

# PGE 385K – Advanced Multi-Well Formation Evaluation

## Final Project

**Misael M. Morales & Oriyomi Raheem**

**Tuesday December 12, 2023**

---

The objectives of the project are: (a) to perform a complete petrophysical/elastic assessment for the 2 wells included in the data set provided to students following the steps covered in Homework Projects 1 through 3, (b) to perform spatial correlation of flow units and fluid boundaries within flow units, and (c) to diagnose vertical and lateral limits of pressure compartments.

The following is a list of items that should be included in your report:

1. Executive summary, background, and introduction.
2. Plots of the main logs organized by well.
3. Quality control of multi-well measurements via depth matching and log balancing.
4. Description of lithology.
5. Description of the sedimentary environment.
6. Identification of porous and permeable units.
7. Identification of fluids saturating the porous and permeable units.
8. Rock classification based on porosity-permeability relations.
9. Quantitative assessment of porosity with adherence to core measurements.
10. Quantitative assessment of water/hydrocarbon saturation with adherence to (a) capillary-pressure and (b) core measurements.
11. Quantitative assessment of permeability with adherence to core measurements.
12. Well-to-well correlation of lithology, flow units, and saturation fluids.
13. Sketch of fluid contacts on the well-to-well correlation and structural plots from the previous item.
14. Assess whether P-wave and/or S-wave impedances could be used to identify lithology, porosity, and fluids.
15. Biot-Gassmann fluid substitution equations and sensitivity analysis of elastic/petrophysical correlations.
16. Conclusions and recommendations.

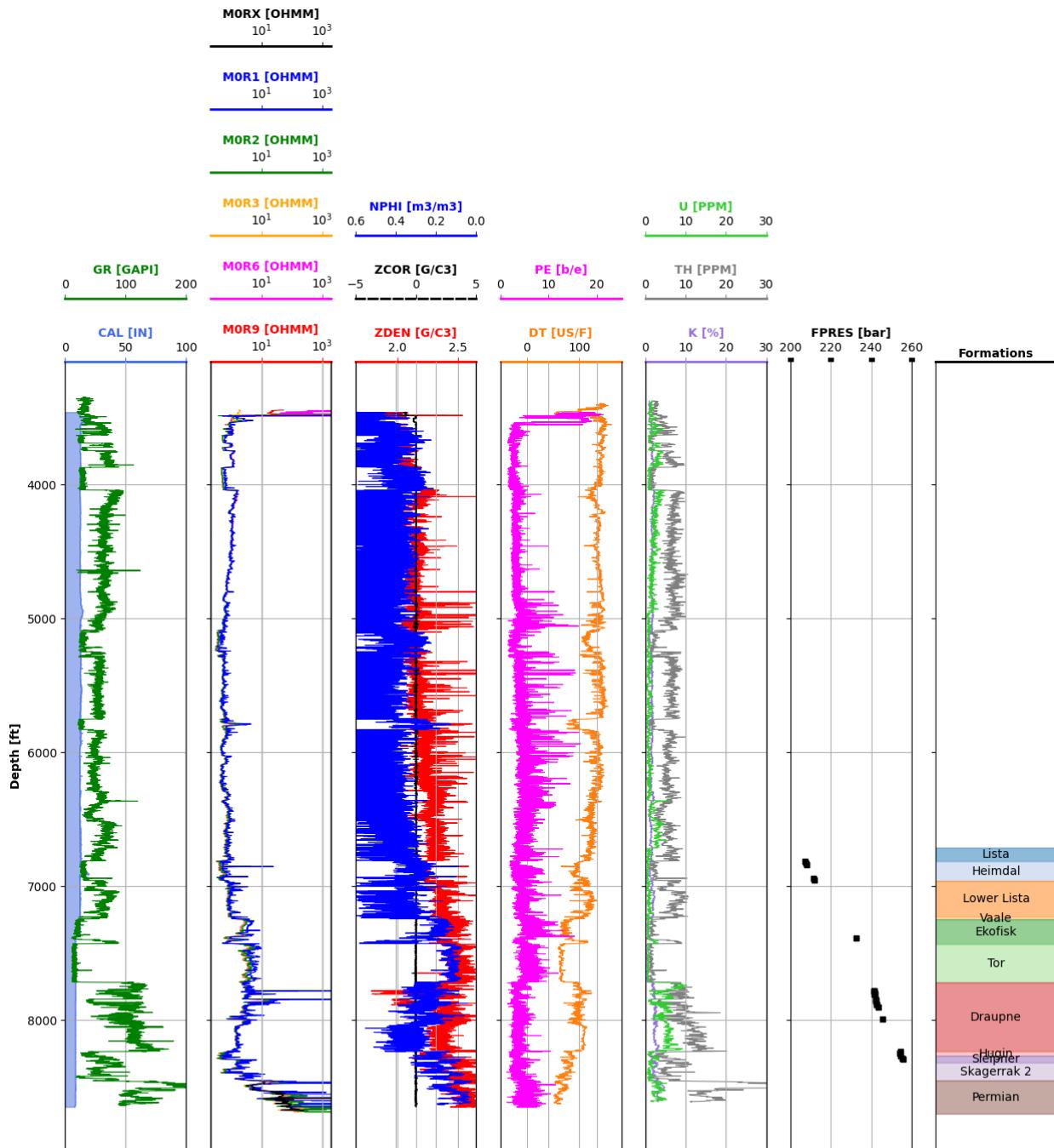
## **Executive Summary / Introduction:**

Petrophysical interpretation of multi-well projects in laminated sedimentary sequences is challenging but essential for accurate characterization of subsurface energy resource systems. We perform a comprehensive analysis to estimate the petrophysical properties of a multi-well shaly-sandstone sequence from a turbidite deposit in the North Sea for its electrical, storage, flow, and elastic properties. The well logs have different depth intervals, though belong to the same formation, so we also require zonation and well correlation analysis. The petrophysical evaluation is performed using TechLog and Python, and we compare several techniques for the estimation of properties based on different petrophysical measurements including volumetric concentration of shale from parallel- and perpendicular-to-bedding-plane resistivities, rock classification, mineral composition estimation, permeability estimation and saturation-height analysis, and acoustic/elastic characteristics. A comprehensive evaluation of the two wells is performed initially. Multiple cross-plots are constructed to aid our interpretation and calculations. Furthermore, we calculate the properties for both of the wells present in the project and compare the calculations to available core data. In conclusion, we prepare a thorough workflow for the petrophysical interpretation of a laminated shaly-sandstone sequence and quantitatively and qualitatively estimate the electrical, storage, flow, and elastic properties of the formation in both wells. The following report is separated into the following sections:

- 1) Preliminary interpretation
  - a. Plots of the main logs organized by well.
- 2) Qualitative interpretation
  - a. Quality control of the well logs
  - b. Description of lithology
  - c. Description of sedimentary environment.
- 3) Flow and Storage interpretation
  - a. Identification of porous and permeable units
  - b. Identification of fluids saturating the porous and permeable units:
- 4) Rock Classification
  - a. Rock classification based on porosity-permeability relations.
- 5) Quantitative interpretation
  - a. Quantitative assessment of porosity with adherence to core measurements,
  - b. Quantitative assessment of water/hydrocarbon saturation with adherence to (a) capillary-pressure and (b) core measurements
  - c. Quantitative assessment of permeability with adherence to core measurements.
- 6) Well-to-well correlation
  - a. Correlation of lithology, flow units, saturation fluids, and fluid contacts.
- 7) Sonic interpretation
  - a. P-wave and S-wave impedances to identify lithology, porosity, and fluids.
  - b. Biot-Gassman fluid substitution

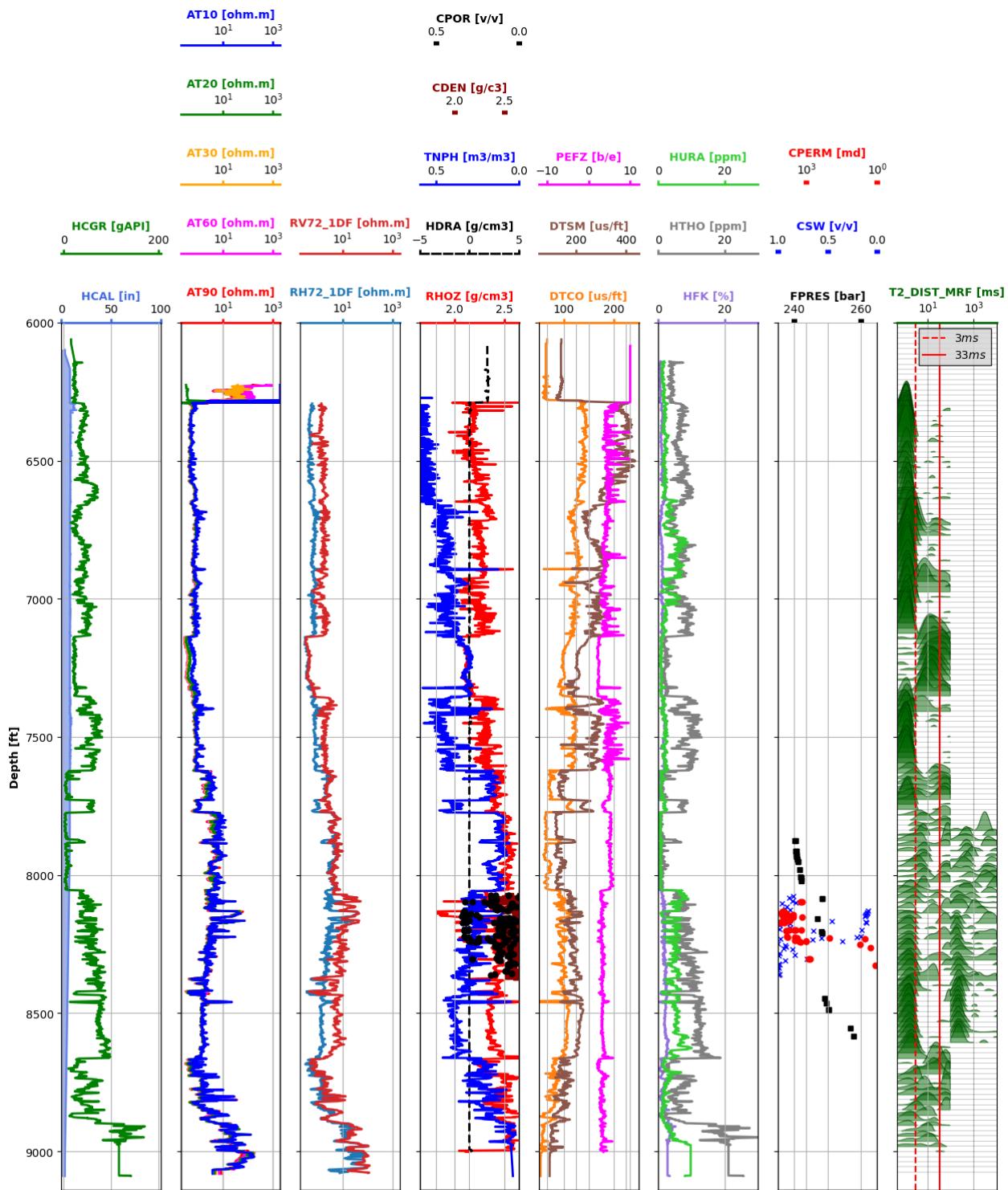
## 1. Main logs organized by well:

We begin by plotting the entire suite of well logs for each of the two wells without any preprocessing. We also include core measurements and formation pressure data if available, as well as formation tops data. Figures 1 and 2 show the logs for Well 8 and Well 16S, respectively.



HANZ PROSPECT | ESSO NORGE A.S. | 25/10-8

Figure 1: Well logs for Well 8.



Hanz Prospect | Aker BP | 25/10-16 S

Figure 2: Well logs for Well 16S.

## 2. Quality control of the well logs, Description of lithology, and Description of sedimentary environment:

We start by cropping the depth intervals of Well 8 and Well 16S to avoid dubious measurements at the very top and bottom of the logs. Well 8 is cropped between 3,490 and 8,465 ft, and Well 16S is cropped between 6,290 and 8,990 ft. We verify that the logs and core data are corresponding in depth, and thus do not require depth shifting. Minor adjustments could potentially be made to completely align the well logs, but the provided data appears to have been already depth-shifted.

From the well log headers, we notice that both wells are in the Hanz Prospect field, in Norway. This North Sea field is characterized as a turbidite sequence with siliciclastic sequences of sand and shale. However, we suspect possible limestone sequences based on the PEF log and the readings from the GR, Resistivity, and RHOB logs. Since both wells span a relatively large vertical column, we decided to perform automatic zonation using dynamic time warping (DTW) and unsupervised machine learning (K-means clustering). This process will help us describe lithology and aid in the sedimentary environment characterization.

DTW is an optimization problem for two sequences,  $x = \{x\}_{i=0}^N$  and  $x' = \{x'\}_{i=0}^M$ . The sequences are aligned based on a path,  $\pi$ , of length  $K$ , where  $\pi_0 = (0,0)$  and  $\pi_{K-1} = (n-1, m-1)$ , and the path increases monotonically following a transition matrix  $A(x, x')$  based on the Euclidean distances of the two sequences,  $D_q(x, x')$ . Thus, DTW can be formally written as shown in Eq. 1:

$$DTW_q(x, x') = \min_{\pi \in A(x, x')} \langle A_\pi, D_q(x, x') \rangle^{\frac{1}{q}} = \min_{\pi \in A(x, x')} \left( \sum_{(i,j) \in \pi} d(x_i, x'_j) \right)^{\frac{1}{q}} \quad (1)$$

Once we obtain the DTW paths based on the GR logs, we apply K-means clustering to separate the different zones into similar classes. We constrain the clustering algorithm to a maximum of 5 classes. Figure 3 shows the results of the automatic zonation algorithm with DTW and K-means. The rightmost track shows the true formation tops for Well 8, indicating a good match between our automatic zonation algorithm and the true zones. The zonation of Well 16S is based on the those of Well 8, where Well 8 is the training set and Well 16S is the testing data. We only use a subset of Well 8, from 5,750 to the bottom, and then use the entire Well 16S, since we believe that these are the matching intervals of the two wells.

These automatic zonation results confirm our initial guess that this is a turbidite sequence of sandstone and shale, however we do observe some massive limestone and shaly limestone units toward the bottom of both wells. This indicates that this was a deepwater sedimentary depositional environment that was initially shallow-water with hemipelagic sedimentation of calcite-rich sources and began to sink into deepwater. In the deepwater environment, mass transport deposits caused recurrent Ta-Td turbidite sequences of laminated shale and sandstone.

