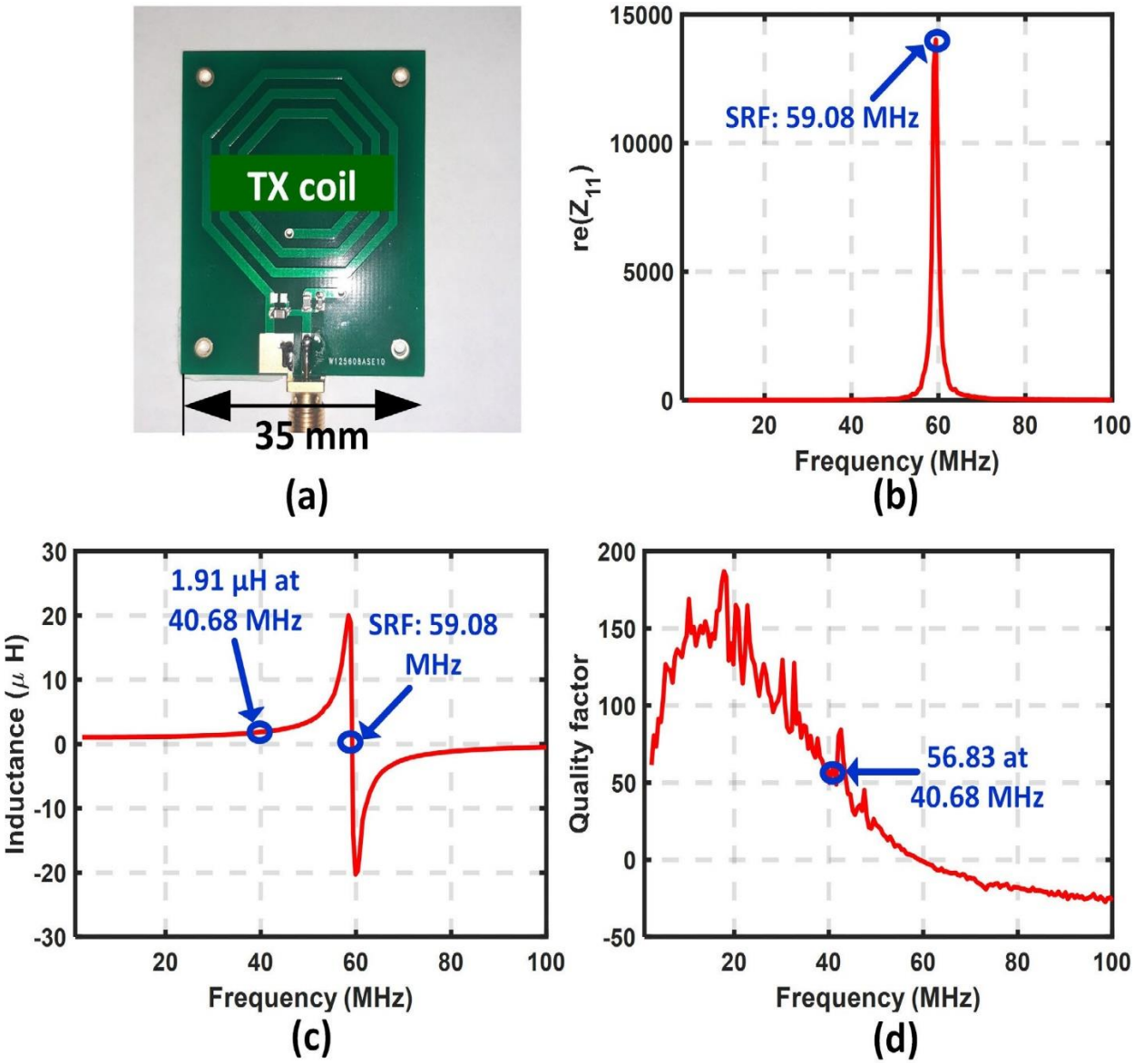


Figure 17: (a) TX coil and its measured (b) re(Z_{11}), (c) inductance, and (d) quality factor.

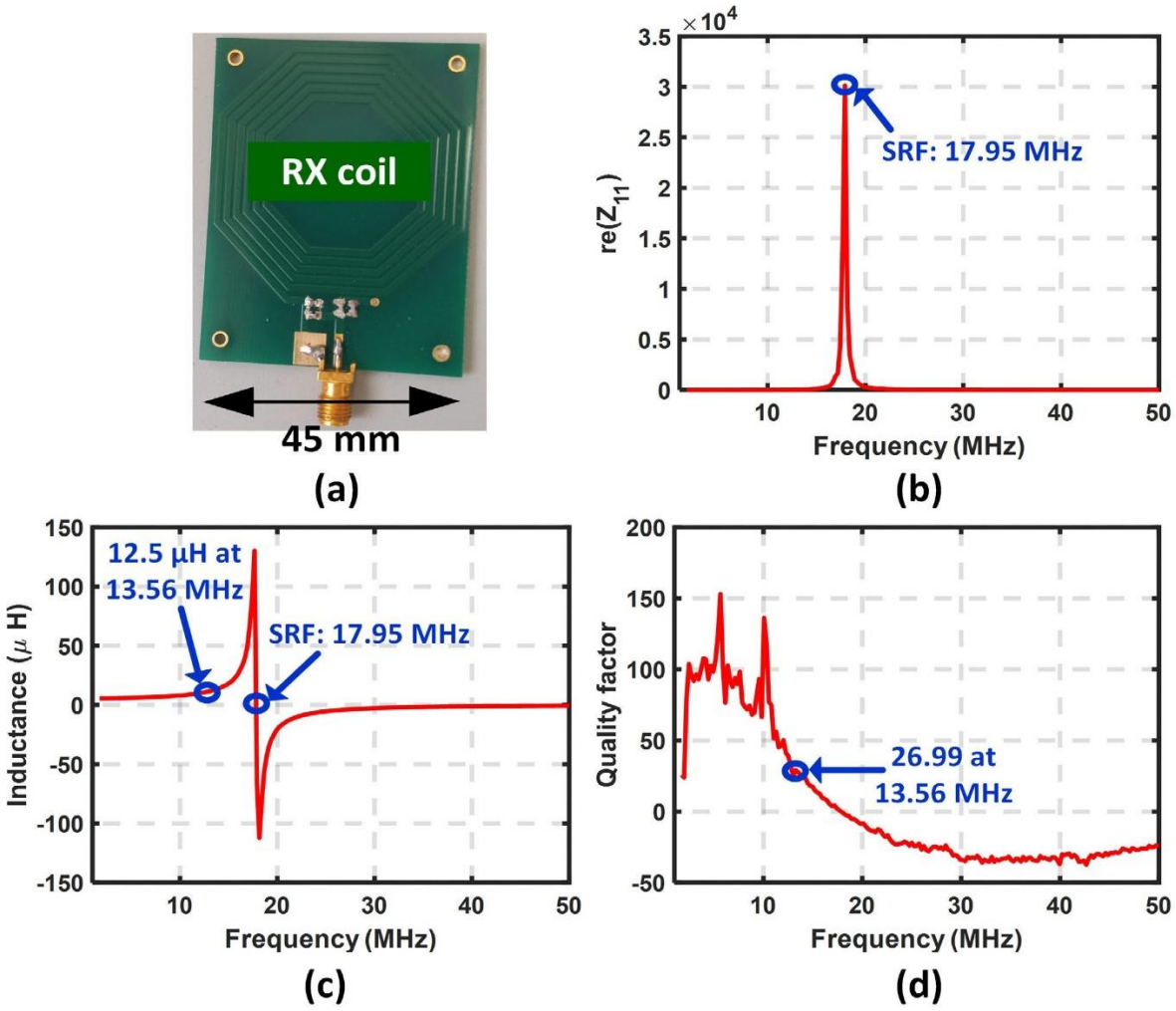


Figure 18: (a) RX coil and its measured (b) re(Z_{11}), (c) inductance, and (d) quality factor.

It can be observed that the TX coil has an SRF of 59.08 MHz, an inductance of 1.91 μH, and a quality factor of 56.83 at 40.68 MHz, while the RX coil has an SRF of 17.95 MHz, an inductance of 12.5 μH, and a quality factor of 26.99 at 13.56 MHz. The TX and RX coils are matched to 50 Q for maximum power transfer. Fig. 19 (a) and (b) show the measured magnitude of S_{11} for the matched TX and RX coils, respectively. Simulations were done in Ansys HFSS to estimate the path loss between the TX and localizer coils. Fig. 20 shows a simulated path loss of −43.765 dB at a separation of 6 cm.

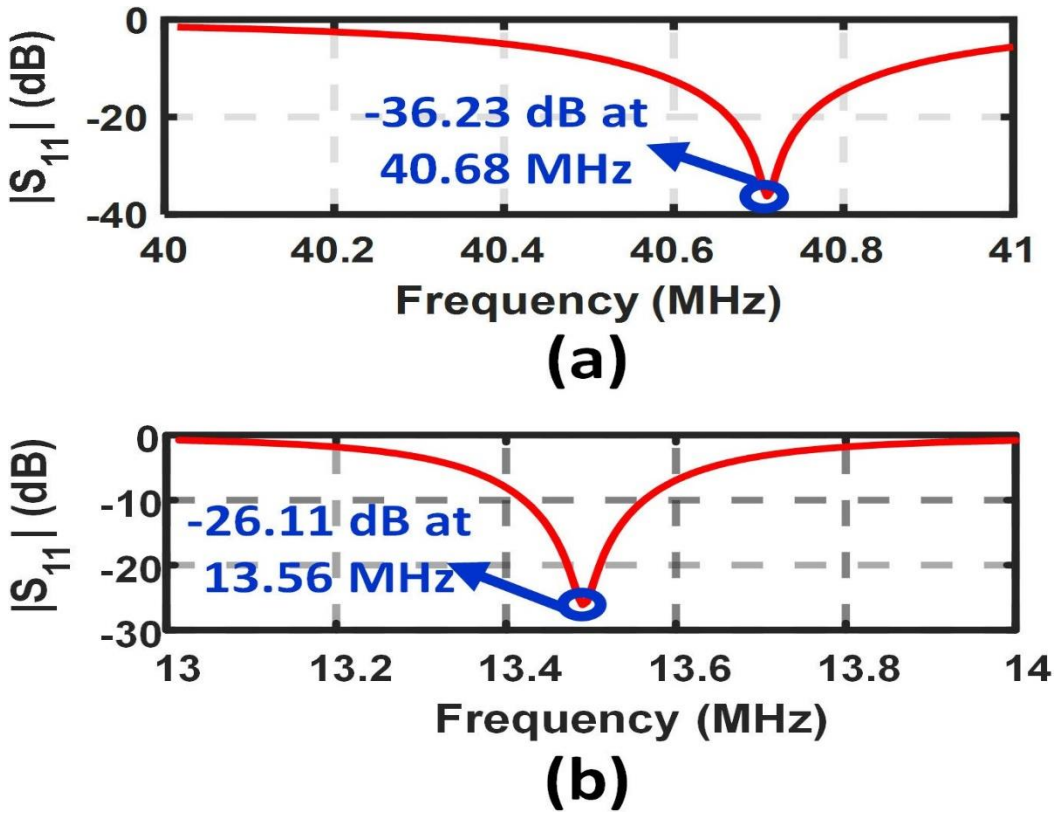


Figure 19: Measured $|S_{11}|$ for matched (a) TX and (b) RX coils.

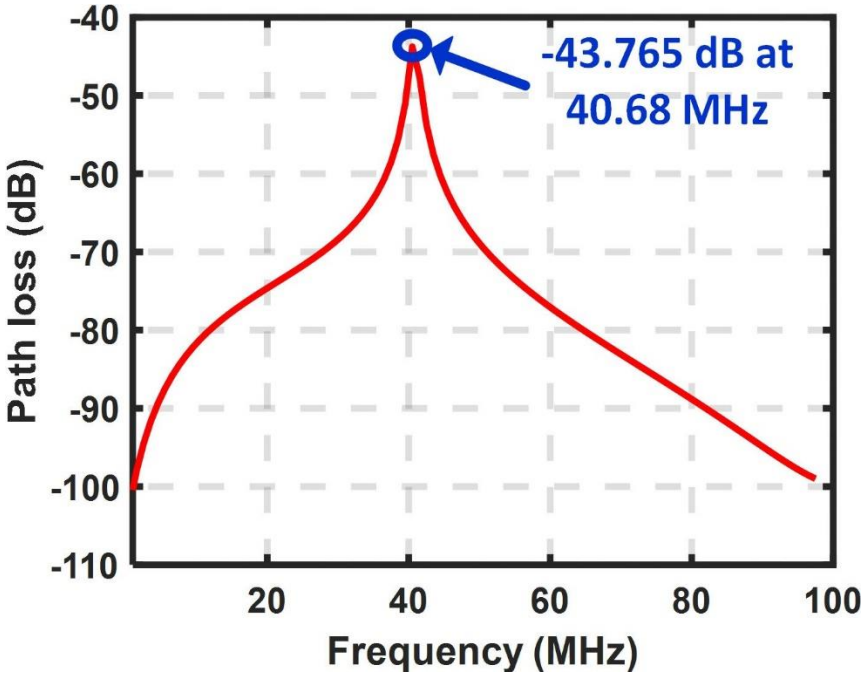


Figure 20: Simulated path loss between TX coil and localizer coil at a separation of 6 cm.

The microchip includes a full-wave rectifier, a diode limiter, and a digital divide-by-3 circuit. The block diagram of the localizer, including the microchip, is shown in Fig. 21.

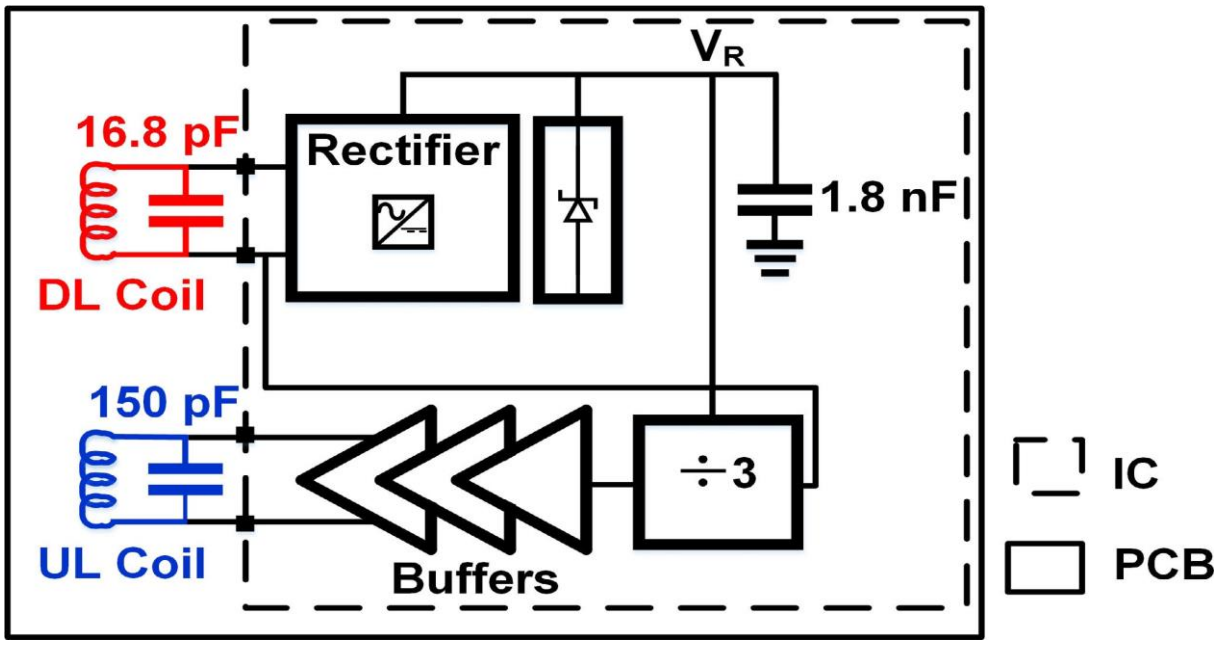


Figure 21: Localizer architecture.

A passive one-stage cross-coupled topology is chosen for the rectifier, as shown in Fig. 22.

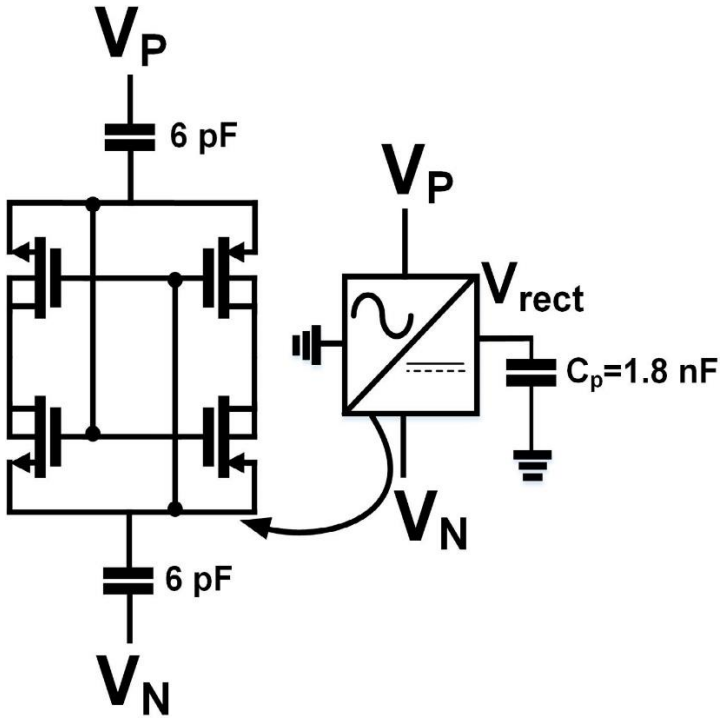


Figure 22: Rectifier schematic.

Cross-coupled rectifiers have been shown to have higher power conversion efficiency (PCE) than other topologies [26], [27], [28], improving the link efficiency of the system. Coupling capacitors of 6 pF are used for the rectifier to pass the differential input signal with less than 1% attenuation. The transistors are sized to have maximum efficiency. A 1.8-nF on-chip storage capacitor (C_p) is used to reduce the ripple at the output of the rectifier.

The diode limiter limits the output voltage of the rectifier to 3.8 V, providing over-voltage protection to the microchip. The output voltage of the rectifier V_R works as a supply voltage for the digital divide-by-3 circuit, which divides the frequency of the received RF power signal by 3. It includes a mod-3 counter and D flip-flops with an asynchronous set–reset operation.

The divide-by-3 ensures subharmonic locking of the frequency of the received RF signal to that of the transmitted signal. The divide-by-3 circuit has an extremely low power consumption of 1.5 μW for the lowest amplitude of the RF input signal, for which it generates a measurable output.

Since multiple localizers placed at different locations transmit signals back to the RX coil, it is imperative to synchronize the frequency and phase of the signals received from different localizers, so that they can constructively add up at the RX. Conventionally, this has been done using oscillators in PLLs as TXs [29].

However, PLLs have the drawbacks of using an off-chip crystal oscillator for precise reference frequency generation. Moreover, PLLs are extremely power-hungry and, therefore, unsuitable for wireless powering applications. Another feasible way to do this is by using injection-locked power oscillators such as TXs [30], [31], [32].

However, these oscillators consume static power and have a limited locking range, which decreases with an increase in the quality factor of inductors, requiring a higher amplitude of injection RF signal for certain applications. For our application, the phase differences between different localizers can be neglected, since the operating range is lesser than the wavelength at the operating frequency by at least an order of magnitude.

The divide-by-3 circuit does not consume static power and does not require the generation of an injection RF signal. Moreover, the frequency of the transmitted signal is always locked to the frequency of the received signal, enabling the use of the entire bandwidth allowed by the industrial, scientific, and medical applications (ISM) band.

The PCE of the rectifier and the peak-to-peak values of the divider output are simulated. Fig. 23 (a) and (b) show the PCE of the rectifier and peak-to-peak divider output voltage versus input power for one, two, three, and four-stage rectifiers, respectively. Reducing the sensitivity and increasing the transmitted power is paramount for increasing the operating range of the system. The sensitivity of the system is determined by that of the rectifier and the divider. The sensitivity of the rectifier is defined as the minimum input power for which it generates a supply voltage that is large enough for the divider. The sensitivity of the divider, on the other hand, is defined as the minimum input power for which it generates a detectable peak-to-peak output. From Fig. 23, it is observed that reducing the number of stages in the rectifier increases the sensitivity of both the rectifier and the divider. Increasing the number of stages generates a

larger supply voltage and, therefore, more transmitted power but reduces the sensitivity. Since a much larger RF power is required to power the microchip than detect the transmitted power, sensitivity dominates over transmitted power. Therefore, only one stage is used for the rectifier.

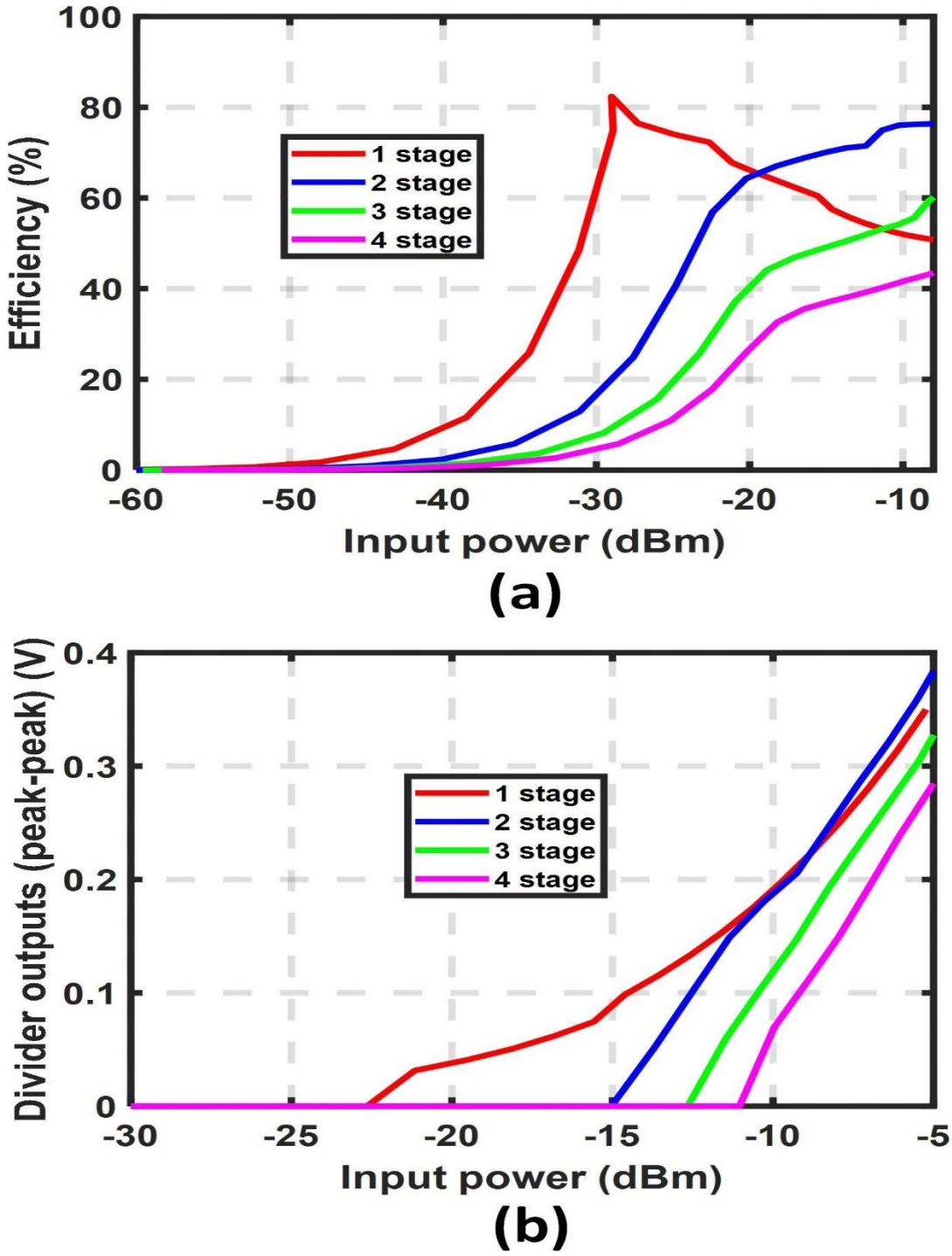


Figure 23: Simulated (a) rectifier efficiency and (b) peak-to-peak divider output voltage versus input power for one, two, three, and four-stage rectifiers. One-stage rectifier shows the best sensitivity.

4.4 Laboratory Testing of Smart MicroChipsfor for Fracture Mapping Under High Pressure and High Temperature

The annotated die micrograph of the microchip having the dimensions of 1.1 mm × 0.56 mm is shown in Fig. 24. The microchip was fabricated using the Taiwan Semiconductor Manufacturing Company (TSMC) 0.18 μm process. A signal generator (HP 8340B) generates the RF signal used for powering the microchips. The RF power of the signal was increased by connecting the output of the signal generator to a power amplifier (PA) (Minicircuits ZHL−20W−13+) having a small signal gain of 50 dB and a saturated output power of 20 W, increasing the operating range of the system.

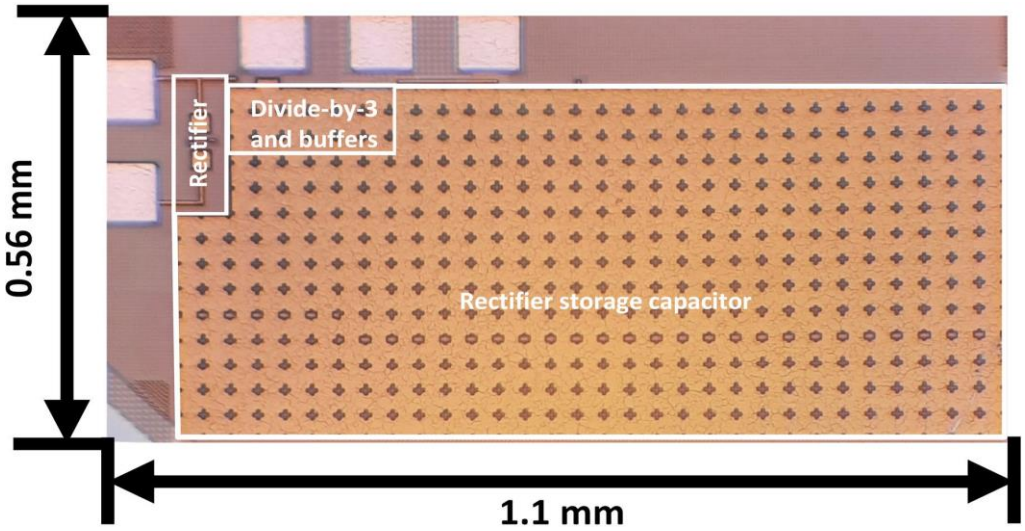


Figure 24: Die micrograph.

A spectrum analyzer (Tektronix RSA 306B) is used to measure the magnitude of the received RF signal. The following sections 4.4.1 to 4.4.6 report all the measurements performed in the lab using the localizers.

4.4.1 Energy Harvesting Verification

For this measurement, the input terminals of the rectifier are wired to the RF signal generator to generate an RF signal at 40.68 MHz, and the dc output of the rectifier is measured using an oscilloscope. Fig. 25 shows the rectifier output voltages for different root mean square (rms) input voltages. It is observed that the rectifier generates sufficient dc voltage to power the microchip at input rms voltages above 0.8 V.

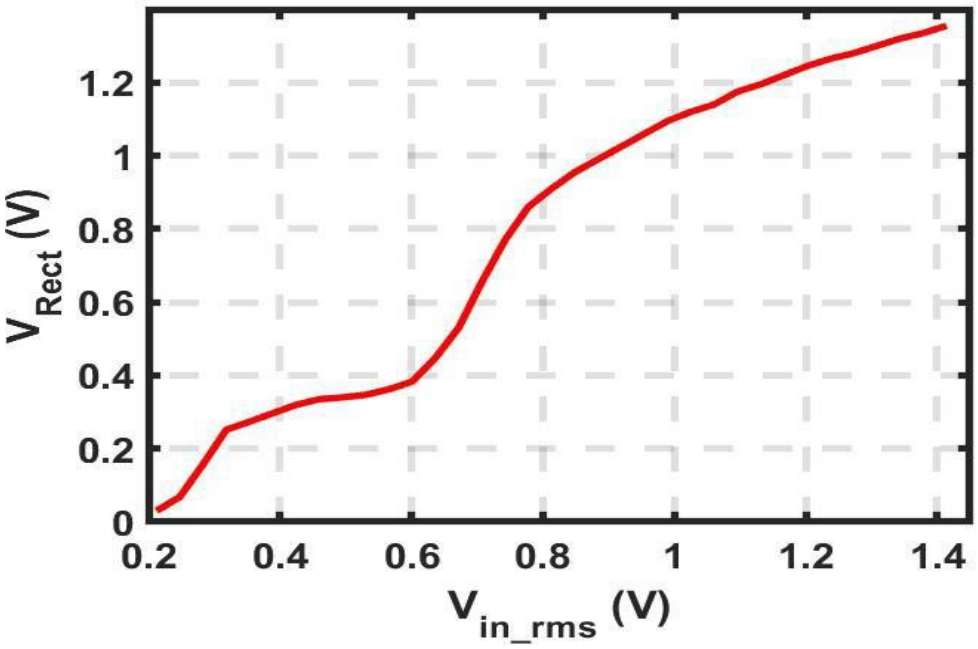


Figure 25: Measured rectifier voltage with respect to input rms voltage.

4.4.2 Microchip Functionality Verification

The measurement setup used for verifying the functionality of the microchip is illustrated in Fig. 26. The maximum operating range of the system is determined using the largest amplitude of RF signal available in the laboratory.

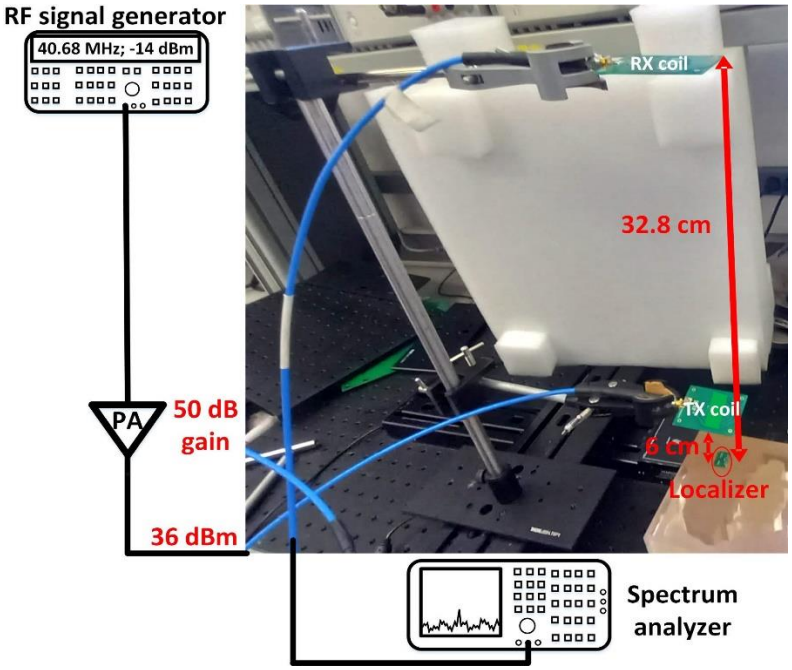


Figure 26: Measurement setup used for chip functionality verification.

Using the RF signal generator, a 40.68 MHz RF signal with −14 dBm power is delivered to the PA with a small signal gain of 50 dB. Therefore, an RF power of 36 dBm is delivered to the TX coil. The RX coil is connected to the spectrum analyzer for detecting the received power from the localizer. Keeping the RX coil very close to the localizer, the distance between the localizer and the TX coil is increased until the received 13.56 MHz tone merges with the noise floor of the spectrum analyzer. It is observed that the localizer can be wirelessly powered at a maximum distance of 6 cm between the localizer and the TX coil with 36 dBm power. Increasing the TX power would increase this maximum distance. Keeping the distance between the localizer and the TX coil fixed at 6 cm, the distance between the localizer and the RX coil is then increased until the received 13.56 MHz tone merges with the noise floor of the spectrum analyzer. It is observed that the RF signal transmitted from the localizer can be received by the RX coil at a maximum distance of 32.8 cm from it. While performing this measurement, since the received signal frequency was locked to that of the transmitted signal, the span and resolution bandwidth of the spectrum analyzer could be reduced, helping in lowering its noise floor. Fig. 27 shows the received 13.56 MHz tone having −118.15 dBm power at 6 and 32.8 cm distance between the localizer and TX and RX coil, respectively.

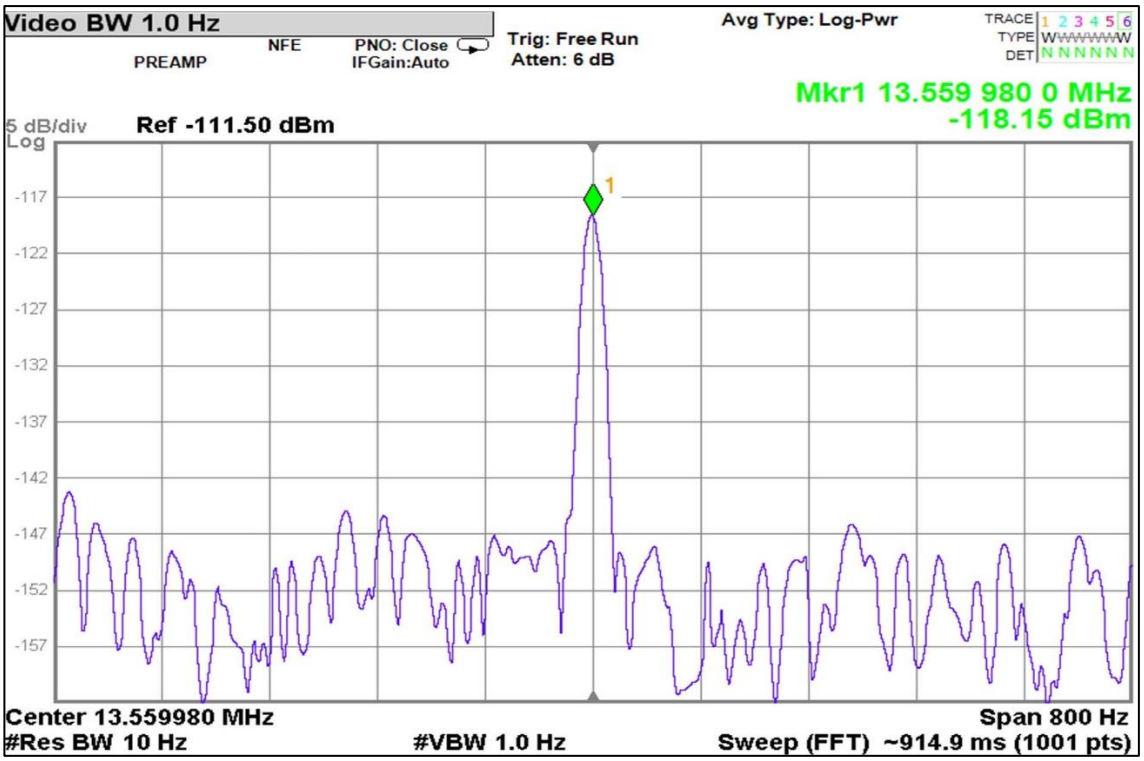


Figure 27: Received signal spectrum at the maximum operating range.

Using the same setup, the received power is recorded for different separations between the following: 1) the TX coil and the localizer DL coil and 2) the RX coil and the localizer UL coil. For each separation distance between the TX coil and the localizer DL coil, the lowest TX power required to power the microchip is used to maintain uniformity in the received power for different distances. Fig. 28 illustrates the contour plot of the received power profiles across different separations.

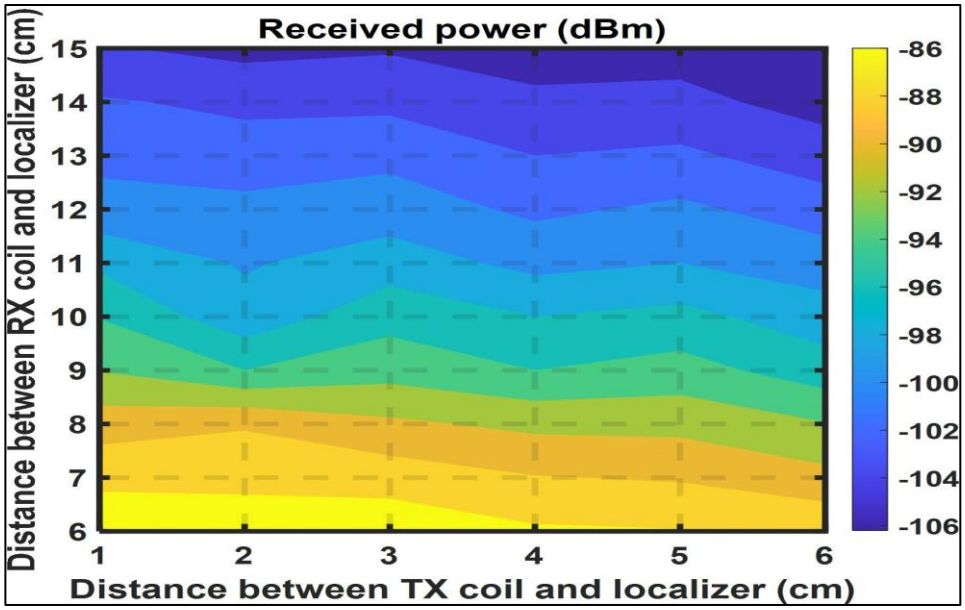


Figure 28: Received 13.56 MHz signal power in dBm across different separations between the TX coil and the localizer DL coil, and the RX coil and the localizer UL coil.

Two paper protractors are added around the TX and RX coils, as illustrated in Fig. 29, and the TX and RX coils are rotated to measure the power profiles at different angular orientations to the localizer. For the purpose of this experiment, a separation of 5.5 cm is used between the TX coil and the localizer, while a separation of 12 cm is used between the RX coil and the localizer.

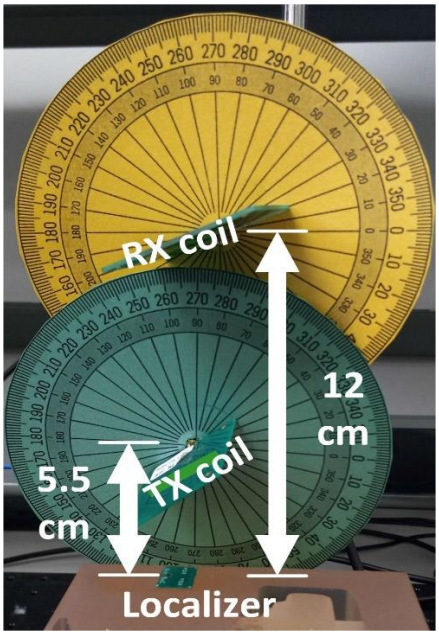


Figure 29: Measurement setup used for obtaining the received power profile with respect to different angular orientations of the TX and RX coils.

Fig. 30 illustrates the contour plot of the received power profiles across different angular orientations. It can be observed that the system is robust to angular misalignments of up to 60° between the TX coil and the localizer and up to 80° between the RX coil and the localizer.

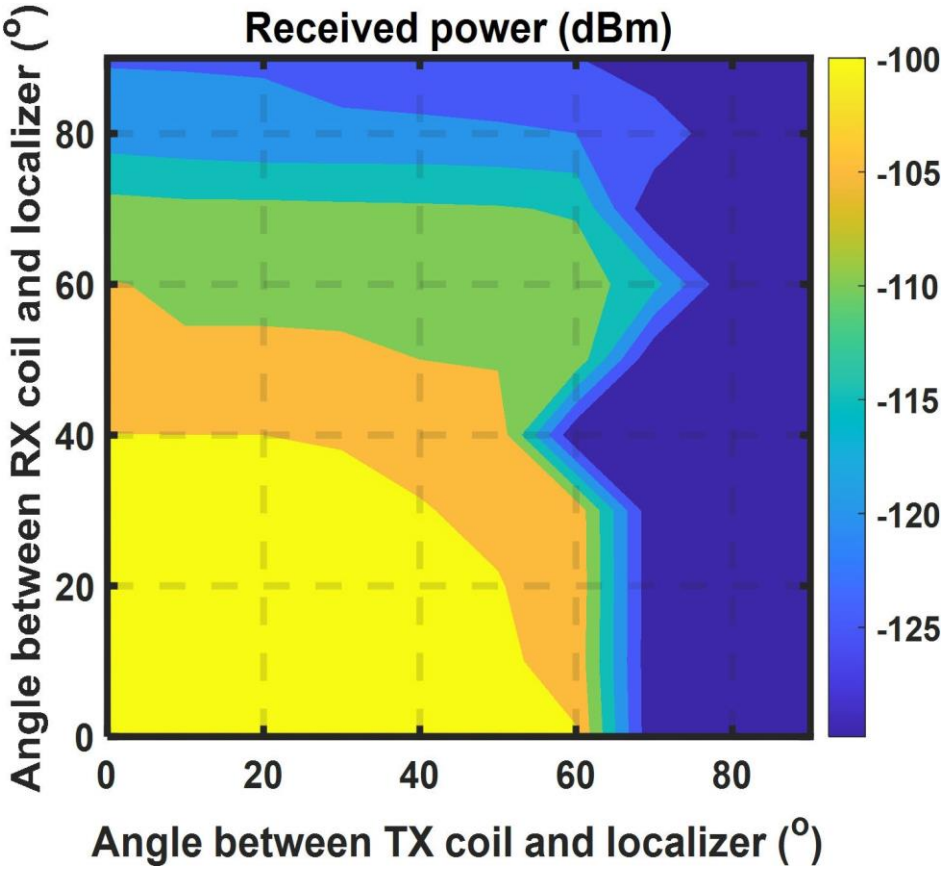


Figure 30: Received 13.56 MHz signal power in dBm across different angular orientations of the TX and RX coils.

As shown in Fig. 20, a path loss of −43.765 dB is expected between the TX coil and the localizer coil at a distance of 6 cm. From Fig. 19, the TX and RX coils have an $|S_{11}|$ of less than −20 dB at the desired frequencies. Therefore, the loss due to matching the TX and RX coils can be considered negligible. For a TX power of 36 dBm, de-embedding the path loss gives us a power of −7.765 dBm at the localizer coil. From the sensitivity simulations illustrated in Fig. 23, a minimum power of around −20 dBm is required at the input of the rectifier to obtain a measurable 13.56 MHz tone at the output. The rest of the losses can, therefore, be attributed to the imperfect matching of the impedance of the localizer coil to the dynamic input impedance of the rectifier.

4.4.3 Coherent Power Combining Verification

The measurement setup used for verifying the coherent power combining of RF signals transmitted by different localizers is illustrated in Fig. 31. A 40.68 MHz TX power signal of

−14 dBm is amplified to 36 dBm using the PA as in the previous measurement. In three different measurements, one, two, and three localizers are placed close to each other on a surface, such that they can be wirelessly powered by the same TX coil. The TX and RX coils are placed at a distance of 6 and 10 cm, respectively, from the localizers to transmit and receive RF signals from them. A spectrum analyzer (Tektronix RSA 306B) is connected to the RX coil to measure the power received from the localizers.

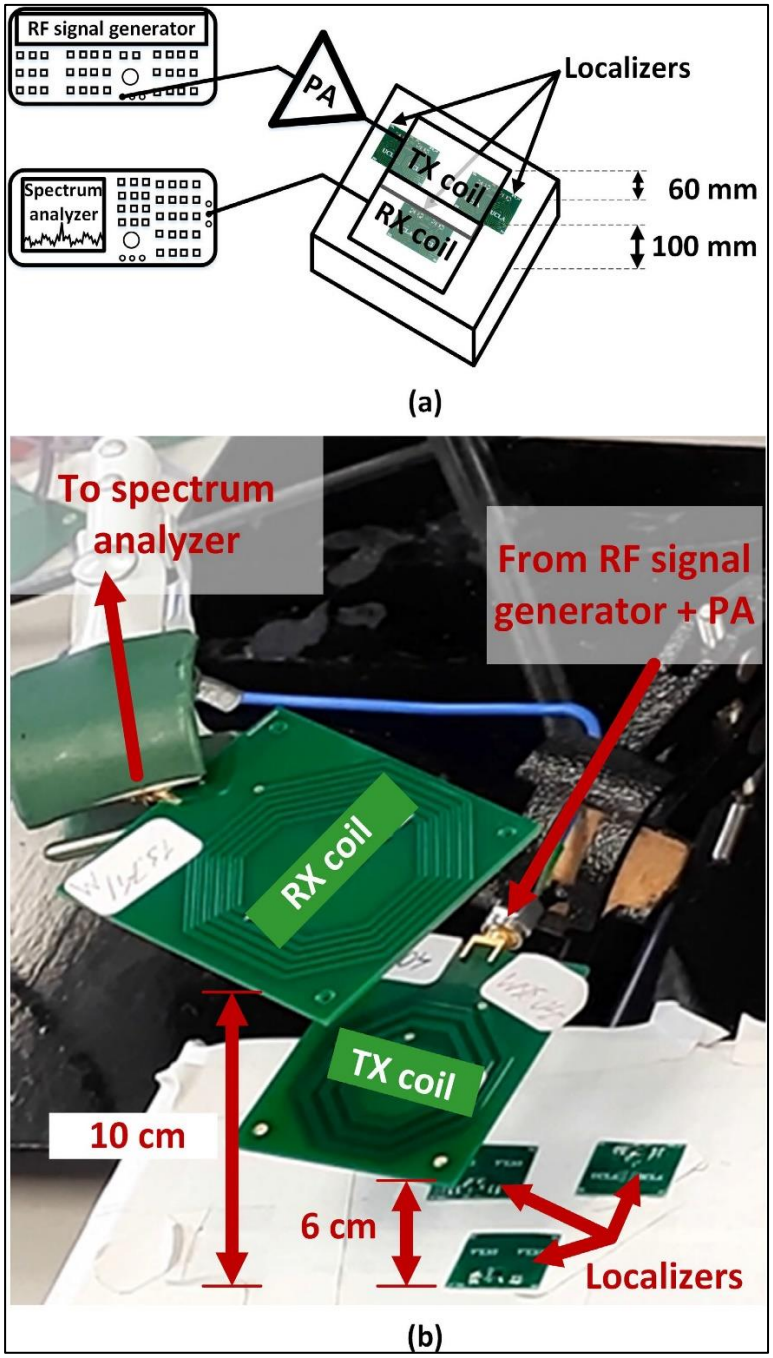


Figure 31: (a) Schematic and (b) picture of the coherent power combining measurement setup.

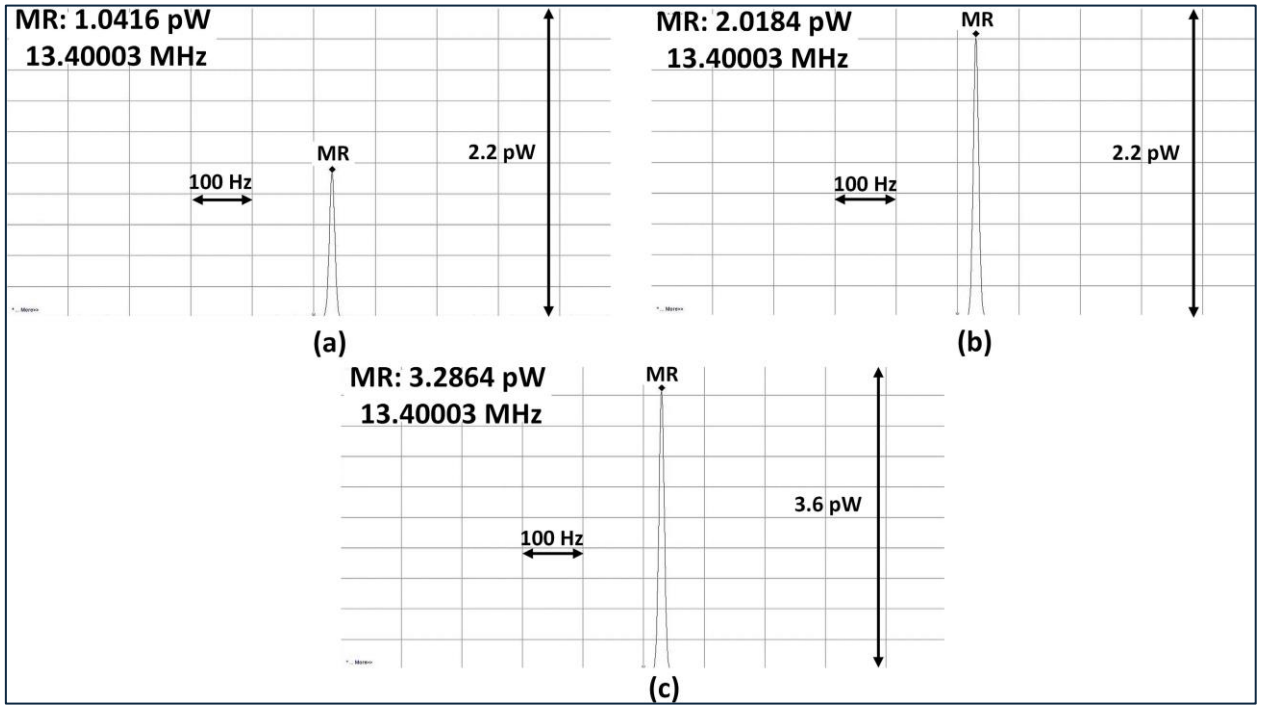


Figure 32: Received power from (a) one, (b) two, and (c) three localizers.

Fig. 32 (a)–(c) shows the power received at 13.56 MHz from one, two, and three localizers, respectively. It is observed that the power received from two and three localizers (around 2 and 3 pW, respectively) is almost equal to twice and thrice that received from one localizer (around 1 pW), which verifies the fact that the signals transmitted by the localizers add up coherently.

4.4.4 Fracture Mapping Verification

The measurement setup for fracture mapping is illustrated in Fig. 33. For this study, a prototype rock with fractures along its height was designed, and multiple prototypes were 3D-printed to test the functionality of smart microchips for fracture mapping in the lab, simulating hydraulic fractures typically found in oil and gas wells. The details of the design and generation of these synthetic core samples with complex fractures are provided in Appendix F.

The localizers are coated with nonconductive epoxy and are placed inside the fractures that are to be mapped. The distances of the TX and RX coils from the fractures are similar to the previous measurements. Using motorized rails in the $X-$ and Y-directions, the TX and RX coils are moved over the entire region that is desired to be mapped. The presence or absence of a fracture at a specific coordinate is determined by the presence or the absence of a 13.56 MHz tone in the received signal.

A spectrum analyzer is connected to the RX coil to record the spectrum at different coordinates, which is, in turn, connected to a laptop for data processing using MATLAB. The fractures are

mapped in the X - and Y -directions separately with a resolution of 1 mm in both X - and Y - directions.

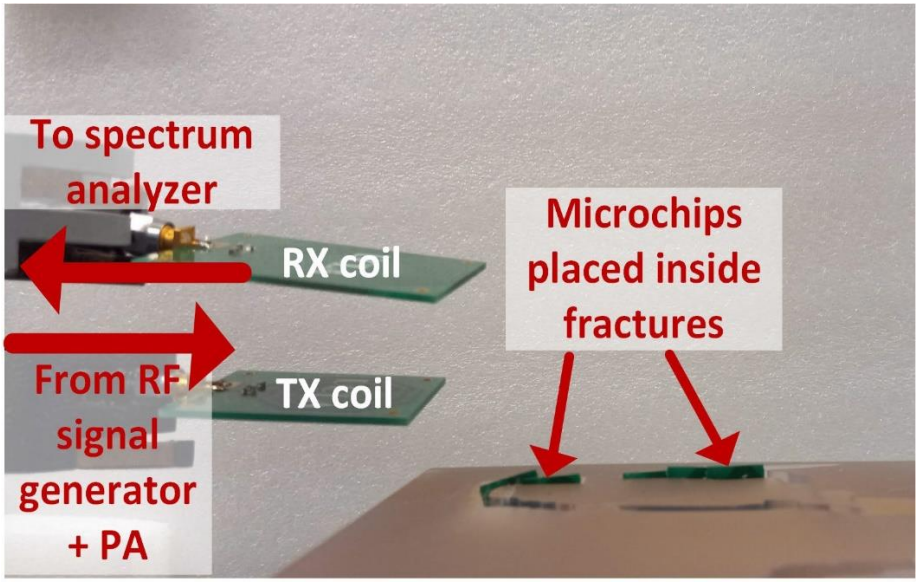


Figure 33: Fracture mapping setup.

Fig. 34 illustrates the fracture mapping results for the Y-Y-direction, while Fig. 35 illustrates the same for the X -X-direction. Fig. 23 illustrates the 2-D fracture mapping results. It is observed that the fractures are mapped with considerable accuracy using the localizers.

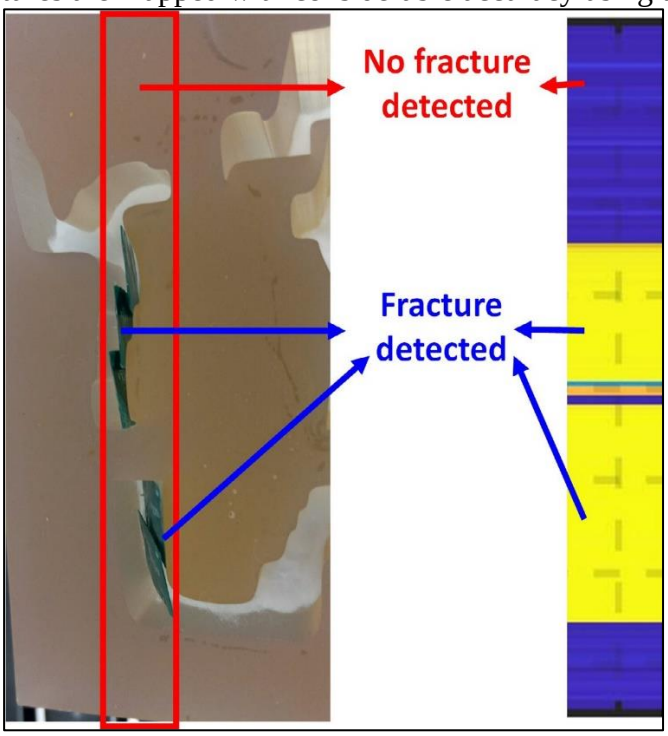


Figure 34: 1-D fracture mapping results for the y-direction. The red box indicates the region that was mapped by smart Microchips that are placed in the fractures (3D printed cores with fractures).

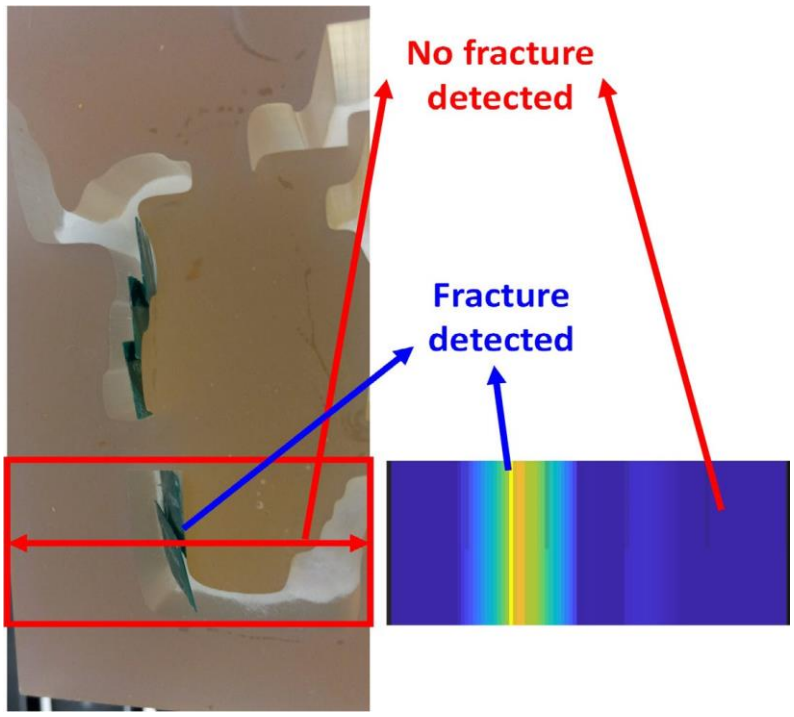


Figure 35: 1-D fracture mapping results for the x-direction. The red box indicates the region that was mapped by smart Microchips that are placed in the fractures (3D printed cores with fractures).

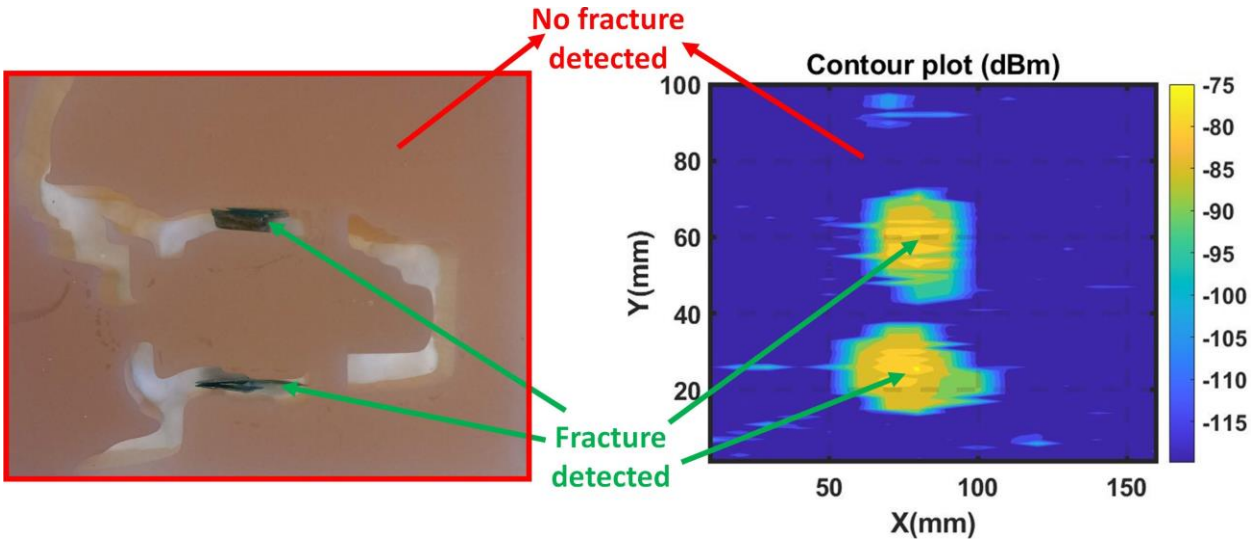


Figure 36: 2-D fracture mapping results by smart Microchips that are placed in the fractures (3D printed cores with fractures).

4.4.5 High-Temperature Verification

The functionality of the localizers is also verified at high temperatures. The schematic and picture of the measurement setup for the high-temperature measurements are shown in Fig. 37 (a) and (b), respectively. In this measurement, the localizer is placed inside a microwave

oven. The RF signal generator and PA are used to generate a 36 dBm 40.68 MHz power signal as in the previous measurements. It is observed that at the previously measured maximum distance of 6 cm between the TX coil and the localizer, the rectifier does not generate enough dc voltage for the proper functioning of the microchip at higher temperatures. The TX coil is, therefore, positioned at a distance of 2 cm from the localizer inside the oven, so that the microchip can be powered at all temperatures in the range of our measurements. The RX coil is connected to the spectrum analyzer and placed at a distance of 10 cm from the localizer. The temperature inside the oven is increased using the knob on the oven. The oven temperature is measured by an oven thermometer (Admetior Kitchen Oven Thermometer).

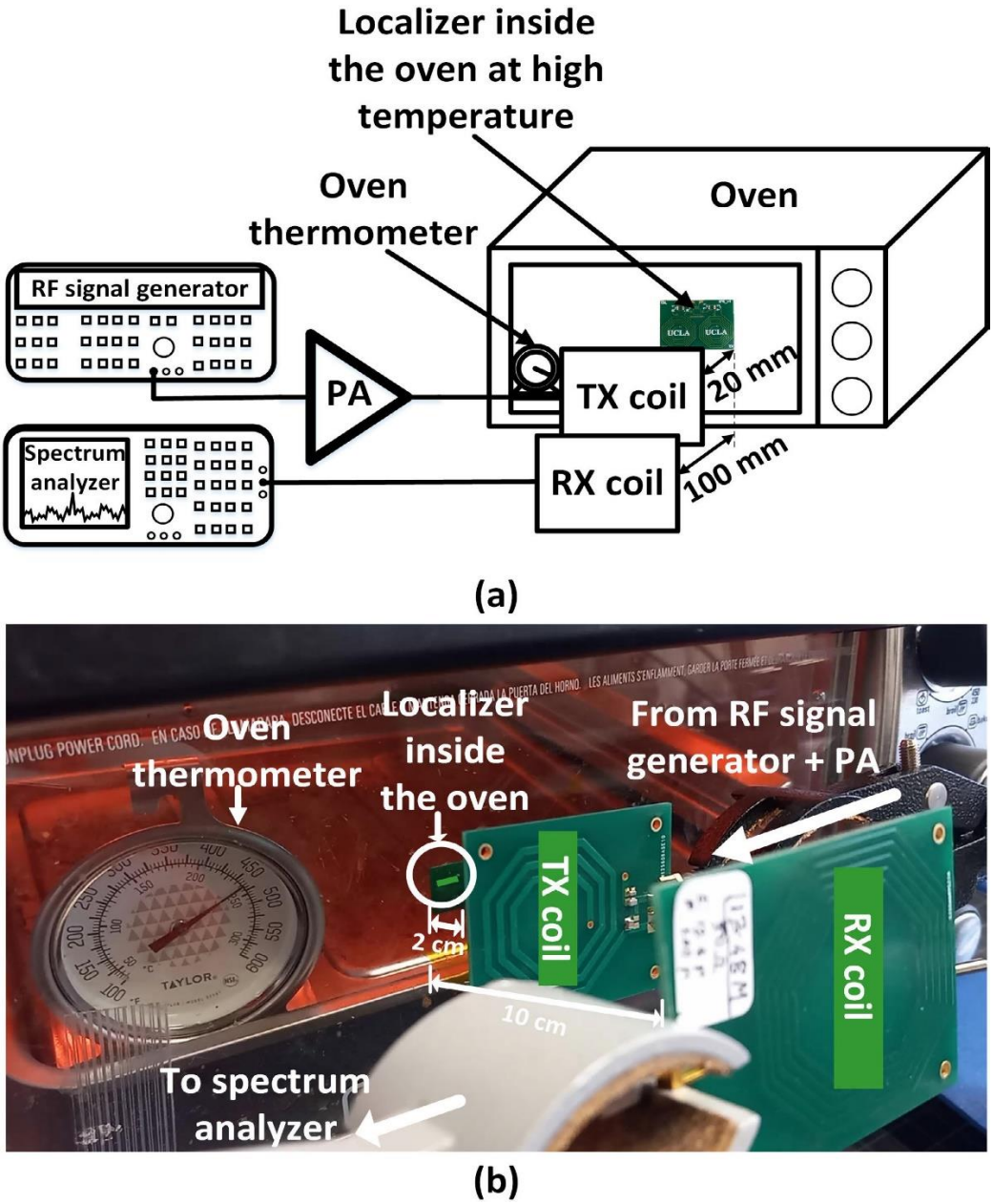


Figure 37: (a) Schematic and (b) picture of the measurement setup for verification at high

temperatures.

Fig. 38 shows the received signal power from the localizer at temperatures from 20 ◦C to 250 ◦C.

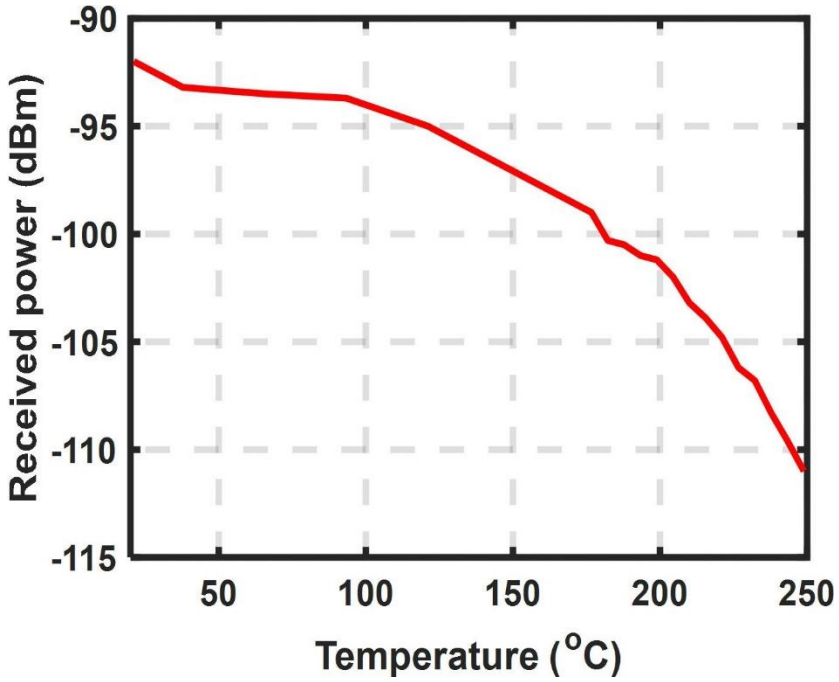


Figure 38: Received signal power at temperatures from 20 °C to 250 °C.

4.4.6 High-Pressure Verification

The localizers are also verified to work under high-pressure conditions using a concrete compression sensing machine. Wet cement is placed inside three different cube-shaped molds and allowed to cure for 24 hrs.

While pouring the wet cement into the molds, one localizer is placed in each mold at a depth of 1 cm from the bottom. Each cube with a localizer embedded in it is then placed in the compression sensing machine, and a compressive force is applied using the cylindrical pistons from the top and bottom. The high-pressure measurement setup is illustrated in Fig. 39.

The RF signal generator and the PA are used to generate a 36 dBm 40.68 MHz power signal as in the previous measurements. Due to the metallic nature of the compression sensing machine, the distance of the TX and RX coils from the localizer is reduced, such that the localizer can be powered.

The applied compressive pressure is slowly increased until the cement cubes fracture. The three cubes fracture at pressures of 8.53 MPa (1237 psi), 24.06 MPa (3490 psi), and 18.40 MPa (2669 psi), respectively.

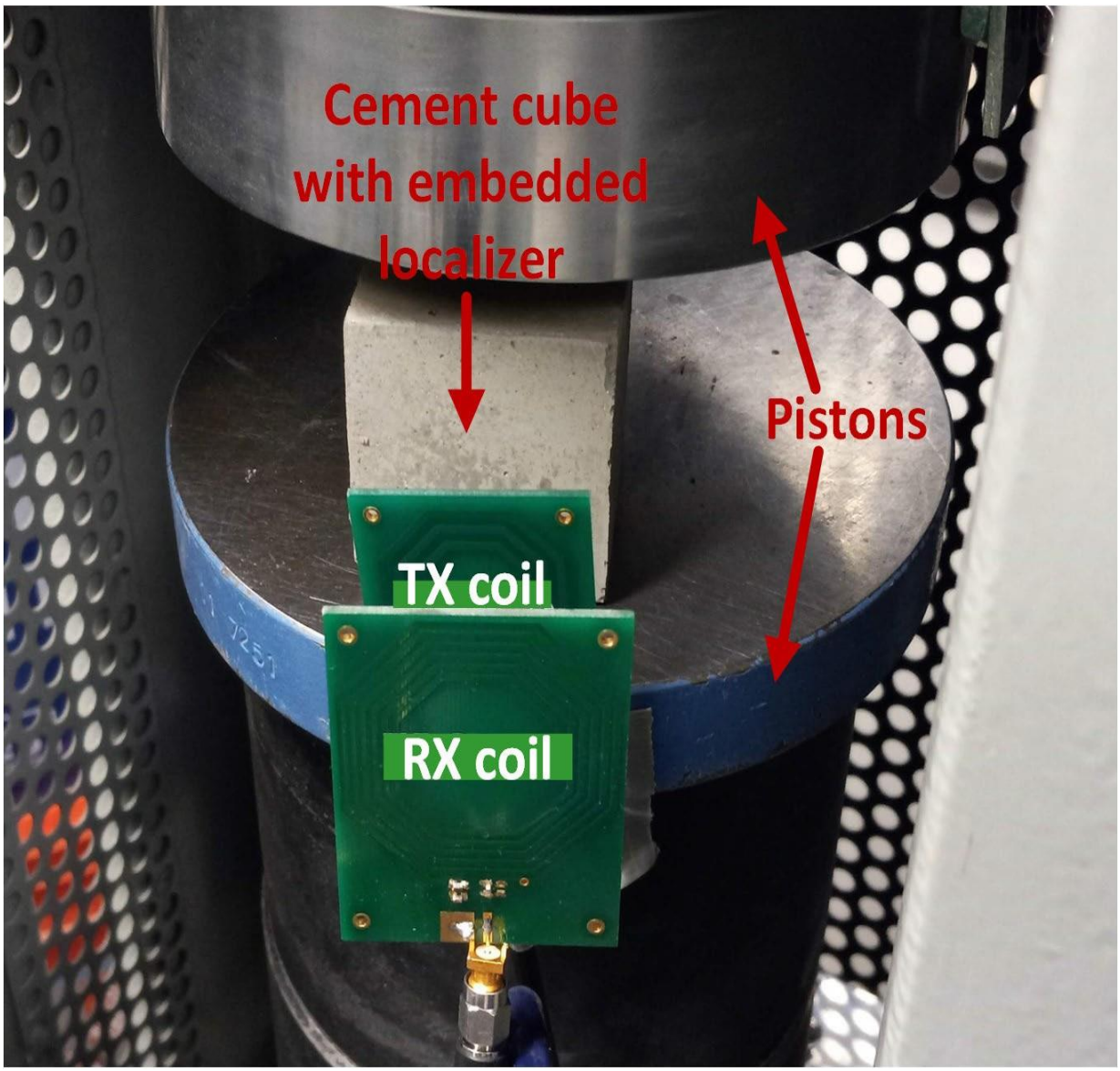
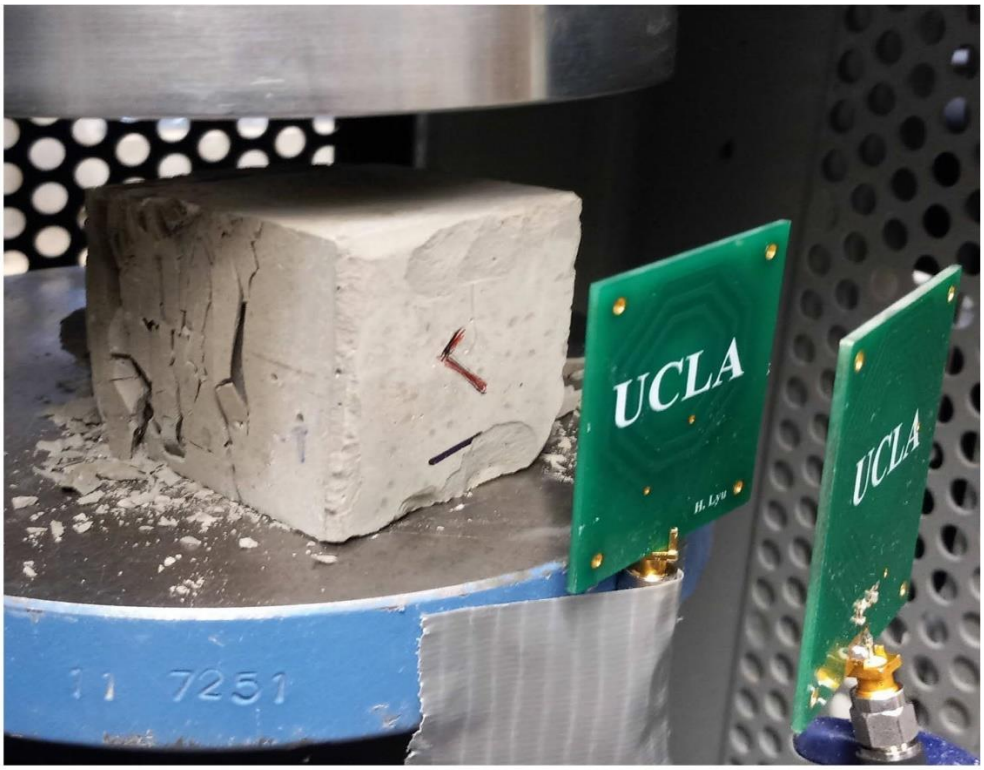
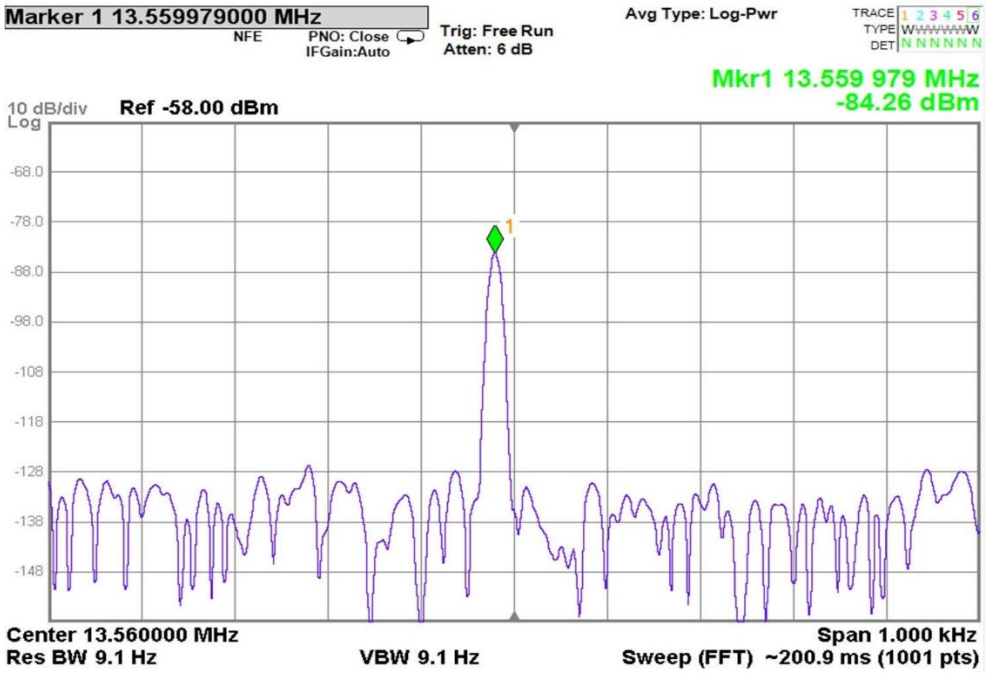


Figure 39: Measurement setup for verification at high pressures. The entire setup is inside a compression sensing machine.

Fig. 40 (a) shows the second cube after it fractures. The received 13.56-MHz tone from the localizer is still observed even at the highest of these pressures (3490 psi). Fig. 40 (b) shows the received signal from the second localizer when the second cube fractures at 24-MPa pressure.



(a)



(b)

Figure 40: (a) Second cube after it fractures at 24 MPa. (b) Received signal from the second localizer when the second cube fractures.

Please note that the verification of MicroChips' functionality under high pressure was conducted without epoxy protection. These tests were performed with chips embedded in cement under compressive forces. With proper epoxy protection during field testing, the chips are expected to withstand pressures of up to 10,000 psi.

In this series of lab testing and verification, a wirelessly powered Smart MicroChip proppants sensing system of coherent sensing nodes has been presented and proposed for use in fracture mapping applications at high temperatures. A power-efficient scheme using the RF power signal and a digital divide-by-3 circuit has been used to generate a locked subharmonic signal to be transmitted by the microchip. This enables an average power consumption of only 1.5 μW for the system. The system has also been verified to work reliably at temperatures up to 250^{o}C and pressures up to 24 MPa, which are one of the highest using a standard CMOS process. The system, having a small form factor and ultra-low power consumption, also finds use in other sensing and localization applications using WSNs.

4.5 Smart Microchip Proppants Manufacturing and Production for Field Testing

We successfully demonstrated the effectiveness of wireless localizers for fracture mapping in laboratory conditions. The wireless nodes consist of a sensing chip wire-bonded onto a miniaturized printed circuit board (PCB). The chip, fabricated using standard 180 nm technology, receives power at 40.68 MHz (RX) and transmits signals back at 13.56 MHz (TX). Both frequency bands are from the ISM bands. These nodes can reliably map fractures at temperatures up to 250°C and pressures up to 24 MPa and have an average power consumption of 1.5µW [30].

For the field testing of Smart MicroChips proppant technology, we have developed several different versions for this project. One of the versions utilizes two side-by-side coils with dimensions of 17 mm x 12 mm x 0.2 mm. Another version has dimensions of 9 mm x 12 mm x 1.6 mm and features a surface for powering and a surface-mount device inductor (SMDL) for transmitting signals (see Fig. 41). The second version offers orthogonal coupling directions for two frequencies, while the first version supports only one direction for RX and TX.

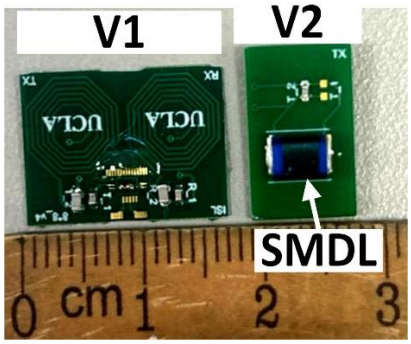


Figure 41: Two versions of PCBs developed for fracture mapping

4.5.1 Resonance Frequency Tuning

For our application, accurate frequency tuning of the samples is essential. We used a wide-band coil and a vector network analyzer (VNA) to detect the precise resonance frequencies of our samples. Accurate and high-quality tuning is crucial for maximizing power transfer efficiency and operational distance. The results of the frequency tuning and component values for each sample are shown in Fig.42.

Matching Network Parameters	
V1:C_{R1},C_{R2},C_{T1},C_{T2}	10 pF, 3.9 pF, 150pF, 0.5pF
V2:C_{R1},C_{T1},SMDL	22pF,18pF, 5.6µH

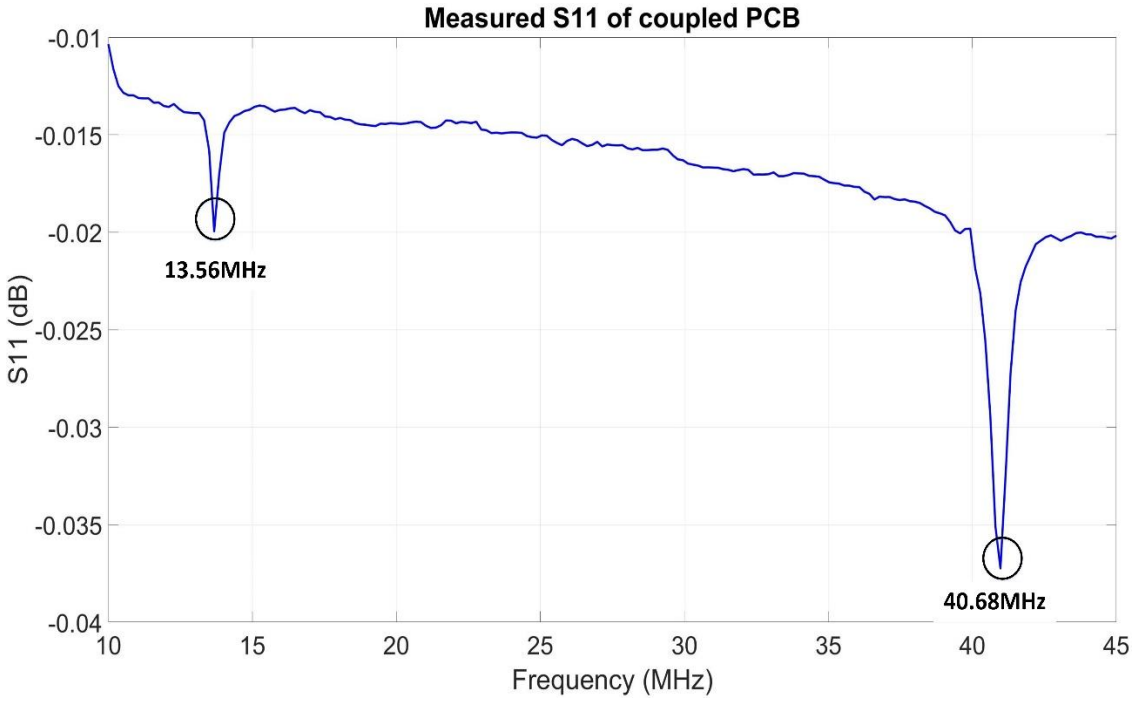


Figure 42: VNA measurements for the resonance of the PCB coils

4.5.2 Smart MicroChips Proppants PCB Mass Production

Both PCBs were measured to be operational at a distance of 4 cm using 20 dBm (100 mW) of RF power. They can function with a misalignment of 45° and a distance of 3 cm. For validation measurements, we used a single double-tuned coil (refer to [31]) for both transferring 40.68 MHz power and reading back 13.56 MHz signals (see Fig. 43). Following initial validation, we ordered 200 samples of each version for manufacturing. The assembly is performed in-house. The mass-produced PCBs are shown in Fig.44.

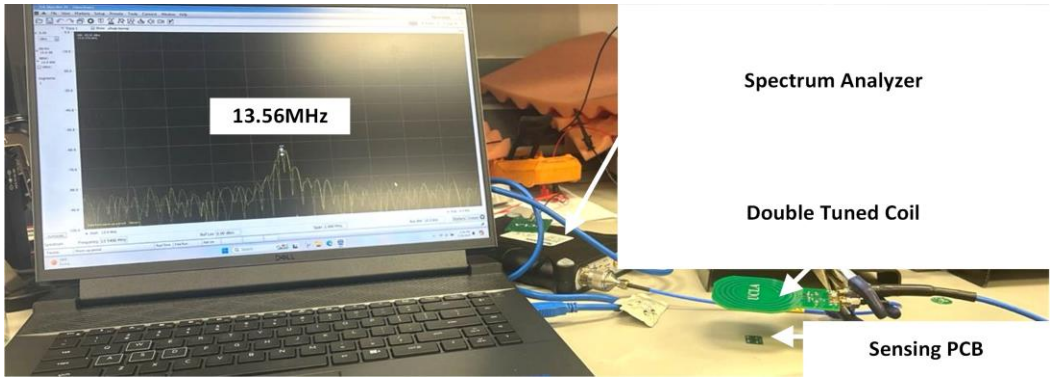


Figure 43: Setup picture for sensing node validation

Figure 44: Mass-produced PCBs

4.5.2 Smart MicroChips Proppants PCB Packaging- Resin-protected

In the end, it is crucial to protect the sensing nodes from heat and pressure with proper packaging. For the initial packaging, we are using UV-cured epoxy resin followed by a HPHT epoxy.

The HPHT epoxy is resistant to acids, bases, and solvents. Once cured, it exhibits several desirable physical properties, including high modulus and exceptional compressive strength. It is a toughened system that can endure rigorous thermal cycling. The formulation includes a quartz filler, which enhances its dimensional stability and abrasion resistance. Additionally, the epoxy is a reliable electrical insulator and has a service temperature range from -60°F to +450°F.

4.5.3 Additional Hardware and MicroChips Built During this Project

In addition to the final design reported in the previous sections, the team has performed a series of experiments on a custom wirelessly powered chip with a new miniaturized antenna configuration. The antenna uses an SMD inductor for receiving the wireless power at 40.68MHz ISM band and transmitting back a signal at 1/3 of the received frequency (13.56GHz ISM band). Figure 45 shows a picture of the pcb assembly. The chip dimensions are 5 mm X 15.5 mm X 1.6 mm, which is suitable for use in fractures that are up to 5 mm wide. This chip was designed to be sensitive to a magnetic field that is aligned to the longest dimension of the chip. This is suitable for fracs that

are orthogonal to the direction of the wellbore (and the downhole tool). Measurements were performed with a powering coil having a 5-cm diameter and matched at 40.68 MHz.

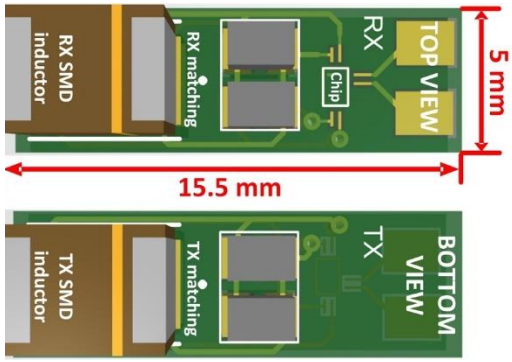


Figure 45: SMD-based antennas used in prior proppant chips

We achieved a wireless powering distance of 35 mm by using 1W of transmit power. Figure 46 shows the measurement setup.

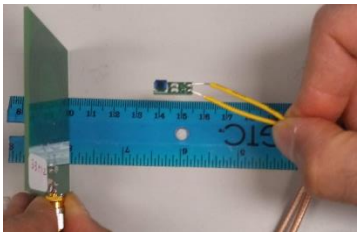


Figure 46: Measurement setup.

Some other variations of the proppant chips were made as shown in Figures 7 and 8. The functionality of previously fabricated miniaturized 4-layer PCBs in which the RX coil was designed using the top two layers, and the TX coil was designed using the top two layers to reduce the form factor (Fig. 47), was successfully verified.

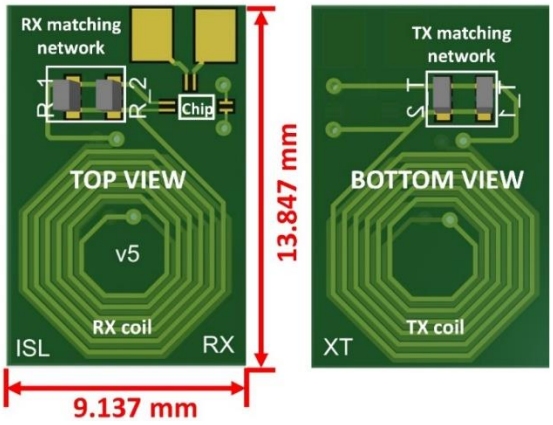


Figure 47: Top and bottom view of the miniaturized 4-layer PCB.

Since the RX and TX coils in this version have the same design as in the old version of the PCB, significant changes in their operating range are not expected. Using this PCB, the microchip could be wirelessly powered at a maximum distance of 5 cm between the localizer and the transmitter (TX) coil with 36 dBm RF power at 40.68 MHz. The 13.56 MHz signal transmitted by the microchip could be received at a maximum distance of around 30 cm by the receiver (RX) coil. Fig. 48 depicts the 13.56 MHz tone having −118.15 dBm power obtained at the maximum operating range possible for wireless powering and transmission. Another version of the PCB having pre-assembled SMD capacitors for the RX and TX matching networks was also fabricated for convenience.