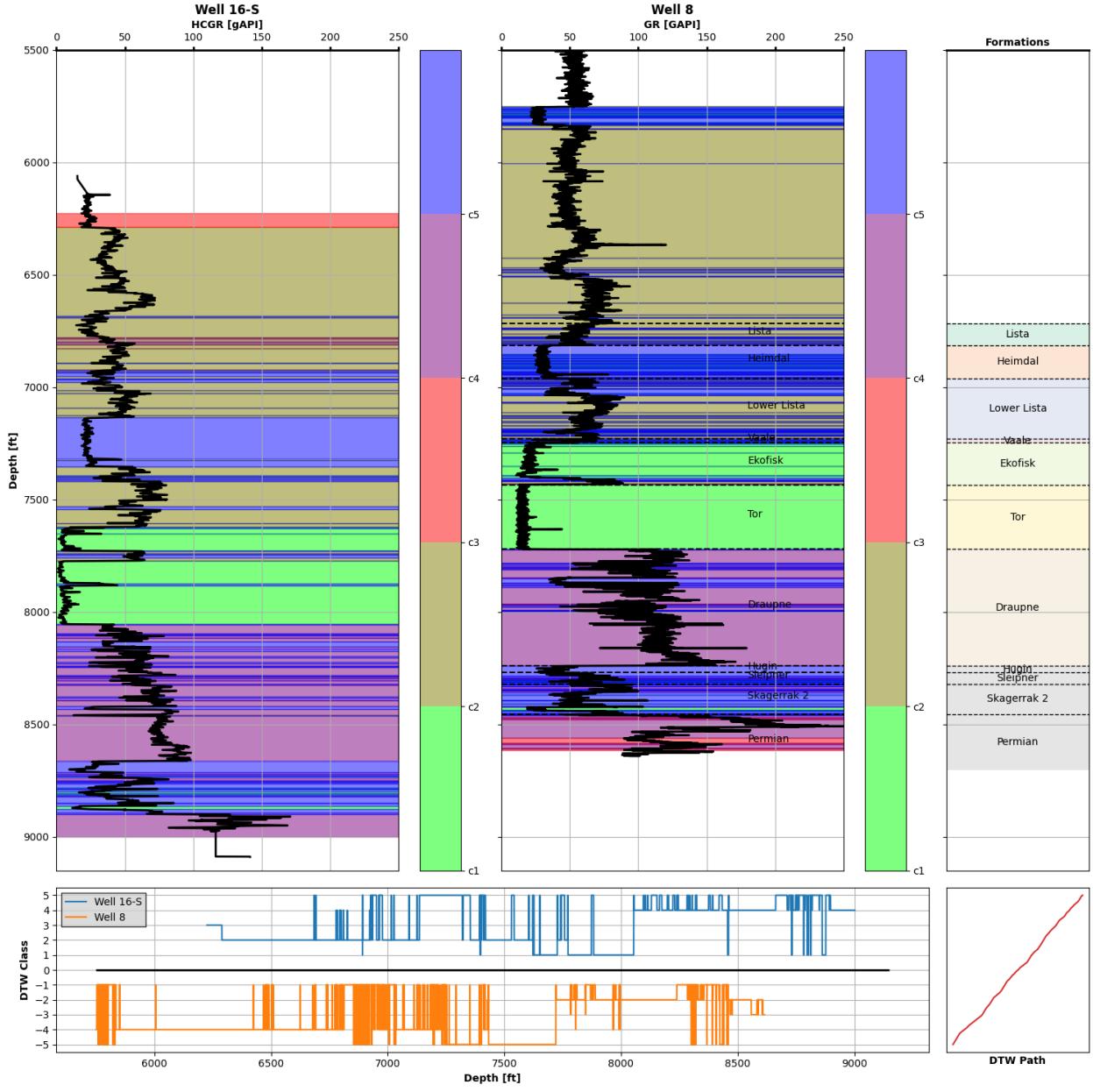


Figure 3: Automatic zonation with dynamic time warping and K-means clustering.

Next, we construct the cross-plot of GR-vs-AT90 for both wells using the zonation results from the automatic DTW-K-means algorithm. Figure 4 shows the cross-plots for each well based on the automatic zonation. We observe that the majority of the points correspond to 3 zones, namely the shale zone, water-filled zone, or light hydrocarbon-filled zone, as expected.

We continue to interpret our wells by constructing cross-plots of GR, PEF, and RHOB. We separate by zones using the automatic zonation algorithm and observe the relationship between the possible lithologies based on the automatic zones. We detect three major zones, interpretable as the shale, sandstone, and limestone zones in both wells, as shown in Figure 5.

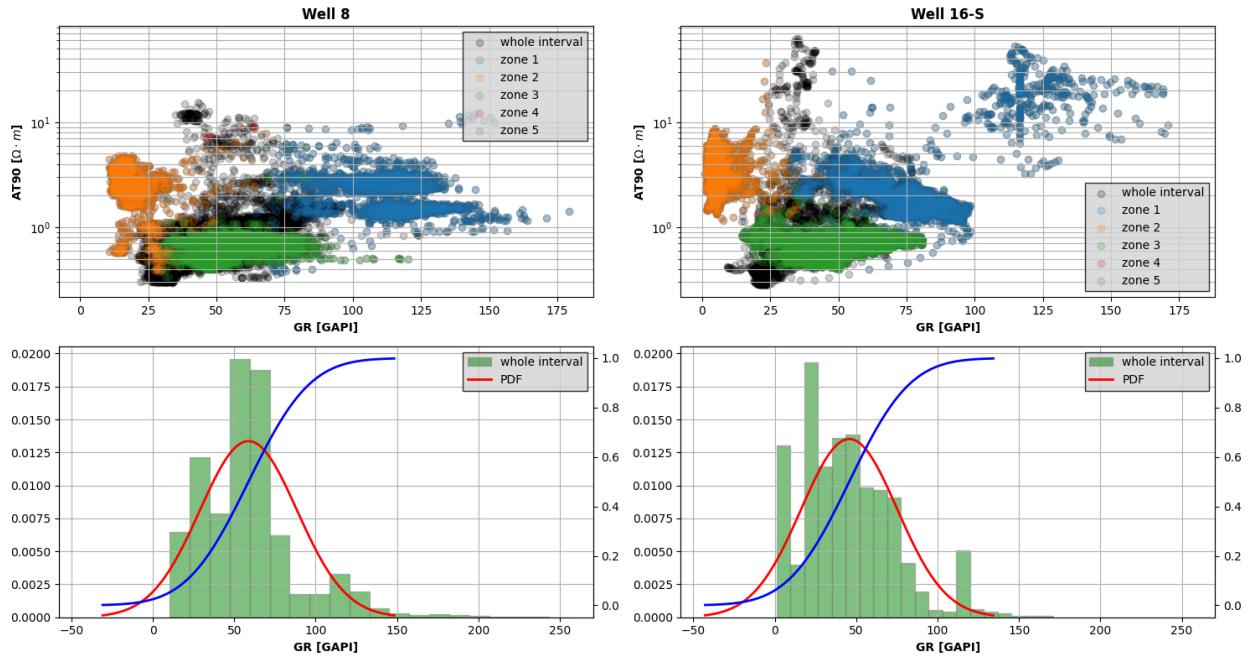


Figure 4: Cross-plot of GR-vs-AT90 for both wells.

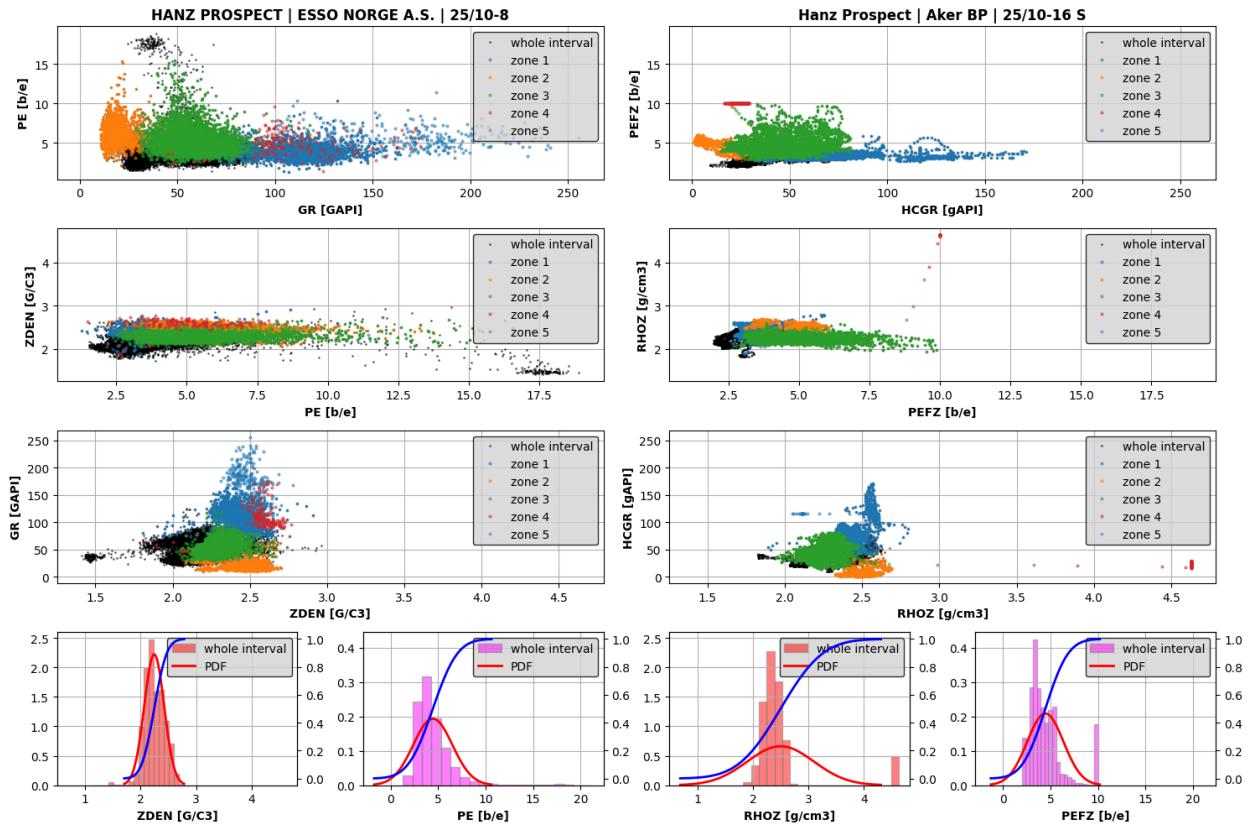


Figure 5: Cross-plots of GR, PEF, and RHOB for both wells.

Furthermore, we construct cross-plots for the spectral gamma ray logs of both wells to determine the type of clay minerals present and to ensure that the shale properties are consistent throughout the entire depth intervals. This analysis is only done for Well 16S since it is the one that has the spectral GR log. Figure 6 shows the K-vs-Th cross-plot of Well 16S separated by aquifer and shale zones.

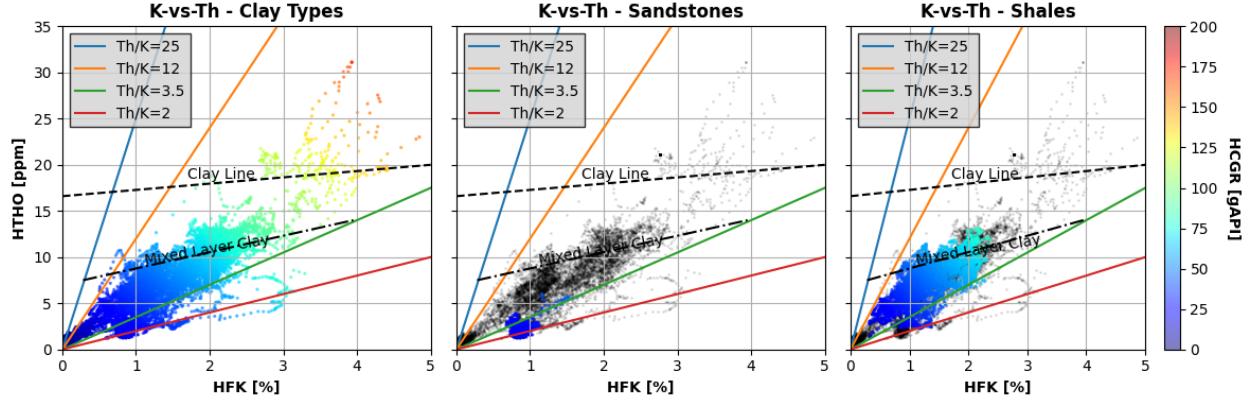


Figure 6: Cross-plot of Potassium-vs-Thorium for Well 16S.

The next step is to calculate the volumetric concentration of shale,  $C_{sh}$ , by inversion of the parallel- and perpendicular-to-bedding-plane resistivities,  $R_V$  and  $R_H$ . We do this for Well 16S only since it contains the  $R_V$  and  $R_H$  logs. First, we construct cross-plots for RHOB-vs-GR and  $R_V$ -vs- $R_H$  to ensure that the formation is laminated and anisotropic, as shown in Figure 7.

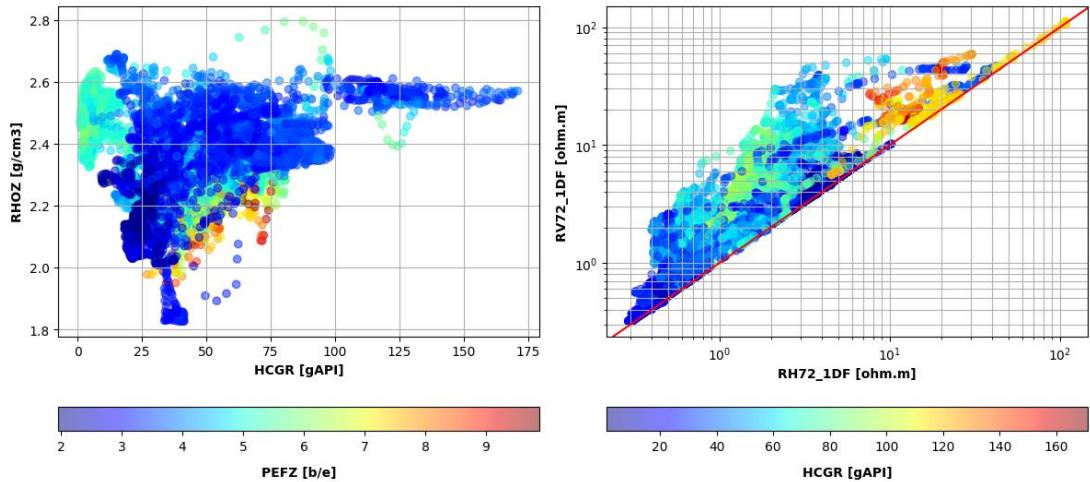


Figure 7: Cross-plots of GR-vs-RHOB (left) and resistivity anisotropy (right) for Well 16S.

Next, we perform inversion to estimate  $C_{sh}$  and  $R_s$  from  $R_V$  and  $R_H$ . We solve the system of equations given by Eq. 2:

$$\begin{cases} R_V = C_{sh}R_V^{sh} + (1 - C_{sh})R_s \\ \frac{1}{R_H} = \frac{C_{sh}}{R_H^{sh}} + \frac{(1 - C_{sh})}{R_s} \end{cases} \quad (2)$$

To aid the inversion, we estimate the horizontal conductivity instead of resistivity,  $\sigma_H$ , and then back-transform. To minimize the norm of the error, we rewrite the system of equations as:

$$\begin{cases} 0 = [C_{sh}R_V^{sh} + (1 - C_{sh})R_S] - R_V = [e(R_V)] \\ 0 = [C_{sh}\sigma_H^{sh} + (1 - C_{sh})R_S] - \sigma_H = [e(R_H)] \end{cases} = e(x)$$

$$\min_x \|W_d e(x)\|_2^2 + \lambda \|x\|_2^2$$

We apply bounds on the inversion such that  $C_{sh} \in [0,1]$ , and apply a data weighting matrix,  $W_d$ , such that  $W_d = \text{diag}\left(\frac{1}{R_V}, R_h\right)$ . The initial guess for  $x^0 = \{C_{sh}^0, R_S^0\} = [0.5, 1.5]$ , and the regularization parameter,  $\lambda$ , is set to  $1 \times 10^{-4}$ . We set the tolerance to  $1 \times 10^{-6}$  and maximum number of iterations to 1,000 using the L-BFGS-B nonlinear solver. Figure 8 shows the results of the parallel- and perpendicular-to-bedding-plane resistivity inversion, along with the re-simulated  $R_V$  and  $R_H$  logs for Well 16S, and the inversion procedure parameters.