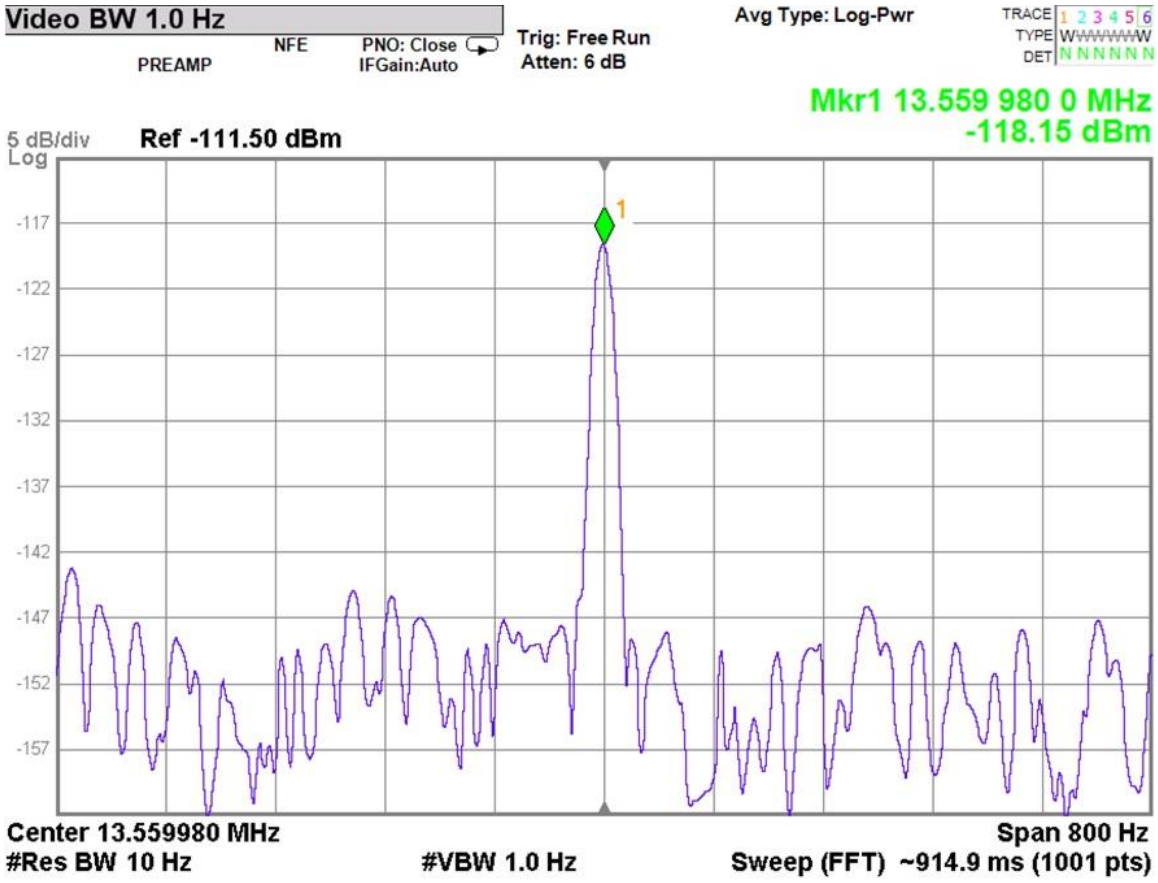


Figure 48: Received signal spectrum at the maximum operating range.

Further miniaturization of the PCBs was investigated using SMD inductors for wireless powering and transmission, which have a lower size than the planar on-PCB coil, as depicted in the next section.

4.5.3.1 Miniaturization of the localizer PCB used for mapping fractures

SMD inductors are currently being investigated for use in place of wireless powering coils to further reduce the form factor of the PCB. To determine the feasibility of the solution, two PCBs have been fabricated. In one of the PCBs (as shown in Fig. 49), only the TX coil is replaced by an SMD inductor. This is a 4-layer PCB in which the RX coil is fabricated using the top two layers.

This is an important intermediate step to study and de-embed the effect of the SMD inductor on the effectiveness and the operating range of the reception of the 13.56 MHz signal transmitted from the microchip.

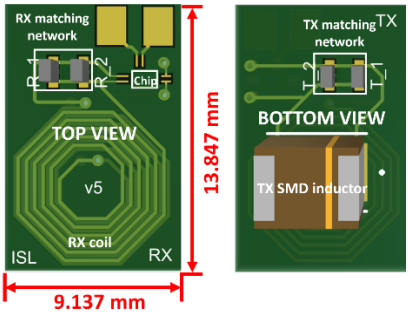


Figure 49: Top and bottom view of the new 4-layer PCB with only TX coil replaced by SMD inductor.

In the second PCB (as shown in Fig. 50), both the RX and TX coils are replaced by SMD inductors. This is now a 2-layer PCB since all the coils are replaced. This PCB demonstrates the final objective, which is to use SMD inductors for both wireless powering and transmission. The use of the SMD inductors also further reduces the form factor of the PCB to 8 mm X 10 mm X 1.6 mm which is lower than the previous version. The vertical dimension (10 mm) can be further reduced by removing the pads (which are included only for the purposes of measurement), which have a length of 3 mm.

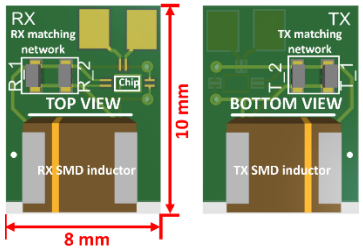


Figure 50: Top and bottom view of the new 2-layer PCB with both TX and RX coils replaced by SMD inductors.

For both PCBs, the largest SMD inductor footprint of 1812 (4532) was chosen as the first step to verify the feasibility of this solution since the quality factors of these inductors at the desired frequencies are comparable to the planar on-PCB coils that were used previously. In future generations, the size of the SMD inductors will be gradually reduced. Both PCBs are currently in fabrication. Once the PCBs are fabricated, future steps will involve the assembly of these SMD inductors and the corresponding matching SMD capacitors for both RX and TX and the measurement of these PCBs with different values and footprints of inductors to have an idea about their effects on wireless powering and transmission.

In some of the fabricated PCB (as shown in Fig. 50), both the RX and TX coils are replaced by SMD inductors having a footprint of 1812 (4532). A 1.8 µH SMD inductor having a typical Q-

factor of 35 at 10 MHz was used for wireless powering and receiving on the RX and TX side respectively. Corresponding SMD capacitors of 8.5 pF and 76.5 pF are used to resonate with both the inductors. Since the axis of the SMD inductors is parallel to the plane of the PCBs, it allows for effective wireless powering of these PCBs inside fractures and does not require a TX coil oriented parallel to the PCB. Using this PCB, the microchip could be wirelessly powered at a maximum distance of 3 cm between the localizer and the transmitter (TX) coil with 36 dBm RF power at 40.68 MHz. The wireless powering distance and orientation are depicted in Fig. 51. The 13.56 MHz signal transmitted by the microchip could be received at a maximum distance of around 20 cm by the receiver (RX) coil. A smaller inductor having a 0805 footprint was also used on the TX side to determine the effects of the size of the SMD inductors. Using this inductor, the signal transmitted by the microchip could be received at a maximum distance of around 3 cm by the RX coil. The reason for the reduction in operating range is currently being investigated. It could potentially be due to the coupling between the two SMD inductors, which have the same orientation on two layers of the PCB.

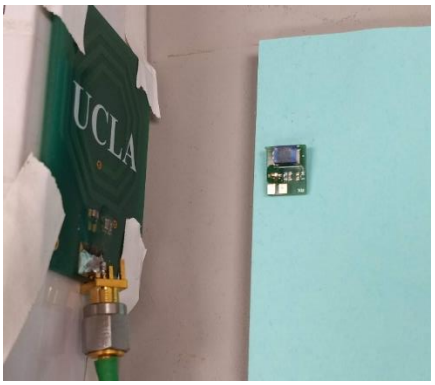


Figure 51: Test setup.

4.5.3.2 Design of TX coils for efficient wireless powering of the microchips

It was observed that the range of operation of the localization system is limited by the maximum separation between the coils for wireless powering. The link efficiency for wireless powering of the microchips depends on the coupling coefficient between the TX coil and localizer coil and the product of the quality factor of these two coils ($\eta \propto k^2 Q_1 Q_2$). Fig. 52 shows the coupling coefficient vs TX coil diameter and the distance between TX coil and localizer for an 8 mm diameter localizer coil. From Fig. 52, it can be observed that the coupling coefficient does not increase significantly for TX coil diameters above 10 cm. Therefore, single-layer TX coil PCBs having dimensions of 5 cm X 5 cm and 10 cm X 10 cm and having a high Q factor were designed to improve the range of wireless powering of the microchips. The coils (shown in Figs. 53 and 54 respectively) were designed using the Webench coil designer tool. The smaller TX coil has 3 turns, with a trace width of 2.54 mm and a 1.78 mm separation between the traces, whereas the larger TX coil has 4 turns, with a trace width of 1.78 mm and a 7.62 mm separation between the traces.

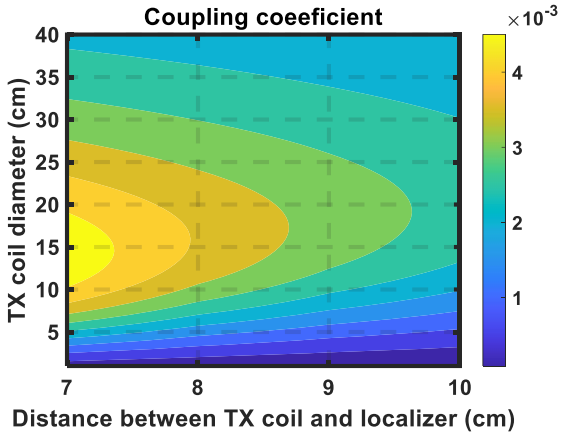


Figure 52: Coupling coefficient vs TX coil diameter and distance between TX coil and localizer.

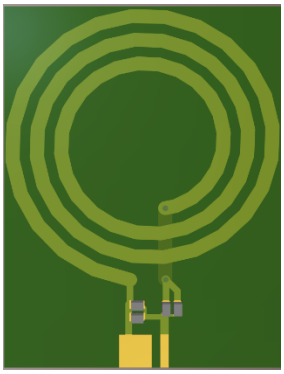


Figure 53: 5 cm X 5 cm TX coil.

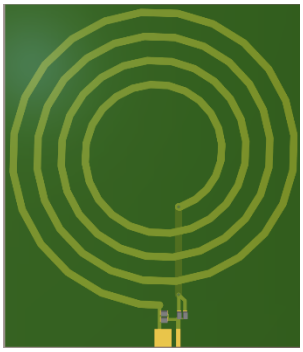


Figure 54:10 cm X 10 cm TX coil.

5. DOWNHOLE TOOL FOR TRANSMITTING POWER AND RECEIVING SIGNAL: DESIGN, DEVELOPMENT, LAB TESTING, AND VERIFICATION FOR THE FIELD TRIAL

5.1 Summary

By the end of the project, the construction and testing of a downhole tool for transmitting power to the chips and receiving their signal were completed. The tool is rated to 100ºC and 8000 psi and has an OD of 3 5/8" making it applicable to a large market within the USA and beyond.

The design/construction was completed in three phases. The first phase was the overall concept and included some mockups of the transmitter/receiver. The second phase included the development of the downhole circuitry to convert incoming power to MHz and transmit that power while simultaneously receiving chip response at a lower MZ value. The circuits were designed to be largely independent of the ultimate antenna, meaning that the circuit manufacturing could be launched concurrently with optimizing Smart Microchips. The third phase was the tool construction for the field trial.

5.2 Initial feasibility study

The Near Field SAS was conceived to map 3-space in a downhole application using high frequency (HF) radio technology and locally dispersed transceiver "chips" designed by UCLA. To determine the feasibility of such an undertaking, it was necessary to 1) understand the capabilities of the UCLA transceiver chips and 2) identify realizable RF technologies capable of extracting sub-100 mV signals in the receive signal chain.

To that end, MicroSilicon built a complete RF prototype system that exploited Liquid Instruments' Moku: Lab instrumentation as a near-term stand-in for a custom design. The overall goals of this phase included identifying likely typical moveout distances for transmission and reception; the relationship between transmitter power and chips excitation; efficiency of the lock-in amplifier for extracting low-level signals in the presence of noise.

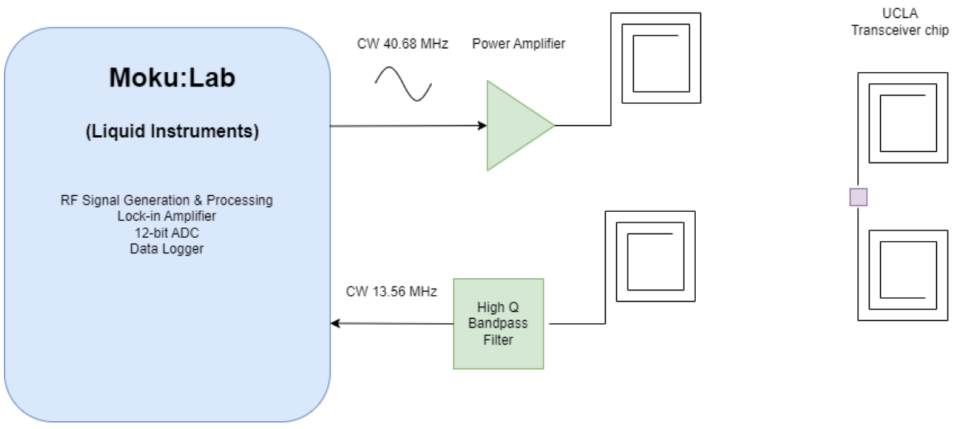


Figure 55: System Context Diagram

The Moku:Lab is an off-the-shelf hardware platform configurable with various lab instruments. The impetus for choosing this device was the integration of a lock-in amplifier (LIA), which is known for its ability to extract signals below the noise floor. The Moku:Lab's LIA contains an adjustable input gain, 12-bit ADC, signal multiplier (mixer), low-pass filter, signal processing, output gain, an auxiliary signal generator, and a data logger. The custom design will require each of these subcomponents, with final implementation being a design-time consideration. The output signal chain contained a generic power amplifier block. While the supplied antennas are not designed for high power, a power stage will be required in the final product. Therefore, some low-power tests were conducted to determine the effectiveness of the current antenna design. On the input side, a high-Q bandpass filter was inserted in the signal chain. Since there is a 1:3 relationship between the transmit and receive frequencies, and the transmitter is expected to be high power, it was necessary to filter the RF input to 1) avoid damaging the receiver and 2) overwhelm the receiver with noise or unwanted signals. The identified, and acquired, band-pass filter was not tested in the circuit because the power levels did not warrant doing so. However, we conducted a VNA test of the KR device to demonstrate that it operated as advertised, and that component was subsequently used inside the downhole tool.

To accurately characterize the prototype system, and therefore extrapolate to a larger system, it was important to isolate the individual capabilities of each system component, namely, the transmitting system, the receiving system, and the signal processing system.

To isolate the transmitting and receiving systems, a simple fixture was constructed to ensure repeatability. The fixture included transmitter antenna frames, receiving antenna frames, and a DSLR (camera) slider instrumented with an adhesive tape measure.

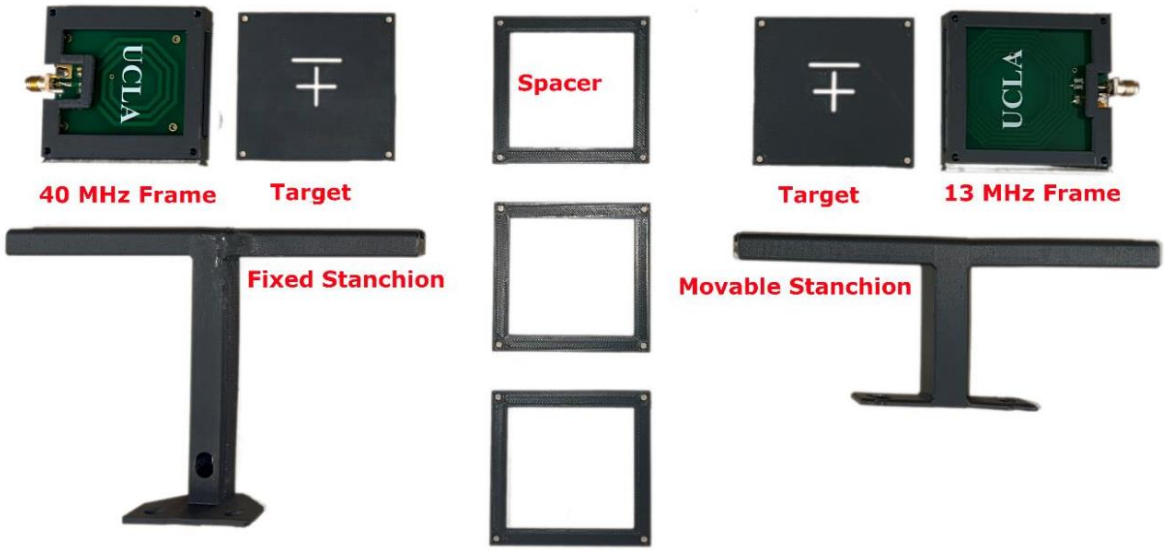


Figure 56: Antenna Fixturing

There were two sides to the test fixture – transmit and receive. Each side was designed to accommodate the specific antenna size (40 MHz or 13 MHz) and are interchangeable on the

stanchions. The stanchions were designed to fit on the DLSR slider and to hold the antenna fixtures while ensuring that the “targets” are aligned regardless of fixture orientation. The fixed stanchion was affixed to the slider frame, while the moving stanchion was affixed to the DSLR based on the slider frame. The spacers provide a gap from the target area to the physical antenna face to optimize the chip/antenna interface. Observations during testing indicated that a chip too close to the receiving antenna severely degraded the received signal power. The frame dimensions were 6 x 6 cm for the individual antenna.

Figure 57: Antenna frames mounted on the slider

The chip on/off characterizing configuration consisted of the 40 MHz TX antenna in the fixed stanchion, the 13.56 RX antenna in the movable stanchion, and the chip taped to the target interface of the RX antenna. The slider was used to change the position of the chip relative to the transmitter while monitoring the microchip's on/off state. The chip on/off state was inferred by the receiver signal.

Figure 58: Receiver testing configuration

The receiver testing configuration had the 40 MHz TX antenna in the fixed stanchion, the 13.56 MHz RX antenna in the movable stanchion, and the chip taped to the target interface of the TX antenna. The slider was used to change the position of the receiver relative to the chip while monitoring the output of the lock-in-amplifier.

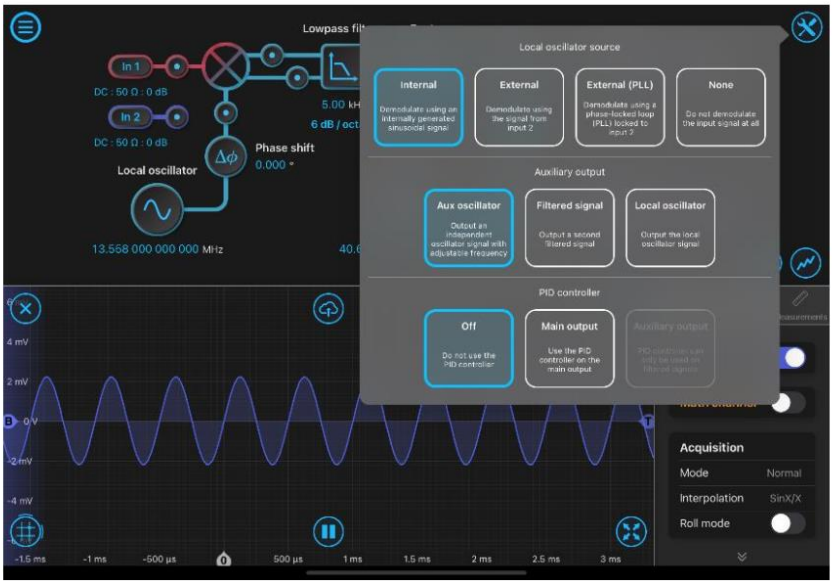


Figure 59: Moku-Lab testing configuration

On the instrumentation side, the Moku:Lab's was configured analogously to the expected custom system design. The general LIA configuration involved using a local oscillator (synchronized to the auxiliary oscillator) to mix with the input signal and the auxiliary oscillator that drove the transmitter signal chain.

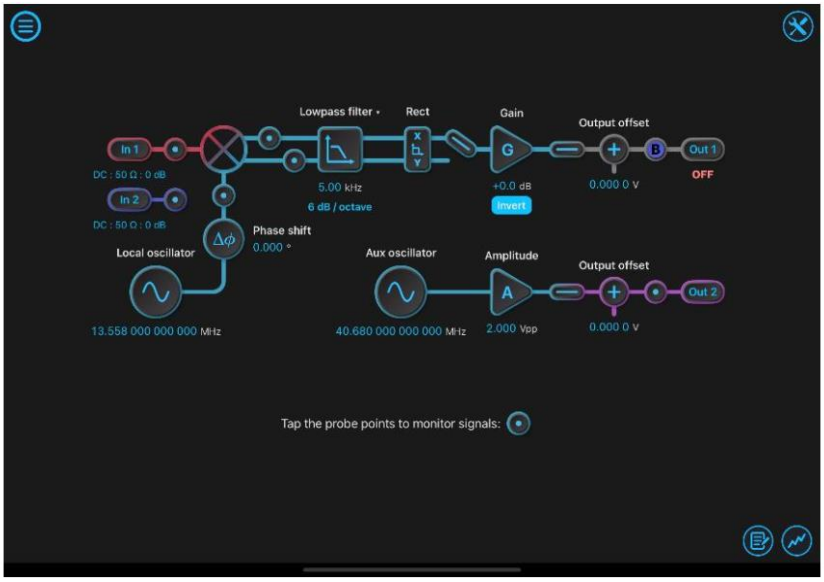


Figure 60: Lock-In-Amplifier configuration

The key LIA configuration parameters were: Auxiliary Oscillator frequency: 40.68 MHz, Local Oscillator (LO) frequency: 13.558 MHz Low-pass filter (LPF) corner frequency: 5 kHz Low-pass filter slope: 6 dB/octave (first order filter) System “output”: Rectangular.

The LO frequency was set to 13.558 MHz, 2 kHz from the expected 13.56 MHz response. The purpose was to demonstrate that the received signal can be “mixed-down” into the audio range (or any range) whereby the final product can utilize high fidelity off-the-shelf ADCs in the order of 20 -24 bits. The LPF filters 1 order and higher-order byproducts of the mixing process.

The purpose of the receiver moveout test was to measure how far the receiving antenna can be from the chip while the LIA continues to discriminate. For this test, the chip was taped to the transmitter target interface on the fixed stanchion. The receiving antenna was affixed to the movable stanchion.

At the minimum distance, the signal amplitude was 5.15 mV pp. At 6.8 cm the 2 kHz signal was discernable on the scope around 67 μV pp and the frequency was locked. As the moveout increased, the sinewave degraded into an increasingly noisy square wave until at approximately 26 cm, the 2 kHz signal was no longer detected. The signal floor appeared to be approximately 64 μV pp.

Using the Spectrum analyzer, a similar test was conducted to get a feel for the signal level. At 8 mm distance, the signal level from the chip is -37 dBm; at 7 cm, the signal level is -103 dBm and at 33 cm, the signal level hits the noise floor of the SA at -103 dBm.

In an attempt to see the effect of two active chips, two chips were taped, one on top of the other, to the target interface on the transmitter side. Unfortunately, the proximity of the two chips prevented either chip from working.

Directionality tests were also conducted and the results demonstrated that the beam width of the PCB planar antennas was very narrow. If the receiver or chips are misaligned, even as little as 1-2 cm, the system will not work.

The purpose of the transceiver reach test was to objectively determine how far the transceiver chip can transmit once energized. There were three tests in this sequence. The first test was a 10-mW transmission test, the second test a 2W test, and the third test a graduated power test.

The chip was affixed to the receiving antenna's target interface and progressively moved away until the chip turned off. The 10 mW transmission power is an estimate of the RMS output of the Moku:Lab Auxiliary oscillator into a nominal 50-ohm load. The ultimate tool design would enable 100W. At a 10mW power level, the chip would abruptly shut off at approximately 2 cm moveout from the transmitter.

For the 2W test, the Akozon RF power amplifier was inserted into the transmit signal chain with 500 mV pp output. At this power level, the chip would abruptly turn off at about 5.4 cm demonstrating that more transmission energy can improve the system performance.

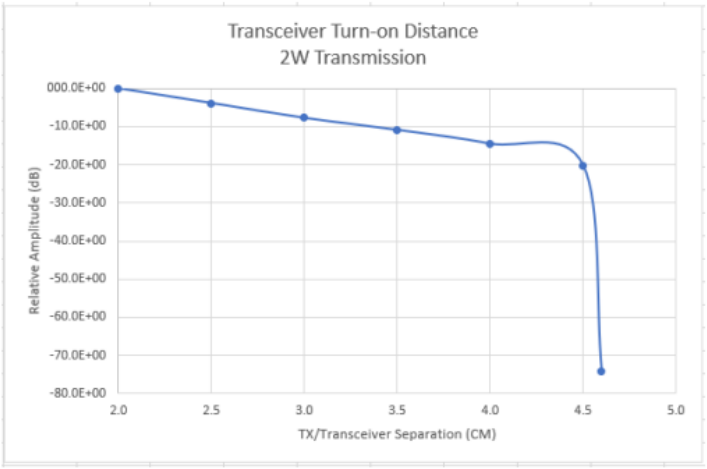


Figure 61: Dramatic drop-off at a fixed distance, this at 2W

The graduated power test was an attempt to extend the transmission reach by slowly increasing the output power of a 100W transmitter, from near 0 W to 100 W.

To facilitate this test, an external signal generator was used. The MRF101AN 100W reference appliance was set up with a 2.5 V bias and the signal generator input was increased to achieve the power gain. For this test, a VSWR/Power meter was inserted in the signal chain. The MRF101AN device was then subsequently also chosen for the downhole tool.

Prior to beginning the system test, a 35 W 50-ohm dummy load was installed as the load to ensure that the VSWR meter accurately measured VSWR. Regardless of the power level, the VSWR reading was close to 1. The results of the test were somewhat disappointing, however, as the performance of the prototype transmitting antenna degraded significantly as power increased.

Table 6: Initial Test Results

Signal Generator Voltage (pp)	Power Meter (W)	Moveout (cm)	SWR Reading
1	0.14	3	1.0
2	0.91	3.8	1.51
2.5	1.52	4.9	2.6
3	2.36	5.3	2.8
4	2.63	5.9	3.2
5	5.54	6.5	3.91
6	10.42	7.0	4.49
7	16.48	7.3	6.21
8	24	7.7	71.2

Technical discussions with Liquid Instruments (makers of Moku:Lab) revealed that the LIA could discriminate signals below the noise floor of the ADC by adding white noise. Further investigation by Liquid Instruments demonstrated this to be true. An explanation offered as the reasoning behind the result is that white noise summed with the signal can bring the amplitude above the noise floor thereby making it detectable. Taking this concept to its next logical step, a test was conducted whereby white noise was injected into the signal chain via a Mini-Circuits combiner/splitter, model Z99SC-62-S+.

For the white noise generator, a Moku:Go was configured as an arbitrary waveform generator outputting a Gaussian waveform with an initial frequency range of 1 MHz. The results showed that the signal was clearly noisier with reduced amplitude, likely due to the combiner.

Testing the moveout, the RX antenna could only be moved to about 21 cm before the LIA lost the 2 kHz IF signal. This is compared to about 26 cm when direct connections are used. Also, changing the frequency of the white noise generator neither improved nor detracted from the results.

It appeared that adding white noise, at least in this rudimentary way, did not add any value and was dropped from subsequent development.

We performed a Zero Hz IF test to learn whether using an identical LO/input signal frequency would improve the performance of the LIA. The only setup change was to set the LIA local oscillator to 13.56 MHz. Results showed that the DC level hits the minimum threshold around 62 uV DC. This minimum was reached at 7.5 cm moveout, nowhere near the capability when using the low frequency IF.

We performed another of this test to learn how far the TX antenna transmits to another 40.68 MHz antenna.

For this test, antennas in both fixtures were 40.68 MHz antennas. The Spectrum Analyzer was used to record a few points along the slider path. As the moveout continued, the signal level dropped in a somewhat linear fashion until about 33 cm out, where it was at a minimum value of -90 dBm. From 33 cm to 67 cm, the signal level seemed to increase back up to -72 dBm, again in a linear fashion.

A series of tests were conducted, and data was collected for moveout distances from 5 to 25 cm in 5 cm increments. Two additional points were added 1 cm and 26 cm to represent the closest and farthest points in the test respectively.

The Moku:Lab Data Logger was configured for 10 k Sa/s and the LIA with a 5 kHz low pass filter to minimize frequency foldback due to Nyquist frequency. The transmitter was set to a nominal 10 mW output. The data were imported into Excel and processed with 4096 sample FFT for each moveout distance. The figure below shows the “baseline” chart at 1 cm moveout.

As expected, the 2 kHz IF frequency was well above the noise floor at approximately 5.4 mV RMS. What was surprising, however, was the 2nd order mixer artifact of 4 kHz, shown below the

fundamental chart. Throughout the test, the 4 kHz signal continued to be above the noise until after the 22 cm moveout, long after the 2 kHz signal was buried in the noise, between 18 and 19 cm.

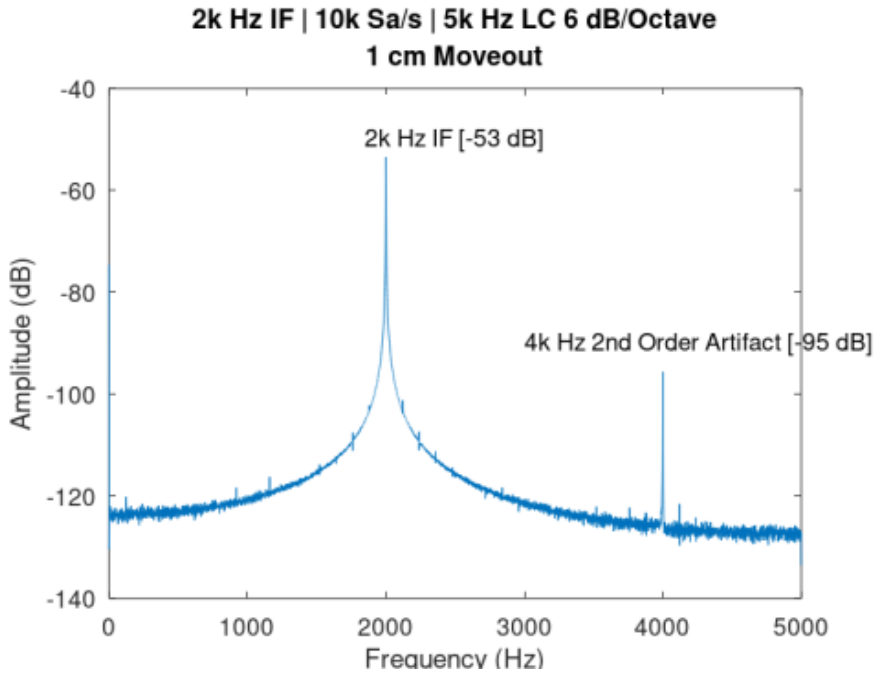


Figure 62: The "baseline" chart at 1 cm moveout.

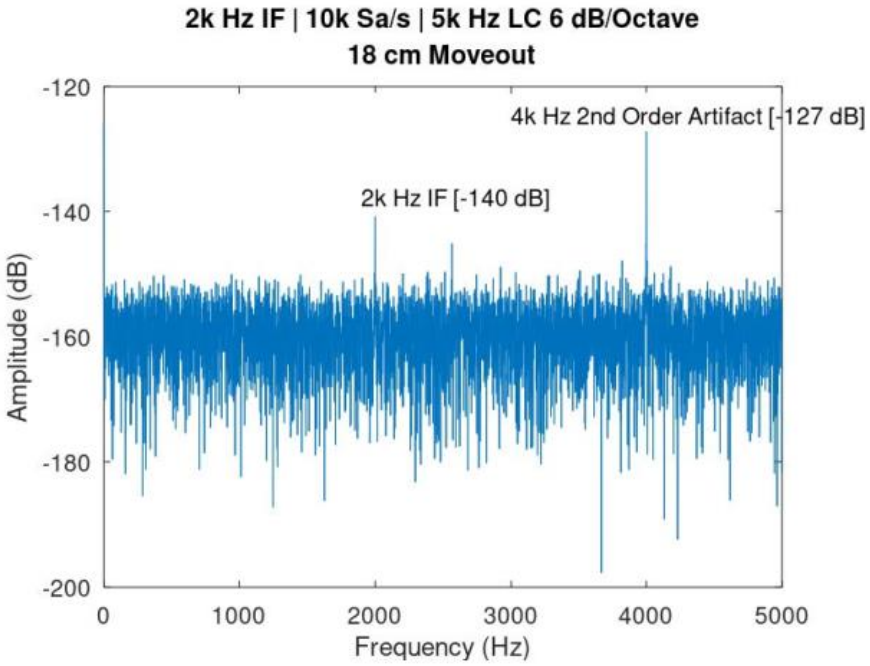


Figure 63: Higher-order artifacts are visible even as the original chip signal degrades with distance

Those similar higher-order artifacts will be noted in the ultimate field data.

Various attempts to excite multiple transceivers were made. For example, taping one transceiver on top of another, taping one transceiver to the transmitter target and the other to the receiver target, and placing one transceiver within excitation distance on either side of the transmitter with another receiving antenna on the same side. All attempts failed due to interference either from the transceivers being too close together or having the additional receiving antenna too close to the working transmitter/receiver pair.

Since the transmission power levels were well below that required to insert the bandpass filter into the receiver signal chain, it was prudent to evaluate the filter for it's potential in the implementation project.

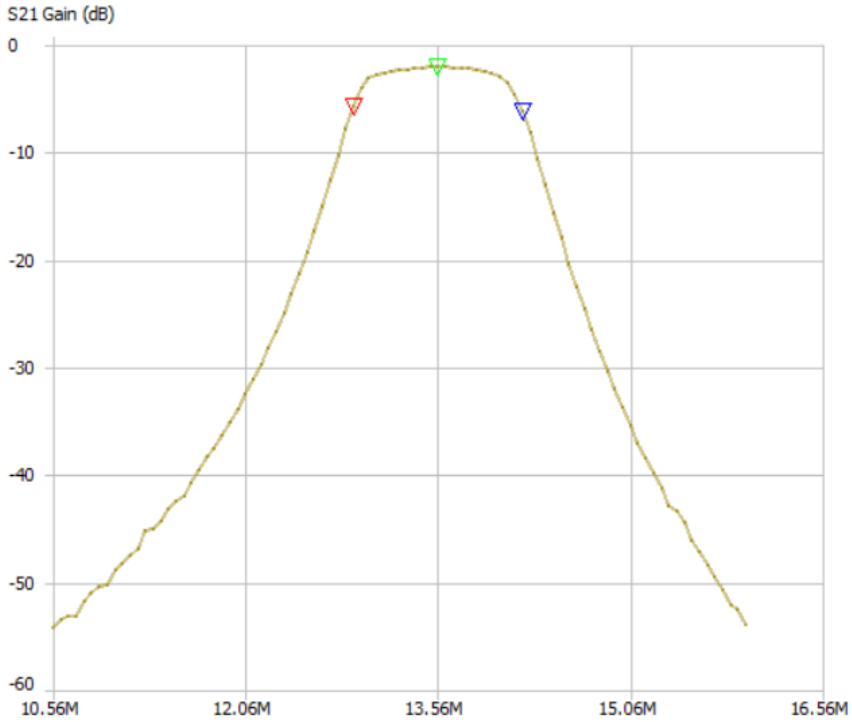


Figure 64: A robust 13.56 filter was identified (and subsequently used in the downhole tool).

The upper and lower markers are at approximately 13.56H Hz +/- 1.28M/2 Hz and are -3 dB down from the peak. The center marker is 13.56M Hz. The data shows that the filter appears to work as advertised. These KR filters were ultimately used in the final product deployed.

Some other observations from this initial feasibility phase:

- On multiple occasions, a transceiver chip was damaged by being too close to the transmitter. It appears that too much near-field energy can destroy a transceiver. This raises the possibility that repeat passes of the downhole tool could end up electrically damaging very close chips.

- The planar antennas have a narrow beam width in the orthogonal direction. Performance quickly dropped for any device that deviated from the center of the beam. This is in agreement with the theoretical radiation.
- The prototype transmitter planar antenna from UCLA was not designed for high power. In one instance (2W), excessive exposure to the power burned the FR4 material and in another, the antenna stopped working after being subjected to higher power. It is assumed that one or more of the capacitors are damaged. Also, when applying higher power signals, the antenna performance dropped off significantly.
- Because the transmitter antennas were not effective under higher power conditions and are extremely directional, it is impossible to compare distance vs power for transceiver turn-on tests.
- Setting the mixer local oscillator input to the transceiver frequency (resulting in a LIA DC output) was not an effective recovery method compared to using a small offset, in this case, 2 kHz.
- The chip transceivers have a turn-on "wall," a threshold where the chip abruptly transitions from off to on. Once on, however, they perform consistently.
- The reach of the system is subject to two constraints: 1) turning on chips and 2) receiver sensitivity. Turning on the chips requires effective transmitter and power, both are design-in qualities. In the above experiments, the chips could "reach" tens of cm.
- While conducting the Receiver Moveout test, the graphic on the LIA output indicated that it could discriminate the 2k Hz IF signal (frequency counter) up to 24- or 25 cm. However, the 10k-point (1 s) FFT of anything over 18-19 cm showed the 2k Hz signal to be below the noise floor (of the FFT). It is possible that a longer FFT could lower the noise floor further, although there is a limit to this technique.
- These feasibility results were very repeatable and gave confidence in moving forward with circuit construction.

The goal of the first phase of the project was to evaluate the feasibility of using the transceiver chips to map 3-space and to take Smart Microchips prototype hardware and investigate what improvements would be needed for a commercial downhole product. The results showed that such a system is possible, however, limitations with the supplied antennas prevented a complete characterization of transceiver chip performance. After reviewing these results, and in discussion with the rest of the team, it was agreed to move to the next phase, that of circuit development.

5.3 Circuit development

The structure of the near field receiver is as shown below:

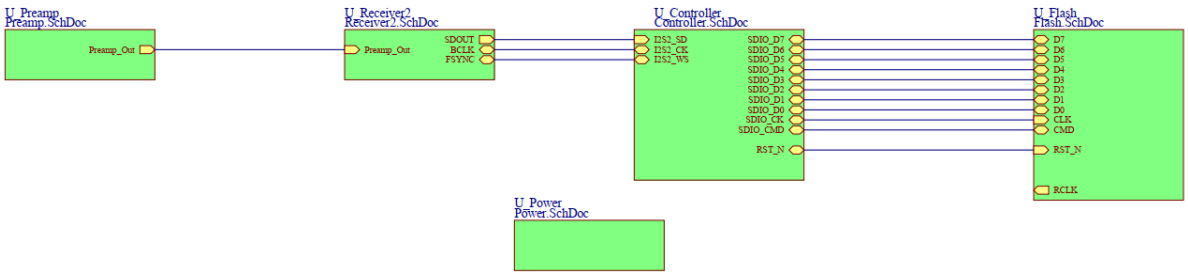


Figure 65: The structure of the near-field receiver

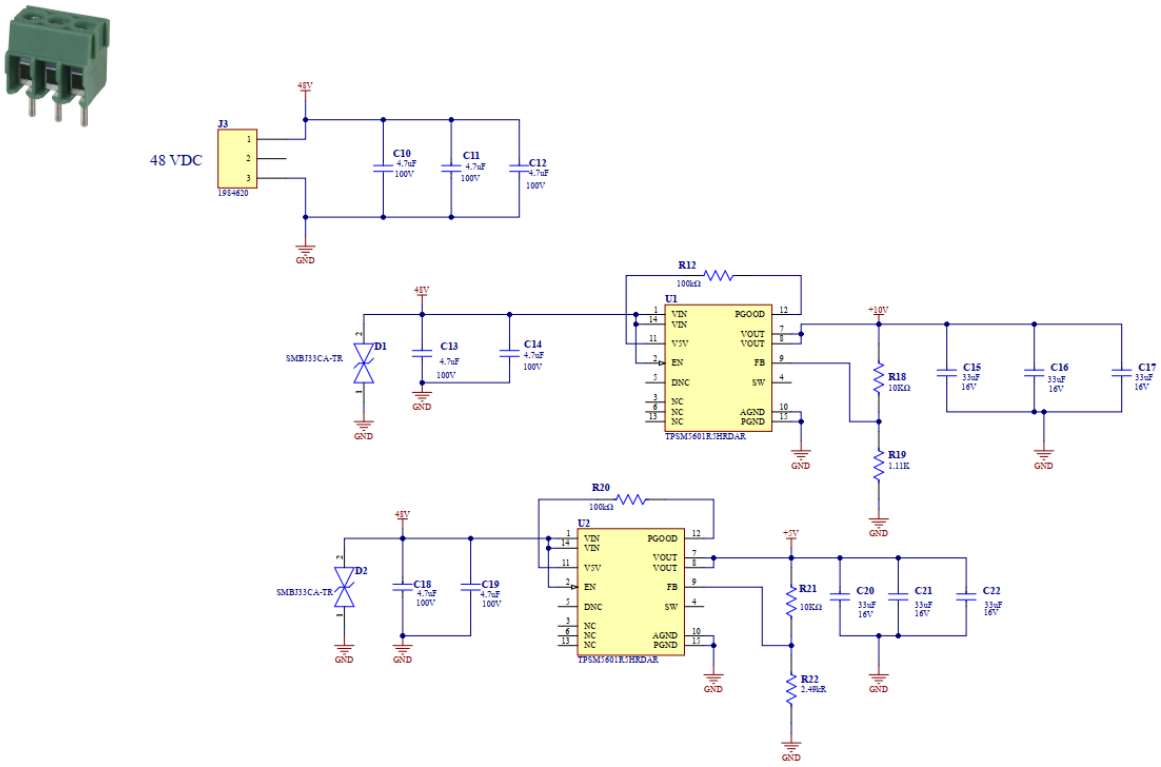


Figure 66: The structure of the near-field receiver components- circuit schematic (1)

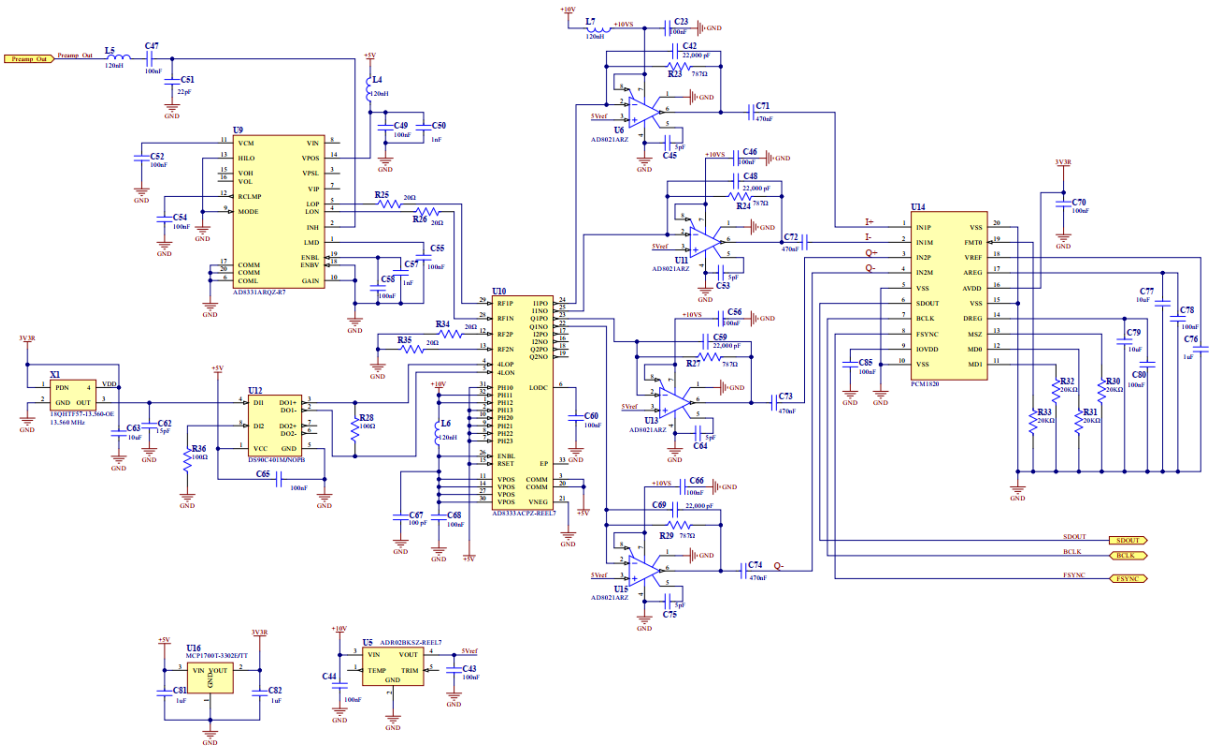


Figure 67: The structure of the near-field receiver components- circuit schematic (2)

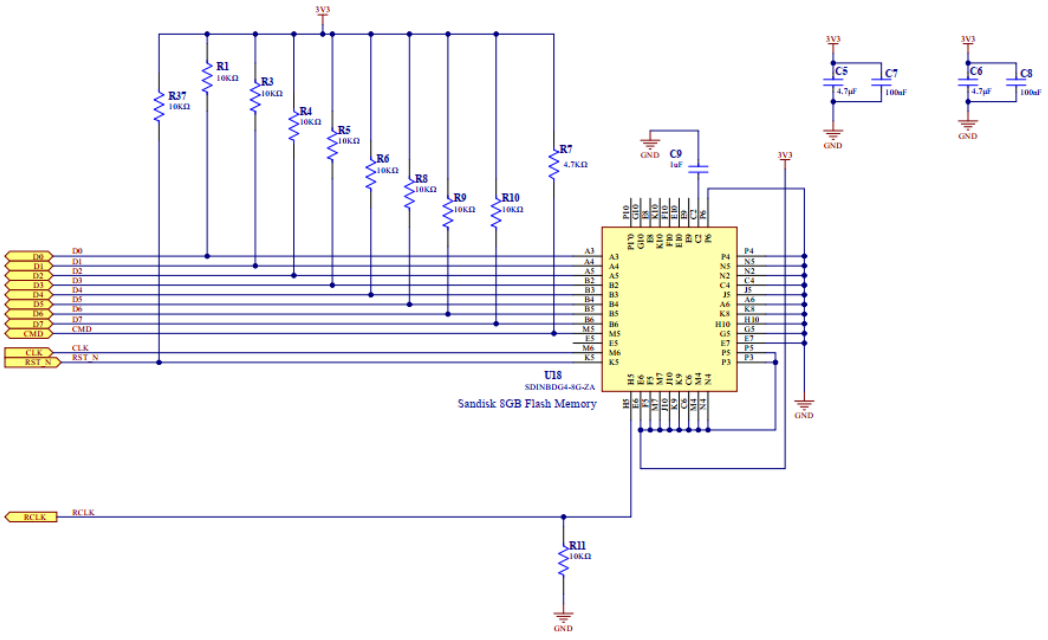


Figure 68: The structure of the near-field receiver components- circuit schematic (3)

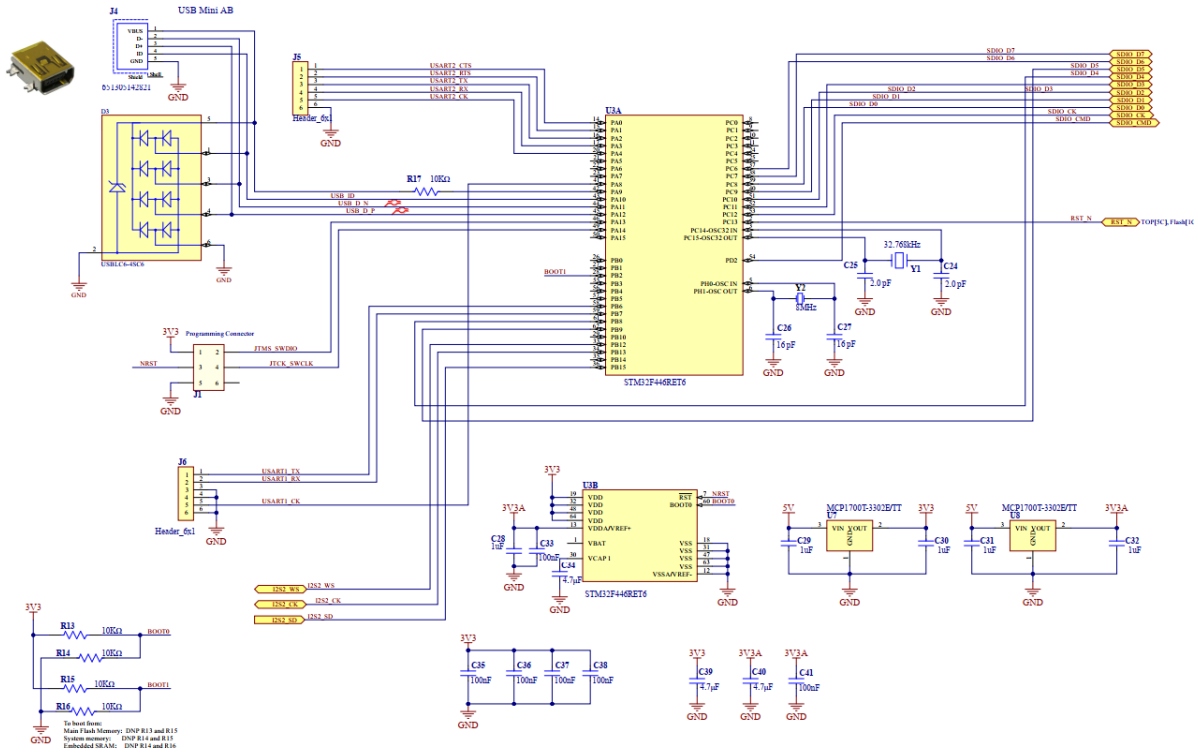


Figure 69: The structure of the near-field receiver components- circuit schematic (4)

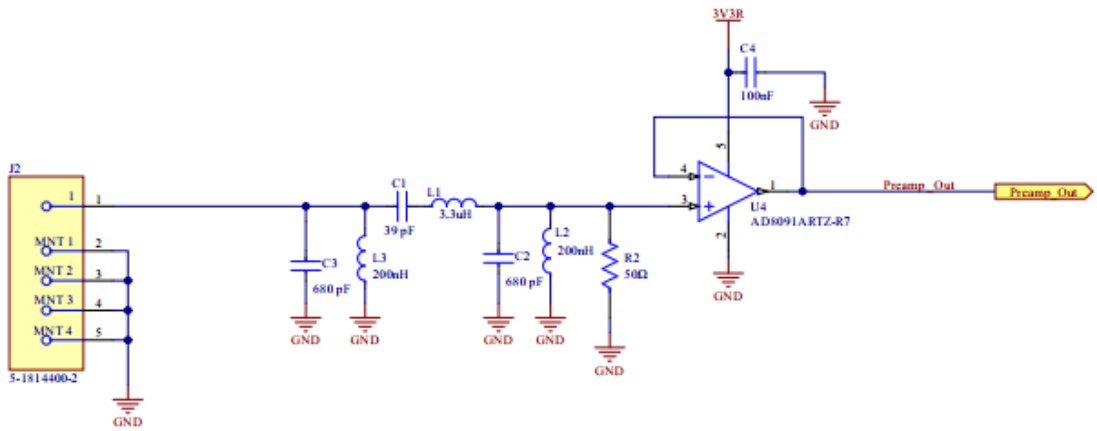


Figure 70: The structure of the near-field receiver components- circuit schematic (5)

Figure 71 shows the structure of the transmitter.

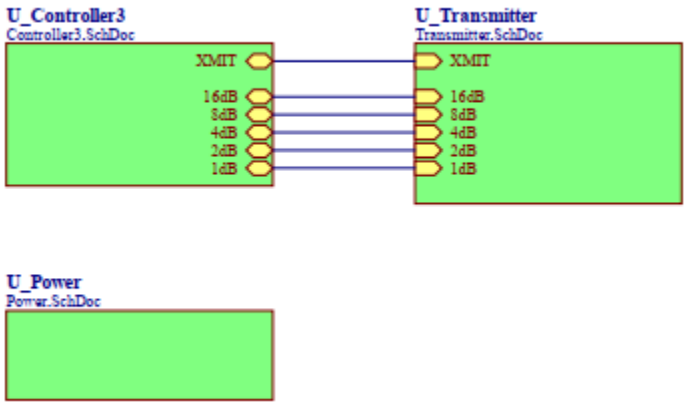


Figure 71: The structure of the transmitter

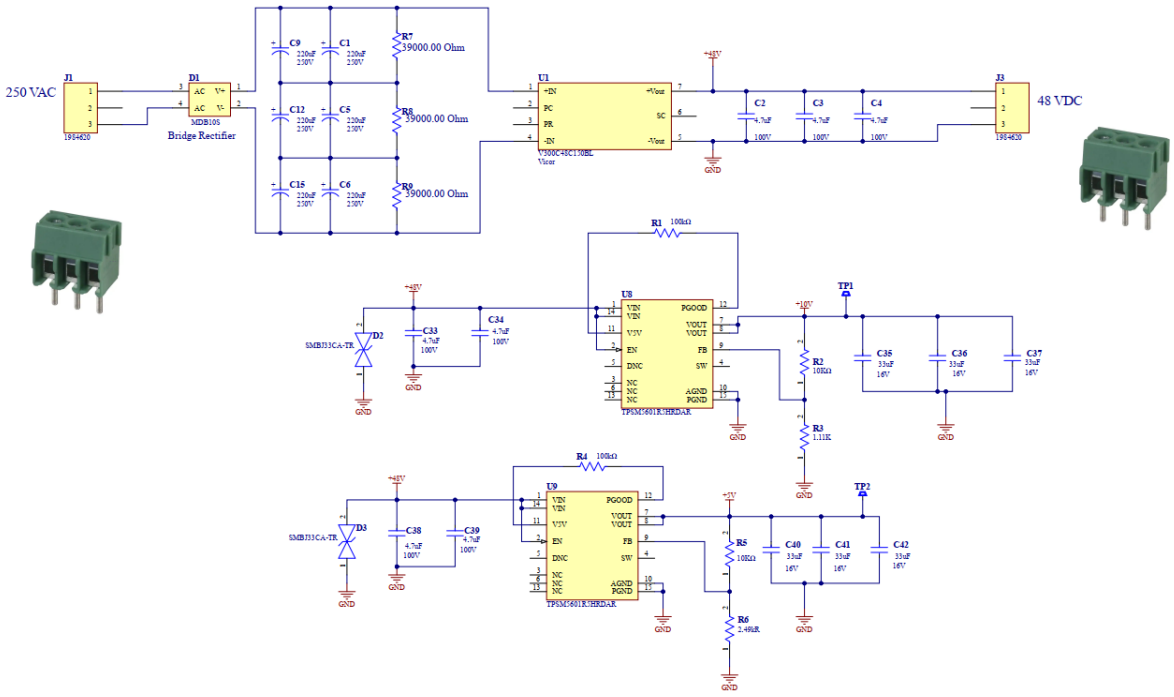


Figure 72: The structure of the near-field transmitter components- circuit schematic (1)

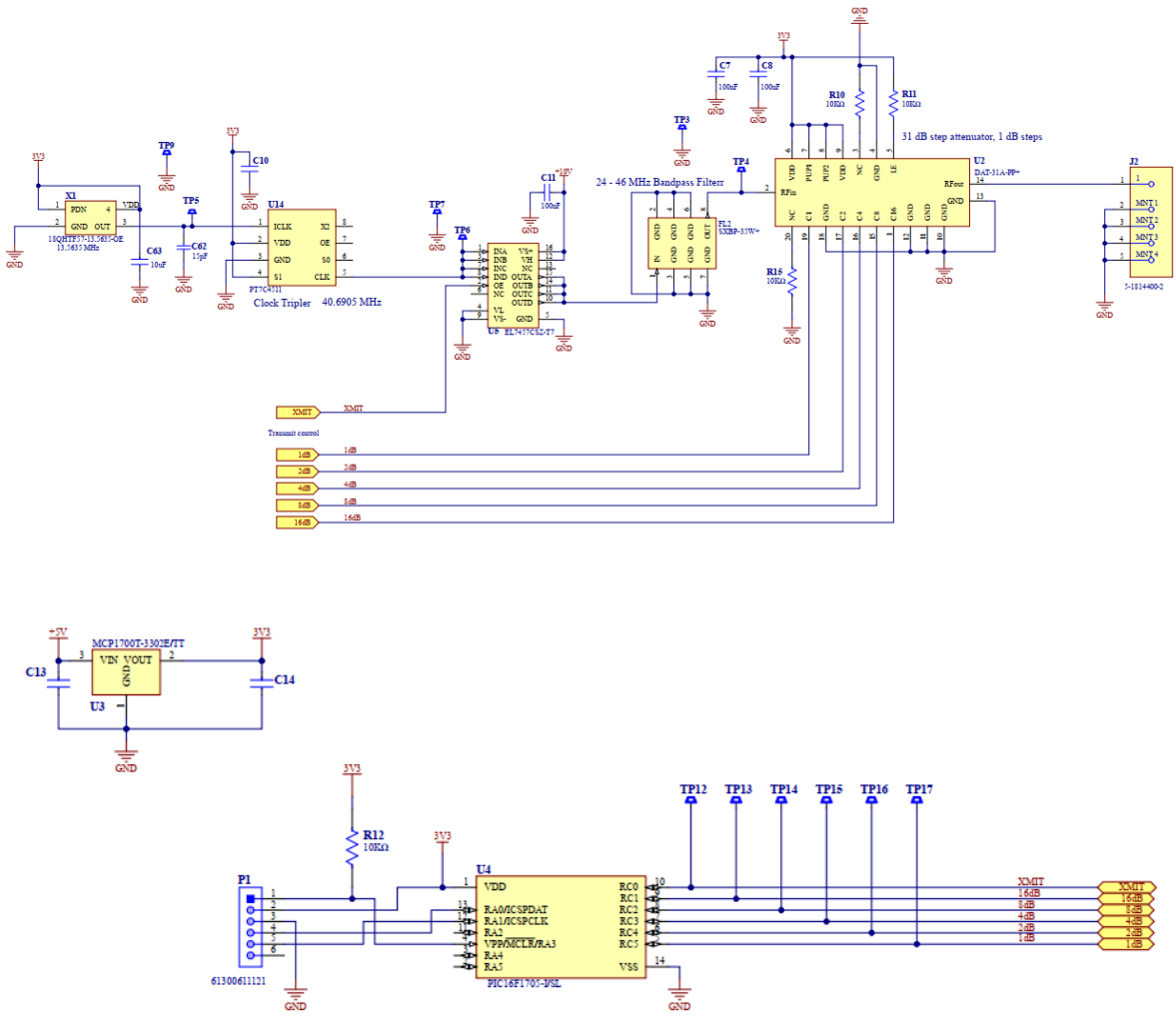


Figure 73: The structure of the near-field transmitter components- circuit schematic (2)

The flowchart for the transmitter is illustrated in Figure 74.

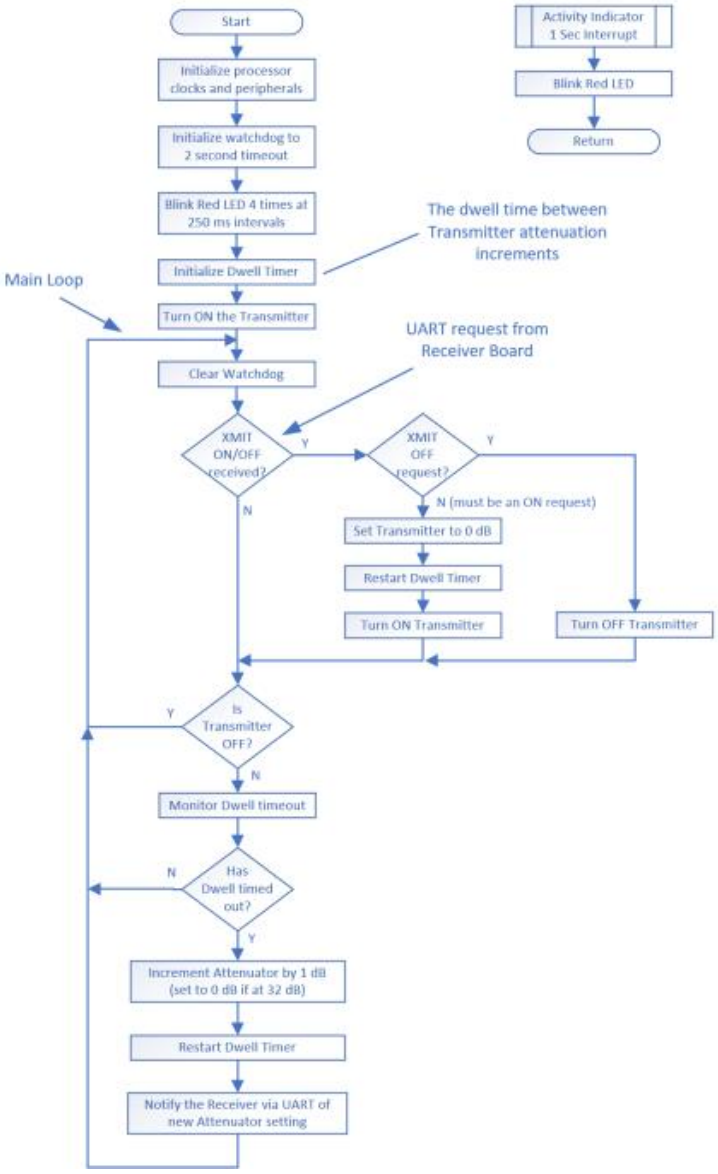


Figure 74: The flowchart for the transmitter

Figure 75 shows the flowchart for the receiver.

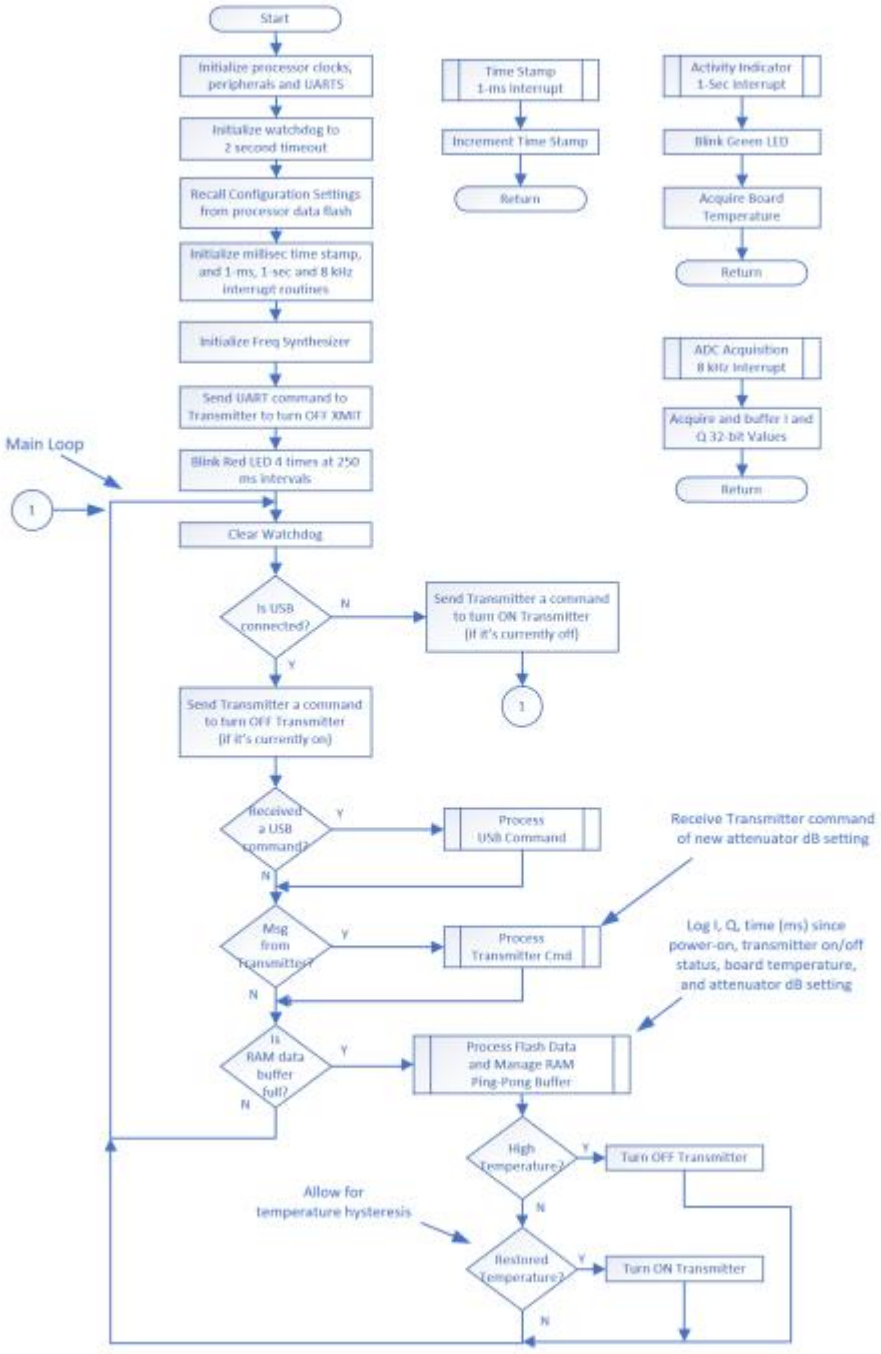


Figure 75: The flowchart for the receiver.

The completed circuit boards are illustrated in Figure 76.

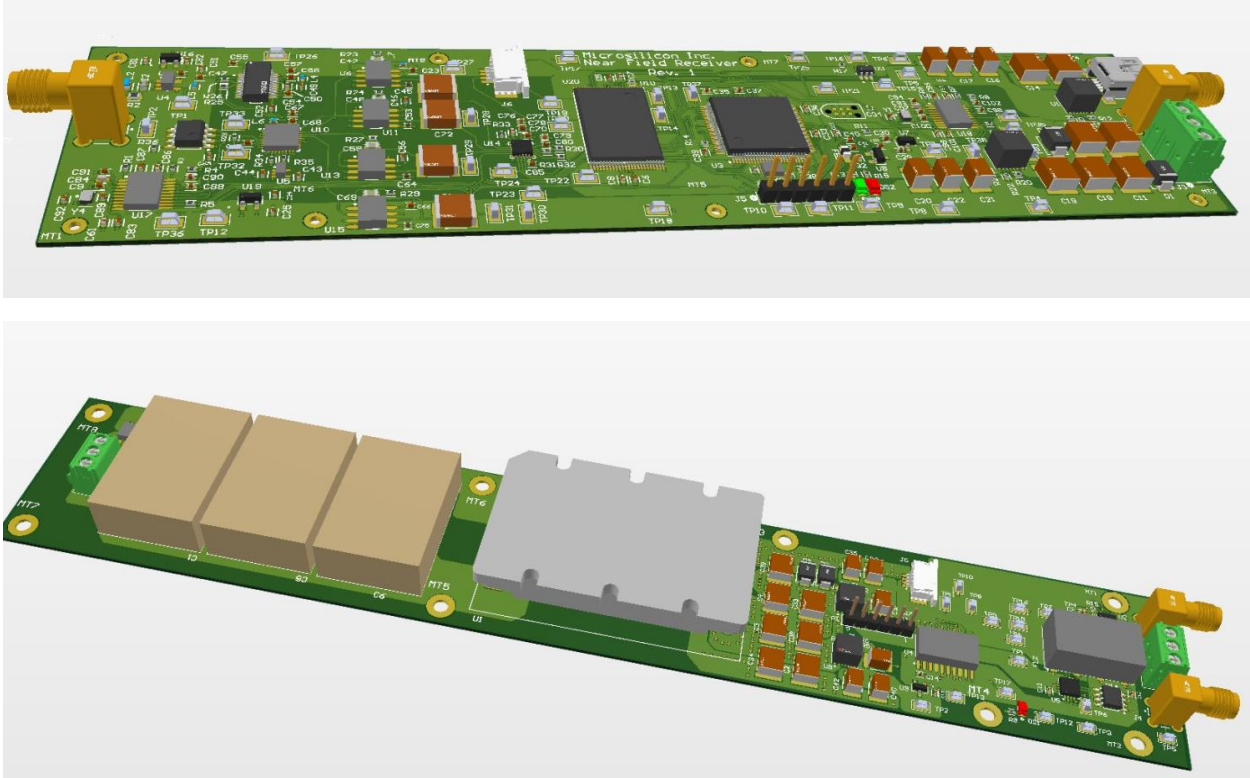


Figure 75: The completed circuit boards designs

Once the circuit boards were built, the components could fit very well into the prototype test configuration designed in the first feasibility phase:

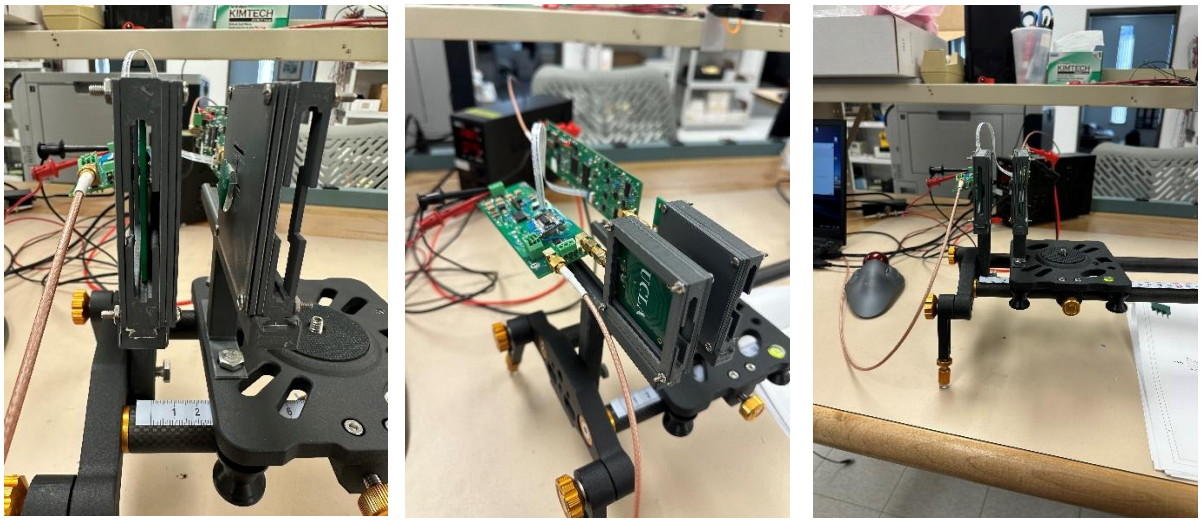


Figure 76: Fit the circuit boards in the prototype test configuration

We also developed a rudimentary GUI to allow us to verify in real time that there was a signal in the desired FFT band:

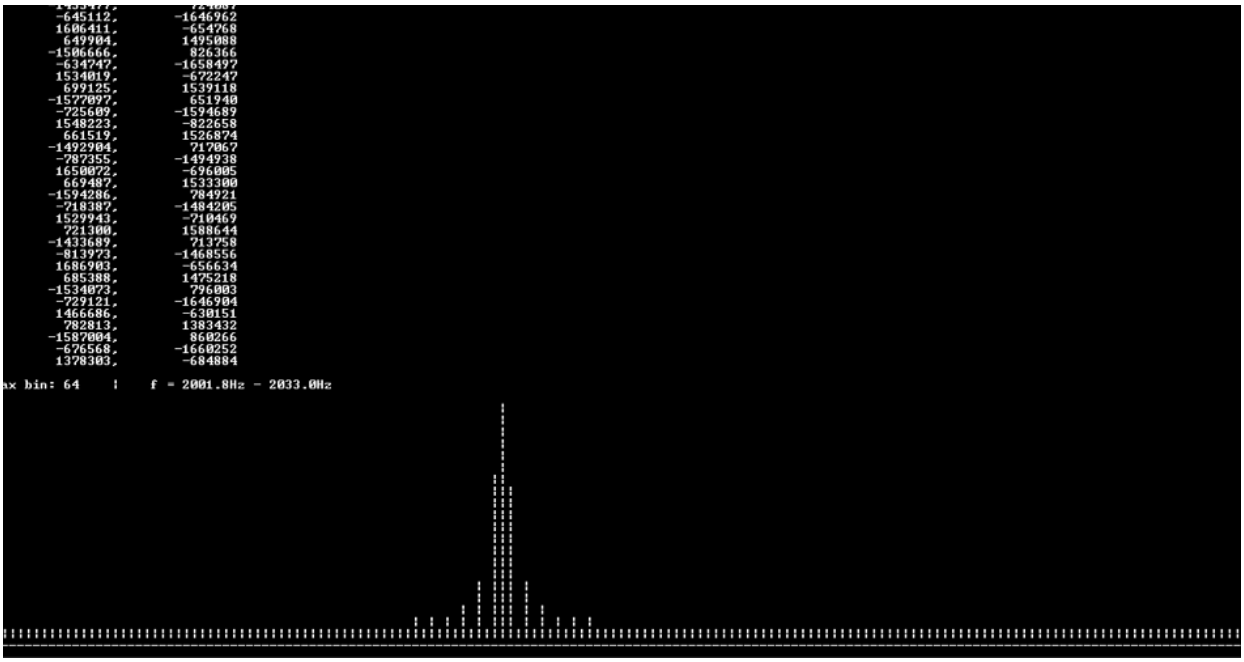


Figure 77: GUI to allow us to verify in real-time that there was a signal in the desired FFT band:

Another decision made during construction was to allow the possibility of both DC and AC power, with DC to be used for the first test (i.e. with batteries) and allowing for AC on subsequent field tests (i.e. with surface power at 240V AC). This reduced the final set of electrical components to the four shown below.

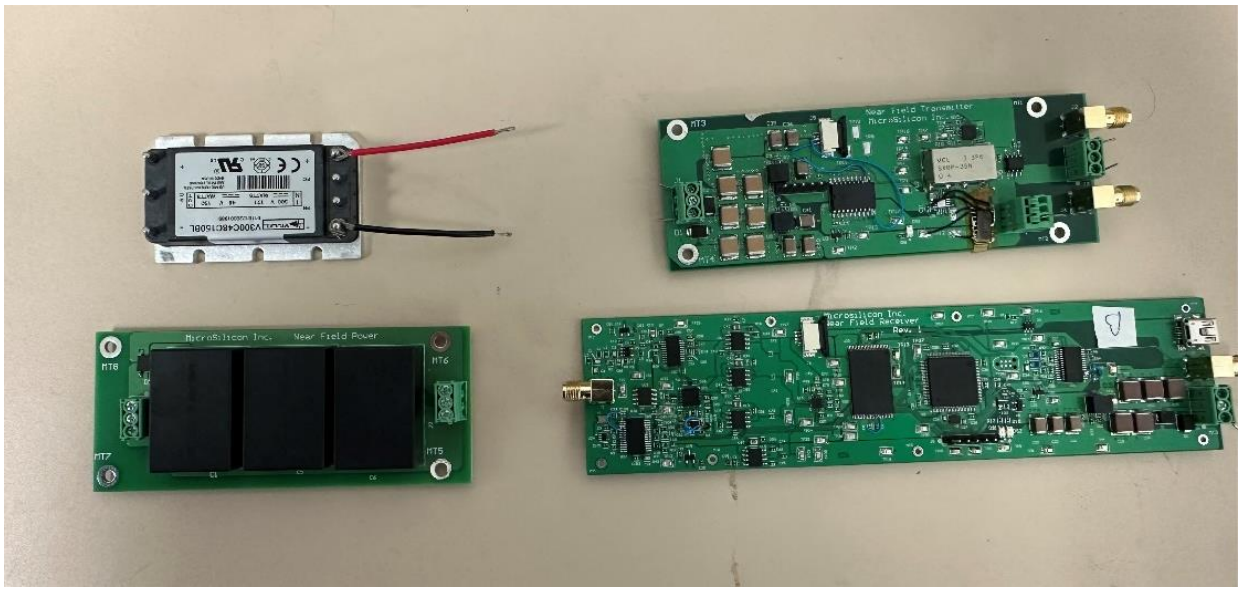


Figure 78: final set of electrical components

5.4 Final Antenna and Downhole Tool Construction for the Field Trial

In close consultation with the team, the design of the antenna was finalized as a 40 MHz transmitter and 13 MHz receiver with a configuration as shown below. In particular, the coupled TX/RX system comprises two coils whose axes are aligned. The transmitter outputs a magnetic field primarily in the direction of that axis, which is also the main sensitivity of the receiver. There is a minimal field generated in the plane of the coil. It is believed that this directionality could prove useful when the tool is deployed downhole: different orientations of the tool will excite different azimuthal planes of the wellbore.

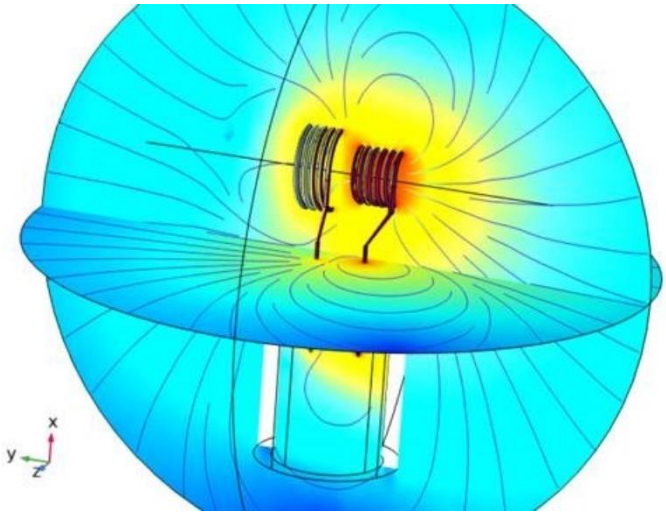


Figure 79: Transmitter outputs a magnetic field

The coils, shown below, are wound of thick, 10 AWG copper so provide minimal resistive losses even at 20 Amps. The coils will be subject to hydrostatic pressure in the wellbore whereas the remaining electronics are inside a pressure housing. This requires a bulkhead, also shown below, to pass the electric field from the high pressure to the low.

Figure 80: Schematic of the designed Coil

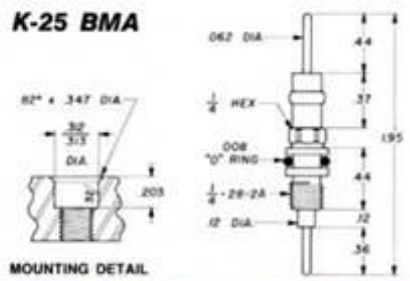
Each termination of the coil has been equipped with a Kemlon K-25 BMA connection, including a rubber “boot” that provides a secondary barrier to pressure. The connectors are rated to 25 kPsi, 16Amps, and 6 kV. The max RF power from the tool is 100 watts.

THREAD CHART (UNF-UNC THREADS)

Thread Designation	UNF / UNC	Threads per Inch	Basic Major Diameter (External Threads)	Basic Minor Diameter (Internal Threads)
0-80	UNF	80	0.060	0.047
1/4-20	UNC	20	0.250	0.196
1/4-28	UNF	28	0.250	0.211
5/16-18	UNC	18	0.313	0.252
5/16-24	UNF	24	0.313	0.267
3/8-16	UNC	16	0.375	0.307
3/8-24	UNF	24	0.375	0.330
7/16-14	UNC	14	0.438	0.360

[redacted]

K-25 BMA



[redacted]

Description	Part Number	Construction	Max. Temp (F)	Max. Pressure (PSIG)	Current (A)	Max. Voltage
K-25-BMA	16-B-1748	polyether	450	20,000	18	3000
	16-B-1537	glass / polyether	500	25,000	7	4000
	*16-B-389	glass / ceramic	500	25,000	7	5000
	16-B-1843	ceramic	500	25,000	6	6000
	16-B-14457	Kemthread/PEK	400	25,000	4	6000
	*16-B-18865	Kemthread/PEK	400	25,000	18	6000

Figure 81: Thread chart and specification

The Kemlon’s attached to the bulkhead are shown below:

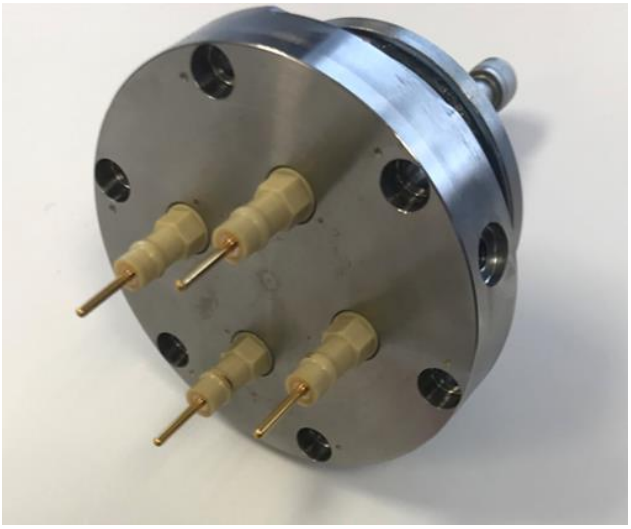


Figure 82: The Kemlon’s attached to the bulkhead

The TX and RX loops are completed inside the pressure housing by attaching to 6kV rated tunable capacitors with a range from 5 pF to 120pF and Q > 500.

Figure 83: The TX and RX loops are completed inside the pressure housing

The antenna coils are highly inductive, but this reactance is canceled by appropriately tuning the capacitors, so the subsequent electronics only see the real part of the antenna impedance, which is less than 1.5 ohms. The transmitter and receiver PCB's are designed around 50 ohm impedance, which creates a requirement for a matching circuit. We investigated the idea of matching with a combination of capacitors as had been done by UCLA on their TX/RX (see figure below)

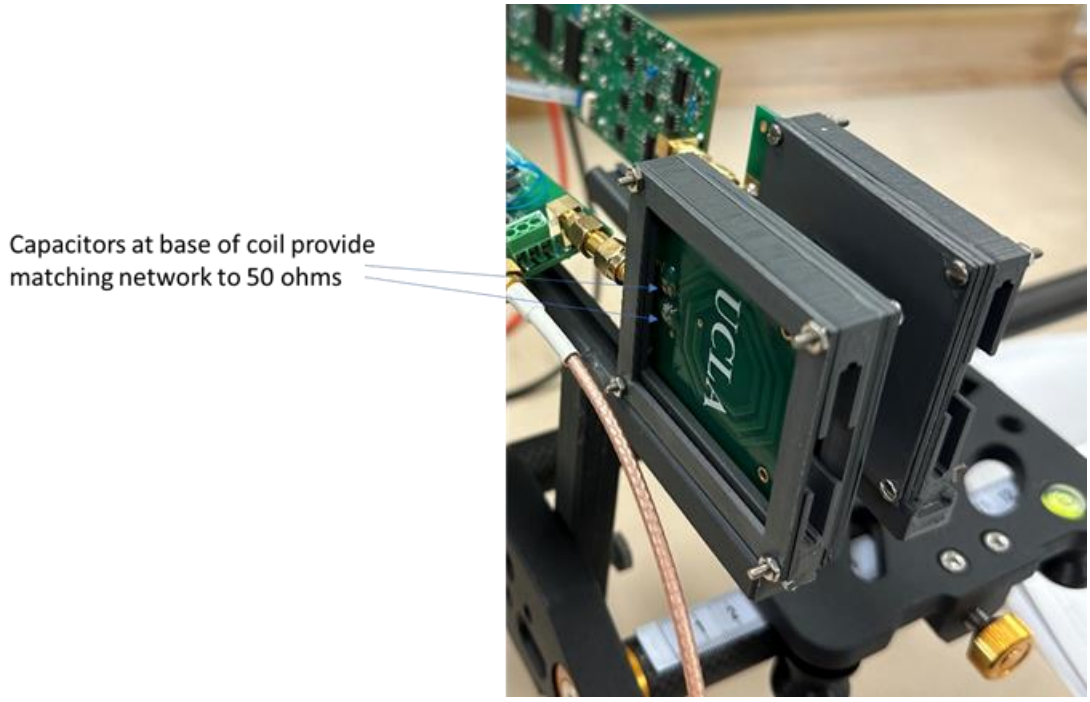


Figure 84: The matching circuit idea testing and demonstration

We were concerned that downhole temperature would cause the capacitance to change, and our analysis showed that even small changes in capacitance would significantly impair the impedance match. So instead, we used a toroid configuration where the primary was the antenna loop and the secondary would have N turns. The impedance scales as N^2 so only a small number of turns will be needed. One key difference between transformers at MHz vs low frequency is that the primary and secondary wires must be wound together (termed “bifilar” winding). The transformer would not work at all if, say, the primary was on the left and the secondary on the right.

The capacitors are toroids mounted on 3D printed “ASA” plastic so that they would be secured in position despite any shock and vibration to the tool in the wellbore. That assembly was mounted onto the bulkhead and then a metal crossover piece was bolted on.

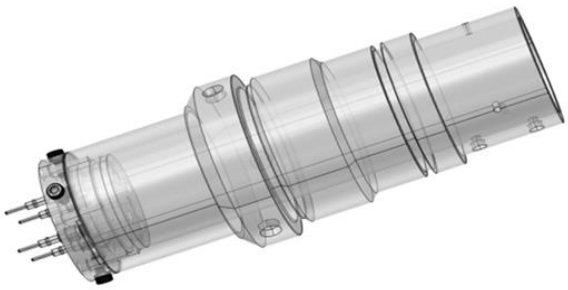


Figure 85: 3D-Printed Capacitor Assembly for Shock and Vibration Resistance in Wellbore Applications

This cross-over provides a crucial link between the inner and outer components of the tool. The bulk of the tool consists of an ~8 ft long, 3 5/8” diameter pressure housing that holds the battery, etc. At the base of the 3 5/8” pipe are threads that can engage with threads, and o-rings on the crossover to provide pressure-tight connection. The interior of the crossover includes connection points to allow attaching the rest of the chassis. The crossover is shown below attached to the bulkhead and antennae.

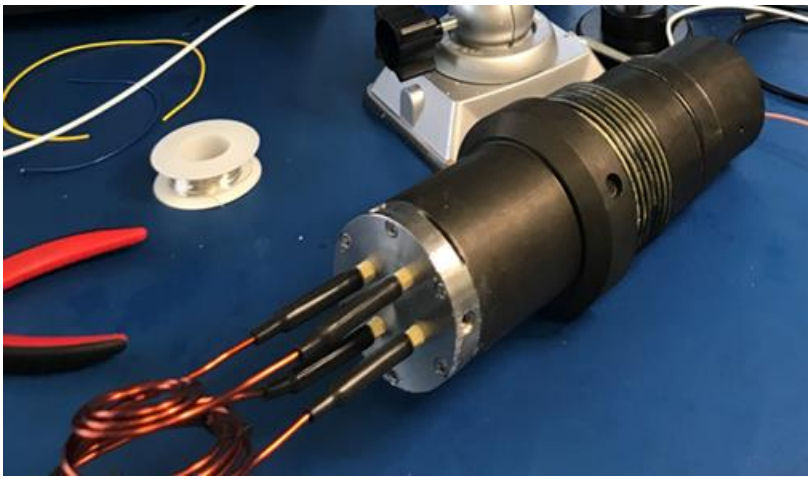


Figure 86: The crossover is attached to the bulkhead and antennae.