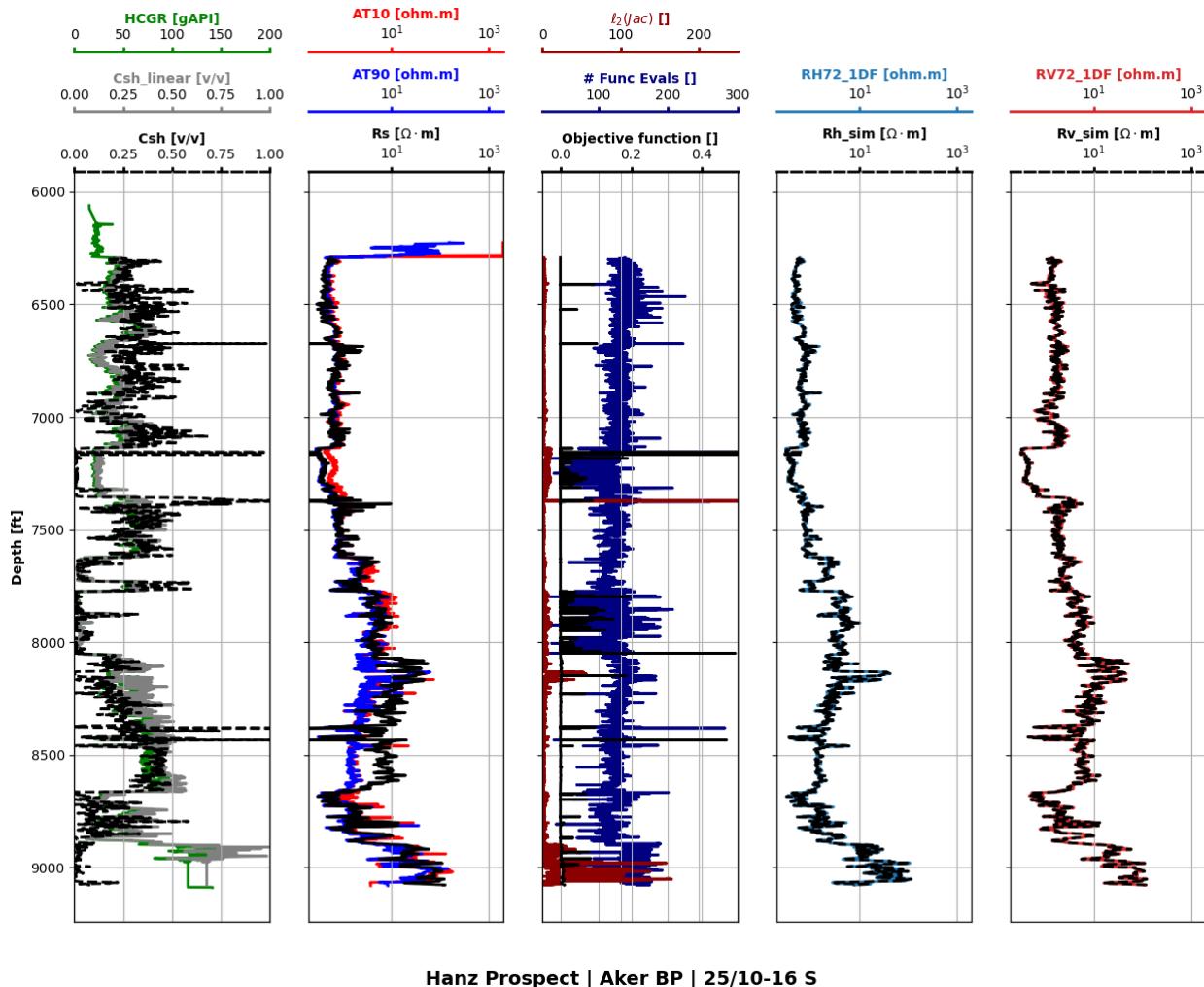


Figure 8: Parallel- and Perpendicular-to-bedding-plane resistivity inversion.

We can also calculate  $C_{sh}$  for Well 8 by using the linear  $C_{sh}$  method. Given that we are in a laminated shaly-sandstone system, we can use the following:

$$C_{sh} = \frac{\gamma - \gamma_s}{\gamma_{Sh} - \gamma_s} \quad (3)$$

Figure 9 shows the GR and  $C_{sh}$  logs for Well 8 and Well 16S, respectively.

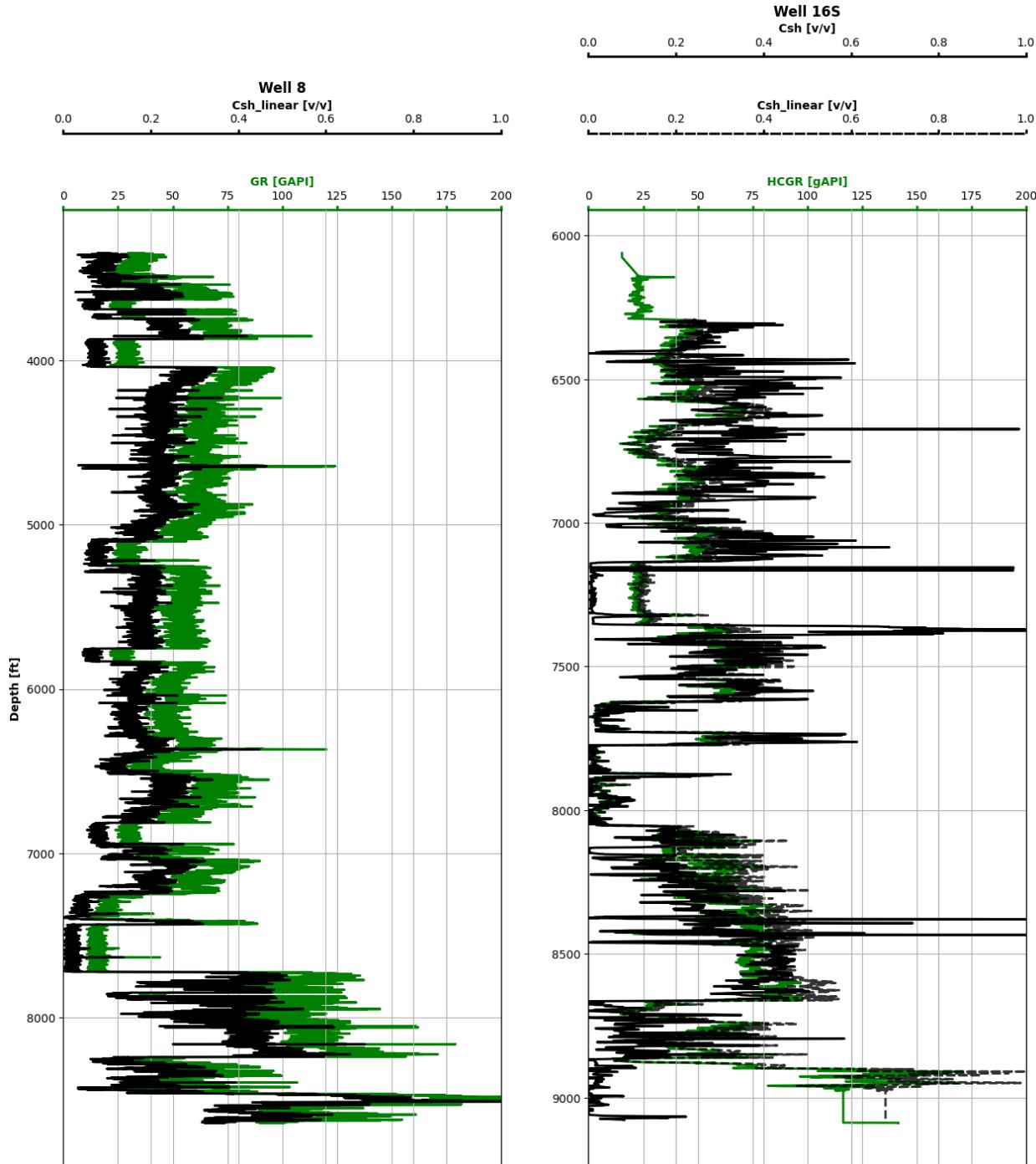
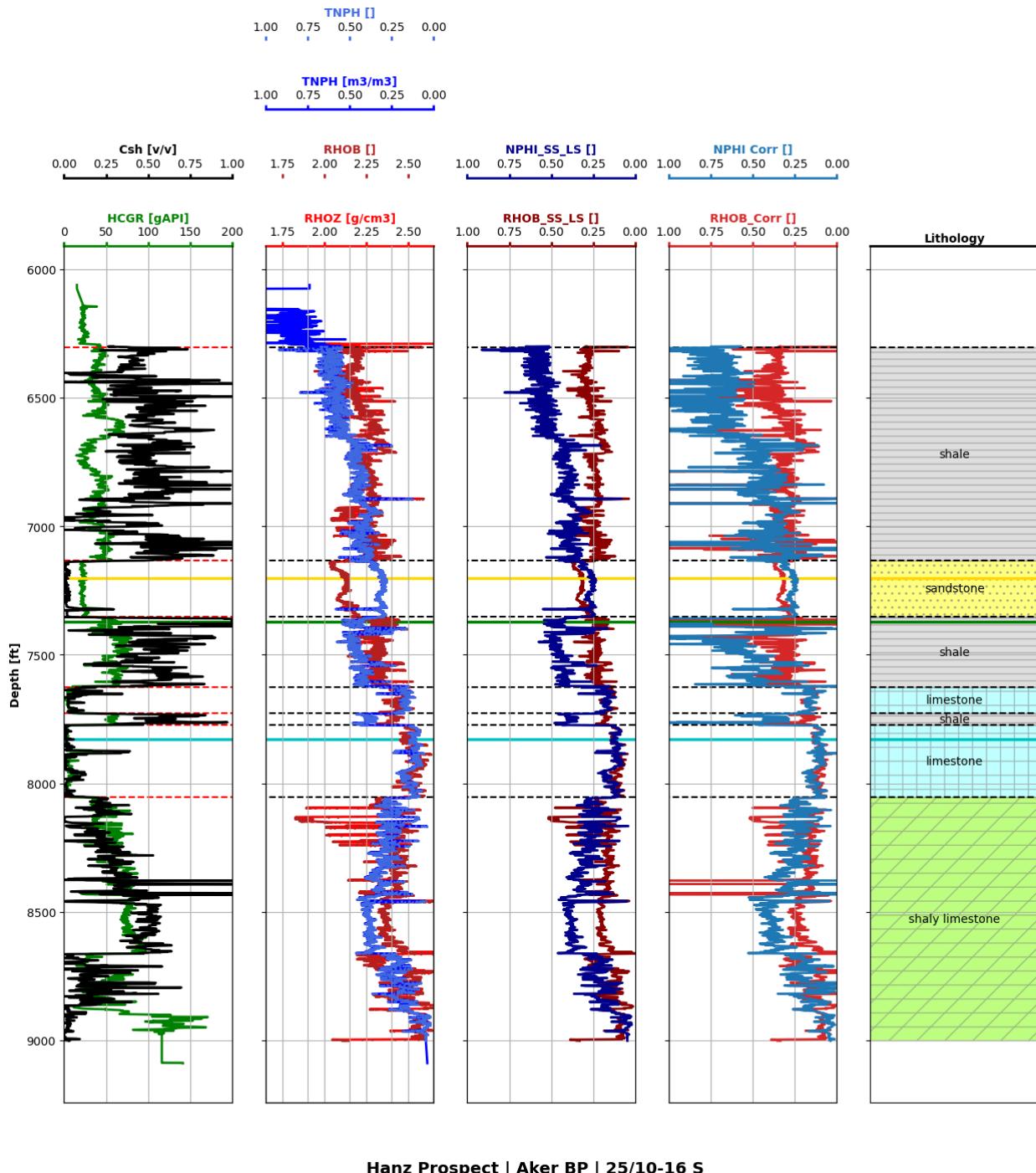


Figure 9:  $C_{sh}$  for Well 8 (left) and Well 16S (right).

We continue our analysis of the lithology by fixing the bulk density and neutron porosity logs. Since we have (shaly) sandstone zones as well as limestone zones, we must correct the RHOB and NPHI logs to the appropriate lithology for the sandstone zones and limestone zones. Figure 10 shows the results of the corrected RHOB and NPHI logs for Well 16S.



Hanz Prospect | Aker BP | 25/10-16 S

Figure 10: RHOB and NPHI correction for Well 16S based on corrected lithology.

### 3. Identification of porous and permeable units, and Identification of fluids saturating the porous and permeable units:

Using the log values, we can now tabulate the “pure” sandstone, shale, and limestone points of the two wells as shown in Tables 1 and 2. We also attempt to manually estimate the potential lithologies by zones for Wells 8 and 16S, respectively.

*Table 1: Log values of “pure” zones for Well 8 and potential zonation.*

	Depth [ft]	GR [GAPI]	PHI-D [v/v]	PHI-N [v/v]	$\rho$ [g/cc]
Shale point	5710.46	64.2973	0.286	0.3688	2.1781
Sandstone	6827.64	28.0833	0.3284	0.6727	2.1081
Limestone	7331.76	18.9014	0.1589	0.0465	2.4382
	Depth [ft]			Lithology	
Zone 1	5856 - 7137.41			Shale	
Zone 2	6813.86 – 6961.09			Sandstone	
Zone 3	6961.09 - 7004.01			Shale	
Zone 4	7004.01 – 7034.97			Sandstone	
Zone 5	7034.97 – 7248.64			Shale	
Zone 6	7248.64 – 7400.99			Limestone	
Zone 7	7400.99 – 7432.1			Shale	
Zone 8	7432.1 – 7721.2			Limestone	
Zone 9	7721.2 – 8640.42			Shaly Limestone	

*Table 2: Log values of “pure” zones for Well 16S and potential zonation.*

	Depth [ft]	EHGR [GAPI]	PHI-D [v/v]	PHI-N [v/v]	$\rho$ [g/cc]
Shale point	7371.52	56.1664	0.1745	0.4221	2.4116
Sandstone	7202.87	20.9332	0.3170	0.2589	2.1269
Limestone	7830.19	3.3967	0.1523	0.1369	2.4496
	Depth [ft]			Lithology	
Zone 1	6303 - 7137.41			Shale	
Zone 2	7137.41 – 7351.03			Sandstone	
Zone 3	7351.03 - 7625.7			Shale	
Zone 4	7625.7 - 7727.81			Limestone	
Zone 5	7727.81 - 7773.59			Shale	
Zone 6	7773.59 - 8054.12			Limestone	
Zone 7	8054.12 - 9000			Shaly Limestone	

Using the “pure” zone values, we can calculate the corrected neutron and density porosity as follows, and use that to estimate the sandstone porosity:

$$\begin{aligned}\phi_N^{corr} &= \frac{\phi_N - C_{sh}\phi_N^{sh}}{1 - C_{sh}} & \phi_D^{corr} &= \frac{\phi_D - C_{sh}\phi_D^{sh}}{1 - C_{sh}} \\ \phi_S &= \left( \frac{\phi_D^{corr^2} + \phi_N^{corr^2}}{2} \right)^{0.5} & \phi_T &= (1 - C_{sh})\phi_S + C_{sh}\phi_{sh}\end{aligned}\quad (4)$$

We construct a Thomas-Steiber diagram of  $C_{sh}$ -vs- $\phi_T$ , as shown in Figure 11. This helps us identify the sand and shale porosity for each well and validate the type of shaly-sandstone system. However, based on the Thomas-Steiber, we observe a grain-coating clay system instead of the assume laminated system that was interpreted from the well logs.

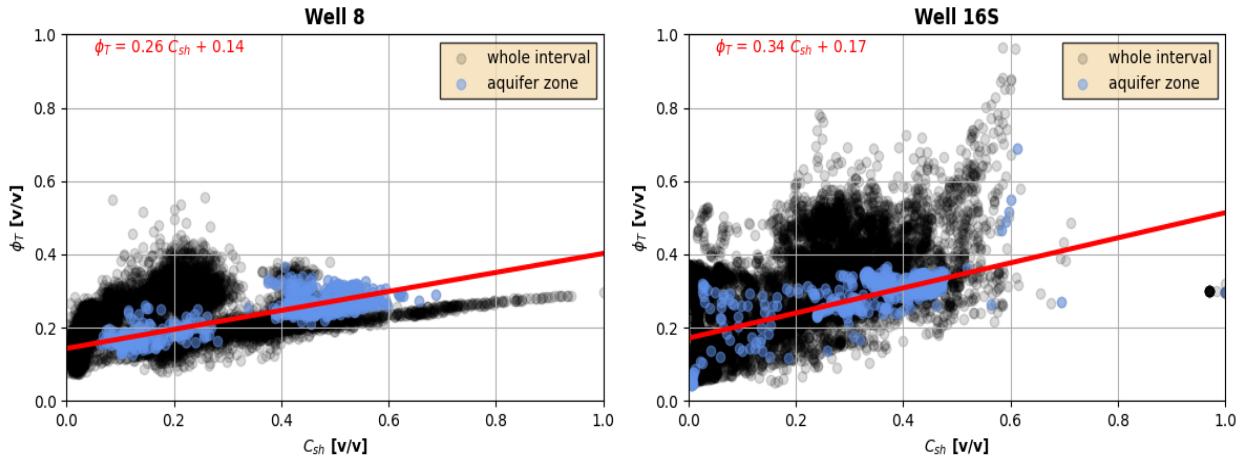


Figure 11: Thomas-Steiber diagram of Well 8 (left) and Well 16S (right).