The top of the crossover provides a mechanism to add a fiber-glass housing. That housing was then (partially) filled with RTV. The RTV covered the copper. The purpose of the RTV was two-fold: partly to protect against vibration and also to provide the antenna with a dielectric constant more similar to what the tool will see downhole. That way the matching of the antenna would still be valid downhole.

Figure 87: Fiber-glass housing Assembly with RTV Filling for Vibration Protection and Antenna Dielectric Matching

Looking down at the inside of the crossover you can see the toroids and the plate holding the capacitors. The capacitors are beneath the plate but their tuning screws are above the plate so the antenna can be tuned after everything is in place. A long, electrically insulating tuning screwdriver was used to avoid accidental contact with high voltage.

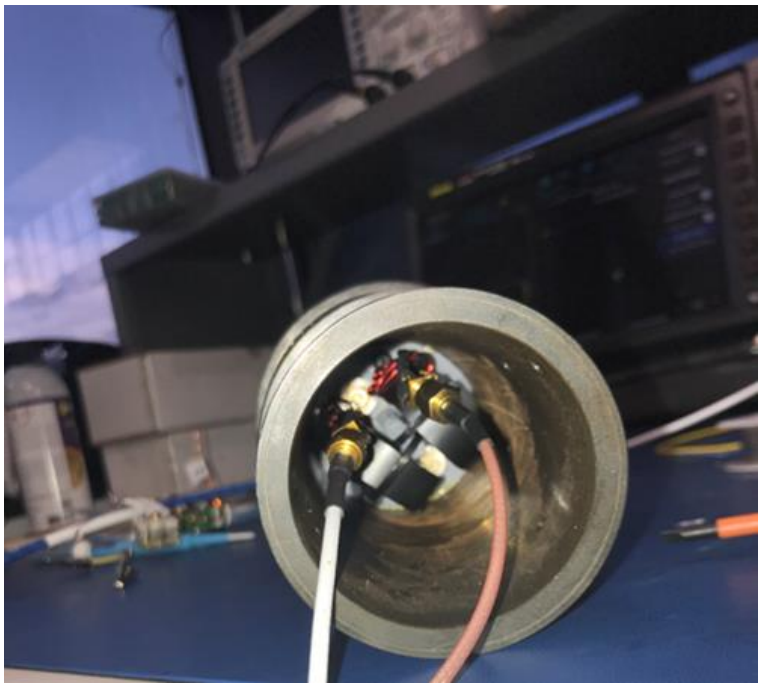


Figure 88: Internal View of the Crossover Assembly Showing Toroids, Capacitor Plate, and Tuning Mechanism

The combination of coils, capacitors, and toroids should present an impedance with zero reactance at the resonant frequency and close to 50 ohms real impedance. We got essentially perfect agreement on the receiver coil (50.04+0.23j)

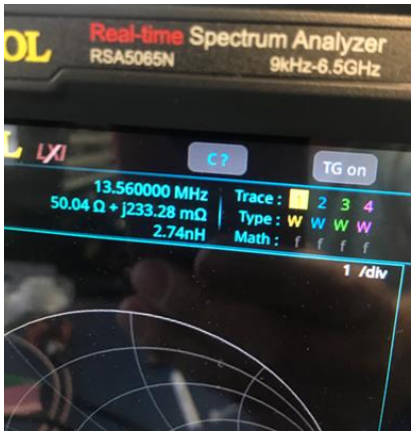


Figure 89: Impedance Measurement of the Receiver Coil at Resonant Frequency Using a Spectrum Analyzer

And fair agreement on the transmitter 61.4 + 1.3j. This transmission impedance was very acceptable because its next component, a MOSFET, could accept VSWR up to 65 and we had plenty of safety margin on the voltage rating of the capacitors.

Figure 90: Transmitter Impedance Measurement and VSWR Tolerance Analysis with MOSFET Integration

The MOSFET had a max power rating of 100W but required a heat sink. The MRFAN101 was commercially available and already attached to an aluminum block. We then mounted that aluminum onto a block of copper (using thermal paste in between)

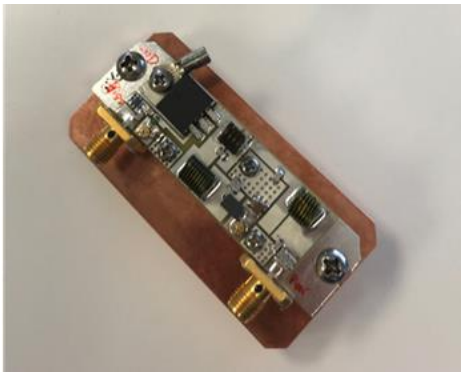


Figure 91: MOSFET Heat Sink Assembly: MRFAN101 Mounted on Aluminum and Copper Blocks for Thermal Management

And then that copper block was bolted to the metal crossover, which itself was going to be in good thermal contact with the 3 5/8” pipe.

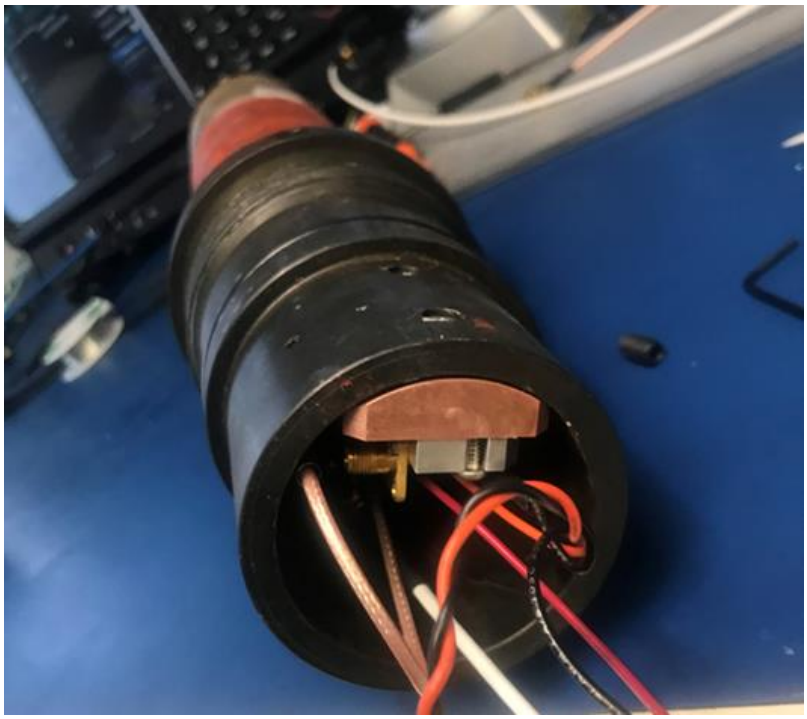


Figure 92: Copper Block Mounted to Metal Crossover for Enhanced Thermal Conductivity

The resulting bandwidth on the receiver was very sharp but when the transmitted was activated measurement circuitry still detected 40MHz, which certainly could not have been coming through the receiver antenna. It was determined to be a common mode, so we added a third toroid and wound the coax around that toroid.

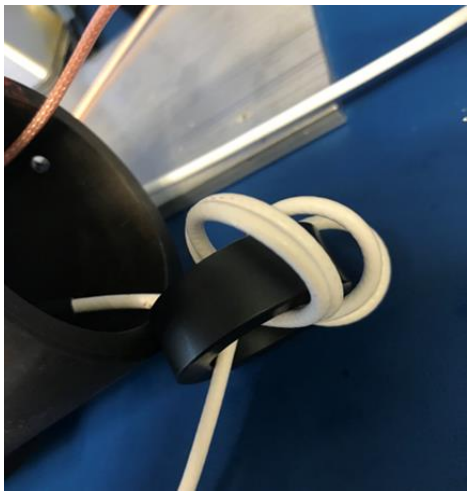


Figure 93: Implementation of a Third Toroid to Mitigate Common Mode Interference in the Receiver

And then finally we added the dedicated bandpass filter that had been identified during the feasibilty phase.

Figure 94: Addition of a 13.56 MHz Bandpass Filter for Signal Optimization

We have previously presented the receiver and transmitter PCB's. These were mounted onto the chassis and the chassis secured to the bulkhead with 1/8" roll pins.

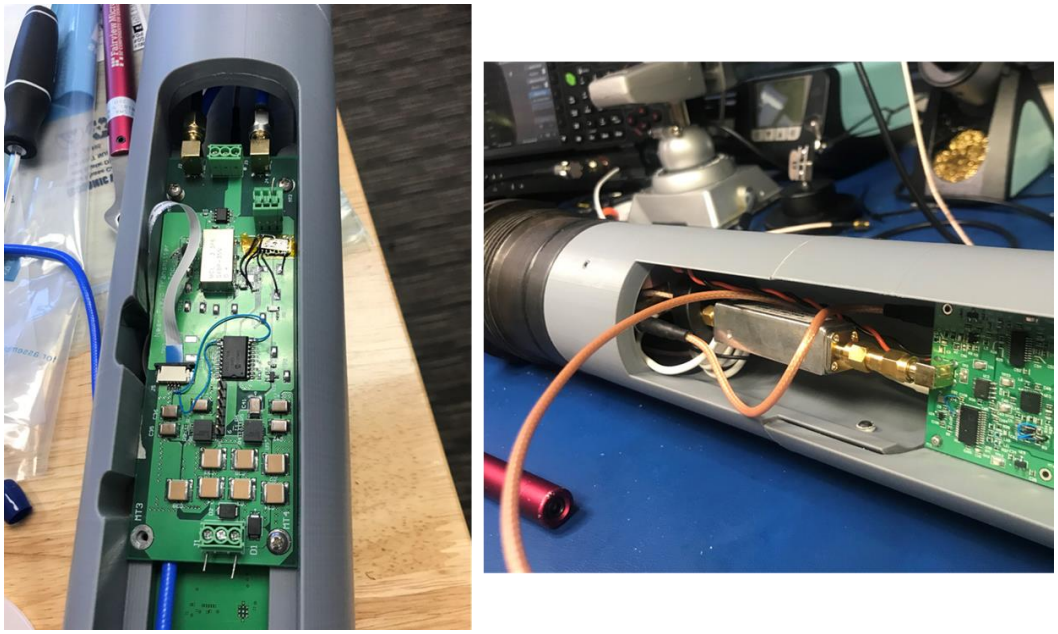


Figure 95: Receiver and Transmitter PCBs Mounted on Chassis and Secured to Bulkhead

The system is powered by a 30V, 29Ah Lithium battery and we placed an on/off switch at the top end of the battery.

Figure 96: Fully Assembled System Powered by a 30V, 29Ah Lithium Battery with On/Off Switch

In addition to the on/off switch, we ran a long USB cable from the receiver PCB to the top of the chassis so that we could download data without pulling the chassis from the 3 5/8” pipe.

Figure 97: Top-End USB Access for Data Retrieval Without Chassis Removal

Lastly, we added an MDM connector to allow communication to a Criterion circuit installed inside the battery that allows us to depassivate the battery before deployment.

Figure 98: Integration of MDM Connector and Criterion Circuit for Battery Depassivation Before Deployment

Finally, the chassis was loaded into the 3 5/8” pipe.

Figure 99: Chassis Installation into the 3 5/8” Pipe

The lower crossover was torqued in place with spanner wrenches.

Figure 100: Lower Crossover Secured with Spanner Wrenches

At the top end, we provide another small sub with threads and o-rings that can be torqued in place and provide the final pressure seal. At the top of that sub is a thread whose profile was provided by EOG for attachment to the rope socket on the slickline unit.

Figure 101: Top-End Subassembly with Threads and O-Rings for Final Pressure Seal and Rope Socket Attachment

And success:

Figure 102: Final downhole tool at the pilot testing site for deployment

6. PHYSICS- INFORMED AND AI-EMPOWERED I-GEO SENSING FRACTURE DIAGNOSTIC SOFTWARE PACKAGE DEVELOPMENT, AN OPEN-SOURCE PYTHON-BASED PACKAGE

6.1. Overview of the i-Geo Sensing

The complete development of the i-Geo Sensing code holds the responsibility of processing the geo-location data from the sensor in an end-to-end manner. This end-to-end manner is summarized via the following statements.

- i-Geo Sensing processes the sensor data or it requests a continuous stream of sensor data that brings the geo-location of the sensors as a triplet (X, Y, Z) coordinate.
- i-Geo Sensing conducts an initial suggestion of the fracture geometry (i.e., expected half-length, fracture height, and average fracture aperture) via the code's unsupervised machine learning (ML) algorithm workflow. This initial suggestion purely relies on the geo-location data provided by the sensors and does not rely on the properties of the formation(s) the sensors are injected into.
- i-Geo Sensing conducts possible corrections of the fracture geometry (i.e., the quantities from the unsupervised ML workflow plus physical-fracture propagation properties) via the code's supervised ML workflow. This later correction relies on the properties of the formation(s) the sensors are injected into.
- i-Geo Sensing provides a secondary Graphical User Interface that connects the user interaction with the research code that is briefly described above.

6.2. Synthetic case description

To comply with the illustrative purpose of demonstrating the ML workflows, two synthetic environments are described in this report. The synthetic environment for the unsupervised ML workflow has a core aim to validate the clustering and fracture planar diagnostic efficacy, and the environment for the supervised ML workflow has a core aim to validate the calibration, history matching, and explainability aspects.

6.2.1. For the unsupervised ML workflow

The efficacy of the unsupervised ML workflow is tested by three synthetic fracture networks. These three synthetic fracture networks are designed based on the 2D scanned core images of fracture networks subsurface at different measured depths in feet (9490-9493, 9560-9563, 9566-9569) and different levels of geometry complexity (referred to Figure 103). Figure 103 provides details of the raw 2D scanned core images (stored in JPEG format). To generate inputs for the unsupervised ML workflow, the format of the input data is processed as Cartesian coordinates or transformable to Cartesian coordinates (preferably TXT format). As a result, additional image processing steps are conducted to achieve the desirable TXT input format.

Core samples' images are initially imported and transformed into grayscale, which is further capable of separating irrelevant pixels from fracture networks' pixels (i.e. "fractured" pixels). In a gray-sale image, pixels are scaled in their intensity values, which vary in a scale between 0-255. Initial analysis for a gray-scale image typically starts with its histogram of pixel intensity. The separation process is performed by Otsu image segmentation. Otsu algorithm chooses the optimal value from an image's histogram of pixel intensity and further detaches the image into two fragments: the main fracture network (which has pixel intensity 255 - white) and irrelevant pixel

body (which has pixel intensity 0 - black). Albeit Otsu segmentation can extract the closest version to the desired base fracture networks, additional algorithms are necessary to extract the desirable and complete synthetic fracture networks. The supporting algorithms include pixel filling (i.e. fill fractured pixels into desired voids), fragment separation (i.e. divide a fracture network into smaller fragments to perform more effective pixel filling), and fractured pixel recovery (i.e. recover a group of fractured pixels in a fracture's network fragment).

Desirable fracture networks extracted from the scanned core images are maintained as 2D images (stored in PVG format). Under the assumption that the propagation of a fracture network is uniform along the remaining dimension, commercial 3D editing & printing software conducts extension of the 2D imaging base fracture networks into 3D imaging fracture networks (stored in STL format). The stored format of the 3D imaging fracture networks is processed and randomly sampled (reflect the practical aspect of the fact that transmissible SMPs have no specific pattern inside the fracture networks) to create synthetic input geo-location data from Smart Microchips as Cartesian coordinates. Figure 104 provides the projected 2D overviews of synthetic 3D imaging fracture networks used in this study (fracture propagation direction is perpendicular to the projected images). For design purposes, the synthetic fracture networks in Figure 104 increase complexity from left to right. The 1st synthetic network (Figure 104, left) is composed of 4 fractures with almost uniformity in shape. The 2nd synthetic network (Figure 104, middle) is composed of 3 fractures and one smaller network with moderate non-uniformity in shape and low complexity in branching. The 3rd synthetic network (Figure 104, right) is composed of one fracture and two smaller networks with non-uniformity in shape and high complexity in branching.

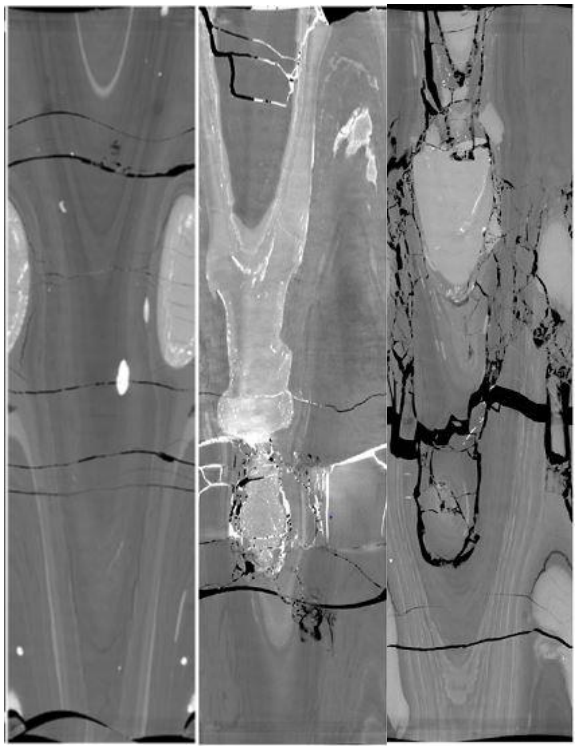


Figure 103: The raw 2D scanned core images used to design synthetic fracture networks

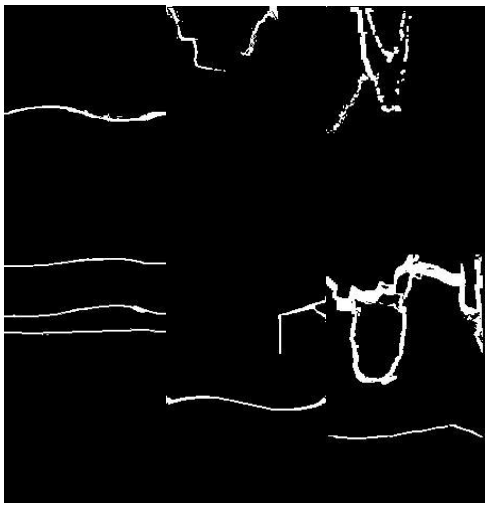


Figure 104: 2D projection of the synthetic fracture networks (complexity increases from left to right)

The unsupervised ML workflow performance is evaluated based on the following criteria:

1. Prediction capability: this is qualitatively defined by the detection of the fracture propagation direction, geometry complexity (i.e. the clustering's efficacy to recognize the fractures in the fracture network), and shape of the predicted geometry (when compared to the synthetic fracture networks as "ground-truth").
2. Robustness in execution: this is defined by similarity in prediction for different runs, under the context of a specified number of transmissible Smart Microchips. This criterion is proposed to test the stability of unsupervised ML workflow to control any stochasticity. The total robustness of all transmissible Smart Microchips is defined as Equation (4).

$$Total\ Robustness = \frac{1}{N_{SMP}} \sum_{j=1}^{N_{SMP}} \frac{\sum_{i=1}^{N_r} R_j}{N_r} \quad (4)$$

3. Consistency: this is defined by the remapping capability for the tested fracture network under the context that not 100% of the Smart Microchips injected subsurface can transmit the geo-location data back (because of the high temperature, high-pressure condition subsurface). To evaluate this criterion, 9 separate consistency cases of the number of transmissible Smart Microchips (between 50-90% of the total Smart Microchips injected, increment of 5%) are used for each synthetic network. The weighted average of the consistency from the 9 scenarios represents the overall consistency for each synthetic network. This weighted average is used based on the convention that the lower percentage of the total injected SMPs incurs more difficulty (since less data is available). Consistency score is defined as Equations (5) and (6).

$$Consistency = \frac{1}{N_{case}} \sum_{i=1}^{N_{case}} W_i C_i \quad (5)$$

$$\sum_{i=1}^{N_{case}} W_i = 1 \quad (6)$$

In Equation (4), R_j denotes the similarity score between the result from the unsupervised ML workflow (one run, one Smart Microchip) and the "ground-truth" result from the design of the synthetic cases. A R_j value of 0 means dissimilarity and a R_j value of 1 means similarity. N_r is the number of runs from the unsupervised ML workflow. N_{SMP} is the total number of transmissible Smart Microchips.

In Equation (5), C_i is the consistency score between the consistency cases. A C_i value of 0 means inconsistency between the cases, and a $C_i$value of 1 means consistency between the cases. W_i is the weighting factor for the consistency cases to determine the weighted average (which directly explains the summation of 1 in Equation (6)).

6.2.2. For the supervised ML workflow

A compositional simulation case for a horizontal well, hydraulically fractured, serves as the comprehensive test case for the supervised machine learning workflow. An overview of the reservoir and the single horizontal well (visual provided in ResFrac® academic license) [65] is provided in Figure 1. 59 geological layers in the reservoir vary between 7500 ft to 8958 ft. A single well penetrates through the target depth at 8243 ft. Reservoir properties (e.g., minimum horizontal stresses, porosity, permeability, relative permeability curves) are in the Static Model and Initial Conditions, Geological Units. A plot of the Young Modulus and Poisson Ratio variation along the model's depth is presented in Figure 106. The academic license of ResFrac allows similar plots for other geological properties in its plot interactive interface.

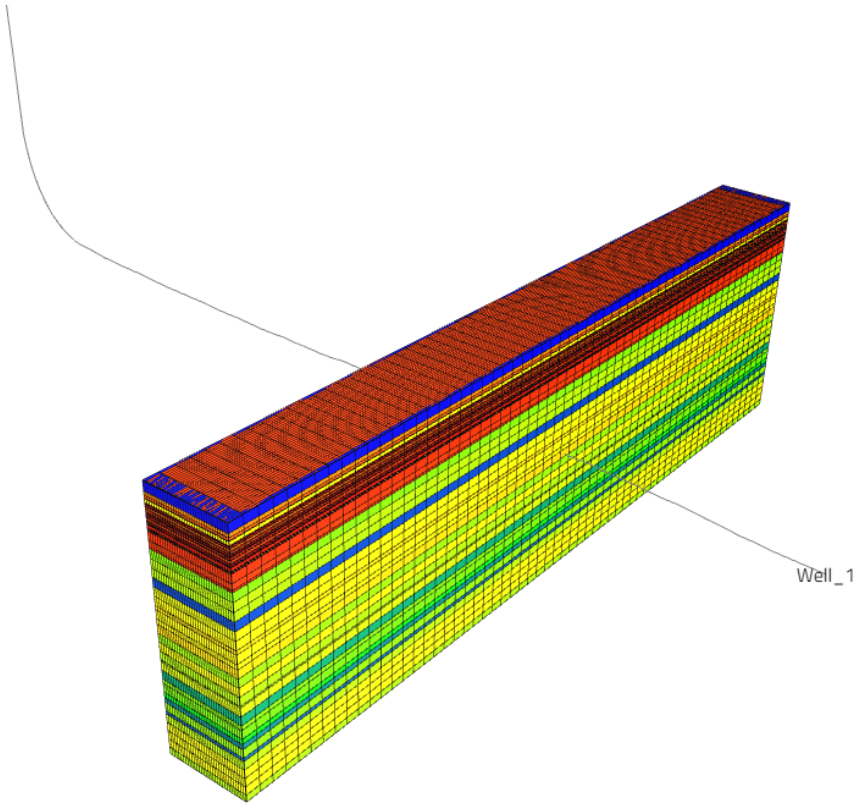


Figure 105: An overview of the test case for the supervised ML workflow

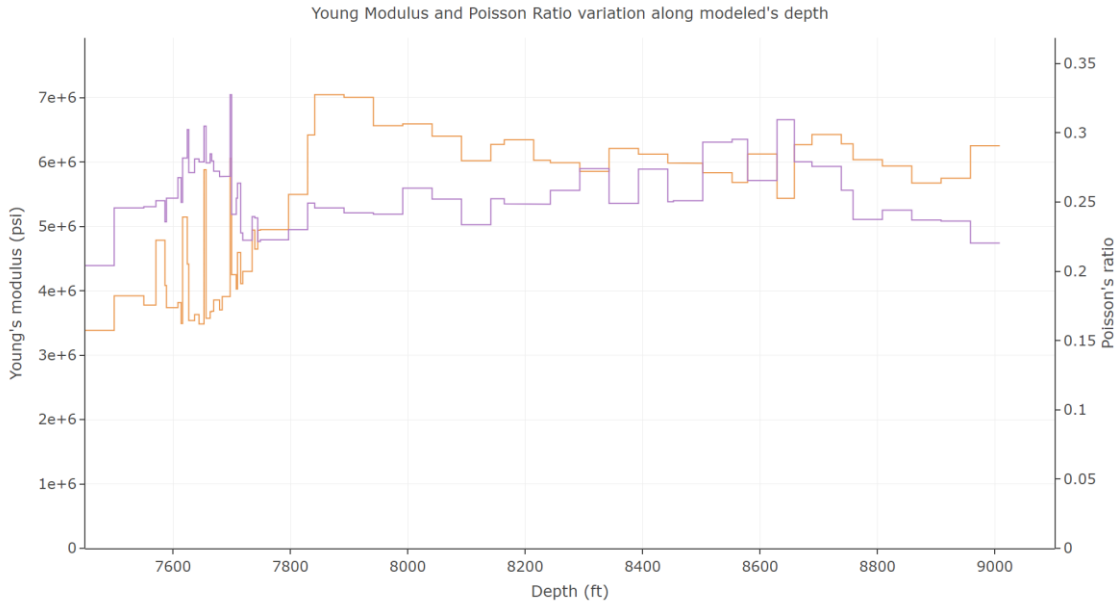


Figure 106: Variation of Young Modulus and Poisson Ratio over the model's depth

The single horizontal well is injected for a total of 238.07 minutes with an injection pressure of 20000 psi. The injection schedule is presented in Figure 107.

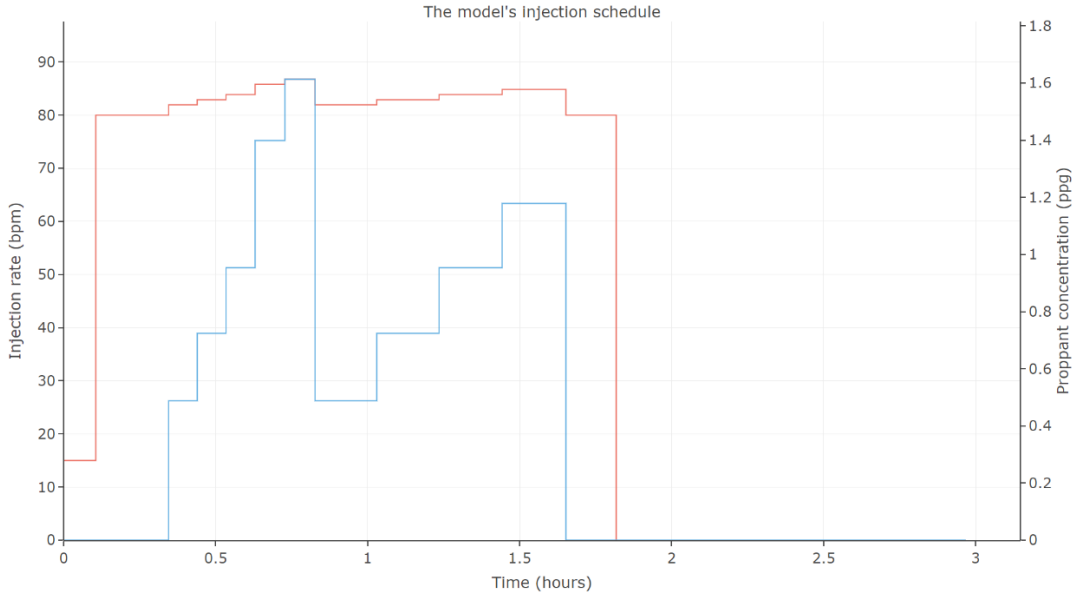


Figure 107: Overview of the injection schedule for the model

After the injection schedule is performed, the well is shut down for an additional 120 minutes and later is converted into the producer mode. The Bottom Hole Pressure (BHP) and the oil production rate are presented in Figures 108 and 109, respectively. The data plot in Figure 108 indicates that the Initial Shut-In Pressure (ISIP) is approximately 5800 psi at close to 20 days of simulation time.

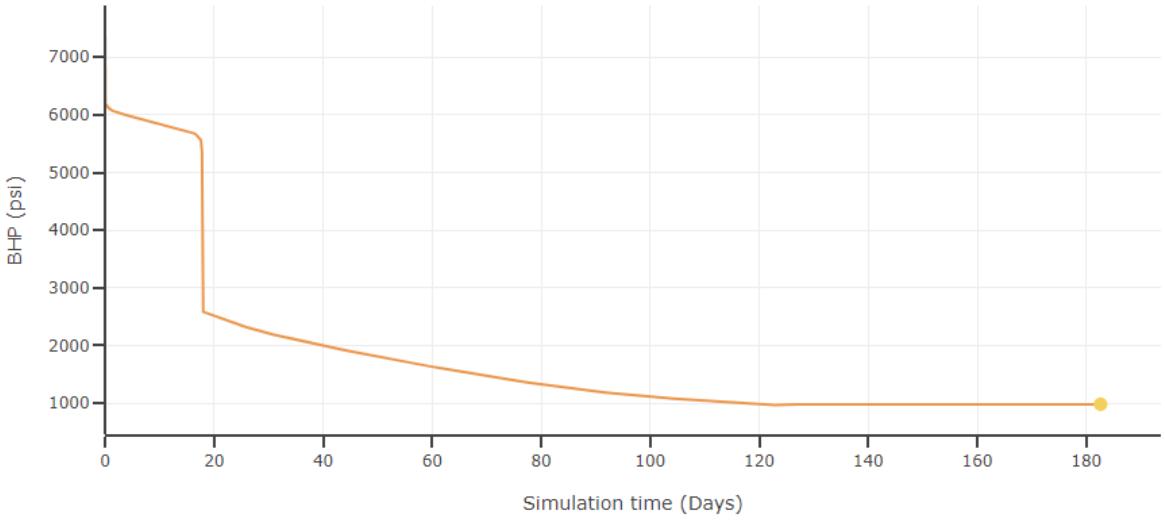


Figure 108: Bottom Hole Pressure data for the model's simulation lifecycle

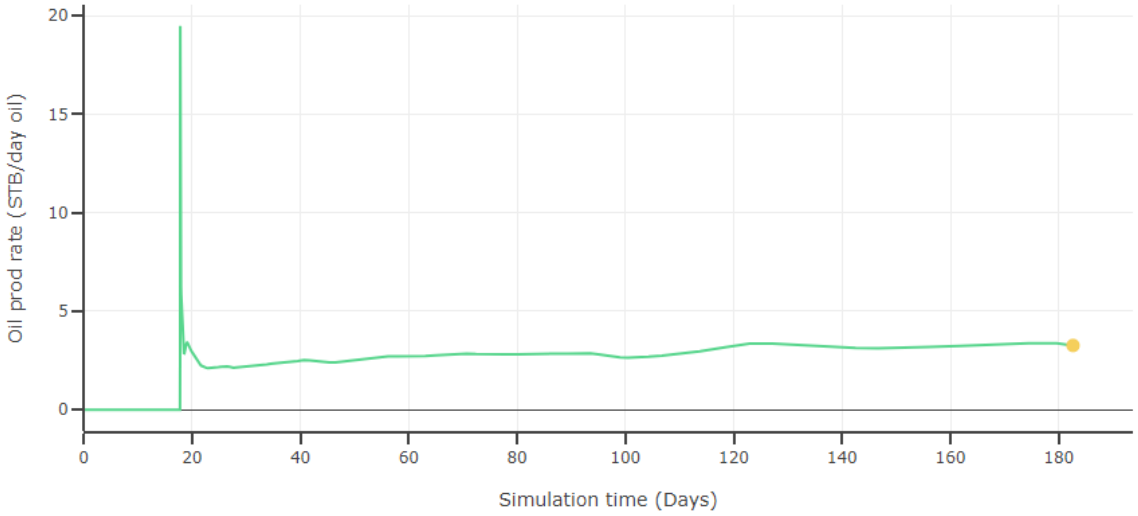


Figure 109: Oil production rate data for the model's simulation lifecycle

Since ResFrac® is a coupled fracture-propagation simulator and a reservoir flow simulator, fracture propagation properties can be visualized along the model's life cycle along with conventional reservoir properties (as in Figures 108 and 109). Figure 110 provides the total fracture aperture in inches at an early propagation time step (left) and a late propagation time step (right). Figure 111 provides the total proppant volume fraction (dimensionless) at the two similar time steps as Figure 110. In Figures 110 and 111, the color scale for the specific property (i.e., total aperture or total proppant volume fraction) is similar across the simulation time in ResFrac, and henceforth these Figures present the expected observations of proppant transport in the test case. In the early time steps, the aperture is continuously opened along the wellbore's perforation direction, and the amount of injected proppant is not substantial. At the later time steps, the fracture is gradually closed (i.e., a gradual decrease in fracture aperture), and a substantial amount of proppant is settled inside the fracture. ResFrac uses a multi-opening fracture tip model to locate the front of the propagating fracture (Multi-El Tip model), which more details can be found in [65].

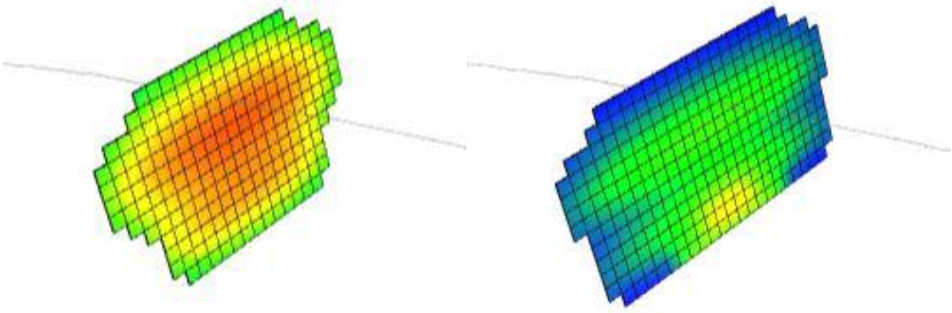


Figure 110: Total fracture aperture at early propagation (left) and late propagation (right)

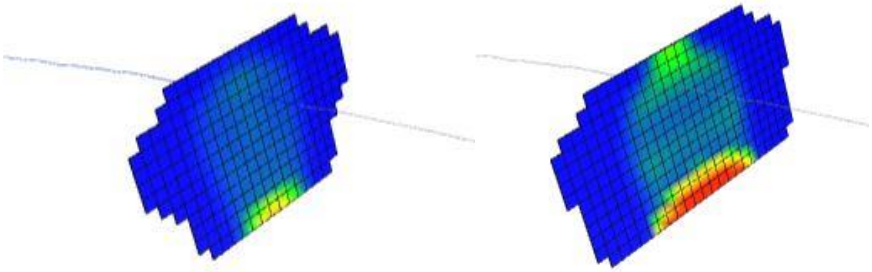


Figure 111: Total proppant volume fraction at early propagation (left) and late propagation (right)

To test against the supervised ML workflow within the context of this test case, the synthetic Micro Chips' data is sampled along the model's life cycle under the following assumptions:

1. A higher proppant volume fraction indicates a higher density of sensors located. This is intuitive since the Micro Chips are injected along with the conventional proppants.
2. There is a random "dead rate" of sensors across fracture dynamically, and this dead rate is a dependent variable on "ground-truth" fracture aperture and fracture pressure (which determine the confinement stress to the conventional proppant, and intuitively the Micro Chips shall be exposed to the similar confinement stress).

Besides the synthetic sensor data as the input to the supervised machine learning workflow, other calibration data such as ISIP, BHPs (as specific time steps), and oil production rates (at specific time steps) are provided to the supervised ML workflow. The calibration objective is to match the ISIP, BHP, and oil production rates within an acceptable uncertainty. Further details are provided in 6.5 and 6.7. From 6.3 and below, geo-location data is the abbreviation of the data from Micro Chips.

6.3. Intuition in understanding the sensor distribution inside the subsurface fracture environment

Throughout the i-Geo Sensing code, the sensor data is requested/processed in the form of three-dimensional coordinates, or a normalization from three-dimensional coordinates. Under the subsurface, the Micro Chips are supposed to be injected into the formation (the 1st assumption in 6.2) Consequently, except for recording geo-location data, the distribution of the Micro Chips inside the fractures(s) shall have a similar look as Figure 112 for a single fracture (at any point of time within the lifetime of the reservoir) [32].

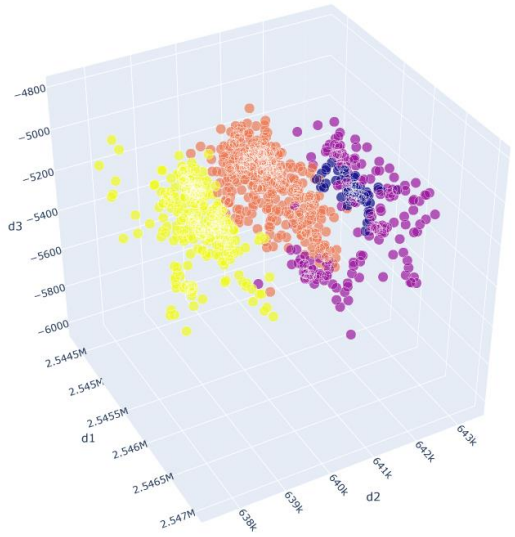


Figure 112: Distribution of the MicroChips subsurface (generated from the synthetic environment)

Coupled with the nature of the proppant distribution inside the fractures, the following statements can be appropriately declared.

1. Sensors are more concentrated at the locations close to the injection point, and the "density" of sensors during the lifetime of the reservoir shall follow the pattern of proppant existence inside the reservoir [32].
2. Since the fractures are propagated with a substantially long half-length and fracture height dimensions (in the degrees of ft) compared to the aperture (in the degrees of in), sensors are expected to locate around a planar surface in which their projection of the geo-location data to that planar surface reveals an approximation of the fracture aperture.
3. Although some Micro Chips are "dead" during the lifetime of the reservoir because of subsurface environments, the amount of data transferred from the sensors is, naturally, proportional to a quantity that correlates with the proppant distribution (e.g., proppant volume fraction).

These 3 statements serve as essential foundations that lead to all further development understanding and algorithmic thinking of the workflows that exist in the i-Geo Sensing. Each of the sub-sections below 6.3 reflects one or more of them and is further explained in the sub-section itself below.

6.4. The unsupervised machine learning workflow

The unsupervised ML workflow is a stand-alone module inside i-Geo Sensing that receives the transmissible, 3D geolocation data. Its architecture is presented in Figure 113 [43, 60]. Reminded of intuitions 1 and 4 from the previous section, the unsupervised module is designed to characterize the fracture networks via fracture clusters. Three core algorithms contribute to this module include Uniform Manifold Approximation and Projection (UMAP) [34, 44], Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [35, 54], and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [35, 54, 56].

UMAP performs dimensional projection of the geo-location data from 3D into a 2D latent space (or sub-space) for unsupervised clustering. The couple DBSCAN – HDBSCAN forms the Ensemble Clustering and performs unsupervised clustering in the 2D latent space to identify the prospective fractures in the fracture network as "fracture clusters", including propagation directions and branching complexity. During the unsupervised clustering by the Ensemble Clustering, the Mixture Clustering Epsilon coefficient is facilitated to control the clustering's performance [54].

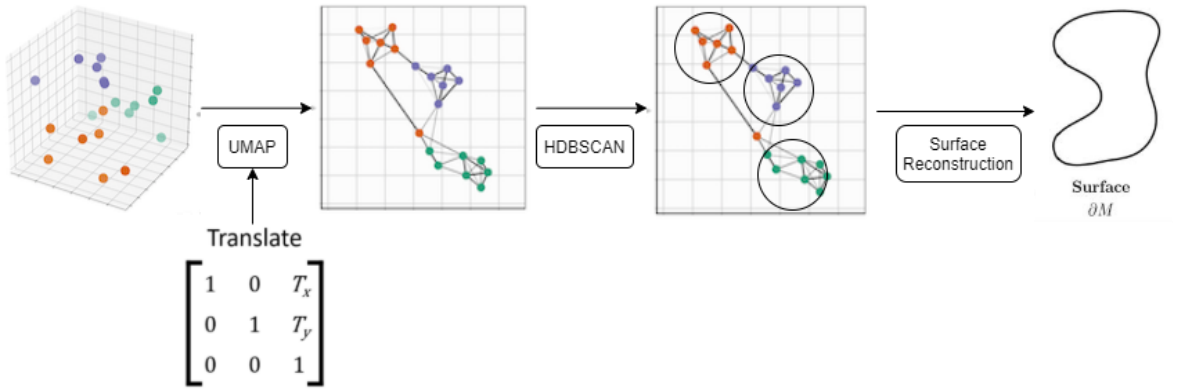


Figure 113: The unsupervised ML workflow

An immediate challenge to control these hyperparameters in the unsupervised ML workflow is the fact that hyperparameters in both UMAP and HDBSCAN are typically controlled manually and empirically based on the specific nature of the input data [34, 44]. In the workflow, applying geometrical assumptions based on the specific nature of fracture networks' geo-location data leads to the following hyperparameter estimations, referred to as Equations (7), (8), (9), and (10).

$$A = \begin{bmatrix} d_{11} & 0 & 0 \\ 0 & d_{22} & 0 \\ 0 & 0 & d_{33} \end{bmatrix} \tag{7}$$

$$d_m = 0.05 \pm 5 \times 10^{-3} \tag{8}$$

$$k = 0.1N \pm 100 \tag{9}$$

$$C_{sm} = S_m = 0.01N + n_c \tag{10}$$

In Equations (7) and (8), N is the number of functional Micro Chips (i.e. the Micro Chips that can transmit signals). In Equation (10), n_c is a positive integer, defined as the additional number of samples to avoid the critical "tiny cluster" and "condensed tree failure" errors while executing HDBSCAN. n_c physically represents a control to avoid small clusters of Micro Chips can be

grouped as a fracture cluster (in fact they belong to another fracture cluster). Additionally, the existence of n_c contributes to minimizing the "noise" classification for the input in HDBSCAN, since HDBSCAN naturally considers a certain amount of noise in its input data. However, noise from the geo-location data is minor because almost all transmissible Micro Chips are useful. The empirical process in this study indicates the value of n_c ranging between 5 and 15. The evaluation noise score metric (s_N) that is defined as Equation (11).

$$s_N = \frac{N_{noise}}{N} \tag{11}$$

In Equation (11), N_{noise} is the number of noise data points classified by HDBSCAN. An optimal hyperparameter set for a specific value of N is the set that minimizes s_N, preferably a zero-proximity value (to reflect the unavoidable presence of "minor clusters"). The value of N ranges between 1000-5000 as a practical measure for the number of injected Micro Chips. The target variable s_N has a mixture of records between zero and non-zero values.

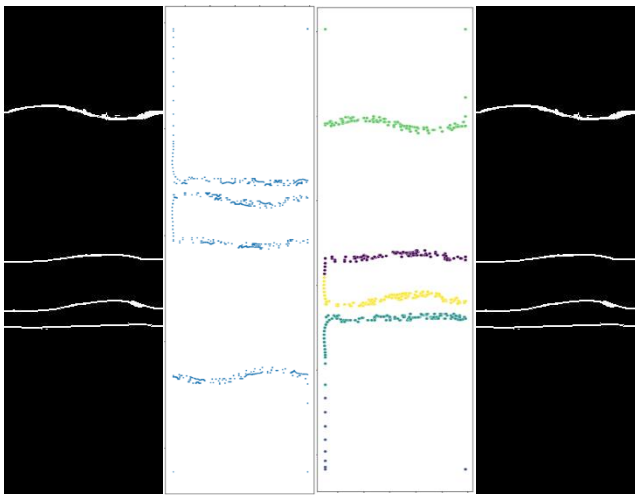


Figure 114: Processing of the geo-location data for the 1st synthetic fracture network

Figure 4 provides the visualization of the unsupervised ML workflow for the 1st synthetic case in this study, and all images in Figure 111 are 2D projections from the 3D realizations. From left to right in Figure 114, the following sub-images are presented: the synthetic fracture network transmissible data from the Micro Chips, the fracture network's structure diagnostic, and one realization of the fracture network exported from the unsupervised ML workflow that has the closest visual compared to the original synthetic input.

Table 7: Evaluation results for the unsupervised ML workflow (synthetic environment)

Fracture network	**Evaluation criterion**		
	Prediction capability	***Robustness in execution***	***Consistency***
1	Highly satisfied	100%	100%
2	Satisfied	90%	100%
3	Fairly satisfied	85%	100%

Table 7 reveals valuable insights into the unsupervised ML workflow's performance. The consistency is achieved at 100% for all synthetic cases, indicating the workflow functionality is independent of the amount of transmissible Micro Chips. The robustness is achieved at 100% and 90% for the 1st and 2nd synthetic cases, however, is capped at 85% for the 3rd synthetic case. Therefore, the use of assisted affine transformation is less effective for the high geometry complexity in the 3rd synthetic case. The rated prediction capability for the 3rd synthetic case falls behind the other 2 synthetic cases, which solidly prompts further reasoning for the impact of high geometry complexity on the workflow's performance.

Figure 115 compares the effect of the affine transformation [41] to the performance of UMAP, and consequently HDBSCAN in the workflow. As aforementioned, the coupled algorithm is implied as a matrix to scale the received geo-location data and alter the absolute distances between the Micro Chips. Affine transformation inherently does not modify the internal structure of the fracture network. Consequently, raw geo-location data processed by any affine transformations improves the low-dimensional projection result from UMAP since UMAP is very sensitive to the absolute distances between neighbors (i.e., the absolute distances between the Micro Chips). Figure 115 compares the effect of using affine transformation and not using affine transformation for the 1st synthetic case. In figure 115 (left), UMAP with the coupled algorithm reflects the designed structure of the 1st synthetic fracture network. The projected subspace has viewed 4 clusters corresponding to the 4 fractures. In contrast, in Figure 115 (right), the standalone UMAP does not reflect the designed structure of the 1st synthetic fracture network, as the projected subspace is viewed as three clusters. The two fractures at the bottom of the 1st synthetic case have extreme proximity, and UMAP is not capable of separating them in the projected subspace without using affine transformation. A similar effect is observed in the 2nd and 3rd synthetic cases.

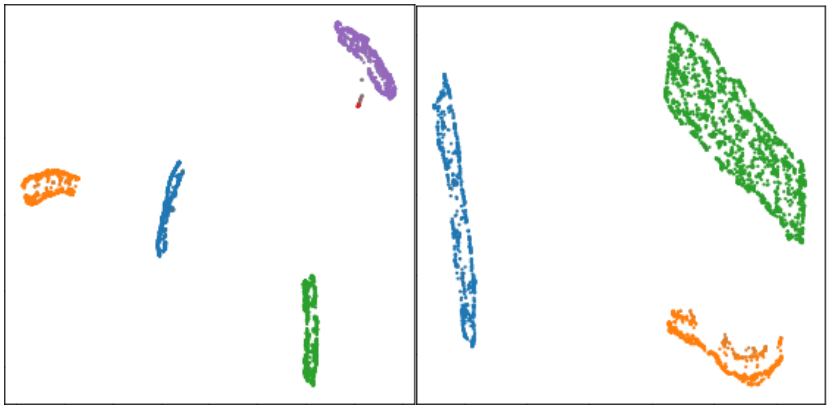


Figure 115: Effect of assisted affine transformation on the performance of UMAP in the 1st synthetic fracture network (with transformation – left, without transformation – right)

6.5. Sensor data profiling

According to the foundations from 6.2, the geo-location data is received as a three-dimensional "point cloud" that has an arbitrary representative geometry. Although there exist processing approaches from this three-dimensional point cloud that map the local and/or global spatial characteristics of the points to the representation of the geometry, these approaches require extensive computational power to be trained and deployed as scalable proxy models (details about proxy modeling is later described in 6.9 and 6.10) [47, 48]. Additionally, these approaches are aimed at developing geometries that are, unfortunately, not representative of fractures subsurface.

Therefore, we implement a processing technique in the i-Geo Sensing code to process the sensor data that is robust and provides scalability for later proxy modeling in 6.9 and 6.10. This technique is named sensor data profiling. To better understand the technique, it is more illustrative to present the fracture geometry that comes from the ResFrac® software. This illustration is provided in Figure 116.

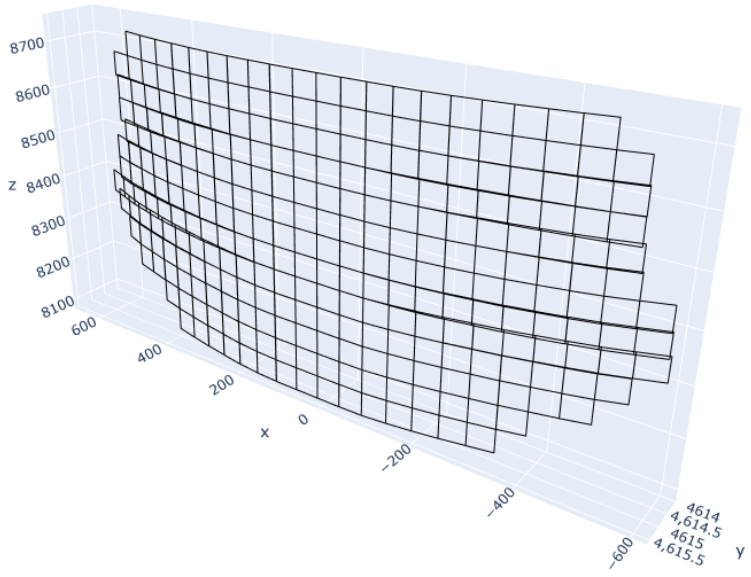


Figure 116: The ground truth fracture geometry (reconstructed from ResFrac® software [50])

One noticeable observation from Figure 116 is that the presented fracture geometry has some degree of complexity in shapes as it is "curved" along the y-axis (the axis that has a contrast between the minimum horizontal stresses). This complexity is caused by the non-drastic difference between the horizontal stresses. Essentially, Figure 116 divides the ground-truth fracture geometry into uniform regions (namely, fracture elements in ResFrac®), and in each of these elements, there is a dynamic occupation of Micro Chips. Throughout the lifetime of the reservoir, it is not abnormal to observe that some of these elements may not have an occupation of Micro Chips. Across the dimensions of the fracture, the density of Micro Chips per element, or propagation length (i.e., half-length), mainly depends on the fracture pressure (which impacts the "dead rate" of the Micro Chips) and fracture aperture (which determines the cumulation of Micro Chips within the fractures, similar to conventional proppants). Consequently, i-Geo Sensing creates sensor data profiles that summarize the fracture aperture (i.e., the y-dimension recording in Figure 116) along the half-length propagation and the fracture height propagation at any time it receives geo-location data.

Within each time step that i-Geo Sensing receives the geo-location data, it deploys the unsupervised machine learning workflow to recognize the fracture plane per cluster. Per the fracture plane, i-Geo Sensing successively determines the smallest dimension in geolocation value (after normalization to the centroid of the data). For example, Figure 116 indicates the smallest dimension is the Y dimension, and consequently, X and Z are the larger dimensions. Reminded that the fracture aperture is a fraction in value compared to the fracture half-length or fracture height (a fraction of an inch versus hundreds of feet), the smallest dimension is the dimension holding the fracture aperture information. Per the larger dimension, it is divided into uniform intervals, and per interval, the average value of the smaller dimension of which the sensors' larger

dimension falls into that interval is computed and averaged. Consequently, the larger dimension provides a 1D array of values with size [N, 1], in which N is the number of divided intervals, or namely, the resolution of the profile. There is a maximum of 2 sensor data profiles per time step. Figure 117 presents a sensor data profile for the half-length dimension between -600 ft and 600 ft (normalized in ResFrac® around the centroid which is at 0 ft), and with a resolution of 50 intervals.

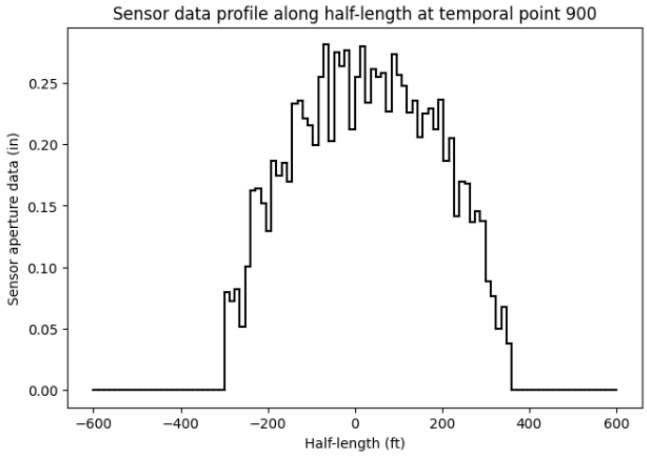


Figure 117: Sample sensor data profiling

To further illustrate the validity of the information the sensor data profiles provide to the i-Geo Sensing, Figure 118 presents the sensor data profiling for one larger dimension at two different time steps. Figure 118 clearly shows that the fracture propagation at the later time step (right) has a growth on the two "tails" of the profile compared to the initial time step in which the two "tails" of the profile remain flat (i.e., zero or significantly low value of aperture). Additionally, the later time step presented in Figure 118 (right) indicates that the fracture is entering its closing phase (since the profile is lower at the centroid and is higher expanding from the centroid). Sensor data profiles contribute their usefulness at the proxy deployment phase, which is again, further described in 6.8 and 6.9.

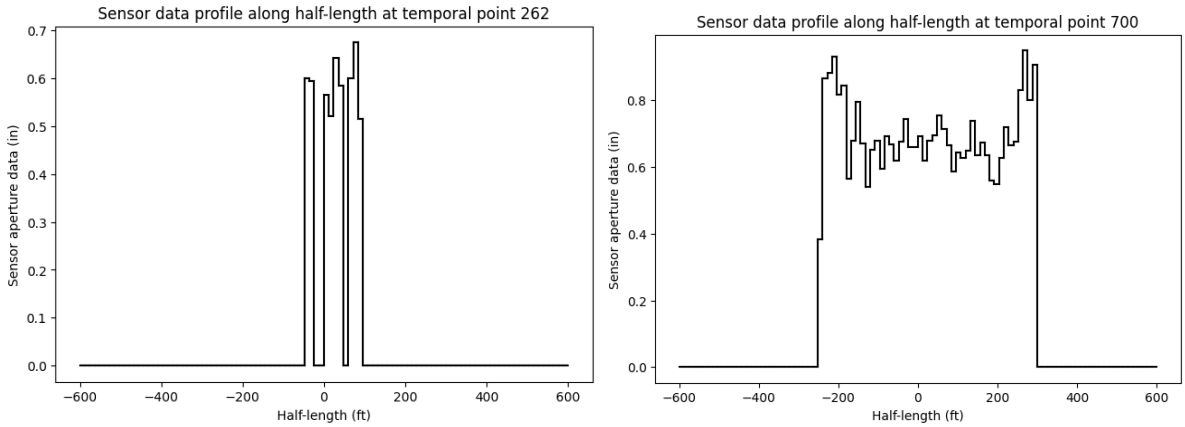


Figure 118: Sample sensor data profiles at two different time steps

6.6. Calibration of the fracture geometry and history matching from the unsupervised machine learning workflow

As described in 6.3, the unsupervised ML workflow is responsible for suggesting an initial diagnostic of the fracture geometry. This initial diagnostic is purely based on the geo-location data and is not informed any further by the reservoir's characteristics. Consequently, this initial diagnosis needs to be corrected in realistic scenarios in which field data is available (besides the geo-location data) [51].

In fracture model calibration, the conventional approach involves matching typical parameters such as the Initial Shut-In Pressure (ISIP) and "early" after-shut-in pressure. After being calibrated, the fracture geometry is coupled with a reservoir simulator (in terms of grid/mesh refinement) for further history matching with the field production data. Under the addition of near-wellbore fracture geometry data from the Smart Microchips, information about the initial fracture aperture (w_{f0}) and initial fracture height (H_{f0}) is available. w_{f0} and H_{f0} are estimated using Single Value Decomposition (SVD). Consequently, fracture model calibration in the i-Geo Sensing additionally includes the match for these parameters besides the match for typical parameters mentioned above (which are conducted once).

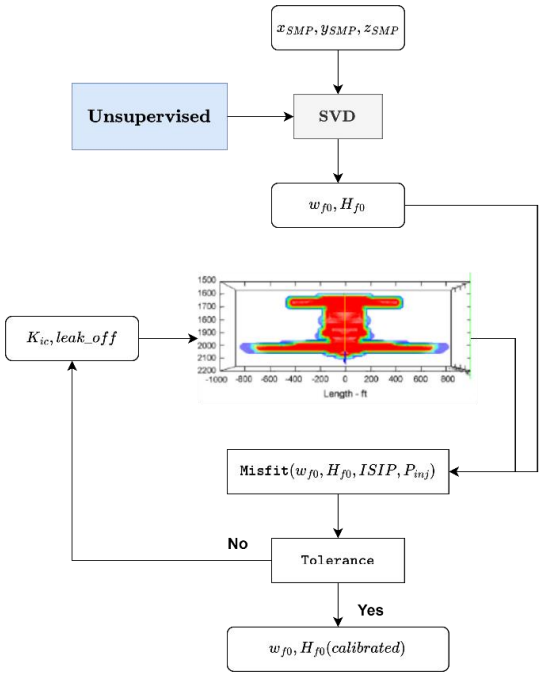


Figure 119: Fracture calibration workflow in i-Geo Sensing

In history matching, the conventional approach involves uncertain parameters (e.g., porosity, permeability & relative permeability curves, and saturation profile), providing the one-time calibration for the fracture model. Under the addition of sensor data profile(s) per simulation time step, history matching in i-Geo Sensing uses the sensor data profiles to calibrate the uncertain parameters. Since i-Geo Sensing facilitates proxy modeling techniques (i.e., estimation of the response parameter(s) via a surrogate model in replacement of a local derivative estimation), history matching in the i-Geo Sensing is performed using the global-space optimization algorithm via Tree Parzen Estimator's Bayesian Optimizer [63, 64, 66].

The objective function for history matching in i-Geo Sensing is defined as the BHP, flow-back/production data misfit function(s) (for example, Mean Square Error). After the sensitivity

study for all uncertain parameters and initial manual tuning (if necessary), a global search space is initialized. Per iteration of the optimization loop, the objective function's metrics are compared against the previous iteration, and the Tree Parzen Estimator (TPE) measures the Estimation of Improvement (EI) between the subsequent iterations. The EI further determines the search direction of all uncertain parameters within the initial search space, until a convergence in the EI is satisfied. Schematic of the optimization loop is presented in Figure 120.

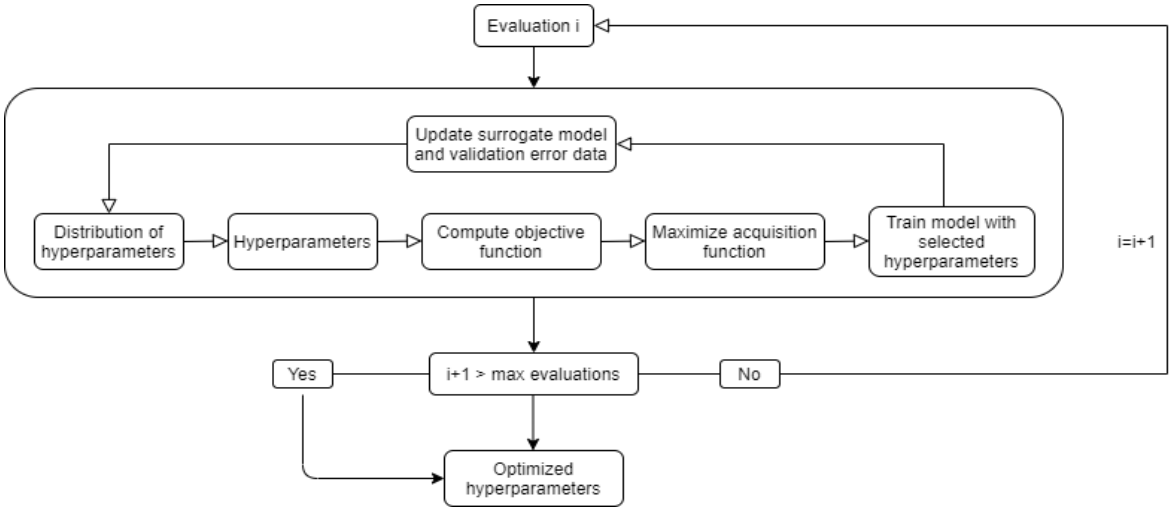


Figure 120: The Bayesian Optimizer engine used in the i-Geo Sensing

Within the scope of history matching, traditionally simulations are required to run for the uncertainty quantification metrics and the misfit evaluation. In i-Geo Sensing, this task is speeded up by using the surrogate modeling inside the supervised ML workflow (quantile-loss Extreme Gradient Boosting/Gradient Boosting Machine). Details about the proxy models and the supervised ML workflow are in 6.8 and 6.9.

6.7. Design of Experiment

Through 6.2-6.6, proxy modeling is mentioned as an effective surrogate to estimate the response parameters without running the specific simulation case at the time of performing fracture calibration or history matching task. To fully leverage the proxy modeling in the i-Geo Sensing, Design of Experiment (DoE, [67]) combined with ResFrac® is used. ResFrac® offers the multi-physics Linear Elastic Fracture Model (LEFM) that is field-scalable, and it has a fracture propagation simulator coupled with a reservoir flow simulator. Therefore, in this project, ResFrac® is selected to serve as i-Geo Sensing backend simulators, in a semi-automated manner. Figure 121 presents the scheme Design of Experiments embedded inside i-Geo Sensing.

The DoE code in the i-Geo Sensing exists as a standalone module, which is responsible for processing base simulation files from ResFrac®. A ResFrac simulation run requires two simulation files, named "settings" and "input" text files, both are human readable. ResFrac defines the simulation model via a system of multiple "entry" variables (in both settings and input files). Each entry variable holds the variable name, the variable length, and the variable data in this sequential order. To generate DoE cases in batches and perform fracture model calibration/history matching studies, i-Geo Sensing's DoE module is capable of:

1. Parse the two simulation files in ResFrac per simulation run into encoded entries and their corresponding data.
2. Diagnose the optimal variable data type per entry.
3. Connect a DoE case data to the correct entry variable (regardless of its location inside settings or input files)
4. Re-write the settings and/or input files and batch-run the simulation after the re-written process.

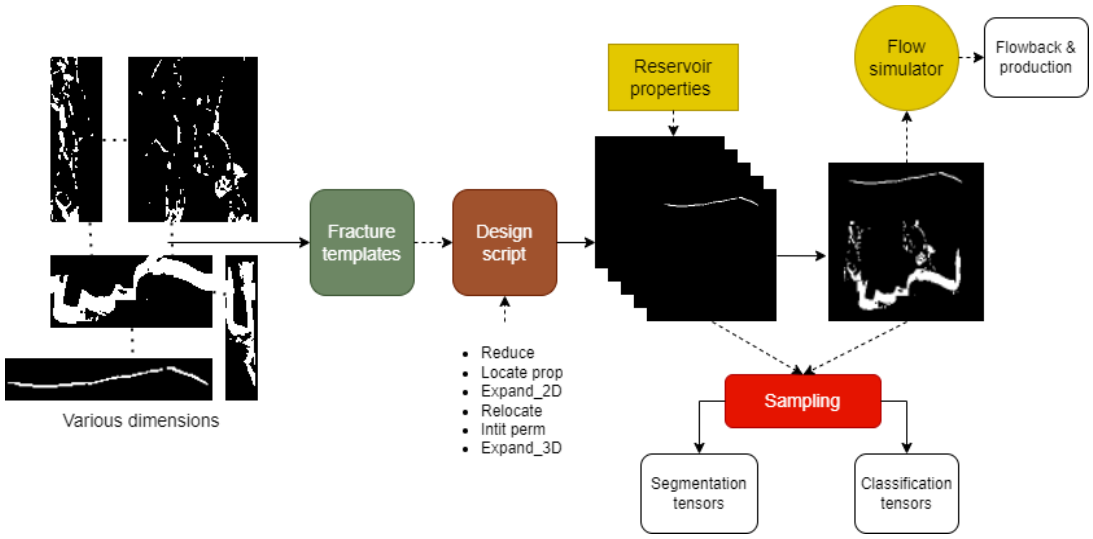


Figure 121: Design of Experiment generator in i-Geo Sensing

i-Geo Sensing provides the DoE for ResFrac in a semi-automatic because of the limitation of the ResFrac academic license, and users are still required to submit the simulation jobs to the ResFrac® server. An illustration of the DoE outcome for fracture geometry realizations is provided in Figure 122.

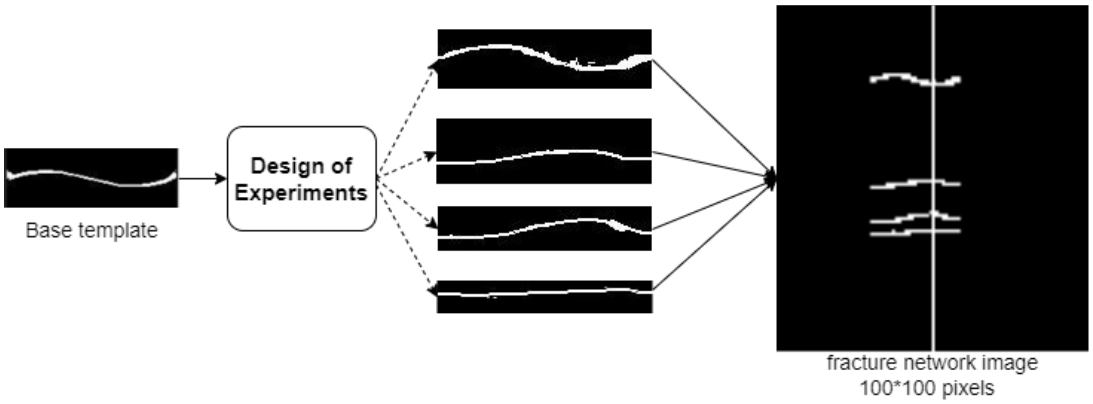


Figure 122: An illustration of the DoE for fracture geometry in i-Geo Sensing

The fracture calibration in i-Geo Sensing fully leverages the sensor data sensor data profiles at (synthetically) recorded time steps. Consequently, the fracture calibration proxy processes the sensor data profiles, additional fracture propagation properties (if requested by the users), and the recorded time (converted to dimensionless) as proxy input variables, and a pressure quantity (e.g., BHP) as the output variable. The prediction of the pressure quantity is sequentially processed as

one of the inputs for the history-matching proxy. Table 8 provides selected parameters to be facilitated for the test case's fracture calibration and history matching.

Table 8: The test case's DoE parameters' distributions

Parameter	***Corresponding ResFrac entry***	***Distribution***
Residual water saturation	`matrixcurvesets`	norm (0.2, 0.05)
Residual oil saturation	`matrixcurvesets`	norm (0.2, 0.05)
Residual gas saturation	`matrixcurvesets`	norm (0.03, 0.005)
Relative fracture toughness	`relativefracturetoughnesspersqrt fracturelengthscale`	uniform (0.5, 1)

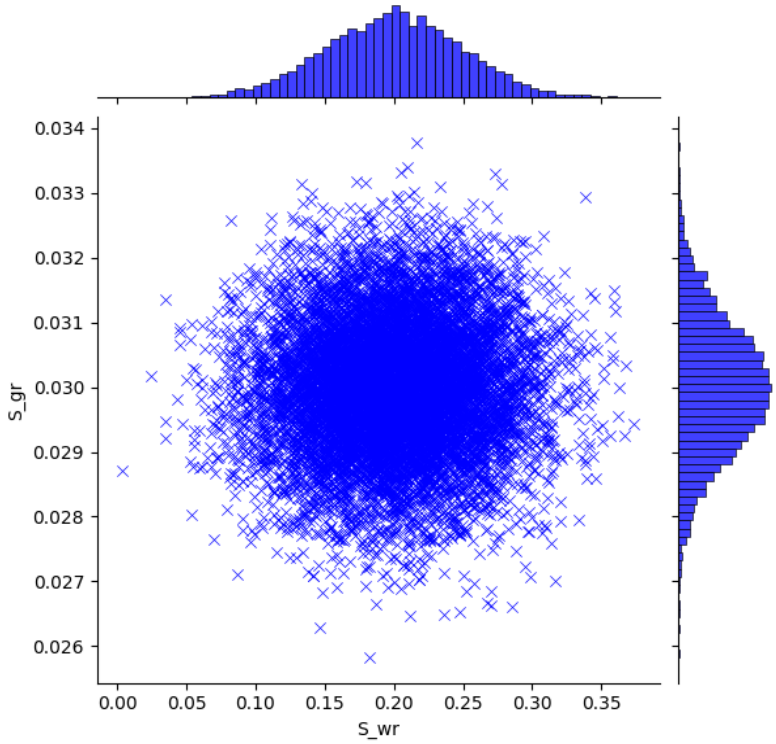


Figure 123: Visual of the joint plot (DoE's coverage) between distributions of two parameters "S_{gr}" and "S_{or}"

Figure 123 illustrates a functionality in i-Geo Sensing to visualize 2 distributions of users' selected variables simultaneously to determine the DoE efficacy in generating data. The example presented in Figure 123 indicates that there needs to be further coverage for the two variables at the four edges of the DoE experimental surface. Figure 124 demonstrates the in-place change of a ResFrac settings text file(s) (inside the DoE simulation cases) for the ResFrac's variable named "relativefracturetoughnesspersqrtfracturelengthscale" after processed through the i-Geo Sensing's DoE module.

Base case

```
Variable name:
relativefracturetoughnesspersqrtfracturelengthscale
Length:
1
Value(s):
0.75
```

DoE case 0

```
Variable name:
relativefracturetoughnesspersqrtfracturelengthscale
Length:
1
Value(s):
0.875
```

Figure 124: In-place change of ResFrac's "relativefracturetoughnesspersqrt fracturelengthscale" entry

Using the DoE module in i-Geo Sensing and the trial parameters in Table 8, a total of 140 realization cases for the 4 conventional DoE parameters and the 50-interval-resolution sensor data profile are generated to serve the training of the supervised machine learning model. 100 realization cases are selected as the initial data batch to train, and the remaining 40 realization cases are maintained as the additional backup to Quality Control the training performance.

Table 9: A snapshot of one DoE case

index	case	surrogate_time	S_wr	S_or	S_gr	relative_frac_toughness	x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2
0	0	2.77778e-07	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0	2.77778e-07	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0	1.38889e-06	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0	0.0837858	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.0	-576.0	-552.0	0.0	-552.0	-528.0	0.0
4	0	0.12605	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.0	-576.0	-552.0	0.0	-552.0	-528.0	0.0
5	0	0.189359	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.0	-576.0	-552.0	0.0	-552.0	-528.0	0.0
6	0	0.284639	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.0	-576.0	-552.0	0.0	-552.0	-528.0	0.0
7	0	0.407052	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.16512175	-576.0	-552.0	0.11871356	-552.0	-528.0	0.1123032818181817
8	0	0.578723	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.2422317399999999	-576.0	-552.0	0.229167125	-552.0	-528.0	0.264571375
9	0	0.716632	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.2740111875	-576.0	-552.0	0.3239723888888889	-552.0	-528.0	0.2809700434782609
10	0	0.855784	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3360523529411764	-576.0	-552.0	0.2837663473684211	-552.0	-528.0	0.315069652631579
11	0	0.985582	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3066376363636364	-576.0	-552.0	0.3475282941176471	-552.0	-528.0	0.3206631578947369
12	0	1.13129	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3339975	-576.0	-552.0	0.319221	-552.0	-528.0	0.3497909615384614
13	0	1.27499	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3126728666666666	-576.0	-552.0	0.3666251428571428	-552.0	-528.0	0.3263827727272726
14	0	1.33822	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3305455	-576.0	-552.0	0.3415690869565218	-552.0	-528.0	0.3651304545454545
15	0	1.57228	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3910372857142857	-576.0	-552.0	0.430424625	-552.0	-528.0	0.3114350370370369
16	0	1.57838	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3418610454545454	-576.0	-552.0	0.3658438750000001	-552.0	-528.0	0.3159668260869565
17	0	1.58927	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3231085199999999	-576.0	-552.0	0.3308139413793106	-552.0	-528.0	0.2969301684210526
18	0	1.68186	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.30275142	-576.0	-552.0	0.3266580678571428	-552.0	-528.0	0.3351949035714286
19	0	1.81784	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.31876041875	-576.0	-552.0	0.3646670051282052	-552.0	-528.0	0.3738109454545453
20	0	1.96784	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3397989636363636	-576.0	-552.0	0.359712792	-552.0	-528.0	0.3457833473684212
21	0	3.10227	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.3183576399999999	-576.0	-552.0	0.3142174702702703	-552.0	-528.0	0.3370922799999999
22	0	3.96784	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.2825145310344828	-576.0	-552.0	0.3197312351351352	-552.0	-528.0	0.293255028
23	0	6.29275	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.2383114454545454	-576.0	-552.0	0.3043404763157897	-552.0	-528.0	0.2592015866666666
24	0	9.6538	0.1277520948071937	0.05758293035489	0.0291761063696614	0.4725	-600.0	-576.0	0.2545000341463413	-576.0	-552.0	0.2361227	-552.0	-528.0	0.2642221071428571

Table 9 provides an overview of a DoE case generated from the i-Geo Sensing, in terms of the tabular data arrangement. Column-wise and from left to right, i-Geo Senning writes in the following sequential order: the simulation time (unit is similar to ResFrac result files, typically in hours), the DoE parameters, the dynamic sensor data profiles (in the X-Y-Z order per interval resolution, described in Algorithm 1), and the response parameters. For illustration, Table 9 shows a fraction of a DoE case, as a complete DoE case for the test in 6.2 shall have 150 X-Y-Z columns

for the sensor data profiles (a single sensor data profile is used for the test case in 6.2, and it has a 50-interval-resolution, henceforth 150 columns).

6.8. Synthetic data generation, wrangling, and tabulation processing

As a figurative description, the complete data generation and processing for the proxy modeling using ResFrac® as the back-end simulator is outlined in Figure 125. Provided that the simulation data stored in ResFrac® for the test case serves as the base case, i-Geo Sensing accesses the corresponding ResFrac® folder, parses the simulation data files (i.e., settings text file and input text file in Figure 125) and extracts the ResFrac® entries (i.e., ResFrac® components defining a specific property in the simulation) that define the complete simulation.

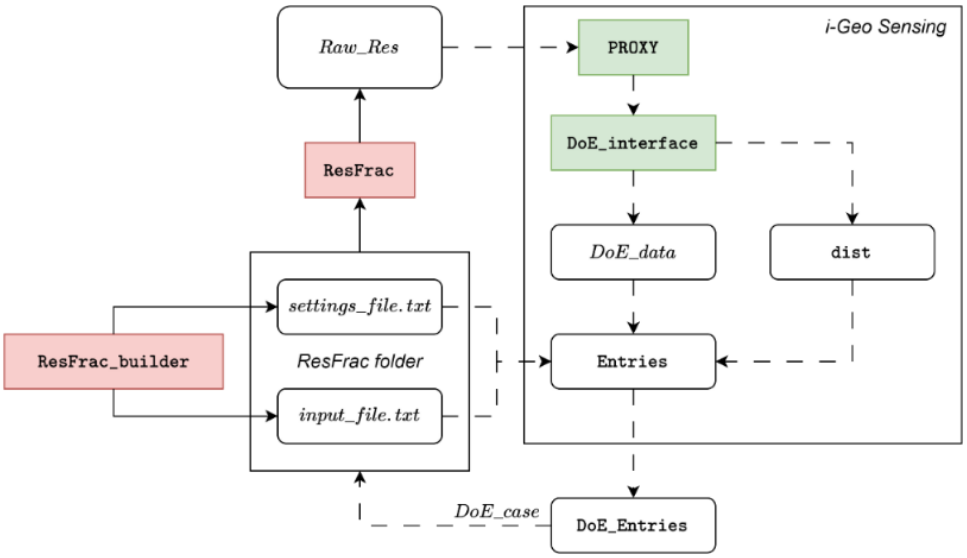


Figure 125: The semi-coupling between ResFrac® and i-Geo Sensing

Per entry defining the simulation, i-Geo Sensing reads the name of the variable representing the entry, its length, and its value (an example is presented in Figure 124). To create realizations via a Design of Experiment study, i-Geo Sensing requests the distributions and written location from the users, in case an entry does not hold a single value (e.g., relative permeability curves). Using the distributions and write locations provided by the users, i-Geo Sensing creates and stores a desirable number of simulation realizations locally. Typically, the entries that hold feasible "Design of Experiment" properties (e.g., fracture propagation properties, fracture confinement properties, layer-based properties, rock physics) are located in the settings text file and not the input text file. Therefore, users are strongly discouraged from selecting the entries that belong to the input text file.

As i-Geo Sensing utilizes the academic license of ResFrac, users need to run simulations in ResFrac® manually. Fortunately, i-Geo Sensing readily performs in-place changes for the Design of Experiment properties, therefore users are not required to use the ResFrac® builder interface to make necessary changes. Users are only required to use the ResFrac® builder interface to import, save, and run/batch run the realizations created in i-Geo Sensing [63, 65].

After all realizations are finished in ResFrac®, users shall provide the directory in which all simulations for the realizations are stored locally (similar to Figure 126). i-Geo Sensing accesses

all realizations and reads the data files that have fracture elements and flow back/production data. By default, the fracture elements are stored in the "RawRes" folder, and the flowback/production data is stored in the "Results" folder. As fracture elements in ResFrac® are recorded at all simulation time steps, sensor data profile(s) are sampled from the fracture elements as described in 6.5 and arranged as a 1D data array. Combined with the Design of Experiment data that is stored previously, i-Geo Sensing generates tabular data that has the columns ordered sequentially as all the DoE data in the order the users request, the sensor data profile(s), and the calibration/history matching data. For test case 6.2, the final tabular data (main training source) has a total of 5572 rows and 57 columns.

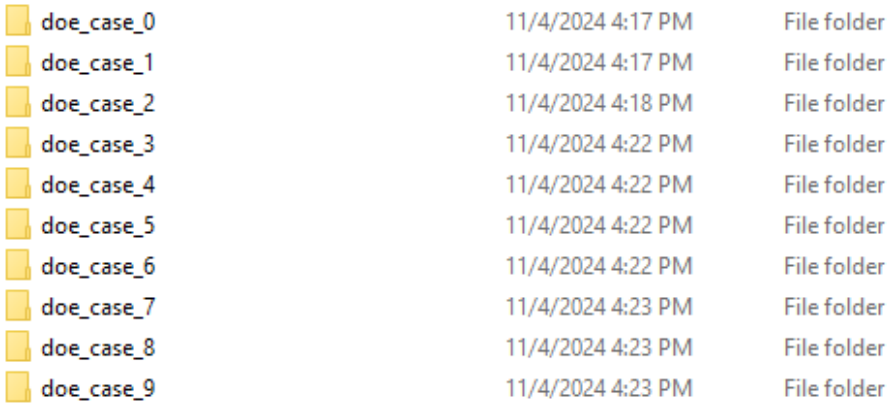


Figure 126: Sample of a directory in which simulation results for realizations are stored

6.9. Supervised machine learning workflow

Figure 127 presents the complete supervised ML workflow in i-Geo Sensing. In Figure 127, the "PDF_{input}" component represents the outcomes from the DoE module, and the grey-shaded components represent the intervention of the Bayes Optimizer engine (detailed in Figure 120 in 6.6). The remaining and central component of this workflow, "Quantile Boosted Trees", is further detailed in 6.9-6.13. As briefly mentioned in 6.2, 6.4, and 6.6, the "Quantile Boosted Trees" component serves as the proxy modeling that backs the supervised ML workflow.

Recalled from 6.7 and 6.8, the proxy data generation process inside i-Geo Sensing returns tabulated dataset(s). Consequently, the ML model that backs the supervised ML workflow in i-Geo Sensing shall have the following characteristics.

1. Highly robustness to tabulated data learning and prediction.
2. Behave consistently in the existence of outliers and abnormality in data.
3. Being resistant to the different scales between input variables (i.e., scaling effect)
4. Highly visualizable and highly explainable.

Under the context that pre-processing for tabulated data is conducted appropriately, classical ML models are proven to outperform DL models. Besides, several classical ML models have a trait to be highly explainable to practitioners, in contrast to DL models which are both not highly efficient and hardly explainable for tabular datasets. Among classical ML models, the model family that satisfies all three criteria above is the Ensembled Boosted Trees algorithm family (e.g., Gradient Boosting Machine, XG-Boost). In i-Geo Sensing, GBM and XGB are implemented as the ML model to back the supervised proxy, and XGB is the default option [68, 69].

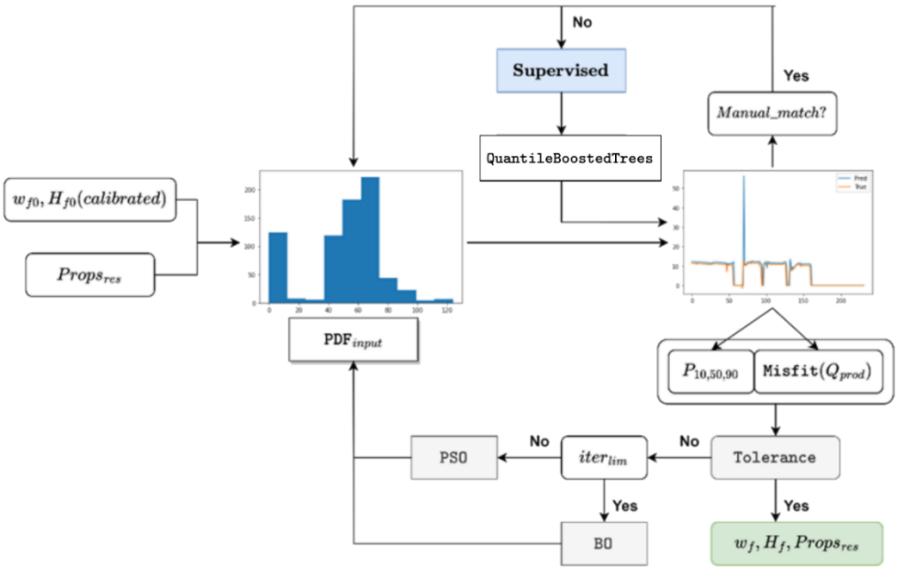


Figure 127: The supervised machine learning workflow

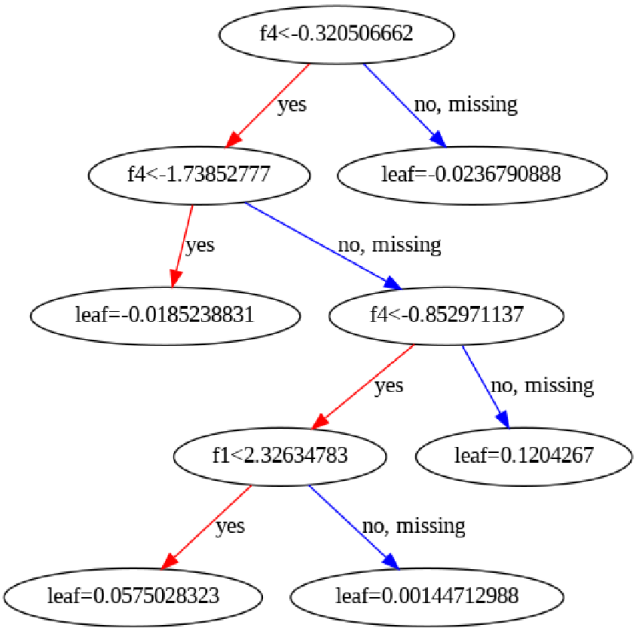


Figure 128: Visual of a decision tree's mechanism

GBM and XGB are both foundational from decision tree models, and an example of a decision tree is presented in Figure 128. Essentially, a decision tree determines internal criteria formed by the input variable(s) ("f1" and "f4" notations in Figure 128) to split toward the predive traits of the output variable(s), until no further split is achievable (i.e., a leaf in Figure 128). Decision tree models are highly interpretable however are prone to unstable behaviors in predictive capabilities. To mitigate the unstable behaviors from decision trees, ensemble tree models are found, e.g., Random Forest (RF), Gradient Boosting Machine (GBM), and Extreme Gradient Boosting (XGB). These ensemble tree models have a higher level of robustness and are proven to extract meaningful traits from tabular datasets. In practice, an ensemble tree model may contain hundreds of sub-trees

and subsequent learning enhancements to perform a better overall learning outcome (i.e., bagging/voting algorithms, as implemented in RF), or to improve the learning outcome from initial "weak" sub-trees (i.e., boosting algorithms, as implemented in GBM/XGB).

Although GBM and XGB have multiple hyperparameters that control their learning performance, i-Geo Senning preserves specific hyperparameter controls internally to avoid the "over-boosting effect" [68]. They include the number of estimators, fraction of sub-samples, learning rate, and the number of boosting rounds (specific to XGB). Except for the restriction above, the i-Geo Sensing code provides both a default hyperparameter selection and a customizable approach for hyperparameter selection from the users.

6.10. ML experimenting in the supervised workflow

Regardless of the ML models that are deployed for prediction, it is never guaranteed that a single training and validation for a model will lead to deployment. To adopt this philosophy, i-Geo Sensing implements ML experimenting in joint with the supervised ML workflow, using the MLflow API [72]. ML experimenting, which is a different concept than DoE mentioned in sub-section 6, refers to the practice of creating, registering, storing, and deploying ML models through the versioning control that is similar to code versioning control. A high overview of the ML experimenting in i-Geo Sensing is provided in Figure 129.

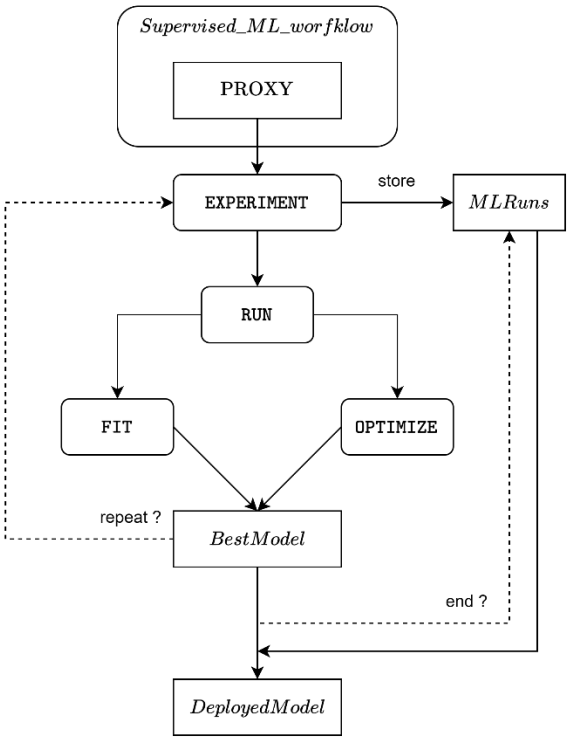


Figure 129: Overview of ML model experimenting design in i-Geo Sensing

In i-Geo Sensing, the code relies on the selected model(s), the provided hyperparameter inputs (if any), and the number of experiments required to run. Since all ML models in i-Geo Sensing are regressors, i-Geo Sensing automatically manages all experiments within a local directory named "mlruns", and default reloads and deploys the ML model with the optimal evaluation metric(s). For example, provided that a proxy in i-Geo Sensing is served for history matching purposes, i-

Geo Sensing reloads the ML model that has its correct serving purpose and the lowest Mean Square Error (MSE).

6.11. Probabilistic and continual-training capabilities of the supervised machine learning proxy

Several ML proxy modeling workflows predict deterministic outputs (i.e., a single output per input). In i-Geo Sensing, all the ML models that are deployed from the supervised ML workflow predict probabilistic values based on the predictive distribution of the output variable(s). Embedded in both GBM and XGB as the i-Geo Sensing central proxy modeling backends, the supervised ML workflow always predicts a lower bound output (the 5% quantile), a mean output (the 50% quantile), and a higher bound output (the 95% quantile). This scheme of predictive ability is possible via the optimization of Pinball Loss, which is formulated as Equation 12 as a conditional loss function [71].

$$L_\alpha = \ (d - f)\ \alpha\ \boldsymbol{if}\ d \geq f, L_\alpha = \ (f - d)\ (1 - \alpha)\ \boldsymbol{if}\ d < f \tag{12}$$

Besides the probabilistic predictive ability as mentioned above, i-Geo Sensing leverages the geo-location data at the received time via a technique named continual training (which occurs during the deployment phase of the supervised ML workflow). The synthetic environment is designed to reflect the reality that geo-location data, once received, becomes ground-truth data that can be used as training data. Therefore, i-Geo Sensing leverages this benefit to perform the following during the deployment phase of the supervised ML workflow.

1. As soon as the geo-location data is received, the ML model(s) inside the supervised ML workflow is deployed for prediction.
2. An immediate computation of the evaluation metric between the prediction and ground-truth response data is performed.
3. In case the evaluation metric exceeds a tolerance, the incident is reported to i-Geo Sensing.
4. The geolocation data and ground-truth response data are joined with previously trained data, and the ML model(s) are re-trained and registered for the upcoming time steps.

Typically, continual training is performed at the early time steps, since the deployed ML model(s) are expected to encounter data drift in reality. As enhancement in continual training progresses, the re-trained ML model(s) shall adapt to reality and improve their dynamic performance at the later time steps. Consequently, the number of incident reports to i-Geo Sensing decreases over the lifetime of deployment.

6.12. Refinement of the supervised proxy inside i-Geo Sensing

In i-Geo Sensing, both fracture calibration and history matching proxies that back the supervised ML workflow are sequentially refined to optimize their prediction validity. The refinement progresses through three stages: the initial deterministic model (first round), the optimized quantile model (second round), and the optimized & continual-trained quantile model (final round). Provided the test environment for the supervised ML workflow in 6.2, Figures 130, 131, and 132 present the refinement for the fracture calibration proxy which predicts the BHP as an output variable from the following input variables: geo-location data received time and the sensor data profiles. Recalled from 6.7, the total number of input variables is 51.

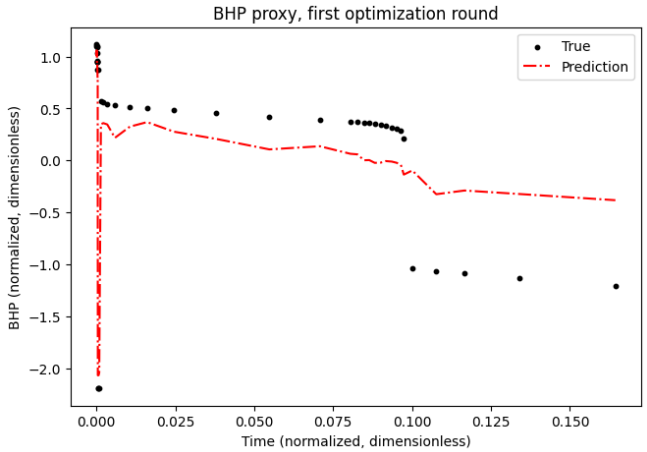


Figure 130: BHP proxy, first-round refinement

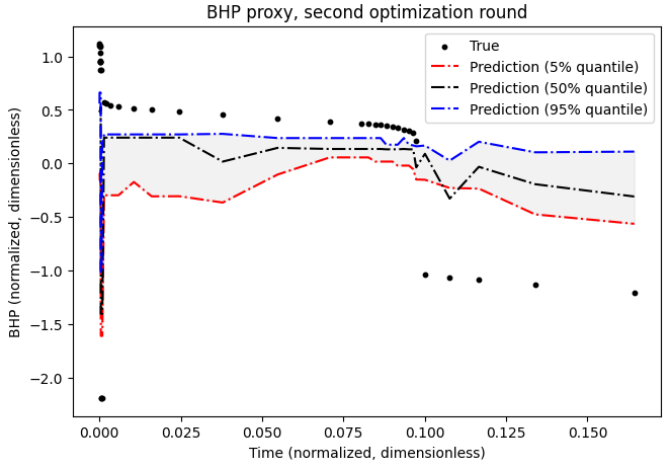


Figure 131: BHP proxy, second-round refinement

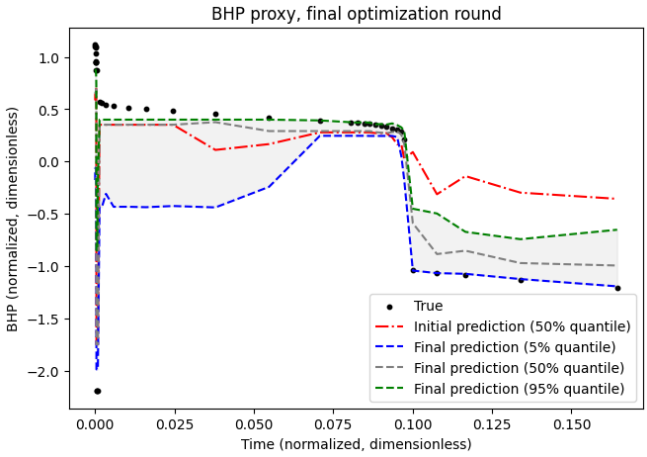


Figure 132: BHP proxy, final-round refinement

In Figures 130-132, the x-axis presents the time (dimensionless), and the y-axis presents the BHP (dimensionless). i-Geo Sensing automatically pre-processes all input and output variables before

training any ML models, commonly via standardization/normalization/scaling methods, Henceforth, it explains the notation “normalized” and the y-axis scale provided in Figures 130-133. Figures 130-133 validate that the supervised ML workflow progressively improves to follow the physical behavior of the output variable. Compared to Figure 130, Figure 131 demonstrates that the quantile-loss ML model in i-Geo Sensing (specifically in this test, XGB) narrows the differences between the ground-truth output data and the prediction confidence (the 95% confidence in Figure 131 at the early time). Compared to Figures 130 and 131, Figure 132 demonstrates that continual training brings benefits later in which there is a drastic drop in the output variable (the 5% confidence in Figure 132).

The evaluation metric, MSE, is progressively reduced through the optimization rounds from 0.0221 to 0.01382 and eventually to 0.0052 for the first, second, and final rounds, respectively.

Figures 133, 134, and 135 present the refinement for the history matching proxy which predicts the oil production rate as an output variable from the following input variables: BHP (predicted from the fracture calibration proxy) and the DoE parameters. Recalled from 6.7, the total number of input variables is 5.

In Figures 133-135, axes have similar representations as Figures 130-132, and similar observations are deducted from them compared to Figures 130-132. Compared to Figure 133, Figure 134 demonstrates that the quantile-loss ML model in i-Geo Sensing (specifically in this test, XGB) narrows the differences between the ground-truth output data and the prediction confidence (the 95% confidence in Figure 134 at the early time). Compared to Figures 130 and 131, Figure 132 demonstrates that continual training brings benefits later in which there is a drastic increase in the output variable (the 95% confidence in Figure 135 at the later time). The 95% confidence is greatly improved in the final round refinement, and this improvement covers the failure to predict the oil rate at the later times (compared to the previous refinement rounds).

The evaluation metric, MSE, is progressively reduced through the optimization rounds from 0.16609 to 0.17814 and eventually to 0.01 for the first, second, and final rounds, respectively.

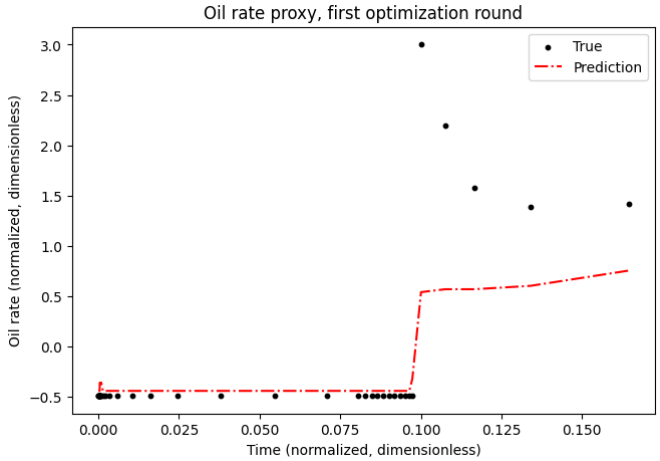


Figure 133: Oil rate proxy, first-round refinement

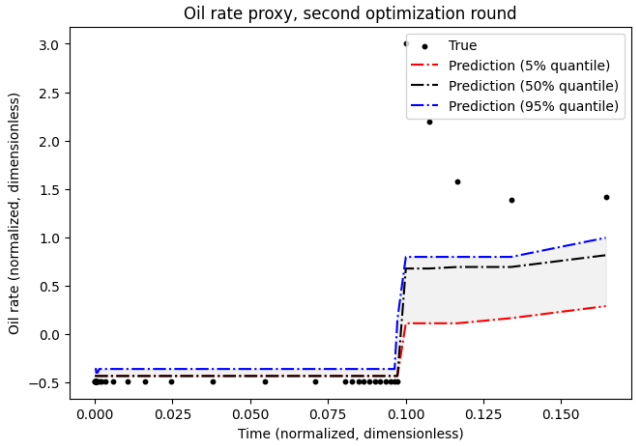


Figure 134: Oil rate proxy, second-round refinement

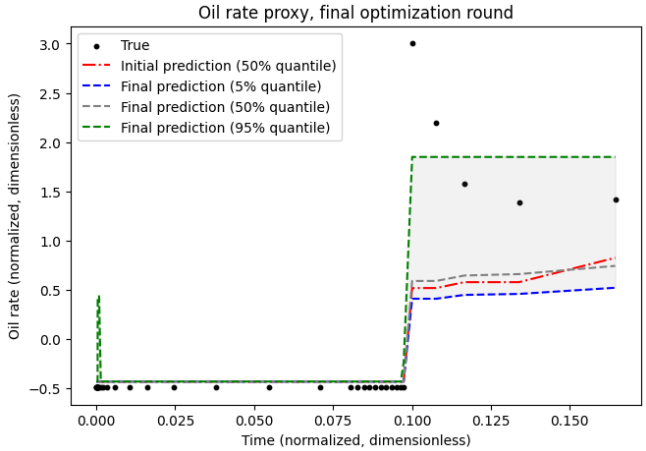


Figure 135: Oil rate proxy, final-round refinement

6.13. Supervised workflow explainability

As mentioned in 6.9, GBM and XGB are implemented as the backbone of the supervised module in i-Geo Sensing. To exceed the explainable capability of these models in i-Geo Sensing, Shapley Additive Explanation (referred to as SHAP [70]) is embedded as the additional criteria. In i-Geo Sensing, there are two levels of mode explainability, as outlined in Figure 136.

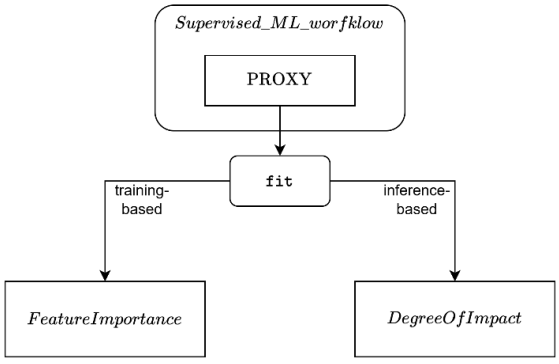


Figure 136: Different levels of model explainability in i-Geo Sensing

The first level of explainability is the training-based level [70]. Since GBM and XGB are used in i-Geo Sensing, this level of explainability is directly extracted via the ranking plot for Key Performance Indicators (presented in Figures 137a and 137b for the fracture calibration task). In Figures 137a and 137b, the x-axis presents the absolute value of an input variable's weight to the decision by the boosted trees, and the y-axis presents the names of all input variables. Both Figures 137a and 137b indicate that time is a dominant factor, however, this phenomenon is observable in case the sensor data profile(s) are not used.

Henceforth, the interest in the model's explainability comes from the other input variables, i.e., the variables forming the data sensor profile(s). The input variables for the data sensor profiles in i-Geo Sensing are encoded in the format of the dimension name followed by the resolution interval number. In Figures 137a and 137b, the encoding means that the z-dimension data is used (and as described in 6.3, this dimension discloses information about the fracture aperture). Although there is not a clear pattern for the base model (Figure 137a), the pattern in Figure 137b is insightful. The top-ranked profile variables are z_0 and z_1, indicating that the most-left edge of the fracture contributes the highest impact to the BHP. The second-top-ranked profile variables are z_{20}, z_{17}, z_{16}, z_{23}, z_{21}, z_{31}, and z_{35}. These profile variables are located approximately in the middle of the propagated fracture, indicating that the central area of the fracture contributes the second-highest impact to the BHP.

Albeit different refinements (as in 6.9) may change the order of the resolution interval indexes slightly, this observation complies with the fracture propagation physics. At the beginning of propagation, the edges of the fractures tend to open because of an increase in BHP. During the middle-late propagation, the central area of the fractures, in which the proppant is primarily settled, holds the fracture pressure and eventually the BHP.

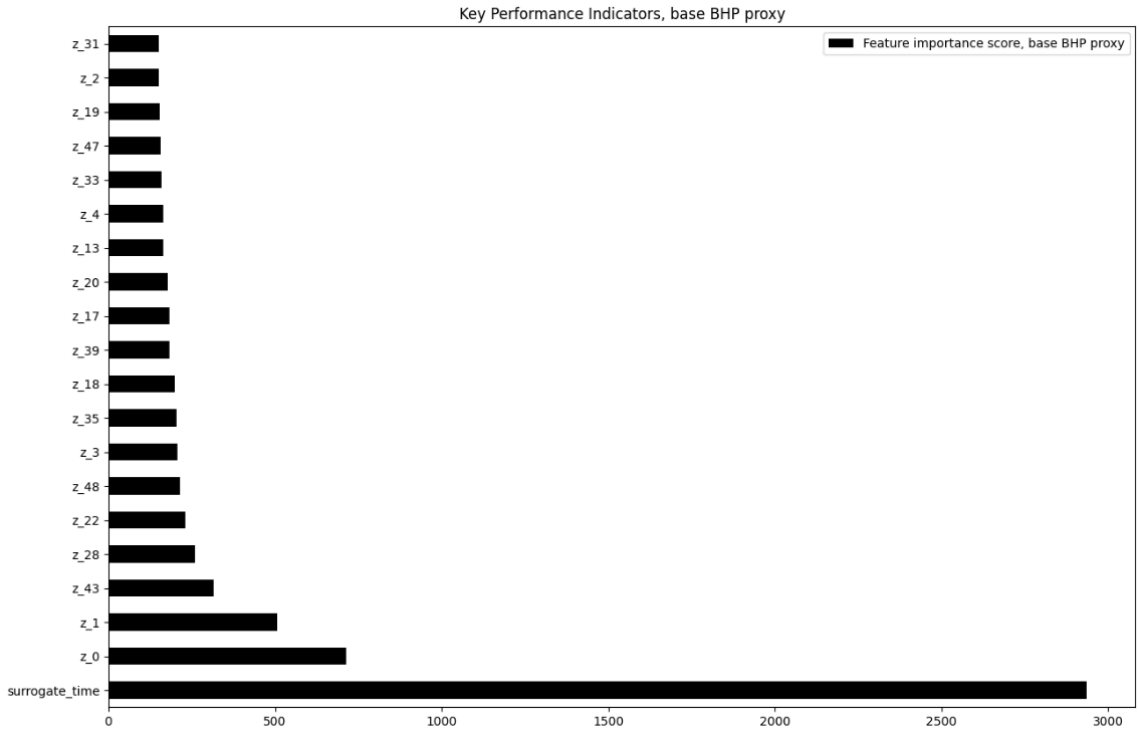


Figure 137a: Key Performance Indicators, the base BHP model

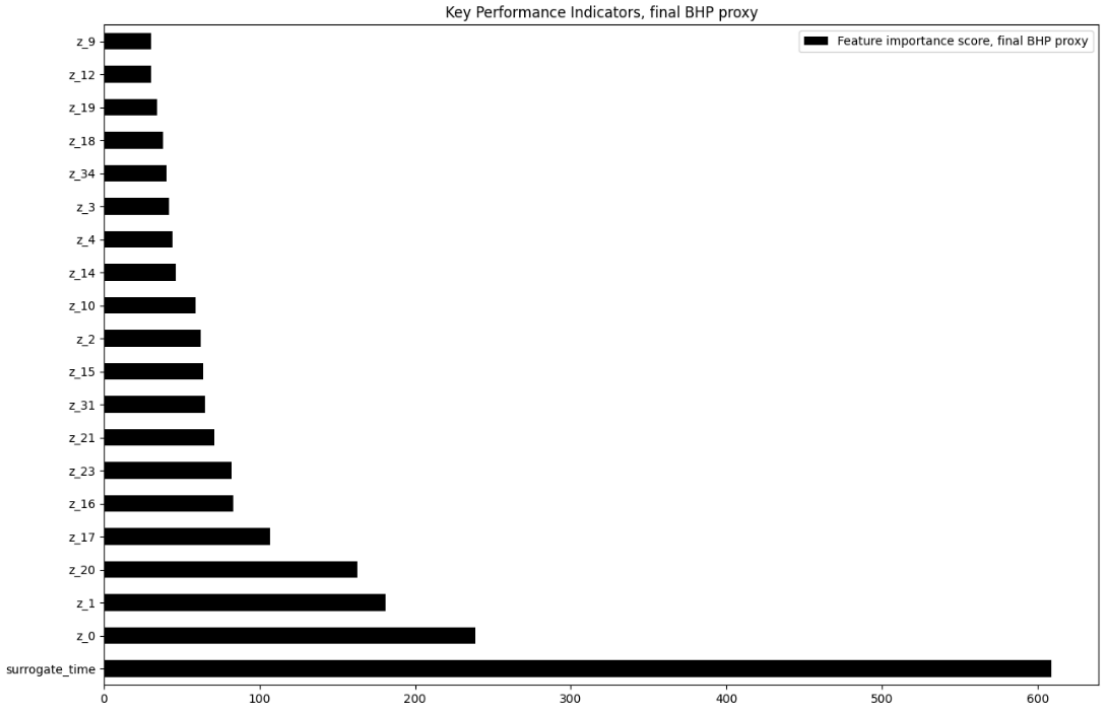


Figure 137b: Key Performance Indicators, the final BHP model

Figures 138a and 138b present the Key Performance Indicators for the model that is responsible for the history-matching task). The axes and their representations in Figures 138a and 138b are similar to Figures 137a and 137b. For the history-matching task, the model still complies with the physics during the well production phase, however, the explanation is more straightforward compared to the fracture-calibration task. BHP plays a major role in controlling the oil rate after the proppant injection is finished and the well is no longer an injector. Furthermore, the relative permeability variables contribute more than the fracture propagation variable during production. Therefore, the ranking for the variable "relative fracture toughness" is lower compared to the relative permeability variables, and this observation is reflected better in the final model (Figure 138b, in which relative fracture toughness ranked the least important).

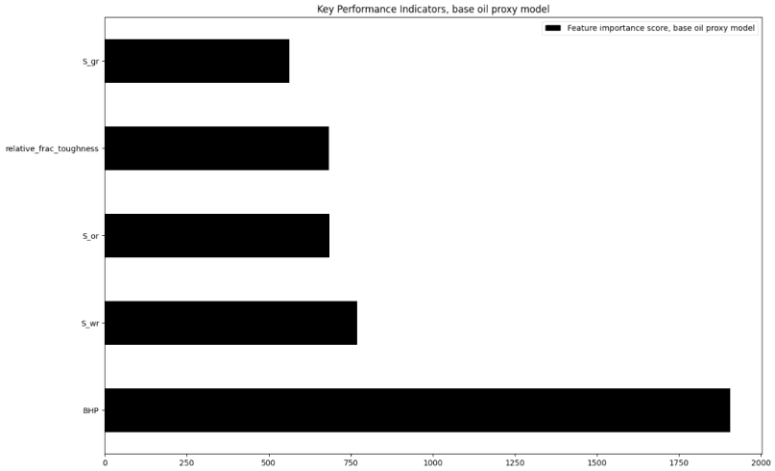


Figure 138a: Key Performance Indicators, the base oil rate model

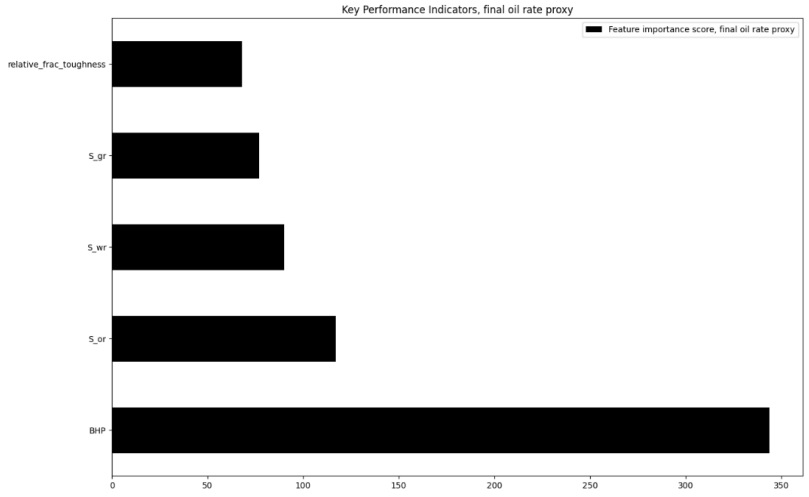


Figure 138b: Key Performance Indicators, the final oil rate model

The second level of explainability is the inference-based level [70]. Different from the training-base level which extracts the feature importance after the model is trained, the inference-based level extracts the influence direction that a variable embeds into the model when the model is deployed for a blind test or a new input sample. The influence direction in the context of i-Geo Sensing includes both the direction and the magnitude.

i-Geo Sensing provides the interpretation for the second-level explainability through two types of plots: the bee-swarm plot (Figures 139a and 140a) and the waterfall plot (Figures 139b and 140b). Figures 139a and 139b present the second-level explainability for the fracture calibration proxy. Per the definition of the second-level explainability, both the fracture calibration and history-matching proxies are readily trained using the datasets in 6.7 and are now deployed for the simulation data in the synthetic environment defined in 6.2.

In Figure 139a, the x-axis presents the mean SHAP impact value, and the input variable contributes to the model's predictive decision of the output variable (in this case, BHP and oil production rate). A similar dominance of the time variable is observed in Figure 139a, as this is readily observed in Figures 137a and 137b. A closer look at the "Feature value" color map, the direction of impact for the time variable is explained as follows. When the time value is small (blue dots), it positively influences the BHP, and the magnitude of the influence is moderate. When the time value is high (magenta dots), it negatively influences the BHP, and the magnitude of influence is lower. A reflection of the reservoir dynamics in the synthetic environment discloses a similar description. Within the synthetic environment, at the early times, a change in time value determines a moderate change in the BHP (since proppant injection and fracture propagation occur at the early times). At later times when the well enters the producer mode, a smaller or larger change in time value fluctuates the BHP slightly. Figure 139a, henceforth, highly correlates with Figure 108 in which the BHP plot from ResFrac® simulator is presented.

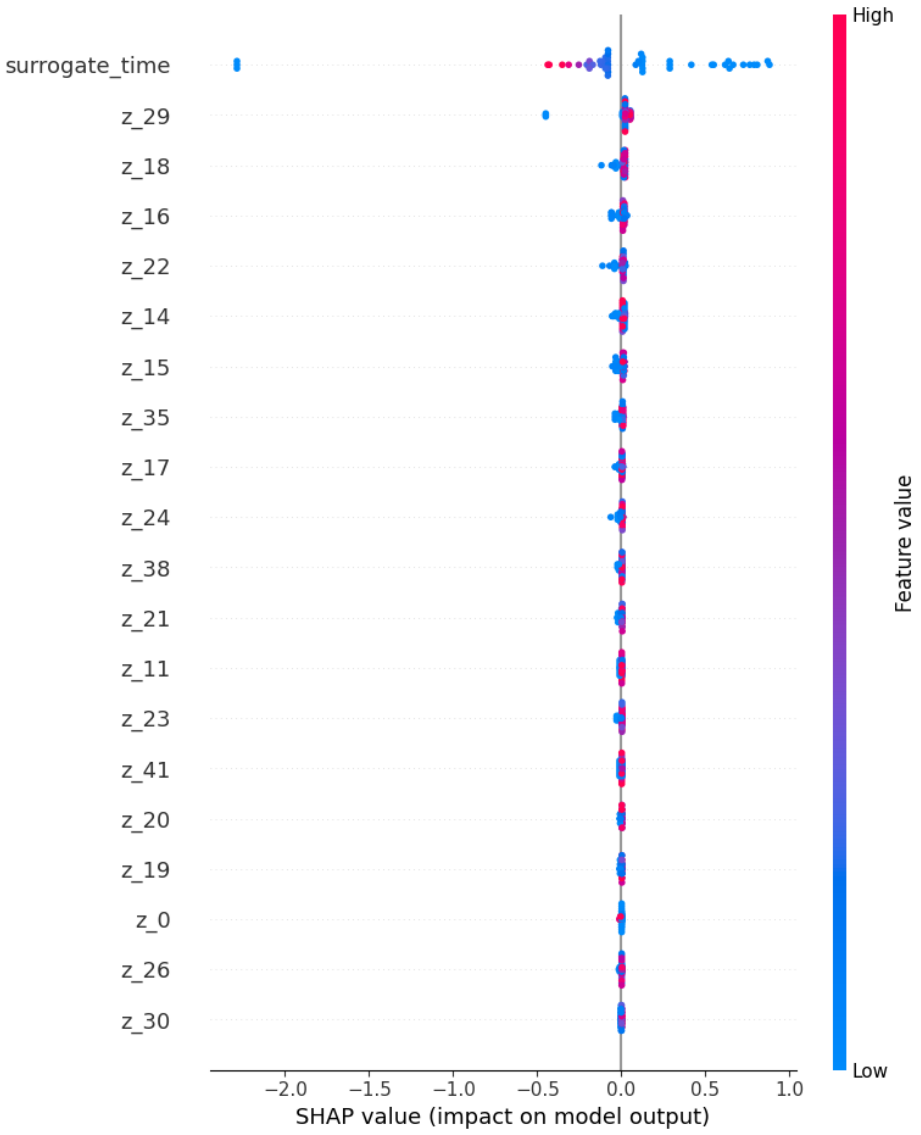


Figure 139a: – SHAP's bee-swarm plot for the BHP model

Figure 139b provides a quantitative model inference for the fracture calibration task. The gist of Figure 139b reveals the contribution of each input variable to the expectation shift of the model's prediction from its baseline prediction (reflected by the expected value "E(x)" in the x-axis). Figure 139b emphasizes the importance of the fracture central area (i.e., the input variables z_{29}, z_{14}, and z_{16}), as this area almost balances the expectation shift of the model's prediction from its baseline to the expectation shift caused by time.

Similar interpretations can be deducted from Figures 140a and 140b, which present the second-level explainability for the history-matching proxy. An additional insight from Figures 140a and 140b is that, under the synthetic environment, the DoE parameters play an almost insignificant role in the predictive behavior of any models. Henceforth it is strongly suggested that, in the case a user requests these DoE parameters to the i-Geo Sensing, he/she may need to re-consider a better selection to interpret the dynamics of the studied reservoir.

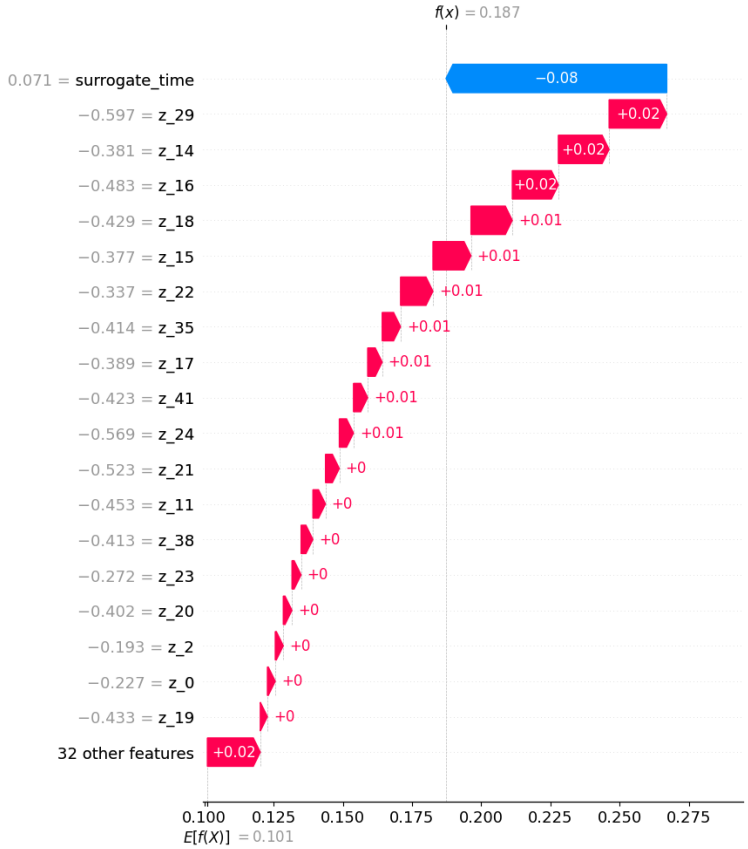


Figure 139b: SHAP's waterfall plot for the BHP model

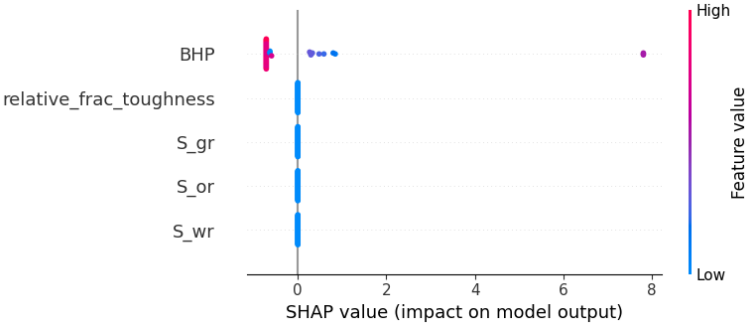


Figure 140a: SHAP's bee-swarm plot for the oil rate model

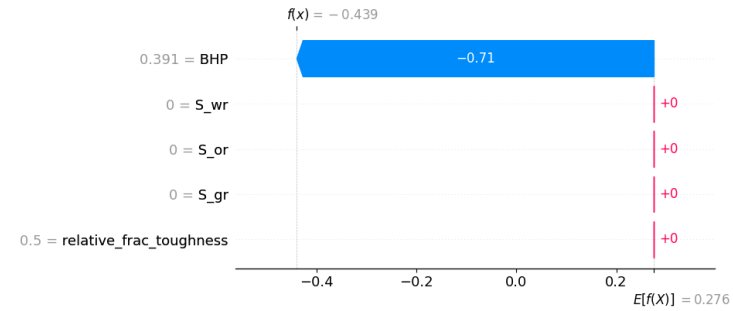


Figure 140b: SHAP's waterfall plot for the oil rate model

7. SUCCESSFUL FINAL FIELD TESTING (PILOT) IN THE EOG RESOURCES OPERATED WELL IN NEW MEXICO

Summary

This section of the report details the successful field testing of a novel "Smart Microchip Proppants" technology for high-precision diagnostics of hydraulic fracture networks. Developed under Department of Energy award DE-FE0031784, the project was led by the University of Kansas in partnership with UCLA, MicroSilicon Inc., and field trial operator and cost share provider EOG Resources Inc.

The pilot test, conducted in August 2024 in Lea County, New Mexico, validated the ability of the Smart Microchip technology to provide unprecedented high-resolution insights into proppant placement and fracture mapping. The success of this field trial marks a major milestone in the development of next-generation direct fracture diagnostic techniques and opens transformative possibilities for characterizing and managing subsurface systems.

7.1 Pilot Testing Details

The "Smart Microchip Proppants" technology leverages Smart Microchips that are injected during a small-scale hydraulic fracturing. These engineered Microchips are designed to withstand the harsh downhole environment. The built specialized downhole tool is later deployed to remotely power the embedded microchips and receive their transmitted signals. This enables detailed mapping of proppant placement and fracture geometry at a resolution of one foot, an unparalleled level of detail compared to conventional fracture diagnostic methods.

Field Testing Methodology The field trial was executed in the Capella BOP Fed #1 well operated by EOG Resources in Lea County, New Mexico. Fig.141 is the satellite imagery of the pilot testing site.

The testing process involved the following key steps:

1. Site Preparation: The well was deepened via a workover rig to the target stimulation interval of 8765'-8800' in preparation for hydraulic fracturing and Smart Microchip injection.

The workover rig was deployed on August 13, 2024, to prepare the well for the trial. EOG's operation team performed several crucial steps, including pulling tubing, deepening the well by drilling out the shoe, and running a packer and work string. The open hole was drilled with a 4.75" bit, targeting the interval from 8765' to 8800' for hydraulic fracturing and smart Microchip injection.

Figure 141(a) and (b) illustrate the wellbore diagram before and after deepening for the pilot testing.

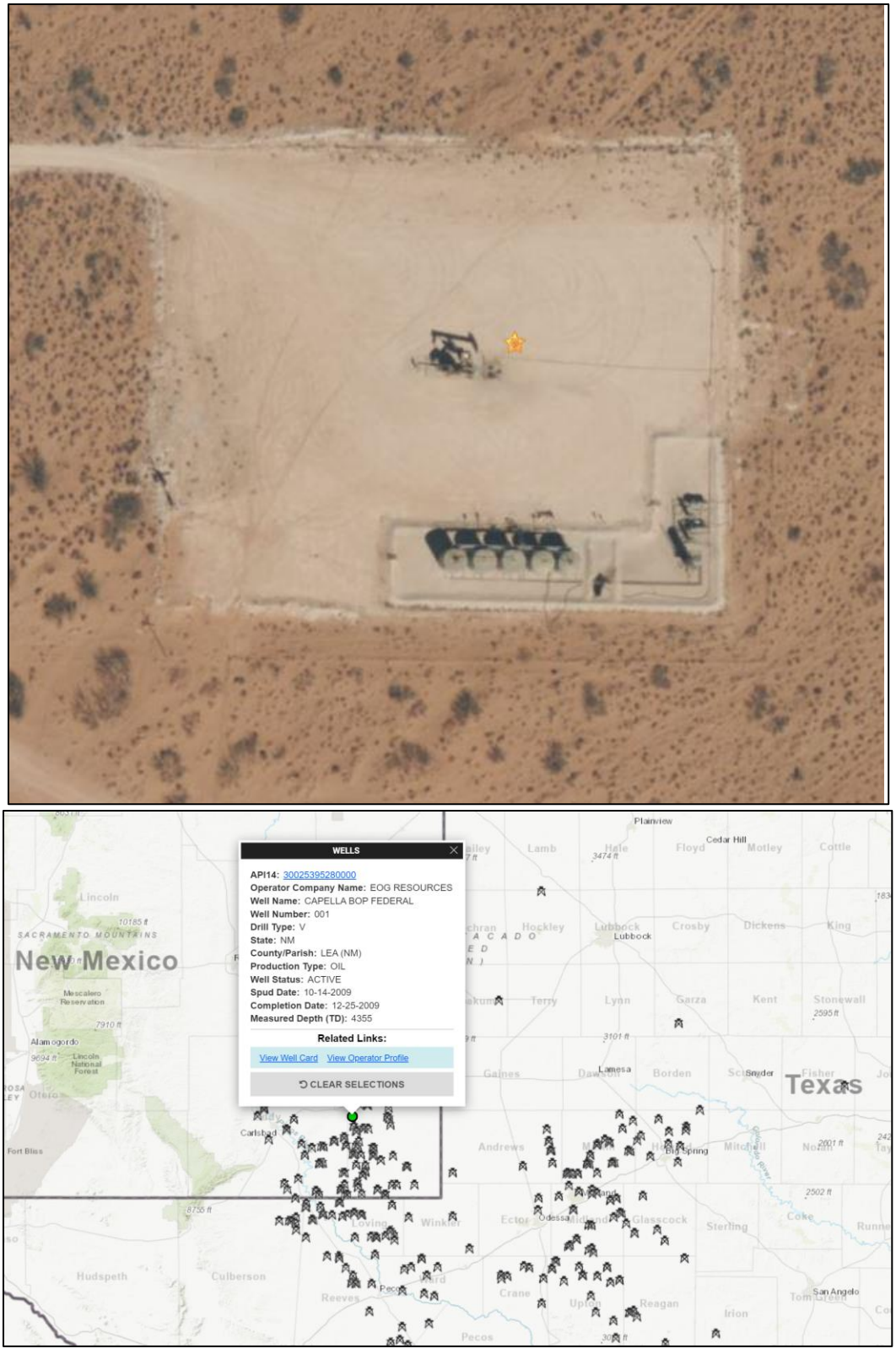


Figure 141: satellite imagery of the pilot testing site (Capella BOP Fed #1)

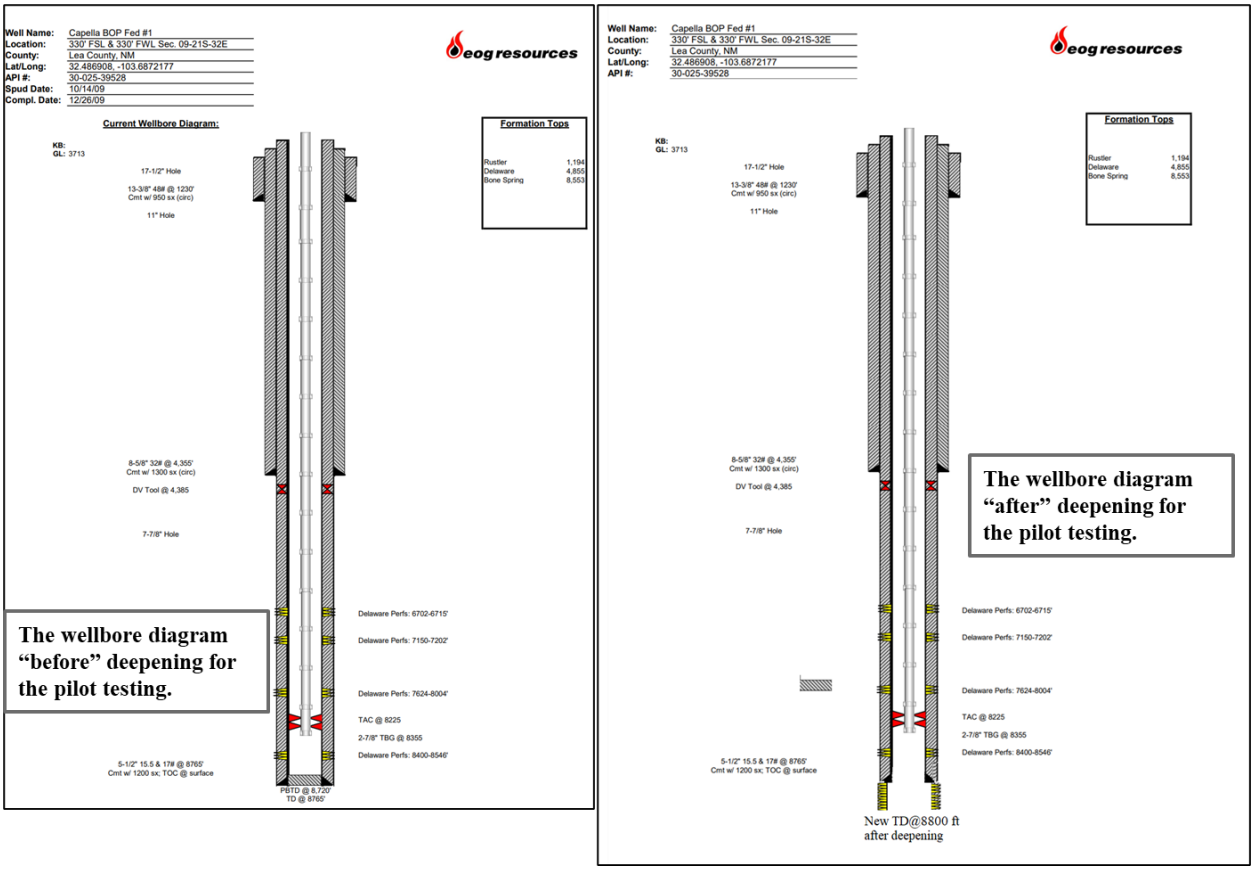


Figure 141: (a: left) and (b: right) - the wellbore diagram before and after deepening for the pilot testing.

2. Small-scale hydraulic fracturing and Smart Microchip Injection:

Hydraulic fracturing was conducted on August 19, 2024. Over 200 lab-verified Smart Microchip samples were injected during the stimulation, which involved pumping 56 barrels of fluid at rates up to 2.2 barrels per minute. Breakdown pressure was approximately 3700 psi, and the well was shut in at 3300 psi.

To increase robustness against misalignment during deployment, two types of Printed Circuit Boards (PCBs) were utilized:

1. First Version: Designed and injected for general alignment robustness.
2. Second Version: Featured an upgraded Surface-Mount Device (SMD) inductor serving as the coil.

The resonance frequencies of the microchips, specifically at 13.56 MHz and 40.68 MHz, were verified using a Vector Network Analyzer (VNA). Prior to deployment, all samples were validated using a spectrum analyzer and signal source to ensure optimal performance and reliability.

The epoxy encapsulating the microchips was custom ordered to withstand high temperatures and high pressure, ensuring durability under the well conditions. Over 100 samples of each Smart Micochips PCBs version were injected into the formation.

Injection Process Details:

- The injection rate was gradually increased to 2.2 barrels per minute (bpm).
- A total of 56 barrels (bbl) of fluid was pumped, which included 5 bbl more than the calculated well displacement.
- The injection pressure was carefully controlled, avoiding a maximum surface pressure of 4,000 psi to ensure that the bottom hole pressure remained below 9,000 psi.

The details of the HF job and operation are illustrated in Figure 142.

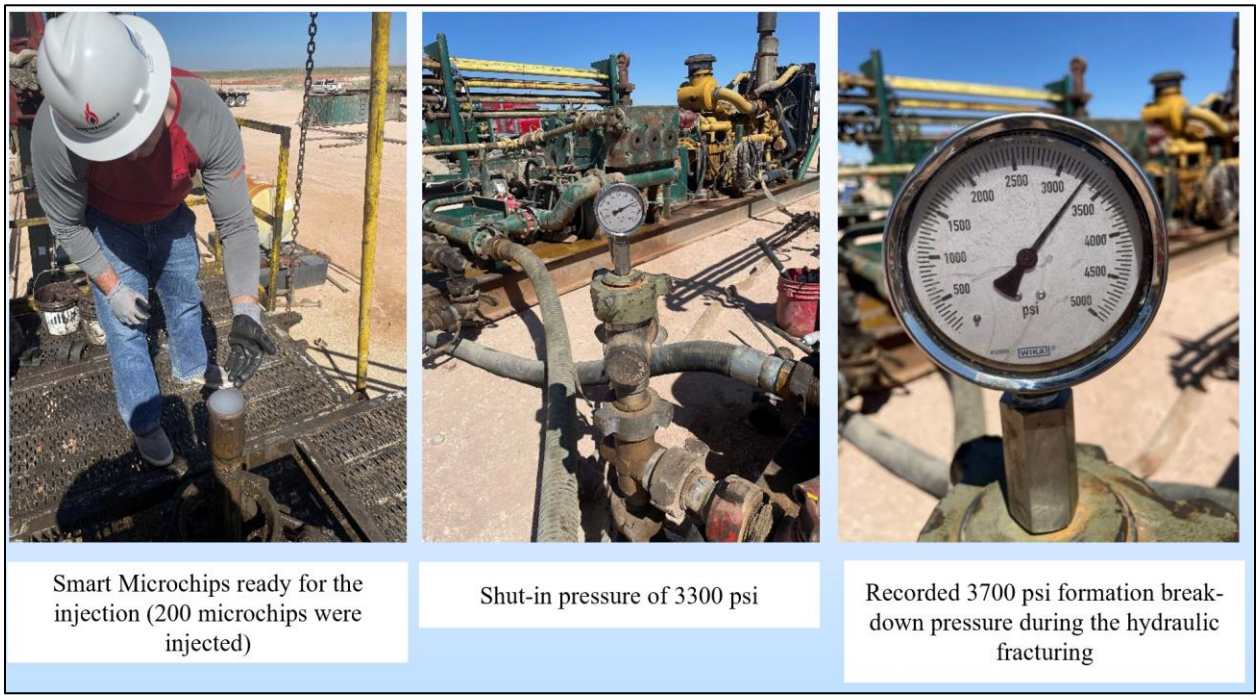


Figure 142: Smart Microchips ready for the injection (200 microchips were injected) (left), Shut-in pressure of 3300 psi (the middle), and Recorded 3700 psi formation break-down pressure during the hydraulic fracturing (right)

3. Extended Shut-In Period: To rigorously test microchip resilience, the well was left shut-in for ten days before initiating chip activation and data collection on August 28, 2024.

On August 19, 2024, the hydraulic fracturing operation was successfully completed, and Smart Microchips were injected into the formation. Following this, a ten-day silence period was implemented to allow the well and the system to stabilize fully before proceeding with the activation of Smart Microchips. This silence period was a deliberate measure to ensure that the

microchips were exposed to the harsh downhole environment for a sufficient duration, providing an opportunity to validate their functionality and durability.

The decision to delay the activation of the Smart Microchips was made to address potential concerns regarding their long-term survivability. Activating the microchips immediately after injection would not have provided a complete demonstration of their ability to withstand extreme conditions over time. By scheduling the activation for August 28, 2024, the team ensured that the microchips endured realistic downhole conditions, which allowed for a more robust evaluation of their performance and resilience.

4. Downhole Tool Deployment: A custom-designed downhole tool, encased in 3-5/8" tubing, was deployed on slickline to TD at 8800 ft. The tool was raised and lowered in three 50 ft sweeps to activate and collect data from the Smart Microchips.

4.1 Preparation Phase

The preparation for the deployment involved assembling and configuring key components to ensure a smooth operation. The chassis was securely loaded into a 3 5/8” pipe, a critical step in preparing the downhole tool for deployment. The lower crossover was torqued into place using spanner wrenches, providing a firm and reliable connection. At the top end of the assembly, a small sub equipped with threads and O-rings was installed, ensuring a final pressure seal. This sub also featured a thread profile supplied by EOG, designed for easy attachment to the rope socket on the slickline unit.

Figure 143 illustrates the details of the downhole tool preparation for the field deployment.

Figure 143: The chassis was loaded into the 3 5/8” pipe (left), and the lower cross-over was torqued in place with spanner wrenches. (the middle), and at the top-end, we provided another small sub with threads and o-rings that can be torqued in place and provide the final pressure seal. At the top of that sub is a thread whose profile was provided by EOG for attachment to the rope socket on slickline unit. (right)

4.2 Transportation to the Site

The well site was located approximately 30 miles east of Carlsbad, New Mexico, accessible only by a dirt road. The journey to the site was challenging due to the poor condition of the road, which caused several vehicles to become stuck and require towing on two separate occasions. Despite these difficulties, the team successfully reached the wellhead and began the setup process (Fig.144).

Figure 144: Downhole tool transportation to the wellsite

4.3 Setup and Assembly

Upon arrival, the slickline crew initiated their preparations by verifying the rope socket thread and mating components on the MicroSilicon tool, ensuring all connections were secure and properly aligned. The downhole tool was successfully activated during this phase.

However, a minor challenge arose when it was discovered that the lubricator on-site was designed for a 2 7/8” tool, whereas the deployed tool had a diameter of 3 5/8”. After consulting with the EOG operations team, it was determined that the well could remain open for a brief period without risk, allowing the lubricator to be adjusted for proper fit. The lubricator was then carefully placed above the tool, which was lowered into the well and securely fastened (Fig.145).

Figure 145: Downhole Tool Setup and Assembly Onsite

4.4 Well Deployment

The deployment process began with the tool being lowered into the well using the lubricator. As the lubricator was opened, the wellhead emitted a noticeable "hiss," indicating air was being drawn into the well. This sound confirmed the safety of proceeding with the deployment.

Once the tool was running, the crew monitored the wire and periodically checked the pressure gauge to ensure there was no buildup of internal pressure. The descent slowed as the tool approached the total depth (TD) of 8,800 feet, with the weight on the wire signaling that the tool had reached the bottom (Figures 146 and 147).

Figure 146: Well deployment of the downhole tool

Figure 147: Well deployment of the downhole tool

4.5 Activation of Smart Microchips and Signal Reception

The Smart Microchips were activated successfully during the deployment process. The tool was maneuvered in the well with sweeps of 50 feet up and down, each lasting approximately 12 minutes. After completing three such sweeps, the crew pulled the tool out of the well and disconnected it. Upon inspection, the tool's bottom was found to have some mud, but the fiberglass antenna housing remained completely undamaged, confirming the structural integrity of the tool (Figure 148).

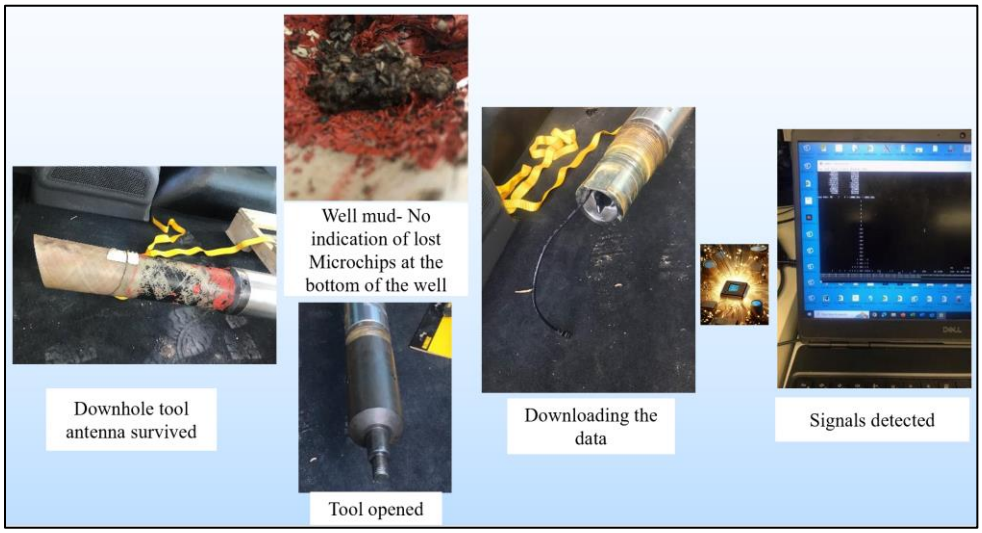


Figure 148: Downhole tool after the operation is complete.

Further analysis revealed no Smart Microchips present in the mud, suggesting that all the microchips had successfully entered the fractures as intended. When the rope-socket joint was opened, a page of memory was successfully read, demonstrating that signals had been recorded during the deployment. Data was downloaded from the tool, and the signal reception was verified, confirming the effectiveness of the operation.

On the activation date, August 28, 2024, the Smart Microchips transitioned from a zero-energy, inactive state to a fully powered and operational state (Figure 149). During this activation process, the microchips were remotely powered and began transmitting signals as designed. This transition marked a significant milestone, as it demonstrated the microchips' ability to function effectively in challenging environments while maintaining their structural integrity and operational capabilities. The ten-day silence period and the subsequent activation of the Smart Microchips successfully validated their robustness and reliability. This methodical approach ensured that the microchips could perform as intended under demanding conditions, addressing any potential concerns about their functionality and long-term durability.

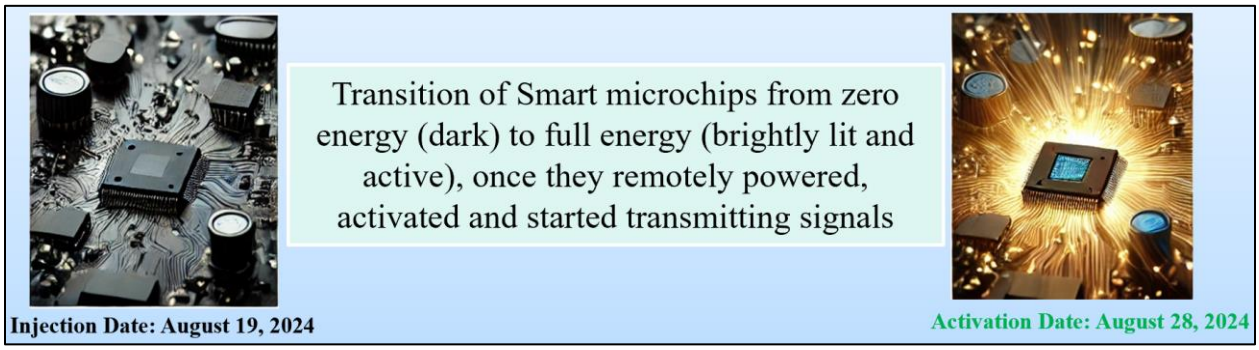


Figure 149: Transition of Smart Microchips from Passive to Active State: Remote Power Activation and Signal Transmission

7.2 Smart Microchip Signal Reception and Analysis

7.2.1 Successful Signal Reception from Smart Microchip Proppants in the Field

The activation of Smart Microchips in the field yielded significant results, confirming their successful placement in the created hydraulic fractures. Signals were detected during multiple sweeps of the downhole tool, validating the robust performance of the microchips under field conditions. Notably, the microchips were observed at the same location during some sweeps but not consistently in every sweep. This variation was attributed to the antenna's directional limitations and potential tool rotation during operation.

A spectrogram analysis revealed three main clusters of signals corresponding to three primary hydraulic fractures. The y-axis represented frequency, while the x-axis denoted time, highlighting the distribution and strength of the detected signals (Figure 150).

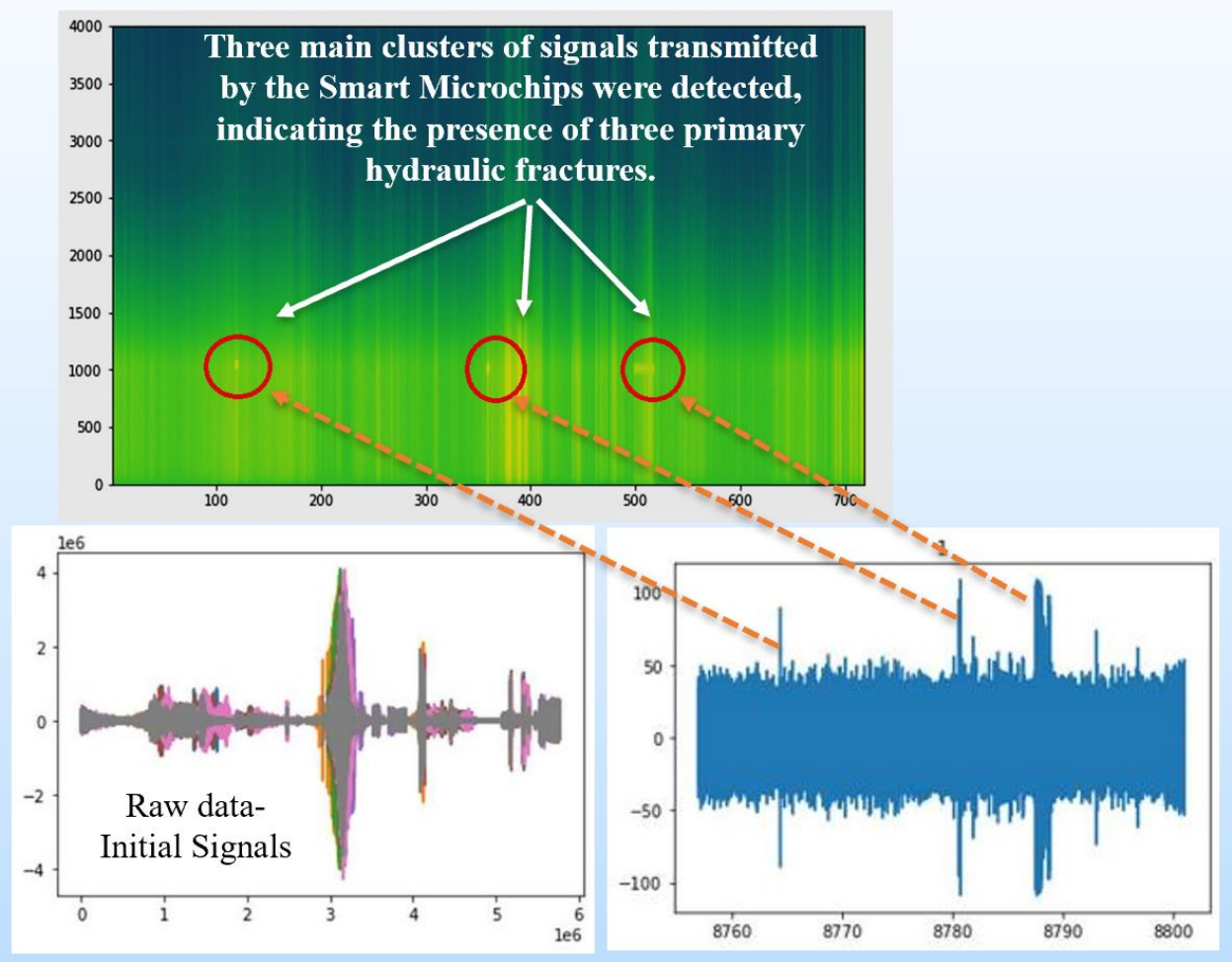


Figure 150: The Most Exciting News: We Received Signals!!!. Three main clusters of signals transmitted by the Smart Microchips were detected, indicating the presence of three primary hydraulic fractures.

7.2.2 Interpreted Smart Microchips' Signal Results

The initially filtered signals from the Smart Microchips demonstrated strong correlations across eight sweeps. Each sweep corresponded to a different angular position of the downhole tool, enabling comprehensive 360-degree data capture (Figure 151).

This thorough coverage confirmed the successful placement of the microchips within the hydraulic fractures. The signal strength, indicative of a high concentration of microchips, provided further evidence of their distribution within the fractures.

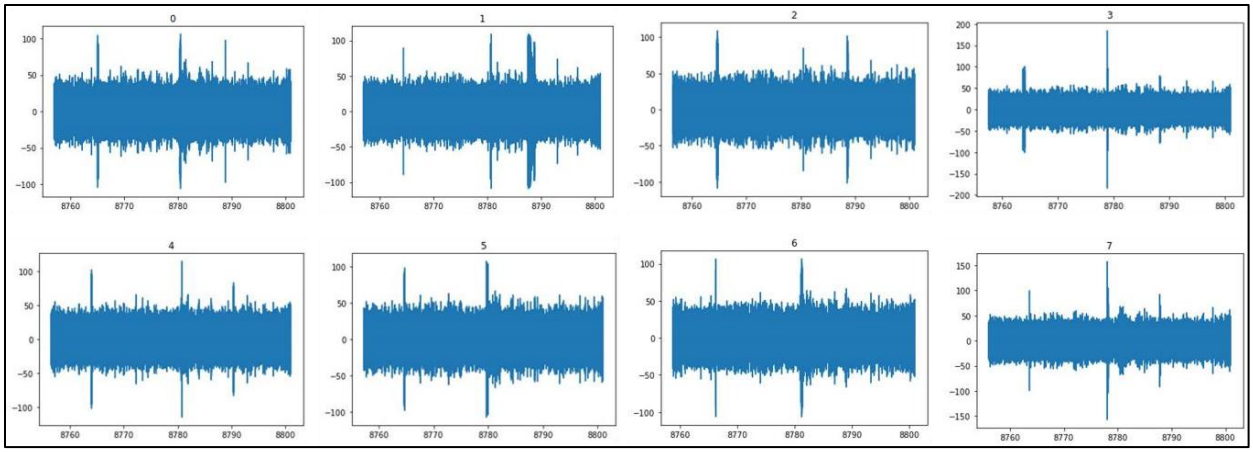


Figure 151: Initially filtered frequency versus depth data (y-axis: Frequency, x-axis: Depth) for all eight sweeps.

The processed signals from Smart Microchips highlight their amplitude versus depth for all eight sweeps (Figure 152). This data demonstrates strong correlations and consistency, confirming the successful activation and functionality of the microchips. The consistent patterns reveal hydraulic fractures with high precision, achieving detailed characterization with an accuracy of just a few feet. This level of detail signifies a breakthrough in high-resolution hydraulic fracture characterization.

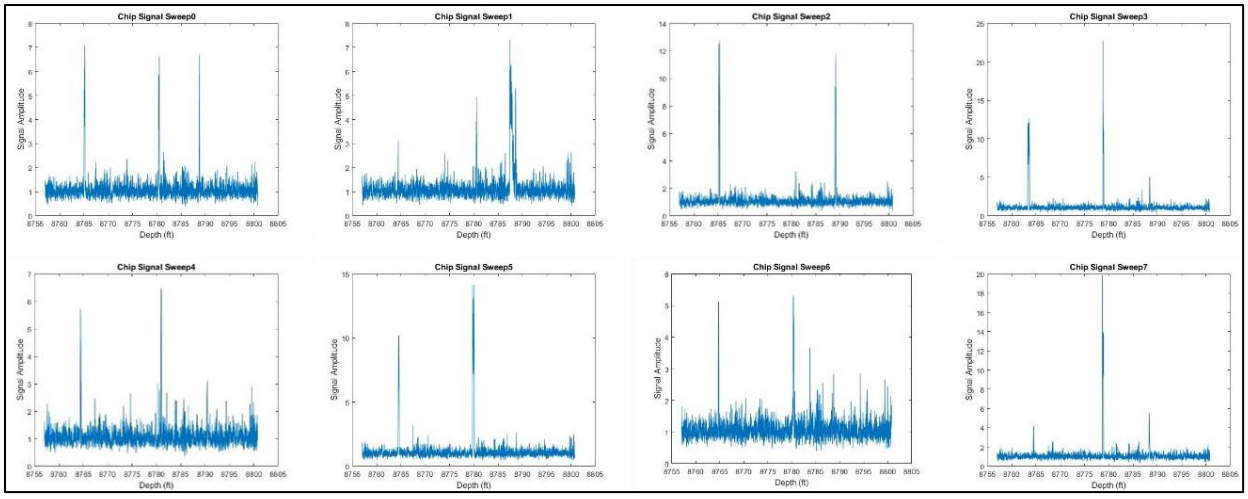


Figure 152: Processed signals amplitude versus depth for all eight sweeps

The processed signals analyzed through the iGeoSensing platform identify key hydraulic fractures. Using Chebyshev Type I low-pass filters, bandwidth frequency analysis, and adaptive depth filtering, three main signal clusters were detected at depths of 8,766 ft, 8,780 ft, and 8,788 ft. These clusters correspond to the primary hydraulic fractures, with the strongest amplitude observed in the third interval, indicating a higher concentration of microchips (Figure 153).

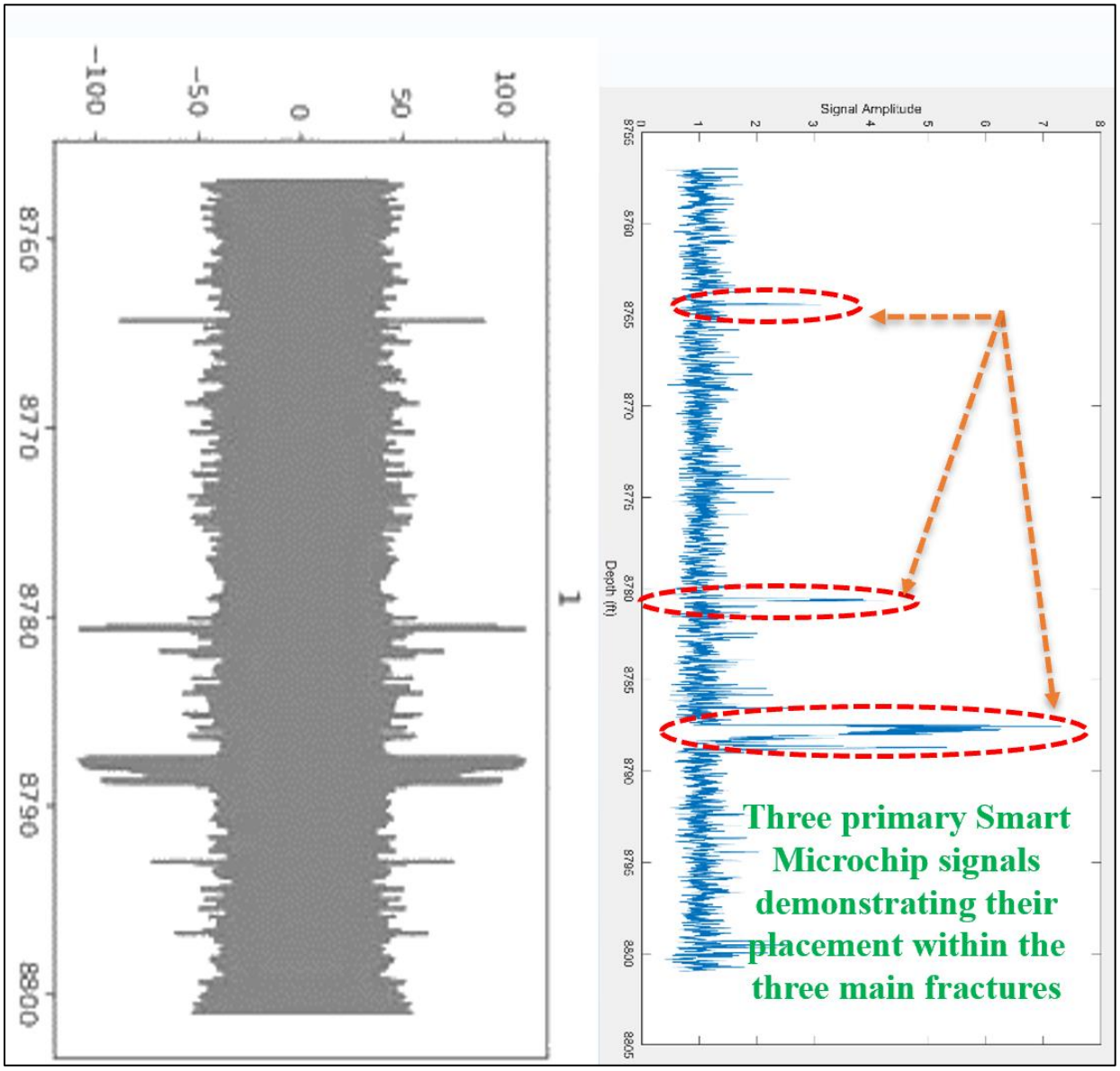


Figure 153: An example of raw and processed signals (Left and right) for one of the sweeps

These fractures are typically represented as homogeneous and simple in most fracture diagnostic tools and models, especially for a few feet stimulated intervals.

These insights are typically difficult for operators to achieve, as indirect fracture measurements often fail to provide this level of high-resolution detail.

Such details could not have been observed or diagnosed previously without obtaining core samples from the hydraulically fractured well which is very unlikely to be obtained in normal operations. However, this is now possible to get this valuable information at less than 1 ft resolutions with our Smart Microchips technology.

To demonstrate the importance of the data generated by Smart Microchips, a "base" high-resolution fracture and numerical simulation model was developed. This model was created using the ResFrac Simulator, seamlessly integrated with iGeoSensing as its backend.

The models were built using a fine-scale grid size of 1x1x1 foot, enabling simulations of a 100x100x100-foot reservoir volume, amounting to one million grid blocks. The fluid model applied to the reservoir was a saturated black oil model, ensuring realistic fluid behavior. The open-hole hydraulic fracture spanned depths between 8,765 feet and 8,800 feet.

The pumping schedule was designed to align with field data, functioning as a mini-fracture operation. This schedule involved injecting a total slurry volume of up to 55 barrels at a rate of 2.2 barrels per minute to deploy the microchips. The hydraulic fracture properties included a fracture half-length (Xf) of 10 feet, fracture conductivity (FC) ranging between 8,236 and 8,312 millidarcy-feet, and fracture height (Hf) of 35 feet.

The simulation accounted for various operational phases, including the mini-fracture, a shut-in period, and field-informed flow-back stages. Critical fracture geometry parameters, such as fracture toughness (K), were made adjustable through iGeoSensing's advanced back-end module, allowing for fine-tuning and precise modeling.

The base model indicated a homogeneous distribution of the stimulated rock volume across the entire treated interval. This is typically the standard output reported by simulation engineers for hydraulic fracturing jobs. However, such models may not fully or accurately capture the complexity of actual hydraulic fractures, potentially leading to significant overestimations or underestimations of the predicted flow behavior.

However, the Smart Microchips signals indicate that fracture growth is primarily restricted to the active and detected depth intervals where hydraulic fractures are generated, with signals detected at approximately 8766 ft, 8780 ft, and 8788 ft. This suggests non-uniform fracture propagation across the entire stimulated interval. A similar phenomenon was observed in another project, where core samples from a hydraulically fractured well revealed heterogeneous fracture initiation. This included both bi-wing and complex fracture networks within a few feet intervals, which typically cannot be characterized using conventional fracture diagnostic tools and are often treated as homogeneous in hydraulic fracturing simulations.

The updated model (Figure 154) leveraged Smart Microchip data to showcase heterogeneous proppant distributions, replacing the simplified homogeneous assumptions of the base model. The transmitted signals from microchips allowed adjustments to the fracture model, creating a more realistic depiction of stimulated intervals. This advanced modeling demonstrates the value of integrating Smart Microchip data for accurate fracture mapping.

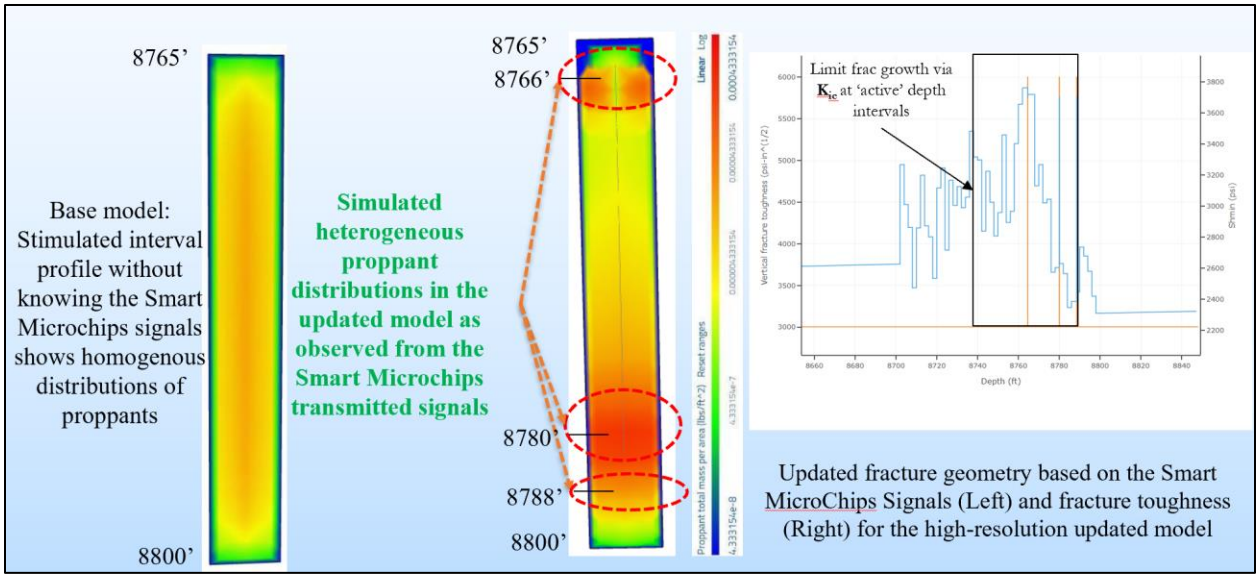


Figure 154: Base model: Stimulated interval profile without knowing the Smart Microchips signals shows homogenous distributions of proppants (left), Simulated heterogeneous proppant distributions in the updated model, as observed from the Smart Microchips, transmitted signals (right), and fracture toughness (right) for the high-resolution updated model

New diagnostic plots generated from the iGeoSensing platform highlighted the differences between the base model (without Smart Microchip input) and the updated model. Dimensionless flowback type curves demonstrated how microchip-derived data significantly impacted fracture characterization. The plots showed that higher initial fracture toughness influenced flowback behavior, with its effect diminishing over time. This diagnostic approach underscores the importance of incorporating advanced data analytics into hydraulic fracture modeling.

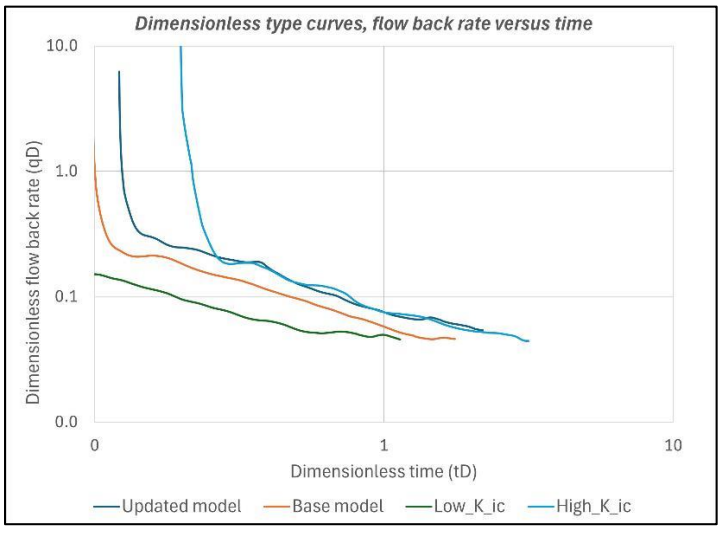


Figure 155: Fracture geometry profiling diagnostic by dimensionless flow back type curves

In summary, the field trial achieved a critical milestone: the successful transmission and reception of data signals from the downhole Smart Microchips. The key outcomes were:

1. Smart MicoChips Signal Detection: Raw signal data revealed three distinct clusters of microchip transmissions, indicating the presence and location of hydraulic fractures at specific depth intervals. The consistent signal detection across multiple tool sweeps at various orientations confirmed the effective placement and survivability of the microchips.
2. Smart MicoChips Data Processing and Modeling: The raw microchip data was processed and integrated into a physics-informed AI modeling platform called iGeoSensing. Detailed analysis of signal amplitudes enabled the resolution of proppant distribution and fracture geometry at each depth interval at a scale of 1ft which revealed significant vertical heterogeneity in the proppant distribution, a level of detail previously obtainable only through extensive coring which is very unlikely to obtain during normal operation. Conventional fracture diagnostics typically lack the resolution to capture such fine-scale variability.
3. Diagnostic plots based on Smart MicoChips data: The Smart Microchip data was used to develop new diagnostic tools, including fracture geometry profiles and dimensionless flowback curves. These powerful visualizations illustrate the transformative impact of integrating high-resolution proppant distribution Smart Microchips data into fracture modeling and simulation.

7.3 Future Applications and Alignment with DOE Priorities

The successful field validation of the Smart Microchip Proppants technology has opened up a wide horizon of potential applications that align with the strategic priorities of the Department of Energy's Office of Fossil Energy and Carbon Management:

1. Near wellbore Fracture mapping: Enhance characterization by enabling Smart Microchips to communicate with each other, amplifying power and signal strength.
2. CCUS (Carbon Capture, Utilization, and Storage): Upgrade Smart Microchips with chemical sensing capabilities to detect low concentrations of CO_2 in monitoring wells. This early detection of leakage minimizes contamination of underground sources of drinking water (USDWs) and supports compliance with DOE and EPA Class VI permit requirements.
3. Hydrogen, CO_2, and Natural Gas Transmission Leak Monitoring and Surveillance: Enhance monitoring and surveillance capabilities for detecting leaks and emissions in hydrogen, CO_2, and natural gas transmission systems with AI-powered unmanned aerial surveillance equipped with Smart Microchip chemical sensors.
4. Natural Hydrogen Production and Underground Storage Integrity: The next generation of Smart Microchips, equipped with chemical sensing capabilities, can detect gases such as CH_4, CO_2, and H_2 in the wellbore. These microchips can be integrated with downhole membranes for efficient bottom-hole separation of H_2 from impurities.
5. Critical Mineral Characterization: Integrating Smart Microchips with advanced EPR, hyperspectral, and THz spectroscopy could revolutionize in-situ mineral characterization for optimized critical mineral recovery.

8. CONCLUSION

The resoundingly successful field trial of the Smart Microchip Proppants technology represents a major leap forward in our ability to understand and characterize subsurface fracture systems and proppant distribution. By enabling proppant mapping and fracture diagnostics at the scale of individual feet, this novel technology has unlocked an unprecedented level of reservoir insight.

The results of this field test strongly validate Smart Microchips' ability to withstand harsh downhole conditions while successfully transmitting valuable data to the surface as a novel direct fracture mapping technology. By integrating this granular data into the open-source, physics-informed, AI-empowered iGeoSensing modeling platform, comprehensive signal processing was achieved. These signals were seamlessly converted into meaningful fracture characterization and flow simulation outcomes through its backend simulators. This approach uncovered previously unresolvable heterogeneity in hydraulic fracture geometry and proppant placement, providing unprecedented insights into the complexity of hydraulic fracturing.

The foundational capabilities demonstrated in this field trial will serve as a launch pad for the development of next-generation subsurface diagnostic technologies aligned with the United States evolving energy and resource priorities.

With further refinement and adaptation, Smart Microchip technology is poised to play a transformative role. This includes enhancing near-wellbore fracture mapping, improving wellbore integrity, and ensuring the security of geologic carbon storage. Additionally, their role in optimizing the recovery of critical minerals, pipeline integrity, and hydrogen production, storage, and transport integrity is expected to be game-changing.

This comprehensive capability underscores the versatility and transformative impact of Smart Microchips, making them an indispensable technology for addressing modern energy challenges.

References:

[1] D. Kandris, C. Nakas, D. Vomvas, and G. Koulouras, "Applications of wireless sensor networks: An up-to-date survey," *Appl. Syst. Innov.*, vol. 3, no. 1, pp. 1–24, Mar. 2020.

[2] H. Landaluce, L. Arjona, A. Perallos, F. Falcone, I. Angulo, and F. Muralter, "A review of IoT sensing applications and challenges using RFID and wireless sensor networks," *Sensors*, vol. 20, no. 9, p. 2495, Apr. 2020.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Comput. Netw.*, vol. 38, no. 4, pp. 393–422, 2002, doi: 10.1016/S1389-1286(01)00302-4.

[4] P. A. Charlez, *Rock Mechanics: Petroleum Applications*. Paris, France: Editions Technip, 1997, p. 239.

[5] *Modern Shale Gas Development in the United States: A Primer*, Ground Water Protection Council (U.S.), ALL Consulting (Firm), Department of Energy, Office of Fossil Energy, and National Energy Technology Laboratory, Washington, DC, USA, 2009.

[6] J. C. Reis, *Environmental Control in Petroleum Engineering*. Houston, TX, USA: Gulf, 1976.

[7] J. H. Le Calvez, R. C. Klem, L. Bennett, A. Erwemi, M. Craven, and J. C. Palacio, "Real-time microseismic monitoring of hydraulic fracture treatment: A tool to improve completion and reservoir management," in *Proc. SPE Hydraulic Fracturing Technol. Conf.*, Jan. 2007.

[8] M. C. Fehler, "Stress control of seismicity patterns observed during hydraulic fracturing experiments at the Fenton Hill hot dry rock geothermal energy site, New Mexico," *Int. J. Rock Mech. Mining Sci. Geomech. Abstr.*, vol. 26, nos. 3–4, pp. 211–219, Jul. 1989.

[9] N. Yekeen, E. Padmanabhan, A. K. Idris, and P. S. Chauhan, "Nanoparticles applications for hydraulic fracturing of unconventional reservoirs: A comprehensive review of recent advances and prospects," *J. Petroleum Sci. Eng.*, vol. 178, pp. 41–73, Jul. 2019.

[10] G. L. Barbruni, P. M. Ros, D. Demarchi, S. Carrara, and D. Ghezzi, "Miniaturised wireless power transfer systems for neurostimulation: A review," *IEEE Trans. Biomed. Circuits Syst.*, vol. 14, no. 6, pp. 1160–1178, Dec. 2020, doi: 10.1109/TBCAS.2020.3038599.

[11] A. Costanzo and D. Masotti, "Energizing 5G: Near- and far-field wireless energy and data trantransfer as an enabling technology for the 5G IoT," *IEEE Microw. Mag.*, vol. 18, no. 3, pp. 125–136, May 2017, doi: 10.1109/MMM.2017.2664001.

[12] A. Aderibigbe, K. Cheng, Z. Heidari, J. Killough, T. Fuss, and W. T. Stephens, "Detection of propping agents in fractures using mag- netic susceptibility measurements enhanced by magnetic nanoparticles," in *Proc. SPE Annu. Tech. Conf. Exhib.*, Oct. 2014, doi: 10.2118/170818- MS.

[13] A. Aderibigbe, K. Cheng, Z. Heidari, J. Killough, and T. Fuss-Dezelic, "Application of magnetic nanoparticles mixed with propping agents in enhancing near-wellbore fracture detection," *J. Petroleum Sci. Eng.*, vol. 141, pp. 133–143, May 2016.

[14] T. Sun, "Study on preparation and performance of nano-ferrofluids used in diagnostic of hydraulic fracture," M.S. thesis, Dept. Oil Natural Gas Eng., China Univ. Petroleum, Beijing, China, 2016.

[15] J. Liu, S. Cao, X. Wu, and J. Yao, "Detecting the propped fracture by injection of magnetic proppant during fracturing," *Geophysics*, vol. 84, no. 3, pp. JM1–JM14, May 2019.

[16] A. A. Al-Shehri, I. F. Akyildiz, J. M. Servin, and H. K. Schmidt, "FracBot technology for mapping hydraulic fractures," in *Proc. SPE Annu. Tech. Conf. Exhib.*, Oct. 2017, doi:

10.2118/187196-MS.
[17] A. A. Alshehri, C. H. Martins, S.-C. Lin, I. F. Akyildiz, and H. K. Schmidt, "FracBot technology for mapping hydraulic fractures," *SPE J.*, vol. 26, no. 2, pp. 610–626, Apr. 2021.
[18] W. G. P. Kumari et al., "Hydraulic fracturing under high temperature and pressure conditions with micro-CT applications: Geothermal energy from hot dry rocks," *Fuel*, vol. 230, pp. 138–154, Oct. 2018.
[19] P. L. Dreike, D. M. Fleetwood, D. B. King, D. C. Sprauer, and T. E. Zipperian, "An overview of high-temperature electronic device technologies and potential applications," *IEEE Trans. Compon., Packag., Manuf. Technol., A*, vol. 17, no. 4, pp. 594–609, Dec. 1994, doi: 10.1109/95.335047.
[20] D. M. Fleetwood, F. V. Thome, S. S. Tsao, P. V. Dressendorfer, J. Dandini, and J. R. Schwank, "High-temperature silicon-on- insulator electronics for space nuclear power systems: Requirements and feasibility," *IEEE Trans. Nucl. Sci.*, vol. NS-35, no. 5, pp. 1099–1112, Oct. 1988, doi: 10.1109/23.7506.
[21] P. G. Neudeck, R. S. Okojie, and L.-Y. Chen, "High-temperature electronics—A role for wide bandgap semiconductors?" *Proc. IEEE*, vol. 90, no. 6, pp. 1065–1076, Jun. 2002, doi: 10.1109/JPROC.2002.1021571.
[22] T.-H. Chen, L. T. Clark, and K. E. Holbert, "Memory design for high- temperature radiation environments," in *Proc. IEEE Int. Rel. Phys. Symp.*, Apr. 2008, pp. 107–114, doi: 10.1109/RELPHY.2008.4558870.
[23] N. Sadeghi, A. Sharif-Bakhtiar, and S. Mirabbasi, "A 0.007-mm^2 108- ppm/°C 1-MHz relaxation oscillator for high-temperature applications up to 180 °C in 0.13 μm CMOS," *IEEE Trans. Circuits Syst.I, Reg. Papers*, vol. 60, no. 7, pp. 1692–1701, Jul. 2013, doi: 10.1109/TCSI.2012.2226500.
[24] C. Davis and I. Finvers, "A 14-bit high-temperature *E* modulator in standard CMOS," *IEEE J. Solid-State Circuits*, vol. 38, no. 6, pp. 976–986, Jun. 2003, doi: 10.1109/JSSC.2003.811973.
[25] I. Habibagahi et al., "Vagus nerve stimulation using a miniaturized wirelessly powered stimulator in pigs," *Sci. Rep.*, vol. 12, no. 1, p. 8184, May 2022, doi: 10.1038/s41598-022-11850-0.
[26] A. Ray, I. Habibagahi, and A. Babakhani, "Fully wireless and batteryless localization and physiological motion detection system for point-of-care biomedical applications," in *Proc. IEEE Biomed. Circuits Syst. Conf. (BioCAS)*, Oct. 2022, pp. 26–30.
[27] J. Jang, I. Habibagahi, H. Rahmani, and A. Babakhani, "Wire- lessly powered, batteryless closed-loop biopotential recording IC for implantable leadless cardiac monitoring applications," in *Proc. IEEE Biomed. Circuits Syst. Conf. (BioCAS)*, Oct. 2021, pp. 1–4, doi: 10.1109/BIOCAS49922.2021.9644988.
[28] U. Guler and M. Ghovanloo, "Power management in wireless power-sipping devices: A survey," *IEEE Circuits Syst. Mag.*, vol. 17, no. 4, pp. 64–82, 4th Quart., 2017, doi: 10.1109/MCAS.2017.2757090.
[29] B. Razavi, "The role of PLLs in future wireline transmitters," IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 56, no. 8, pp. 1786–1793, Aug. 2009.
[30] Ray, Arkaprova, and Aydin Babakhani. "A Wirelessly Powered System of Coherent Sensing Nodes for Fracture Mapping Applications at Temperatures up to 250° C and Pressures up to 24 MPa." IEEE Sensors Journal 23.10 (2023): 10605-10615.

[31] I. Habibagahi, R. P. Mathews, A. Ray and A. Babakhani, "Design and Implementation of Multisite Stimulation System Using a Double-Tuned Transmitter Coil and Miniaturized Implants," in IEEE Microwave and Wireless Technology Letters, vol. 33, no. 3, pp. 351-354, March 2023, doi: 10.1109/LMWC.2022.3217519.