We plot the continuous porosity and permeability logs. Porosity is calculated following the set of equations in Eq. 4, and permeability is estimated using the Wylie-Rose correlation. Later, we will estimate permeability by co-kriging the porosity and bulk density logs. Figure 12 shows the continuous porosity log for Well 8 and Well 16S, respectively. Figure 13 shows the estimated hydraulic permeability log following the Wylie-Rose correlation based on the porosity log. Permeability is given by  $k = k_w \frac{\phi_S^\kappa}{0.25^\eta}$ , where  $k_w$  is the estimated permeability from the aquifer zone and flushed-zone water saturations, and the parameters  $\{\kappa, \eta\}$  are estimated as  $\{8, 1.75\}$ .

Furthermore, we can use the formation pressure data to estimate the fluid density based on the gradients in both wells. Figure 14 shows the pressure gradients and Tables 3 and 4 show the fluid density calculations for Well 8 and Well 16S, respectively.

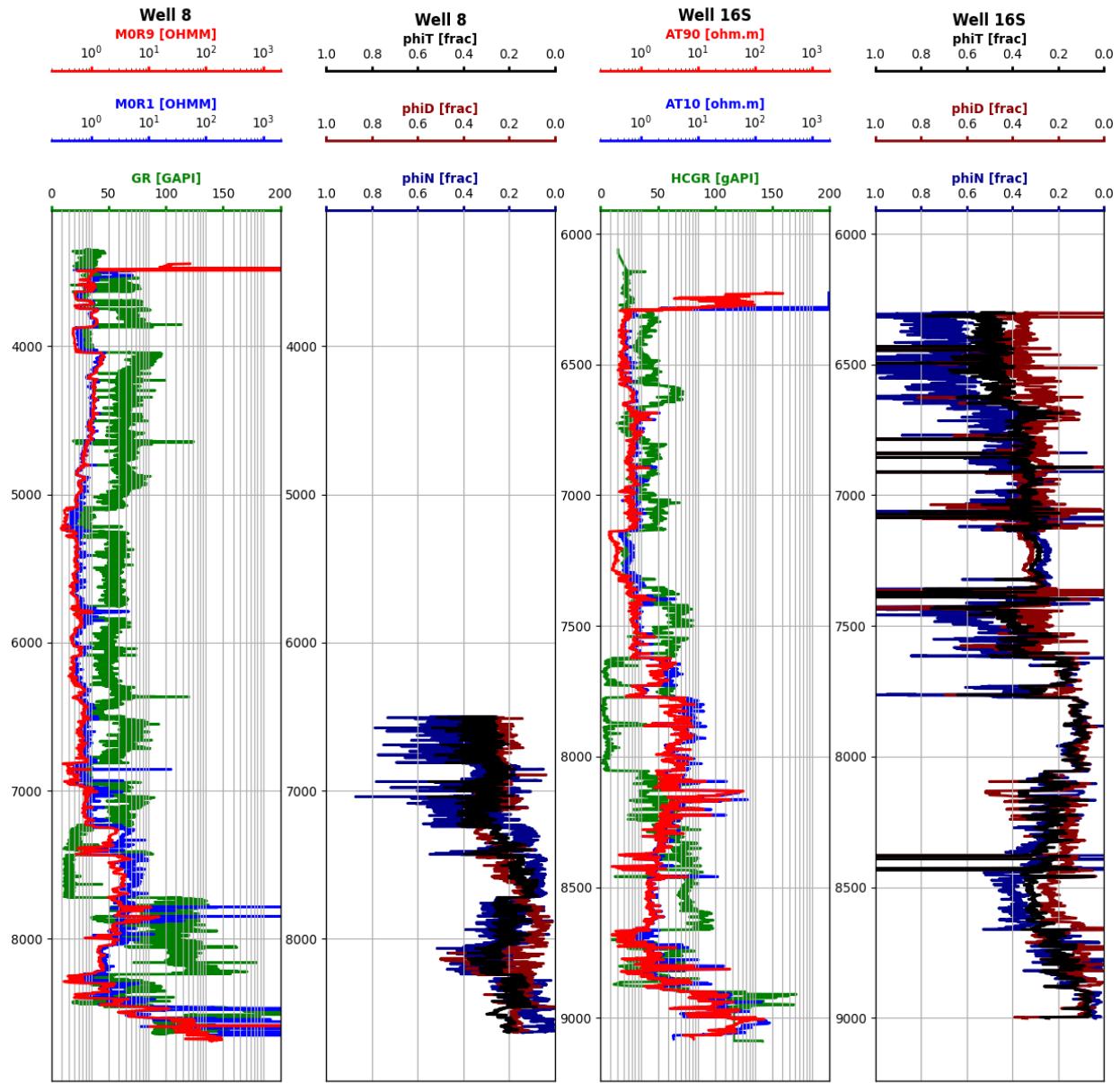


Figure 12: Porosity log for Well 8 (left) and Well 16S (right).

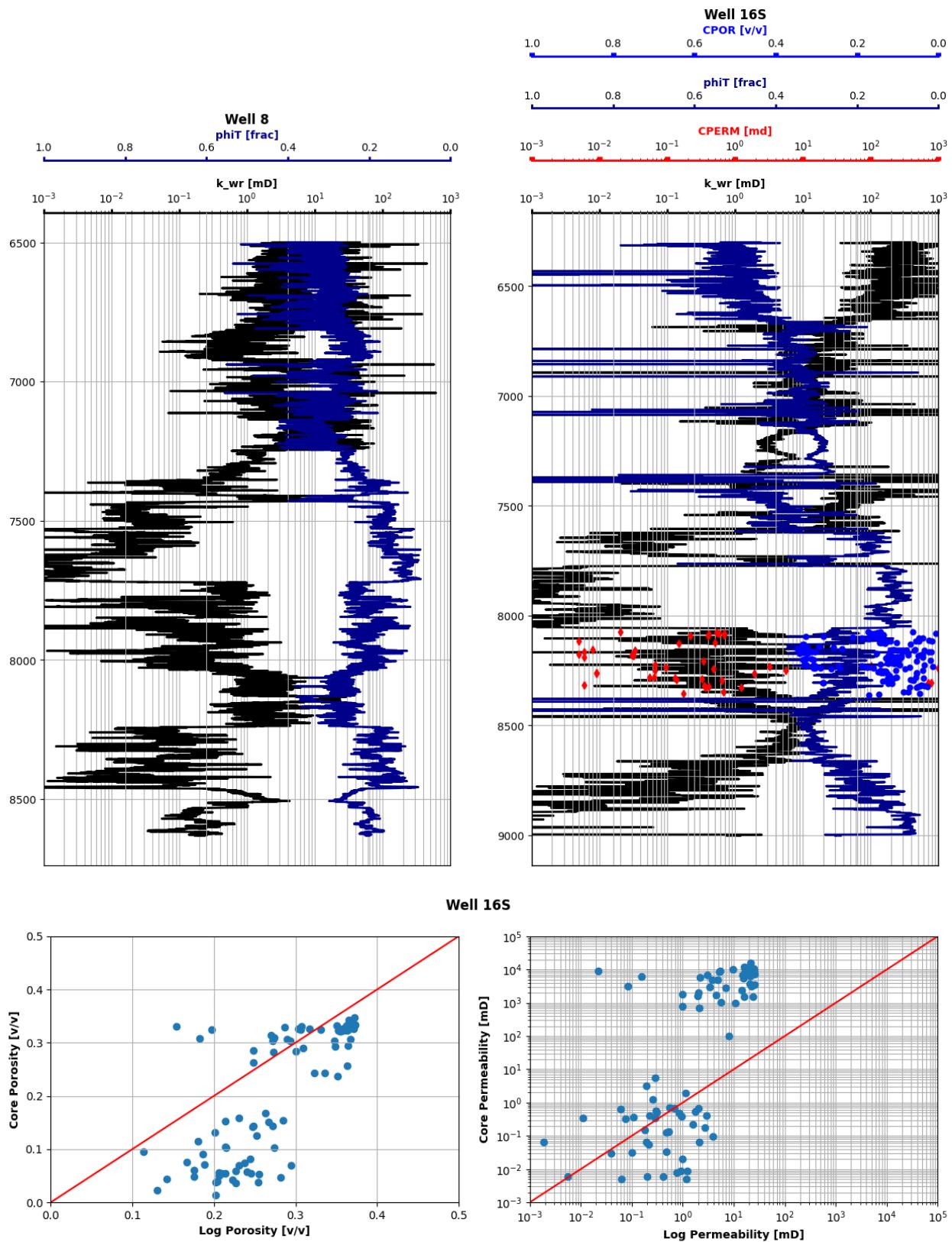


Figure 13: Permeability log for Well 8 (left) and Well 16S (right).

Table 3: Fluid Density calculations for Well 8.

	Gas	Oil	Water
<b>Depth interval [ft]</b>	6800-7000	7650-8000	8150-9000
<b>Pressure gradient [psi/ft]</b>	0.448	0.284	0.406
<b>Density, <math>\rho</math> [g/cc]</b>	1.035	0.656	0.938

Table 4: Fluid density calculations for Well 16S.

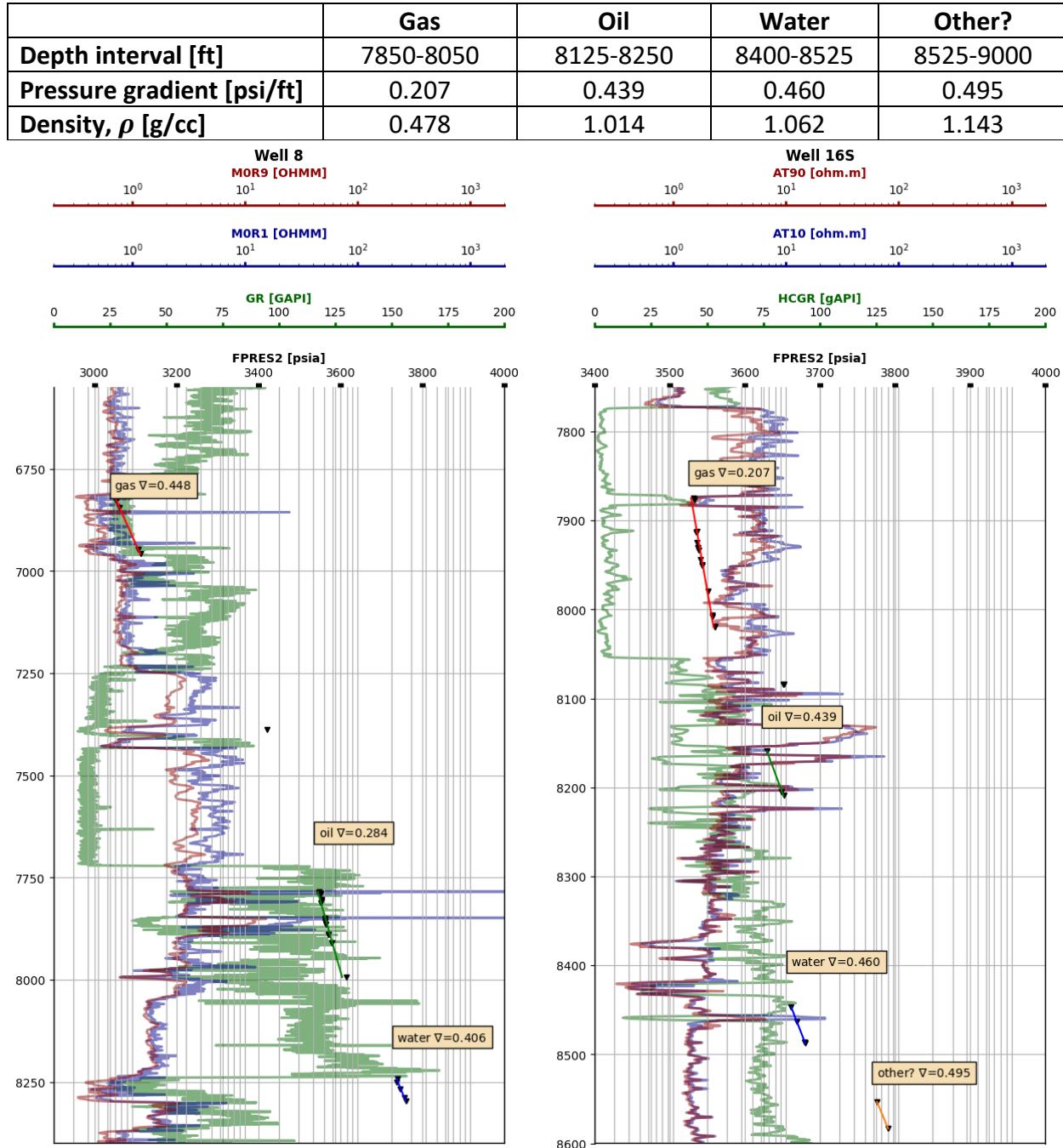


Figure 14: Formation pressure gradients for Well 8 (left) and Well 16S (right).

Firstly, we notice that what we thought of as a gas zone in Well 8 does not match the expected gas gradient. Therefore, this could be another water-bearing zone that is not hydraulically connected to the main reservoir, as the density (1.035 g/cc) matches that of brine. The more likely gas zone is therefore around the interval of 7350-7750 ft.

We begin by constructing the Pickett plot of AT90-vs- $\phi_T$  and highlight the water-saturated zones, as shown in Figure 15. From the trendline, we estimate  $R_w$  for each well. On the other hand, from Archie's formula we can obtain a deep resistivity value of  $R_s$  and sandstone porosity  $\phi_s$ , for each well, and correct for the effect of temperature using a North Sea gradient of  $1.23^{\circ}\text{F}/100\text{ft}$  such that  $R_w(T_2) = R_w(T_1) \frac{T_1+6.77}{T_2+6.77}$ . This helps us obtain another version of  $R_w = \frac{R_s}{\phi^m}$ .

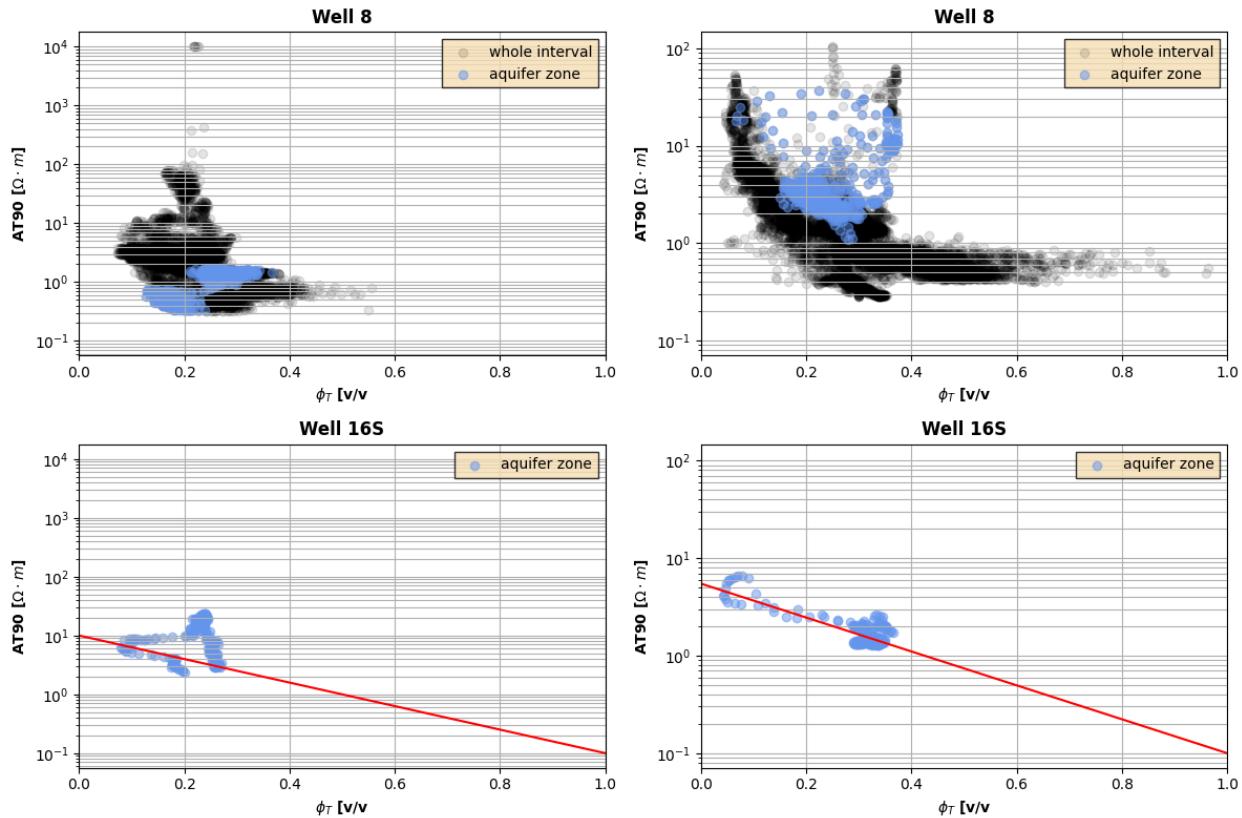


Figure 15: Pickett Plot of Well 8 (left) and Well 16S (right).