[32] *Andreas Wuestefeld, Thedor I. Urbancic, Adam Baig, Marc Prince*. A decade monitoring Shale Gas plays using microseismicity: advances in the understanding of Hydraulic Fracturing. SPE Annual Technical Conference and Exhibition, San Antonio, Texas, USA, 8-10 October 2012

[33] *Alexander Katashov et.al*. Using markers for Production Logging in horizontal gas wells with multistage hydraulic fracturing. SPE-201624-MS, SPE Annual Technical Conference & Exhibition, Denver, Colorado, USA, 5 – 7 October 2020.

[34] *Becht, E. et al, 2019.* Dimensionality reduction for visualizing single-cell data using UMAP

[35] *Claudia Malzer and Marcus Baum*. A hybrid approach To Hierarchical Density-based Cluster Selection. 2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI), Virtual Conference, Sept. 14-16, 2020.

[36] *Gustavo A. Ugueto et.al*. Application of Integrated Advanced Diagnostics and Modeling to Improve Hydraulic Fracture Stimulation Analysis and Optimization. SPE-168603, SPE Hydraulic Fracturing Technology Conference held in The Woodlands, Texas, USA, 4–6 February 2014.

[37] *King G.E.* Thirsty Years of Gas Shale Fracturing: What Have We Learned? SPE 133456, SPE Annual Technical Conference and Exhibition, Florence, Italy, Sep 19-22, 2010.

[38] *Laurens van der Maaten and Geoffrey Hinton*. Visualizing data using t-SNE. Journal of Machine Learning Research 9, p2579-2605., 2008.

[39] *Marian Morys, Sergei Knizhnik, Andrew R Duncan, Brady E. Tingey*. Advances in Borehole Imaging in Unconventional Reservoirs. Unconventional Resources Technology Conference (URTeC), Houston, Texas, USA, 23-25 July 2018.

[40] *Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe*. Poisson Surface Reconstruction. Eurographics Symposium on Geometry Processing, 2006.

[41] *S. C. Shrivastava*. A Review on Affine Transformation. International Advanced Research Journal in Science, Engineering and Technology, Vol3, Issue 8, August 2016.

[42] *Soumyadeep Ghosh et.al*. Insights into fracture fluid distribution and fracture geometry in hydraulically fractured horizontal wells through thermal simulations and fiber optics distributed temperature sensing (FO - DTS) measurements. DOI 10.15530/urtec-2020-3028, Unconventional Resources Technology Conference, Austin, Texas, USA, 20- 22 July 2020.

[43] *Vuong Van Pham et.al*. i-GeoSensing Fracture Diagnostic (i-GSFD) For Fast Processing Of The Smart Microchip Proppants Data. Annual Technical Conference and Exhibition (ATCE) 2021, Dubai, UAE, 9/21-9/23, 2021.

[44] *Yingfan Wang, Haiyang Huang, Cynthia Rudin, and Yaron Shaposhnik*. Understanding How Dimension Reduction Tools Work: An Empirical Approach to Deciphering t-SNE, UMAP, TriMAP, and PaCMAP for Data Visualization. Journal of Machine Learning Research 22 (1-73), 7/21/2021.

[45] *Andreas Wuestefeld, Thedor I. Urbancic, Adam Baig, Marc Prince*. A decade monitoring Shale Gas plays using micro-seismicity: advances in the understanding of Hydraulic Fracturing. SPE Annual Technical Conference and Exhibition, San Antonio, Texas, USA, 8-10 October 2012

[46] *Alexander Katashov et.al*. Using markers for Production Logging in horizontal gas wells with multistage hydraulic fracturing. SPE-201624-MS, SPE Annual Technical Conference & Exhibition, Denver, Colorado, USA, 5 – 7 October 2020.

[47] *Biggs, Norman (1993), Algebraic Graph Theory, Cambridge Mathematical Library (2nd ed.), Cambridge University Press, Definition 2.1, p. 7.*

[48] Charles R. Qi* et.al, *Point Net: Deep Learning on Point Sets for 3D Classification and Segmentation*. arXiv:1612.00593v2. April 10th, 2017.

[49] *Claudia Malzer and Marcus Baum*. A hybrid approach To Hierarchical Density-based Cluster Selection. 2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI), Virtual Conference, Sept. 14-16, 2020.

[50] *Gidley, J.L., Holditch, S.A., Nierode, D.E. et al*. Three-Dimensional Fracture-Propagation Models. In Recent Advances in Hydraulic Fracturing, 12. Chap. 5, 95. SPE, 1989.

[51] *Gustavo A. Ugueto et.al*. Application of Integrated Advanced Diagnostics and Modeling to Improve Hydraulic Fracture Stimulation Analysis and Optimization. SPE-168603, SPE Hydraulic Fracturing Technology Conference held in The Woodlands, Texas, USA, 4–6 February 2014.

[52] *John Lawson*. Design and Analysis of Experiments with R. CRC Press, Taylor & Francis Group, 2017.

[53] *King G.E.* Thirsty Years of Gas Shale Fracturing: What Have We Learned? SPE 133456, SPE Annual Technical Conference and Exhibition, Florence, Italy, Sep 19-22, 2010.

[54] *Laurens van der Maaten and Geoffrey Hinton*. Visualizing data using t-SNE. Journal of Machine Learning Research 9, p2579-2605., 2008.

[55] *Marian Morys, Sergei Knizhnik, Andrew R Duncan, Brady E. Tingey*. Advances in Borehole Imaging in Unconventional Reservoirs. Unconventional Resources Technology Conference (URTeC), Houston, Texas, USA, 23-25 July 2018.

[56] *Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu*. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)

[57] *Meng Tang, Yimin Liu, and Louis J. Durlofsky, Stanford University.* History Matching Complex 3D Systems Using Deep-Learning-Based Surrogate Flow Modeling and CNN-PCA Geological Parameterization. SPE Reservoir Simulation Conference, 19 October 2021. SPE-203924-MS

[58] *Michael J. Economides et.al,* 2002. Unified Fracture Design: bridging the Gap between Theory and Practice. *Nolte, K.G. and Smith, M.G.* Interpretation of Fracturing Pressures. J Pet Technol 33 (9): 1767–1775, 1981. SPE-8297-PA.

[59] *Paul Webster*, Barbara Cox, Mathieu Molenaar; Shell Canada*. Developments in Diagnostic Tools for Hydraulic Fracture Geometry Analysis. Unconventional Resources Technology Conference (URTeC). Denver, Colorado, USA, 12-14 August 2013.

[60] *Vuong Van Pham et.al*. i-GeoSensing Fracture Diagnostic (i-GSFD) For Fast Processing of The Smart Microchip Proppants Data. Annual Technical Conference and Exhibition (ATCE) 2021, Dubai, UAE, 9/21-9/23, 2021.

[61] Sergey Stolyarov, Eduardo Cazeneuve, Karim Sabaa, David Katz, and Junjei Yang, BHGE. A Novel Technology for Hydraulic Fracture Diagnostics in the Vicinity and Beyond the Wellbore. SPE Hydraulic Fracturing Technology Conference and Exhibition. Woodlands, Texas, USA, 5-7 February 2019. SPE-194373-MS

[62] James Bergstra, Remi Bardenet, Yoshua Bengio, Balazs Kegl. *Algorithms for Hyper-Parameter Optimization*. Advances in Neural Information Processing Systems 24 (NIPS 2011) https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

[63] Charles A. Kang, Mark W. McClure, Somasekhar Reddy. *Description of ResFrac automated history matching and optimization workflow*. Submitted 25th November 2021.

[64] Garrido-Merchan and D. Hernandez-Lobato. *Dealing with categorical and integer-valued variables in Bayesian Optimization with Gaussian processes*. Neurocomputing 380 (2020) 20–35.

[65] Mark McClure, Charles Kang, Soma Medam, Chris Hewson, Egor Dontsov, Ankush Singh, Carlo Peruzzo, Elizaveta Gordeliy. *ResFrac technical write-up.* https://arxiv.org/abs/1804.02092. Submitted 6th April 2018, last revised 22nd April 2024.

[66] Hyper-opt documentation. https://hyperopt.github.io/hyperopt/

[67] Sacks, J. and Schiller, S. B., and Welch, W. J. *Designs for computer experiments*, Technometrics 31 (1) (1989) 41–47.

[68] Tianqi Chen, Carlos Guestrin. *XGBoost: A Scalable Tree Boosting System.* arXiv:1603.02754.

[69] Friedman, Jerome H. *Greedy Function Approximation: A Gradient Boosting Machine.* Annals of Statistics (2001): 1189-1232.

[70] Scott M. Lundberg, Su-In Lee. *A Unified Approach to Interpreting Model Predictions.* Advances in Neural Information Processing Systems 30 (NIPS 2017)

[71] Tilmann GNEITING and Adrian E. RAFTERY. *Strictly Proper Scoring Rules, Prediction, and Estimation*. Journal of the American Statistical Association March 2007, Vol. 102, No. 477, Review Article DOI 10.1198/016214506000001437

[72] MLflow documentation. https://github.com/mlflow/mlflow

APPENDIX A: EOG CORE SAMPLES AND LOGS FROM A PLUG-BACK PILOT BOYD X STATE X STATE #15H – API 30-015-42223-00-00, PADDOCK FORMATION.

Figure A.1.2507-2510 ft interval

Figure A.2. 2507-2510 ft interval

Figure A.3 2507-2510 original core and logs in box_IMG_5627

Figure A.4 2597-2600

APPENDIX B

BOYD STATE #15H (PADDOCK FORMATION)

QUALITY ASSURANCE AND MINERALOGY

Table B.1: Mineralogy testing of the Paddock Formation

Mineralogy of the Paddock Formation

ID (#)	Depth (m)		Quartz (%)	Calcite (%)	Dolomite (%)	Illite (%)	Smectite (%)	Kaolinite (%)	Chlorite (%)	Pyrite (%)	Orthoclase Feldspar (%)	Oglioclase Feldspar (%)	Mixed Clays (%)	Albite (%)	Anhydrite (%)	Siderite (%)	Apatite (%)	Aragonite (%)	Grain Density (g/cc)
1	2507.50	1	0	4	90	0	0	0	0	0	0	2	0	2	2	0	0	0	2.824
2	2507.75	1	0	3	89	0	0	0	0	0	0	1	1	0	6	0	0	0	2.842
3	2508.25	1	0	4	91	0	0	0	0	0	0	0	0	1	4	0	0	0	2.842
4	2508.50	1	0	4	90	1	0	0	0	0	0	1	1	0	3	0	0	0	2.803
5	2508.75	1	0	2	92	0	0	0	0	0	0	1	0	2	3	0	0	0	2.870
6	2509.50	1	0	1	91	0	0	0	0	0	0	2	0	0	6	0	0	0	2.848
7	2509.75	1	0	3	92	0	0	0	0	0	0	0	0	0	5	0	0	0	2.845
8	2597.10	2	2	1	67	8	2	0	2	0	0	0	0	2	17	0	0	0	2.822
9	2597.40	2	0	0	66	1	0	0	0	0	0	0	0	0	32	0	0	0	2.855
10	2597.90	2	0	0	90	4	0	0	0	0	2	0	0	0	3	0	0	0	2.813
11	2600.20	2	0	0	66	4	2	0	0	0	0	0	0	0	27	0	0	0	2.830
12	2600.60	2	0	0	85	6	0	0	0	0	0	0	0	2	7	0	0	0	2.815
13	2600.90	2	0	0	56	4	1	0	0	0	1	0	0	0	38	0	0	0	2.851
	Average	1	0.0	3.0	90.7	0.1	0.0	0.0	0.0	0.0	0.0	1.0	0.3	0.7	4.1	0.0	0.0	0.0	2.839
	Average	2	0.3	0.1	71.8	4.6	0.8	0.0	0.3	0.0	0.4	0.0	0.0	0.7	20.8	0.0	0.1	0.0	2.831
	Average		0.1	1.7	82.0	2.2	0.4	0.0	0.1	0.0	0.2	0.5	0.2	0.7	11.8	0.0	0.1	0.0	2.835

-:)

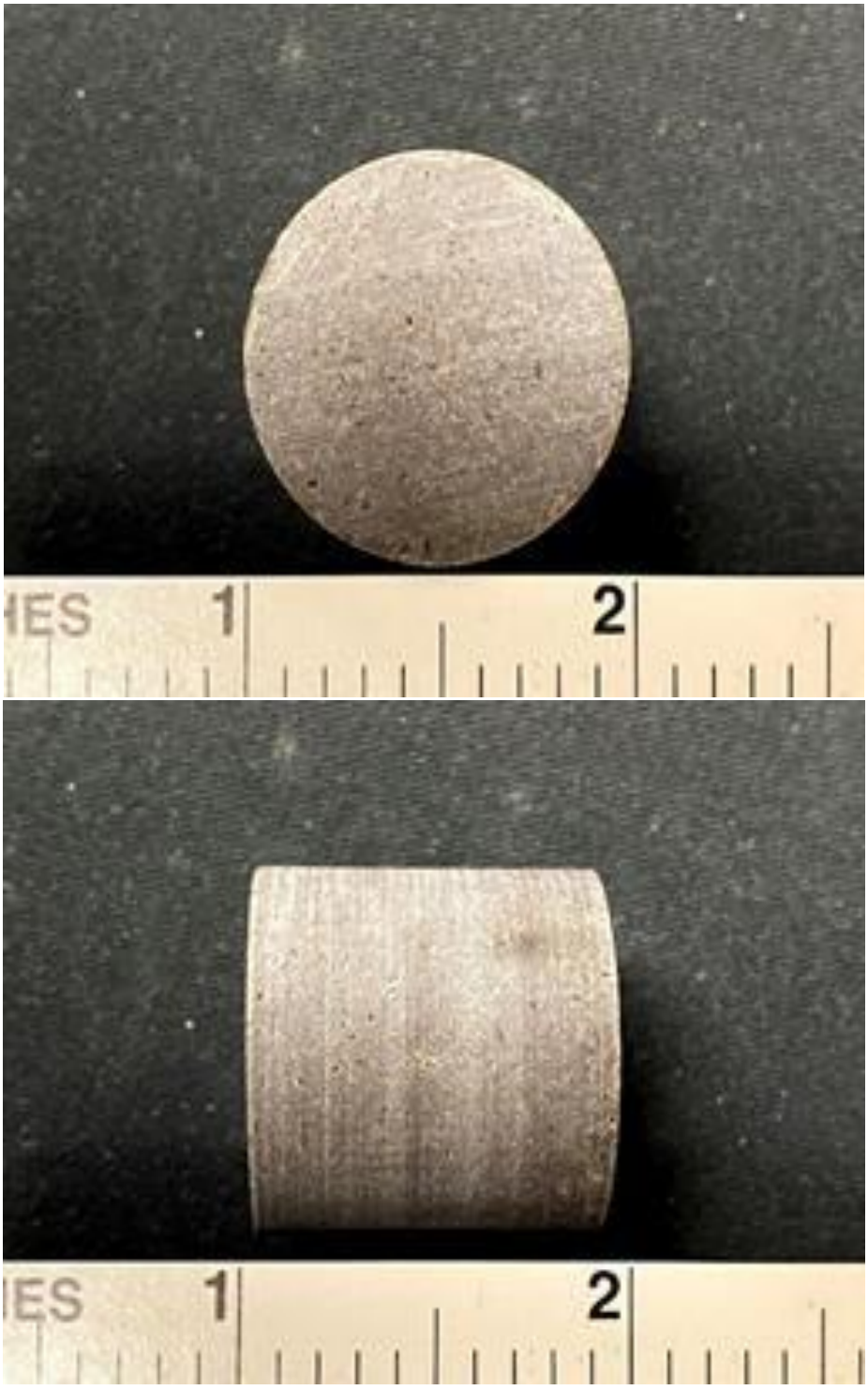


Figure B.1 2507.50 feet (Top and Side View)

Figure B.2 2507.50 feet (Top and Side View)

2508.25feet (Top and Side View)

Figure B.3. 2508.25 feet (Top and Side View)

feet (Top and Side View)

Figure B.4. 2808.50 feet (Top and Side View)

2808.75 feet (Top and Side View)

Figure B.5. 2808.75 feet (Top and Side View)

: 2809.50 feet (Top and Side)View

Figure B.6. 2809.50 feet (Top and Side View)

2509.75 feet (Top and Side)View

Figure B.7. 2509 feet (Top and Side View)

2597.10 feet (Top and Side View)

Figure B.8. 2597 feet (Top and Side View)

2597.40 feet (Top and Side View)

Figure B.9. 2597.4 feet (Top and Side View)

2597.90 feet (Top and Side View)

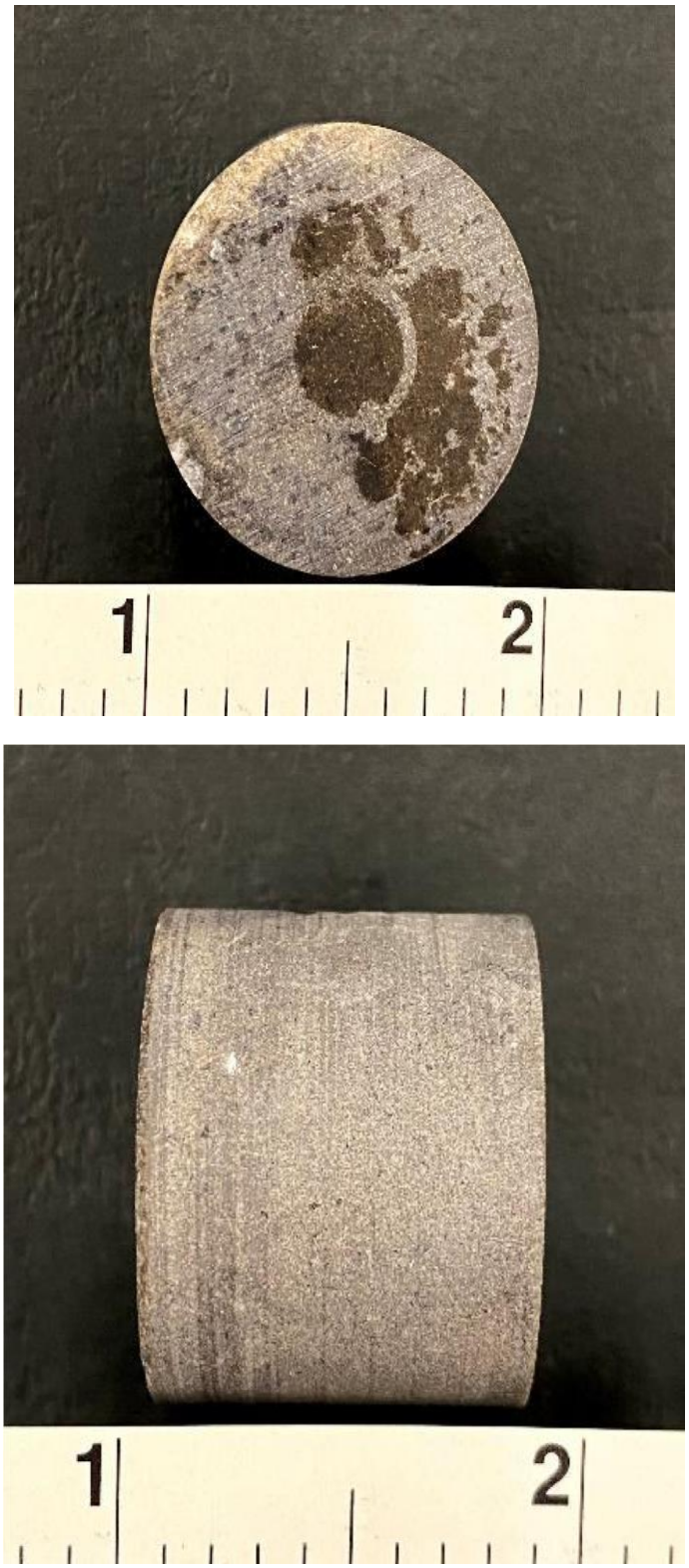


Figure B.10. 2597.9 feet (Top and Side View)

2600.20 feet (Top and Side View)

Figure B.11. 2600.2 feet (Top and Side View)

2600.60 feet (Top and Side View

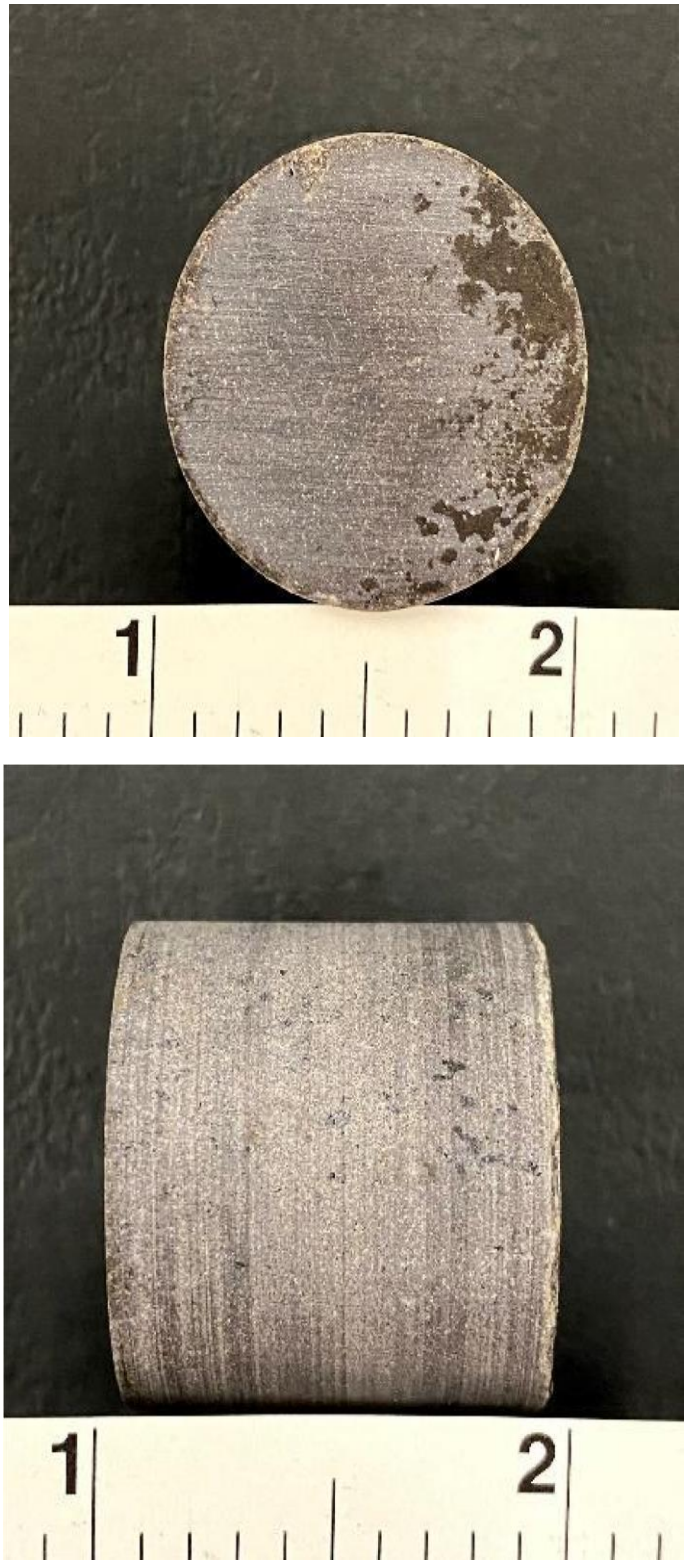


Figure B.12. 2600.6 feet (Top and Side View)

2600.90 feet (Top and Side View

Figure B.13. 2600.9 feet (Top and Side View)

APPENDIX C

SHEAR AND COMPRESSIONAL VELOCITY TESTING PROCEDURES AND RESULTS

BOYD STATE #15H (PADDOCK FORMATION)

Dynamic moduli can be derived from sonic measurement. In this approach, compressional velocity, V_p, and shear velocity, V_s, are measured with a Pulse Transmission technique with nominal velocities of 300-500 KHz. The bulk density of each core sample is measured and the Young's Modulus, E, Shear Modulus, and Poisson's Ratio, n, are calculated from the following equations.

$$S_{dynamic} = \rho * \frac{13{,}400{,}000{,}000}{V_s} \quad \text{.....B-1}$$

$$v_{dynamic} = \frac{(0.5)\left(V_s^{\,2} - 2 * V_p^{\,2}\right)}{\left(V_s^{\,2} - V_p^{\,2}\right)} \quad \text{.....B-2}$$

$$E_{dynamic} = 2 * S_{dynamic} * \left(1 + v_{dynamic}\right) \quad \text{.....B-3}$$

The procedures for conducting laboratory shear and compressional velocity tests are, for the most part, relatively standardized. The assembled sample and instrumentation fixtures are installed in a pressure vessel. After this, typical procedures might include the following steps:

- The core plugs are cleaned, evacuated, and allowed to come to thermal and vapor equilibrium with the atmosphere.
- The sample is then saturated with 25,000 ppm NaCl brine under a pressure of 1,000 psi for 12 hours.
- The samples are then placed in a pressure vessel, confining pressure and pore pressure increased to 250 psi for five minutes and then released.
- Velocities are measured using the Pulse Transmission technique. The nominal frequency of the measurements is 500 KHz for the compressional wave velocity and 350 KHz for the shear wave velocity.

Table C.1. Compressional and Shear Wave Velocity Analysis

Compressional and Shear Wave Velocity Analysis

		Compressional Travel Times Confining Pressure, psi				Confining Pressure, psi		
ID	Depth m	1,000 ft/sec	2,000 ft/sec	3,000 ft/sec	Blk Den g/cc	1,000 μs/ft	2,000 μs/ft	3,000 μs/ft
1	2507.50	21201	22093	22513	2.824	47.16759	45.26321	44.41878
2	2507.75	21814	22730	22969	2.842	45.84212	43.99472	43.53694
3	2508.25	21430	22326	22441	2.842	46.66356	44.79083	44.56129
4	2508.50	21539	21850	21959	2.803	46.42741	45.76659	45.53941
5	2508.75	19642	20420	21486	2.870	50.91131	48.9716	46.54193
6	2509.50	22208	22818	23281	2.848	45.02882	43.82505	42.95348
7	2509.75	21352	21919	21988	2.845	46.83402	45.62252	45.47935
8	2597.10	21133	21905	22170	2.822	47.31936	45.65168	45.106
9	2597.40	21168	21995	22069	2.855	47.24112	45.46488	45.31243
10	2597.90	21535	21996	22114	2.813	46.43603	45.46281	45.22022
11	2600.20	21845	21969	22015	2.830	45.77707	45.51869	45.42357
12	2600.60	22050	22112	22305	2.815	45.35147	45.22431	44.833
13	2600.90	22224	22552	23405	2.851	44.9964	44.34197	42.72591

		Shear Travel Times Confining Pressure, psi				Confining Pressure, psi	
ID	Depth m	1,000 ft/sec	2,000 ft/sec	3,000 ft/sec	1,000 μs/ft	2,000 μs/ft	3,000 μs/ft
1	2507.50	12254	12648	12785	81.6060	79.0639	78.2167
2	2507.75	11772	11965	12270	84.9473	83.5771	81.4996
3	2508.25	11391	11578	11900	87.7886	86.3707	84.0336
4	2508.50	11368	11512	11693	87.9662	86.8659	85.5213
5	2508.75	10554	10919	11214	94.7508	91.5835	89.1742
6	2509.50	11998	12323	12477	83.3472	81.1491	80.1475
7	2509.75	11827	12021	12297	84.5523	83.1878	81.3206
8	2597.10	13133	13409	13426	76.1441	74.5768	74.4823
9	2597.40	10833	11064	11098	92.3105	90.3832	90.1063
10	2597.90	12138	12540	12568	82.3859	79.7448	79.5672
11	2600.20	12225	12479	12650	81.7996	80.1346	79.0514
12	2600.60	12313	12578	12720	81.2150	79.5039	78.6164
13	2600.90	13273	14135	14966	75.3409	70.7464	66.8181

		Shear Wave Travel Times Confining Pressure, psi		
ID	Depth m	1,000 ft/sec	2,000 ft/sec	3,000 ft/sec
1	2507.50	11512	11890	12070
2	2507.75	12077	12316	12566
3	2508.25	12248	12640	12781
4	2508.50	12112	12325	12580
5	2508.75	11443	11930	12014
6	2509.50	12226	12603	12775
7	2509.75	0	0	0
8	2597.10	11808	11708	11910
9	2597.40	11119	11301	11745
10	2597.90	12216	12869	12923
11	2600.20	12720	12898	12926
12	2600.60	12512	12615	12949
13	2600.90	13170	14284	14965

Compressional and Shear Wave Velocity Analysis

		Poisson's Ratio Confining Pressure, psi			Dynamic Shear Confining Pressure, psi			Dynamic Young's Modulus Confining Pressure, psi		
ID	Depth m	1,000 *psi*	2,000 *psi*	3,000 *psi*	1,000 *psi*	2,000 *psi*	3,000 *psi*	1,000 *psi*	2,000 *psi*	3,000 *psi*
1	2507.50	*0.25*	*0.26*	*0.26*	*5682314*	*6053593*	*6185445*	*14196302*	*15209483*	*15611922*
2	2507.75	*0.29*	*0.31*	*0.30*	*5277514*	*5451980*	*5733476*	*13664082*	*14266170*	*14910929*
3	2508.25	*0.30*	*0.32*	*0.30*	*4941429*	*5105003*	*5392905*	*12878327*	*13437056*	*14069010*
4	2508.50	*0.31*	*0.31*	*0.30*	*4853958*	*4977708*	*5135465*	*12687690*	*13020445*	*13373947*
5	2508.75	*0.30*	*0.30*	*0.31*	*4283718*	*4585138*	*4836239*	*11112403*	*11919452*	*12698103*
6	2509.50	*0.29*	*0.29*	*0.30*	*5493669*	*5795323*	*5941076*	*14216612*	*14999731*	*15429210*
7	2509.75	*0.28*	*0.28*	*0.27*	*5332566*	*5508943*	*5764816*	*13637453*	*14157146*	*14670766*
8	2597.10	*0.19*	*0.20*	*0.21*	*6522134*	*6799149*	*6816400*	*15462809*	*16322839*	*16501566*
9	2597.40	*0.32*	*0.33*	*0.33*	*4489608*	*4683120*	*4711947*	*11875764*	*12462969*	*12540925*
10	2597.90	*0.27*	*0.26*	*0.26*	*5553526*	*5927473*	*5953973*	*14074795*	*14928215*	*15021296*
11	2600.20	*0.27*	*0.26*	*0.25*	*5667467*	*5905420*	*6068373*	*14418120*	*14903193*	*15213858*
12	2600.60	*0.27*	*0.26*	*0.26*	*5718880*	*5967692*	*6103197*	*14565310*	*15048443*	*15368143*
13	2600.90	*0.22*	*0.18*	*0.15*	*6730390*	*7632972*	*8556842*	*16459396*	*17960179*	*19751763*

		Shear Wave Anisotropy Confining Pressure, psi						Dynamic Young's Modulus Confining Pressure, psi		
ID	Depth m	1,000 fraction	2,000 fraction	3,000 fraction		ID	Depth m	1,000 *Mmpsi*	2,000 *Mmpsi*	3,000 *Mmpsi*
1	2507.50	*0.06*	*0.06*	*0.06*		1	2507.50	*14.20*	*15.21*	*15.61*
2	2507.75	*-0.03*	*-0.03*	*-0.02*		2	2507.75	*13.66*	*14.27*	*14.91*
3	2508.25	*-0.08*	*-0.09*	*-0.07*		3	2508.25	*12.88*	*13.44*	*14.07*
4	2508.50	*-0.07*	*-0.07*	*-0.08*		4	2508.50	*12.69*	*13.02*	*13.37*
5	2508.75	*-0.08*	*-0.09*	*-0.07*		5	2508.75	*11.11*	*11.92*	*12.70*
6	2509.50	*-0.02*	*-0.02*	*-0.02*		6	2509.50	*14.22*	*15.00*	*15.43*
7	2509.75	*1.00*	*1.00*	*1.00*		7	2509.75	*13.64*	*14.16*	*14.67*
8	2597.10	*0.10*	*0.13*	*0.11*		8	2597.10	*15.46*	*16.32*	*16.50*
9	2597.40	*-0.03*	*-0.02*	*-0.06*		9	2597.40	*11.88*	*12.46*	*12.54*
10	2597.90	*-0.01*	*-0.03*	*-0.03*		10	2597.90	*14.07*	*14.93*	*15.02*
11	2600.20	*-0.04*	*-0.03*	*-0.02*		11	2600.20	*14.42*	*14.90*	*15.21*
12	2600.60	*-0.02*	*0.00*	*-0.02*		12	2600.60	*14.57*	*15.05*	*15.37*
13	2600.90	*0.01*	*-0.01*	*0.00*		13	2600.90	*16.46*	*17.96*	*19.75*

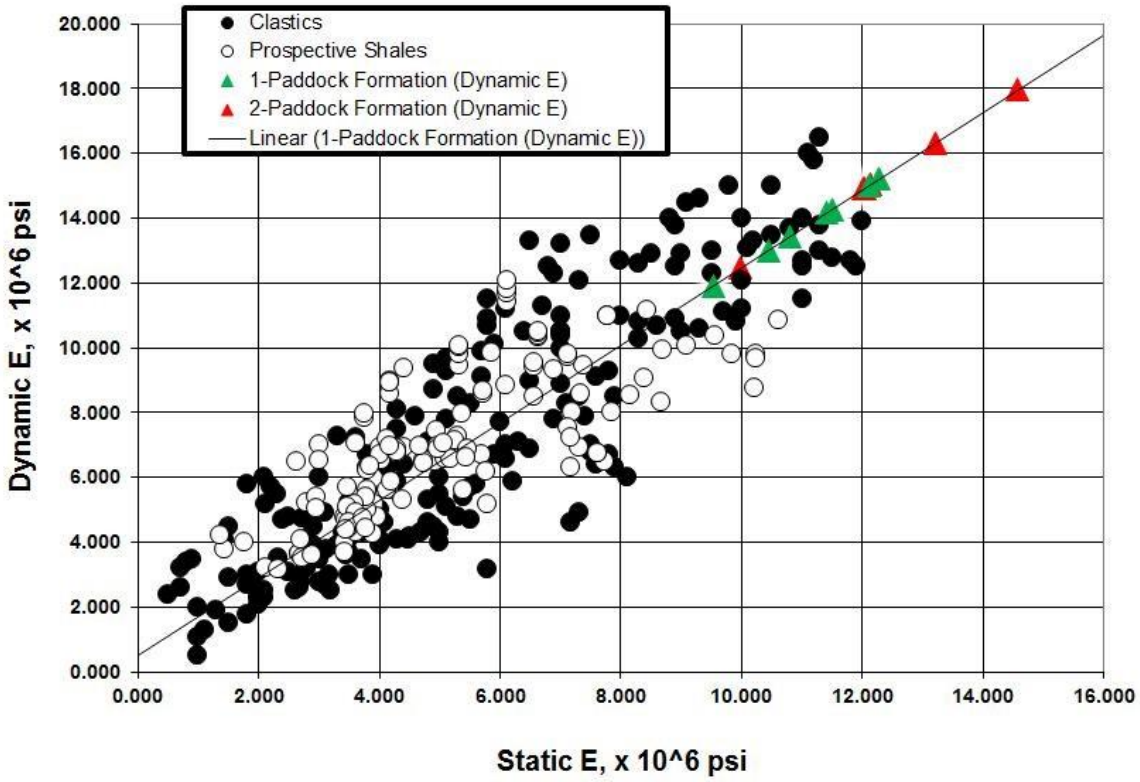
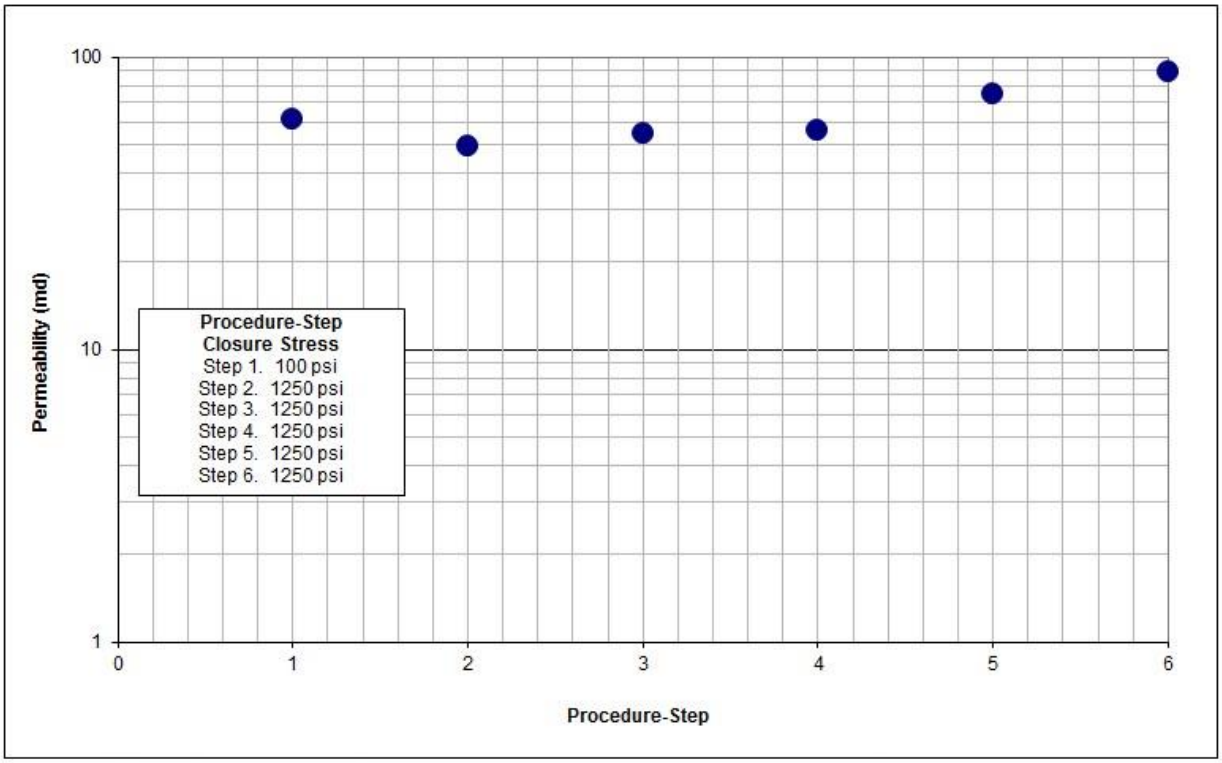


Figure C.1: Dynamic E vs Static E

APPENDIX D

BOYD STATE #15H (PADDOCK FORMATION)

UN-PROPPED CRACK TESTING



Well Name: Boyd State #15H
Field:
Formation: Paddock
Temperature (degrees F): 69
Note: Horizontal, Naturally Fractured Sample

Province/County: Eddy
Country/State: New Mexico
Sample Length (inches): .9155
Sample Diameter (inches): .9775
Depth (feet) - Id.: 2600.20 - 11, Core Set 2

Figure D.1: Un-Propped Crack Test: Paddock Core Set 2, ID-11, (2600.20')

Table D.1: Procedures: Paddock Formation, Core Set 2, ID-11 (2600.20 ft)

Procedure-Step 1. Naturally fractured core sample, closure stress 100 psi, 68 degrees F, record permeability using 4% KCL water. (Monday AM)
Comment: Good test

Procedure-Step 2. Ramp closure stress to 1250 psi, 68 degrees F, record permeability using 4% KCL water. (Monday PM)
Comment: Good test

Procedure-Step 3. Hold closure stress at 1250 psi, 68 degrees F, record permeability using 4% KCL water. (Tuesday AM)
Comment: Good test

Procedure-Step 4. Hold closure stress at 1250 psi, 68 degrees F, record permeability using 4% KCL water. (Tuesday PM)
Comment: Good Test

Procedure-Step 5. Hold closure stress at 1250 psi, 68 degrees F, record permeability using 4% KCL water. (Wednesday AM)
Comment: Good Test

Procedure-Step 6. Hold closure stress at 1250 psi, 68 degrees F, record permeability using 4% KCL water. (Wednesday PM)
Comment: Good Test

Table D.2: Laboratory Data: Paddock Formation, Core Set 2, ID-11 (2600.20 ft)

perm (md)	test temp (deg F)	M1 (cc)	M2 (cc)	volume (cc)	minutes	seconds	delta t (sec)	prod rate (cc/min)	Core Conditions Temp viscosity (cp)	length (in)	diameter (in)	P1 (psig)	P2 (psig)	delta P (psig)	units - autoconversion			Closure Stress (psi)	Procedure-Step	4% KCL Water (md)
Procedure-Step 1. Closure Stress 100 psi, 4% KCL Water, Monday 04/12/2021 9:00 AM																				
62.1908	68	0	4.30	4.30	0	300	300	0.8600	1.0020	0.9155	0.9775	1.63	0	1.63	2.3254	2.4829	0.1109	100	1	61.6997
60.0082	68	0	4.20	4.20	0	300	300	0.8400	1.0020	0.9155	0.9775	1.65	0	1.65	2.3254	2.4829	0.1123			
62.4870	68	0	4.40	4.40	0	300	300	0.8800	1.0020	0.9155	0.9775	1.66	0	1.66	2.3254	2.4829	0.1130			
62.1129	68	0	4.40	4.40	0	300	300	0.8800	1.0020	0.9155	0.9775	1.67	0	1.67	2.3254	2.4829	0.1136			
Procedure-Step 2. Ramp Closure Stress to 1250 psi, 4% KCL Water, Monday 04/12/2021 2:30 PM																				
50.6745	68	0	4.60	4.60	0	300	300	0.9200	1.0020	0.9155	0.9775	2.14	0	2.14	2.3254	2.4829	0.1456	1250	2	49.9646
49.5729	68	0	4.50	4.50	0	300	300	0.9000	1.0020	0.9155	0.9775	2.14	0	2.14	2.3254	2.4829	0.1456			
49.8056	68	0	4.50	4.50	0	300	300	0.9000	1.0020	0.9155	0.9775	2.13	0	2.13	2.3254	2.4829	0.1449			
49.8056	68	0	4.50	4.50	0	300	300	0.9000	1.0020	0.9155	0.9775	2.13	0	2.13	2.3254	2.4829	0.1449			
Procedure-Step 3. Hold Closure Stress at 1250 psi, 4% KCL Water, Tuesday 04/13/2021 9:00 AM																				
54.2007	68	0	5.15	5.15	0	300	300	1.0300	1.0020	0.9155	0.9775	2.24	0	2.24	2.3254	2.4829	0.1524	1250	3	54.9044
53.9151	68	0	5.10	5.10	0	300	300	1.0200	1.0020	0.9155	0.9775	2.23	0	2.23	2.3254	2.4829	0.1517			
56.2818	68	0	5.30	5.30	0	300	300	1.0600	1.0020	0.9155	0.9775	2.22	0	2.22	2.3254	2.4829	0.1511			
55.2199	68	0	5.20	5.20	0	300	300	1.0400	1.0020	0.9155	0.9775	2.22	0	2.22	2.3254	2.4829	0.1511			
Procedure-Step 4. Hold Closure Stress at 1250 psi, 4% KCL Water, Tuesday 04/13/2020 2:30 PM																				
49.6309	68	0	4.40	4.40	0	300	300	0.8800	1.0020	0.9155	0.9775	2.09	0	2.09	2.3254	2.4829	0.1422	1250	4	56.1872
55.5364	68	0	4.90	4.90	0	300	300	0.9800	1.0020	0.9155	0.9775	2.08	0	2.08	2.3254	2.4829	0.1415			
59.2214	68	0	5.20	5.20	0	300	300	1.0400	1.0020	0.9155	0.9775	2.07	0	2.07	2.3254	2.4829	0.1409			
60.3602	68	0	5.30	5.30	0	300	300	1.0600	1.0020	0.9155	0.9775	2.07	0	2.07	2.3254	2.4829	0.1409			
Procedure-Step 5. Hold Closure Stress at 1250 psi, 4% KCL Water, Wednesday 04/14/2021 9:00 AM																				
76.4583	68	0	6.00	6.00	0	300	300	1.2000	1.0020	0.9155	0.9775	1.85	0	1.85	2.3254	2.4829	0.1259	1250	5	75.0494
75.5926	68	0	5.90	5.90	0	300	300	1.1800	1.0020	0.9155	0.9775	1.84	0	1.84	2.3254	2.4829	0.1252			
76.0057	68	0	5.90	5.90	0	300	300	1.1800	1.0020	0.9155	0.9775	1.83	0	1.83	2.3254	2.4829	0.1245			
72.1410	68	0	5.60	5.60	0	300	300	1.1200	1.0020	0.9155	0.9775	1.83	0	1.83	2.3254	2.4829	0.1245			
Procedure-Step 6. Hold Closure Stress at 1250 psi, 4% KCL Water, Wednesday 04/14/2021 2:30 PM																				
85.3316	68	0	5.90	5.90	0	300	300	1.1800	1.0020	0.9155	0.9775	1.63	0	1.63	2.3254	2.4829	0.1109	1250	6	89.4738
88.7688	68	0	6.10	6.10	0	300	300	1.2200	1.0020	0.9155	0.9775	1.62	0	1.62	2.3254	2.4829	0.1102			
89.4964	68	0	6.15	6.15	0	300	300	1.2300	1.0020	0.9155	0.9775	1.62	0	1.62	2.3254	2.4829	0.1102			
94.2986	68	0	6.40	6.40	0	300	300	1.2800	1.0020	0.9155	0.9775	1.60	0	1.60	2.3254	2.4829	0.1089			

Appendix E Boyd State #15H (Paddock Formation)

Fluid Sensitivity Testing

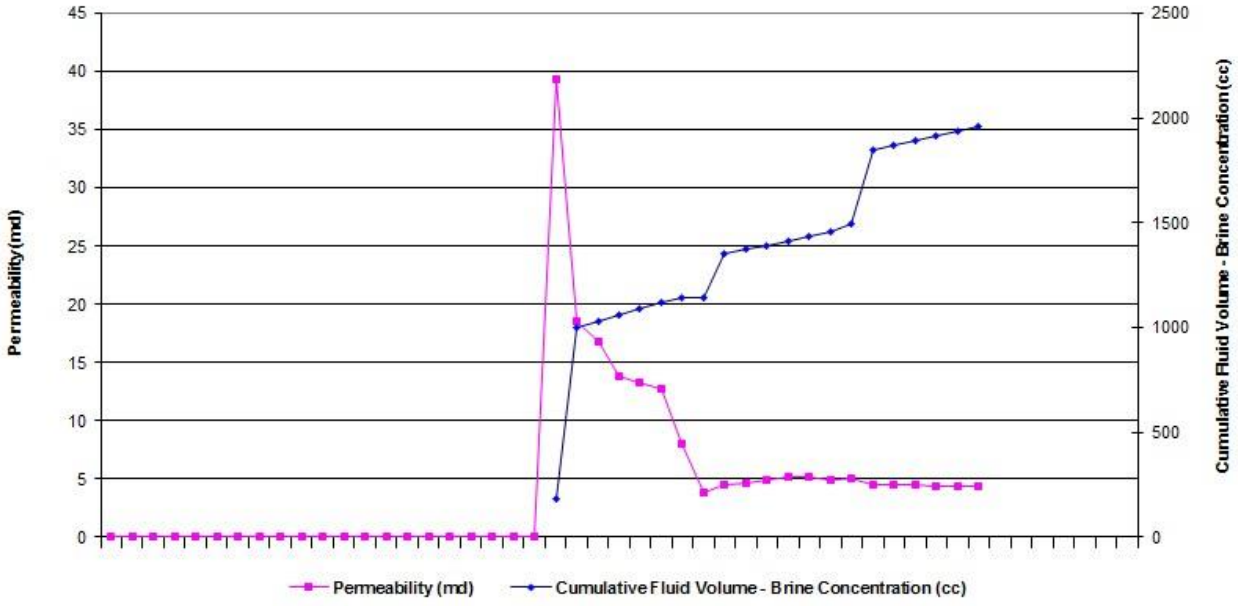


Figure E.1: Fluid Sensitivity Test: Paddock Formation, Core Set 1, ID-7 (2509.75')

Table E.1 Procedures: Boyd State #15H (Paddock Formation) Core Set 1, ID-7

Step 1. Closure Stress 1250 psi, 69 degrees F, Permeability With 2% KCL Water (Tuesday PM) Comments: Total injected volume 14.15cc. Good test. Cumulative volume 185cc.
Step 2. Closure Stress 1250 psi, 69 degrees F, Permeability With 6% KCL Water Wednesday AM) Comments: Total injected volume 12.65cc. Good test. Cumulative volume 812.65cc.
Step 3. Closure Stress 1250 psi, 69 degrees F, Permeability With 6% KCL Water (Wednesday AM) Comments: Total injected volume 12.2cc. Good test. Cumulative volume 32.4cc.
Step 4. Closure Stress 1250 psi, 69 degrees F, Permeability With 6% KCL Water (Wednesday AM) Comments: Total injected volume 10.60cc. Good test. Cumulative volume 30.60cc.
Step 5. Closure Stress 1250 psi, 69 degrees F, Permeability With 6% KCL Water (Wednesday AM) Comments: Total injected volume 10.8cc. Good test. Cumulative volume 30.90cc.
Step 6. Closure Stress 1250 psi, 69 degrees F, Permeability With 6% KCL Water (Wednesday AM) Comments: Total injected volume 10.95cc. Good test. Cumulative volume 31.05cc.
Step 7. Closure Stress 1250 psi, 69 degrees F, Permeability With 6% KCL Water (Wednesday AM) Comments: Total injected volume 7.4cc. Good test. Cumulative volume 17.85cc.

Step 1. Closure Stress 1250 psi, 69 degrees F, Permeability With 4% KCL Water (Wednesday PM)
Comments: Total injected volume .85cc. Good test. Cumulative volume 3.85cc.

Step 2. Closure Stress 1250 psi, 69 degrees F, Permeability With 4% KCL Water (Thursday AM)
Comments: Total injected volume 7.30cc. Good test. Cumulative volume 207.30cc.

Step 3. Closure Stress 1250 psi, 69 degrees F, Permeability With 4% KCL Water (Thursday AM)
Comments: Total injected volume 7.8cc. Good test. Cumulative volume 22.30cc.

Step 4. Closure Stress 1250 psi, 69 degrees F, Permeability With 4% KCL Water (Thursday AM)
Comments: Total injected volume 8.4cc. Good test. Cumulative volume 18.80cc.

Step 5. Closure Stress 1250 psi, 69 degrees F, Permeability With 4% KCL Water (Thursday AM)
Comments: Total injected volume 9.10cc. Good test. Cumulative volume 24.10cc.

Step 6. Closure Stress 1250 psi, 69 degrees F, Permeability With 4% KCL Water (Thursday AM)
Comments: Total injected volume 9.10cc. Good test. Cumulative volume 19.60cc.

Step 7. Closure Stress 1250 psi, 69 degrees F, Permeability With 4% KCL Water (Thursday AM)
Comments: Total injected volume 8.95cc. Good test. Cumulative volume 23.45cc.

Step 1. Closure Stress 1250 psi, 69 degrees F, Permeability With 2% KCL Water (Thursday PM)
Comments: Total injected volume 6.05cc. Good test. Cumulative volume 36.25cc.

Step 2. Closure Stress 1250 psi, 69 degrees F, Permeability With 2% KCL Water (Frisday AM)
Comments: Total injected volume 8.90cc. Good test. Cumulative volume 348.90cc.

Step 3. Closure Stress 1250 psi, 69 degrees F, Permeability With 2% KCL Water (Friday AM)
Comments: Total injected volume 8.95cc. Good test. Cumulative volume 23.95cc.

Step 4. Closure Stress 1250 psi, 69 degrees F, Permeability With 2% KCL Water (Friday AM)
Comments: Total injected volume 9.05cc. Good test. Cumulative volume 24.05cc.

Step 5. Closure Stress 1250 psi, 69 degrees F, Permeability With 2% KCL Water (Friday AM)
Comments: Total injected volume 8.85cc. Good test. Cumulative volume 22.85cc.

Step 6. Closure Stress 1250 psi, 69 degrees F, Permeability With 2% KCL Water (FridayAM)
Comments: Total injected volume 9.20cc. Good test. Cumulative volume 24.2cc.

Step 7. Closure Stress 1250 psi, 69 degrees F, Permeability With 2% KCL Water (Friday AM)
Comments: Total injected volume 9.20cc. Good test. Cumulative volume 24.20cc.

Table E.2: Lab Data: Boyd State #15H (Paddock Formation) Core Set 1, ID-7

STEP 1. Closure Stress 1250 psi, Permeability With 6% KCL Water (Tuesday 03/30/2021 2:00 PM)																		
43.4088	69	0	3.60	3.60	0	300	300	0.7200	1.0020	0.9680	0.9840	2.04	0	2.04	2.4587	2.4994	0.1388	
37.4076	69	0	3.30	3.30	0	300	300	0.6600	1.0020	0.9680	0.9840	2.17	0	2.17	2.4587	2.4994	0.1477	
37.7606	69	0	3.50	3.50	0	300	300	0.7000	1.0020	0.9680	0.9840	2.28	0	2.28	2.4587	2.4994	0.1551	
39.0863	69	0	3.75	3.75	0	300	300	0.7500	1.0020	0.9680	0.9840	2.36	0	2.36	2.4587	2.4994	0.1606	
STEP 2. Closure Stress 1250 psi, Permeability With 6% KCL Water (Wednesday 03/31/2021 7:00 AM)																		
12.9775	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	4.17	0	4.17	2.4587	2.4994	0.2838	
14.0228	69	0	2.40	2.40	0	300	300	0.4800	1.0020	0.9680	0.9840	4.21	0	4.21	2.4587	2.4994	0.2865	
24.5983	69	0	4.20	4.20	0	300	300	0.8400	1.0020	0.9680	0.9840	4.20	0	4.20	2.4587	2.4994	0.2858	
22.4949	69	0	3.85	3.85	0	300	300	0.7700	1.0020	0.9680	0.9840	4.21	0	4.21	2.4587	2.4994	0.2865	
STEP 3. Closure Stress 1250 psi, Permeability With 6% KCL Water (Wednesday 03/31/2021 8:00 AM)																		
13.3264	69	0	2.40	2.40	0	300	300	0.4800	1.0020	0.9680	0.9840	4.43	0	4.43	2.4587	2.4994	0.3014	
16.6205	69	0	3.00	3.00	0	300	300	0.6000	1.0020	0.9680	0.9840	4.44	0	4.44	2.4587	2.4994	0.3021	
19.8552	69	0	3.60	3.60	0	300	300	0.7200	1.0020	0.9680	0.9840	4.46	0	4.46	2.4587	2.4994	0.3035	
17.5702	69	0	3.20	3.20	0	300	300	0.6400	1.0020	0.9680	0.9840	4.48	0	4.48	2.4587	2.4994	0.3048	
STEP 4. Closure Stress 1250 psi, Permeability With 6% KCL Water (Wednesday 03/31/2021 9:00 AM)																		
13.6950	69	0	2.60	2.60	0	300	300	0.5200	1.0020	0.9680	0.9840	4.67	0	4.67	2.4587	2.4994	0.3178	
12.5342	69	0	2.40	2.40	0	300	300	0.4800	1.0020	0.9680	0.9840	4.71	0	4.71	2.4587	2.4994	0.3205	
14.0413	69	0	2.70	2.70	0	300	300	0.5400	1.0020	0.9680	0.9840	4.73	0	4.73	2.4587	2.4994	0.3219	
14.9864	69	0	2.90	2.90	0	300	300	0.5800	1.0020	0.9680	0.9840	4.76	0	4.76	2.4587	2.4994	0.3239	
STEP 5. Closure Stress 1250 psi, Permeability With 6% KCL Water (Wednesday 03/31/2021 10:00 AM)																		
12.3486	69	0	2.50	2.50	0	300	300	0.5000	1.0020	0.9680	0.9840	4.98	0	4.98	2.4587	2.4994	0.3389	
14.2670	69	0	2.90	2.90	0	300	300	0.5800	1.0020	0.9680	0.9840	5.00	0	5.00	2.4587	2.4994	0.3402	
13.7202	69	0	2.80	2.80	0	300	300	0.5600	1.0020	0.9680	0.9840	5.02	0	5.02	2.4587	2.4994	0.3416	
12.6645	69	0	2.60	2.60	0	300	300	0.5200	1.0020	0.9680	0.9840	5.05	0	5.05	2.4587	2.4994	0.3436	
STEP 6. Closure Stress 1250 psi, Permeability With 6% KCL Water (Wednesday 03/31/2021 11:00 AM)																		
14.5803	69	0	3.10	3.10	0	300	300	0.6200	1.0020	0.9680	0.9840	5.23	0	5.23	2.4587	2.4994	0.3559	
12.1589	69	0	2.60	2.60	0	300	300	0.5200	1.0020	0.9680	0.9840	5.26	0	5.26	2.4587	2.4994	0.3579	
12.0899	69	0	2.60	2.60	0	300	300	0.5200	1.0020	0.9680	0.9840	5.29	0	5.29	2.4587	2.4994	0.3600	
12.2529	69	0	2.65	2.65	0	300	300	0.5300	1.0020	0.9680	0.9840	5.32	0	5.32	2.4587	2.4994	0.3620	
STEP 7. Closure Stress 1250 psi, Permeability With 6% KCL Water (Wednesday 03/31/2021 12:00 AM)																		
9.2409	69	0	2.10	2.10	0	300	300	0.4200	1.0020	0.9680	0.9840	5.59	0	5.59	2.4587	2.4994	0.3804	
8.3014	69	0	1.90	1.90	0	300	300	0.3800	1.0020	0.9680	0.9840	5.63	0	5.63	2.4587	2.4994	0.3831	
7.3622	69	0	1.70	1.70	0	300	300	0.3400	1.0020	0.9680	0.9840	5.68	0	5.68	2.4587	2.4994	0.3865	
7.2979	69	0	1.70	1.70	0	300	300	0.3400	1.0020	0.9680	0.9840	5.73	0	5.73	2.4587	2.4994	0.3899	
STEP 1. Closure Stress 1250 psi, Permeability With 4% KCL Water (Wednesday 03/31/2021 3:00 PM)																		
4.0325	69	0	0.20	0.20	0	300	300	0.0400	1.0020	0.9680	0.9840	1.22	0	1.22	2.4587	2.4994	0.0830	
4.6588	69	0	0.25	0.25	0	300	300	0.0500	1.0020	0.9680	0.9840	1.32	0	1.32	2.4587	2.4994	0.0898	
3.4646	69	0	0.20	0.20	0	300	300	0.0400	1.0020	0.9680	0.9840	1.42	0	1.42	2.4587	2.4994	0.0966	
3.2581	69	0	0.20	0.20	0	300	300	0.0400	1.0020	0.9680	0.9840	1.51	0	1.51	2.4587	2.4994	0.1027	
STEP 2. Closure Stress 1250 psi, Permeability With 4% KCL Water (Thursday 04/01/2021 7:00 AM)																		
4.7352	69	0	1.90	1.90	0	300	300	0.3800	1.0020	0.9680	0.9840	9.87	0	9.87	2.4587	2.4994	0.6716	
4.4724	69	0	1.80	1.80	0	300	300	0.3600	1.0020	0.9680	0.9840	9.90	0	9.90	2.4587	2.4994	0.6737	
4.4589	69	0	1.80	1.80	0	300	300	0.3600	1.0020	0.9680	0.9840	9.93	0	9.93	2.4587	2.4994	0.6757	
4.4455	69	0	1.80	1.80	0	300	300	0.3600	1.0020	0.9680	0.9840	9.96	0	9.96	2.4587	2.4994	0.6777	
STEP 3. Closure Stress 1250 psi, Permeability With 4% KCL Water (Thursday 04/01/2021 8:00 AM)																		
4.8470	69	0	2.00	2.00	0	300	300	0.4000	1.0020	0.9680	0.9840	10.15	0	10.15	2.4587	2.4994	0.6907	
4.5910	69	0	1.90	1.90	0	300	300	0.3800	1.0020	0.9680	0.9840	10.18	0	10.18	2.4587	2.4994	0.6927	
4.5776	69	0	1.90	1.90	0	300	300	0.3800	1.0020	0.9680	0.9840	10.21	0	10.21	2.4587	2.4994	0.6947	
4.8044	69	0	2.00	2.00	0	300	300	0.4000	1.0020	0.9680	0.9840	10.24	0	10.24	2.4587	2.4994	0.6968	
STEP 4. Closure Stress 1250 psi, Permeability With 4% KCL Water (Thursday 04/01/2021 9:00 AM)																		
4.7214	69	0	2.00	2.00	0	300	300	0.4000	1.0020	0.9680	0.9840	10.42	0	10.42	2.4587	2.4994	0.7090	
4.9432	69	0	2.10	2.10	0	300	300	0.4200	1.0020	0.9680	0.9840	10.45	0	10.45	2.4587	2.4994	0.7111	
4.9291	69	0	2.10	2.10	0	300	300	0.4200	1.0020	0.9680	0.9840	10.48	0	10.48	2.4587	2.4994	0.7131	
5.1539	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	10.50	0	10.50	2.4587	2.4994	0.7145	
STEP 5. Closure Stress 1250 psi, Permeability With 4% KCL Water (Thursday 04/01/2021 10:00 AM)																		
5.0766	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	10.66	0	10.66	2.4587	2.4994	0.7254	
5.2974	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	10.68	0	10.68	2.4587	2.4994	0.7267	
5.2826	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	10.71	0	10.71	2.4587	2.4994	0.7288	
5.2727	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	10.73	0	10.73	2.4587	2.4994	0.7301	
STEP 6. Closure Stress 1250 psi, Permeability With 4% KCL Water (Thursdday 04/01/2021 11:00 AM)																		
5.2000	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	10.88	0	10.88	2.4587	2.4994	0.7403	
5.1905	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	10.90	0	10.90	2.4587	2.4994	0.7417	
4.9512	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	10.93	0	10.93	2.4587	2.4994	0.7437	
5.1715	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	10.94	0	10.94	2.4587	2.4994	0.7444	
STEP 7. Closure Stress 1250 psi, Permeability With 4% KCL Water (Thursday 04/01/2021 12:00 AM)																		
4.8710	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	11.11	0	11.11	2.4587	2.4994	0.7560	
4.8666	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	11.12	0	11.12	2.4587	2.4994	0.7567	
4.9638	69	0	2.25	2.25	0	300	300	0.4500	1.0020	0.9680	0.9840	11.15	0	11.15	2.4587	2.4994	0.7587	
5.0650	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	11.17	0	11.17	2.4587	2.4994	0.7601	
STEP 1. Closure Stress 1250 psi, Permeability With 2% KCL Water (Thursday 04/01/2021 3:00 PM)																		
4.7566	69	0	1.40	1.40	0	300	300	0.2800	1.0020	0.9680	0.9840	7.24	0	7.24	2.4587	2.4994	0.4927	
5.0545	69	0	1.50	1.50	0	300	300	0.3000	1.0020	0.9680	0.9840	7.30	0	7.30	2.4587	2.4994	0.4967	
5.1874	69	0	1.55	1.55	0	300	300	0.3100	1.0020	0.9680	0.9840	7.35	0	7.35	2.4587	2.4994	0.5001	
5.3114	69	0	1.60	1.60	0	300	300	0.3200	1.0020	0.9680	0.9840	7.41	0	7.41	2.4587	2.4994	0.5042	
STEP 2. Closure Stress 1250 psi, Permeability With 2% KCL Water (Friday 04/02/2021 7:00 AM)																		
4.6083	69	0	2.25	2.25	0	300	300	0.4500	1.0020	0.9680	0.9840	12.01	0	12.01	2.4587	2.4994	0.8172	
4.4984	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	12.03	0	12.03	2.4587	2.4994	0.8186	
4.4910	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	12.05	0	12.05	2.4587	2.4994	0.8200	
4.5854	69	0	2.25	2.25	0	300	300	0.4500	1.0020	0.9680	0.9840	12.07	0	12.07	2.4587	2.4994	0.8213	
STEP 3. Closure Stress 1250 psi, Permeability With 2% KCL Water (Friday 04/02/2021 8:00 AM)																		
4.4321	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	12.21	0	12.21	2.4587	2.4994	0.8308	
4.4249	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	12.23	0	12.23	2.4587	2.4994	0.8322	
4.6185	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.25	0	12.25	2.4587	2.4994	0.8336	
4.5107	69	0	2.25	2.25	0	300	300	0.4500	1.0020	0.9680	0.9840	12.27	0	12.27	2.4587	2.4994	0.8349	
STEP 4. Closure Stress 1250 psi, Permeability With 2% KCL Water (Friday 04/02/2021 9:00 AM)																		
4.4634	69	0	2.25	2.25	0	300	300	0.4500	1.0020	0.9680	0.9840	12.40	0	12.40	2.4587	2.4994	0.8438	
4.4562	69	0	2.25	2.25	0	300	300	0.4500	1.0020	0.9680	0.9840	12.42	0	12.42	2.4587	2.4994	0.8451	
4.5479	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.44	0	12.44	2.4587	2.4994	0.8465	
4.4419	69	0	2.25	2.25	0	300	300	0.4500	1.0020	0.9680	0.9840	12.46	0	12.46	2.4587	2.4994	0.8479	
STEP 5. Closure Stress 1250 psi, Permeability With 2% KCL Water (Friday 04/02/2021 10:00 AM)																		
4.3926	69	0	2.25	2.25	0	300	300	0.4500	1.0020	0.9680	0.9840	12.60	0	12.60	2.4587	2.4994	0.8574	
4.2881	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	12.62	0	12.62	2.4587	2.4994	0.8587	
4.2847	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	12.63	0	12.63	2.4587	2.4994	0.8594	
4.2780	69	0	2.20	2.20	0	300	300	0.4400	1.0020	0.9680	0.9840	12.65	0	12.65	2.4587	2.4994	0.8608	
STEP 6. Closure Stress 1250 psi, Permeability With 2% KCL Water (Friday 04/02/2021 11:00 AM)																		
4.4339	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.76	0	12.76	2.4587	2.4994	0.8683	
4.4269	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.78	0	12.78	2.4587	2.4994	0.8696	
4.4200	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.80	0	12.80	2.4587	2.4994	0.8710	
4.4131	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.82	0	12.82	2.4587	2.4994	0.8723	
STEP 7. Closure Stress 1250 psi, Permeability With 2% KCL Water (Friday 04/02/2021 12:00 AM)																		
4.3756	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.93	0	12.93	2.4587	2.4994	0.8798	
4.3722	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.94	0	12.94	2.4587	2.4994	0.8805	
4.3654	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.96	0	12.96	2.4587	2.4994	0.8819	
4.3621	69	0	2.30	2.30	0	300	300	0.4600	1.0020	0.9680	0.9840	12.97	0	12.97	2.4587	2.4994	0.8826	

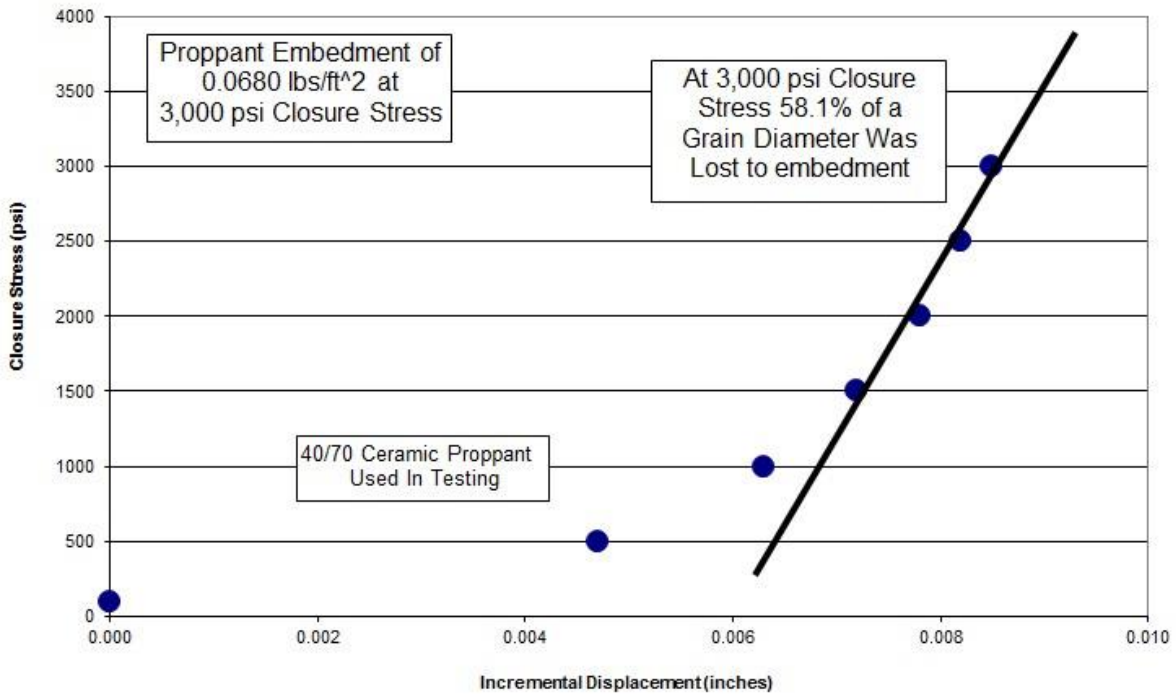


Figure E.2: Embedment Test: Paddock Formation, Core Set 1, ID-7 (2509.75')

Table E.3: Procedures: Embedment Test: Core Set 1, ID-7 (2509.75')

Incremental Fracture Width Change	Closure Stress	Cumulative Fracture Width Change	Results Embedment	Results Embedment
inches	psi	inches	inches	lbs/ft^2
0	100	0.2500	0.0000	0.0000
0.0047	500	0.2453	0.0047	0.0376
0.0063	1000	0.2437	0.0063	0.0504
0.0072	1500	0.2428	0.0072	0.0576
0.0078	2000	0.2422	0.0078	0.0624
0.0082	2500	0.2418	0.0082	0.0656
0.0085	3000	0.2322	0.0085	0.0680
0.0240	8000	0.2260	0.0240	0.1920
0.0515	10000	0.1985	0.0515	0.4120
0.0451	1000	0.2049	0.0451	0.3608
0.0551	10000	0.1949	0.0551	0.4408
0.0490	1000	0.2010	0.0490	0.3920
0.0180	10000	0.2320	0.0180	0.1440
		Grain Diameter Basis:	0.01462	inches
		Cumulative Fracture Width Retained at Ultimate Stress:	92.88	percent

Thursday 04/22/2021 Displacement	Friday 04/23/2021 Displacement	Monday 04/26/2021 Displacement	Tuesday 04/27/2021 Displacement	Wednesday 04/28/2021 Displacement
8:00 am - .0000 inch	8:00 am - .0047 inch	8:00 am - .0063 inch	8:00 am - .0072 inch	8:00 am - .0078 inch
9:00 am - .0044 inch	9:00 am - .0060 inch	9:00 am - .0072 inch	9:00 am - .0077 inch	9:00 am - .0081 inch
10:00 am - .0045 inch	10:00 am - .0060 inch	10:00 am - .0072 inch	10:00 am - .0077 inch	10:00 am - .0081 inch
11:00 am - .0045 inch	11:00 am - .0061 inch	11:00 am - .0072 inch	11:00 am - .0077 inch	11:00 am - .0081 inch
12:00 pm - .0045 inch	12:00 pm - .0061 inch	12:00 pm - .0072 inch	12:00 pm - .0077 inch	12:00 pm - .0081 inch
1:00 pm - .0045 inch	1:00 pm - .0060 inch	1:00 pm - .0072 inch	1:00 pm - .0077 inch	1:00 pm - .0081 inch
2:00 pm - .0045 inch	2:00 pm - .0060 inch	2:00 pm - .0071 inch	2:00 pm - .0077 inch	2:00 pm - .0081 inch
3:00 pm - .0045 inch	3:00 pm - .0060 inch	3:00 pm - .0071 inch	3:00 pm - .0077 inch	3:00 pm - .0081 inch
4:00 pm - .0045 inch	4:00 pm - .0060 inch	4:00 pm - .0071 inch	4:00 pm - .0077 inch	4:00 pm - .0081 inch

Thursday 04/29/2021 Displacement	Friday 04/30/2021 Displacement	Tuesday 5/12/09 Displacement	Wednesday 5/13/09 Displacement	Thursday 5/14/09 Displacement
8:00 am - .0082 inch	8:00 am - .0085 inch	8:00 am - .0166 inch	8:00 am - .0186 inch	8:00 am - .0240 inch
9:00 am - .0084 inch	9:00 am - .0163 inch	9:00 am - .0180 inch	9:00 am - .0225 inch	9:00 am - .0432 inch
10:00 am - .0084 inch	10:00 am - .0164 inch	10:00 am - .0182 inch	10:00 am - .0230 inch	10:00 am - .0442 inch
11:00 am - .0084 inch	11:00 am - .0165 inch	11:00 am - .0183 inch	11:00 am - .0232 inch	11:00 am - .0448 inch
12:00 pm - .0084 inch	12:00 pm - .0165 inch	12:00 pm - .0184 inch	12:00 pm - .0233 inch	12:00 pm - .0453 inch
1:00 pm - .0084 inch	1:00 pm - .0165 inch	1:00 pm - .0184 inch	1:00 pm - .0235 inch	1:00 pm - .0455 inch
2:00 pm - .0084 inch	2:00 pm - .0165 inch	2:00 pm - .0184 inch	2:00 pm - .0235 inch	2:00 pm - .0457 inch
3:00 pm - .0083 inch	3:00 pm - .0165 inch	3:00 pm - .0184 inch	3:00 pm - .0236 inch	3:00 pm - .0458 inch
4:00 pm - .0084 inch	4:00 pm - .0165 inch	4:00 pm - .0184 inch	4:00 pm - .0236 inch	4:00 pm - .0458 inch

Monday 5/18/09 Displacement	Tuesday 5/19/09 Displacement	Wednesday 5/20/09 Displacement	Thursday 5/21/09 Displacement	Tuesday 5/26/09 Displacement
8:00 am - .0515 inch	8:00 am - .0451 inch	8:00 am - .0551 inch	8:00 am - .0490 inch	8:00 am - .0573 inch
9:00 am - Ramp	9:00 am - Ramp	9:00 am - Ramp	9:00 am - Ramp	9:00 am - .0180 inch
10:00 am - Ramp	10:00 am - Ramp	10:00 am - Ramp	10:00 am - Ramp	10:00 am - .0182 inch
11:00 am - Ramp	11:00 am - Ramp	11:00 am - Ramp	11:00 am - Ramp	11:00 am - .0183 inch
12:00 pm - Ramp	12:00 pm - Ramp	12:00 pm - Ramp	12:00 pm - .0560 inch	12:00 pm - .0184 inch
1:00 pm - .0454 inch	1:00 pm - .0541 inch	1:00 pm - .0492 inch	1:00 pm - .0563 inch	1:00 pm - .0184 inch
2:00 pm - .0453 inch	2:00 pm - .0544 inch	2:00 pm - .0491 inch	2:00 pm - .0564 inch	2:00 pm - .0184 inch
3:00 pm - .0453 inch	3:00 pm - .0545 inch	3:00 pm - .0491 inch	3:00 pm - .0566 inch	3:00 pm - .0184 inch
4:00 pm - .0453 inch	4:00 pm - .0546 inch	4:00 pm - .0491 inch	4:00 pm - .0566 inch	4:00 pm - .0184 inch

APPENDIX F:

CONSTRUCTING MULTIPLE SYNTHETIC FRACTURE NETWORK MODELS TO BUILD SYNTHETIC CORES USING 3D PRINTING TECHNOLOGY TO TEST THE FUNCTIONALITY OF SMART MICROCHIPS FOR FRACTURE MAPPING IN THE LAB

Synthetic fracture networks are designed based on the 2D scanned core images (JPG format) of fracture networks subsurface at different measured depths (9490-9493 ft, 9560-9563 ft, 9566-9569 ft) and different levels of geometry complexity. To generate inputs for the i-Geo Sensing, the format of the input data is required to be Cartesian coordinates or transformable to Cartesian coordinates (preferably TXT format). As a result, additional image processing steps are further conducted to achieve the desirable TXT input format.

Core samples' images are imported and transformed into grayscale, which is further capable of separating irrelevant pixels from fracture networks' pixels (i.e. "fractured" pixels). In a gray-sale image, pixels are scaled in their intensity values, which vary in a scale between 0-255. Initial analysis for a gray-scale image typically starts with its histogram of pixel intensity. The separation process is performed by Otsu image segmentation. Otsu algorithm chooses the optimal value from an image's histogram of pixel intensity and further detaches the image into two fragments: the main fracture network (which has pixel intensity 255 - white) and irrelevant pixel body (which has pixel intensity 0 - black). Albeit Otsu segmentation can extract the closest version to the desired base fracture networks, supporting algorithms are necessary to extract the desirable and complete synthetic fracture networks. The supporting algorithms include pixel filling (i.e. filling fractured pixels into desired voids), pixel sampling (i.e. dividing a fracture network into smaller fragments to perform more effective pixel filling), and pixel tracking (i.e. recovering a group of fractured pixels in a fracture's network fragment).

Desirable base fracture networks extracted from the scanned core images are maintained as 2D images (PVG format). Under the assumption that the propagation of a fracture network is uniform along the remaining dimension, commercial 3D editing & printing software as Blender® conducts extension of the 2D imaging base fracture networks into 3D imaging fracture networks (STL format).

The stored format of 3D imaging fracture networks is capable of being seamlessly processed from Blender® and randomly sampled to create synthetic input geo-sensor data from Smart Microchip Proppants as Cartesian coordinates. Figure 4 provides the projected 2D overviews of synthetic 3D imaging fracture networks used in this study. For design purposes, the synthetic fracture networks in Figure F.2 increase complexity from left to right. The 1st synthetic network (left) is composed of 4 fractures with almost uniformity in shape. The 2nd synthetic network (middle) is composed of 3 fractures and one smaller network with moderate non-uniformity in shape and low complexity in branching. The 3rd synthetic network (right) is composed of 1 fracture and two smaller networks with non-uniformity in shape and high complexity in branching.

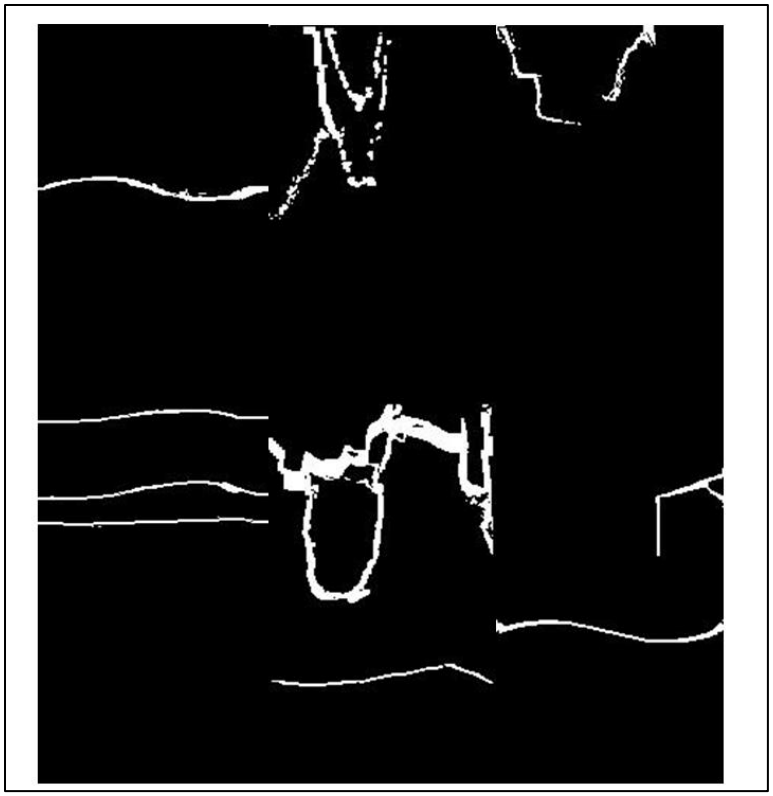


Figure F.1: Projected 2D overviews of the synthetic fracture networks

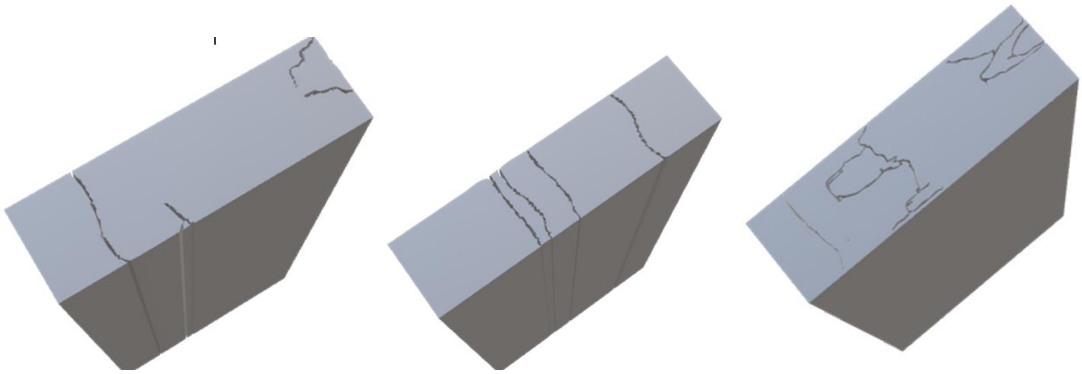


Figure F.2: 3D synthetic fracture network from core sample 2

The following final setup of the complex fracture geometry is 3D printed using the high-temperature material and will be used for the next level laboratory testing of MicroChips with varying sizes. The 3D-printed synthetic core dimensions are 0.115 m × 0.115 m × 0.15 m (4.5 inch × 4.5 inch × 5.9 inch).

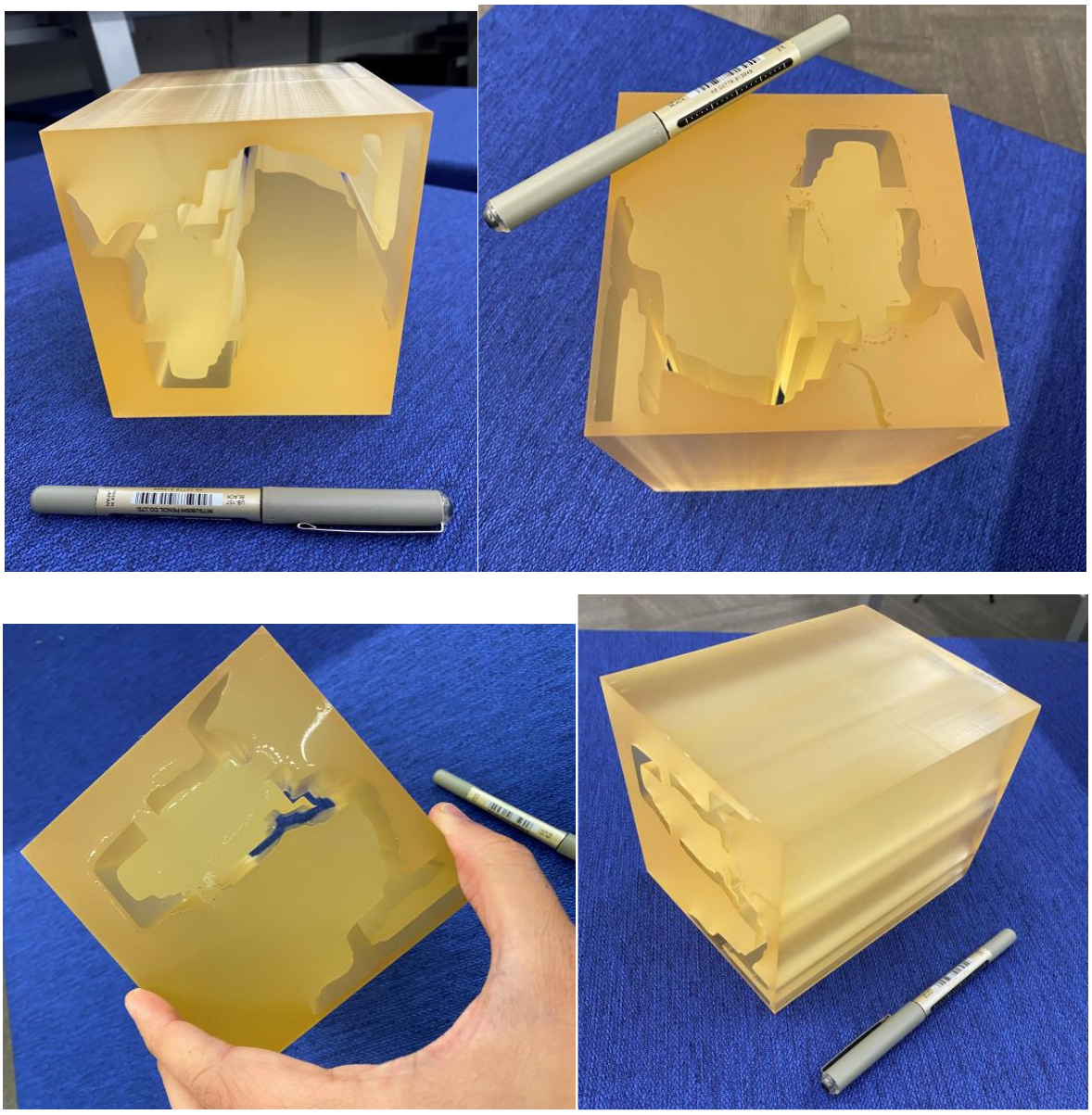


Figure F.3 – 3D printed Synthetic Core with the complex fracture geometry for the Microchips Testing

APPENDIX G:

I-GEO SENSING GRAPHICAL USER INTERFACE

As described throughout in section 6 of this report, i-Geo Sensing contributes two major algorithmic workflows and one supportive Design of Experiments module. i-Geo Sensing, additionally, comes with a Graphical User Interface (GUI) that allows users to interact and analyzes the technical aspects of the embedded workflows. This Appendix G provides a walk-through of the GUI in i-Geo Sensing.

Inside the code package includes a README.txt file that reads the instructions to install the appropriate environment (preferably Anaconda 3) to run i-Geo Sensing. As i-Geo Sensing is written in Python 3.10 and has a web-based interactive interface, the initial launch of the i-Geo Sensing brings a user a similar capture as in Figure G.1.