The last step in this section is to calculate the  $S_w$  and  $S_{HC}$  as well as the HPV as a continuous log. We use Archie's formula to estimate  $S_w$  given that we are in a shaly-sandstone laminated system. We assume Archie parameters of  $\{a, m, n\} = \{1, 1.8, 1.7\}$  (although a better procedure would be to perform inversion to estimate the proper parameters). The calculation for  $S_w$  is given by:

$$S_w = \left( a \frac{R_w}{R_s \phi_s^m} \right)^{1/n} \quad (5)$$

Moreover, for Well 16S, we apply the parallel- and perpendicular-to-bedding-plane resistivity circuit to estimate the  $S_w$  from Archie's equation. We calculate  $R_s$  as follows:

$$\frac{1}{R_t} = \frac{C_{sh}}{R_{sh}} + \frac{1 - C_{sh}}{R_s} \Rightarrow R_s = \frac{1 - C_{sh}}{\frac{1}{R_t} - \frac{C_{sh}}{R_{sh}}}$$

Figure 16 shows the  $S_w$  and  $S_{HC}$  as well as the HPV for Well 8 and Well 16S, respectively.

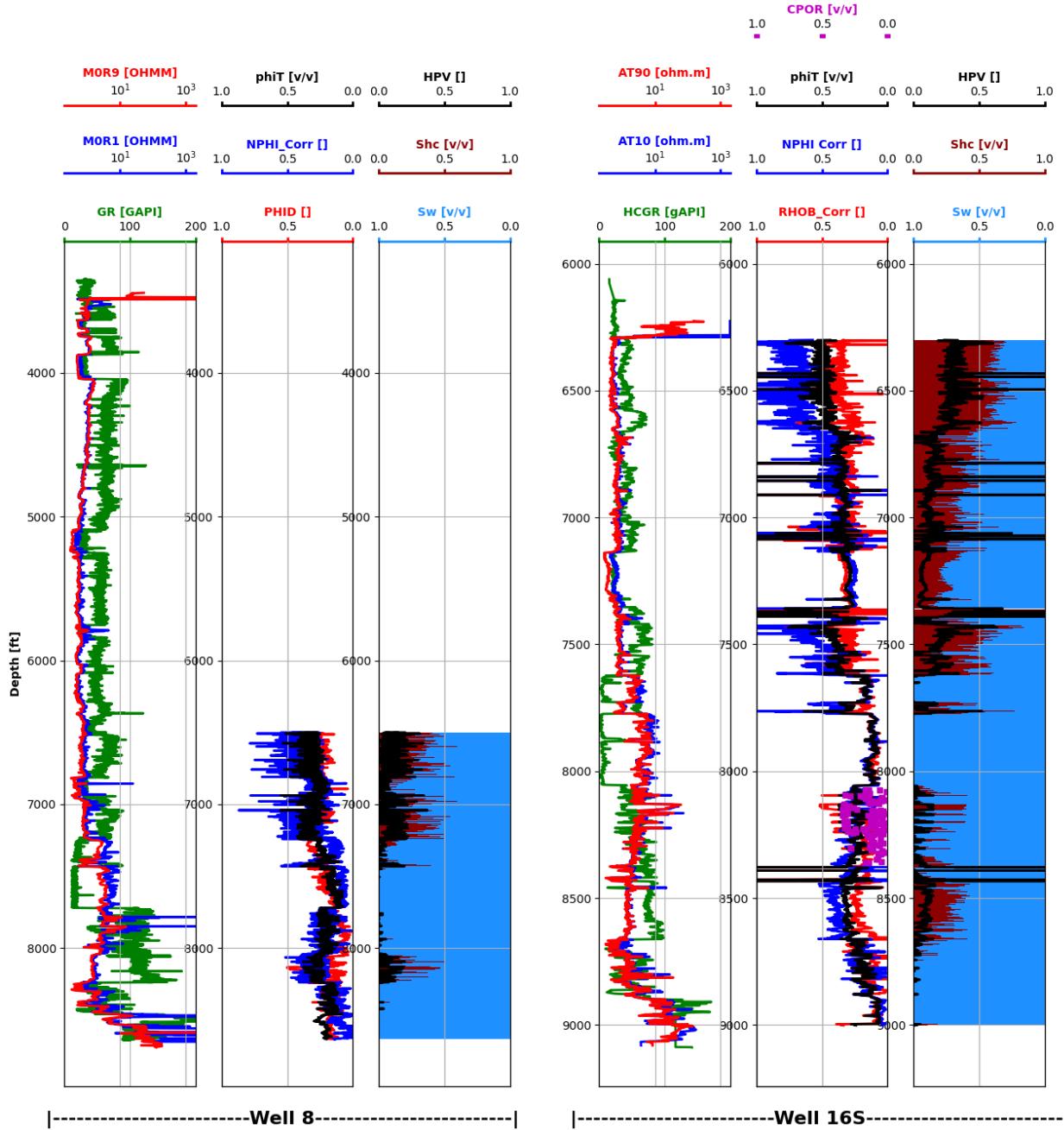


Figure 16:  $Sw$ ,  $Shc$ , and HPV for Well 8 (left) and Well 16S (right).

We also calculate the hydraulic permeability log by co-kriging. First, we calculate the bulk density from grain density values from core data using the equation below:

$$\rho_b = \phi_T(\rho_f + \rho_m) + \rho_m$$

Then, we train the kriging model with input features from the core bulk density calculated from the equation above and core permeability values to predict the logarithm of permeability at log depths. We fit a variogram and select the best model based on the rankings produced. Figure 17 shows the estimated permeability variogram.

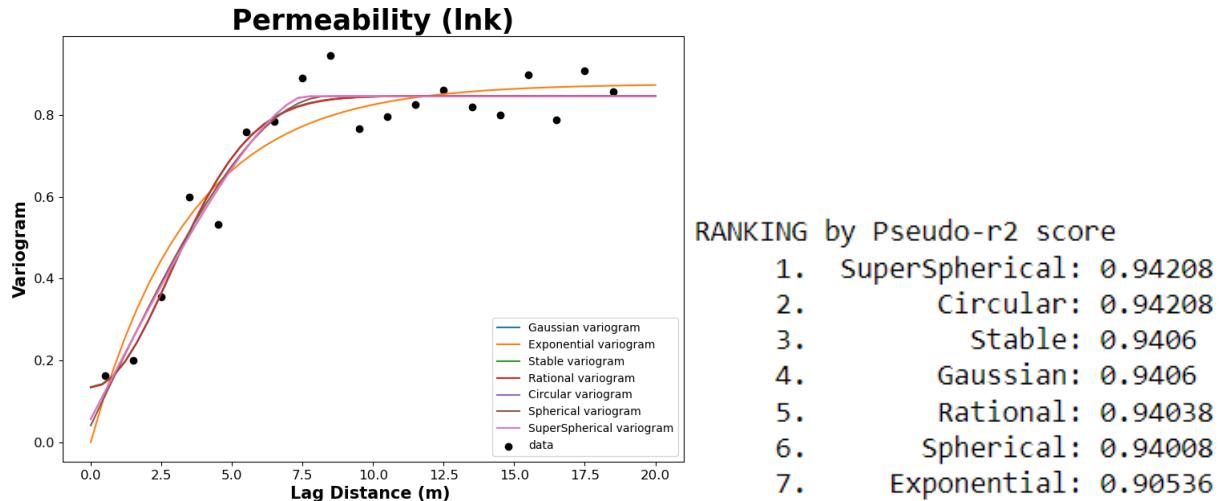


Figure 17: Permeability Variogram.

We also perform a stationarity test by analyzing the autocorrelation and Dickey-Fuller test results, as shown in Figure 18.

```
Dickey-Fuller Test Results:
Test Statistic           -3.9
p-value                  0.0
#Lags Used               1.0
Number of Observations Used 96.0
Critical Value (1%)      -3.5
Critical Value (5%)       -2.9
Critical Value (10%)      -2.6
dtype: float64
```

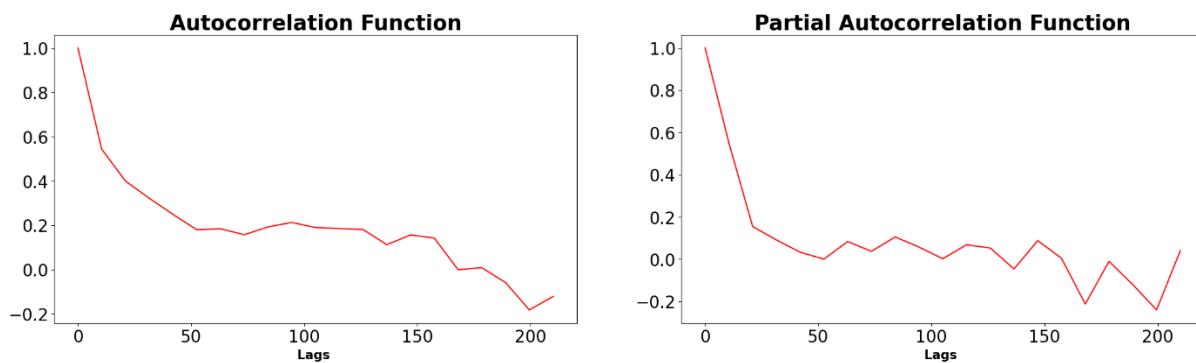


Figure 18: Stationarity and Dickey-Fuller Test.

The test statistic is less than the critical value, therefore the data is stationary. Then, we adopt the co-kriging equation below:

$$Z^*(u_0) = \sum_{a=1}^N \lambda_a Z(u_a) + \lambda_{\gamma_0} Y(u_0),$$

where  $Z^*(u_0)$  is the estimate at the log depths,  $Y(u_0)$  is the value of the secondary variable at the estimation location, and the  $\lambda_a$ 's are the kriging weights. The predicted results are shown in Figure 19.

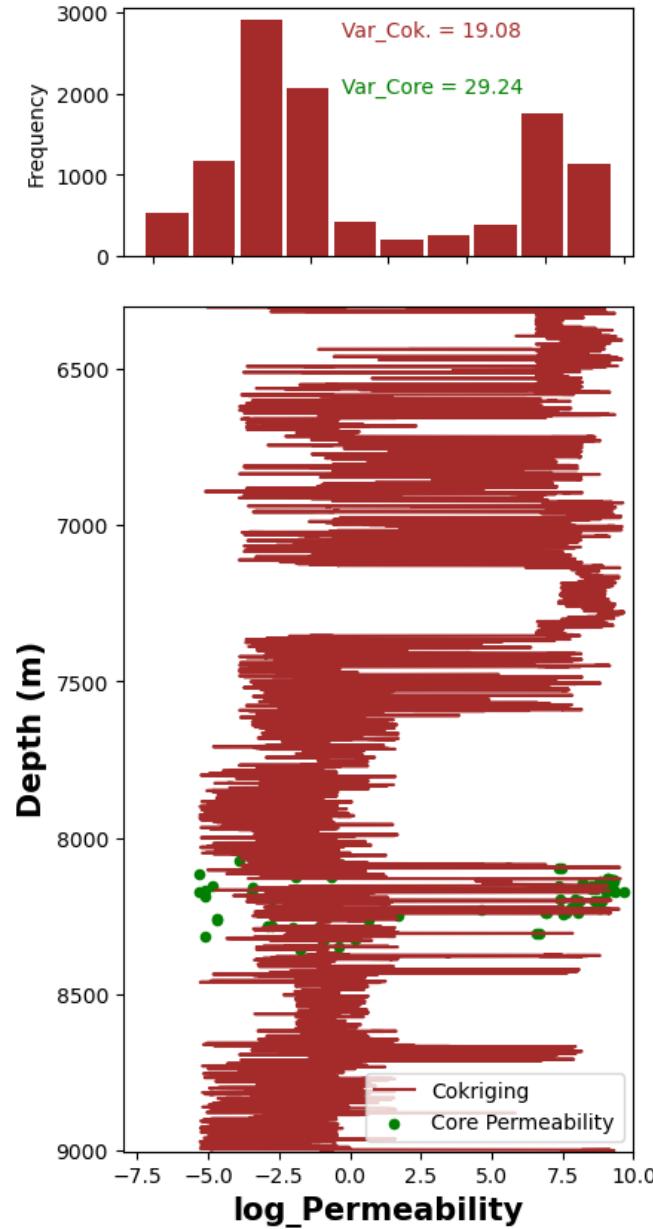


Figure 19: Co-Kriged permeability log.

To ensure that the variance of the core permeability is preserved, we evaluate the prediction results by comparing the rolling statistics of the core and log permeability, as shown in Figure 20.

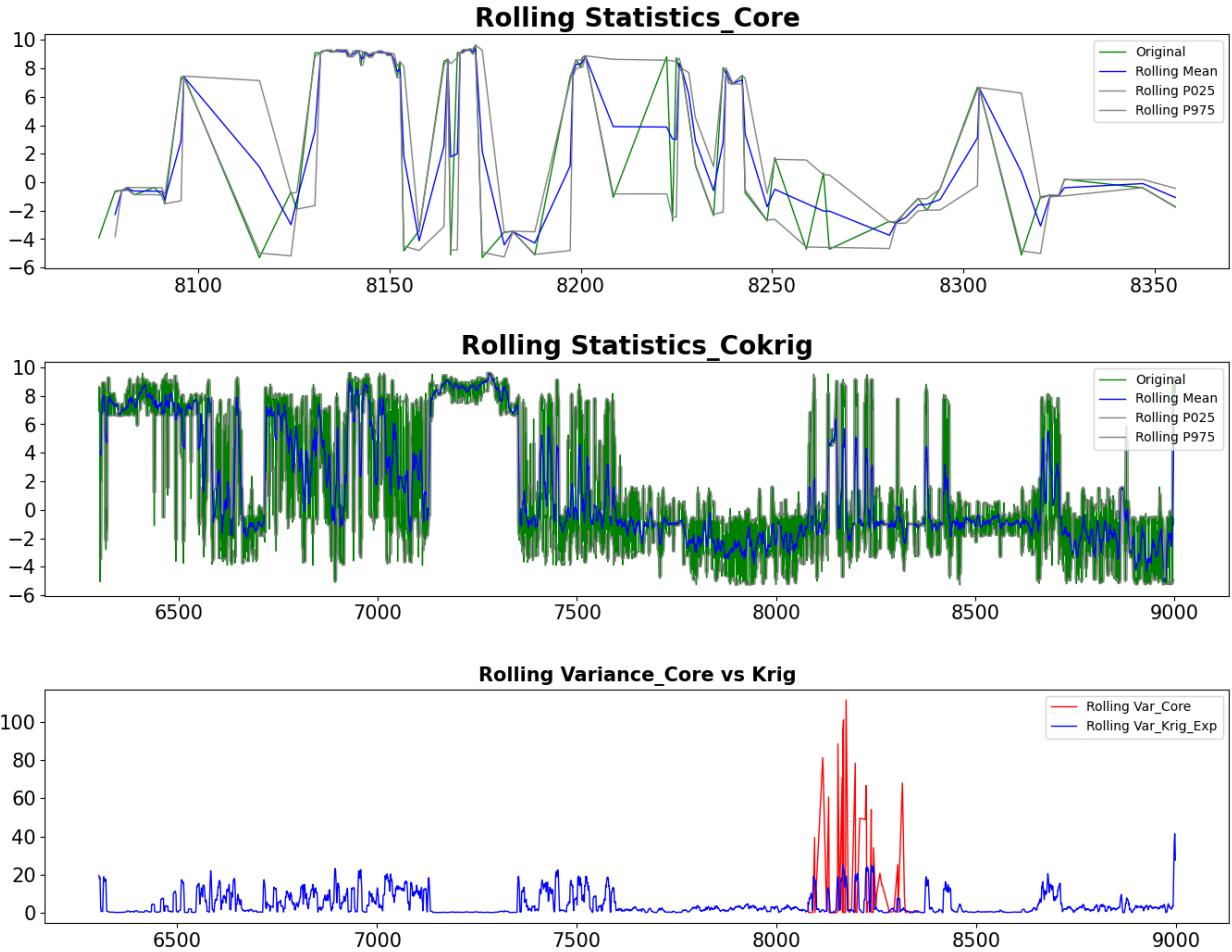


Figure 20: Rolling statistics of co-kriged permeability log.

#### 4. Rock classification based on porosity-permeability relations:

In this section, we implement rock classification based on the Leverett, Winland, and Lorenz techniques to correlate porosity and permeability. This section will only use measurements from Well 16S since it is the only one that contains core data. Figures 21, 22, and 23 show the rock classification based on the porosity-permeability relations for Leverett, Winland, and Lorenz methods, respectively, for Well 16S.

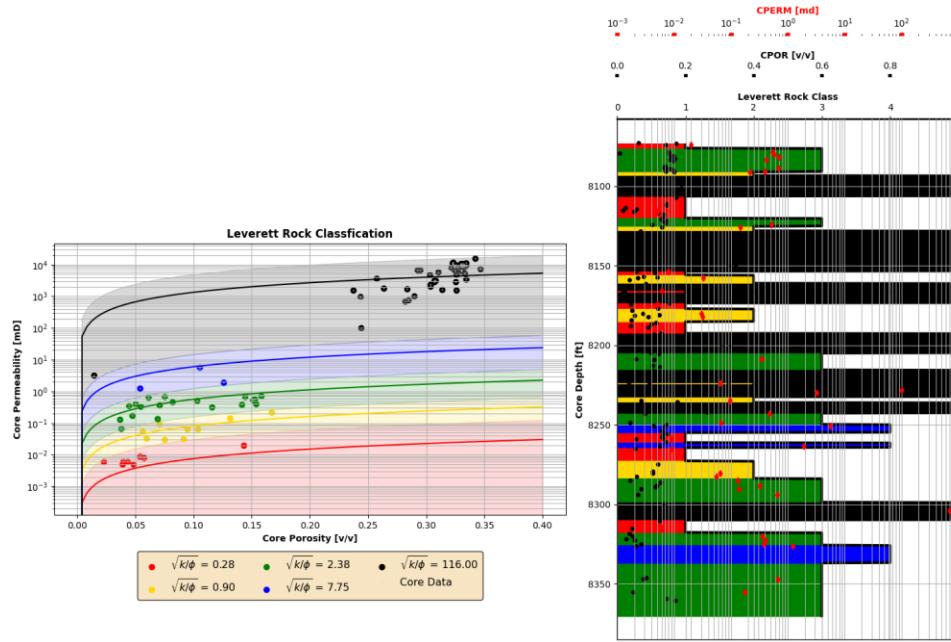


Figure 21: Rock Classification based on Leverett's method for Well 16S.

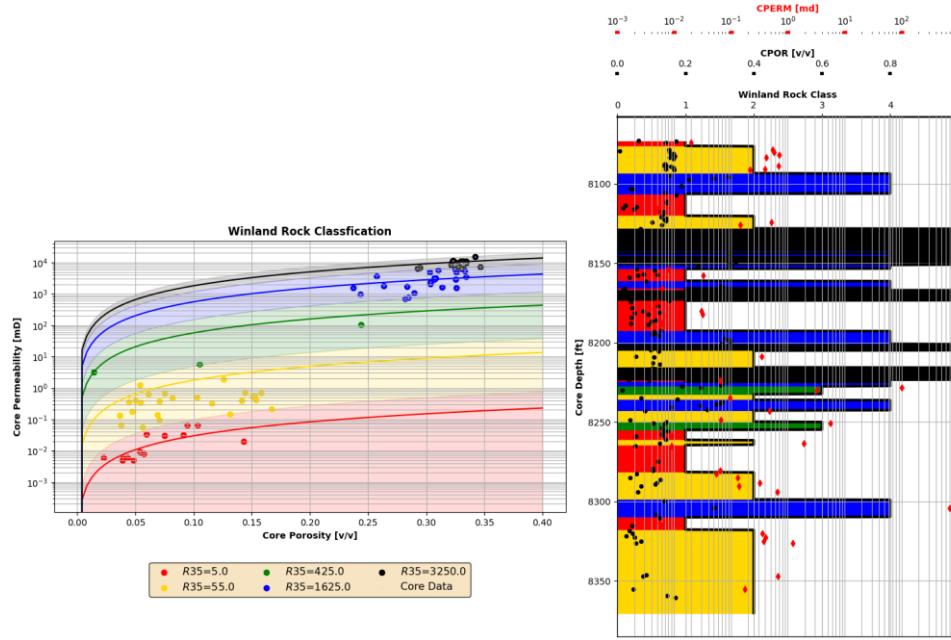


Figure 22: Rock Classification based on Winland's method for Well 16S.

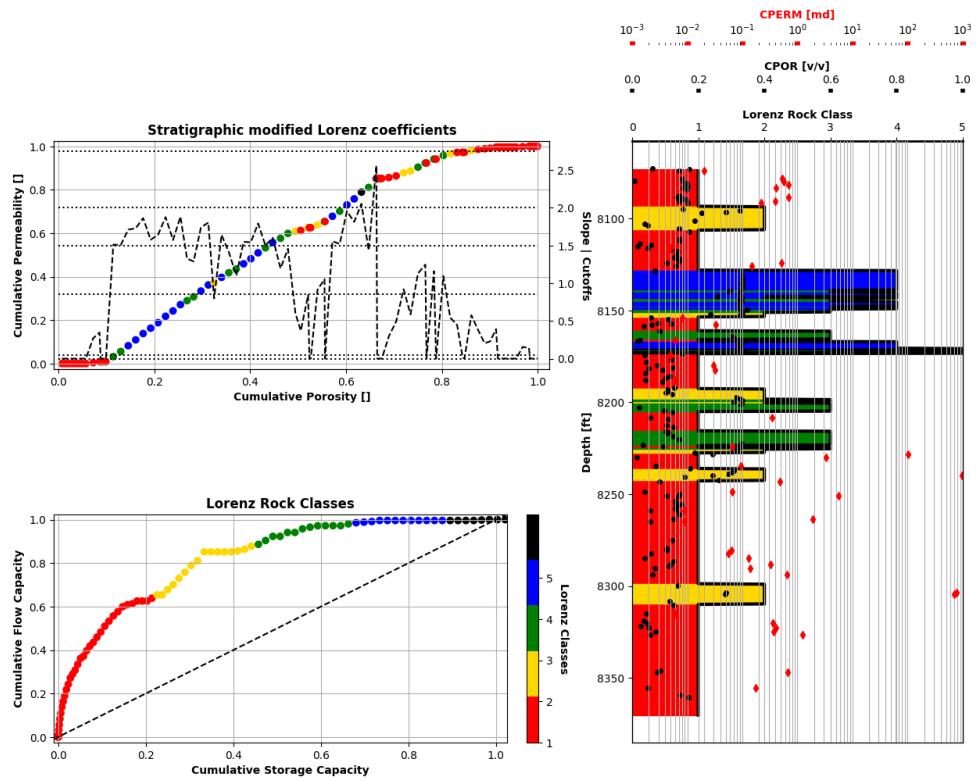


Figure 23: Rock Classification based on Lorenz's method for Well 16S.

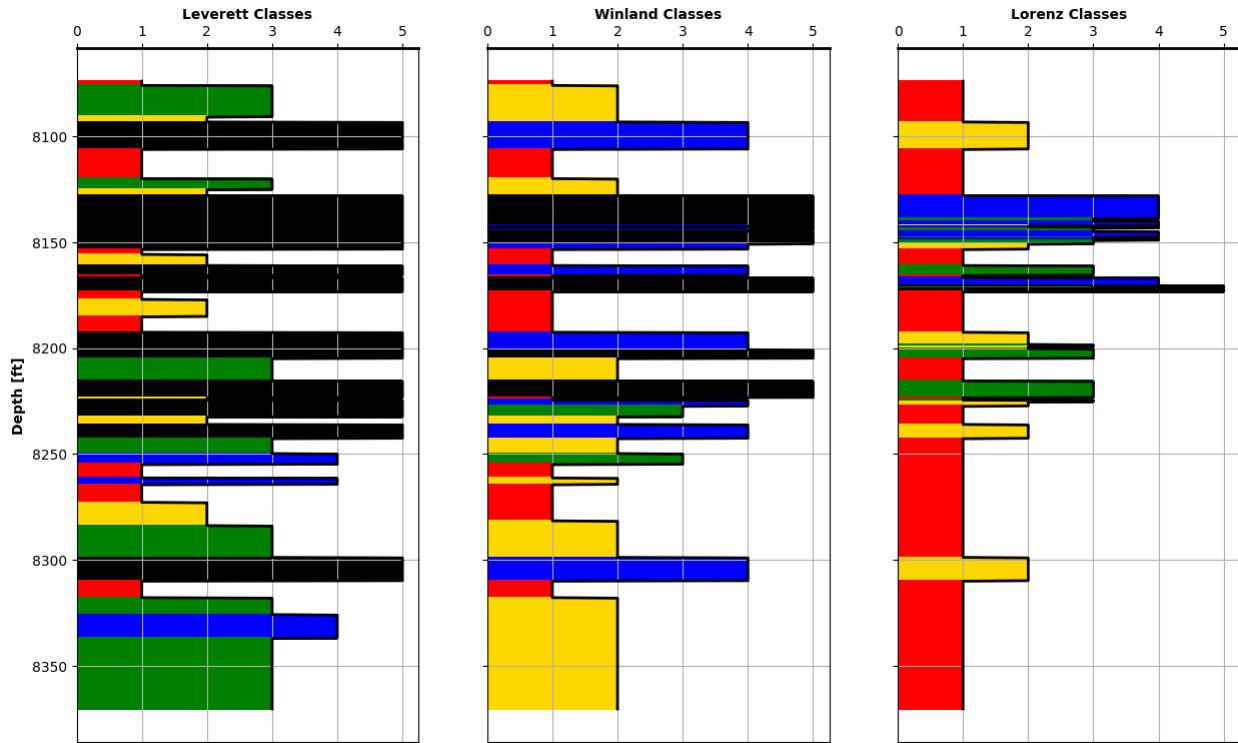


Figure 24: Comparison of rock classes from the 3 techniques for Well 16S.

**5. Quantitative assessment of porosity with adherence to core measurements, Quantitative assessment of water/hydrocarbon saturation with adherence to (a) capillary-pressure and (b) core measurements, and Quantitative assessment of permeability with adherence to core measurements.**

This section will combine the quantitative assessments of porosity, saturation, and permeability with adherence to core measurements and capillary pressure. Recall that core measurements and capillary pressure data is only available for Well 16S, and not Well 8. Figure 25 shows the depth-by-depth comparison of core and log porosity, permeability, and water saturation, as well as the cross-plot of core-vs-log measurements.

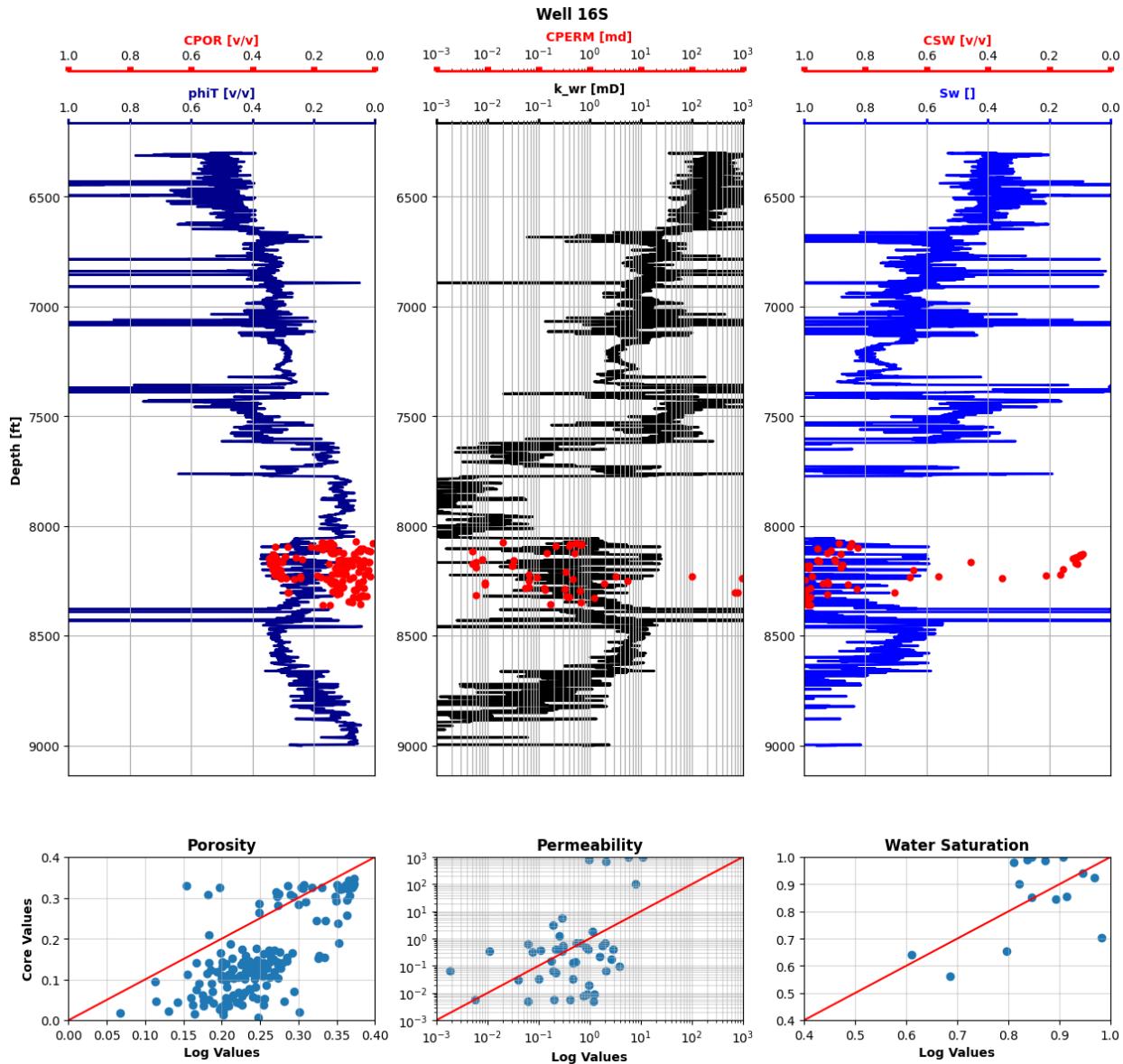


Figure 25: Quantitative assessment of core-vs-log porosity, permeability, and water saturation for Well 16S.

Next, we analyze the capillary pressure data. The normalized pore size distribution shows a peak corresponding to approximately 12.95 microns. Using the capillary pressure data, we perform nonlinear inversion to obtain the optimal parameters for the J-function, as follows:

$$\min_{\theta} \|e(\theta)\|_2^2 + \lambda \|\theta_i\|_2^2,$$

where  $\theta = \{a, b\}$  in the equation  $S_w = aJ^b + S_{w,irr}$ , and  $S_{w,irr} = 0.05$  from the well logs. The mismatch is given by:  $e(\theta) = S_w - (aJ^b + S_{w,irr})$ .

We find a solution at  $\theta^* = \{0.3441, -0.5177\}$ . Alternatively, we can estimate the J-function as follows:  $J = \frac{P_C \sqrt{k/\phi}}{2\sigma \cos(\theta)}$ . Figure 26 shows the results of the capillary pressure analysis using Leverett's J-function.