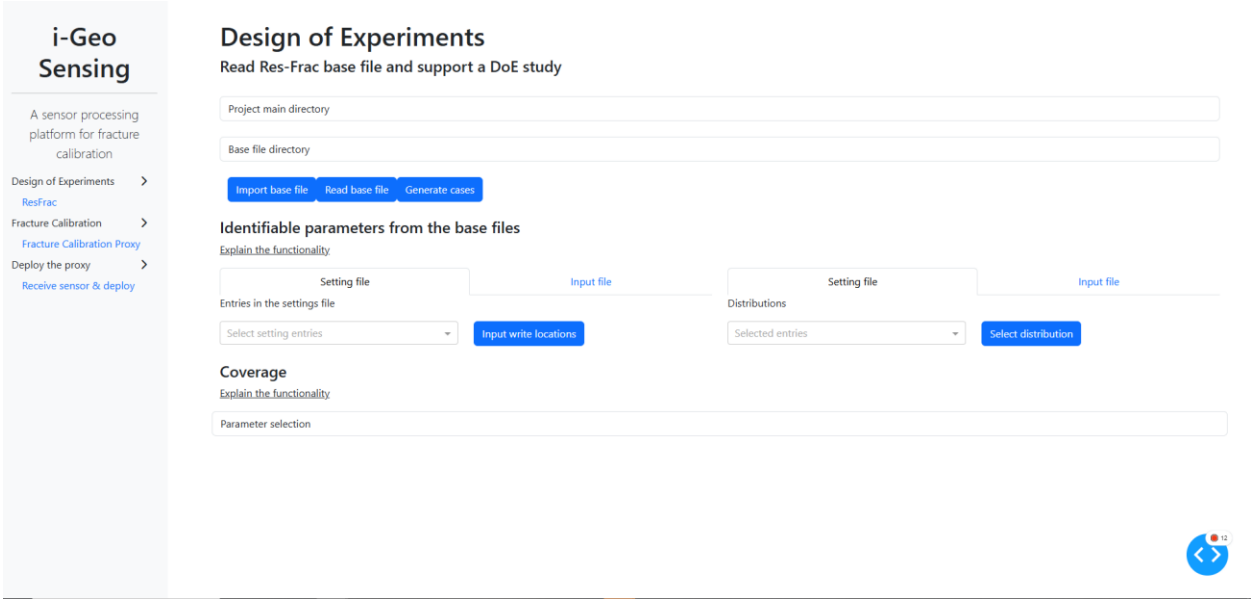


Figure G.1: Welcome interface in i-Geo Sensing

The welcome interface in i-Geo Sensing loads the Design of Experiments module by default (corresponds to 6.7 and 6.8). As seen in the left-sided task bar, users may find the two other modules: “Fracture Calibration Proxy” and “Receive Sensor & deploy”. The module “Fracture Calibration Proxy” corresponds to 6.9, 6.10 and 6.11. The module “Fracture Calibration Proxy” loads the processing of the Micro Chips’ geo-location data in the synthetic environment (presented as local directory with necessary data as the geo-location data).

Within the “Design of Experiments” module, users are required to input the module’ project main directory and the directory which contains the simulation files in ResFrac® (and is considered as the location of the base case in this module). A base case is provided in the code package for convenience, named “doe_simulation_2 SOP”. This folder contains all simulation results that are downloadable from ResFrac® server for the synthetic environment described in 6.2. Users now may click “Import base file” to allow i-Geo Sensing the access ability to the base case’s folder and

click “Read base file” to allow i-Geo Sensing parsing the settings and input text files that define the base case. (referred to 6.7). The following notification shall pop-up:

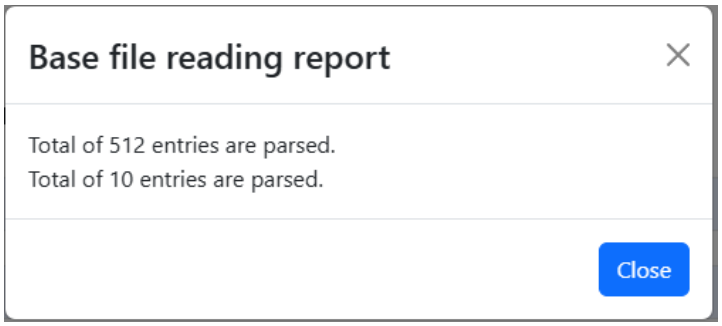


Figure G.2: Reading report in -Geo Sensing for the base case

Users are notified, similarly to Figure G.2, that i-Geo Sensing parses the text files successfully and recognizes the entries encoded in them (referred to 6.7 about the definition of an entry in ResFrac®). At this point, users may move to the “Identifiable parameters from the base files” section in the module. In the “Setting file” tab on the left, users now can select the entries that are preferable to conduct a Design of Experiments study/data generation for the supervised ML workflow in i-Geo Sensing. For example, selecting the entry “maxtrixcurvesets” brings the users to Figure G.3.

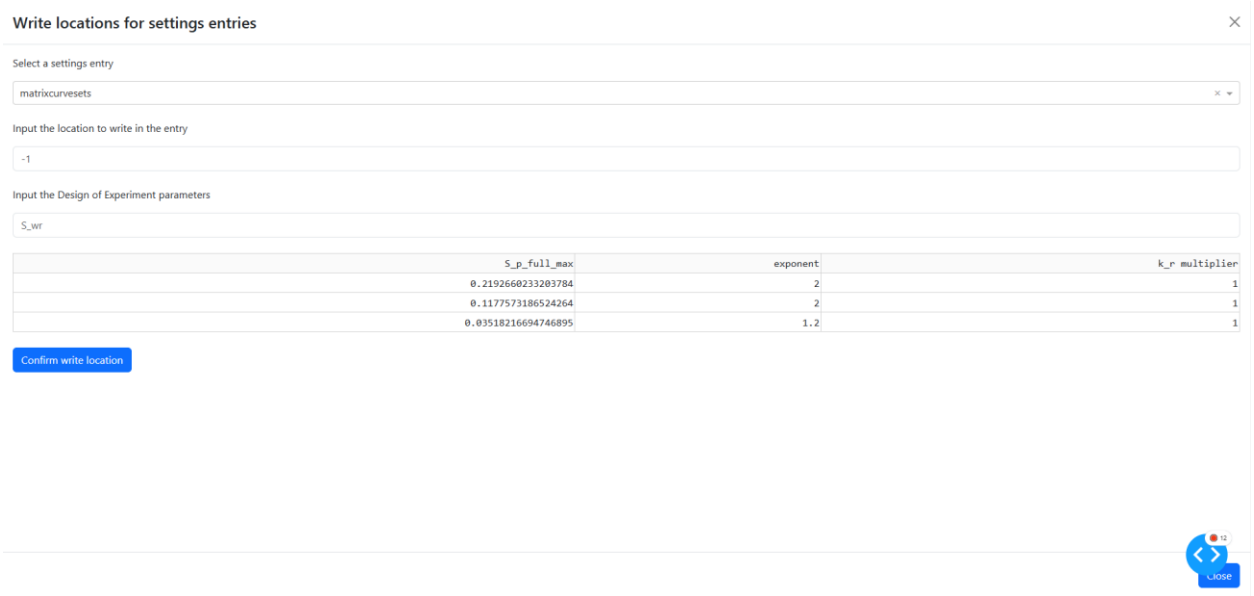


Figure G.3: An example pop-up window to request the entry’s necessary inputs for data generation

In this pop-up, users are allowed to select an entry (in case more than one entry is selected for data generation, users are required to select all entries in any order) and select the location that writes the changes to create new Design of Experiments case. In most of the entries, the location to write is -1, however some specific entries have multiple locations to write. i-Geo Sensing supports displaying certain specific entries’ written locations for the users’ convenience (referred to Figure G.3). In this pop-up, user may see an input named “Input the Design of Experiment parameters”.

Users may enter the names of the parameters that are later read by i-Geo Sensing as the parameters to be used in the supervised ML workflow. For example, in case the users need to study the water saturation, they may enter "S_wr" in the input box named above. Users shall click "Confirm write location" per entry to enable the recording of the written locations to i-Geo Sensing. After completing, users may close this pop-up.

Now users shall move to the "Setting file" tab on the right (Figure G.4) and select the entries one more time. Note that, under this selection, users shall see the entry list that is selected previously on the "Setting file" tab on the left, as now i-Geo Sensing limits the selection options to the entries that users request as being further used for data generation (referred to 6.7). After clicking on "Select distribution", a pop-up window as Figure G.5 appears.

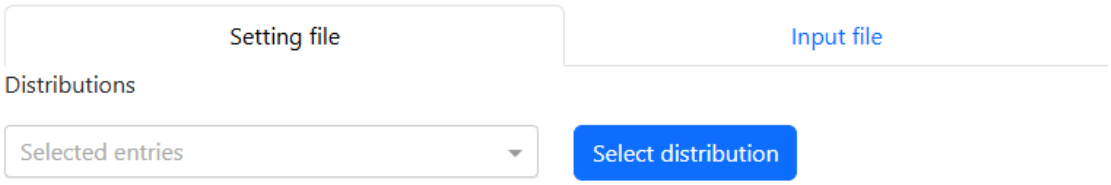


Figure G.4: Distribution section in the Design of Experiments module

The list of supported distributions (e.g., normal, log-normal, gamma, beta) are provided as dropdown in the pop-up window as Figure G.5. As a reminder, an entry may come with multiple DoE parameters, therefore users shall pay attention in choosing the correct entry and its DoE parameters, one at a time. In the input box "Input parameters for the distribution", users have to provide all required parameters that define the distribution of interest. For example, a normal distribution must require an input of 2 numbers, i.e., mean and standard deviation, separated by a comma between the numbers. I-Geo Sensing shall parse the numbers for the users appropriately.

After providing all necessary inputs to define a distribution for a specific DoE parameter, users shall click "Confirm distribution" to complete the process (per entry, per DoE parameter).

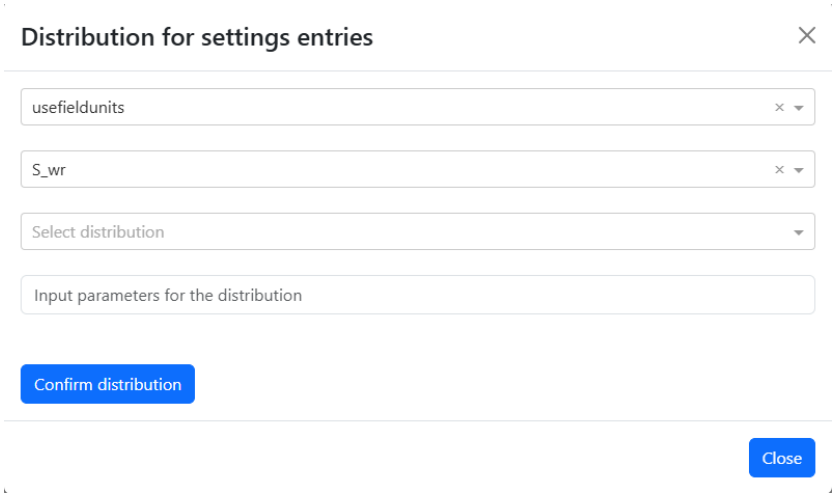


Figure G.5: Pop-up window for distribution of a DoE parameter

At this time, i-Geo Sensing receives all required inputs to deploy a batch data generation for a DoE. Users shall proceed to click on “Generate cases” in the module to open the pop-up for the DoE execution. This pop-up is similar to Figure G.6. The information displayed in Figure G.6 shall be straight-forward enough for the users to follow. In Figure G. 6, the input boxes for “Amount” refers to the amount of DoE cases to be generated in a single batch. The input boxes for “Last case” and “Batch number” receive any non-negative number, although users are recommended to input them in a sequential manner when referenced to the previous batch run. For example, in case a DoE batch run was previously executed to generate 100 cases indexed from 0-100 (batch 0), the next batch run is expected to generate an additional 100 cases indexed from 101 to 200 (batch 1). Therefore, users shall input 0 in the box “Ast case” and 1 in the box “Batch number” (provided that the batch 0 was run previously). After entering all inputs in this pop-up window, users click “Generate” and expect a wait time for the batch run to finish. After a batch run is executed, users may find inside the directory provided to save the DoE files, a similar look as Figure G.7.

As described in 6.7, the support of i-Geo Sensing to ResFrac® is semi-automatic, and therefore users are required to import all generated settings/input files into ResFrac® GUI and batch run the corresponding simulations to ResFrac® server. Users have to save changes in ResFrac® GUI prior to submitting the files to the server, otherwise simulations may not be completed properly.

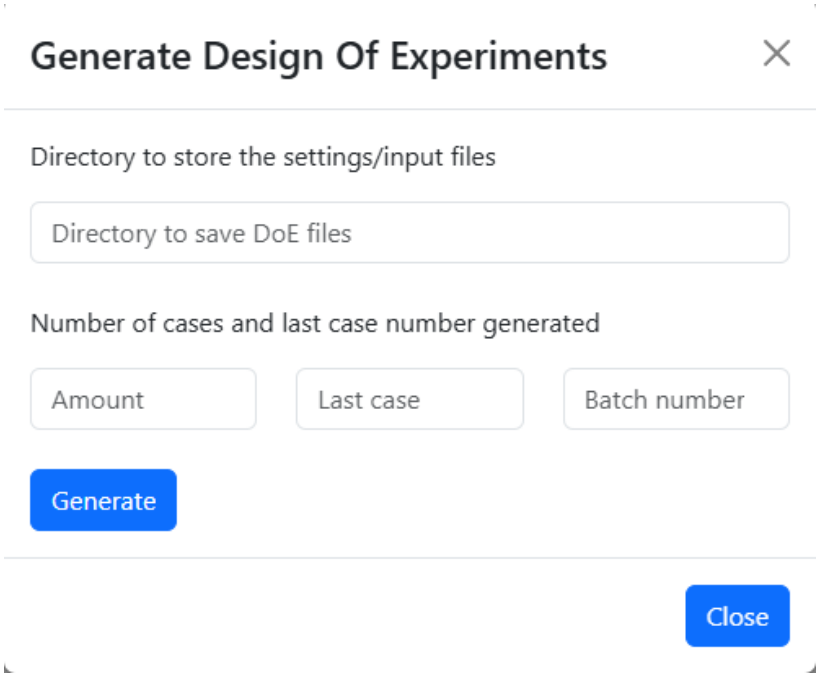


Figure G.6. The pop-up to execute a batch data generation

After all simulations are completed and downloaded from the ResFrac® server, users shall find simulation results for all generated cases from the current batch run in i-Geo Sensing, under the folders having naming conventions that are similar to Figure 126. Users may keep those folders in-place or move to another directory of interest, however users have to provide i-Geo Sensing the directory of which the folders are located/moved into later in the “Fracture Calibration” module.

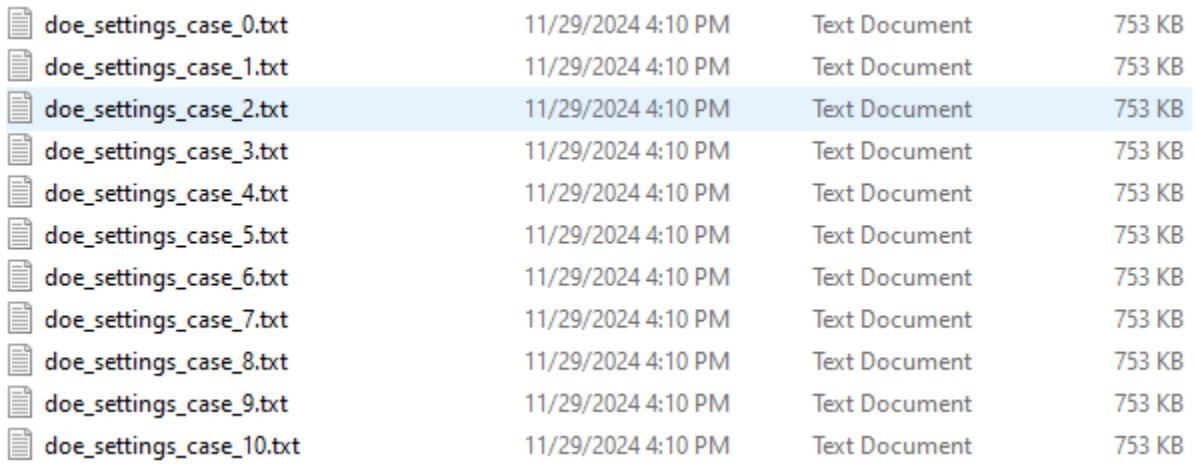


Figure G.7. An example of generated files post DoE batch run execution

At the bottom of the Design of Experiments module, users find the "Coverage" section. In this section, users may input the DoE parameters that were previously inputted to generate the recent batch run, and generate the coverage plot (i.e., a "joint" plot as in Figure 123). This section is designed to provide users an overview of the current batch run's efficacy in covering the expected DoE area for the parameters of study.

Below the "Design of Experiments" module is the "Fracture Calibration Proxy" module (Figure G.8). This module is essentially central to the supervised ML workflow. Similar to the "design of Experiments" module, users require to input the project main directory in which all proxy models are stored and re-loaded, if necessary, and select the task that the supervised ML workflow shall be trained and deployed for (e.g., fracture calibration or history-matching, referred to 6.9). The look of the "Fracture Calibration Proxy" module is practically similar to the "Design of Experiments" module. Therefore, users have a better view of the module's functionality at this point.

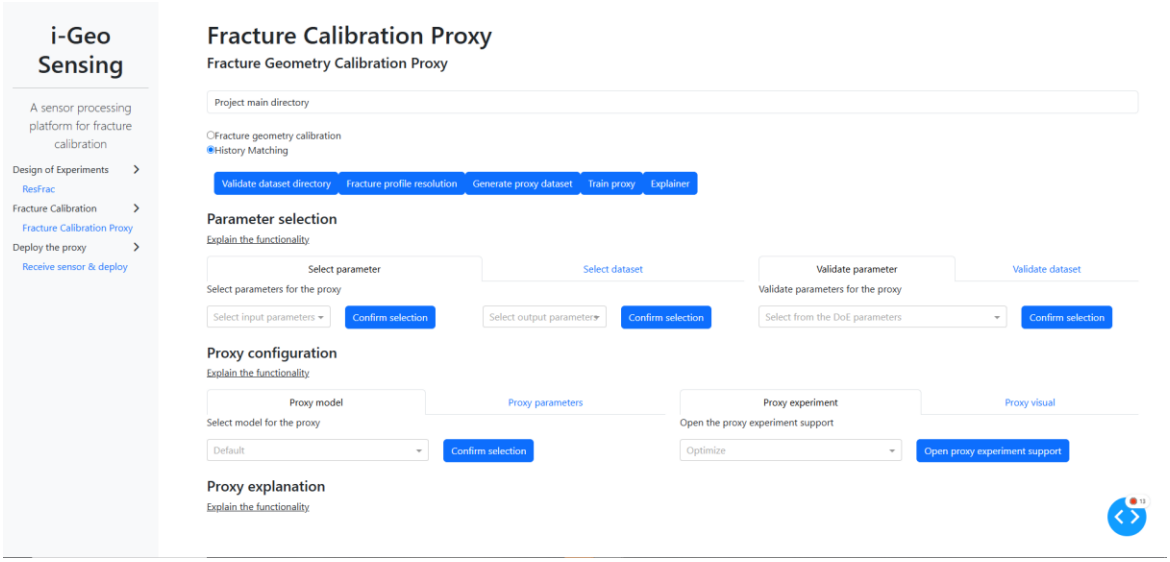


Figure G.8: The "Fracture Calibration Proxy" module

As recalled in the “Design of Experiments” module, users input certain DoE parameters to execute a batch run. Since the “Fracture Calibration Proxy” module uses the information from the “Design of Experiments” module, it receives those DoE parameters. Therefore, users shall see them in the “Select parameter” tab. Next to the “Select parameter” tab is the “Select dataset” tab, in which users may find all folders that are within the provided project main directory above. Users may select any of those folders and click “Confirm selection” to request to i-Geo Sensing that the selected folder(s) shall be used as training and/or validation data.

To ensure that the selected folder(s) mentioned above do have the ResFrac® simulation results needed to the module, users may click “Validate dataset directory” to confirm. In case the folder contains necessary data, the pop-up as Figure G.9 appears.

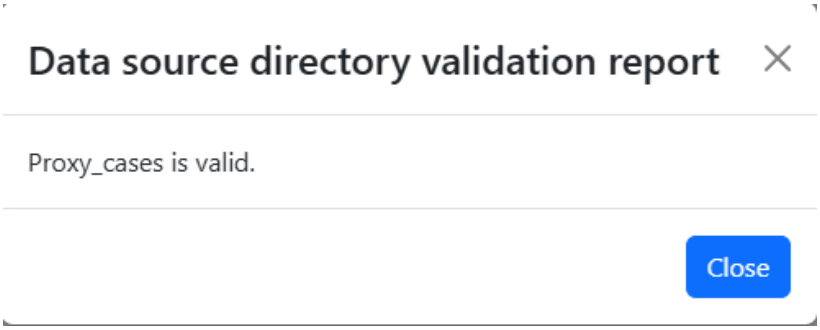


Figure G.9: A successful validation for the selected folder(s)

Recalled in 6.5 regarding the sensor data profiles, the Fracture Calibration Proxy” module provides a button named “Fracture profile resolution” for the users to click and input the necessary parameters that define a sensor data profile. They include the minimum & maximum values of the larger dimensions, and the resolution interval. As common sense in i-Geo Sensing, users shall click to confirm the inputs prior to existing the pop-up window. At this point, i-Geo Sensing receives enough information to generate the complete dataset to train and validate the proxy. Users need to click “Generate proxy dataset” and prompt to wait for a while before this process is complete.

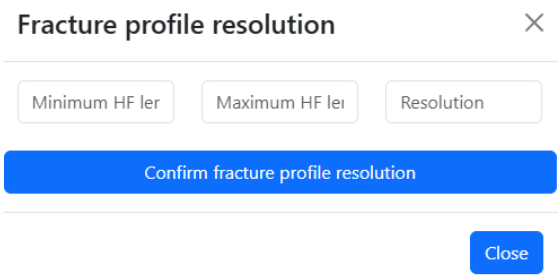


Figure G.10: Input pop-up window for the sensor data profile(s)

After the proxy dataset is generated, users may visualize the properties of the dataset via the “Validate parameter” and “Validate dataset” tabs. Under the “Validate parameter” tab users may open a pop-up window to visualize distributions of the DoE parameters, and time series data for the response parameters (e.g., BHP, production rate). An example is provided in Figure G.12. The “Validate dataset” is used to provide the users with any abnormality in the generated dataset. In

case the dataset has missing values, i-Geo Senning automatically in-place imputes those missing values.

Figure G.11: Completion notification for the proxy dataset generation

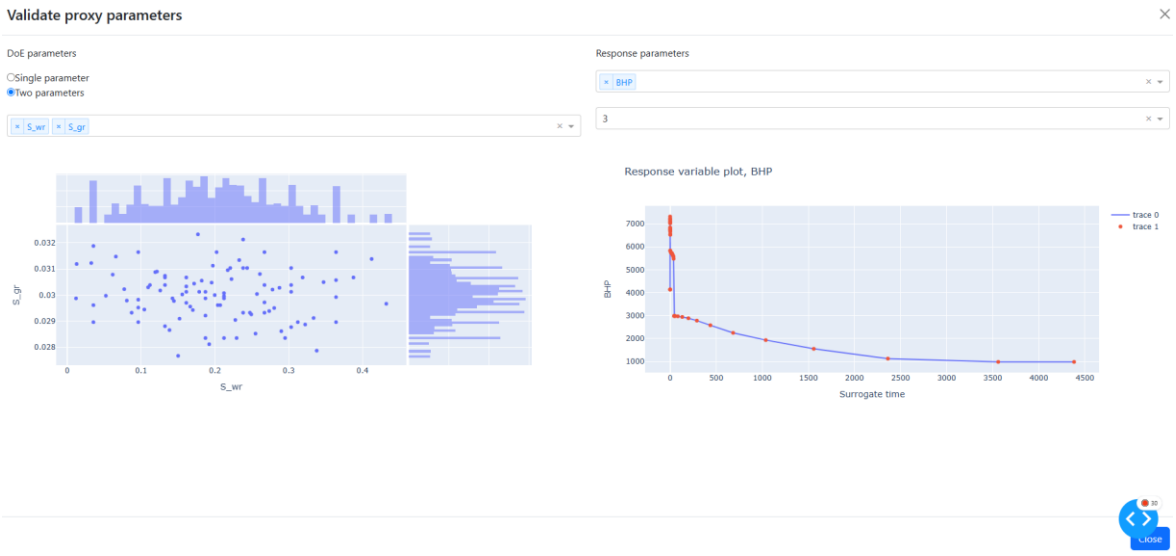


Figure G.12: An overview of the functional hidden in the "Validate parameter" tab

The "Proxy configuration" and "Proxy explanation" sections all refer to 6.9-6.11. Users have the option to select either the GBM or XGB to back the supervised ML workflow. As mentioned in 6.9, i-Geo Sensing limits certain hyperparameters to be tuned by the users, and consequently users may only see a subset of the model's hyperparameters compared to the official model's documentation (for example, referred to XGB: https://xgboost.readthedocs.io/en/stable/). For the ML experimenting functionality (found in "Proxy experiment" and "Proxy visual" tab, users may select either the "Fit" mode or the "Optimize" mode. The "Fit" mode essentially does not optimize the hyperparameters via a optimization space, in contrast the "Optimize" mode performs the optimization for the hyperparameters via an optimization space. Users may expect to access the model's explainability in the "Proxy explanation" section, in which plots similar to Figures 137-140 are provided.

The module "Receive sensor & deploy proxy" module shall automatically process outputs from both the "Design of Experiments" and the "Fracture Calibration Proxy" modules. Users only need to provide this module with the directory in which the synthetic Micro Chips' geo-location data is stored, and the module shall update periodically. The final proxy performance may look similar to Figures 133 and 135.

APPENDIX H:

I-GEO SENSING CODE EXCERPTS

```
.fa-chevron-right {
  transition: transform 0.2s ease-in-out 0s;
}

/* rotate the chevron when the open class is applied */
li.open .fa-chevron-right {
  transform: rotate(90deg);
}

.nav li {
  font-size: 18px;
}

.sidebar .nav-link {
  width: 100%;
  max-width: 100%;
  overflow: hidden;
  white-space: nowrap;
  display: flex;
  padding-right:1rem;
  padding-left:1rem;
  padding-top:1rem;
}
import dash
import dash_bootstrap_components as dbc
```

```
from dash import dcc
from dash import html, dash_table
from dash import Input, Output, State
from dash import callback, callback_context
from dash import DiskcacheManager

from dash.exceptions import PreventUpdate
from dash import callback_context as ctx
from dash import register_page

from dash_bootstrap_components import Tab, Table, InputGroup, Col, Row
from dash_bootstrap_components import Modal, ModalTitle, ModalBody, ModalHeader, ModalFooter
from dash_bootstrap_components import Placeholder

import diskcache
import os, shutil, time, timeit
import pickle, joblib, jsonpickle
import numpy as np
import pandas as pd

import scipy
from scipy.stats import distributions

from matplotlib import pyplot as plt
import seaborn as sns
import plotly
from plotly import express as px
from plotly import graph_objects as go
```

```
import hyperopt
from hyperopt import hp, space_eval
from hyperopt.pyll.base import scope

cache = diskcache.Cache("./cache")
background_callback_manager = DiskcacheManager(cache)
from base.base import *
from gui_utils.utils import *

from simulator.base import utils
from simulator.base import regex_templates
from simulator.base import parse
from simulator.simulation.simulation_helpers import *

from DoE.doe.doe_v1 import *



################################################################################
#############
################################## Page's main UI callbacks
###################################
################################################################################
#############


@callback(
    [Output("doe_main_dir_placeholder", "data")],
    [Input("doe_main_dir_gui", "value"),
```

```
     Input("doe_import_base_file", "n_clicks")],
    prevent_initial_call=True)
def get_doe_main_directory(main_dir, n_clicks):
    if n_clicks > 0:
        print("Main dir: ", main_dir)
        return [main_dir]
    else:
        raise PreventUpdate



@callback(
    [Output("doe_base_file_placeholder", "data")],
    [Input("doe_base_file_gui", "value"),
     Input("doe_import_base_file", "n_clicks")],
    prevent_initial_call=True)
def get_doe_main_directory(base_file, n_clicks):
    if n_clicks > 0:
        print("Base file: ", base_file)
        return [base_file]
    else:
        raise PreventUpdate



@callback(
    [Output("doe_settings_entry_placeholder", "data"),
     Output("doe_input_entry_placeholder", "data"),
     Output("doe_settings_var_names", "data"),
     Output("doe_input_var_names", "data")],
```

```
    [Input("doe_main_dir_placeholder", "data"),
     Input("doe_base_file_placeholder", "data"),
     Input("doe_read_base_file", "n_clicks")],
    prevent_initial_call=True)
def read_base_file(main_dir, base_file, n_clicks: int):
    if n_clicks > 0:
        settings_file_name = 'settings_' + base_file + '.txt'
        input_file_name = 'input_' + base_file + '.txt'
        base_file_dir = os.path.join(main_dir, base_file)
        settings_file_dir = os.path.join(base_file_dir, settings_file_name)
        input_file_dir = os.path.join(base_file_dir, input_file_name)
        all_settings_entries = utils.parse_file(file_name=settings_file_dir)
        all_input_entries = utils.parse_file(file_name=input_file_dir)
        #
        parsed_settings_entries = list()
        parsed_input_entries = list()
        for (_, setting_entry) in enumerate(all_settings_entries):
            parsed_ = parse.parse_entry(setting_entry)
            parsed_settings_entries.append(parsed_)
        for (_, input_entry) in enumerate(all_input_entries):
            parsed_ = parse.parse_entry(input_entry)
            parsed_input_entries.append(parsed_)
        #
        settings_file_size = len(all_settings_entries)
        input_file_size = len(all_input_entries)
        #
        settings_var_names = [parse.parse_entry(all_settings_entries[_]).variable_name for _ in
                              range(settings_file_size)]
```

```
        input_var_names = [parse.parse_entry(all_input_entries[_]).variable_name for _ in range(input_file_size)]
        return jsonpickle.encode(value=parsed_settings_entries), jsonpickle.encode(
            value=parsed_input_entries), settings_var_names, input_var_names
    else:
        raise PreventUpdate



@callback(
    Output("doe_settings_entry_input", "options"),
    Input("doe_settings_var_names", "data"),
    config_prevent_initial_callbacks=True
)
def parse_all_settings_entries(settings_var_names):
    if settings_var_names is not None:
        return settings_var_names
    else:
        raise PreventUpdate



@callback(
    Output("doe_input_entry_input", "options"),
    Input("doe_input_var_names", "data"),
    config_prevent_initial_callbacks=True
)
def parse_all_settings_entries(input_var_names):
    if input_var_names is not None:
        return input_var_names
    else:
```

```
        raise PreventUpdate



@callback(
    [Output("doe_notify_settings", "children"),
     Output("doe_notify_input", "children")],
    [Input("doe_settings_var_names", "data"),
     Input("doe_input_var_names", "data")],
    State("doe_read_base_file", "n_clicks"),
    config_prevent_initial_callbacks=True
)
def write_doe_read_base_file_report(settings_var_names, input_var_names, n_clicks):
    if n_clicks == 0:
        raise PreventUpdate
    else:
        setting_report = "Total of " + str(len(settings_var_names)) + " entries are parsed."
        input_report = "Total of " + str(len(input_var_names)) + " entries are parsed."
        return setting_report, input_report



@callback(
    Output("doe_settings_write_loc_placeholder", "data"),
    Input("doe_settings_entry_input", "value"),
    config_prevent_initial_callbacks=True
)
def update_settings_write_loc_placeholder(settings_var_names):
    if settings_var_names is not None:
        return settings_var_names
```

```
    else:
        raise PreventUpdate


@callback(
    Output("doe_settings_write_loc_dropdown", "options"),
    Input("doe_settings_write_loc_placeholder", "data"),
    config_prevent_initial_callbacks=True
)
def update_settings_write_loc_dropdown(settings_var_names):
    if settings_var_names is not None:
        return settings_var_names
    else:
        raise PreventUpdate


@callback(
    Output("doe_settings_write_loc_entry", "children"),
    [Input("doe_settings_write_loc_dropdown", "value"),
     Input("doe_settings_entry_placeholder", "data")],
    config_prevent_initial_callbacks=True
)
def display_settings_write_loc_entry(settings_var_name, settings_entries):
    if settings_var_name is not None:
        settings_entry = None
        settings_entries_ = jsonpickle.decode(settings_entries)
        for (_, settings_entry_) in enumerate(settings_entries_):
            if settings_entry_.variable_name == settings_var_name:
```

```
                settings_entry = settings_entry_
            if settings_var_name == 'matrixcurvesets':
                matrix_rel_perm = settings_entry.value_struct.value_struct[0]['matrixrelperm'][-1]
                return convert_numpy_to_data_table(matrix_rel_perm, columns=['S_p_full_max',
                                                   'exponent', 'k_r multiplier'])
            elif settings_var_name == 'facieslist':
                facies_list = settings_entry.value_struct.value_struct
                return convert_dict_to_data_table(facies_list, columns=None)
            elif settings_var_name not in irregular_variable_names:
                return str(settings_entry.value_struct.value_struct)
            else:
                # TODO: Complete this to display correct data
                return dash.no_update
        else:
            raise PreventUpdate


@callback(
    Output("doe_params_dist_placeholder", "data"),
    [Input("doe_settings_dist_confirm", "n_clicks"),
     Input("doe_settings_dist_entry_dropdown", "value"),
     Input("doe_settings_dist_params_dropdown", "value"),
     Input("doe_settings_dist_dropdown", "value")],
    State("doe_params_dist_placeholder", "data"),
    config_prevent_initial_callbacks=True
)
def update_all_distributions(n_clicks, entry_name, param_name, dist_name, all_dists):
    print('All distributions: ', all_dists)
```

```
    if callback_context.triggered_id == "doe_settings_dist_confirm" and n_clicks:
        if bool(all_dists) is False:
            all_dist = {'entry_name': [entry_name], 'param_name': [param_name], 'dist_name': [dist_name]}
            return all_dist
        else:
            all_dists_ = all_dists
            if 'entry_name' in all_dists_.keys():
                all_dists_['entry_name'] = all_dists['entry_name'] + [entry_name]
            if 'param_name' in all_dists_.keys():
                all_dists_['param_name'] = all_dists['param_name'] + [param_name]
            if 'dist_name' in all_dists_.keys():
                all_dists_['dist_name'] = all_dists['dist_name'] + [dist_name]
            return all_dists_
    else:
        raise PreventUpdate



@callback(
    Output("doe_params_write_locs_placeholder", "data"),
    [Input("doe_settings_write_loc_confirm", "n_clicks"),
     Input("doe_settings_write_loc_dropdown", "value"),
     Input("doe_settings_write_loc_input", "value"),
     Input("doe_settings_params", "value")],
    State("doe_params_write_locs_placeholder", "data"),
    config_prevent_initial_callbacks=True
)
def update_all_write_locations(n_clicks, entry_name, write_loc, param_names, all_write_locs):
    print('All write locations: ', all_write_locs)
```

```
    if callback_context.triggered_id == "doe_settings_write_loc_confirm" and n_clicks:
        param_names_ = param_names.split(sep=",")
        if bool(all_write_locs) is False:
            return {'entry_name': [entry_name], 'write_loc': [write_loc], 'param_names': [param_names_]}
        else:
            all_write_locs_ = all_write_locs
            if 'entry_name' in all_write_locs_.keys():
                all_write_locs_['entry_name'] = all_write_locs['entry_name'] + [entry_name]
            if 'param_name' in all_write_locs_.keys():
                all_write_locs_['write_loc'] = all_write_locs['write_loc'] + [write_loc]
            if 'dist_name' in all_write_locs_.keys():
                all_write_locs_['param_names'] = all_write_locs['param_names'] + [param_names_]
            return all_write_locs_
    else:
        raise PreventUpdate



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

################################## Page's modal callbacks
###################################

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



@callback(
    Output("doe_settings_write_loc_modal", "is_open"),
    [Input("doe_settings_entry_write_loc", "n_clicks"),
     Input("doe_settings_write_loc_button", "n_clicks")],
```

```
    [State("doe_settings_write_loc_modal", "is_open")],
)
def toggle_doe_settings_write_loc_modal(n_write_loc, n_write_loc_modal,
                                        is_open):
    if n_write_loc or n_write_loc_modal:
        return not is_open
    else:
        return is_open


@callback(
    Output("doe_notify_read_base_file_modal", "is_open"),
    [Input("doe_read_base_file", "n_clicks"),
     Input("doe_notify_read_base_file_button", "n_clicks")],
    [State("doe_notify_read_base_file_modal", "is_open")],
)
def toggle_doe_notify_read_base_file_modal(n_read_base_file, n_read_base_file_modal,
                                           is_open):
    if n_read_base_file or n_read_base_file_modal:
        return not is_open
    return is_open


@callback(
    Output("doe_settings_dist_modal", "is_open"),
    [Input("doe_settings_entry_dist", "n_clicks"),
     Input("doe_settings_dist_button", "n_clicks")],
    State("doe_settings_dist_modal", "is_open"),
```

```
)
def toggle_doe_settings_dist_modal(n_dist, n_dist_modal,
                                   is_open):
    if n_dist or n_dist_modal:
        return not is_open
    else:
        return is_open



@callback(
    Output("doe_case_generator_modal", "is_open"),
    [Input("doe_generate_cases", "n_clicks"),
     Input("doe_case_generator_button", "n_clicks")],
    State("doe_case_generator_modal", "is_open"),
)
def toggle_doe_case_generator_modal(n_case, n_case_modal,
                                    is_open):
    if n_case or n_case_modal:
        return not is_open
    else:
        return is_open
from base.base import *
from gui_utils.utils import *
from gui_utils.experimental import *

from workflow.surrogate import *
from workflow.objective_function import *
from workflow.calibrate_fracture import *
```

```
from workflow.history_match import *

from proxy.proxy_experiment import *



################################################################################
#############
################################### Page's modal callbacks
#################################
################################################################################
#############



@callback(
    Output("frac_cal_frac_profile_modal", "is_open"),
    [Input("frac_cal_profile_res", "n_clicks"),
     Input("frac_cal_frac_profile_button", "n_clicks")],
    [State("frac_cal_frac_profile_modal", "is_open")],
    config_prevent_initial_callbacks=True
)
def toggle_fracture_profile_modal(n_profile_res, n_profile_button, is_open):
    if n_profile_res or n_profile_button:
        return not is_open
    else:
        return is_open



@callback(
    Output("frac_cal_val_data_dir_modal", "is_open"),
```

```
    [Input("frac_cal_validate_data_dir", "n_clicks"),
     Input("frac_cal_val_data_dir_button", "n_clicks")],
    [State("frac_cal_val_data_dir_modal", "is_open")],
    config_prevent_initial_callbacks=True
)
def toggle_validate_data_dir_modal(n_val_dir, n_val_dir_button, is_open):
    if n_val_dir or n_val_dir_button:
        return not is_open
    else:
        return is_open


@callback(
    Output("frac_cal_generate_data_modal", "is_open"),
    [Input("frac_cal_generate_data", "n_clicks"),
     Input("frac_cal_generate_data_button", "n_clicks")],
    [State("frac_cal_generate_data_modal", "is_open")],
    config_prevent_initial_callbacks=True
)
def toggle_generate_proxy_data_modal(n_gen_data, n_gen_data_button, is_open):
    if n_gen_data or n_gen_data_button:
        return not is_open
    else:
        return is_open


@callback(
    Output("frac_cal_validate_params_modal", "is_open"),
```

```
    [Input("frac_cal_proxy_config_right_params_validate", "n_clicks"),
     Input("frac_cal_validate_params_button", "n_clicks")],
    [State("frac_cal_validate_params_modal", "is_open")],
    config_prevent_initial_callbacks=True
)
def toggle_validate_proxy_params_modal(n_val_params, n_val_params_button, is_open):
    if n_val_params or n_val_params_button:
        return not is_open
    else:
        return is_open


@callback(
    Output("frac_cal_validate_datasets_modal", "is_open"),
    [Input("frac_cal_proxy_config_right_sources_validate", "n_clicks"),
     Input("frac_cal_validate_datasets_button", "n_clicks")],
    [State("frac_cal_validate_datasets_modal", "is_open")],
    config_prevent_initial_callbacks=True
)
def toggle_validate_proxy_datasets_modal(n_val_data, n_val_data_button, is_open):
    if n_val_data or n_val_data_button:
        return not is_open
    else:
        return is_open


@callback(
    Output("frac_cal_proxy_hyper_params_modal", "is_open"),
```

```
    [Input("frac_cal_proxy_params_left_confirm", "n_clicks"),
     Input("frac_cal_proxy_hyper_params_button", "n_clicks")],
    [State("frac_cal_proxy_hyper_params_modal", "is_open")],
    config_prevent_initial_callbacks=True
)
def toggle_proxy_hyper_params_modal(n_proxy_params, n_proxy_params_button, is_open):
    if n_proxy_params or n_proxy_params_button:
        return not is_open
    else:
        return is_open


@callback(
    Output("frac_cal_train_proxy_modal", "is_open"),
    [Input("frac_cal_train_proxy", "n_clicks"),
     Input("frac_cal_train_proxy_button", "n_clicks")],
    [State("frac_cal_train_proxy_modal", "is_open")],
    config_prevent_initial_callbacks=True
)
def toggle_train_proxy_modal(n_train, n_train_button, is_open):
    if n_train or n_train_button:
        return not is_open
    else:
        return is_open


@callback(
    Output("frac_cal_proxy_exp_modal", "is_open"),
```

```
    [Input("frac_cal_proxy_exp_right_confirm", "n_clicks"),
     Input("frac_cal_proxy_exp_button", "n_clicks")],
    [State("frac_cal_proxy_exp_modal", "is_open")],
)
def toggle_proxy_expriment(n_exp, n_exp_button, is_open):
    if n_exp or n_exp_button:
        return not is_open
    else:
        return is_open



################################################################################
#############
################################## Page's storage callbacks
##################################
################################################################################
#############


@callback(
    Output("frac_cal_project_main_dir_store", "data"),
    Input("frac_cal_proxy_main_dir", "value"),
    config_prevent_initial_callbacks=True
)
def store_project_main_dir(main_dir):
    return main_dir



@callback(
```

```
    Output("frac_cal_frac_profile_res_store", "data"),
    Input("frac_cal_frac_profile_confirm", "n_clicks"),
    [State("min_hf_length_input", "value"),
     State("max_hf_length_input", "value"),
     State("resolution", "value")],
    config_prevent_initial_callbacks=True
)
def store_fracture_profile_resolution(n_clicks, min_hf, max_hf, res):
    if n_clicks:
        print(min_hf, max_hf, res)
        return {"min_hf": min_hf, "max_hf": max_hf, "res": res}
    else:
        raise PreventUpdate



@callback(
    Output("frac_cal_proxy_input_params_store", "data"),
    [Input("frac_cal_proxy_config_input_params_dropdown", "value"),
     Input("frac_cal_proxy_config_input_params_confirm", "n_clicks")],
    config_prevent_initial_callbacks=True
)
def store_proxy_input_params(input_params, n_clicks):
    if n_clicks:
        if experimental_mode:
            return experimental_doe_params
        else:
            return input_params
    else:
```

```
        raise PreventUpdate



@callback(
    Output("frac_cal_proxy_output_params_store", "data"),
    [Input("frac_cal_proxy_config_output_params_dropdown", "value"),
     Input("frac_cal_proxy_config_output_params_confirm", "n_clicks")],
    config_prevent_initial_callbacks=True
)
def store_proxy_output_params(output_params, n_clicks):
    if n_clicks:
        if experimental_mode:
            return experimental_response_params
        else:
            return output_params
    else:
        raise PreventUpdate



@callback(
    Output("frac_cal_data_sources_store", "data"),
    Input("frac_cal_proxy_config_left_sources_confirm", "n_clicks"),
    State("frac_cal_proxy_config_left_sources_dropdown", "value"),
    config_prevent_initial_callbacks=True
)
def store_proxy_data_sources(n_clicks, data_sources):
    if n_clicks:
        return data_sources
```

```
    else:
        raise PreventUpdate


@callback(
    Output("frac_cal_proxy_hyper_params_store", "data"),
    Input("frac_cal_proxy_params_left_confirm", "n_clicks"),
    State("frac_cal_proxy_params_left_dropdown", "value"),
    config_prevent_initial_callbacks=True
)
def store_proxy_hyper_params(n_clicks, hyper_params):
    if n_clicks:
        return hyper_params
    else:
        raise PreventUpdate


@callback(
    Output("frac_cal_proxy_hyper_params_dist_store", "data"),
    Input("frac_cal_proxy_hyper_params_confirm", "n_clicks"),
    [State("frac_cal_proxy_hyper_params_name", "value"),
     State("frac_cal_proxy_hyper_params_dist", "value"),
     State("frac_cal_proxy_hyper_params_dist_params", "value"),
     State("frac_cal_proxy_hyper_params_dist_store", "data")],
    config_prevent_initial_callbacks=True
)
def store_proxy_hyper_params_dist(n_clicks, param_name, param_dist, param_dist_params,
all_hyper_params_dists):
    if n_clicks:
```

```
        param_dist_params_ = param_dist_params.split(sep=",")
        print(param_dist_params_)
        if bool(all_hyper_params_dists) is False:
            all_hyper_params_dists_ = {'param_name': [param_name], 'param_dist': [param_dist],
                                       'param_dist_params': [param_dist_params_]}
            print("All proxy's hyper-parameters' distributions: ", all_hyper_params_dists_)
            return all_hyper_params_dists_
        else:
            all_hyper_params_dists_ = all_hyper_params_dists
            if param_name not in all_hyper_params_dists['param_name']:
                all_hyper_params_dists_['param_name'] = all_hyper_params_dists['param_name'] + \
                                                        [param_name]
                all_hyper_params_dists_['param_dist'] = all_hyper_params_dists['param_dist'] + \
                                                        [param_dist]
                all_hyper_params_dists['param_dist_params'].append(param_dist_params_)
                all_hyper_params_dists_['param_dist_params'] =
all_hyper_params_dists['param_dist_params']
            else:
                pass
            print("All proxy's hyper-parameters' distributions: ", all_hyper_params_dists_)
            return all_hyper_params_dists_
    else:
        raise PreventUpdate


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

################################## Page's main UI callbacks
##################################
```

```
################################## Parameter selection ######################################

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

@callback(
    Output("frac_cal_proxy_config_left_sources_dropdown", "options"),
    Input("frac_cal_project_main_dir_store", "data"),
    config_prevent_initial_callbacks=True
)
def extract_proxy_data_sources(main_dir):
    """

    :param main_dir:
    :return:
    """
    print('Main dir to extract: ', main_dir)
    workflows_dir = os.path.join(main_dir, "workflows")
    if os.path.exists(workflows_dir) is True:
        return [f for f in os.listdir(workflows_dir)]
    else:
        return dash.no_update


@callback(
    Output("frac_cal_data_main_dir_text", "children"),
    [Input("frac_cal_project_main_dir_store", "data"),
     Input("frac_cal_data_sources_store", "data")],
```

```
    config_prevent_initial_callbacks=True
)
def validate_proxy_data_sources(main_dir, data_sources):
    """
    Validate whether the provided data sources have data for the proxy
    :param main_dir:
    :param data_sources:
    :return:
    """
    print('Data sources to extract: ', data_sources)
    workflows_dir = os.path.join(main_dir, "workflows")
    validate_text = ""
    if main_dir is None or data_sources is None:
        raise PreventUpdate
    else:
        for (_, data_source) in enumerate(data_sources):
            data_source_valid = validate_res_frac_workflows_dir(workflows_dir, data_source)
            if data_source_valid:
                validate_text += data_source + " is valid. \n"
            else:
                validate_text += data_source + "is NOT valid. \n"
        return validate_text



@callback(
    Output("frac_cal_generate_data_text", "children"),
    [State("frac_cal_project_main_dir_store", "data"),
     State("frac_cal_frac_profile_res_store", "data"),
```

```
    State("frac_cal_proxy_input_params_store", "data"),
    State("frac_cal_proxy_output_params_store", "data"),
    State("frac_cal_data_sources_store", "data")],
   Input("frac_cal_generate_data", "n_clicks"),
   background=True,
   running=[(Output("frac_cal_generate_data", "disabled"), True, False),
            (Output("frac_cal_generate_data_button", "disabled"), True, False)
            ],
   config_prevent_initial_callbacks=True
)
def generate_proxy_datasets(main_dir, fracture_profile,
                            doe_params, response_params, data_sources, n_clicks):
   generate_proxy_datasets_text = ""
   if callback_context.triggered_id == "frac_cal_generate_data" and n_clicks > 0:
      fracture_profile_resolution = [float(fracture_profile["min_hf"]),
float(fracture_profile["max_hf"]),
                                     int(fracture_profile["res"])]
      for (_, data_source) in enumerate(data_sources):
         proxy_data_file = data_source + "_data.csv"
         proxy_data_dir = os.path.join(main_dir, proxy_data_file)
         if os.path.exists(proxy_data_dir):
            proxy_df = pd.read_csv(proxy_data_dir, index_col=0, header=0)
         else:
            result_dir = os.path.join(main_dir, "workflows")
            result_dir = os.path.join(result_dir, data_source)
            result_dir = os.path.join(result_dir, "simulations")
            surrogate_manager = SurrogateDirectory(result_dir=result_dir)
            surrogate_manager.experimental_doe_params = doe_params
```

```
surrogate_manager.init_fracture_profile(fracture_profile_resolution=fracture_profile_resolution)

surrogate_manager.init_reservoir_response(reservoir_response_var_names=response_params)
            if experimental_mode:
                surrogate_manager.init_experimental_doe_data()
            proxy_df                                                                    =
surrogate_manager.assemble_surrogate_directory(surrogate_dir=proxy_data_dir)
        generate_proxy_datasets_text_ = "Generate proxy data from source " + data_source + ", "
        generate_proxy_datasets_text_ += " containing " + str(proxy_df.shape[0]) + " rows and "
+ \
                                    str(proxy_df.shape[-1]) + " columns. "
        generate_proxy_datasets_text += generate_proxy_datasets_text_
    return generate_proxy_datasets_text
  else:
    raise PreventUpdate



@callback(
  Output("frac_cal_proxy_config_right_params_dropdown", "options"),
  [Input("frac_cal_proxy_input_params_store", "data"),
   Input("frac_cal_proxy_output_params_store", "data")],
  config_prevent_initial_callbacks=True
)
def retrieve_proxy_input_params(doe_params, response_params):
  if experimental_mode:
    return experimental_doe_params + experimental_response_params
  else:
    return doe_params + response_params
```

```
@callback(
    Output("frac_cal_proxy_config_right_sources_dropdown", "options"),
    Input("frac_cal_data_sources_store", "data"),
    config_prevent_initial_callbacks=True
)
def retrieve_proxy_data_sources(data_sources):
    if experimental_mode:
        return experimental_data_sources
    else:
        return data_sources



@callback(
    Output("frac_cal_datasets_store", "data"),
    [Input("frac_cal_data_sources_store", "data"),
     Input("frac_cal_project_main_dir_store", "data")],
    config_prevent_initial_callbacks=True
)
def retrieve_datasets(data_sources, main_dir):
    datasets = dict()
    for (_, data_source) in enumerate(data_sources):
        dataset_file = data_source + "_data.csv"
        dataset_dir = os.path.join(main_dir, dataset_file)
        if os.path.exists(dataset_dir):
            df = pd.read_csv(dataset_dir, index_col=0, header=0).to_dict("records")
            datasets[data_source] = df
        else:
```

```
            pass
    return datasets



@callback(
    Output("frac_cal_validate_doe_params_dropdown", "options"),
    Input("frac_cal_proxy_config_right_params_dropdown", "value"),
    config_prevent_initial_callbacks=True
)
def retrieve_validated_doe_params(params):
    if experimental_mode:
        doe_params_ = list()
        for p in params:
            if p not in experimental_response_params:
                doe_params_.append(p)
        return doe_params_
    else:
        return params



@callback(
    Output("frac_cal_validate_res_params_dropdown", "options"),
    Input("frac_cal_proxy_config_right_params_dropdown", "value"),
    config_prevent_initial_callbacks=True
)
def retrieve_validated_res_params(params):
    if experimental_mode:
        res_params_ = list()
```

```
        for p in params:
            if p not in experimental_doe_params:
                res_params_.append(p)
        return res_params_
    else:
        return params


@callback(
    Output("frac_cal_validate_case_dropdown", "options"),
    Input("frac_cal_proxy_config_right_sources_dropdown", "value"),
    State("frac_cal_datasets_store", "data"),
    config_prevent_initial_callbacks=True
)
def retrieve_cases(data_source, datasets):
    dataset = pd.DataFrame(data=datasets[data_source])
    cases = dataset['case'].to_numpy()
    cases = list(np.unique(cases))
    print('All cases: ', cases)
    return cases


@callback(
    Output("frac_cal_validate_doe_params_content", "children"),
    [Input("frac_cal_validate_doe_params_dropdown", "value"),
     Input("frac_cal_proxy_config_right_sources_dropdown", "value"),
     Input("frac_cal_validate_doe_params_radio_items", "value")],
    State("frac_cal_datasets_store", "data"),
```

```
    config_prevent_initial_callbacks=True
)
def visualize_doe_distributions(params, data_source, plot_type, datasets):
    if datasets is None:
        raise PreventUpdate
    else:
        print('DoE params: ', params)
        dataset = datasets[data_source]
        if plot_type == "Single parameter":
            fig = px.histogram(dataset, x=params[0])
            graph_id = params[0] + '_dist_plot'
            return dcc.Graph(figure=fig, id=graph_id)
        elif plot_type == "Two parameters":
            param_0 = params[0]
            param_1 = params[1]
            graph_id = params[0] + '_' + params[1] + '_joint_plot'
            fig = px.scatter(dataset, x=param_0, y=param_1, marginal_x="histogram",
                             marginal_y="histogram")
            return dcc.Graph(figure=fig, id=graph_id)
        else:
            raise PreventUpdate



@callback(
    Output("frac_cal_validate_res_params_content", "children"),
    [Input("frac_cal_proxy_config_right_sources_dropdown", "value"),
     Input("frac_cal_validate_res_params_dropdown", "value"),
     Input("frac_cal_validate_case_dropdown", "value")],
```

```
    State("frac_cal_datasets_store", "data"),
    config_prevent_initial_callbacks=True
)
def visualize_res_distributions(data_source, params, case, datasets):
    dataset = pd.DataFrame(datasets[data_source])
    dataset = dataset[dataset['case'] == case]
    if len(params) != 1:
        raise PreventUpdate
    else:
        print('Response params: ', params)
        fig_title = 'Response variable plot, ' + params[0]
        graph_id = params[0] + '_res_plot'
        fig = go.Figure()
        fig.add_trace(go.Scatter(x=dataset.loc[:, 'surrogate_time'],
                                 y=dataset.loc[:, params[0]], mode='lines'))
        fig.add_trace(go.Scatter(x=dataset.loc[:, 'surrogate_time'],
                                 y=dataset.loc[:, params[0]], mode='markers'))
        fig.update_layout(title=fig_title)
        fig.update_xaxes(title_text='Surrogate time')
        fig.update_yaxes(title_text=params[0])
        return dcc.Graph(figure=fig, id=graph_id)



@callback(
    Output("frac_cal_validate_datasets_content", "children"),
    Input("frac_cal_proxy_config_right_sources_dropdown", "value"),
    [State("frac_cal_project_main_dir_store", "data"),
     State("frac_cal_datasets_store", "data")],
```

```
    config_prevent_initial_callbacks=True
)
def validate_datasets(data_source, main_dir, datasets):
    dataset = pd.DataFrame(datasets[data_source])
    validate_text = ""
    if dataset.isnull().values.any():
        num_na = dataset.isna().sum().sum()
        dataset_ = dataset.fillna(value=0, inplace=False)
        dataset_file_ = data_source + '_data_.csv'
        dataset_dir_ = os.path.join(main_dir, dataset_file_)
        dataset_.to_csv(dataset_dir_)
        validate_text += "Data source at " + data_source + " has " + str(num_na) + " nan values. " \
                                                    "In-place correction is made."
        return validate_text
    else:
        dataset_ = dataset
        dataset_file_ = data_source + '_data_.csv'
        dataset_dir_ = os.path.join(main_dir, dataset_file_)
        dataset_.to_csv(dataset_dir_)
        raise PreventUpdate


################################################################################
#############
################################## Page's main UI callbacks
##################################
################################## Proxy configuration
########################################
################################################################################
#############
```

```
@callback(
    Output("frac_cal_proxy_params_left_dropdown", "options"),
    Input("frac_cal_proxy_model_confirm", "n_clicks"),
    State("frac_cal_proxy_model_left_dropdown", "value"),
    config_prevent_initial_callbacks=True
)
def retrieve_proxy_hyper_params(n_clicks, model_name):
    if n_clicks:
        if model_name == "GBM":
            return gbm_hyper_params
        elif model_name == "XGB":
            return xgb_hyper_params
        else:
            return default_hyper_params
    else:
        raise PreventUpdate



@callback(
    Output("frac_cal_proxy_hyper_params_name", "options"),
    Input("frac_cal_proxy_hyper_params_store", "data"),
    config_prevent_initial_callbacks=True
)
def update_proxy_hyper_params(hyper_params):
    return hyper_params
```

```
@callback(
    Output("frac_cal_proxy_hyper_params_content", "children"),
    Input("frac_cal_proxy_hyper_params_confirm", "n_clicks"),
    [State("frac_cal_proxy_hyper_params_name", "value"),
     State("frac_cal_proxy_hyper_params_dist", "value"),
     State("frac_cal_proxy_hyper_params_dist_params", "value")],
    config_prevent_initial_callbacks=True
)
def visualize_proxy_hyper_params_dist(n_clicks, param_name, param_dist, param_dist_params):
    if n_clicks:
        print("Dist for: ", param_name, param_dist)
        param_dist_params_ = param_dist_params.split(sep=",")
        param_dist_params_ = [float(_) for _ in param_dist_params_]
        if param_dist == "uniform":
            param_dist_generator = distributions.uniform(param_dist_params_[0], param_dist_params_[-1])
            param_dist_data = param_dist_generator.rvs(1000)
        elif param_dist == "normal":
            param_dist_generator = distributions.norm(param_dist_params_[0], param_dist_params_[-1])
            param_dist_data = param_dist_generator.rvs(1000)
        else:
            param_dist_generator = distributions.lognorm(param_dist_params_[0], param_dist_params_[-1])
            param_dist_data = param_dist_generator.rvs(1000)
        param_dist_data = pd.DataFrame({param_name: param_dist_data})
        fig = px.histogram(data_frame=param_dist_data, x=param_name)
        graph_id = param_name + '_distribution'
        return dcc.Graph(figure=fig, id=graph_id)
```

```
    else:
        raise PreventUpdate


@callback(
    [Output("frac_cal_proxy_exp_inputs", "options"),
     Output("frac_cal_proxy_exp_outputs", "options")],
    Input("frac_cal_proxy_config_right_params_validate", "n_clicks"),
    State("frac_cal_proxy_config_right_params_dropdown", "value"),
    config_prevent_initial_callbacks=True
)
def retrieve_proxy_exp_params(n_clicks, proxy_params):
    if n_clicks:
        return proxy_params, proxy_params
    else:
        raise PreventUpdate


@callback(
    Output("frac_cal_proxy_exp_content", "children"),
    # Button to execute the proxy experiment run(s)
    Input("frac_cal_proxy_exp_exec", "n_clicks"),
    [   # Type of proxy task
        State("task_to_use_proxy", "value"),
        # Type of proxy
        State("frac_cal_proxy_exp_proxy_type", "value"),
        # Fit or Optimize this experiment
        State("frac_cal_proxy_exp_right_dropdown", "value"),
```

```
        # Proxy's hyper-parameter distributions
        State("frac_cal_proxy_hyper_params_dist_store", "data"),
        # Define the experiment
        State("frac_cal_proxy_exp_dir", "value"),
        State("frac_cal_proxy_exp_name", "value"),
        State("frac_cal_proxy_exp_description", "value"),
        # Define the run
        State("frac_cal_proxy_exp_run_name", "value"),
        State("frac_cal_proxy_exp_model_folder", "value"),
        State("frac_cal_proxy_exp_artifact_folder", "value"),
        # Data sources, inputs and outputs
        State("frac_cal_proxy_exp_inputs", "value"),
        State("frac_cal_proxy_exp_outputs", "value"),
        State("frac_cal_data_sources_store", "data"),
        State("frac_cal_project_main_dir_store", "data")
    ], config_prevent_initial_callbacks=True
)
def run_proxy_exp(n_clicks, proxy_task, proxy_type,
                  exp_mode, hyper_params_dists,
                  exp_dir, exp_name, exp_description,
                  run_name, model_folder, artifact_folder,
                  input_variables, output_variables, data_source, main_dir):
    if n_clicks:
        # Create data object (XGBDataset)
        print('Create data object ...')
        dataset_file = data_source + '_data_.csv'
        dataset_dir = os.path.join(main_dir, dataset_file)
        df = pd.read_csv(dataset_dir, index_col=0, header=0)
```

```
df_cols = list(df.columns)
data_object = XGBDataset(df=df)
fracture_profile_cols = list()
for col in df_cols:
    if 'z_' in col:
        fracture_profile_cols.append(col)
if proxy_task == "Fracture geometry calibration":
    data_object.x_cols = data_object.time_param + fracture_profile_cols
else:
    data_object.x_cols = input_variables
data_object.y_cols = output_variables
# Create proxy object (QuantileXGBRegressor)
print('Create proxy object ...')
proxy_object = QuantileXGBRegressor(xgb_data=data_object)
# Create MLFlowProxyWrapper
print('Create proxy wrapper object ...')
proxy_wrapper = MLFlowProxyWrapper(proxy=proxy_object)
# Create hyper-parameter space
print('Create space ...')
space, proxy_exp_mode = create_proxy_hyper_params_space(hyper_params_dists, proxy_type, exp_mode)
# Create experiment object
print('Create experiment object ...')
experiment_object = ProxyExperiment(mlflow_proxy_wrapper=proxy_wrapper)
experiment_object.experiment_dir = exp_dir
experiment_object.experiment_name = exp_name
experiment_object.experiment_description = exp_description
experiment_object.experiment_mode = proxy_exp_mode
print('Execute the experiment & run ...')
```

```
        # Execute experiment & corresponding run
        if proxy_exp_mode == 'fit':
            experiment_object.log_new_fit_experiment(run_name, model_folder, artifact_folder, space)
        else:
            experiment_object.log_new_opt_experiment(run_name, model_folder, artifact_folder, space)
    else:
        raise PreventUpdate


################################################################################################
################################## Page's main UI callbacks ##################################
################################## Proxy explanation ##################################
```

```
from base.base import *


def toggle_collapse(n, is_open):
    if n:
        return not is_open
    return is_open


def set_navitem_class(is_open):
    if is_open:
        return "open"
```

```
    return ""


def set_sidebar(app: dash.Dash):
    app.callback(
        Output("doe_sidebar_collapse", "is_open"),
        [Input("doe_sidebar", "n_clicks")],
        [State("doe_sidebar_collapse", "is_open")],
    )(toggle_collapse)

    app.callback(
        Output("doe_sidebar", "className"),
        [Input("doe_sidebar_collapse", "is_open")],
    )(set_navitem_class)

    app.callback(
        Output("frac_cal_sidebar_collapse", "is_open"),
        [Input("frac_cal_sidebar", "n_clicks")],
        [State("frac_cal_sidebar_collapse", "is_open")],
    )(toggle_collapse)

    app.callback(
        Output("frac_cal_sidebar", "className"),
        [Input("frac_cal_sidebar_collapse", "is_open")],
    )(set_navitem_class)

    app.callback(
        Output("proxy_deploy_sidebar_collapse", "is_open"),
```

```
        [Input("proxy_deploy_sidebar", "n_clicks")],
        [State("proxy_deploy_sidebar_collapse", "is_open")],
    )(toggle_collapse)

    app.callback(
        Output("proxy_deploy_sidebar", "className"),
        [Input("proxy_deploy_sidebar_collapse", "is_open")],
    )(set_navitem_class)
from DoE.doe.utils import *
from simulator.base.entry import *


################################################################################
# "Connection" classes to create and write new ResFrac simulation files
# Using the classes:
# 1. Entry: allow integrating with ResFrac entry system
# 2. ValueStruct: allow altering the data in a specific entry
################################################################################


class Distribution(object):
    # TODO: Set-up class to manage all DoE distributions (that alter value_struct
    # in a ResFrac's entry
    def __init__(self, entry: Entry):
        super(Distribution, self).__init__()
        self.entry = entry

    def verify_entry(self, expected_variable_name: str):
```

```
        try:
            assert self.entry.variable_name == expected_variable_name
        except AssertionError:
            warnings.warn('Incorrect entry.')
            sys.exit(1)

    def verify_value_struct(self, expected_value_struct):
        try:
            assert type(self.entry.value_struct.value_struct) == expected_value_struct
        except AssertionError:
            warnings.warn('Incorrect value struct.')
            sys.exit(1)

    def generate_sample(self, *args, **kwargs):
        pass



class SingleValueDistribution(Distribution):
    # Distribution class for an entry with single value
    def __init__(self, entry: Entry):
        super(Distribution, self).__init__()
        self.entry = entry

        self.value_type = None
        self.dist = None

    def set_value_type(self, value_type: str):
        self.value_type = value_type
```

```
    def set_dist(self, dist):
        self.dist = dist

    def generate_sample(self):
        sample_value = ' '  # TODO: ? Sampling a single value here
        new_value_struct = self.entry.value_struct.value_struct
        new_value_struct['Value(s):'] = sample_value
        self.entry.value_struct.change_value_struct(new_value_struct=new_value_struct)


class MatrixValueDistribution(Distribution):
    # Distribution class for an entry with matrix value (e.g., facies list)
    def __init__(self, entry: Entry):
        super(Distribution, self).__init__()
        self.entry = entry

        self.value_type = None
        self.dist = None

    def set_value_type(self, value_type: str):
        self.value_type = value_type

    def set_dist(self, dist):
        self.dist = dist

    def generate_sample(self, *args, **kwargs):
        pass
```

```
class LayerValueDistribution(MatrixValueDistribution):
    """
    Distribution class for an entry with "matrix" value, as follows:
    - Rows: number of layers for a property (as a col)
    - Cols: number of properties (a col is a property)
    """
    def __init__(self, entry: Entry):
        super(MatrixValueDistribution, self).__init__()
        self.entry = entry
        # Additional parameters to indicates layers and property names
        self.value_type = None
        self.dist = None
        self.property_names = None
        self.layer_names = None

    def set_property_names(self, property_names: list):
        self.property_names = property_names

    def set_layer_names(self, layer_names: list):
        self.layer_names = layer_names

    def generate_sample(self, loc: dict):
        for layer_name, property_name in loc.items():
            try:
                assert layer_name in self.layer_names
                assert property_name in self.property_names
```

```
            except AssertionError:
                warnings.warn('Input layer name: ' + layer_name + ' is incorrect.')
                warnings.warn(' OR ')
                warnings.warn('Input property name: ' + property_name + ' is incorrect.')
                sys.exit(1)
        for layer_name, property_name in loc.items():
            property_idx = self.property_names.index(property_name)
            sample_value = ' '  # TODO: ? Distribution sampling here
            new_value_struct = self.entry.value_struct.value_struct
            new_value_struct[layer_name][property_idx] = sample_value
            self.entry.value_struct.change_value_struct(new_value_struct=new_value_struct)


################################################################################
# Distribution classes for fracture model calibration entries
################################################################################


class RelativeFracToughnessDistribution(SingleValueDistribution):
    def __init__(self, entry: Entry):
        super(SingleValueDistribution, self).__init__()
        self.entry = entry
        self.value_type = ' '
        self.dist = ' '  # TODO: ? Set-up uniform distribution here


class FracGradientDistribution(LayerValueDistribution):
    def __init__(self, entry: Entry):
```

```
        super(LayerValueDistribution, self).__init__()
        self.entry = entry


class VerticalKDistribution(LayerValueDistribution):
    def __init__(self, entry: Entry):
        super(LayerValueDistribution, self).__init__()
        self.entry = entry


class HorizontalKDistribution(LayerValueDistribution):
    def __init__(self, entry: Entry):
        super(LayerValueDistribution, self).__init__()
        self.entry = entry


class NWComplexityDistribution(LayerValueDistribution):
    def __init__(self, entry: Entry):
        super(LayerValueDistribution, self).__init__()
        self.entry = entry

    def generate_sample(self):
        pass


class VerticalPropHolUpDistribution(SingleValueDistribution):
    # Distribution class to manage vertical prop flow holdup factor
    def __init__(self, entry: Entry):
```

```
        super(SingleValueDistribution, self).__init__()
        self.entry = entry


class MaxImmobilePropMassDistribution(SingleValueDistribution):
    # Distribution class to manage maximum immobilized prop mass per area
    def __init__(self, entry: Entry):
        super(SingleValueDistribution, self).__init__()
        self.entry = entry


################################################################################
# Distribution classes for production data HM entries
################################################################################


class GlobalPermMultiplierDistribution(SingleValueDistribution):
    def __init__(self, entry: Entry):
        super(SingleValueDistribution, self).__init__()
        self.entry = entry
        self.value_type = ''
        self.dist = ''  # TODO: ? Set-up uniform distribution here


class PorosityDistribution(LayerValueDistribution):
    def __init__(self, entry: Entry):
        super(LayerValueDistribution, self).__init__()
        self.entry = entry
```

```
    def generate_sample(self):
        self.verify_entry('facieslist')


class InitialSwDistribution(LayerValueDistribution):
    def __init__(self, entry: Entry):
        super(LayerValueDistribution, self).__init__()
        self.entry = entry

    def generate_sample(self):
        self.verify_entry('facieslist')


class RelPermCurveDistribution(Distribution):
    # Distribution class to manage rel perm curve (RelPermStruct)
    def __init__(self, entry: Entry):
        super(Distribution, self).__init__()
        self.entry = entry
        # Additional parameters for relative perm curves
        self.max_Sr = np.zeros(3)  # S_wr, S_or, S_gr
        self.kr_multipliers = np.zeros(3)  # k_rw, k_ro, k_rg
        self.Sr_dist = None
        self.kr_dist = None

    def set_saturation_dist(self, Sr_dist):
        self.Sr_dist = Sr_dist
```

```
    def set_kr_dist(self, kr_dist):
        self.kr_dist = kr_dist

    def generate_sample(self):
        self.verify_entry('matrixcurvesets')
        self.max_Sr = ''  # TODO: ? Implement distribution sampling here
        self.kr_multipliers = ''  # TODO: ? Implement distribution sampling here
        rel_perm_curve = np.zeros([3, 3])
        rel_perm_curve[:, 0] = self.Sr_dist
        rel_perm_curve[:, 1] = np.array([2, 2, 1.2])
        rel_perm_curve[:, -1] = self.kr_dist
        new_value_struct = self.entry.value_struct.value_struct
        new_value_struct['matrixrelperm'][-1] = rel_perm_curve
        self.entry.value_struct.change_value_struct(new_value_struct=new_value_struct)


class PDPDistribution(Distribution):
    # Distribution class to manage Pressure Dependent Permeability (RelPermStruct)
    def __init__(self, entry: Entry):
        super(Distribution, self).__init__()
        self.entry = entry
        # Additional parameters for relative perm curves
        self.pressures = np.zeros(3)  # S_wr, S_or, S_gr
        self.multipliers = np.zeros(3)  # k_rw, k_ro, k_rg
        self.pressure_dist = None
        self.multiplier_dist = None

    def set_pressure_dist(self, pressure_dist):
```

```
        self.pressure_dist = pressure_dist

    def set_kr_dist(self, multiplier_dist):
        self.multiplier_dist = multiplier_dist

    def generate_sample(self):
        self.verify_entry('matrixcurvesets')
        self.pressures = ''  # TODO: ? Implement distribution sampling here
        self.multipliers = ''  # TODO: ? Implement distribution sampling here
        pdp_data = np.zeros([3, 2])
        pdp_data[:, 0] = self.pressures
        pdp_data[:, -1] = self.multipliers
        new_value_struct = self.entry.value_struct.value_struct
        new_value_struct[('pressuredependentpermeability', 'reversible')][0] = pdp_data
        self.entry.value_struct.change_value_struct(new_value_struct=new_value_struct)


class StressAnisotropyDistribution(LayerValueDistribution):
    # Layers' Sh_max - Sh_min
    def __init__(self, entry: Entry):
        super(LayerValueDistribution, self).__init__()
        self.entry = entry


class NetToGrossDistribution(LayerValueDistribution):
    # Layers' net to gross ratios
    def __init__(self, entry: Entry):
        super(LayerValueDistribution, self).__init__()
```

```
        self.entry = entry
from DoE.doe.utils import *
from simulator.base.entry import *
from simulator.simulation import simulation_helpers, simulation_files


################################################################################
# "Connection" classes to create and write new ResFrac simulation files
# Using the classes:
# 1. Entry: allow integrating with ResFrac entry system
# 2. ValueStruct: allow altering the data in a specific entry
################################################################################


class Connector(object):
    def __init__(self, entry: Entry):
        """
        Base connector class, inherit this class to:
        1. Connect data from a DoE case to the correct entry's value struct
        2. Write data from a DoE case to the correct entry's value struct
        (depend on the type of the value struct)
        :param entry:
        """
        super(Connector, self).__init__()
        self.entry = entry  # The entry the connector shall write data into
        """
        :param case_loc: type int, the case number in DoE data (2D np.ndarray)
        :param data_loc: type int, the DoE parameter location in the DoE data
```

```
    :param write_loc: denote as follows:
    1. For single value: -1
    2. For value(s) in a list/1D array: list(int)
    3. For values(s) in a list of list/2D array: list(list(int))
    4. For value(s) in layer: list(layer name, property index)
    5. For irregular value structs:
        5.1. Relative perm: list(list(int)) for relative perm curve(s) & curve location
        5.2. PDP curve: list(list(int)) for pressure dependent perm curve(s) & curve location
    """
    self.case_loc = None  # The case number the DoE data shall locate
    self.data_loc = None  # The location(s) of the data in the DoE data
    self.write_loc = None  # The location(s) of the data in the entry's value struct

def set_doe_loc(self, case_loc: int, data_loc):
    # Set the case location & data location in the DoE data
    self.case_loc = case_loc
    self.data_loc = data_loc

def set_write_loc(self, write_loc):
    # Set the written location in the entry's value struct
    self.write_loc = write_loc

def verify_entry(self, expected_variable_name: str):
    # Verify the correct entry the connector shall connect
    # If incorrect entry, exit
    try:
        assert self.entry.variable_name == expected_variable_name
    except AssertionError:
```

```
            warnings.warn('Incorrect entry.')
            sys.exit(1)

    def verify_value_struct(self, expected_value_struct):
        # Verify the correct value struct type the connector shall connect
        # If incorrect value struct type, exit
        try:
            assert type(self.entry.value_struct.value_struct) == expected_value_struct
        except AssertionError:
            warnings.warn('Incorrect value struct.')
            sys.exit(1)

    def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
        # Write data from DoE (doe_data) to the entry's value_struct
        # Must override
        pass


class SingleValueConnector(Connector):
    def __init__(self, entry: Entry, data_type):
        """
        Connector class to connect DoE data of a single value (i.e., str, float, int)
        :param entry:
        """
        super(Connector, self).__init__()
        self.entry = entry
        self.write_loc = -1
        self.data_type = data_type
```

```
def verify_data_type(self, value):
    if self.data_type in [int, float]:
        try:
            value_ = self.data_type(value)
            return True
        except ValueError:
            return False
    elif self.data_type == bool:
        try:
            assert value.lower() in ['true', 'false']
            return True
        except AssertionError:
            try:
                assert self.data_type == str
                return True
            except AssertionError:
                return False


def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
    if self.case_loc is None or self.data_loc is None:
        warnings.warn('Case location and data location are required.')
        sys.exit(1)
    elif not self.verify_data_type(doe_data[self.case_loc, self.data_loc]):
        warnings.warn('Incorrect data type for a single value.')
        sys.exit(1)
    else:
        new_value_struct = doe_data[self.case_loc, self.data_loc]
```

```
            self.entry.change_values(new_value_struct=self.data_type(new_value_struct))


class ListValueConnector(Connector):
    def __init__(self, entry: Entry):
        """
        Connector class to connect DoE data of a list value
        (i.e., str, float, int)
        :param entry:
        """
        super(Connector, self).__init__()
        self.entry = entry
        self.write_loc = None  # Location(s) in the value struct the connector shall write into

    def set_write_loc(self, write_loc: List[int]):
        self.write_loc = write_loc

    def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
        self.verify_value_struct(expected_value_struct=list)
        if self.case_loc is None or self.data_loc is None:
            warnings.warn('Case location and data location are required.')
            sys.exit(1)
        elif self.write_loc is None:
            warnings.warn("Entry value' struct written location is required.")
            sys.exit(1)
        else:
            new_value_struct = self.entry.value_struct.value_struct
            assert len(self.write_loc) == len(self.data_loc)
```

```
            for (_, loc) in enumerate(self.write_loc):
                new_value_struct[self.write_loc[_]] = doe_data[self.case_loc, self.data_loc[_]]
            self.entry.change_values(new_value_struct=new_value_struct)


class ListOfListValueConnector(Connector):
    def __init__(self, entry: Entry):
        """
        Connector class to connect DoE data of a list of list value
        (i.e., str, float, int)
        :param entry:
        """
        super(Connector, self).__init__()
        self.entry = entry
        self.write_loc = None  # Location(s) in the value struct the connector shall write into

    def set_write_loc(self, write_loc: List[List[int]]):
        self.write_loc = write_loc

    def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
        self.verify_value_struct(expected_value_struct=list)
        if self.case_loc is None or self.data_loc is None:
            warnings.warn('Case location and data location are required.')
            sys.exit(1)
        elif self.write_loc is None:
            warnings.warn("Entry value' struct written location is required.")
            sys.exit(1)
        else:
```

```
        assert type(self.write_loc) == list
        assert len(self.write_loc) == 2
        assert type(self.data_loc) == list
        assert len(self.write_loc[0]) == len(self.write_loc[-1])
        assert len(self.write_loc[0]) == len(self.data_loc)
        new_value_struct = self.entry.value_struct.value_struct
        for _ in range(len(self.write_loc[0])):
            new_value_struct[self.write_loc[0][_]][self.write_loc[-1][_]] = \
                doe_data[self.case_loc, self.data_loc[_]]
        self.entry.change_values(new_value_struct=new_value_struct)


class Array1DValueConnector(Connector):
    def __init__(self, entry: Entry):
        """
        Connector class to connect DoE data of a 1D numpy array value
        (i.e., str, float, int)
        :param entry:
        """
        super(Connector, self).__init__()
        self.entry = entry
        self.write_loc = None

    def set_write_loc(self, write_loc: List[int]):
        self.write_loc = write_loc  # Location(s) in the value struct the connector shall write into

    def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
        self.verify_value_struct(expected_value_struct=np.ndarray)
```

```
        if self.case_loc is None or self.data_loc is None:
            warnings.warn('Case location and data location are required.')
            sys.exit(1)
        elif self.write_loc is None:
            warnings.warn("Entry value' struct written location is required.")
            sys.exit(1)
        else:
            new_value_struct = self.entry.value_struct.value_struct
            assert len(self.write_loc) == len(self.data_loc)
            for (_, loc) in enumerate(self.write_loc):
                new_value_struct[self.write_loc[_]] = doe_data[self.case_loc, self.data_loc[_]]
            self.entry.change_values(new_value_struct=new_value_struct)


class Array2DValueConnector(Connector):
    def __init__(self, entry: Entry):
        """
        Connector class to connect DoE data of a 2D numpy array value
        (i.e., str, float, int)
        :param entry:
        """
        super(Connector, self).__init__()
        self.entry = entry
        self.write_loc = None

    def set_write_loc(self, write_loc: Union[List[int], List[List[int]]]):
        self.write_loc = write_loc  # Location(s) in the value struct the connector shall write into
```

```
    def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
        self.verify_value_struct(expected_value_struct=np.ndarray)
        if self.case_loc is None or self.data_loc is None:
            warnings.warn('Case location and data location are required.')
            sys.exit(1)
        elif self.write_loc is None:
            warnings.warn("Entry value' struct written location is required.")
            sys.exit(1)
        else:
            assert type(self.write_loc) == list
            assert len(self.write_loc) == 2
            assert type(self.data_loc) == list
            assert len(self.write_loc[0]) == len(self.write_loc[-1])
            assert len(self.write_loc[0]) == len(self.data_loc)
            new_value_struct = self.entry.value_struct.value_struct
            for _ in range(len(self.write_loc[0])):
                new_value_struct[self.write_loc[0][_], self.write_loc[-1][_]] = \
                    doe_data[self.case_loc, self.data_loc[_]]
            self.entry.change_values(new_value_struct=new_value_struct)


class LayerValueConnector(ListOfListValueConnector):
    def __init__(self, entry: Entry, layer_names: List[str], prop_names: List[str]):
        """
        Connector class to connect DoE data of a "layer data" value struct
        (i.e., static data per layer, modulus properties, initial Sw per layer)
        :param entry:
        :param layer_names:
```

```
    :param prop_names:
    """
    super(ListOfListValueConnector, self).__init__(entry=entry)
    self.num_layers = None
    # Additional attributes to define layer & property names
    self.prop_names = prop_names
    self.layer_names = layer_names
    self.connected_prop_names = None
    self.connected_layer_names = None

def set_number_of_layers(self, num_layers: int):
    self.num_layers = num_layers

def set_connected_data(self, connected_prop_names: List[str],
                       connected_layer_names: List[str]):
    self.connected_prop_names = connected_prop_names
    self.connected_layer_names = connected_layer_names

def set_write_loc(self, write_loc=None):
    # Re-write this function to intake connected layer names & property names
    assert len(self.connected_layer_names) == len(self.connected_prop_names)
    write_loc_ = list()
    for (_, layer_name) in enumerate(self.connected_layer_names):
        prop_name = self.connected_prop_names[_]
        try:
            prop_idx = self.prop_names.index(prop_name)
            write_loc_.append([layer_name, prop_idx])
        except ValueError:
```

```
            warnings.warn('Provided property is not correct.')
            sys.exit(1)
        self.write_loc = write_loc_

    def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
        self.verify_value_struct(expected_value_struct=dict)
        if self.case_loc is None or self.data_loc is None:
            warnings.warn('Case location and data location are required.')
            sys.exit(1)
        elif self.write_loc is None:
            warnings.warn("Entry value' struct written location is required.")
            sys.exit(1)
        else:
            assert len(self.write_loc) == len(self.data_loc)
            new_value_struct = self.entry.value_struct.value_struct
            for (_, [layer_name, prop_idx]) in enumerate(self.write_loc):
                new_value_struct[layer_name][prop_idx] = doe_data[self.case_loc, self.data_loc[_]]
            self.entry.change_values(new_value_struct=new_value_struct)



def init_connector(entry: Entry, param_name: str):
    """
    # TODO: Return the correct connector class for an entry
    :param entry:
    :param param_name:
    :return: connector_object: the connector class object
    """
    try:
```

```
        assert entry.is_doe is True
    except AssertionError:
        warnings.warn('This is not a DoE entry.')
        sys.exit(1)
    if entry.length == 0:
        warnings.warn('An entry with 0 length can not be a DoE entry.')
        sys.exit(1)
    elif entry.length == 1:
        if type(entry.value_struct) is SingleValueStruct:
            connector_object = SingleValueConnector(entry=entry, data_type=None)
        elif type(entry.value_struct) is MatrixValueStruct:
            if type(entry.value_struct.value_struct) == np.ndarray:
                connector_object = Array1DValueConnector(entry=entry)
            else:
                connector_object = ListValueConnector(entry=entry)
        else:
            rel_perm_curve = -1
            if 'pdp' not in param_name:
                connector_object = RelPermCurveConnector(entry=entry,
rel_perm_curve=rel_perm_curve)
            else:
                connector_object = PDPConnector(entry=entry, rel_perm_curve=rel_perm_curve)
    else:
        if type(entry.value_struct) is FaciesListStruct:
            layer_names_ = ['']
            prop_names_ = ['']
            connector_object = LayerValueConnector(entry=entry, layer_names=layer_names_,
prop_names=prop_names_)
        else:
```

```
        if verify_1d_list(entry.value_struct.value_struct) is True:
            connector_object = ListValueConnector(entry=entry)
        elif verify_1d_array(entry.value_struct.value_struct) is True:
            connector_object = Array1DValueConnector(entry=entry)
        elif verify_1d_list(entry.value_struct.value_struct) is False:
            connector_object = ListOfListValueConnector(entry=entry)
        elif verify_1d_array(entry.value_struct.value_struct) is False:
            connector_object = Array2DValueConnector(entry=entry)
        else:
            warnings.warn('The entry value struct is not supported by any connectors. Set to base connector.')
            connector_object = Connector(entry=entry)
    return connector_object


################################################################################
# Distribution classes for production data HM entries
################################################################################


class RelPermCurveConnector(Connector):
    def __init__(self, entry: Entry, rel_perm_curve: int):
        """
        Connector class specified for rel perm curve (as np.ndarray)
        :param entry:
        """
        super(Connector, self).__init__()
        self.entry = entry
        self.rel_perm_curve = rel_perm_curve
```

```
    def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
        # This function to write rel perm curve at the correct location in 'matrixcurvesets' entry
        self.verify_entry(expected_variable_name='matrixcurvesets')
        new_value_struct = self.entry.value_struct.value_struct
        rel_perm_curve = new_value_struct[self.rel_perm_curve]['matrixrelperm'][-1]
        if self.case_loc is None or self.data_loc is None:
            warnings.warn('Case location and data location are required.')
            sys.exit(1)
        elif self.write_loc is None:
            warnings.warn("Entry value' struct written location is required.")
            sys.exit(1)
        else:
            assert type(self.write_loc) == list
            assert type(self.data_loc) == list
            for _ in range(len(self.write_loc)):
                rel_perm_curve[self.write_loc[_][0], self.write_loc[_][-1]] = doe_data[self.case_loc, self.data_loc[_]]
            new_value_struct[self.rel_perm_curve]['matrixrelperm'][-1] = rel_perm_curve
        self.entry.value_struct.change_value_struct(new_value_struct=new_value_struct)



class PDPConnector(Connector):
    def __init__(self, entry: Entry, rel_perm_curve: int):
        super(Connector, self).__init__()
        self.entry = entry
        self.rel_perm_curve = rel_perm_curve


    def write_doe_data(self, doe_data: np.ndarray, *args, **kwargs):
```

```
        # This function to write rel perm curve at the correct location in 'matrixcurvesets' entry
        self.verify_entry(expected_variable_name='matrixcurvesets')
        new_value_struct = self.entry.value_struct.value_struct
        pdp_curve = new_value_struct[self.rel_perm_curve][('pressuredependentpermeability',
'reversible')][0]
        if self.case_loc is None or self.data_loc is None:
            warnings.warn('Case location and data location are required.')
            sys.exit(1)
        elif self.write_loc is None:
            warnings.warn("Entry value' struct written location is required.")
            sys.exit(1)
        else:
            assert type(self.write_loc) == list
            assert len(self.write_loc) == 2
            assert type(self.data_loc) == list
            assert len(self.write_loc[0]) == len(self.write_loc[-1])
            assert len(self.write_loc[0]) == len(self.data_loc)
            for _ in range(len(self.write_loc[0])):
                pdp_curve[self.write_loc[0][_], self.write_loc[-1][_]] = \
                    doe_data[self.case_loc, self.data_loc[_]]
        new_value_struct[self.rel_perm_curve][('pressuredependentpermeability', 'reversible')][0] =
pdp_curve
        self.entry.value_struct.change_value_struct(new_value_struct=new_value_struct)


################################################################################
# Design of Experiments classes/functions using pyDOE, scipy and numpy
################################################################################
```

```
class DesignOfExperiments(object):
    def __init__(self, design: str):
        # TODO: ? Design of Experiments class that is integrated to the connector classes above
        super(DesignOfExperiments, self).__init__()
        self.data_locs = None
        self.data = None
        self.design = design

    def verify_design(self, design: str):
        try:
            assert self.design == design
        except AssertionError:
            warnings.warn('Incorrect design.')
            sys.exit(1)

    def set_data_locs(self, data_locs: Dict[Tuple[str, str],
                                            Union[int, List[int], List[List[int]]]]):
        """
        Set up the correlation between DoE data and the connector classes
        :param data_locs: correlate the data location in DoE data to the DoE parameter name & entry
        (eventually correlate to the written location in the entry's value struct)
        :return: self.data_locs (type dict)
        self.data_locs.keys(): type tuple (property name, entry name)
        self.data_locs.values(): int (list/1D array), list(int) (list of list/2D array), -1 (single value)
        """
        self.data_locs = data_locs
```

```
    def generate(self, num_cases: int, *args, **kwargs):
        pass


class CCC(DesignOfExperiments):
    def __init__(self, design='ccc'):
        super(DesignOfExperiments, self).__init__()
        self.design = design

    def generate(self, num_cases=100, *args, **kwargs):
        pass


class LHS(DesignOfExperiments):
    def __init__(self, design='lhs'):
        super(DesignOfExperiments, self).__init__()
        self.dist = None
        self.design = design

    def set_distributions(self, dist: list):
        self.dist = dist

    def generate(self, num_cases=100, *args, **kwargs):
        props = list(self.data_locs.keys())
        try:
            assert self.dist is not None
        except AssertionError:
            warnings.warn('Distributions are required.')
```

```
            sys.exit(1)
        lhs_data = pyDOE.lhs(len(props), samples=num_cases, criterion='center')
        for _ in range(len(props)):
            # Customize the distributions here via scipy
            lhs_data[:, _] = self.dist[_].ppf(lhs_data[:, _])
        self.data = lhs_data


################################################################################
# Interface classes to
# 1. Couple DesignOfExperiments class & Connector class
# 2. Record DesignOfExperiments in ResFrac files via comment lines
################################################################################


class DesignOfExperimentsAssembly(object):
    def __init__(self, base_entries: List[Entry]):
        super(DesignOfExperimentsAssembly, self).__init__()
        self.base_entries = base_entries
        self.doe_params = None  # DoE parameter names
        self.doe_entry_var_names = None  # DoE entry names
        self.doe_distributions = None  # DoE distributions
        self.write_locs = None  # Written locations inside the entries' value structs
        self.doe_data = None # DoE data
        #
        self.doe_batch = 0
        self.doe_dir = None
```

```
def reset_all_entries(self):
    # Reset all entries to non-doe entries
    for (_, entry) in enumerate(self.base_entries):
        if entry.is_doe is True:
            self.base_entries[_].is_doe = False
        else:
            pass

def set_main_dir(self, doe_dir):
    self.doe_dir = doe_dir

def set_doe_params(self, doe_params: List[str]):
    self.doe_params = doe_params

def set_doe_entry_var_names(self, doe_entry_var_names: List[str]):
    all_entry_var_names = [e.variable_name for e in self.base_entries]
    for (_, var_name) in enumerate(doe_entry_var_names):
        if var_name not in all_entry_var_names:
            warnings.warn('Incorrect entry variable name(s).')
            sys.exit(1)
        else:
            for e in self.base_entries:
                if e.variable_name == var_name:
                    e.is_doe = True
                else:
                    pass
    self.doe_entry_var_names = doe_entry_var_names
```

```
    def set_doe_distributions(self, doe_distributions: list):
        self.doe_distributions = doe_distributions


    def set_data_locs(self):
        data_locs = dict()
        for (_, param) in enumerate(self.doe_params):
            if param not in data_locs.keys():
                data_locs[(param, self.doe_entry_var_names[_])] = self.write_locs[_]
            else:
                pass
        return data_locs


    def set_write_locs(self, write_locs):
        self.write_locs = write_locs


    def verify_doe_interface(self):
        # Verify the number of doe parameters equal the number of entries (two parameters can
belong to one entry)
        try:
            assert self.doe_params is not None
            assert self.doe_entry_var_names is not None
            assert self.doe_distributions is not None
            assert self.write_locs is not None
        except AssertionError:
            warnings.warn('Design of Experiments interface is not defined.')
            sys.exit(1)
        try:
            assert len(self.doe_params) == len(self.doe_entry_var_names)
            assert len(self.doe_params) == len(self.write_locs)
```

```
        assert len(self.doe_params) == len(self.doe_distributions)
    except AssertionError:
        warnings.warn('Incorrect Design of Experiments interface.')
        sys.exit(1)

def generate_doe_object(self, design: str, num_cases: int):
    self.verify_doe_interface()
    data_locs = self.set_data_locs()
    doe_object = LHS(design=design)
    doe_object.data_locs = data_locs
    doe_object.verify_design(design=design)
    doe_object.set_distributions(self.doe_distributions)
    doe_object.generate(num_cases=num_cases)
    return doe_object

def write_doe_entries(self, doe_object: DesignOfExperiments, last_case: int):
    doe_data_ = doe_object.data
    # Loop through all doe cases to write settings/input files
    for case_loc in range(doe_data_.shape[0]):
        case_doe_entries = list()
        # Loop through all doe entries (using their variable names)
        for (_, (param_name, entry_var_name)) in enumerate(doe_object.data_locs):
            # Loop through all base entries to find the correct doe entries
            for entry in self.base_entries:
                if entry.variable_name == entry_var_name:
                    # The doe entry, init connector & modify using entry's variable name & write loc
                    entry_connector = init_connector(entry=entry, param_name=param_name)
                    if type(entry.value_struct) == SingleValueStruct:
```

```
                    # SingleValueConnector
                    entry_connector.data_type = float  # TODO: ? Fix me
                elif type(entry.value_struct) == RelPermStruct:
                    # RelPermConnector or PDPConnector
                    entry_connector.set_write_loc(write_loc=self.write_locs[_])
                    entry_connector.rel_perm_curve = 0
                elif type(entry.value_struct) == FaciesListStruct:
                    # LayerValueConnector
                    num_layers = len(entry.value_struct.value_struct.keys())
                    connected_layer_names = [i[0] for i in self.write_locs[_]]
                    connected_prop_names = [i[-1] for i in self.write_locs[_]]
                    entry_connector.layer_names = layer_names
                    entry_connector.prop_names = layer_props
                    entry_connector.set_number_of_layers(num_layers=num_layers)

entry_connector.set_connected_data(connected_layer_names=connected_layer_names,
                                       connected_prop_names=connected_prop_names)
                    entry_connector.set_write_loc()
                else:
                    # List/ListOfList/Array1D/Array2D-Connector
                    entry_connector.set_write_loc(write_loc=self.write_locs[_])
                entry_connector.set_doe_loc(case_loc=case_loc, data_loc=[_])
                entry_connector.write_doe_data(doe_data=doe_data_)
                case_doe_entries.append(entry_connector.entry)
            else:
                # The non-doe entry, remain
                case_doe_entries.append(entry)
        #
        doe_settings_file_name = 'doe_settings_case_' + str(case_loc) + '.txt'
```

```
            doe_settings_file_dir = os.path.join(self.doe_dir, doe_settings_file_name)
            simulation_file = simulation_files.SimulationFile(entries=case_doe_entries,
                                                              file_type='settings')
            simulation_file.write_file(file_name=doe_settings_file_dir)
        # Loop through all doe cases to write data for surrogate assembly
        doe_data_file_name = 'doe_data_' + str(self.doe_batch) + '.csv'
        doe_data_file_dir = os.path.join(self.doe_dir, doe_data_file_name)
        doe_data = list()
        for case_loc in range(doe_data_.shape[0]):
            case_data = dict()
            if 'case' not in case_data.keys():
                case_data['case'] = case_loc + last_case
            for _ in range(doe_data_.shape[-1]):
                if self.doe_params[_] not in case_data.keys():
                    case_data[self.doe_params[_]] = doe_data_[case_loc, _]
            doe_data.append(case_data)
        doe_data = pd.DataFrame(data=doe_data)
        self.doe_data = doe_data
        doe_data.to_csv(doe_data_file_dir, header=True)
        return doe_data
from src.base.base_libs import *


##########################################################################
# Design of Experiments helper variables
##########################################################################
```

```
layer_props = 'top bottom xperm yperm zperm curvesetname ' \
        'porositycompressibility referenceporosity ' \
        'stressdeviation dualporosity fractureporositycompressibility ' \
        'fracturereferenceporosity shapefactor matrixpermeability rockdensity ' \
        'rockheatcapacity thermalconductivity coefficientoflinearexpansion dphidT ' \
        'biotcoefficient Tstr horizontalfracturetoughness verticalfracturetoughness ' \
        'E0max sn90percentclosure Eresmax maximumflowingmolarmass ' \
        'optionalinitialwatersaturation langmuirpressure langmuirvolume' \
        'showinvisualizationtool proppant embedment'
layer_props = layer_props.split(sep=' ')
layer_names = ['Layer ' + str(_) for _ in range(60)]


################################################################################
# Design of Experiments helper functions
################################################################################


def modify_layer_names(original_layer_names: List[str], locs: List[int],
                       loc_layer_names: List[str]):
    mod_layer_names = deepcopy(original_layer_names)
    for (_, layer_name) in enumerate(layer_names):
        if _ in locs:
            mod_layer_names[_] = loc_layer_names[locs.index(_)]
    return mod_layer_names


def verify_single_value_type(data: Union[int, float, bool, str]):
```

```
    try:
        value = int(data)
        return int
    except AssertionError:
        try:
            value = float(data)
            return float
        except AssertionError:
            if data.lower() in ['true', 'false']:
                return bool
            else:
                return str


def verify_1d_list(data: list):
    assert type(data) == list
    assert len(data) > 0
    if type(data[0]) is not list:
        return True
    else:
        return False


def verify_1d_array(data: np.ndarray):
    assert type(data) == np.ndarray
    if type(data[0]) is not np.ndarray:
        return True
    else:
```

```
        return False


def layer_location(layer_name_: str, layer_names_: List[str]):
    try:
        assert layer_name_ in layer_names_
    except AssertionError:
        warnings.warn('Incorrect input layer name.')
        sys.exit(1)
    return layer_names_.index(layer_name_)


##############################################################################
# Design of Experiments helper variables (modified if necessary)
##############################################################################


layer_names = modify_layer_names(original_layer_names=layer_names,
     from base.base import *


experimental_doe_params = ['S_wr', 'S_or', 'S_gr', 'relative_frac_toughness']
experimental_response_params = ['BHP', 'Oil prod rate']
experimental_data_sources = ['Proxy_cases']

experimental_mode = True                    locs=[41], loc_layer_names=['Target depth'])
from base.base import *
```

```
#########################################################################
# Helper variables to support validating Res-Frac file structure
#########################################################################

res_frac_required_folders = ['Additional_Files', 'Input_Files', 'Settings_Files', 'Results']

res_frac_required_files = ['cpuinfo.txt', 'dmesg.txt', 'matrix_prop_names.txt',
                           'misc_visualization_data.txt', 'static_matrix.bin',
                           'static_wellfrac.bin', 'stderr.txt', 'stdout.txt', 'syslog.txt']

default_hyper_params = ['max_depth', 'gamma', 'reg_alpha', 'reg_lambda',
                        'colsample_bytree', 'min_child_weight',
                        'learning_rate', 'random_rate', 'max_bin']

gbm_hyper_params = ['max_depth', 'gamma', 'reg_alpha', 'reg_lambda',
                    'colsample_bytree', 'min_child_weight', 'n_estimators'
                    'learning_rate', 'random_rate', 'max_bin']

xgb_hyper_params = ['max_depth', 'gamma', 'reg_alpha', 'reg_lambda',
                    'colsample_bytree', 'min_child_weight',
                    'learning_rate', 'random_rate', 'max_bin']

int_hyper_params = ['max_depth', 'n_estimators', 'random_state', 'tree_method']

objective_hyper_params = ['objective']

quantile_hyper_params = ['quantile_alpha']
```

```
#############################################################################
# Helper functions for Dash callbacks and interfaces
#############################################################################


def convert_numpy_to_data_table(arr: np.ndarray, columns):
    data = pd.DataFrame(columns=columns, data=arr)
    data_table = dash_table.DataTable(
        data=data.to_dict('records'),
        columns=[{"name": i, "id": i} for i in data.columns])
    return data_table


def convert_dict_to_data_table(arr: dict, columns):
    data = list()
    for _ in arr.values():
        data.append(_)
    data = np.array(data)
    if columns is not None:
        data = pd.DataFrame(columns=columns, data=data)
    else:
        columns_ = ['Column ' + str(_) for _ in range(data.shape[-1])]
        data = pd.DataFrame(columns=columns_, data=data)
    data_table = dash_table.DataTable(
        data=data.to_dict('records'),
        columns=[{"name": i, "id": i} for i in data.columns])
    return data_table
```

```
def validate_res_frac_workflows_dir(workflows_dir, simulations_dir):
    """
    Validate the provided folder simulations_dir follows Res-Frac file structure
    :param workflows_dir:
    :param simulations_dir:
    :return:
    """
    simulation_runs_dir = os.path.join(workflows_dir, simulations_dir)
    if os.path.exists(simulation_runs_dir) is False:
        return False
    else:
        simulation_runs_data = os.listdir(simulation_runs_dir)
        if 'simulations' not in simulation_runs_data or \
                'metadata.json' not in simulation_runs_data:
            return False
        else:
            simulation_runs = os.path.join(simulation_runs_dir, 'simulations')
            simulation_run_folders = os.listdir(simulation_runs)
            num_simulation_runs = len(simulation_run_folders)
            simulation_run_folder_valid = 0
            #
            for simulation_run_folder in simulation_run_folders:
                simulation_run_dir = os.path.join(simulation_runs, simulation_run_folder)
                simulation_run_files = os.listdir(simulation_run_dir)
                if all(_ in simulation_run_files for _ in res_frac_required_files) is True \
                        and all(_ in simulation_run_files for _ in res_frac_required_folders) is True:
                    simulation_run_folder_valid += 1
```

```
        if simulation_run_folder_valid == num_simulation_runs:
            return True
        else:
            return False


def create_distribution_object(param_name: str, param_dist: str,
                               param_dist_params, object_wrapper='scipy'):
    if object_wrapper == 'scipy':
        if param_dist == 'uniform':
            dist_object = distributions.uniform(param_name, float(param_dist_params[0]),
                                                float(param_dist_params[-1]))
        elif param_dist == 'normal':
            dist_object = distributions.norm(param_name, float(param_dist_params[0]),
                                             float(param_dist_params[-1]))
        else:
            dist_object = distributions.lognorm(param_name, float(param_dist_params[0]),
                                                float(param_dist_params[-1]))
        return dist_object
    elif object_wrapper == 'hyperopt':
        if param_name in int_hyper_params:
            if param_dist == 'uniform':
                dist_object = scope.int(hp.uniform(param_name,
                                                   float(param_dist_params[0]),
                                                   float(param_dist_params[-1])))
            elif param_dist == 'normal':
                dist_object = scope.int(hp.normal(param_name,
                                                  float(param_dist_params[0]),
```

```
                                float(param_dist_params[-1])))
            else:
                dist_object = scope.int(hp.lognormal(param_name,
                                float(param_dist_params[0]),
                                float(param_dist_params[-1])))
        else:
            if param_dist == 'uniform':
                dist_object = hp.uniform(param_name,float(param_dist_params[0]),
                                         float(param_dist_params[-1]))
            elif param_dist == 'normal':
                dist_object = hp.normal(param_name, float(param_dist_params[0]),
                                        float(param_dist_params[-1]))
            else:
                dist_object = hp.lognormal(param_name, float(param_dist_params[0]),
                                           float(param_dist_params[-1]))
        return dist_object
    else:
        return None


def create_proxy_hyper_params_space(hyper_params_dists, proxy_type: str, exp_mode: str):
    hyper_params_space = {}
    hyper_param_names = hyper_params_dists['param_name']
    #
    if proxy_type == "Normal":
        pass
    else:
        hyper_params_space['objective'] = 'reg:quantileerror'
```

```
    hyper_params_space['quantile_alpha'] = np.array([0.05, 0.5, 0.95])
#
if exp_mode == "Fit":
    # Sample randomly a space
    for (_, name) in enumerate(hyper_param_names):
        dist = hyper_params_dists['param_dist'][_]
        dist_params = hyper_params_dists['param_dist_params'][_]
        dist_object = create_distribution_object(name, dist, dist_params)
        if name in int_hyper_params:
            val = np.round(dist_object.rvs(size=1)[0]).astype(int)
        else:
            val = dist_object.rvs(size=1)[0]
        if name not in hyper_params_space.keys():
            hyper_params_space[name] = val
    exp_mode_ = "fit"
    return hyper_params_space, exp_mode_
elif exp_mode == "Optimize":
    # Create a optimization space
    for (_, name) in enumerate(hyper_param_names):
        dist = hyper_params_dists['param_dist'][_]
        dist_params = hyper_params_dists['param_dist_params'][_]
        dist_object = create_distribution_object(name, dist, dist_params,
                                                 object_wrapper='hyperopt')
        if name not in hyper_params_space.keys():
            hyper_params_space[name] = dist_object
    exp_mode_ = "opt"
    return hyper_params_space, exp_mode_
else:
```

```
        return None, None
from DoE.doe.doe_v1 import *

import dash
import dash_bootstrap_components as dbc

from dash import dcc
from dash import html
from dash import Input, Output, State
from dash import callback, callback_context

from dash.exceptions import PreventUpdate
from dash import callback_context as ctx
from dash import register_page

from dash_bootstrap_components import Tab, Table, InputGroup, Col, Row
from dash_bootstrap_components import Modal, ModalTitle, ModalBody, ModalHeader, ModalFooter
from dash_bootstrap_components import Placeholder

import json, jsonschema
from proxy.proxy_data import *



class QuantileGBRegressor(object):
    def __init__(self, proxy_data: ProxyData):
        super(QuantileGBRegressor, self).__init__()
        self.proxy_data = proxy_data
        self.regressor_trait = GradientBoostingRegressor
```

```
    self.eval_metric = mean_pinball_loss
    self.eval_index = 1
    #
    self.response_param_index: int = 0
    self.test_size = 0.1
    #
    self.quantiles = [0.05, 0.5, 0.95]

def validate(self):
    assert self.proxy_data.proxy_data.empty is False

def set_evaluation_metric(self, eval_metric):
    self.eval_metric = eval_metric

def split(self, response_param_index: int, test_size: float):
    x_cols = list()
    for col in list(self.proxy_data.proxy_data.columns):
        if col not in ['case'] and col not in self.proxy_data.reservoir_response_params:
            x_cols.append(col)
    y_cols = self.proxy_data.reservoir_response_params
    x = self.proxy_data[x_cols]
    y = self.proxy_data[y_cols[response_param_index]]
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size,
                                                        shuffle=True, random_state=0)
    return x_train, x_test, y_train, y_test

def __fit__(self, params):
    x_train, x_test, y_train, y_test = self.split(self.response_param_index, self.test_size)
```

```
    quantile_models = {}
    eval_metrics = {}
    for quantile in self.quantiles:
        model = self.regressor_trait(loss='quantile', alpha=quantile,
                                     **params)
        if quantile not in quantile_models.keys():
            quantile_models[quantile] = model.fit(x_train, y_train)
            eval_metrics[quantile] = self.eval_metric(model.predict(x_test), y_test,
                                                      alpha=quantile)
    return quantile_models, eval_metrics


def __fit_cross_validation__(self, params, num_folds=5):
    x_cols = list()
    for col in list(self.proxy_data.proxy_data.columns):
        if col not in ['case'] and col not in self.proxy_data.reservoir_response_params:
            x_cols.append(col)
    y_cols = self.proxy_data.reservoir_response_params
    x = self.proxy_data.proxy_data[x_cols].to_numpy()
    y = self.proxy_data.proxy_data[y_cols[self.response_param_index]].to_numpy()
    #
    k_fold = KFold(n_splits=num_folds, shuffle=True, random_state=0)
    k_fold_metrics = []
    #
    model = self.regressor_trait(loss='quantile',
                                 alpha=self.quantiles[self.eval_index], **params)
    for train_index, test_index in k_fold.split(x):
        x_train, x_test = x[train_index], x[test_index]
        y_train, y_test = y[train_index], y[test_index]
```

```
            model.fit(x_train, y_train)
            y_pred = model.predict(x_test)
            k_fold_metric = self.eval_metric(y_test, y_pred,
                                             alpha=self.quantiles[self.eval_index])
            k_fold_metrics.append(k_fold_metric)
        return model, k_fold_metrics

    def __optimize__(self, opt_space: dict):
        model = self.regressor_trait(**opt_space, loss='quantile',
                                     alpha=self.quantiles[self.eval_index])
        x_train, x_test, y_train, y_test = self.split(self.response_param_index, self.test_size)
        #
        model.fit(x_train, y_train)
        y_pred = model.predict(x_test)
        test_metric = self.eval_metric(y_pred, y_test)
        return {'loss': test_metric, 'status': STATUS_OK, 'model': model}

    def __optimize_cross_validation__(self, opt_space: dict):
        model, k_fold_metrics = self.__fit_cross_validation__(opt_space)
        k_fold_metric = sum(k_fold_metrics) / len(k_fold_metrics)
        return {'loss': k_fold_metric, 'status': STATUS_OK, 'model': model}
from proxy.proxy_data import *



class NonQuantileRegressor(object):
    def __init__(self, proxy_data: ProxyData):
        super(NonQuantileRegressor, self).__init__()
        self.proxy_data = proxy_data
```

```
    self.regressor_trait = XGBRegressor
    self.eval_metric = mean_squared_error
    #
    self.response_param_index: int = 0
    self.test_size = 0.1

def validate(self):
    assert self.proxy_data.proxy_data.empty is False

def set_evaluation_metric(self, eval_metric):
    self.eval_metric = eval_metric

def split(self, response_param_index: int, test_size: float):
    x_cols = list()
    for col in list(self.proxy_data.proxy_data.columns):
        if col not in ['case'] and col not in self.proxy_data.reservoir_response_params:
            x_cols.append(col)
    y_cols = self.proxy_data.reservoir_response_params
    x = self.proxy_data[x_cols]
    y = self.proxy_data[y_cols[response_param_index]]
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=test_size,
                                            shuffle=True, random_state=0)
    return x_train, x_test, y_train, y_test

def __fit__(self, params):
    x_train, x_test, y_train, y_test = self.split(self.response_param_index, self.test_size)
    self.regressor_trait(**params).fit(x_train, y_train)
    return self.eval_metric(x_test, y_test)
```

```
def __fit_cross_validation__(self, params, num_folds=5):
    x_cols = list()
    for col in list(self.proxy_data.proxy_data.columns):
        if col not in ['case'] and col not in self.proxy_data.reservoir_response_params:
            x_cols.append(col)
    y_cols = self.proxy_data.reservoir_response_params
    x = self.proxy_data.proxy_data[x_cols].to_numpy()
    y = self.proxy_data.proxy_data[y_cols[self.response_param_index]].to_numpy()
    #
    k_fold = KFold(n_splits=num_folds, shuffle=True, random_state=0)
    k_fold_metrics = {}
    if self.regressor_trait == XGBRegressor:
        model = self.regressor_trait(**params, eval_metric=self.eval_metric)
    else:
        model = self.regressor_trait(**params)
    #
    _ = 0
    for train_index, test_index in k_fold.split(x):
        x_train, x_test = x[train_index], x[test_index]
        y_train, y_test = y[train_index], y[test_index]
        model.fit(x_train, y_train)
        y_pred = model.predict(x_test)
        score = self.eval_metric(y_test, y_pred)
        if 'fold_' + str(_) not in k_fold_metrics.keys():
            k_fold_metrics['fold_' + str(_)] = score
        _ += 1
    return model, k_fold_metrics
```

```
    def __optimize__(self, opt_space: dict):
        model = self.regressor_trait(**opt_space)
        model.eval_metric = self.eval_metric
        x_train, x_test, y_train, y_test = self.split(self.response_param_index, self.test_size)
        #
        evaluation = [(x_train, y_train), (x_test, y_test)]
        model.fit(x_train, y_train, eval_set=evaluation, verbose=False)
        y_pred = model.predict(x_test)
        test_metric = self.eval_metric(y_pred, y_test)
        return {'loss': test_metric, 'status': STATUS_OK, 'model': model}

    def __optimize_cross_validation__(self, opt_space: dict):
        model, cross_val_metrics = self.__fit_cross_validation__(opt_space)
        cross_val_metric = sum(cross_val_metrics) / len(cross_val_metrics)
        return {'loss': cross_val_metric, 'status': STATUS_OK, 'model': model}
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

import os
import shutil
from typing import List, Deque, Union, Dict, Any
from tqdm import tqdm

import sklearn
from sklearn.utils.validation import check_is_fitted
```

```
from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, mean_pinball_loss
from sklearn.ensemble import GradientBoostingRegressor, HistGradientBoostingRegressor

import xgboost as xgb
from xgboost import XGBRegressor, QuantileDMatrix, DeviceQuantileDMatrix

from hyperopt.pyll.base import scope
from hyperopt import fmin, tpe, hp, STATUS_OK, Trials, space_eval

import mlflow
import mlflow.sklearn
import mlflow.xgboost

from mlflow.models import infer_signature
from mlflow import log_metric, log_param, log_artifacts, log_input, log_text
from mlflow.tracking import MlflowClient

from mlflow import pyfunc
from mlflow.pyfunc import PythonModel, PythonModelContext, PyFuncModel, \
    PyFuncInput, PyFuncOutput
from mlflow.pyfunc import log_model, load_model
from proxy.proxy_base import *
from workflow.surrogate import *


class ProxyRawData(object):
```

```
    def __init__(self, project_dir: str, proxy_data_dir: str):
        super(ProxyRawData, self).__init__()
        self.project_dir = project_dir
        self.proxy_data_dir = proxy_data_dir
        self.proxy_data = pd.DataFrame()
        #
        self.num_doe_cases = None
        self.doe_assembler = None
        #
        self.fracture_profile_resolution = None
        self.doe_params = None
        self.reservoir_response_params = None
        #
        self.experimental_mode = True

    def init_data_properties(self, fracture_profile_resolution, doe_params,
                             reservoir_response_params):
        self.fracture_profile_resolution = fracture_profile_resolution
        self.doe_params = doe_params
        self.reservoir_response_params = reservoir_response_params

    def write_proxy_raw_data(self, proxy_raw_data_dir):
        proxy_data_manager = SurrogateDirectory(result_dir=self.project_dir)
        proxy_data_manager.experimental_doe_params = self.doe_params

proxy_data_manager.init_reservoir_response(reservoir_response_var_names=self.reservoir_resp
onse_params)

proxy_data_manager.init_fracture_profile(fracture_profile_resolution=self.fracture_profile_resol
ution)
```

```
        #
        if self.experimental_mode:
            proxy_data_manager.init_experimental_doe_data()
            proxy_raw_data =
proxy_data_manager.assemble_surrogate_directory(surrogate_dir=self.proxy_data_dir)
        else:
            assert self.doe_assembler is not None
            proxy_raw_data =
proxy_data_manager.assemble_surrogate_directory(surrogate_dir=self.proxy_data_dir)
        self.proxy_data = proxy_raw_data
        proxy_raw_data.to_csv(proxy_raw_data_dir)
        return proxy_raw_data



class ProxyData(object):
    def __init__(self, proxy_raw_data: pd.DataFrame, proxy_raw_data_dir: str):
        super(ProxyData).__init__()
        self.proxy_raw_data = proxy_raw_data
        self.proxy_raw_data_dir = proxy_raw_data_dir
        self.proxy_data: pd.DataFrame = pd.DataFrame()
        #
        self.doe_params = None
        self.reservoir_response_params = None
        self.fracture_profile_params = None
        self.proxy_time = ['surrogate_time']
        #
        self.experimental_mode = True
        self.experimental_Swr = [0.2, 0.01]
        self.experimental_Sor = [0.2, 0.01]
```

```
        self.experimental_Sgr = [0.03, 0.001]
        self.experimental_K = [0., 0.5]

    def validate(self):
        assert self.proxy_raw_data_dir is not None or self.proxy_raw_data.empty is True

    def init_data_properties(self, doe_params, reservoir_response_params):
        self.doe_params = doe_params
        self.reservoir_response_params = reservoir_response_params
        #
        if self.proxy_raw_data.empty is False:
            proxy_raw_data_ = self.proxy_raw_data.fillna(value=0., inplace=False)
        else:
            proxy_raw_data_ = pd.read_csv(self.proxy_raw_data_dir, index_col=0, header=0)
            proxy_raw_data_ = proxy_raw_data_.fillna(value=0., inplace=False)
        #
        fracture_profile_params_ = list()
        for col in list(proxy_raw_data_.columns):
            if 'z_' in 'col':
                fracture_profile_params_.append(col)
        #
        self.fracture_profile_params = fracture_profile_params_
        self.proxy_raw_data = proxy_raw_data_

    def scale_proxy_time(self):
        scaled_proxy_time_                                                    =
MinMaxScaler().fit_transform(self.proxy_raw_data[self.proxy_time])
        scaled_proxy_time = pd.DataFrame(data=scaled_proxy_time_, columns=self.proxy_time)
        return scaled_proxy_time
```

```
    def scale_fracture_profile(self):
        scaled_frac_profile_ = np.zeros([self.proxy_raw_data.shape[0],
                                         len(self.fracture_profile_params)])
        for (_, param) in enumerate(self.fracture_profile_params):
            scaled_fp = StandardScaler().fit_transform(self.proxy_raw_data[[param]])[:, 0]
            scaled_frac_profile_[:, _] = scaled_fp
        scaled_frac_profile = pd.DataFrame(data=scaled_frac_profile_,
columns=self.fracture_profile_params)
        return scaled_frac_profile


    def scale_reservoir_response_params(self):
        scaled_response_ = np.zeros([self.proxy_raw_data.shape[0],
                                     len(self.reservoir_response_params)])
        for (_, param) in enumerate(self.reservoir_response_params):
            scaled_rs = StandardScaler().fit_transform(self.proxy_raw_data[[param]])[:, 0]
            scaled_response_[:, _] = scaled_rs
        scaled_response = pd.DataFrame(data=scaled_response_,
columns=self.reservoir_response_params)
        return scaled_response


    def scale_doe_params(self):
        if self.experimental_mode:
            scaled_doe_ = np.zeros([self.proxy_raw_data.shape[0], len(self.doe_params)])
            scaled_doe_[:, 0] = (self.proxy_raw_data[['S_wr']].to_numpy()[:, 0] - 0.2) / 0.1
            scaled_doe_[:, 1] = (self.proxy_raw_data[['S_or']].to_numpy()[:, 0] - 0.2) / 0.1
            scaled_doe_[:, 2] = (self.proxy_raw_data[['S_gr']].to_numpy()[:, 0] - 0.03) / 0.001
            scaled_doe_[:, -1] = (self.proxy_raw_data[['relative_frac_toughness']].to_numpy()[:, 0] -
0.) / 0.5
```

```
            scaled_doe = pd.DataFrame(data=scaled_doe_, columns=self.doe_params)
            return scaled_doe
        else:
            return pd.DataFrame()

    def scale_proxy_raw_data(self, proxy_scaled_data_dir):
        scaled_time = self.scale_proxy_time()
        scaled_doe = self.scale_doe_params()
        scaled_frac_profile = self.scale_fracture_profile()
        scaled_response = self.scale_reservoir_response_params()
        proxy_scaled_data = pd.concat([scaled_time, scaled_doe,
                                       scaled_frac_profile, scaled_response], axis=1)
        proxy_scaled_data.insert(0, 'case', self.proxy_raw_data['case'].to_numpy(dtype=np.int8))
        self.proxy_data = proxy_scaled_data
        proxy_scaled_data.to_csv(proxy_scaled_data_dir)
        return proxy_scaled_data
from proxy.gb_proxy import *
from proxy.xgb_proxy import *

from proxy.proxy_opt import *
from proxy.proxy_utils import *


class MLFlowProxyWrapper(PythonModel):
    def __init__(self, proxy: QuantileXGBRegressor):
        super(MLFlowProxyWrapper, self).__init__()
        self.proxy = proxy
```

```
    def load_context(self, context):
        pass

    def predict(self, context: PythonModelContext,
                model_input: np.ndarray, params: Optional[dict[str, Any]]):
        return self.proxy.__predict__(model_input)


class ProxyExperiment(object):
    def __init__(self, mlflow_proxy_wrapper: MLFlowProxyWrapper,
                 mlflow_client=mlflow.MlflowClient()):
        super(ProxyExperiment, self).__init__()
        self.mlflow_client = mlflow_client
        self.mlflow_proxy_wrapper = mlflow_proxy_wrapper
        #
        self.experiment_dir = None
        self.experiment_name = None
        self.experiment_description = None
        #
        self.proxy_name = type(self.mlflow_proxy_wrapper.proxy).__name__
        self.experiment_mode = 'fit'
        self.proxy_registry_name = self.experiment_mode + '_' + self.proxy_name

    def set_expriment_mode(self, experiment_mode):
        self.experiment_mode = experiment_mode

    def set_experiment(self, experiment_dir, experiment_name, experiment_description):
        self.experiment_name = experiment_name
```

```
    self.experiment_dir = experiment_dir
    self.experiment_description = experiment_description
    if mlflow.get_experiment_by_name(name=experiment_name) is None:
        mlflow.create_experiment(name=experiment_name, tags=experiment_description)
    else:
        mlflow.set_experiment(experiment_name=experiment_name)

def log_new_fit_experiment(self, run_name, model_dir, artifact_dir, params):
    try:
        assert self.experiment_mode == 'fit'
        with mlflow.run(experiment_name=self.experiment_name, run_name=run_name):
            model, eval_metrics = self.mlflow_proxy_wrapper.proxy.__fit__(params)
            eval_metrics_ = read_xgb_eval_metrics(eval_metrics)
            mlflow.log_params(params=params)
            #
            mlflow.log_metrics(metrics=eval_metrics_)
            log_model(python_model=self.mlflow_proxy_wrapper,
                      artifact_path=model_dir,
                      registered_model_name=self.proxy_registry_name)
            artifact_file = os.path.join(artifact_dir, 'experiment_mode.txt')
            mlflow.log_text(text=self.experiment_mode, artifact_file=artifact_file)
            mlflow.end_run(status='FINISHED')
    except AssertionError:
        warnings.warn("Incorrect experiment mode. Fatal model error.")

def log_new_opt_experiment(self, run_name, model_dir, artifact_dir, opt_space):
    try:
        assert self.experiment_mode == 'opt'
```

```
        with mlflow.run(experiment_name=self.experiment_name, run_name=run_name):
            proxy_optimizer = ProxyOptimization(proxy_object=self.mlflow_proxy_wrapper.proxy)
            proxy_optimizer.init_optimizer(opt_space=opt_space)
            opt_instance = proxy_optimizer.exec_optimizer(opt_func_attr="__optimize_cross_validation__")
            opt_params = proxy_optimizer.eval_optimizer(opt_instance=opt_instance)
            #
            model, eval_metrics = self.mlflow_proxy_wrapper.proxy.__fit__(opt_params)
            eval_metrics_ = read_xgb_eval_metrics(eval_metrics)
            #
            mlflow.log_params(params=opt_params)
            mlflow.log_metrics(metrics=eval_metrics_)
            log_model(python_model=self.mlflow_proxy_wrapper,
                    artifact_path=model_dir,
                    registered_model_name=self.proxy_registry_name)
            artifact_file = os.path.join(artifact_dir, 'experiment_mode.txt')
            mlflow.log_text(text=self.experiment_mode, artifact_file=artifact_file)
            mlflow.end_run(status='FINISHED')
    except AssertionError:
        warnings.warn("Incorrect experiment mode. Fatal model error.")


def load_experiment(self, experiment_name: str):
    if experiment_name is not None:
        experiment_name_ = experiment_name
    else:
        experiment_name_ = self.experiment_name
    experiments = mlflow.search_experiments()
    experiment_dir = [exp.name for exp in experiments if
```

```
                experiment_name_ in exp.name][0]
        experiments = dict(mlflow.get_experiment_by_name(experiment_dir))
        experiment_id = experiments['experiment_id']
        experiment_runs = mlflow.search_runs(experiment_id)
        return experiment_runs


    def load_criteria_run(self, experiment_name: str, criteria: str):
        runs = self.load_experiment(experiment_name=experiment_name)
        criteria_cols = [col for col in runs.columns if col.startswith('params.') or
col.startswith('metrics.')]
        criteria_runs = runs[runs['status'] == 'FINISHED'][criteria_cols].dropna()
        runs_indices = criteria_runs.sort_values(f"metrics.{criteria}", ascending=False).index
        runs = runs.loc[runs_indices, :]
        best_run_id = runs['run_id'].tolist()[0]
        return best_run_id


    @staticmethod
    def load_criteria_model(run_id: str):
        return pyfunc.load_model(run_id)
from proxy.proxy_base import *



class ProxyOptimization(object):
    def __init__(self, proxy_object, trial=Trials()):
        super(ProxyOptimization, self).__init__()
        self.trials = trial
        #
        self.proxy_object = proxy_object
        self.opt_space = None
```

```
        self.opt_algo = tpe.suggest
        self.max_evals = 500

    def init_optimizer(self, opt_space):
        self.opt_space = opt_space

    def exec_optimizer(self, opt_func_attr):
        opt_func = getattr(self.proxy_object, opt_func_attr)
        opt_instance = fmin(fn=opt_func, space=self.opt_space,
                            algo=self.opt_algo, max_evals=self.max_evals,
                            trials=self.trials)
        return opt_instance

    def eval_optimizer(self, opt_instance):
        return space_eval(self.opt_space, opt_instance)
from proxy_base import *


mlflow.set_tracking_uri("sqlite:///mlruns.db")
mlflow.set_registry_uri("models")
from proxy.proxy_base import *


def read_xgb_eval_metrics(eval_metrics: dict):
    """
    Function to custom the mlflow log for eval_results in xgboost
    Structure of xgboost eval_results: dict("eval": dict(metric_name, metric_val))
    :param eval_metrics:
```

```
    :return:
    """
    eval_metrics_: dict = eval_metrics['eval']
    return eval_metrics_
from proxy.proxy_data import *


class XGBDataset(object):
    def __init__(self, df):
        super(XGBDataset, self).__init__()
        self.df = df
        self.time_param = ['surrogate_time']
        self.doe_params = [None]
        self.fracture_profile_params = [None]
        self.response_params = [None]
        #
        self.x_cols = [None]
        self.y_cols = [None]

    def set_up_params(self, doe_params_, fracture_profile_params_, response_params_):
        self.doe_params = doe_params_
        self.fracture_profile_params = fracture_profile_params_
        self.response_params = response_params_

    def set_up_columns(self, mode: str, response_index: int):
        if mode == 'default':
            self.x_cols = self.time_param + self.doe_params + self.fracture_profile_params
        elif mode == 'exclude_doe':
```

```
            self.x_cols = self.time_param + self.fracture_profile_params
        elif mode == 'exclude_time_doe':
            self.x_cols = self.fracture_profile_params
        self.y_cols = [self.response_params[response_index]]


    def split(self):
        x = self.df[self.x_cols]
        y = self.df[self.y_cols]
        x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1, random_state=0)
        x_train, x_test, y_train, y_test = x_train.to_numpy(), x_test.to_numpy(), y_train.to_numpy(),
y_test.to_numpy()
        xy_train = QuantileDMatrix(x_train, x_train)
        xy_test = QuantileDMatrix(x_test, y_test, ref=xy_train)
        return {'train_set': [xy_train, x_train, y_train], 'test_set': [xy_test, x_test, y_test]}


    def split_cross_validation(self, num_folds: int = 5):
        x = self.df[self.x_cols].to_numpy()
        y = self.df[self.y_cols].to_numpy()
        k_fold = KFold(n_splits=num_folds, shuffle=True, random_state=0)
        train_test = dict()
        i = 0
        for train_index, test_index in k_fold.split(x):
            x_train, x_test = x[train_index], x[test_index]
            y_train, y_test = y[train_index], y[test_index]
            key_train, key_test = 'train_set_' + str(i), 'test_set_' + str(i)
            xy_train = QuantileDMatrix(x_train, y_train)
            xy_test = QuantileDMatrix(x_test, y_test, ref=xy_train)
            if key_train not in train_test.keys():
                train_test[key_train] = [xy_train, x_train, y_train]
```

```
            if key_test not in train_test.keys():
                train_test[key_test] = [xy_test, x_test, y_test]
            i += 1
        return train_test


class QuantileXGBRegressor(object):
    def __init__(self, xgb_data: XGBDataset):
        super(QuantileXGBRegressor, self).__init__()
        self.xgb_data = xgb_data
        self.num_folds = 5
        #
        self.eval_metric = mean_squared_error
        self.eval_index: int = 1
        self.quantiles = np.array([0.05, 0.5, 0.95], dtype=np.float32)
        #
        self.model: xgb.Booster = None

    def __fit__(self, params: dict):
        train_test = self.xgb_data.split()
        eval_metrics: Dict[str, Dict] = {}
        params_ = params
        if 'quantile_alpha' not in params_.keys():
            params_['quantile_alpha'] = self.quantiles
        else:
            pass
        params_['objective'] = "reg:quantileerror"
        #
```

```
    model = xgb.train(params_, train_test['train_set'][0],
                      num_boost_round=32,
                      evals=[(train_test['train_set'][0], "train"),
                             (train_test['test_set'][0], "test")],
                      evals_result=eval_metrics, verbose_eval=False)
    self.model = model
    return model, eval_metrics


def __fit_cross_validation__(self, params: dict):
    train_test = self.xgb_data.split_cross_validation(num_folds=self.num_folds)
    model = None
    mean_metrics = {}
    params_ = params
    if 'quantile_alpha' not in params_.keys():
        params_['quantile_alpha'] = self.quantiles
    else:
        pass
    params_['objective'] = "reg:quantileerror"
    #
    for _ in range(self.num_folds):
        eval_metrics: Dict[str, Dict] = {}
        xy_train, x_train, y_train = train_test['train_set_' + str(_)]
        xy_test, x_test, y_test = train_test['test_set_' + str(_)]
        model = xgb.train(params_, xy_train,
                          num_boost_round=32,
                          evals=[(xy_train, "train"),
                                 (xy_test, "test")],
                          evals_result=eval_metrics, verbose_eval=False)
```

```
            y_pred = model.inplace_predict(x_test)
            mean_metric = self.eval_metric(y_pred[:, self.eval_index], y_test)
            if 'fold_' + str(_) not in mean_metrics.keys():
                mean_metrics['fold_' + str(_)] = mean_metric
        self.model = model
        return model, mean_metrics

    def __predict__(self, x: np.ndarray):
        try:
            assert self.model is not None
            return self.model.inplace_predict(x)
        except AssertionError:
            warnings.warn('Model instance is not set, may cause fatal prediction.')
            return None

    def __optimize_cross_validation__(self, opt_space: dict):
        model, mean_metrics = self.__fit_cross_validation__(opt_space)
        mean_metric = sum(mean_metrics.values()) / len(mean_metrics.keys())
        return {'loss': mean_metric, 'status': STATUS_OK, 'model': model}
from proxy_explainer_base import *

from proxy.proxy_data import *
from proxy.proxy import *
from proxy.gb_proxy import *
from proxy.xgb_proxy import *


class ProxyExplainer(object):
```

```
def __init__(self, model, new_proxy_data: ProxyData):
    super(ProxyExplainer, self).__init__()
    self.model = model
    self.new_proxy_data = new_proxy_data
    #
    self.mode = 'tree'
    self.x_cols = None

def validate(self):
    if check_is_fitted(self.model) is True and \
            self.new_proxy_data.proxy_data.empty is False:
        return True
    else:
        return False

def init_explainer(self, mode, x_cols, sample=100):
    sampled_x = self.new_proxy_data.proxy_data[x_cols].sample(sample, random_state=0)
    if mode == 'default':
        return Explainer(self.model, sampled_x), sampled_x
    else:
        return TreeExplainer(self.model, sampled_x), sampled_x

def plot_bee_swarm(self):
    fig = plt.figure()
    explainer, sampled_x = self.init_explainer(self.mode, self.x_cols)
    shap_values = explainer.shap_values(sampled_x)
    shap.summary_plot(shap_values, sampled_x)
    return fig
```

```
    def plot_bar(self):
        fig = plt.figure()
        explainer, sampled_x = self.init_explainer(self.mode, self.x_cols)
        shap_values = explainer.shap_values(sampled_x)
        shap.summary_plot(shap_values, sampled_x, plot_size='bar')
        return fig

    def plot_water_fall(self, instance_index: int, max_display=20):
        fig = plt.figure()
        explainer, sampled_x = self.init_explainer(self.mode, self.x_cols)
        shap_values = explainer.shap_values(sampled_x)
        shap.waterfall_plot(shap_values[instance_index], max_display=max_display)
        return fig

    def plot_scatter(self, instance: str):
        fig = plt.figure()
        explainer, sampled_x = self.init_explainer(self.mode, self.x_cols)
        shap_values = explainer.shap_values(sampled_x)
        shap.plots.scatter(shap_values[:, instance])
        return fig

    @staticmethod
    def display_to_gui(fig):
        return plotly_tools.mpl_to_plotly(fig)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
import pandas as pd

import os
import shutil
from typing import List, Deque, Union, Dict
from tqdm import tqdm

import shap
from shap import Explainer, TreeExplainer, DeepExplainer

import plotly.tools as plotly_tools
from base.base import *



# the style arguments for the sidebar. We use position:fixed and a fixed width
SIDEBAR_STYLE = {
    "position": "fixed",
    "top": 0,
    "left": 0,
    "bottom": 0,
    "width": "16rem",
    "padding": "2rem 1rem",
    "background-color": "#f8f9fa",
}



doe_sidebar = [
    html.Div(
```

```
        dbc.Row(
            [
                dbc.Col("Design of Experiments"),
                dbc.Col(
                    html.I(className="fas fa-chevron-right me-3"),
                    width="auto",
                ),
            ],
            className="doe_sidebar_row",
        ),
        style={"cursor": "pointer"},
        id="doe_sidebar",
    ),
    dbc.Collapse(
        [
            dbc.NavLink("ResFrac", href="/res_frac"),
        ],
        id="doe_sidebar_collapse",
    ),
]

frac_cal_sidebar = [
    html.Div(
        dbc.Row(
            [
                dbc.Col("Fracture Calibration"),
                dbc.Col(
                    html.I(className="fas fa-chevron-right me-3"),
```

```
                width="auto",
            ),
        ],
        className="frac_cal_sidebar_row",
    ),
    style={"cursor": "pointer"},
    id="frac_cal_sidebar",
),
dbc.Collapse(
    [
        dbc.NavLink("Fracture Calibration Proxy", href="/frac_cal_proxy"),
    ],
    id="frac_cal_sidebar_collapse",
),
]

proxy_deploy_sidebar = [
    html.Div(
        dbc.Row(
            [
                dbc.Col("Deploy the proxy"),
                dbc.Col(
                    html.I(className="fas fa-chevron-right me-3"),
                    width="auto",
                ),
            ],
            className="hist_match_sidebar_row",
        ),
```

```
        style={"cursor": "pointer"},
        id="proxy_deploy_sidebar",
    ),
    dbc.Collapse(
        [
            dbc.NavLink("Receive sensor & deploy", href="/deploy_proxy"),
        ],
        id="proxy_deploy_sidebar_collapse",
    ),
]



sidebar = html.Div(
    [
        html.H1("i-Geo Sensing", style={'textAlign': 'center'}),
        html.Hr(),
        html.P(
            "A sensor processing platform for fracture calibration", className="lead",
            style={'textAlign': 'center'}),
        dbc.Nav(doe_sidebar + frac_cal_sidebar + proxy_deploy_sidebar,
                vertical=True),
    ],
    style=SIDEBAR_STYLE,
    id="sidebar",
)
from simulator.simulation.simulation_helpers import *
from simulator.base.utils import *
```

```
######################################################################
# Base Entry & ValueStructs in ResFrac
######################################################################


class ValueStruct(object):
    def __init__(self, value_struct: None):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct

    def change_value_struct(self, new_value_struct):
        if type(new_value_struct) != type(self.value_struct):
            warnings.warn('Change the value structure.')
            self.value_struct = new_value_struct
        else:
            self.value_struct = new_value_struct

    def write_value_struct(self, file):
        """
        TODO: ? Write the value structure into the opened file
        Support np.array (most ResFrac values) and dict (Rel Perm
        & Time PErm dependency values)
        :param file:
        :return:
        """
        pass
```

```
class SingleValueStruct(ValueStruct):
    def __init__(self, value_struct):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct

    def write_value_struct(self, file):
        if type(self.value_struct) != list:
            file.write(str(self.value_struct))
            file.write('\n')
        else:
            write_list_values(file, self.value_struct)


class MatrixValueStruct(ValueStruct):
    def __init__(self, value_struct, length):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
        self.length = length

    def write_value_struct(self, file):
        if type(self.value_struct) == np.ndarray:
            write_matrix_values(file=file, array=self.value_struct)
        else:
            write_list_of_list_values(file, values=self.value_struct)


class Entry(object):
```

```
def __init__(self, variable_name: str, length: int,
             value_struct: ValueStruct, is_doe: False):
    super(Entry, self).__init__()
    self.comments = None
    self.variable_name = variable_name
    self.length = length
    self.value_struct = value_struct
    self.is_doe = is_doe

def add_comments(self, comments):
    self.comments = comments

def change_variable_name(self, new_variable_name):
    if self.is_doe is True:
        self.variable_name = new_variable_name
    else:
        pass

def change_length(self, new_length):
    if self.is_doe is True:
        self.length = new_length
    else:
        pass

def change_values(self, new_value_struct):
    if self.is_doe is True:
        self.value_struct.change_value_struct(new_value_struct)
    else:
```

```
            warnings.warn('Not a DoE variable. Value structure is not altered.')
            pass

    def write_entry_values(self, file):
        """
        TODO: ? Implement method to write entry's data
        - Method is override by specific value structs (e.g., rel_perm)
        - Method is supplemented by another method to change specific data in the
        value struct (e.g, change rel perm kr in rel_perm)
        :return:
        """
        if self.is_doe is True:
            self.value_struct.write_value_struct(file=file)
        else:
            warnings.warn('Not a DoE variable. Value structure shall not be altered.')
            self.value_struct.write_value_struct(file=file)


######################################################################
# ValueStruct & Entry belong to settings in ResFrac
######################################################################


class RelPermStruct(ValueStruct):
    def __init__(self, value_struct: List[dict]):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
        self.rel_perm_scope = '***'
```

```
def write_value_struct(self, file):
    """
    TODO: ? Fix this function to correlate with the parsing & regex templates
    :param file:
    :return:
    """
    for (_, rel_perm_curve) in enumerate(self.value_struct):
        for rel_perm_key, rel_perm_value in rel_perm_curve.items():
            if rel_perm_key == 'curvesetname':
                file.write(rel_perm_key)
                file.write('\n')
                file.write(rel_perm_value)
                file.write('\n')
            elif rel_perm_key == 'matrixrelperm':
                file.write(rel_perm_key)
                file.write('\n')
                file.write(rel_perm_value[0])
                file.write('\n')
                write_matrix_values(file=file, array=rel_perm_value[-1])
                file.write('\n')
            elif rel_perm_key == ('pressuredependentpermeability', 'reversible'):
                file.write(rel_perm_key[0])
                file.write('\n')
                file.write(rel_perm_key[-1])
                file.write('\n')
                write_matrix_values(file=file, array=rel_perm_value[0])
                file.write('\n')
```

```
                for _ in rel_perm_value[-1]:
                    file.write(_)  # _ is '---'
                    file.write('\n')
            elif rel_perm_key == ('pressuredependentpermeability', 'irreversible'):
                file.write(rel_perm_key[0])
                file.write('\n')
                file.write(rel_perm_key[-1])
                file.write('\n')
                for _ in rel_perm_value:
                    file.write(str(_))  # _ is '---'
                    file.write('\n')
            else:
                file.write(rel_perm_key)
                file.write('\n')
                for _ in rel_perm_value:
                    file.write(str(_))  # _ is '---'
                    file.write('\n')
        file.write(self.rel_perm_scope)
        file.write('\n')


class FaciesListStruct(ValueStruct):
    def __init__(self, value_struct: dict):
        super(ValueStruct).__init__()
        self.value_struct = value_struct

    def write_value_struct(self, file):
        """
```

```
        TODO: Write the facies list data into an opened file per line as follows:
        Layer + layer_number + row data of value_struct
        :param file:
        :return:
        """
        for (layer_name, layer_data) in self.value_struct.items():
            file.write(layer_name)
            file.write('\t')
            for data in layer_data:
                file.write(str(data))
                file.write('\t')
            file.write('\n')
        file.write('\n')


class BlackOilModelStruct(ValueStruct):
    def __init__(self, value_struct: tuple):
        super(ValueStruct).__init__()
        self.value_struct = value_struct
        self.size = 4

    def write_value_struct(self, file):
        """
        TODO: Write the black oil data into an opened file per line as follows:
        - Initial bubble/dew point, co, cw, muw, Bwatbubblepoint, oilspecificgravity, gasspecificgravity, waterspecificgravity,
        'unsaturated properties' method
        - Number of rows in black oil property table
        - Black oil property table
```

```
        (There is no keys in the value_struct)
        :param file:
        :return:
        """
        write_list_values(file, self.value_struct[0])
        file.write(str(self.value_struct[1]))  # Number of rows in black oil property table
        file.write('\n')
        write_matrix_values(file=file, array=self.value_struct[-1])
        file.write('\n')


class ClusterPerStageStruct(ValueStruct):
    def __init__(self, value_struct: dict):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
        self.scope_str = '***'

    def write_value_struct(self, file):
        """
        TODO: ? Write prop types into an opened file per line as follows:
        :param file:
        :return:
        """
        for (_, cluster_per_stage) in self.value_struct.items():
            file.write(self.scope_str)
            file.write('\n')
            for data in cluster_per_stage:
                file.write(str(data))
```

```
            file.write('\n')
        file.write('\n')


class WaterByLayerStruct(ValueStruct):
    def __init__(self, value_struct: dict):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
        self.scope_str = '***'

    def write_value_struct(self, file):
        """
        TODO: ? Write prop types into an opened file per line as follows:
        :param file:
        :return:
        """
        for (_, init_Sw) in self.value_struct.items():
            file.write(self.scope_str)
            file.write('\n')
            file.write(str(init_Sw))
            file.write('\n')
        file.write('\n')


class PropTypesStruct(ValueStruct):
    def __init__(self, value_struct: dict):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
```

```
    def write_value_struct(self, file):
        """
        TODO: ? Write prop types into an opened file per line as follows:
        :param file:
        :return:
        """
        for (prop_name, prop_data) in self.value_struct.items():
            file.write(prop_name)
            file.write('\t')
            for data in prop_data:
                file.write(str(data))
                file.write('\t')
            file.write('\n')
        file.write('\n')


class PropMixtureStruct(ValueStruct):
    def __init__(self, value_struct: dict):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct

    def write_value_struct(self, file):
        """
        TODO: ? Write prop mixtures into an opened file per line as follows:
        :param file:
        :return:
        """
```

```
        for (mix_name, mix_data) in self.value_struct.items():
            file.write(mix_name)
            file.write('\t')
            for data in mix_data:
                file.write(str(data))
                file.write('\t')
            file.write('\n')
        file.write('\n')


class PropPermModelStruct(ValueStruct):
    def __init__(self, value_struct: np.ndarray):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct

    def write_value_struct(self, file):
        """
        TODO: ? Write prop mixtures into an opened file per line as follows:
        :param file:
        :return:
        """
        write_matrix_values(file=file, array=self.value_struct)


class FracPermModelStruct(ValueStruct):
    def __init__(self, value_struct: np.ndarray):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
```

```
    def write_value_struct(self, file):
        """
        TODO: ? Write prop mixtures into an opened file per line as follows:
        :param file:
        :return:
        """
        write_matrix_values(file=file, array=self.value_struct)


class WaterSoluteStruct(ValueStruct):
    def __init__(self, value_struct: list):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct

    def write_value_struct(self, file):
        for water_solute_value in self.value_struct:
            file.write(str(water_solute_value))
            file.write('\t')
        file.write('\n')


class FluidMixtureStruct(ValueStruct):
    def __init__(self, value_struct: dict):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct

    def write_value_struct(self, file):
```

```
        for (mix_name, mix_data) in self.value_struct.items():
            file.write(mix_name)
            file.write('\t')
            for data in mix_data:
                file.write(str(data))
                file.write('\t')
            file.write('\n')
        file.write('\n')



class WellSequenceStruct(ValueStruct):
    def __init__(self, value_struct: list, sequence_type: str):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
        self.sequence_type = sequence_type
        # Data within 2 sequence_scope_str is written for a sequence of a well (example: injection/production)
        self.sequence_scope_str = '***'


    def write_value_struct(self, file):
        if self.value_struct is None:
            pass
        else:
            # Write sequence_scope_str to start a sequence
            file.write(self.sequence_scope_str)
            file.write('\n')
            if self.sequence_type == 'Shut-in':
                file.write(self.value_struct[0])
                file.write('\n')
```

```
            write_list_values(file, self.value_struct[1:])
        elif self.sequence_type == 'Injection':
            for _ in self.value_struct[: -1]:
                file.write(str(_))
                file.write('\n')
            write_list_of_list_values(file, self.value_struct[-1])
        elif self.sequence_type == 'Production':
            file.write(self.value_struct[0])
            file.write('\n')