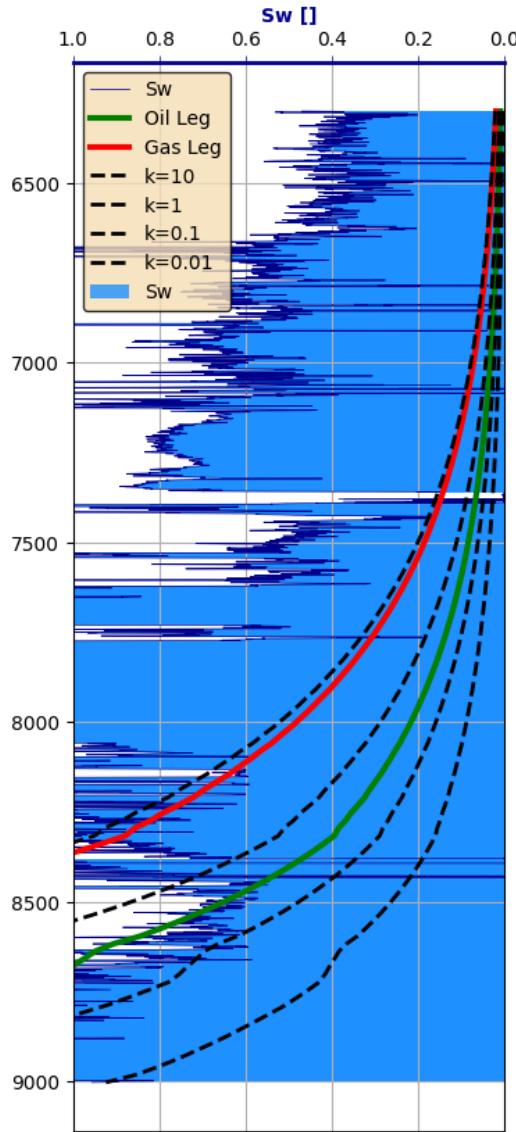


Figure 26: Leverett J-function for Well 16S.

## 6. Well-to-well correlation of lithology, flow units, and saturation fluids, and Sketch of fluid contacts on the well-to-well correlation and structural plots.

This section will combine the correlation of lithology, flow units, and saturation fluids between the two wells with the sketch of the fluid contacts and structural plots.

In order to correlate the two wells optimally, we perform inversion for the mineral volumetric concentrations of each well. We assume only 2 minerals are present, namely quartz and calcite. Table 5 shows the typical parameters for well logs of the pure mineral components:

*Table 5: Typical parameters for well logs.*

Mineral	$\rho$ [g/cc]	PEF [b/e]	$\phi_N$ [LS]	$\Delta t$ [ $\mu\text{s}/\text{ft}$ ]
Quartz	2.65	1.8	-0.04	51.3-55.5
Calcite	2.71	5.1	0.00	47.6

Also, for the fluid components, let  $\rho_f = 1.0 \text{ g/cc}$ ,  $\Delta t_f = 189 \mu\text{s}/\text{ft}$ ,  $\phi_{N,f} = 1.0$ , and  $PEF_f = 0$ , assuming only water is present in the pore space.

The linear inversion procedure then becomes:

$$A = \begin{bmatrix} \rho(m_1) & \rho(m_2) & \rho_f \\ PEF(m_1) & PEF(m_2) & PEF_f \\ \phi_N(m_1) & \phi_N(m_2) & \phi_{N,f} \\ \Delta t(m_1) & \Delta t(m_2) & \Delta t_f \end{bmatrix} \quad x = \begin{bmatrix} c_1 \\ c_2 \\ \phi \end{bmatrix} \quad d = \begin{bmatrix} \rho_b \\ PEF \\ \phi_N \\ \Delta t \end{bmatrix}$$

We can apply a data weighting matrix,  $W_d$ , such that  $W_d(d_i) = \text{diag}(1/d_i)$ . The inversion problem can be formulated as follows:

$$\min_x \|W_d Ax - W_d d\|_2^2 + \lambda \|x\|_2^2 \quad \text{s.t.} \begin{cases} x_i \in [0,1] \\ \sum x_i = 1 \end{cases}$$

We implement a Sequential Least-Squares Quadratic-Programming (SLSQP) algorithm to solve the linear inversion with bounds and constraints. We perform the inversion for both Well 8 and Well 16S. Figures 27 and 28 show the inversion results for Well 8 and Well 16S, respectively.

With the constitutive mineral component depth-by-depth obtained through the inversion, we can now correlate the two wells in terms of lithology, flow units, and saturation fluids, as well as sketching the fluid contacts between the two wells.

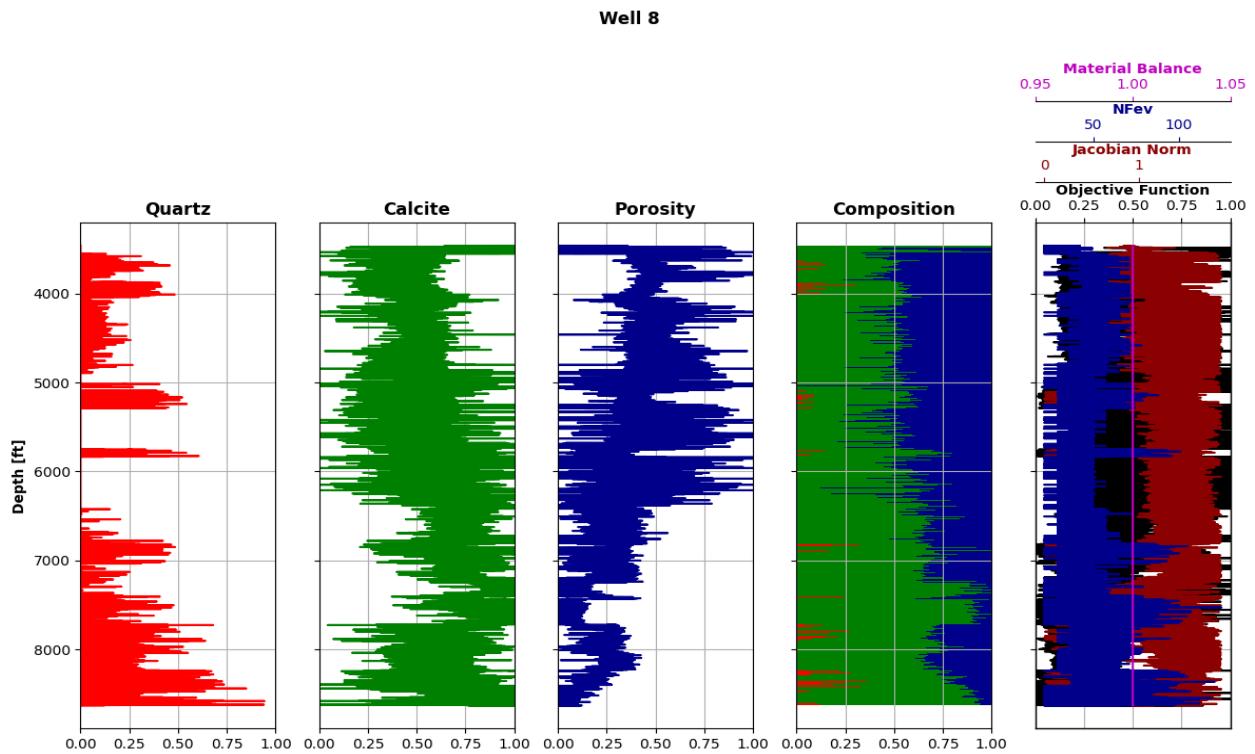


Figure 27: Mineral composition inversion for Well 8.

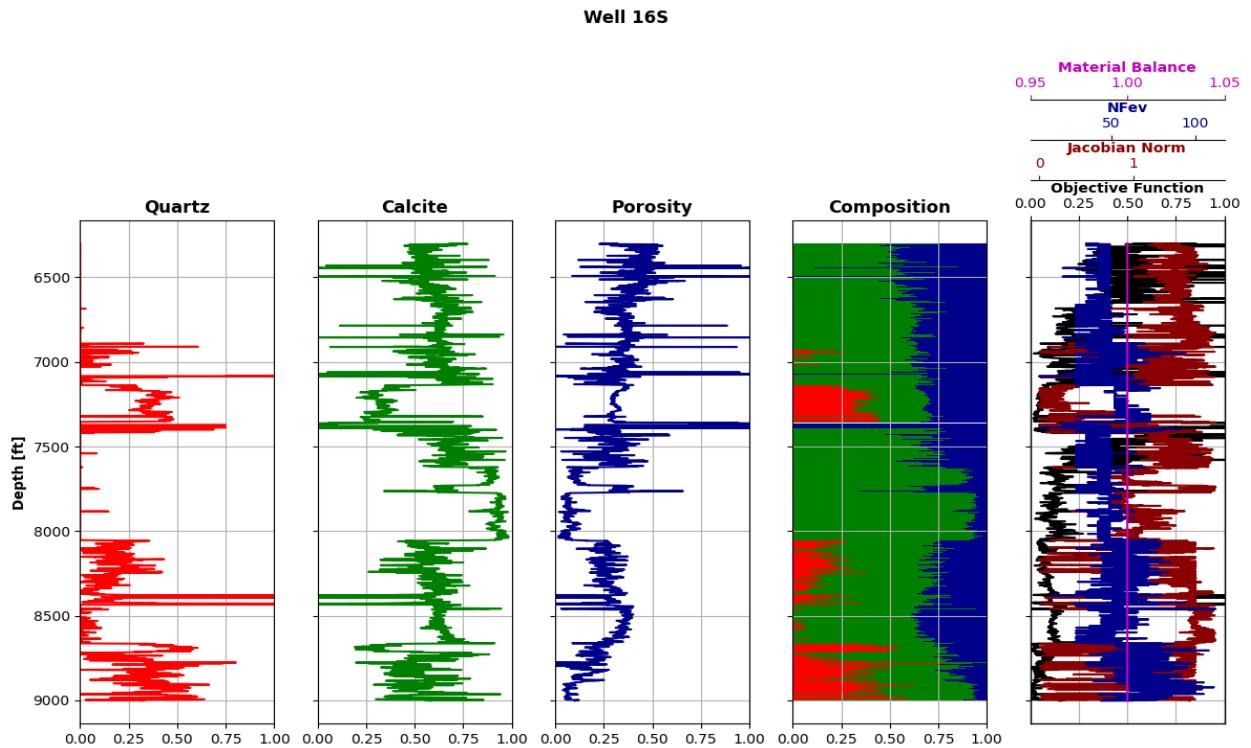


Figure 28: Mineral composition inversion for Well 16S.

To correlate the two wells in terms of lithology (e.g., GR), flow units (e.g.,  $\phi$ ,  $k$ ), saturation fluids and fluid contacts (e.g.,  $S_w$ ), we perform again DTW based on the mineral composition. Figure 29 shows the correlated zones between Well 8 and Well 16S based on the mineral composition estimation.

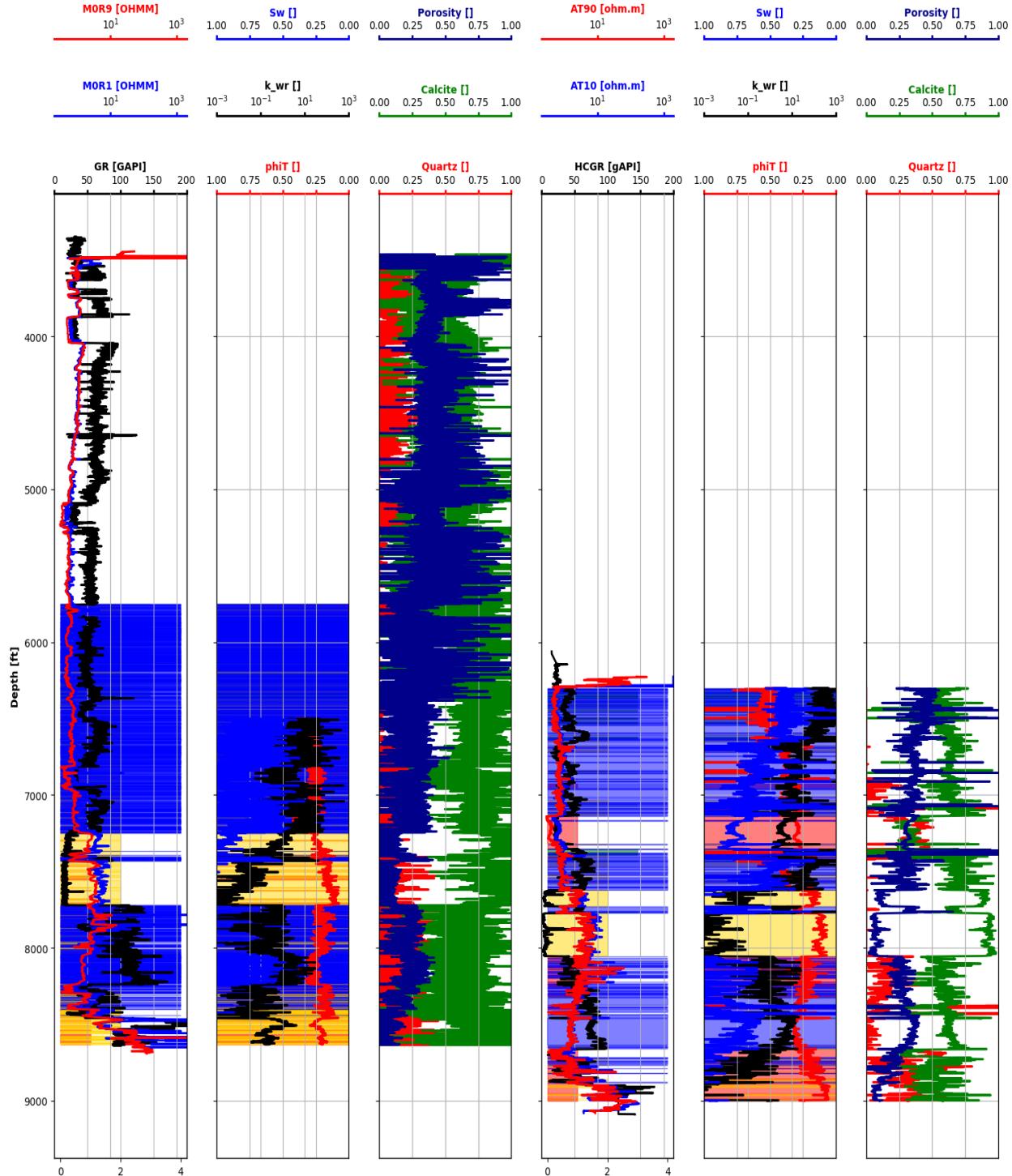


Figure 29: Well-to-well Correlation based on mineral composition estimation.

**7. Assess whether P-wave and/or S-wave impedances could be used to identify lithology, porosity, and fluids. Biot-Gassmann fluid substitution equations and sensitivity analysis of elastic/petrophysical correlations:**

From Wylie's time-average equation, if we know the pore fluid and matrix depth-by-depth, we can estimate porosity from the sonic logs as follows:

$$\Delta t \approx \phi \Delta t_f + (1 - \phi) \Delta t_m; \quad \phi = \frac{\Delta t - \Delta t_m}{\Delta t_f - \Delta t_m}.$$

Alternatively, if we already know the porosity along the well, and assume the matrix sonic slowness, we can reformulate Wylie's time-average equation to estimate the sonic slowness of the pore fluids and help us understand what the pore fluid depth-by-depth is. Similarly, we can also reformulate to estimate the matrix slowness as follows:

$$\Delta t_f = \Delta t_m + \frac{1}{\phi} (\Delta t - \Delta t_m); \quad \Delta t_m = \frac{\Delta t - \phi \Delta t_f}{1 - \phi}$$

Figure 30 shows the comparison of the estimated porosity from the sonic log, and the estimated porosity based on the mineral composition inversion, as well as the pore fluids and matrix composition based on the sonic logs.