            file.write(self.value_struct[1])
            file.write('\n')
            write_list_values(file, self.value_struct[2: -1])
            file.write('\n')
            write_list_of_list_values(file, self.value_struct[-1])
        else:
            warnings.warn('Can not detect the sequence.')
            sys.exit(1)
        file.write('\n')



class WellBoundaryConditionStruct(ValueStruct):
    def __init__(self, value_struct: dict):
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
        # Data within 2 well_scope_str is written for a well
        self.well_scope_str = '********'

    def write_value_struct(self, file):
```

```
        """
        TODO: Write the boundary condition control (for well(s)) into an opened file as follows:
        -
        :param file:
        :return:
        """
        for (well_name, well_sequences) in self.value_struct.items():
            # Write well_scope_str and well_name to start a well
            file.write(self.well_scope_str)
            file.write('\n')
            file.write(well_name)
            file.write('\n')
            # Write all sequences for a well
            shut_in_sequence = well_sequences['Shut-in']
            injection_sequence = well_sequences['Injection']
            production_sequence = well_sequences['Production']
            WellSequenceStruct(value_struct=shut_in_sequence, sequence_type='Shut-
in').write_value_struct(file)
            WellSequenceStruct(value_struct=injection_sequence,
sequence_type='Injection').write_value_struct(file)
            WellSequenceStruct(value_struct=production_sequence,
sequence_type='Production').write_value_struct(file)
        # Write well_scope_str to end
        file.write(self.well_scope_str)
        file.write('\n')


class WellTruncateSequenceStruct(ValueStruct):
    def __init__(self, value_struct: dict):
```

```
        super(ValueStruct, self).__init__()
        self.value_struct = value_struct
        self.begin_str = '***'

    def write_value_struct(self, file):
        for (_, truncate_data) in self.value_struct.items():
            file.write(self.begin_str)
            file.write('\n')
            for data in truncate_data:
                file.write(str(data))
                file.write('\n')
        file.write('\n')


######################################################################
# ValueStruct & Entry belong to input in ResFrac
######################################################################


class WellVerticesStruct(ValueStruct):
    def __init__(self, value_struct: dict):
        super(ValueStruct).__init__()
        self.value_struct = value_struct
        self.begin_str = '***'

    def write_value_struct(self, file):
        for (_, well_vertices) in self.value_struct.items():
            file.write(self.begin_str)
```

```
            file.write('\n')
            for data in well_vertices:
                write_list_values(file=file, values=data)
        file.write('\n')


######################################################################
# Irregular ValueStruct list
######################################################################


irregular_value_structs = [RelPermStruct, FaciesListStruct, BlackOilModelStruct,
                           ClusterPerStageStruct, WaterByLayerStruct,
                           PropTypesStruct, PropMixtureStruct,
                           PropPermModelStruct, FracPermModelStruct,
                           WaterSoluteStruct, FluidMixtureStruct,
                           WellBoundaryConditionStruct,                WellTruncateSequenceStruct,
WellVerticesStruct]
from simulator.simulation.simulation_helpers import *
from simulator.base.entry import *
from simulator.base.regex_templates import *
from simulator.base.utils import *


######################################################################
# Method to parse entry data from ResFrac files
######################################################################
```

```
def parse_entry(entry: dict):
    """
    TODO: ? Parse the entry's value(s) using ResFrac embedded format to obtain:
    1.
    :param entry:
    :return:
    """
    # Create entry_ (being parsed from entry)
    value_struct_ = ValueStruct(value_struct=None)
    entry_ = Entry(variable_name=' ', length=0, value_struct=value_struct_, is_doe=True)
    # Extract the entry's variable name
    variable_name_ = entry['Variable name:'][0]
    variable_name_ = variable_name_.split(sep='\n')[0]
    # Extract the entry's length
    length_ = entry['Length:'][0]
    length_ = int(length_.split(sep='\n')[0])
    # Extract the entry's value struct
    raw_value_struct_ = entry['Value(s):']
    if length_ == 0:
        # Entry has no value
        entry_.change_values(None)
    elif length_ == 1 and len(raw_value_struct_) == 1:
        # Entry has one single value
        parsed_value = parse_value(raw_value_struct_=raw_value_struct_)
        entry_.value_struct = SingleValueStruct(value_struct=parsed_value)
    elif variable_name_ in irregular_variable_names:
        # Entry has irregular value
        variable_name_idx_ = irregular_variable_names.index(variable_name_)
```

```
        variable_regex_                                                                    =
irregular_regex_classes[variable_name_idx_](raw_str=raw_value_struct_)
        parsed_value = variable_regex_.extract()
        entry_.value_struct = irregular_value_structs[variable_name_idx_](value_struct=None)
        entry_.change_values(parsed_value)
    else:
        parsed_value = ListOfListRegex(raw_str=raw_value_struct_).extract()
        entry_.value_struct = MatrixValueStruct(value_struct=None, length=length_)
        entry_.change_values(parsed_value)
    # Complete the parsing
    entry_.change_variable_name(variable_name_)
    entry_.change_length(length_)
    return entry_



def search_entry(entries: List[Entry], entry_name: str):
    for (ie, entry) in enumerate(entries):
        if entry_name in entry.variable_name:
            return entry
    return None



def parse_value(raw_value_struct_):
    # TODO: ? Test robustness of this function for common value_struct
    # Common value struct has only 1 value (float, str, int, bool), no repeat.
    parsed_data_ = raw_value_struct_[0]
    if '\t' not in parsed_data_:
        parsed_data_ = remove_new_line_char(parsed_data_)
        return extract_primitive_data_type(parsed_data_)
```

```
    else:
        parsed_data_ = remove_new_line_char(parsed_data_)
        parsed_data_ = parsed_data_.split(sep='\t')
        return extract_list_data_type(parsed_data_)
import numpy as np

from src.base.base_libs import *
from simulator.base.utils import remove_new_line_char, remove_empty_in_block

##########################################################################
# Regular expression helpers to match data in ResFrac files
##########################################################################

# Regex existed in ResFrac
blank_regex = '[BLANK]'
default_regex = 'Default'

# Pre-defined regexes to parse specific data (e.g, relative perm, well control)
nan_regex = 'nan'
sequence_scope_regex = '***'
well_scope_regex = '********'
split_regex = '---'
repetition_regex = 'REPEAT'
end_regex = 'END'

# Pre-defined data types to parse specific data (i.e., relative perm, well control)
repeated_int = {repetition_regex: int}
repeated_float = {repetition_regex: float}
```

```
repeated_str = {repetition_regex: str}
repeated_split = {repetition_regex: split_regex}

repeated_list = {repetition_regex: list}
repeated_list_int = {repetition_regex: (list, int)}
repeated_list_float = {repetition_regex: (list, float)}


# TODO: ? Use pair of keywords as tuple() if 2 keywords are required
brooks_corey_regex = ['curvesetname', 'matrixrelperm',
                      ('pressuredependentpermeability', 'reversible'),
                      ('pressuredependentpermeability', 'irreversible'),
                      'tenxreversiblepermeabilitylossperpressureincrement',
                      'tenxirreversiblepermeabilitylossperpressureincrement',
                      'lowerpressurethresholdforreversiblepermeabilityincrease',
                      'upperpressurethresholdforreversiblepermeabilityincrease',
                      'permmultiplierforreversiblepermeabilityincrease',
                      'lowerpressurethresholdforirreversiblepermeabilityincrease',
                      'upperpressurethresholdforirreversiblepermeabilityincrease',
                      'permmultiplierforirreversiblepermeabilityincrease',
                      'permeabilitymultiplier',
                      'waterbankthicknesstorelpermincreasescalingthickness',
                      'waterbankthicknesstorelpermdecreasescalingthickness',
                      'waterbankimmobilefraction', end_regex]

x_curve_regex = ['curvesetname', 'matrixrelperm',
                 ('pressuredependentpermeability', 'reversible'),
                 ('pressuredependentpermeability', 'irreversible'),
```

```
            'tenxreversiblepermeabilitylossperpressureincrement',
            'tenxirreversiblepermeabilitylossperpressureincrement',
            'lowerpressurethresholdforreversiblepermeabilityincrease',
            'upperpressurethresholdforreversiblepermeabilityincrease',
            'permmultiplierforreversiblepermeabilityincrease',
            'lowerpressurethresholdforirreversiblepermeabilityincrease',
            'upperpressurethresholdforirreversiblepermeabilityincrease',
            'permmultiplierforirreversiblepermeabilityincrease',
            'permeabilitymultiplier',
            'waterbankthicknesstorelpermincreasescalingthickness',
            'waterbankthicknesstorelpermdecreasescalingthickness',
            'waterbankimmobilefraction', end_regex]

brooks_corey_data = [str, (str, repeated_list_float),
                     (repeated_list_float, repeated_split),
                     repeated_str,
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str),
                     (float, str)]
```

```
x_curve_data = [str, (str, repeated_list_float),
                repeated_split,
                repeated_split,
                (float, str),
                (float, str),
                (float, str),
                (float, str),
                (float, str),
                (float, str),
                (float, str),
                (float, str),
                (float, str),
                (float, str),
                (float, str),
                (float, str)]

injection_sequence_data_regex = [str, str, str, str, float, float, repeated_list]
production_sequence_data_regex = [str, str, list, repeated_list]
shut_in_sequence_data_regex = [str, list]


class RepeatedDataType(object):
    def __init__(self, data_type: dict):
        super(RepeatedDataType, self).__init__()
        try:
            assert type(data_type) == dict
            assert list(data_type.keys())[0] == repetition_regex
```

```
        self.data_type = data_type
    except AssertionError:
        warnings.warn('Incorrect repeated data type.')
        self.data_type = None


def value_error(self, raw_data: str):
    data_type_ = self.data_type[repetition_regex]
    raw_data_ = remove_new_line_char(raw_data)
    if type(data_type_) != tuple:
        # Repeated int, float, str, list (1 single value or 1 list per line)
        if '\t' not in raw_data_:
            if data_type_ in [int, float]:
                try:
                    value_ = data_type_(raw_data_)
                    return False
                except ValueError:
                    return True
            elif data_type_ in [split_regex]:
                return False
            else:
                return False
        else:
            return False
    else:
        # Repeated list of all ints/floats (1 list per line)
        try:
            assert '\t' in raw_data_
            assert len(data_type_) == 2
```

```
            assert data_type_[0] == list
            value_ = raw_data_.split(sep='\t')
            for i in value_:
                if data_type_[-1] in [int, float]:
                    try:
                        i_ = data_type_[-1](i)
                    except ValueError:
                        return True
                else:
                    return False
            return False
        except AssertionError:
            return True


def extract(self, raw_data: str):
    value_error = self.value_error(raw_data)
    if value_error is True:
        # Can not extract
        return None, True
    else:
        # Can extract
        data_type_ = self.data_type[repetition_regex]
        raw_data_ = remove_new_line_char(raw_data)
        if data_type_ != tuple:
            if '\t' not in raw_data_:
                if data_type_ in [int, float]:
                    value_ = data_type_(raw_data_)
                else:
```

```
                value_ = raw_data_
            return value_, False
        else:
            value_ = raw_data_.split(sep='\t')
            value_ = extract_list_data_type(raw_data=value_)
            return value_, False
    else:
        assert '\t' in raw_data_
        assert data_type_[0] == list
        value_ = raw_data_.split(sep='\t')
        value_ = extract_list_data_type(raw_data=value_)
        return value_, False


def extract_between_keywords(current_keyword, next_keyword, raw_data):
    current_keyword_loc = 0
    next_keyword_loc = 0
    for (j, j_line) in enumerate(raw_data):
        if type(current_keyword) is not tuple:
            if current_keyword == j_line.split(sep='\n')[0]:
                current_keyword_loc = j + 1
        else:
            if (current_keyword[0] == raw_data[j].split(sep='\n')[0]) and \
                    (current_keyword[-1] == raw_data[j + 1].split(sep='\n')[0]):
                current_keyword_loc = j + 2
        if type(next_keyword) is not tuple:
            if next_keyword == j_line.split(sep='\n')[0]:
                next_keyword_loc = j
```

```
        else:
            if (next_keyword[0] == raw_data[j].split(sep='\n')[0]) and \
                    (next_keyword[-1] == raw_data[j + 1].split(sep='\n')[0]):
                next_keyword_loc = j
    if next_keyword is end_regex:
        raw_data_ = raw_data[current_keyword_loc:]
    else:
        raw_data_ = raw_data[current_keyword_loc: next_keyword_loc]
    for (_, data_) in enumerate(raw_data_):
        raw_data_[_] = remove_new_line_char(data_)
    return raw_data_


def extract_primitive_data_type(raw_data):
    # Extract for a single primitive data type (float, int, str)
    try:
        i = int(raw_data)
        return i
    except ValueError:
        try:
            i = float(raw_data)
            return i
        except ValueError:
            if raw_data.lower() == 'true':
                return True
            elif raw_data.lower == 'false':
                return False
            else:
```

```
            warnings.warn('Not a primitive data type or keyword string.')
            return raw_data


def extract_list_data_type(raw_data: List[str]):
    # Extract for a list of different primitive data types (float, int, str),
    data = [None] * len(raw_data)
    for (_, i) in enumerate(raw_data):
        i_ = remove_new_line_char(i)
        try:
            i_ = int(i_)
            data[_] = i_
        except ValueError:
            try:
                i_ = float(i_)
                data[_] = i_
            except ValueError:
                if i_.lower() == 'true':
                    data[_] = True
                elif i_.lower == 'false':
                    data[_] = False
                else:
                    warnings.warn('Not a primitive data type or keyword string.')
                    data[_] = i_
    data = remove_empty_in_block(data)
    return data
```

```
def extract_repeated_data_type(raw_data: List[str], data_type: dict, current_extract_loc: int):
    # Extract pre-defined data types, repeated >=2 lines
    value_error = False
    data = []
    while value_error is False and current_extract_loc < len(raw_data):
        repeated_data_type = RepeatedDataType(data_type=data_type)
        value, value_error = repeated_data_type.extract(raw_data=raw_data[current_extract_loc])
        # This if-else is to prevent adding None and stop counting current_extract_loc
        if value is None and value_error is True:
            pass
        else:
            data.append(value)
            current_extract_loc += 1
    if type(data_type[repetition_regex]) == tuple:
        if data_type[repetition_regex][-1] in [int, float]:
            data = np.array(data)
        else:
            pass
    else:
        pass
    return data, current_extract_loc


def extract_nested_data_type(raw_data: List[str], data_types: tuple, current_extract_loc: int):
    # Extract multiple primitive/pre-defined repeated datatypes nested in a tuple
    data = list()
    for data_type in data_types:
        if data_type in [int, float, str]:
```

```
            value = extract_primitive_data_type(raw_data[current_extract_loc])
            data += [value]
            current_extract_loc += 1
        elif data_type in [list]:
            value = raw_data[current_extract_loc].split(sep='\t')
            value = extract_list_data_type(value)
            data += value
            current_extract_loc += 1
        else:
            value, current_extract_loc = extract_repeated_data_type(raw_data, data_type,
current_extract_loc)
            data.append(value)
    return data, current_extract_loc



def extract_irregular_data_type(raw_data: List[str], data_types: list):
    data = list()
    current_extract_loc = 0
    for (_, data_type) in enumerate(data_types):
        if data_type in [int, float, str]:
            value = extract_primitive_data_type(raw_data[current_extract_loc])
            data += [value]
            current_extract_loc += 1
        elif data_type in [list]:
            value = raw_data[current_extract_loc].split(sep='\t')
            value = extract_list_data_type(value)
            data += value
            current_extract_loc += 1
        elif type(data_type) == dict:
```

```
            value, current_extract_loc = extract_repeated_data_type(raw_data, data_type, current_extract_loc)
            data.append(value)
        else:
            value, current_extract_loc = extract_nested_data_type(raw_data, data_type, current_extract_loc)
            data += value
    return data
import numpy as np


from src.base.base_libs import *
from simulator.base.regex_helpers import *



##########################################################################
# Regular expression templates to match data in ResFrac files
##########################################################################



class BaseRegex(object):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex).__init__()
        self.raw_str = raw_str

    def extract(self, pattern):
        pass



# TODO: ? Validate this class's robustness in extracting values from a list
```

```
class ListRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex).__init__()
        self.raw_str = raw_str

    def extract(self, pattern=None):
        """
        Extract data from list(list) of values separated by \t or from a single value
        (different primitive data types possible)
        :param pattern:
        :return: data:
        """
        data_ = remove_new_line_char(self.raw_str)
        if '\t' not in data_:
            data_ = extract_primitive_data_type(data_)
        else:
            data_ = data_.split(sep='\t')
            data_ = extract_list_data_type(data_)
        return data_


class ListOfListRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex).__init__()
        self.raw_str = raw_str

    def extract(self, pattern=None):
        """
```

```
        Extract data from list(list) of values separated by \t or from a single value
        (different primitive data types possible)
        :param pattern:
        :return: data:
        """
        data = [None] * len(self.raw_str)
        for (_, line) in enumerate(self.raw_str):
            data_ = remove_new_line_char(line)
            if '\t' not in data_:
                data_ = extract_primitive_data_type(data_)
            else:
                data_ = data_.split(sep='\t')
                data_ = extract_list_data_type(data_)
            data[_] = data_
        return data


class RelPermRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex).__init__()
        self.raw_str = raw_str
        self.rel_perm_regex = None
        self.rel_perm_data = None

    def set_rel_perm_model(self, raw_str):
        if 'BrooksCorey' in raw_str[3]:
            self.rel_perm_regex = brooks_corey_regex
            self.rel_perm_data = brooks_corey_data
```

```
    else:
        self.rel_perm_regex = x_curve_regex
        self.rel_perm_data = x_curve_data


def extract(self, pattern=None):
    """
    Extract data for relative permeability in ResFrac, using:
    - rel_perm_keyword_regex: required keywords for relative permeability
    - rel_perm_data_regex: required data types corresponding to the keywords
    :param pattern:
    :return: data
    """
    pattern_separator = '***'
    separator_locs = list()
    separator_locs.append(0)
    # Determine start/end locations of rel perm curves
    for (_, line) in enumerate(self.raw_str):
        if pattern_separator in line:
            separator_locs.append(_)
    raw_data = [None] * (len(separator_locs) - 1)
    # Extract raw data for all rel perm curves
    for i_rel_perm in range(len(separator_locs) - 1):
        i_raw_data = self.raw_str[separator_locs[i_rel_perm]: separator_locs[i_rel_perm + 1]]
        self.set_rel_perm_model(raw_str=i_raw_data)
        assert len(self.rel_perm_regex) == len(self.rel_perm_data) + 1
        i = 0
        i_rel_perm_data = dict()
        while i < len(self.rel_perm_data):
```

```
        # Extract raw data between 2 keywords
        current_keyword = self.rel_perm_regex[i]
        next_keyword = self.rel_perm_regex[i + 1]
        current_keyword_data = extract_between_keywords(current_keyword, next_keyword,
                                                        i_raw_data)
        i_rel_perm_data[current_keyword] = current_keyword_data
        i += 1
    raw_data[i_rel_perm] = i_rel_perm_data
# Extract data for all rel perm curves from raw data
data = [None] * (len(separator_locs) - 1)
for i_rel_perm in range(len(separator_locs) - 1):
    i_rel_perm_raw_data = raw_data[i_rel_perm]
    i_rel_perm_data = dict()
    for (i_regex, i_data_regex) in enumerate(self.rel_perm_data):
        i_keyword_regex = self.rel_perm_regex[i_regex]
        i_data_regex = self.rel_perm_data[i_regex]
        i_data = i_rel_perm_raw_data[i_keyword_regex]
        current_extract_loc = 0
        while current_extract_loc < len(i_data):
            if i_data_regex in [int, float, str]:
                i_value = extract_primitive_data_type(i_data[current_extract_loc])
                current_extract_loc += 1
            elif i_data_regex in [list]:
                i_value = i_data[current_extract_loc].split(sep='\t')
                i_value = extract_list_data_type(i_value)
                current_extract_loc += 1
            elif type(i_data_regex) == dict:
                i_value, current_extract_loc = extract_repeated_data_type(i_data, i_data_regex,
```

```
                                                    current_extract_loc)
                else:
                    assert type(i_data_regex) == tuple

                    i_value, current_extract_loc = extract_nested_data_type(i_data, i_data_regex,
                                                    current_extract_loc)
            i_rel_perm_data[i_keyword_regex] = i_value
        data[i_rel_perm] = i_rel_perm_data
    return data


class WellRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex).__init__()
        self.raw_str = raw_str

    def extract(self, pattern=None):
        N = len(self.raw_str)
        well_scopes = list()
        well_sequences = dict()
        for n in range(N):
            if well_scope_regex in self.raw_str[n]:
                well_scopes.append(n)
        for iw in range(len(well_scopes) - 1):
            well_data = self.raw_str[well_scopes[iw] + 1: well_scopes[iw + 1] + 1]
            well_name = well_data[0].split(sep='\n')[0]
            sequence_scopes = list()
            for (iwd, well_line) in enumerate(well_data):
```

```
            if sequence_scope_regex in well_line:
                sequence_scopes.append(iwd)
        sequences_ = {'Shut-in': None, 'Injection': None, 'Production': None}
        for iq in range(len(sequence_scopes) - 1):
            sequence_data = well_data[sequence_scopes[iq] + 1: sequence_scopes[iq + 1]]
            sequence_data = [remove_new_line_char(data_) for data_ in sequence_data]
            sequence_regex = SequenceRegex(sequence_data)
            sequence_type = sequence_regex.get_sequence_type()
            if sequence_type == 'Injection':
                sequence_ = sequence_regex.extract(injection_sequence_data_regex)
                sequences_['Injection'] = sequence_
            elif sequence_type == 'Production':
                sequence_ = sequence_regex.extract(production_sequence_data_regex)
                sequences_['Production'] = sequence_
            else:
                sequence_ = sequence_regex.extract(shut_in_sequence_data_regex)
                sequences_['Shut-in'] = sequence_
            well_sequences[well_name] = sequences_
    return well_sequences


class SequenceRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex).__init__()
        self.raw_str = raw_str

    def get_sequence_type(self):
        # Sequence type is always located at 1st line
```

```
        sequence_type = remove_new_line_char(self.raw_str[0])
        if 'Injection' in sequence_type:
            return 'Injection'
        elif 'Production' in sequence_type:
            return 'Production'
        else:
            return 'Shut-in'

    def extract(self, pattern):
        # TODO: ? Re-write this function
        data = extract_irregular_data_type(self.raw_str, pattern)
        return data


class FaciesListRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex).__init__()
        self.raw_str = raw_str
        self.num_layers = None

    def get_number_of_layers(self):
        self.num_layers = len(self.raw_str)

    def extract(self, pattern='\t'):
        data = dict()
        self.get_number_of_layers()
        try:
            assert self.num_layers is not None
```

```
        except AssertionError:
            warnings.warn('Can not extract due to incorrect number of layers.')
            sys.exit(1)
        for i in range(self.num_layers):
            layer_data = remove_new_line_char(self.raw_str[i])
            layer_data = layer_data.split(sep=pattern)
            layer_name = layer_data[0]
            layer_data = layer_data[1:]
            layer_data = remove_empty_in_block(layer_data)
            layer_data = ListOfListRegex(raw_str=layer_data).extract()
            data[layer_name] = layer_data
        return data


class BlackOilRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str

    def extract(self, pattern=None):
        black_oil_props = remove_new_line_char(self.raw_str[0])
        black_oil_props = ListRegex(raw_str=black_oil_props).extract()
        prop_table_lines = remove_new_line_char(self.raw_str[1])
        prop_table_lines = int(prop_table_lines)
        prop_table = self.raw_str[2:]
        prop_table = ListOfListRegex(raw_str=prop_table).extract()
        return [black_oil_props, prop_table_lines, np.array(prop_table)]
```

```
class ClustersPerStageRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str

    def get_number_of_stages(self):
        num_of_stages = 0
        locs = list()
        for (_, line) in enumerate(self.raw_str):
            if '***' in line:
                num_of_stages += 1
                locs.append(_)
        locs.append(len(self.raw_str))
        return num_of_stages, locs

    def extract(self, pattern='\t'):
        data = dict()
        num_of_stages, locs = self.get_number_of_stages()
        for _ in range(num_of_stages):
            start_loc, end_loc = locs[_] + 1, locs[_ + 1]
            _data = self.raw_str[start_loc: end_loc]
            _data = ListOfListRegex(raw_str=_data).extract()
            data['Stage_' + str(_+1)] = _data
        return data


class WaterByLayer(BaseRegex):
```

```
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str
        self.num_of_layers = None

    def get_number_of_layers(self):
        num_of_layers = 0
        locs = list()
        for (_, line) in enumerate(self.raw_str):
            if '***' in line:
                num_of_layers += 1
                locs.append(_ + 1)
        locs.append(len(self.raw_str))
        return num_of_layers, locs

    def extract(self, pattern=None):
        data = dict()
        num_of_layers, locs = self.get_number_of_layers()
        for _ in range(num_of_layers):
            _data = remove_new_line_char(self.raw_str[locs[_]])
            data['Layer_' + str(_+1)] = float(_data)
        return data


class PropRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str
```

```
        self.num_of_props = None

    def get_number_of_props(self):
        return len(self.raw_str)

    def extract(self, pattern='\t'):
        data = dict()
        for prop in self.raw_str:
            prop_data = remove_new_line_char(prop)
            prop_data = prop_data.split(sep=pattern)
            prop_name = prop_data[0]
            prop_data = prop_data[1:]
            prop_data = ListOfListRegex(raw_str=prop_data).extract()
            data[prop_name] = prop_data
        return data


class PropMixtureRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str
        self.num_of_mixtures = None

    def get_number_of_mixtures(self):
        return len(self.raw_str)

    def extract(self, pattern='\t'):
        data = dict()
```

```
        for mix in self.raw_str:
            mix_data = remove_new_line_char(mix)
            mix_data = mix_data.split(sep=pattern)
            mix_name = mix_data[0]
            mix_data = mix_data[1:]
            mix_data = ListOfListRegex(raw_str=mix_data).extract()
            data[mix_name] = mix_data
        return data



class WaterSoluteRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str
        self.num_of_mixtures = None

    def get_number_of_solutes(self):
        return len(self.raw_str)

    def extract(self, pattern=None):
        data = dict()
        for solute in self.raw_str:
            solute_data = solute.split(sep='\n')[0]
            solute_data = solute_data.split(sep=pattern)
            solute_name = solute_data[0]
            solute_data = solute_data[1:]
            solute_data = ListOfListRegex(raw_str=solute_data).extract()
            data[solute_name] = solute_data
```

```
        return data


class FluidMixtureRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str
        self.num_of_mixtures = None

    def get_number_of_mixtures(self):
        return len(self.raw_str)

    def extract(self, pattern=None):
        data = dict()
        for mix in self.raw_str:
            mix_data = mix.split(sep='\n')[0]
            mix_data = mix_data.split(sep=pattern)
            mix_name = mix_data[0]
            mix_data = mix_data[1:]
            mix_data = ListOfListRegex(raw_str=mix_data).extract()
            data[mix_name] = mix_data
        return data


class PropPermModelRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str
```

```
    def extract(self, pattern=None):
        data = ListOfListRegex(raw_str=self.raw_str).extract()
        return np.array(data)


class FracPermModelRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str

    def extract(self, pattern=None):
        data = ListOfListRegex(raw_str=self.raw_str).extract()
        return np.array(data)


class DurationCutOffRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str

    def get_num_of_wells(self):
        num_of_wells = 0
        locs = list()
        for (_, line) in enumerate(self.raw_str):
            if '***' in line:
                num_of_wells += 1
                locs.append(_)
```

```
        locs.append(len(self.raw_str))
        return num_of_wells, locs

    def extract(self, pattern=None):
        data = dict()
        num_of_wells, locs = self.get_num_of_wells()
        for _ in range(num_of_wells):
            start_loc, end_loc = locs[_] + 1, locs[_ + 1]
            _data = self.raw_str[start_loc: end_loc]
            _data = ListOfListRegex(raw_str=_data).extract()
            data['Well_' + str(_+1)] = _data
        return data


class WellVerticesRegex(BaseRegex):
    def __init__(self, raw_str: List[str]):
        super(BaseRegex, self).__init__()
        self.raw_str = raw_str

    def get_num_of_wells(self):
        num_of_wells = 0
        locs = list()
        for (_, line) in enumerate(self.raw_str):
            if '***' in line:
                num_of_wells += 1
                locs.append(_ + 1)
        locs.append(len(self.raw_str))
        return num_of_wells, locs
```

```
    def extract(self, pattern=None):
        data = dict()
        num_of_wells, locs = self.get_num_of_wells()
        for _ in range(num_of_wells):
            start_loc, end_loc = locs[_], locs[_ + 1] - 1
            _data = self.raw_str[start_loc: end_loc]
            _data = ListOfListRegex(raw_str=_data).extract()
            data['Well_' + str(_+1)] = _data
        return data



irregular_regex_classes = [RelPermRegex, FaciesListRegex, BlackOilRegex,
                           ClustersPerStageRegex, WaterByLayer,
                           PropRegex, PropMixtureRegex,
                           PropPermModelRegex, FracPermModelRegex,
                           WaterSoluteRegex, FluidMixtureRegex,
                           WellRegex, DurationCutOffRegex, WellVerticesRegex]
from src.base.base_libs import *



def write_matrix_values(file, array: np.ndarray):
    """
    The numpy array has dimensions (N, 1) or (N, 2)
    :param array:
    :param file:
    :return:
    """
```

```
    for row in array:
        if type(row) == np.ndarray:
            for _ in row:
                file.write(str(_))
                file.write('\t')
            file.write('\n')
        else:
            file.write(str(row))
            file.write('\n')
    file.write('\n')


def write_ndarray_values(file, array: np.ndarray, length: int):
    """
    The numpy array has dimensions (N, 1) or (N, 2)
    :param array:
    :param file:
    :param length:
    :return:
    """
    if length == 0:
        warnings.warn('Incorrect variable length for matrix values.')
        sys.exit(1)
    elif length == 1:
        for _ in array:
            file.write(str(_))
            file.write('\t')
        file.write('\n')
```

```
    else:
        for row in array:
            if type(row) == np.ndarray:
                for _ in row:
                    file.write(str(_))
                    file.write('\t')
                file.write('\n')
            else:
                file.write(str(row))
                file.write('\n')
    file.write('\n')


def write_list_values(file, values: list):
    """
    TODO: ? Write a numpy array dimensions to an opened file object
    values is a list of different data types
    :param values:
    :param file:
    """
    for value in values:
        file.write(str(value))
        file.write('\t')
    file.write('\n')


def write_list_of_list_values(file, values: List[List]):
    for value in values:
```

```
        if type(value) != list:
            file.write(str(value))
            file.write('\n')
        else:
            for value_ in value:
                file.write(str(value_))
                file.write('\t')
            file.write('\n')
    file.write('\n')



def parse_file(file_name):
    """
    TODO: ? Scan and extract simulation data from a ResFrac's settings/input file
    :param file_name:
    :return:
    """
    file = open(file=file_name, mode='r')
    file_lines = file.readlines()
    entry_begin_idx = list()
    entry_end_idx = list()
    file_entries = list()
    for (i, line) in enumerate(file_lines):
        if 'Begin entry' in line:
            # Begin entry comment
            entry_begin_idx.append(i)
        elif 'End entry' in line:
            # End entry comment
```

```
        entry_end_idx.append(i)
    else:
        # Entry comments or data
        pass
try:
    assert len(entry_begin_idx) == len(entry_end_idx)
except AssertionError:
    sys.exit(1)
number_of_entries = len(entry_begin_idx)
for ie in range(number_of_entries):
    entry = dict.fromkeys(['Comments:', 'Variable name:', 'Length:', 'Value(s):'])
    entry['Comments:'] = list()
    entry['Variable name:'] = list()
    entry['Length:'] = list()
    entry['Value(s):'] = list()
    entry_line_data = file_lines[entry_begin_idx[ie]:entry_end_idx[ie]+1]
    for (je, entry_line) in enumerate(entry_line_data):
        if entry_line.__contains__("//"):
            entry['Comments:'].append(entry_line)
        elif 'Variable name:' in entry_line:
            entry['Variable name:'].append(entry_line_data[je+1])
        elif 'Length:' in entry_line:
            entry['Length:'].append(entry_line_data[je+1])
        elif 'Value(s):' in entry_line:
            value_line_data = entry_line_data[je+1: -1]
            remove_comment_in_block(value_line_data)
            while '\n' in value_line_data:
                remove_new_line_char_in_block(value_line_data)
```

```
            for value_line in value_line_data:
                if value_line.__contains__("//"):
                    pass
                else:
                    entry['Value(s):'].append(value_line)
        else:
            pass
    file_entries.append(entry)
file.close()
return file_entries


def remove_new_line_char(line):
    if line == '\n':
        return None
    elif '\n' in line:
        return line.split(sep='\n')[0]
    else:
        return line


def remove_new_line_char_in_block(block: List[str]):
    for line in block:
        if line == '\n':
            block.remove(line)


def remove_comment_in_block(block: List[str]):
```

```
    for line in block:
        if line.__contains__("//"):
            block.remove(line)


def remove_empty_in_block(block: List[str]):
    block_ = block
    if '' in block_:
        block_.remove('')
    elif ' ' in block_:
        block_.remove(' ')
    else:
        pass
    return block_


def compare_two_txt_files(file_name_1, file_name_2, comp_file_1, comp_file_2):
    f1 = open(file_name_1, 'r')
    f2 = open(file_name_2, 'r')
    file_data_1 = f1.readlines()
    file_data_2 = f2.readlines()
    for (_, line) in enumerate(file_data_1):
        if '//' in line:
            file_data_1.remove(line)
        elif line in ['\n', '\r\n']:
            file_data_1.remove(line)
        else:
            pass
```

```
    for (_, line) in enumerate(file_data_2):
        if '//' in line:
            file_data_2.remove(line)
        elif line in ['\n', '\r\n']:
            file_data_2.remove(line)
        else:
            pass
    try:
        assert len(file_data_1) == len(file_data_2)
    except AssertionError:
        warnings.warn('Compared files do not hold equal data.')
    diff = 0
    for (_, line) in enumerate(file_data_2):
        if line not in file_data_1:
            print(line)
            diff += 1
    print('Total number of different lines is ' + str(diff))
    with open(comp_file_1, 'w+') as cf1:
        for line in file_data_1:
            cf1.write(line)
    with open(comp_file_2, 'w+') as cf2:
        for line in file_data_2:
            cf2.write(line)
    f1.close()
    f2.close()
    cf1.close()
    cf2.close()
"""
```

```
    TODO: ? Store ResFrac comments in the settings or input files
    - To support writing/reading ResFrac simulation files
    - Comment files start with // (not scanned by ResFrac)
"""


entry_begin = '// ------------------ Begin entry ------------------------ '
entry_end = '// ------------------ End entry ------------------------ '
entry_describe = '// Description: '
entry_name = '// Name in builder interface: '
entry_internal_variable_name = '// ResFrac internal variable name '
from simulator.base.entry import *
from simulator.simulation.simulation_headers import *
from simulator.simulation.simulation_comments import entry_begin, entry_end


class SimulationFile(object):
    def __init__(self, entries: List[Entry], file_type: str):
        super(SimulationFile, self).__init__()
        self.entries = entries
        self.file_type = file_type

    def add_entry(self, entry: Entry):
        self.entries.append(entry)

    def write_file(self, file_name):
        """
        TODO: ? Implement writing all entries to a .txt file
```

```
:return:
"""
with open(file_name, mode='w+') as f:
    if self.file_type == 'settings':
        for header in settings_headers:
            f.write(header)
            f.write('\n')
    elif self.file_type == 'input':
        for header in input_headers:
            f.write(header)
            f.write('\n')
    else:
        warnings.warn('Not a ResFrac file type.')
        sys.exit(1)
    for e in self.entries:
        # print('Write entry: ', e.variable_name)
        f.write(entry_begin)
        f.write("\n")
        f.write("Variable name:")
        f.write("\n")
        f.write(e.variable_name)
        f.write("\n")
        f.write("Length:")
        f.write("\n")
        f.write(str(e.length))
        f.write("\n")
        f.write("Value(s):")
        f.write("\n")
```

```
            e.write_entry_values(f)
            f.write("\n")
            f.write(entry_end)
            f.write("\n")
            f.write("\n")
        f.close()
"""
   ResFrac commands in settings/input files that does not belong to an entry
"""

settings_headers = ['validation passed', 'memory usage regular', 'using field units']
input_headers = ['using field units']
from src.base.base_libs import *

#########################################################################
# Helper variables to support parsing/writing ResFrac files
#########################################################################


rel_perm_keys = ['curvesetname', 'matrixrelperm',
                 ('pressuredependentpermeability', 'reversible'),
                 ('pressuredependentpermeability', 'irreversible'),
                 'tenxreversiblepermeabilitylossperpressureincrement',
                 'tenxirreversiblepermeabilitylossperpressureincrement',
                 'lowerpressurethresholdforreversiblepermeabilityincrease',
                 'upperpressurethresholdforreversiblepermeabilityincrease',
                 'permmultiplierforreversiblepermeabilityincrease',
                 'lowerpressurethresholdforirreversiblepermeabilityincrease',
```

```
            'upperpressurethresholdforirreversiblepermeabilityincrease',
            'permmultiplierforirreversiblepermeabilityincrease',
            'permeabilitymultiplier',
            'waterbankthicknesstorelpermincreasescalingthickness',
            'waterbankthicknesstorelpermdecreasescalingthickness',
            'waterbankimmobilefraction']

irregular_variable_names = ['matrixcurvesets', 'facieslist', 'blackoil',
                            'clustersperstage', 'initialwatersolutemassfractionsbylayer',
                            'proppants', 'proppantmixtures',
                            'proppantbedbrookscoreymodel', 'fracturebrookscoreymodel',
                            'watersolutes', 'fluidmixtures',
                            'nextgenboundaryconditioncontrols', 'durationcutoff', 'wellvertices']
import sys
import os
import shutil

import math
import csv
import warnings

from typing import Union, List, Optional, Tuple, Dict
from itertools import permutations, combinations, product
from copy import deepcopy
from enum import Enum

import regex
import re
```

```
import pickle

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import scipy
from scipy.stats import distributions
import pyDOE

import plotly
from plotly import graph_objects as go
from src.dir.dir import *



class OrientedBox(object):
    """
    TODO: Class that manages an oriented box, i.e., a box representing the 3D fracture element in ResFrac
    Description:
    Functionality:
    """
    def __init__(self, central_coors: np.asarray, dimensions: np.ndarray, oriented_angle: float):
        super(OrientedBox, self).__init__()
        self.central_coors = central_coors
        self.dimensions = dimensions  # dimensions = [element dimensions and element aperture]
        self.oriented_angle = oriented_angle
        #
```

```
        self.central_plane_coors = np.zeros(shape=[4, 3])  # Coordinates of 'central' plane's corners
        self.coors = np.zeros(shape=[8, 3])  # Coordinates of the oriented box's corners
        #
        self.normal = np.nan * np.ones((3, ))  # Normal vector of the oriented box and the central plane


    def compute_central_corners(self):
        # Compute the corners of the oriented box
        center_x = self.central_coors[0]
        center_y = self.central_coors[1]
        center_z = self.central_coors[-1]
        corners_x = [center_x + self.dimensions[0] / 2 * np.sin(self.oriented_angle),
                     center_x - self.dimensions[0] / 2 * np.sin(self.oriented_angle)]
        corners_y = [center_y + self.dimensions[0] / 2 * np.cos(self.oriented_angle),
                     center_y - self.dimensions[0] / 2 * np.cos(self.oriented_angle)]
        corners_z = [center_z + self.dimensions[1] / 2,
                     center_z - self.dimensions[1] / 2]
        coors_xy = list()
        coors_z = corners_z
        coors_xy.append((corners_x[0], corners_y[0]))
        coors_xy.append((corners_x[-1], corners_y[-1]))
        coors = list(product(coors_xy, coors_z))
        for i in range(4):
            x = coors[i][0][0]
            y = coors[i][0][-1]
            z = coors[i][-1]
            self.central_plane_coors[i, :] = [x, y, z]


    def compute_corners(self):
```

```
    self.compute_central_corners()
    for i in range(4):
        central_x = self.central_plane_coors[i, 0]
        central_y = self.central_plane_coors[i, 1]
        central_z = self.central_plane_coors[i, -1]
        self.coors[2 * i, :] = [central_x - self.dimensions[-1]/2 * np.cos(self.oriented_angle),
                                central_y + self.dimensions[-1]/2 * np.sin(self.oriented_angle), central_z]
        self.coors[2 * i + 1, :] = [central_x + self.dimensions[-1]/2 * np.cos(self.oriented_angle),
                                    central_y - self.dimensions[-1]/2 * np.sin(self.oriented_angle), central_z]

def compute_normal(self):
    self.compute_central_corners()
    p1 = self.central_plane_coors[0, :]
    p2 = self.central_plane_coors[1, :]
    p3 = self.central_plane_coors[2, :]
    self.normal = np.cross(p2 - p1, p3 - p1)

def sample(self):
    # Sample a random location inside the oriented box
    loc = [0., 0., 0.]
    min_x_coor, max_x_coor = self.coors[:, 0].min(), self.coors[:, 0].max()
    min_y_coor, max_y_coor = self.coors[:, 1].min(), self.coors[:, 1].max()
    min_z_coor, max_z_coor = self.coors[:, -1].min(), self.coors[:, -1].max()
    loc[0] = np.random.uniform(min_x_coor, max_x_coor)
    loc[1] = np.random.uniform(min_y_coor, max_y_coor)
    loc[-1] = np.random.uniform(min_z_coor, max_z_coor)
    return loc
```

```
class FractureElement(object):
    def __init__(self, elem_center: np.ndarray, elem_dimensions: np.ndarray,
                 elem_aperture: float, elem_angle: float):
        super(FractureElement, self).__init__()
        self.elem_center = elem_center
        self.elem_dimensions = elem_dimensions
        self.elem_aperture = elem_aperture / 12  # Convert in to ft
        self.elem_angle = elem_angle / 180 * math.pi  # Convert deg to rad
        # Attributes defining the element in 2D (central plane) and 3D (oriented box)
        self.elem_corners = None
        self.elem_coors = None
        self.elem_box = None

    def set_element_corners(self):
        """
        TODO: ? Compute element corners from its dimensions and center
        :return:
        """
        elem_center_x = self.elem_center[0]
        elem_center_y = self.elem_center[1]
        elem_center_z = self.elem_center[-1]
        elem_corners_x = [elem_center_x + self.elem_dimensions[0] / 2 * np.sin(self.elem_angle),
                          elem_center_x - self.elem_dimensions[0] / 2 * np.sin(self.elem_angle)]
        elem_corners_y = [elem_center_y + self.elem_dimensions[0] / 2 * np.cos(self.elem_angle),
                          elem_center_y - self.elem_dimensions[0] / 2 * np.cos(self.elem_angle)]
        elem_corners_z = [elem_center_z + self.elem_dimensions[-1] / 2,
                          elem_center_z - self.elem_dimensions[-1] / 2]
```

```
    self.elem_corners = np.zeros([3, 2])
    self.elem_corners[0, :] = elem_corners_x
    self.elem_corners[1, :] = elem_corners_y
    self.elem_corners[-1, :] = elem_corners_z

def set_element_coordinates(self):
    if self.elem_corners is None:
        pass
    else:
        # This mode computes coordinate of the element's central 'plane' (i.e.
        # not include aperture, replicate similar visual as in ResFrac)
        self.elem_coors = np.zeros([4, 3])
        elem_coors_xy = list()
        elem_coors_z = list(self.elem_corners[-1, :])
        elem_coors_xy.append((self.elem_corners[0, 0], self.elem_corners[1, 0]))
        elem_coors_xy.append((self.elem_corners[0, 1], self.elem_corners[1, 1]))
        elem_coors = list(product(elem_coors_xy, elem_coors_z))
        for i in range(4):
            x = elem_coors[i][0][0]
            y = elem_coors[i][0][-1]
            z = elem_coors[i][-1]
            self.elem_coors[i, :] = [x, y, z]

def generate_elem_2d_grid(self):
    assert self.elem_coors.shape[0] == 4
    elem_grid = np.zeros([5, 3])
    elem_grid[0, :] = self.elem_coors[0, :]
    elem_grid[1, :] = self.elem_coors[1, :]
```

```
        elem_grid[2, :] = self.elem_coors[-1, :]
        elem_grid[3, :] = self.elem_coors[2, :]
        elem_grid[-1, :] = self.elem_coors[0, :]
        return elem_grid

    def set_oriented_box(self):
        elem_box_dimensions = np.zeros(shape=(3,), dtype=np.float32)
        elem_box_dimensions[:-1] = self.elem_dimensions
        elem_box_dimensions[-1] = self.elem_aperture
        self.elem_box = OrientedBox(central_coors=self.elem_center,
dimensions=elem_box_dimensions,
                                    oriented_angle=self.elem_angle)

    def generate_elem_3d_grid(self):
        pass
from src.utils.read_data import *
from src.data.properties.properties import *
from src.data.element.element import *


class FractureGeometry(SimulationProperty):
    def __init__(self, raw_result_dir: RawResultDirectory):
        super(SimulationProperty).__init__()
        self.property_names = ['Elm center x',
                               'Elm center y',
                               'Elm center z (depth)',
                               'Angle', 'Aperture',
                               'Element number',
                               'Fracture element number',
```

```
                    'Fracture number']
    self.property_dimensions = ['ft', 'ft', 'ft',
                                'Degrees clockwise from positive y-axis direction',
                                'in',
                                'unitless', 'unitless', 'unitless']
    self.raw_result_dir = raw_result_dir
    self.time_step = None

    self.fracture_elements = None
    self.fracture_numbers = None
    self.fracture_surface = None

    self.element_dimensions = None
    self.element_numbers = None

def set_raw_result_file(self, raw_result_file):
    self.raw_result_dir.set_new_result_dir(raw_result_file)
    raw_result_file_encoding = raw_result_file.split(sep="_")
    self.time_step = raw_result_file_encoding[-1]

def set_fracture_geometry_data(self, file_data, file_data_format: DataFormat):
    self.set_property_data(file_data, file_data_format)
    if len(self.property_data) == 0:
        pass
    else:
        fracture_numbers_data = np.array([n for n in self.property_data[:, -1]])
        self.fracture_numbers = np.unique(fracture_numbers_data)
        self.element_dimensions = np.array(
```

```
            [file_data_format.data_header[-5], file_data_format.data_header[-2]]).astype(
            float)
        self.element_numbers = self.property_data.shape[0]
        self.fracture_elements = list()
        for elem_num in range(self.element_numbers):
            elem_center = np.array(self.property_data[elem_num, :3]).astype(float)
            elem_angle = float(self.property_data[elem_num, 3])
            elem_aperture = float(self.property_data[elem_num, 4])
            elem = FractureElement(elem_center=elem_center,
elem_dimensions=self.element_dimensions,
                                   elem_angle=elem_angle, elem_aperture=elem_aperture)
            elem.set_element_corners()
            elem.set_element_coordinates()
            self.fracture_elements.append(elem)

    def set_fracture_surface_data(self):
        if self.fracture_elements is None or self.element_numbers is None:
            pass
        else:
            self.fracture_surface = np.zeros(shape=[4 * self.element_numbers, 3])
            for elem_num in range(self.element_numbers):
                self.fracture_surface[4 * elem_num:4 * (elem_num + 1), :] =
self.fracture_elements[elem_num].elem_coors

    def plot_fracture(self, fracture_number: Union[str, int], fig_name: str):
        """
        TODO: ? Plot fracture as surface for the given fracture number in ResFrac
        TODO: ? Consider moving this function out
        :param fracture_number:
```

```
:param fig_name:
:return:
"""
if fracture_number != -1:
    fracture_number_geometry = self.property_data[self.property_data[:, -1]
                                                  == fracture_number]
    fracture_x = fracture_number_geometry[:, 0].astype(float)
    fracture_y = fracture_number_geometry[:, 1].astype(float)
    fracture_z = fracture_number_geometry[:, 2].astype(float)
else:
    fracture_number_geometry = self.fracture_surface
    if fracture_number_geometry is not None:
        fracture_x = fracture_number_geometry[:, 0].astype(float)
        fracture_y = fracture_number_geometry[:, 1].astype(float)
        fracture_z = fracture_number_geometry[:, 2].astype(float)
fig_data = list()
if self.element_numbers is None:
    pass
else:
    for elem_num in range(self.element_numbers):
        elem_grid = self.fracture_elements[elem_num].generate_elem_2d_grid()
        fig_data.append(go.Scatter3d(x=elem_grid[:, 0],
                                     y=elem_grid[:, 1],
                                     z=elem_grid[:, -1],
                                     mode='lines', marker=dict(color="black"),
                                     name=''))
    fig = go.Figure(data=fig_data)
    fig.update_layout(scene=dict(
```

```
            aspectmode='manual',
            aspectratio=dict(x=10, y=1, z=5)))
        fig_dir = os.path.join(self.raw_result_dir.data_main_dir, fig_name)
        fig.write_html(fig_dir)
from src.utils.read_data import *
from src.dir.dir import *


class SimulationProperty(object):
    def __init__(self):
        super(SimulationProperty, self).__init__()
        self.property_names = None
        self.property_dimensions = None
        self.property_units = None
        self.property_data = None

    def set_property_name(self, name):
        self.property_names = name

    def set_property_dimensions(self, dimensions):
        self.property_dimensions = dimensions

    def set_property_units(self, units):
        self.property_units = units

    def set_property_data(self, file_data, file_data_format):
        self.property_data = property_reader(file_data, file_data_format,
                                             self.property_names)
```

```
class SimulationTrackProperty(object):
    def __init__(self):
        super(SimulationTrackProperty, self).__init__()
        self.property_names = None
        self.property_dimensions = None
        self.property_units = None
        self.property_data = None



class DailyProductionProperty(object):
    def __init__(self):
        super(DailyProductionProperty, self).__init__()
        self.property_names = None
        self.property_dimensions = None
        self.property_units = None
        self.property_data = None



def property_reader(file_data, file_data_format: DataFormat, property_names):
    searched_idx = search_properties(file_data_format, property_names)
    property_data = list()
    for i, data in enumerate(file_data):
        property_data.append([data[idx] for idx in searched_idx])
    return np.array(property_data)
from src.utils.read_data import *
from src.data.properties.properties import *
```

```
from src.data.element.element import *


class PropVolFrac(SimulationProperty):
    def __init__(self, raw_result_dir: RawResultDirectory):
        super(SimulationProperty).__init__()
        self.property_names = ['Total proppant volume fraction']
        self.property_dimensions = ['unitless']
        self.raw_result_dir = raw_result_dir
        #
        self.time_step = None
        self.fracture_numbers = None
        self.element_dimensions = None
        self.element_numbers = None

    def set_raw_result_file(self, raw_result_file):
        self.raw_result_dir.set_new_result_dir(raw_result_file)
        raw_result_file_encoding = raw_result_file.split(sep="_")
        self.time_step = raw_result_file_encoding[-1]

    def set_vol_frac_data(self, file_data, file_data_format: DataFormat):
        self.set_property_data(file_data, file_data_format)
        if len(self.property_data) == 0:
            pass
        else:
            fracture_numbers_data = np.array([n for n in self.property_data[:, -1]])
            self.fracture_numbers = np.unique(fracture_numbers_data)
            self.element_dimensions = np.array([file_data_format.data_header[-5],
file_data_format.data_header[-2]]).astype(
```

```
            float)
        self.element_numbers = self.property_data.shape[0]

    def plot_vol_frac(self):
        pass
from src.utils.read_data import *
from src.data.properties.properties import *
from src.data.element.element import *


class Aperture(SimulationProperty):
    # TODO: ? Define class to analyze fracture aperture data
    def __init__(self, raw_result_dir: RawResultDirectory):
        super(SimulationProperty).__init__()
        self.property_names = ['Aperture']
        self.property_dimensions = ['in']
        self.raw_result_dir = raw_result_dir

        self.time_step = None
        self.fracture_numbers = None
        self.element_dimensions = None
        self.element_numbers = None

    def set_raw_result_file(self, raw_result_file):
        self.raw_result_dir.set_new_result_dir(raw_result_file)
        raw_result_file_encoding = raw_result_file.split(sep="_")
        self.time_step = raw_result_file_encoding[-1]
```

```
    def set_aperture_data(self, file_data, file_data_format: DataFormat):
        self.set_property_data(file_data, file_data_format)
        if len(self.property_data) == 0:
            pass
        else:
            fracture_numbers_data = np.array([n for n in self.property_data[:, -1]])
            self.fracture_numbers = np.unique(fracture_numbers_data)
            self.element_dimensions = np.array([file_data_format.data_header[-5],
                                                file_data_format.data_header[-2]]).astype(float)
            self.element_numbers = self.property_data.shape[0]

    def plot_aperture(self):
        pass
from src.dir.dir import *
from src.utils.read_data import *


class TimeProperty(object):
    def __init__(self, data_main_dir: str, data_type: str):
        super(TimeProperty, self).__init__()
        self.raw_result_dir = RawResultDirectory(data_main_dir=data_main_dir)
        self.file_type = data_type

    def set_raw_result_file(self, raw_result_file: str):
        self.raw_result_dir.set_new_result_dir(raw_result_file)

    def get_raw_data(self, data_format: SimulationTrackDataFormat):
        data_reader = SimulationTrackDataReader(data_dir=self.raw_result_dir,
data_format=data_format)
```

```
        file_data, file_data_format = data_reader.read_data()
        return file_data, file_data_format
from src.base.base_libs import *


class DataDirectory(object):
    def __init__(self, data_main_dir):
        super(DataDirectory, self).__init__()
        self.data_main_dir = data_main_dir

    def set_new_directory(self, new_data_main_dir):
        self.data_main_dir = new_data_main_dir

    def access_directory(self, accessed_dir):
        return os.path.join(self.data_main_dir, accessed_dir)

    def search_file(self, accessed_dir, searched_file_name):
        searched_dir = self.access_directory(accessed_dir=accessed_dir)
        for file_name in os.listdir(searched_dir):
            if file_name == searched_file_name:
                return True
        return False

    def search_file_end_with(self, accessed_dir, searched_file_name, end_str):
        searched_dir = self.access_directory(accessed_dir=accessed_dir)
        for file_name in os.listdir(searched_dir):
            if file_name == searched_file_name:
                if file_name.endswith(end_str):
```

```
                return True
            else:
                pass
        else:
            pass
    return False

def search_file_include(self, accessed_dir, searched_file_name, include_str):
    searched_dir = self.access_directory(accessed_dir=accessed_dir)
    for file_name in os.listdir(searched_dir):
        if file_name == searched_file_name:
            if include_str in file_name:
                return True
            else:
                pass
        else:
            pass
    return False

def find_file_end_with(self, accessed_dir, end_str):
    searched_dir = self.access_directory(accessed_dir=accessed_dir)
    searched_file_names = list()
    for file_name in os.listdir(searched_dir):
        if file_name.endswith(end_str):
            searched_file_names.append(file_name)
    return searched_file_names

def find_file_include(self, accessed_dir, include_str):
```

```
        searched_dir = self.access_directory(accessed_dir=accessed_dir)
        searched_file_names = list()
        for file_name in os.listdir(searched_dir):
            if include_str in file_name:
                searched_file_names.append(file_name)
        return searched_file_names

    def move(self, new_dir):
        shutil.move(self.data_main_dir, new_dir)


class InputDirectory(DataDirectory):
    def __init__(self, data_main_dir):
        super(DataDirectory, self).__init__(data_main_dir)
        self.input_dir = None

    def set_new_input_directory(self, new_input_dir):
        self.input_dir = self.access_directory(new_input_dir)


class SettingDirectory(DataDirectory):
    def __init__(self, data_main_dir):
        super(SettingDirectory, self).__init__()
        self.data_main_dir = data_main_dir
        self.setting_dir = None

    def set_new_setting_dir(self, new_setting_dir):
        self.setting_dir = self.access_directory(new_setting_dir)
```

```
class RawResultDirectory(DataDirectory):
    def __init__(self, data_main_dir):
        super(DataDirectory, self).__init__()
        self.data_main_dir = data_main_dir
        self.result_dir = None

    def set_new_result_dir(self, new_result_dir):
        self.result_dir = self.access_directory(accessed_dir=new_result_dir)



class ProcessedResultDirectory(DataDirectory):
    def __init__(self, data_main_dir):
        super(DataDirectory, self).__init__()
        self.data_main_dir = data_main_dir
        self.result_dir = None

    def set_new_result_dir(self, new_result_dir):
        self.result_dir = self.access_directory(new_result_dir)
from src.base.base_libs import *

##############################################################################
############################################
# "Experimental" classes/functions that are supposed not to be discarded in later fixes
##############################################################################
############################################
```

```
class ExperimentalResFracFiles(object):
    def __init__(self, file_dir):
        super(ExperimentalResFracFiles, self).__init__()
        self.file_dir = file_dir

    @staticmethod
    def remove_new_line_char(line):
        if line == '\n':
            return None
        elif '\n' in line:
            return line.split(sep='\n')[0]
        else:
            return line

    def read(self):
        file_lines = list()
        with open(self.file_dir, mode='r') as f:
            file_lines = f.readlines()
        for (_, line) in enumerate(file_lines):
            if self.remove_new_line_char(line) == 'BrooksCorey':
                Swr_line = file_lines[_+2]
                Sor_line = file_lines[_+3]
                Sgr_line = file_lines[_+4]
                Swr_line = self.remove_new_line_char(Swr_line)
                Sor_line = self.remove_new_line_char(Sor_line)
                Sgr_line = self.remove_new_line_char(Sgr_line)
                Swr = float(Swr_line.split(sep='\t')[0])
                Sor = float(Sor_line.split(sep='\t')[0])
```

```
            Sgr = float(Sgr_line.split(sep='\t')[0])
        elif self.remove_new_line_char(line) ==
'relativefracturetoughnesspersqrtfracturelengthscale':
            Kic_line = self.remove_new_line_char(file_lines[_+6])
            Kic = float(Kic_line)
        else:
            pass
    return Swr, Sor, Sgr, Kic
from src.dir.dir import *


class DataFormat(object):
    def __init__(self):
        super(DataFormat, self).__init__()
        self.data_header = None
        self.property_names = None
        self.units = None


class SimulationTrackDataFormat(object):
    def __init__(self):
        super(SimulationTrackDataFormat, self).__init__()
        self.unit_type = None
        self.init_res_pres = None
        self.data_header = None
        self.track_locs = None
        self.property_names = None
        self.units = None
```

```
class DailyProductionDataFormat(object):
    def __init__(self):
        super(DailyProductionDataFormat, self).__init__()
        self.time_and_locations = None
        self.property_names = None
        self.units = None



class DataReader(object):
    def __init__(self, data_dir: RawResultDirectory, data_format: DataFormat):
        self.data_dir = data_dir
        self.data_format = data_format


    def read_data(self):
        data = list()
        with open(self.data_dir.result_dir, 'r') as file:
            file_header = csv.reader(file)
            for i, file_row in enumerate(file_header):
                if i == 0:
                    self.data_format.data_header = file_row
                elif i == 1:
                    self.data_format.property_names = file_row
                elif i == 2:
                    self.data_format.units = file_row
                else:
                    data.append(file_row)
            file.close()
```

```
        return data, self.data_format



class SimulationTrackDataReader(object):
    def __init__(self, data_dir: RawResultDirectory, data_format: SimulationTrackDataFormat):
        self.data_dir = data_dir
        self.data_format = data_format


    def read_data(self):
        data = list()
        with open(self.data_dir.result_dir, 'r') as file:
            file_header = csv.reader(file)
            for i, file_row in enumerate(file_header):
                if i == 0:
                    self.data_format.unit_type = extract_unit_type(file_row)
                    self.data_format.init_res_pres = extract_init_res_pres(file_row)
                elif i == 1:
                    self.data_format.track_locs = file_row
                elif i == 2:
                    self.data_format.property_names = file_row
                elif i == 3:
                    self.data_format.units = file_row
                else:
                    data.append(file_row)
            file.close()
        return data, self.data_format
```

```
class DailyProductionDataReader(object):
    def __init__(self, data_dir: RawResultDirectory, data_format: DailyProductionDataFormat):
        self.data_dir = data_dir
        self.data_format = data_format


    def read_data(self):
        data = list()
        with open(self.data_dir.result_dir, 'r') as file:
            file_header = csv.reader(file)
            for i, file_row in enumerate(file_header):
                if i == 0:
                    self.data_format.time_and_locations = file_row
                elif i == 1:
                    self.data_format.property_names = extract_property_names(file_row)
                    self.data_format.units = extract_units(file_row)
                else:
                    data.append(file_row)
            file.close()
        return data, self.data_format



class DataManagement(object):
    def __init__(self, simulation_dir):
        super(DataManagement, self).__init__()
        self.simulation_dir = simulation_dir


    def transfer_data(self):
        # TODO: ? Implement copy/move from the simulation directory to another preferable
directory
```

```
        pass


def search_properties(file_data_format: DataFormat, searched_property_names):
    searched_idx = list()
    for i, property_name in enumerate(file_data_format.property_names):
        if property_name in searched_property_names:
            searched_idx.append(i)
    if len(searched_idx) == 0:
        return None
    else:
        return searched_idx


def extract_unit_type(file_row):
    unit_type = file_row[0]
    return unit_type


def extract_init_res_pres(file_row):
    init_res_pres = float(file_row[-1])
    return init_res_pres


def extract_units(file_row):
    units = list()
    units.append(file_row[0])
    for (_, data_str) in enumerate(file_row[1:]):
```

```
        data = data_str.split(sep=' ')
        if data[0] in ['WHP', 'BHP']:
            units.append(data[-1])
        else:
            units.append(data[0])
    return units


def extract_property_names(file_row):
    property_names = list()
    property_names.append('Time elapsed')
    for (_, data_str) in enumerate(file_row[1:]):
        data = data_str.split(sep=' ')
        if data[0] in ['WHP', 'BHP']:
            property_names.append(data[-1])
        else:
            property_name = ''
            for __ in data[1:]:
                property_name += __
                property_name += '_'
            property_names.append(property_name)
    return property_names


def extract_time_step(file_name: str, prefix: str, suffix: str):
    time_step = file_name.split(sep=prefix)[-1]
    time_step = time_step.split(sep=suffix)[0]
    return int(time_step)
```

```
from src.base.base_libs import *


class Sample(object):
    def __init__(self, frac_geometry, frac_aperture, prop_vol_frac):
        super(Sample, self).__init__()
        self.fracture_geometry = frac_geometry
        self.fracture_aperture = frac_aperture
        self.proppant_volume_fraction = prop_vol_frac
        self.sampled_dimension = None

    def set_sampled_dimensions(self, sampled_dim: str):
        self.sampled_dimension = sampled_dim

    def validate_inputs(self):
        property_data = self.fracture_geometry.property_data
        if len(property_data) == 0:
            return False
        else:
            return True

    def sample(self):
        """
        Sample microchip signal data based on fracture aperture prop volume fraction
        Scheme to sample per fracture grid:
        1. Get the corresponding prop volume fraction from prop_vol_frac
        2. Larger prop volume fraction = more microchips (TODO: TBD about correlation)
        3. For one microchip, sample its location within the fracture grid
```

```
    using the fracture element's boundary coordinates in frac_geometry
    4. All microchips within a fracture grid reports similar fracture
    aperture using data from frac_aperture
    :return samples for all fracture grids
    """
    sample_data = list()
    if self.validate_inputs() is False:
        return np.array([], dtype=np.float32)
    else:
        for i in range(self.fracture_geometry.element_numbers):
            frac_elem = self.fracture_geometry.fracture_elements[i]
            frac_aperture = float(self.fracture_aperture.property_data[i, 0])
            prop_vol_frac_data = float(self.proppant_volume_fraction.property_data[i, 0])
            n = number_of_micro_chips(prop_vol_frac_data=prop_vol_frac_data)
            sample_data_i = list()
            for i_n in range(n):
                i_loc = random_location(frac_elem=frac_elem)
                i_frac_aperture = [frac_aperture]
                sample_data_i.append(i_loc+i_frac_aperture)
            sample_data += sample_data_i
        sample_data = np.array(sample_data)
        return sample_data


def sample_inplace(self, sample_dir: str):
    """
    Similar functionality to self.sample and include saving the sample
    """
    sample_data = list()
```

```
    if self.validate_inputs() is False:
        return np.array([], dtype=np.float32)
    else:
        for i in range(self.fracture_geometry.element_numbers):
            frac_elem = self.fracture_geometry.fracture_elements[i]
            frac_aperture = float(self.fracture_aperture.property_data[i, 0])
            prop_vol_frac_data = float(self.proppant_volume_fraction.property_data[i, 0])
            n = number_of_micro_chips_inplace(prop_vol_frac_data=prop_vol_frac_data)
            sample_data_i = list()
            for i_n in range(n):
                frac_elem.set_oriented_box()
                frac_elem.elem_box.compute_corners()
                i_loc = frac_elem.elem_box.sample()
                i_frac_aperture = [frac_aperture]
                sample_data_i.append(i_loc+i_frac_aperture)
            sample_data += sample_data_i
        sample_data = np.array(sample_data)
        sample_data_file = open(sample_dir, 'wb')
        pickle.dump(sample_data, sample_data_file)
        return sample_data


def sample_profile_along_height(self, height_resolution: np.ndarray):
    sample = self.sample()
    profile = np.zeros([height_resolution.shape[0]-1, 3])
    for i in range(profile.shape[0]):
        avg_aperture = list()
        lower_height = height_resolution[i]
        upper_height = height_resolution[i+1]
```

```
        if sample.size == 0:
            profile = np.ones([height_resolution.shape[0]-1, 3]) * np.nan
        else:
            for s in sample:
                if lower_height <= s[2] < upper_height:
                    avg_aperture.append(s[-1])
                else:
                    pass
            if len(avg_aperture) == 0:
                avg_aperture = 0.0
            else:
                avg_aperture = sum(avg_aperture)/len(avg_aperture)
            profile[i, 0] = lower_height
            profile[i, 1] = upper_height
            profile[i, 2] = avg_aperture
    return profile


def sample_profile_along_half_length(self, half_length_resolution: np.ndarray):
    sample = self.sample()
    profile = np.zeros([half_length_resolution.shape[0]-1, 3])
    for i in range(profile.shape[0]):
        avg_aperture = list()
        lower_hf = half_length_resolution[i]
        upper_hf = half_length_resolution[i+1]
        if sample.size == 0:
            profile = np.ones([half_length_resolution.shape[0]-1, 3]) * np.nan
        else:
            for s in sample:
```

```
                if lower_hf <= s[0] < upper_hf:
                    avg_aperture.append(s[-1])
                else:
                    pass
            if len(avg_aperture) == 0:
                avg_aperture = 0.0
            else:
                avg_aperture = sum(avg_aperture)/len(avg_aperture)
            profile[i, 0] = lower_hf
            profile[i, 1] = upper_hf
            profile[i, 2] = avg_aperture
        return profile


def number_of_micro_chips(prop_vol_frac_data: float):
    # Sample number of microchips as constant (experimental)
    micro_chips = 5
    return micro_chips


def number_of_micro_chips_inplace(prop_vol_frac_data: float):
    # Sample number of microchips proportional to prop-pant volume fraction
    min_vol_frac = 0.
    max_vol_frac = 1.
    micro_chips = (prop_vol_frac_data - min_vol_frac) / (max_vol_frac - min_vol_frac) * 100
    return math.floor(micro_chips)
```

```
def random_location(frac_elem):
    loc = [0, 0, 0]
    loc[0] = np.random.uniform(frac_elem.elem_corners[0, 0], frac_elem.elem_corners[0, -1])
    loc[1] = np.random.uniform(frac_elem.elem_corners[1, 0], frac_elem.elem_corners[1, -1])
    loc[-1] = np.random.uniform(frac_elem.elem_corners[-1, 0], frac_elem.elem_corners[-1, -1])
    return loc



def plot_sample_profile(sample_profile: np.ndarray):
    plt.figure()
    for i_profile in sample_profile:
        plt.plot(i_profile[: -1], [i_profile[-1], i_profile[-1]], 'k')
    plt.show()
import sys
import os
import shutil

import math
import csv
import warnings

from typing import Union, List, Optional, Tuple, Dict
from itertools import permutations, combinations, product
from functools import partial, partialmethod

from copy import deepcopy
from enum import Enum
```

```
import numpy as np
import pandas as pd

import scipy
import pyDOE

import hyperopt
from hyperopt import fmin, hp, tpe, Trials, space_eval
from hyperopt import STATUS_OK, STATUS_NEW, STATUS_RUNNING, STATUS_FAIL

from hyperopt.pyll import scope as hyperopt_scope
from hyperopt.pyll.stochastic import sample as hyperopt_sample
from base import *



##############################################################################################
#############################################
# Components to perform fracture calibration and history match (Experimental)
##############################################################################################
#############################################



class SurrogateWrapper(object):
    def __init__(self, surrogate, pred_time_steps: List[float]):
        """
        Wrapper to call surrogate
        :param surrogate:
        :param pred_time_steps:
        """
```

```
        super(SurrogateWrapper).__init__()
        self.surrogate = surrogate
        self.pred_time_steps = pred_time_steps
        self.observation_indexes = range(len(pred_time_steps))

    def compute(self, input_params: dict, fixed_params: dict):
        input_params_ = np.array([_ for _ in input_params.values()])
        fixed_params_ = np.array([_ for _ in fixed_params.values()])
        predictions = np.zeros([len(self.pred_time_steps), ])
        for _ in self.observation_indexes:
            time_step_ = self.pred_time_steps[_]
            params_ = np.concatenate([time_step_, input_params_, fixed_params_])
            predictions[_] = self.surrogate(params_)
        return predictions



class SingleObjective(object):
    def __init__(self, surr_wrapper: SurrogateWrapper):
        """
        Objective value for a single response variable
        :param surr_func_wrapper: the wrapper that calls the surrogate function (i.e., proxy model)
        """
        super(SingleObjective, self).__init__()
        self.surr_wrapper = surr_wrapper

    def compute(self, input_params: dict, fixed_params: dict, observation_indexes: List[int]):
        surr_pred = np.zeros(shape=[len(observation_indexes), ])
        surr_pred_raw = self.surr_wrapper.compute(input_params, fixed_params)
```

```
        for _ in range(len(observation_indexes)):
            surr_pred[_] = surr_pred_raw[observation_indexes[_]]
        return surr_pred


class MultiObjective(object):
    def __init__(self, objective_list: List[SingleObjective], num_objectives: int, observation_indexes: List[int]):
        """
        Objective value for multiple response variables
        :param surr_func: List of SingleObjectiveSurrogate that calls all surrogate functions corresponding to all
        response variables
        :param num_objectives: Total number of response variables
        :param num_observations: Total number of observation points for all response variables (i.e., each response
        variable has equal number of observation points)
        """
        super(MultiObjective, self).__init__()
        self.objective_list = objective_list
        self.num_objectives = num_objectives
        self.observation_indexes = observation_indexes

    def compute(self, input_params: dict, fixed_params: dict):
        pred = np.zeros(shape=[self.num_objectives, len(self.observation_indexes)])
        for i in range(self.num_objectives):
            surr_pred = self.objective_list[i].compute(input_params, fixed_params,
                                                       observation_indexes=self.observation_indexes)
            pred[i, :] = surr_pred
```

```
        return pred



class ObjectiveFunction(object):
    def __init__(self, data: np.ndarray, pred: np.ndarray):
        """
        Objective function for single/multiple response variable(s)
        :param data:
        :param pred:
        """
        super(ObjectiveFunction, self).__init__()
        self.data = data
        self.pred = pred
        self.mean = None

    def set_mean(self, mean: np.ndarray):
        self.mean = mean

    def compute(self):
        objective = np.power((self.pred - self.data) / self.mean, 2)
        return np.sum(objective, axis=0)



def objective_function(input_params, fixed_params, num_objectives: int, observation_indexes: List[int],
                       objective_list: List[SingleObjective], data, mean):
    multi_object_surrogate = MultiObjective(objective_list=objective_list, num_objectives=num_objectives,
                                            observation_indexes=observation_indexes)
```

```
    pred = multi_object_surrogate.compute(input_params=input_params, fixed_params=fixed_params)
    objective_func = ObjectiveFunction(data=data, pred=pred)
    objective_func.set_mean(mean=mean)
    return objective_func.compute()


def partial_objective_function(fixed_params: dict, num_objectives: int, observation_indexes: List[int],
                               objective_list: List[SingleObjective], data, mean):
    return partial(objective_function, fixed_params=fixed_params, num_objectives=num_objectives,
                   observation_indexes=observation_indexes, objective_list=objective_list, data=data, mean=mean)
from workflow.fracture_profile import *


if __name__ == "__main__":
    data_main_dir = r'C:\Users\v183p176\Desktop'
    raw_result_dir = RawResultDirectory(data_main_dir=data_main_dir)
    height_resolution = [-600, 600, 51]

    simulation_dir = os.path.join(data_main_dir, 'doe_simulation_2 SOP')
    frac_profile_assembler = FractureProfileAssembly(resolution=height_resolution)
    frac_profile_assembler.set_root_directories(simulation_dir=simulation_dir)
    frac_profile_df = frac_profile_assembler.asemble_fracture_profile(save_fracture_profile=True)
    frac_profile_df.to_csv('Fracture_profile.csv')
from src.dir.dir import *
```

```
if __name__ == "__main__":
    simulation_dir = r'F:\PhD work (Spring 2024)\ResFrac simulations\workflows\Base HM\simulations'
    new_dir = r'F:\PhD work (Fall 2024)'
    data_dir_obj = DataDirectory(data_main_dir=simulation_dir)
    data_dir_obj.move(new_dir=new_dir)
from workflow.reservoir_response import *



if __name__ == "__main__":
    data_main_dir = r'F:\PhD work (Spring 2024)\ResFrac simulations\workflows\Base HM\simulations'
    raw_result_dir = RawResultDirectory(data_main_dir=data_main_dir)


    reservoir_response_assembler = ReservoirResponseAssembly()
    reservoir_response_assembler.set_root_directories(simulation_dir=data_main_dir,
                                        simulation_file_name='doe_simulation_2 SOP',
                                        sim_track_file_name='sim_track_doe_simulation_2 SOP.csv',
                                        response_var_names=['Oil prod rate'])
    res_response_df, time_steps_df = reservoir_response_assembler.assemble_reservoir_response()
    res_response_df.to_csv('Reservoir_response.csv')
    time_steps_df.to_csv('Time_steps.csv')
from simulator.base import utils
from simulator.base import regex_templates
from simulator.base import parse


from DoE.doe.doe_v1 import *
```

```
class ResFracFileParseTest(object):
    def __init__(self):
        super(ResFracFileParseTest).__init__()
        simulation_dir = r'C:\Users\v183p176\Desktop'
        simulation_case_name = 'doe_simulation_2 SOP'
        settings_file_name = 'settings_' + simulation_case_name + '.txt'
        input_file_name = 'input_' + simulation_case_name + '.txt'
        simulation_case_dir = os.path.join(simulation_dir, simulation_case_name)
        self.settings_file_dir = os.path.join(simulation_case_dir, settings_file_name)
        self.input_file_dir = os.path.join(simulation_case_dir, input_file_name)

    def test_rel_perm_parse(self):
        all_entries = utils.parse_file(file_name=self.settings_file_dir)
        entry = all_entries[1]
        var_name = entry['Variable name:'][0]
        var_length = entry['Length:'][0]
        var_raw_data = entry['Value(s):']
        regex_temp = regex_templates.RelPermRegex(raw_str=var_raw_data)
        var_data = regex_temp.extract(pattern=None)
        return var_name, var_length, var_data

    def test_well_control_parse(self):
        all_entries = utils.parse_file(file_name=self.settings_file_dir)
        entry = all_entries[-1]
        var_name = entry['Variable name:'][0]
        var_length = entry['Length:'][0]
```

```
    var_raw_data = entry['Value(s):']
    regex_temp = regex_templates.WellRegex(raw_str=var_raw_data)
    var_data = regex_temp.extract(pattern=None)
    return var_name, var_length, var_data

def test_settings_file_parse(self):
    all_entries = utils.parse_file(file_name=self.settings_file_dir)
    all_parsed_entries = list()
    for (_, entry) in enumerate(all_entries):
        parsed_entry = parse.parse_entry(entry=entry)
        all_parsed_entries.append(parsed_entry)
    for (_e, entry) in enumerate(all_parsed_entries):
        if entry.variable_name in simulation_helpers.irregular_variable_names:
            print('Entry: ', _e, ' with var_name: ', entry.variable_name,
                  ' and length: ', entry.length)
            print(entry.value_struct.value_struct)
    return all_parsed_entries

def test_input_file_parse(self):
    all_entries = utils.parse_file(file_name=self.input_file_dir)
    all_parsed_entries = list()
    for (_, entry) in enumerate(all_entries):
        parsed_entry = parse.parse_entry(entry=entry)
        all_parsed_entries.append(parsed_entry)
    for (_e, entry) in enumerate(all_parsed_entries):
        if entry.variable_name in simulation_helpers.irregular_variable_names:
            print('Entry: ', _e, ' with var_name: ', entry.variable_name,
                  ' and length: ', entry.length)
```

```
        print(entry.value_struct.value_struct)
    return all_parsed_entries


def test_doe_settings_file(self):
    doe_dir = r"E:\Vuong's ResFrac"
    num_cases, last_case = 100, 0
    base_entries = self.test_settings_file_parse()
    doe_params = ['S_wr', 'S_or', 'S_gr', 'relative_frac_toughness']
    doe_entry_var_names = ['matrixcurvesets',
                           'matrixcurvesets',
                           'matrixcurvesets',
                           'relativefracturetoughnesspersqrtfracturelengthscale']
    doe_distributions = [distributions.norm(loc=0.2, scale=0.1),
                         distributions.norm(loc=0.2, scale=0.1),
                         distributions.norm(loc=0.03, scale=0.001),
                         distributions.uniform(loc=0.0, scale=0.5)]
    write_locs = [[[0, 0]], [[1, 0]], [[2, 0]], -1]
    #
    doe_assembler = DesignOfExperimentsAssembly(base_entries=base_entries)
    doe_assembler.doe_batch = 0
    doe_assembler.doe_dir = doe_dir
    #
    doe_assembler.reset_all_entries()
    doe_assembler.set_doe_params(doe_params)
    doe_assembler.set_doe_entry_var_names(doe_entry_var_names)
    doe_assembler.set_doe_distributions(doe_distributions)
    doe_assembler.set_write_locs(write_locs)
    doe_object = doe_assembler.generate_doe_object(design='lhs', num_cases=num_cases)
```

```
        doe_df = doe_assembler.write_doe_entries(doe_object, last_case=last_case)
        return doe_df


if __name__ == '__main__':
    #
    all_settings_entries = ResFracFileParseTest().test_settings_file_parse()
    all_input_entries = ResFracFileParseTest().test_input_file_parse()
    doe_df = ResFracFileParseTest().test_doe_settings_file()
    #
    sim_file = simulation_files.SimulationFile(entries=all_settings_entries,
                                               file_type='settings')
    sim_file.write_file(file_name='../Proxy_cases/Input_files/settings.txt')
    sim_file = simulation_files.SimulationFile(entries=all_input_entries,
                                               file_type='input')
    sim_file.write_file(file_name='../Proxy_cases/Input_files/input.txt')
from workflow.surrogate import *


class SurrogateTest(object):
    def __init__(self):
        super(SurrogateTest).__init__()
        simulation_dir = r"E:\Vuong's ResFrac\DoE_cases\workflows\Proxy_cases_batch_0\simulations"
        simulation_case_name = 'doe_case_0'
        settings_file_name = 'settings_' + simulation_case_name + '.txt'
        input_file_name = 'input_' + simulation_case_name + '.txt'

        self.simulation_dir = simulation_dir
```

```
        self.simulation_case_name = simulation_case_name

        simulation_case_dir = os.path.join(simulation_dir, simulation_case_name)
        self.settings_file_dir = os.path.join(simulation_case_dir, settings_file_name)
        self.input_file_dir = os.path.join(simulation_case_dir, input_file_name)

    def test_fracture_profile(self):
        raw_result_dir = RawResultDirectory(data_main_dir=self.simulation_dir)
        height_resolution = [-600, 600, 51]

        simulation_dir = os.path.join(self.simulation_dir, self.simulation_case_name)
        frac_profile_assembler = FractureProfileAssembly(resolution=height_resolution)
        frac_profile_assembler.set_root_directories(simulation_dir=simulation_dir)
        frac_profile_df = frac_profile_assembler.asemble_fracture_profile()
        frac_profile_df.to_csv('Fracture_profile.csv')
        return frac_profile_assembler

    def test_reservoir_response(self):
        raw_result_dir = RawResultDirectory(data_main_dir=self.simulation_dir)
        reservoir_response_assembler = ReservoirResponseAssembly()
        reservoir_response_assembler.set_root_directories(simulation_dir=self.simulation_dir,
                                                          simulation_file_name=self.simulation_case_name,
                                                          sim_track_file_name='sim_track_doe_case_0.csv',
                                                          response_var_names=['BHP', 'Oil prod rate'])
        res_response_df, time_steps_df = reservoir_response_assembler.assemble_reservoir_response()
        res_response_df.to_csv('Reservoir_response.csv')
        time_steps_df.to_csv('Time_steps.csv')
        return reservoir_response_assembler
```

```
    def test_surrogate(self):
        surrogate_dir_assembler = SurrogateDirectory(result_dir=self.simulation_dir)
        surrogate_dir_assembler.num_cases = 1
        surrogate_dir_assembler.init_fracture_profile(fracture_profile_resolution=[-600, 600, 51])
        surrogate_dir_assembler.init_reservoir_response(reservoir_response_var_names=['BHP',
'Oil prod rate'])
        surrogate_dir_assembler.experimental_doe_params = ['S_wr', 'S_or', 'S_gr',
'relative_frac_toughness']
        surrogate_dir_assembler.init_experimental_doe_data()
        surrogate_df =
surrogate_dir_assembler.assemble_surrogate_directory('Test_surrogate_0.csv')
        return surrogate_df


if __name__ == "__main__":
    test_surrogate_df = SurrogateTest().test_surrogate()
from workflow.surrogate import *


if __name__ == "__main__":
    result_dir = r"E:\Vuong's ResFrac\DoE_cases\workflows\Proxy_cases\simulations"
    surrogate_dir = r"E:\Vuong's ResFrac\surrogate_data.csv"
    fracture_profile_resolution = [-600, 600, 51]
    surrogate_manager = SurrogateDirectory(result_dir=result_dir)
    surrogate_manager.experimental_doe_params = ['S_wr', 'S_or', 'S_gr',
'relative_frac_toughness']

surrogate_manager.init_fracture_profile(fracture_profile_resolution=fracture_profile_resolution)
```

```
    surrogate_manager.init_reservoir_response(reservoir_response_var_names=['BHP', 'Oil prod rate'])
    surrogate_manager.init_experimental_doe_data()
    surrogate_df = surrogate_manager.assemble_surrogate_directory(surrogate_dir=surrogate_dir)
from src.data.frac_geometry.frac_geometry import *
from src.data.properties.total_aperture import *
from src.data.properties.proppant_volume import *
from src.utils.sample_by_distribution import *



if __name__ == '__main__':
    data_main_dir = r'F:\PhD work (Spring 2024)\ResFrac simulations\workflows\Base HM\simulations\doe_simulation_2 SOP\Results'
    raw_result_dir = RawResultDirectory(data_main_dir=data_main_dir)


    raw_result_file = r'Raw_Res' + '/frac_elms_512.csv'
    raw_result_dir.set_new_result_dir(raw_result_file)


    data_format = DataFormat()
    data_reader = DataReader(data_dir=raw_result_dir, data_format=data_format)
    file_data, file_data_format = data_reader.read_data()


    frac_geometry = FractureGeometry(raw_result_dir=raw_result_dir)
    frac_geometry.set_raw_result_file(raw_result_file=raw_result_file)


    frac_geometry.set_fracture_geometry_data(file_data=file_data,
                                             file_data_format=file_data_format)
    frac_geometry.set_fracture_surface_data()
```

```
frac_aperture = Aperture(raw_result_dir=raw_result_dir)
frac_aperture.set_raw_result_file(raw_result_file=raw_result_file)

frac_aperture.set_aperture_data(file_data=file_data,
                                file_data_format=file_data_format)

prop_vol_frac = PropVolFrac(raw_result_dir=raw_result_dir)
prop_vol_frac.set_raw_result_file(raw_result_file=raw_result_file)

prop_vol_frac.set_vol_frac_data(file_data=file_data,
                                file_data_format=file_data_format)

sample_by_dist = Sample(frac_geometry=frac_geometry, frac_aperture=frac_aperture,
                        prop_vol_frac=prop_vol_frac)
height_resolution = np.linspace(-600, 600, 51)
height_profile = sample_by_dist.sample_profile_along_half_length(height_resolution)
plot_sample_profile(sample_profile=height_profile)

raw_result_file = 'sim_track_doe_simulation_2 SOP.csv'
raw_result_dir.set_new_result_dir(raw_result_file)
data_format = SimulationTrackDataFormat()
data_reader = SimulationTrackDataReader(data_dir=raw_result_dir, data_format=data_format)
sim_track_file_data, sim_track_file_data_format = data_reader.read_data()

raw_result_file = 'daily_prod_doe_simulation_2 SOP.csv'
raw_result_dir.set_new_result_dir(raw_result_file)
data_format = DailyProductionDataFormat()
data_reader = DailyProductionDataReader(data_dir=raw_result_dir, data_format=data_format)
```

```
    daily_prod_file_data, daily_prod_file_data_format = data_reader.read_data()
from workflow.surrogate import *
from workflow.objective_function import *

from proxy.proxy import *
from proxy.gb_proxy import *
from proxy.xgb_proxy import *

from proxy.proxy_opt import *



################################################################################
##########################################
# Workflow to perform fracture calibration (TBD)
################################################################################
##########################################



class FractureCalibrationAssembly(object):
    def __init__(self, proxy, non_cal_params: dict, objectives: dict):
        super(FractureCalibrationAssembly, self).__init__()
        self.proxy = proxy
        self.non_cal_params = non_cal_params
        self.objectives = objectives
        #
        self.proxy_pred_method: str = ""
        self.input_signature = None

    def set_proxy_prediction_method(self, proxy_pred_method: str):
```

```
        self.proxy_pred_method = proxy_pred_method

    def set_input_signature(self, input_signature):
        self.input_signature = input_signature

    def set_objective_function(self, cal_params: dict):
        inputs: dict = self.non_cal_params
        inputs.update(cal_params)
        obj_func = ObjectiveFunction(inputs=inputs, objectives=self.objectives,
                                     proxy=self.proxy)
        try:
            assert getattr(self.proxy, self.proxy_pred_method) is not None
            return obj_func.compute(proxy_pred_method=self.proxy_pred_method)
        except AssertionError:
            warnings.warn("Proxy prediction method is not found. Fatal proxy object.")
            return None

    def assemble_fracture_calibration(self, opt_space: dict):
        proxy_optimizer = ProxyOptimization(proxy_object=self.proxy)
        proxy_optimizer.init_optimizer(opt_space=opt_space)
        opt_instance = proxy_optimizer.exec_optimizer(
            opt_func_attr=self.proxy_pred_method)
        return space_eval(opt_instance)
from src.base.base_libs import *
from proxy.xgb_proxy import *
from proxy.proxy_opt import *
```

```
#############################################################################################
################

########################## Calibrate continually
################################################

################ Proxy re-training when sensor data is received continuously
#################

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


class ContinualCalibration(object):
    def __init__(self, new_X, new_Y, trained_xgb_regressor: QuantileXGBRegressor):
        super(ContinualCalibration, self).__init__()
        self.trained_xgb_regressor = trained_xgb_regressor
        self.new_X = new_X
        self.new_Y = new_Y


    def fit(self, opt_quantile_hyperparams):
        quantile_xgb_model,                                    _                                    =
self.trained_xgb_regressor.__fit__(params=opt_quantile_hyperparams)
        return quantile_xgb_model


    def train(self, opt_quantile_hyperparams):
        hyperparams_ = opt_quantile_hyperparams
        if 'quantile_alpha' not in hyperparams_.keys():
            hyperparams_['quantile_alpha'] = np.array([0.1, 0.5, 0.9])
        else:
            pass
        hyperparams_['objective'] = 'reg:quantileerror'
```

```
        hyperparams_['tree_method'] = 'hist'
        #
        quantile_xgb_model = self.fit(hyperparams_)
        #
        xgb_data = self.trained_xgb_regressor.xgb_data
        org_x = xgb_data.df[xgb_data.x_cols].to_numpy()
        org_y = xgb_data.df[xgb_data.y_cols].to_numpy()
        it = 0
        while it < self.new_X.shape[0]:
            new_x_ = self.new_X[it:it + 1, :]
            new_y_ = quantile_xgb_model.inplace_predict(new_x_)
        if mean_squared_error(new_y_[:, 1], self.new_Y[it:it + 1]) >= 0.01:
            err_str = 'MSE'
            thres = 0.01
            warnings.warn(
                'continual prediction at time interval {} exceeds threshold {} at {}.'.format(it, err_str,
thres))
        else:
            pass
        cont_x = self.new_X[:it + 1, :]
        cont_y = self.new_Y[:it + 1].reshape([it + 1, 1])
        new_x_train = np.concatenate((org_x, cont_x), axis=0)
        new_y_train = np.concatenate((org_y, cont_y), axis=0)
        #
        quantile_xgb_model: xgb.Booster = xgb.train(
            hyperparams_, xgb.QuantileDMatrix(new_x_train, new_y_train),
            num_boost_round=32)
        it += 1
        pred_new_y = quantile_xgb_model.inplace_predict(self.new_X)
```

```
        return pred_new_y
from src.base.base_libs import *


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

#################################### Fracture geometry (Unsupervised algorithms) ########################################

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


class SensorDataProjection(object):
    """
    Class to determine planar surface for a set of sensor data via SVD decomposition
    """

    def __init__(self, sensor_data: np.ndarray):
        super(SensorDataProjection, self).__init__()
        self.sensor_data: np.ndarray = sensor_data

    def compute_centroid(self):
        return self.sensor_data.mean(axis=0)

    def compute_svd(self):
        centroid = self.sensor_data.mean(axis=0)
        U, S, Vh = np.linalg.svd(self.sensor_data - centroid)
        return U, S, Vh, centroid
```

```
    def project(self):
        _, _, Vh, centroid = self.compute_svd()
        A, B, C = Vh[-1] / np.linalg.norm(Vh[-1])
        D = -np.dot(Vh[-1] / np.linalg.norm(Vh[-1]), centroid)
        return A, B, C, D

    def compute_planar_dimensions(self):
        pass

    def plot(self, fig_dir):
        data_projection = np.zeros(shape=self.sensor_data.shape)
        data_aperture = np.zeros(shape=[self.sensor_data.shape[0], 1])
        A, B, C, D = self.project()
        normal = np.array([A, B, C])
        fixed_projection = np.array([0, 0, -D / C])
        for _ in range(self.sensor_data.shape[0]):
            projection_vector = self.sensor_data[_, :] - fixed_projection
            projection_vector = np.dot(projection_vector, normal) / np.dot(normal, normal) * normal
            data_projection[_, :] = self.sensor_data[_, :] - projection_vector
            #
            d = np.absolute(self.sensor_data[_, 0] * A + self.sensor_data[_, 1] * B + self.sensor_data[_,
-1] * C + D)
            d = d / np.linalg.norm(normal)
            data_aperture[_, 0] = d
            data_projection = pd.DataFrame(data=data_projection, columns=['x', 'y', 'z'])
            fig = px.scatter_3d(data_frame=data_projection, x='x', y='y', z='z')
            fig.add_scatter3d(x=self.sensor_data[:, 0], y=self.sensor_data[:, 1],
                              z=self.sensor_data[:, -1], mode='markers')
            fig.update_layout(scene=dict(
```

```
            aspectmode='manual',
            aspectratio=dict(x=10, y=1, z=5)))
        fig.update_traces(marker_size=1)
        fig.write_html(fig_dir)
    return data_projection, data_aperture
from src.dir.dir import *
from src.utils import read_data, sample_by_distribution

from src.data.frac_geometry import frac_geometry
from src.data.properties import properties, proppant_volume, total_aperture
from src.data.time_steps import *

from simulator.base import utils
from simulator.base import regex_templates
from simulator.base import parse

from DoE.doe.doe_v1 import *



################################################################################
##########################################
# "Miscellaneous" workflows to assemble fracture profile (TBD)
################################################################################
##########################################



class FractureProfileAssembly(object):
    def __init__(self, resolution: List):
        super(FractureProfileAssembly).__init__()
```

```
        self.resolution = resolution
        self.simulation_results_dir = None
        #
        self.settings_file_dir = None
        self.input_file_dir = None

    def set_root_directories(self, simulation_dir):
        simulation_results_dir = os.path.join(simulation_dir, 'Results')
        self.simulation_results_dir = RawResultDirectory(data_main_dir=simulation_results_dir)

    def wrap_fracture_profile(self, frac_elms_file_names, save_fracture_profile=False):
        fracture_profile = dict()
        fracture_profile_time = dict()
        for frac_elms_file_name in frac_elms_file_names:
            frac_elms_file_dir = r'Raw_Res' + '/' + frac_elms_file_name
            self.simulation_results_dir.set_new_result_dir(new_result_dir=frac_elms_file_dir)
            time_step = read_data.extract_time_step(file_name=frac_elms_file_name,
prefix='frac_elms_', suffix='.csv')
            # Read all raw data
            data_format = read_data.DataFormat()
            data_reader = read_data.DataReader(data_dir=self.simulation_results_dir,
data_format=data_format)
            file_data, file_data_format = data_reader.read_data()
            # Extract time from the file header ( in hours )
            print('Process fracture profile time step ', time_step)
            time = float(file_data_format.data_header[1])
            # Extract the fracture geometry
            frac_geometry_obj =
frac_geometry.FractureGeometry(raw_result_dir=self.simulation_results_dir)
```

```
        frac_geometry_obj.set_raw_result_file(raw_result_file=frac_elms_file_dir)
        frac_geometry_obj.set_fracture_geometry_data(file_data=file_data,
                                                     file_data_format=file_data_format)
        frac_geometry_obj.set_fracture_surface_data()
        frac_geometry_obj.plot_fracture(fracture_number=-1, fig_name='fracture_geometry_' +
                                                          str(time_step) + '.html')
        # Extract the total aperture
        frac_aperture_obj = total_aperture.Aperture(raw_result_dir=self.simulation_results_dir)
        frac_aperture_obj.set_raw_result_file(raw_result_file=frac_elms_file_dir)
        frac_aperture_obj.set_aperture_data(file_data=file_data,
                                            file_data_format=file_data_format)
        # Extract the prop volume fraction
        prop_vol_frac_obj                                                                    =
proppant_volume.PropVolFrac(raw_result_dir=self.simulation_results_dir)
        prop_vol_frac_obj.set_raw_result_file(raw_result_file=frac_elms_file_dir)
        prop_vol_frac_obj.set_vol_frac_data(file_data=file_data,
                                            file_data_format=file_data_format)
        # Perform the sampling to generate sensor data
        sample_by_dist = sample_by_distribution.Sample(frac_geometry=frac_geometry_obj,
                                                       frac_aperture=frac_aperture_obj,
                                                       prop_vol_frac=prop_vol_frac_obj)
        height_resolution = np.linspace(self.resolution[0], self.resolution[1], self.resolution[-1])
        height_profile = sample_by_dist.sample_profile_along_half_length(height_resolution)
        # Save the generated sensor data
        if save_fracture_profile:
            sample_dir = 'sensor_data_' + str(time_step) + '.pkl'
            sample_by_dist.sample_inplace(sample_dir=sample_dir)
        # Obtain the sampled sensor data per time step (x, y, z)
        if time_step not in fracture_profile.keys():
```

```
            fracture_profile[time_step] = height_profile
        if time_step not in fracture_profile_time.keys():
            fracture_profile_time[time_step] = time
    fracture_profile = {k: fracture_profile[k] for k in sorted(fracture_profile)}
    fracture_profile_time = {k: fracture_profile_time[k] for k in sorted(fracture_profile_time)}
    return fracture_profile, fracture_profile_time


def asemble_fracture_profile(self, save_fracture_profile=False):
    """
    Assemble fracture profile for all recorded time steps as .csv
    1. Call self.wrap_fracture_profile()
    Per time step:
        2. Extract fracture profile data  from fracture_profile
        3. Arrange fracture profile dimensions as follows:
        - Number of columns = 3 * fracture_profile.shape[0]
        - Columns are named 'x_1, x_2, .... x_fracture_profile.shape[0]+1' (similar for y & z)
        - Arrange the fracture profile data at ordered for x, y, z above
    :return: fracture_profile_df
    """
    frac_elms_file_names                                                    =
self.simulation_results_dir.find_file_include(accessed_dir='Raw_Res',
                                                  include_str='frac_elms')
    fracture_profile, fracture_profile_time = \
        self.wrap_fracture_profile(frac_elms_file_names=frac_elms_file_names,
save_fracture_profile=
                                   save_fracture_profile)
    fracture_profile_df = list()
    time_steps = list(fracture_profile.keys())
    for time_step in time_steps:
```

```
            fracture_profile_samples = fracture_profile[time_step].shape[0]
            fracture_profile_data = dict()
            fracture_profile_data['fracture_profile_time'] = fracture_profile_time[time_step]
            for _ in range(fracture_profile_samples):
                key_x = 'x_' + str(_)
                key_y = 'y_' + str(_)
                key_z = 'z_' + str(_)
                if key_x not in fracture_profile_data.keys():
                    fracture_profile_data[key_x] = fracture_profile[time_step][_, 0]
                if key_y not in fracture_profile_data.keys():
                    fracture_profile_data[key_y] = fracture_profile[time_step][_, 1]
                if key_z not in fracture_profile_data.keys():
                    fracture_profile_data[key_z] = fracture_profile[time_step][_, -1]
            fracture_profile_df.append(fracture_profile_data)
        fracture_profile_df = pd.DataFrame(data=fracture_profile_df)
        fracture_profile_df.index = time_steps
        return fracture_profile_df
from workflow.surrogate import *
from workflow.objective_function import *

from proxy.proxy import *
from proxy.gb_proxy import *
from proxy.xgb_proxy import *

from proxy.proxy_opt import *



################################################################################
##############################################
```

```
# Workflow to perform history matching
################################################################################
##########################################


class HistoryMatchAssembly(object):
    def __init__(self, proxy, non_cal_params: dict, objectives: dict):
        super(HistoryMatchAssembly, self).__init__()
        self.proxy = proxy
        self.non_cal_params = non_cal_params
        self.objectives = objectives
        #
        self.proxy_pred_method: str = ""
        self.input_signature = None

    def set_proxy_prediction_method(self, proxy_pred_method: str):
        self.proxy_pred_method = proxy_pred_method

    def set_input_signature(self, input_signature):
        self.input_signature = input_signature

    def set_objective_function(self, cal_params: dict):
        inputs: dict = self.non_cal_params
        inputs.update(cal_params)
        obj_func = ObjectiveFunction(inputs=inputs, objectives=self.objectives,
                                     proxy=self.proxy)
        try:
            assert getattr(self.proxy, self.proxy_pred_method) is not None
            return obj_func.compute(proxy_pred_method=self.proxy_pred_method)
```

```
        except AssertionError:
            warnings.warn("Proxy prediction method is not found. Fatal proxy object.")
            return None

    def assemble_fracture_calibration(self, opt_space: dict):
        proxy_optimizer = ProxyOptimization(proxy_object=self.proxy)
        proxy_optimizer.init_optimizer(opt_space=opt_space)
        opt_instance = proxy_optimizer.exec_optimizer(
            opt_func_attr=self.proxy_pred_method)
        return space_eval(opt_instance)
from src.base.base_libs import *



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

################ Objective function for fracture calibration and history matching
########################################

########################################## Single-objective, multi-point objective
########################################

############################### Refer to task.experimental for multi-objective function
###################################

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



class ObjectiveFunction(object):
    def __init__(self, inputs: dict, objectives: dict, proxy):
        """
        Objective function class for fracture calibration and history matching
        Class attributes
```

```
        inputs: inputs to compute the objective function, dict("name" : value)
        objectives: outputs to compute the objective function, dict("name", value)
        proxy: the proxy object to deploy the prediction, must have a prediction method (e.g, QuantileXGBRegressor)
        input_signature: the order of the input values to comply with the proxy object
        weights: the weights for the objectives (optional, default 1.)
    Class method:
        compute:
    """
    super(ObjectiveFunction, self).__init__()
    self.inputs = inputs
    self.objectives = objectives
    self.proxy = proxy
    #
    self.input_signature = None
    self.weights = np.array([1. for _ in range(len(objectives))], dtype=np.float32)

def set_input_signature(self, input_signature: str):
    self.input_signature = input_signature

def set_weights(self, weights):
    self.weights = weights

def compute(self, proxy_pred_method: str):
    inputs_ = {}
    for input_sig_ in self.input_signature:
        if input_sig_ not in inputs_.keys():
            inputs_[input_sig_] = self.inputs[input_sig_]
    inputs_ = np.array([_ for _ in inputs_.values()], dtype=np.float32)
```

```
        objectives_ = [_ for _ in self.objectives.values()]
        objectives_ = np.array(objectives_, dtype=np.float32)
        try:
            assert getattr(self.proxy, proxy_pred_method) is not None
            pred_objectives_ = getattr(self.proxy, proxy_pred_method)(inputs_)
            obj_f = np.power(objectives_ - pred_objectives_, 2) * self.weights
            return 1. / len(self.objectives) * obj_f.sum(axis=0)
        except AssertionError:
            warnings.warn("Proxy prediction method is not found. Fatal proxy object.")
            return None
import pandas as pd

from src.dir.dir import *
from src.utils import read_data, sample_by_distribution

from src.data.frac_geometry import frac_geometry
from src.data.properties import properties, proppant_volume, total_aperture
from src.data.time_steps import time_properties

from simulator.base import utils
from simulator.base import regex_templates
from simulator.base import parse

from DoE.doe.doe_v1 import *



################################################################################
##############################################
```

```
# "Miscellaneous" workflows to assemble reservoir response parameters (i.e, BHP, oil/gas rate) (TBD)
################################################################################################################################################


class ReservoirResponseAssembly(object):
    def __init__(self):
        super(ReservoirResponseAssembly, self).__init__()
        self.simulation_results_dir = None
        self.sim_track_file_name = None
        self.response_var_names = None

    def set_root_directories(self, simulation_dir, simulation_file_name, sim_track_file_name, response_var_names):
        simulation_results_dir = os.path.join(simulation_dir, simulation_file_name)
        simulation_results_dir = os.path.join(simulation_results_dir, 'Results')
        self.simulation_results_dir = RawResultDirectory(data_main_dir=simulation_results_dir)
        self.simulation_results_dir.set_new_result_dir(sim_track_file_name)
        self.response_var_names = response_var_names

    def wrap_reservoir_response(self):
        data_format = time_properties.SimulationTrackDataFormat()
        data_reader = time_properties.SimulationTrackDataReader(
            data_dir=self.simulation_results_dir, data_format=data_format)
        sim_track_file_data, sim_track_file_data_format = data_reader.read_data()
        return sim_track_file_data, sim_track_file_data_format

    def wrap_time_steps(self, time_steps_file_name='timesteps.csv'):
```

```
        time_steps_dir = os.path.join(self.simulation_results_dir.data_main_dir, time_steps_file_name)
        time_steps_df = pd.read_csv(time_steps_dir, header=None)
        time_steps_df.columns = ['time_steps', 'time_steps_sec', 'time_steps_str']
        return time_steps_df


    def assemble_reservoir_response(self):
        """
        Assemble reservoir response for all time steps as .csv
        1. Call self.wrap_reservoir_response()
        Per time step:
            2. Extract sim_track data for all recorded properties from sim_track_file_data
            3. Find location of response_var_names in sim_track_file_data_format
            4. Arrange corresponding value of a response variable name
        :return: reservoir_data_df
        """
        sim_track_file_data, sim_track_file_data_format = self.wrap_reservoir_response()
        reservoir_response_df = list()
        time_steps = len(sim_track_file_data)
        property_names = sim_track_file_data_format.property_names
        for _ in range(time_steps):
            reservoir_response_data = dict()
            for response_var_name in self.response_var_names:
                try:
                    assert response_var_name in property_names
                    response_var_index = property_names.index(response_var_name)
                    if 'reservoir_response_time' not in reservoir_response_data.keys():
                        reservoir_response_data['reservoir_response_time'] = sim_track_file_data[_][0]
                    if response_var_name not in reservoir_response_data.keys():
```

```
                    reservoir_response_data[response_var_name] =
sim_track_file_data[_][response_var_index]
                except AssertionError:
                    warnings.warn('Incorrect reservoir response variable name. Assemble NaN')
                    if response_var_name not in reservoir_response_data.keys():
                        reservoir_response_data[response_var_name] = np.nan
            reservoir_response_df.append(reservoir_response_data)
        reservoir_response_df = pd.DataFrame(data=reservoir_response_df)
        time_steps_df = self.wrap_time_steps()
        return reservoir_response_df, time_steps_df



def interpolate_reservoir_response(reservoir_response_df: pd.DataFrame, time: float):
    interp_reservoir_response = dict()
    reservoir_response_time = [float(_) for _ in reservoir_response_df['reservoir_response_time']]
    interp_lower_idx: int = -1
    interp_upper_idx: int = -1
    for _ in range(len(reservoir_response_time) - 1):
        if reservoir_response_time[_] <= time <= reservoir_response_time[_ + 1]:
            interp_lower_idx = _
            interp_upper_idx = _ + 1
    for reservoir_var_name in list(reservoir_response_df.keys()):
        reservoir_var_name_idx = list(reservoir_response_df.keys()).index(reservoir_var_name)
        if reservoir_var_name not in interp_reservoir_response.keys() and \
                reservoir_var_name != 'reservoir_response_time':
            lower_value = float(reservoir_response_df.iloc[interp_lower_idx,
reservoir_var_name_idx])
            lower_time = reservoir_response_time[interp_lower_idx]
```

```
        upper_value = float(reservoir_response_df.iloc[interp_upper_idx, reservoir_var_name_idx])
        upper_time = reservoir_response_time[interp_upper_idx]
        interp_value = lower_value + (time - lower_time) * (upper_value - lower_value) / (upper_time - lower_time)
        interp_reservoir_response[reservoir_var_name] = interp_value
    return interp_reservoir_response
from workflow.fracture_profile import *
from workflow.reservoir_response import *

from src.utils.experimental import ExperimentalResFracFiles
from smt import surrogate_models

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

# "Miscellaneous" workflows to form surrogate function(s) (TBD)

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

class SurrogateDirectory(object):
    def __init__(self, result_dir):
        super(SurrogateDirectory, self).__init__()
        self.result_dir = result_dir
        self.surrogate_case_prefix = 'doe_case_'
        self.num_cases = 100
        self.doe_assembler: DesignOfExperimentsAssembly = None
        # Init the fracture profile
        self.fracture_profile_resolution = None
```

```
    # Init the reservoir response
    self.reservoir_response_prefix = None
    self.reservoir_response_var_names = [None]
    # Experimental, fix to remove in future
    self.experimental_doe_params = None
    self.experimental_doe_data: pd.DataFrame = pd.DataFrame()

def init_fracture_profile(self, fracture_profile_resolution):
    self.fracture_profile_resolution = fracture_profile_resolution

def init_reservoir_response(self, reservoir_response_var_names):
    self.reservoir_response_prefix = 'sim_track_'
    self.reservoir_response_var_names = reservoir_response_var_names

def init_experimental_doe_data(self):
    # Experimental, fix to remove in future
    doe_df = list()
    for _ in range(self.num_cases):
        # Get the case name and its directory
        case_name = self.surrogate_case_prefix + str(_)
        case_dir = os.path.join(self.result_dir, case_name)
        settings_file_name = 'settings_' + case_name + '.txt'
        settings_file_dir = os.path.join(case_dir, settings_file_name)
        exp_resfrac_file = ExperimentalResFracFiles(file_dir=settings_file_dir)
        doe_values = exp_resfrac_file.read()
        doe_dict = dict()
        if 'case' not in doe_dict.keys():
            doe_dict['case'] = _
```

```
        for (_, doe_param) in enumerate(self.experimental_doe_params):
            doe_dict[doe_param] = doe_values[_]
        doe_df.append(doe_dict)
    doe_df = pd.DataFrame(data=doe_df)
    self.experimental_doe_data = doe_df


def assemble_surrogate_directory(self, surrogate_dir):
    fracture_profile_assemblers = list()
    reservoir_response_assemblers = list()
    for _ in range(self.num_cases):
        # Get the case name and its directory
        case_name = self.surrogate_case_prefix + str(_)
        case_dir = os.path.join(self.result_dir, case_name)
        # Create the fracture profile assembler
        fracture_profile_assembler = FractureProfileAssembly(resolution=
                                            self.fracture_profile_resolution)
        fracture_profile_assembler.set_root_directories(simulation_dir=case_dir)
        fracture_profile_assemblers.append(fracture_profile_assembler)
        # Create the reservoir response assembler
        reservoir_response_file_name = self.reservoir_response_prefix + case_name + '.csv'
        reservoir_response_assembler = ReservoirResponseAssembly()
        reservoir_response_assembler.set_root_directories(simulation_dir=self.result_dir,
                                            simulation_file_name=case_name,
                                            sim_track_file_name=reservoir_response_file_name,
                                            response_var_names=self.reservoir_response_var_names)
        reservoir_response_assemblers.append(reservoir_response_assembler)
    # Create the surrogate assembler
    self.init_experimental_doe_data()
```

```
        if self.doe_assembler is None:
            surrogate_assembler = SurrogateAssembly(frac_profile_assemblers=fracture_profile_assemblers,
                                                    res_response_assemblers=reservoir_response_assemblers,
                                                    doe_assembler=self.doe_assembler)
            surrogate_assembler.experimental_doe_data = self.experimental_doe_data
        else:
            surrogate_assembler = SurrogateAssembly(frac_profile_assemblers=fracture_profile_assemblers,
                                                    res_response_assemblers=reservoir_response_assemblers,
                                                    doe_assembler=self.doe_assembler)
        surrogate_df = surrogate_assembler.assemble_surrogate()
        surrogate_df.to_csv(surrogate_dir)
        return surrogate_df



class SurrogateAssembly(object):
    def __init__(self, frac_profile_assemblers: List[FractureProfileAssembly],
                 res_response_assemblers: List[ReservoirResponseAssembly],
                 doe_assembler: DesignOfExperimentsAssembly):
        super(SurrogateAssembly, self).__init__()
        self.frac_profile_assemblers = frac_profile_assemblers
        self.res_response_assemblers = res_response_assemblers
        self.doe_assembler = doe_assembler
        self.surrogate_x_var_names = None
        self.surrogate_y_var_names = None
        # Experimental, fix to remove in future
        self.experimental_doe_data = pd.DataFrame()
```

```
def assemble_surrogate(self):
    """
    Assemble surrogate data from FractureProfileAssembly, ReservoirResponseAssembly &
DesignOfExperimentsAssembly
    1. Call DesignOfExperimentsAssembly to generate settings/input files (ResFrac files) and the
DoE data (.csv)
    Per DoE case:
        2. Call FractureProfileAssembly to generate sensor data (fixed parameters in the surrogate)
        3. Call ReservoirResponseAssembly to generate surrogate data (.csv) & to train surrogate
        4. Arrange surrogate data as follows:
        - DoE data (from DesignOfExperimentsAssembly)
        - Sensor data (from FractureProfileAssembly)
        - Reservoir response data (from ReservoirResponseAssembly)
    5. Return the surrogate data (.csv) & surrogate wrapper (further used in
FractureCalibrationAssembly &
    HistoryMatchingAssembly)
    """
    try:
        if self.doe_assembler is None:
            assert self.experimental_doe_data.empty is False
            num_cases = self.experimental_doe_data.shape[0]
            doe_params = list(self.experimental_doe_data.columns)[1:]
        else:
            # assert self.doe_assembler.doe_data.shape[0] == len(self.frac_profile_assemblers)
            # assert self.doe_assembler.doe_data.shape[0] == len(self.res_response_assemblers)
            num_cases = self.doe_assembler.doe_data.shape[0]
            doe_params = self.doe_assembler.doe_params
        surrogate_data_df = list()
        # Loop through all DoE cases
```

```
        for _ in range(num_cases):
            print('Process case ', _)
            # Extract the corresponding assemblers fo the case
            frac_profile_assembler: FractureProfileAssembly = self.frac_profile_assemblers[_]
            res_response_assembler: ReservoirResponseAssembly = self.res_response_assemblers[_]
            # Assemble the fracture profile and reservoir response
            fracture_profile_df = frac_profile_assembler.asemble_fracture_profile()
            reservoir_response_df, time_steps_df = res_response_assembler.assemble_reservoir_response()
            # Extract the recorded time steps for the surrogate ( always fracture profile time steps )
            fracture_profile_time_steps = list(fracture_profile_df.index)
            surrogate_time_steps = fracture_profile_time_steps
            # Loop through all similar recorded time steps
            for time_step in surrogate_time_steps:
                surrogate_data = dict()
                surrogate_time = fracture_profile_df.loc[time_step, 'fracture_profile_time']
                # Add DoE case number
                if 'case' not in surrogate_data.keys():
                    surrogate_data['case'] = _
                # Add time step (in sec)
                if 'surrogate_time' not in surrogate_data.keys():
                    surrogate_data['surrogate_time'] = surrogate_time
                # Loop through all DoE params
                for doe_param in doe_params:
                    # Add DoE params
                    if doe_param not in surrogate_data:
                        if self.doe_assembler is not None:
                            surrogate_data[doe_param] = self.doe_assembler.doe_data.loc[_, doe_param]
```

```
                else:
                    # Experimental, fix to remove in future
                    surrogate_data[doe_param] = self.experimental_doe_data.loc[_, doe_param]
            # Assemble the fracture profile and reservoir response per time step
            fracture_profile_time_step = fracture_profile_df.loc[time_step, :]
            # Need interpolation here
            reservoir_response_time_step                                    =
interpolate_reservoir_response(reservoir_response_df,
                                                surrogate_time)
            # Loop through all fracture profile params
            for frac_profile_param in list(fracture_profile_time_step.index):
                # Add fracture profile params
                if frac_profile_param not in surrogate_data.keys() and \
                    frac_profile_param != 'fracture_profile_time':
                    surrogate_data[frac_profile_param]                      =
fracture_profile_time_step[frac_profile_param]
            # Loop through reservoir response params
            for res_response_param in reservoir_response_time_step:
                # Add reservoir response params
                if res_response_param not in surrogate_data.keys():
                    surrogate_data[res_response_param]                      =
reservoir_response_time_step[res_response_param]
            # Assemble the surrogate data
            surrogate_data_df.append(surrogate_data)
            self.surrogate_x_var_names = \
                ['surrogate_time'] + doe_params + list(fracture_profile_time_step.index)
            self.surrogate_y_var_names = list(reservoir_response_time_step.keys())
        # Complete the assembly as pd.DataFrame
        surrogate_data_df = pd.DataFrame(data=surrogate_data_df)
```

```
            return surrogate_data_df
        except AssertionError:
            warnings.warn('Number of Design Of Experiment cases and number of fracture profile/reservoir response'
                          'do not match')
            return pd.DataFrame()



class Surrogate(object):
    def __init__(self, surrogate_assembler: SurrogateAssembly):
        super(Surrogate).__init__()
        self.surrogate_assembler = surrogate_assembler
        self.surrogate_backend = None


    def train(self, surrogate_y_var_name):
        # TODO: Implement method to train a surrogate
        surrogate_data = self.surrogate_assembler.assemble_surrogate()
        surr_backend = surrogate_models.KRG(theta0=[1e-2])
        surrogate_x_data = surrogate_data[self.surrogate_assembler.surrogate_x_var_names]
        try:
            assert surrogate_y_var_name in self.surrogate_assembler.surrogate_y_var_names
            surrogate_y_data = surrogate_data[surrogate_y_var_name]
            surr_backend.set_training_values(xt=surrogate_x_data.to_numpy(),
yt=surrogate_y_data.to_numpy())
            surr_backend.train()
            self.surrogate_backend = surr_backend
        except AssertionError:
            warnings.warn('Incorrect surrogate y variable name. Can not train.')
```

```
    def validate(self, surrogate_y_var_name):
        # TODO: Implement method to validate a surrogate
        surrogate_data = self.surrogate_assembler.assemble_surrogate()
        surr_backend = surrogate_models.KRG(theta0=[1e-2])
        surrogate_x_data = surrogate_data[self.surrogate_assembler.surrogate_x_var_names]
        try:
            assert surrogate_y_var_name in self.surrogate_assembler.surrogate_y_var_names
            surrogate_y_data = surrogate_data[surrogate_y_var_name]
            surr_backend.set_training_values(xt=surrogate_x_data.to_numpy(),
yt=surrogate_y_data.to_numpy())
            surr_backend.train()
            self.surrogate_backend = surr_backend
        except AssertionError:
            warnings.warn('Incorrect surrogate y variable name. Can not validate.')


    def __call__(self, params: np.array, *args, **kwargs):
        try:
            assert self.surrogate_backend is not None
            return self.surrogate_backend(params)
        except AssertionError:
            warnings.warn('No surrogate backend set. Return None')
            return None
from sidebar.main_sidebar import *
from callbacks.sidebar import *


from callbacks.doe_gui import *
from callbacks.frac_cal_gui import *


app = dash.Dash(__name__,
```

```
        external_stylesheets=[dbc.themes.BOOTSTRAP, dbc.icons.FONT_AWESOME],
        background_callback_manager=background_callback_manager,
        use_pages=True)

app.layout = html.Div(children=[sidebar, dash.page_container], id='layout')
set_sidebar(app)

if __name__ == "__main__":
    app.run_server(port=5000, debug=True)
```