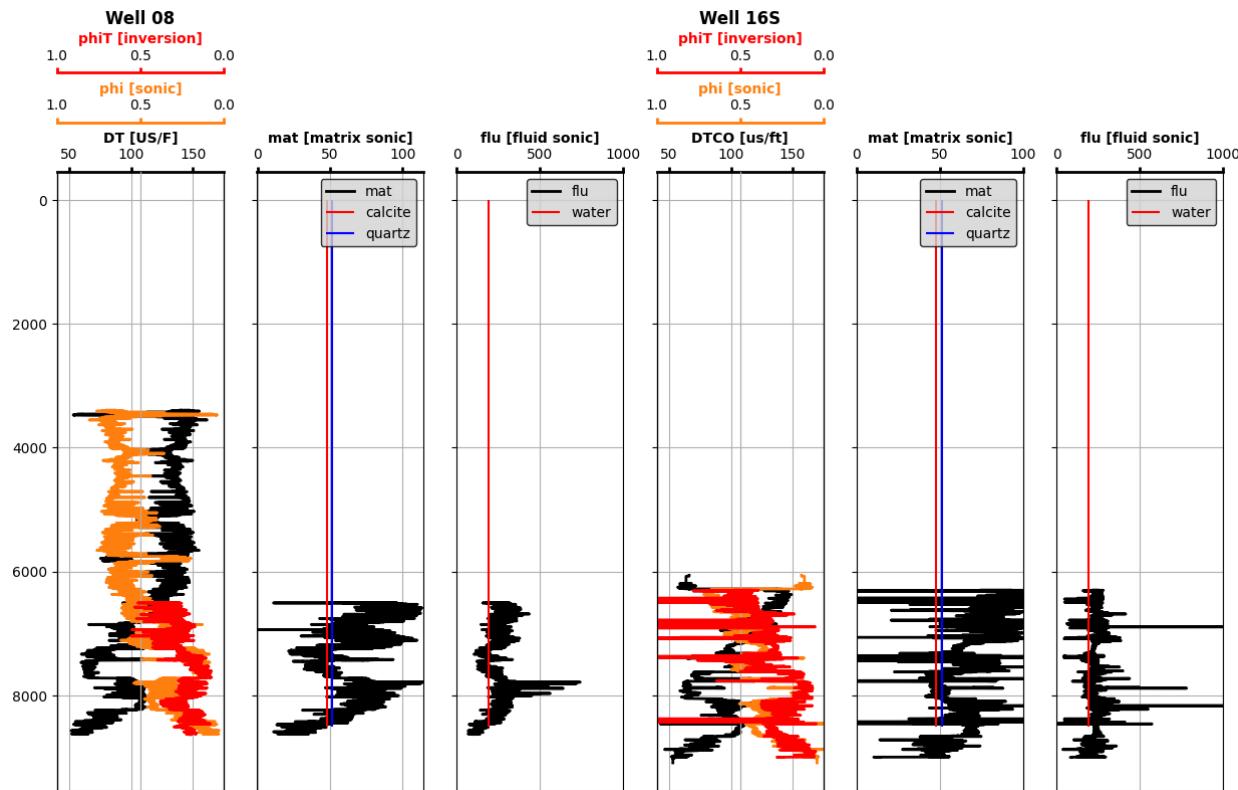


Figure 30: Sonic measurements for Well 8 (left) and Well 16S (right).

Finally, we perform Biot-Gassmann fluid substitution equations and sensitivity analysis of the elastic correlations. For Biot-Gassmann, we have:

$$V_s = \sqrt{\mu/\rho} ; \quad V_p = \sqrt{\frac{K + \frac{4}{3}\mu}{\rho}}$$

Where  $\mu$  is the shear modulus of the “skeleton” or grain matrix,  $\rho$  is the bulk density, and  $K$  is the bulk modulus. We can obtain  $\rho$  either by reading the bulk density log, or by summing the mineral constituents such that:

$$\rho_b = \phi\rho_f + c_1\rho_1 + \cdots + c_N\rho_N.$$

Moreover, we can estimate the bulk modulus using a system of equations such that:

$$\begin{cases} \frac{1}{K} = \frac{1-\phi}{K_m} + \frac{\phi}{K_f} \\ K = \phi K_f + (1-\phi)K_m \end{cases},$$

where  $\frac{1}{K_f} = \frac{S_w}{K_w} + \frac{1-S_w}{K_H}$ . We obtain the P-wave and S-wave velocities as shown in Figure 31.

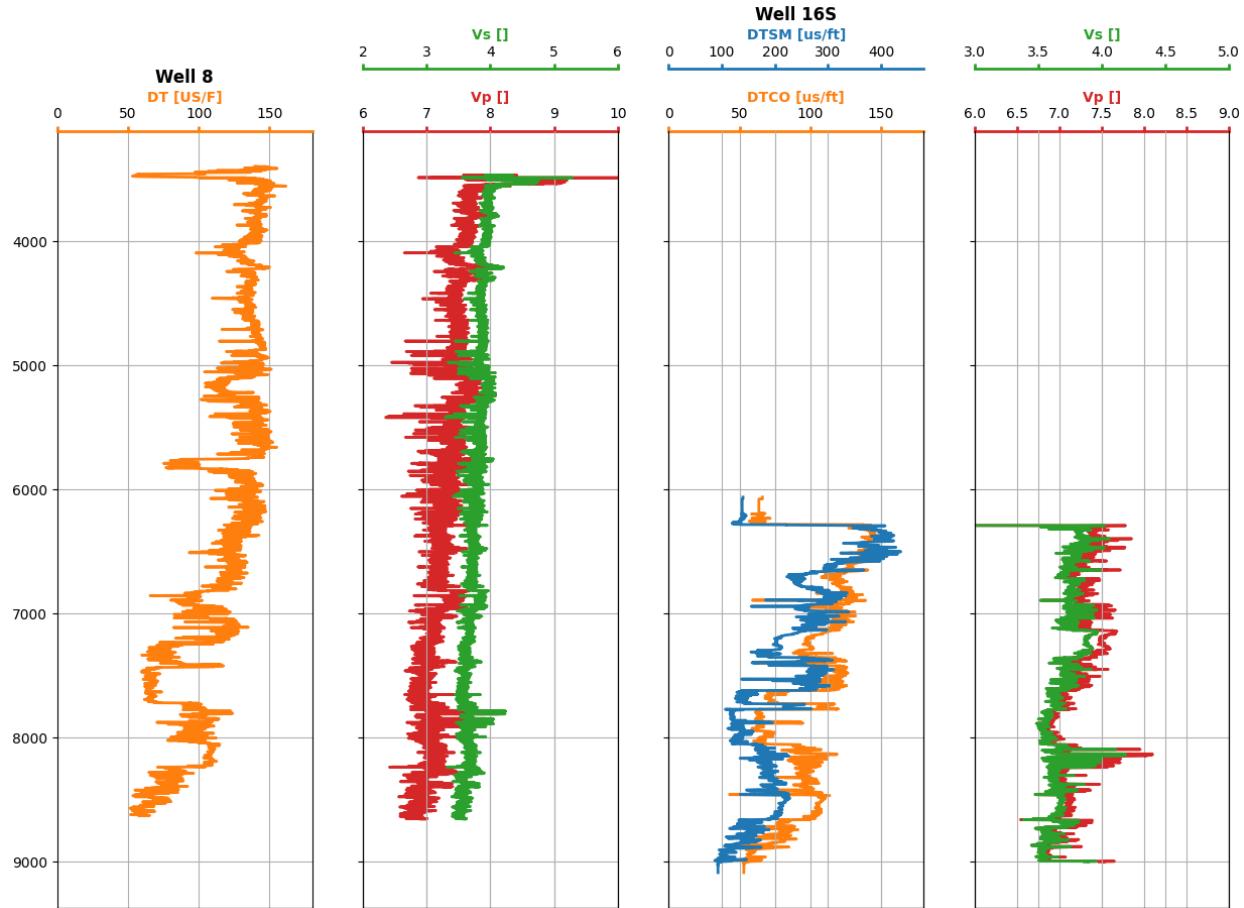


Figure 31: Biot-Gassmann for Well 8 (left) and Well 16S (right).

## **Conclusions and Recommendations:**

We perform a comprehensive petrophysical analysis for a set of 2 wells from a North Sea field.

The analysis began with an initial visualization and naïve interpretation of the well logs, and quality control and cross-plots to ensure appropriate measurements and interpretation. To make sure that we are performing the correct interpretation, we separate the wells by zones using an automatic zonation algorithm based on Dynamic Time Warping and K-Means Clustering.

Next, we performed a comprehensive analysis for lithology, sedimentary environment, and identification of porous and permeable units and fluids saturating the porous units. We estimate the volumetric concentration of shale using the linear formulation, as well as solving the nonlinear inversion problem for the parallel- and perpendicular-to-bedding-plane resistivities. We correct the neutron and density logs to calculate a continuous porosity log, and then estimate the hydraulic permeability log by applying the Wylie-Rose correlation and by co-kriging the bulk density and core measurements. We confirm the pore fluids by computing the fluid gradients from pore pressure data, and estimate  $S_w$ ,  $S_{HC}$ , and HPV using Archie's formulae.

We perform rock classification by implementing Leverett's, Winland's, and Lorenz's techniques on the core measurements, and obtain a continuous rock class log. We then compare the results of porosity and permeability between the estimated logs and core measurements and calculate Leverett's J-function based on the rock classes.

We perform well-to-well correlation of lithology, flow units, and saturation fluids via inversion of mineral composition in both wells. We use the bulk density, photoelectric factor, neutron porosity, and sonic logs to estimate the concentration of calcite and quartz as well as porosity. Using a bounded and constrained inversion technique, we obtain main zones that correlate in both wells. Finally, we use the sonic logs to identify lithology, porosity, and fluids in the wells, and perform Biot-Gassmann fluid substitution to obtain the compressional and shear wave velocities.

In conclusion, we present a robust and thorough workflow for petrophysical analysis and interpretation of a multi-well project. We thoroughly calculate the estimated rock properties based on the log, core and formation pressure data, as well as the capillary pressure measurements.

We recommend cropping the wells to the interval of interest instead of working with the entire depth interval, as well as calculating the uncertainty and sensitivity of the estimated measurements. We also recommend performing a more robust shaly-sandstone analysis for the wells, given that there is a potential mixed laminated-dispersed shaly-sandstone sequence. Moreover, we recommend confirming the results with further capillary pressure data at different core sample depths, and to potentially correlate the multi-well project to seismic data in order to gain spatial context.

```

import numpy as np
import pandas as pd

import lasio
import numdifftools as nd
from rockphypy import Fluid
from dtw import dtw, rabinerJuangStepPattern
from scipy import linalg, optimize, stats

import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from matplotlib.ticker import FixedLocator
from matplotlib.colors import LinearSegmentedColormap, ListedColormap, Normalize

from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import make_pipeline

class Petrophysics:
    def __init__(self):
        self.save_plot = True
        self.return_d = False
        self.logs_8 = self.read_logs_8('25_10-8.las')
        self.fpres_8 = lasio.read('25_10-8_FormPres.las')
        self.logs_16 = lasio.read('25_10-16_S.LAS')
        self.core_16 = lasio.read('25_10-16_S_Core.las')
        self.fpres_16 = lasio.read('25_10-16_S_FormPres.las')
        self.nmr_16 = self.read_nmr_16('25_10-16_S_NMR.LAS')
        self.pc_data = pd.read_excel('PC_data.xlsx', sheet_name='Sheet1')
        self.mask_i_8 = [3490, 8465]
        self.mask_i_16 = [6290, 8990]
        self.sand_pt = 7203.0
        self.shale_pt = 7372.0
        self.lime_pt = 7830.0
        self.rock_cls = None
        self.corr_vals = self.read_corrected_values()
        self.zones08, self.zones16 = self.make_zones()

    def make_zones(self):
        w16_z1 = {'Top': 6303.00, 'Bottom': 7134.41, 'Mask': [6303.00, 7134.41], 'Hatch': '--',
        'Color': '#bebebe', 'Lith': 1, 'Name': 'shale'}
        w16_z2 = {'Top': 7134.41, 'Bottom': 7351.03, 'Mask': [7134.41, 7351.03], 'Hatch': '..',
        'Color': '#ffff00', 'Lith': 0, 'Name': 'sandstone'}
        w16_z3 = {'Top': 7351.03, 'Bottom': 7625.70, 'Mask': [7351.03, 7625.70], 'Hatch': '--',
        'Color': '#bebebe', 'Lith': 1, 'Name': 'shale'}
        w16_z4 = {'Top': 7625.70, 'Bottom': 7727.81, 'Mask': [7625.70, 7727.81], 'Hatch': '+',
        'Color': '#80ffff', 'Lith': 2, 'Name': 'limestone'}
        w16_z5 = {'Top': 7727.81, 'Bottom': 7773.59, 'Mask': [7727.81, 7773.59], 'Hatch': '--',
        'Color': '#bebebe', 'Lith': 1, 'Name': 'shale'}
        w16_z6 = {'Top': 7773.59, 'Bottom': 8054.12, 'Mask': [7773.59, 8054.12], 'Hatch': '+',
        'Color': '#80ffff', 'Lith': 2, 'Name': 'limestone'}
        w16_z7 = {'Top': 8054.12, 'Bottom': 9000.00, 'Mask': [8054.12, 9000.00], 'Hatch': '-/',
        'Color': '#7fcf00', 'Lith': 3, 'Name': 'shaly limestone'}
        zones16 = pd.DataFrame([w16_z1, w16_z2, w16_z3, w16_z4, w16_z5, w16_z6, w16_z7])

        w08_z1 = {'Top': 5856.00, 'Bottom': 6813.86, 'Mask': [5856.00, 6813.86], 'Hatch': '--',
        'Color': '#bebebe', 'Lith': 1, 'Name': 'shale'}
        w08_z2 = {'Top': 6813.86, 'Bottom': 6961.09, 'Mask': [6813.86, 6961.09], 'Hatch': '..',
        'Color': '#ffff00', 'Lith': 0, 'Name': 'sandstone'}
        w08_z3 = {'Top': 6961.09, 'Bottom': 7004.01, 'Mask': [6961.09, 7004.01], 'Hatch': '--',
        'Color': '#bebebe', 'Lith': 1, 'Name': 'shale'}
        w08_z4 = {'Top': 7004.01, 'Bottom': 7034.97, 'Mask': [7004.01, 7034.97], 'Hatch': '+',
        'Color': '#ffff00', 'Lith': 0, 'Name': 'sandstone'}
        w08_z5 = {'Top': 7034.97, 'Bottom': 7248.64, 'Mask': [7034.97, 7248.64], 'Hatch': '--',
        'Color': '#bebebe', 'Lith': 1, 'Name': 'shale'}
        w08_z6 = {'Top': 7248.64, 'Bottom': 7400.99, 'Mask': [7248.64, 7400.99], 'Hatch': '+',
        'Color': '#80ffff', 'Lith': 2, 'Name': 'limestone'}
        w08_z7 = {'Top': 7400.99, 'Bottom': 7432.10, 'Mask': [7400.99, 7432.10], 'Hatch': '--',
        'Color': '#bebebe', 'Lith': 1, 'Name': 'shale'}
        w08_z8 = {'Top': 7432.10, 'Bottom': 7721.20, 'Mask': [7432.10, 7721.20], 'Hatch': '+',
        'Color': '#80ffff', 'Lith': 2, 'Name': 'limestone'}
```

```

w08_z9 = {'Top': 7721.20, 'Bottom': 8640.42, 'Mask': [7721.20, 8640.42], 'Hatch': '-/'}
'Color': '#7cf000', 'Lith': 3, 'Name': 'shaly limestone'}
zones08 = pd.DataFrame([w08_z1, w08_z2, w08_z3, w08_z4, w08_z5, w08_z6, w08_z7, w08_z8, w08_z9])
return zones08, zones16

def read_corrected_values(self, name='DEPTH'):
    w08_neutron = pd.read_excel('Bothwells.xlsx', sheet_name='10-8').set_index('DEPTH (ft)', inplace=False).rename_axis(index={'DEPTH (ft)':name})
    w16_neutron = pd.read_excel('Bothwells.xlsx', sheet_name='10-16').set_index('DEPTH (ft)', inplace=False).rename_axis(index={'DEPTH (ft)':name})
    w08_calcvalues = pd.read_excel('calcvalues_well8.xlsx').set_index('Depth ', inplace=False).rename_axis(index={'Depth ':name})
    w16_calcvalues = pd.read_excel('calcvalues.xlsx', sheet_name='Sheet1').set_index('Depth ', inplace=False).rename_axis(index={'Depth ':name})
    corr_vals = {'w08_neutron': w08_neutron, 'w16_neutron': w16_neutron, 'w08_calcvalues': w08_calcvalues, 'w16_calcvalues': w16_calcvalues}
    return corr_vals

def read_logs_8(self, filename='25_10-8.las'):
    logs_8_df = pd.read_csv(filename, skiprows=84, sep=', ', engine='python')
    logs_8_df = logs_8_df.rename(columns={'#DEPTH':'DEPTH'}, inplace=False).replace(-999,np.nan).replace(-9.99,np.nan)
    logs_8_fmtns = pd.read_csv(filename, skiprows=50, nrows=11, names=['Top', 'Bottom', 'Name'], usecols=[0,1,2])
    logs_8_units = {}
    with open(filename, 'r') as file:
        for _ in range(65):
            next(file)
        for _ in range(83-65):
            line = next(file)
            columns = line.split()
            if len(columns) >= 1:
                mnemonic = columns[0]
                units = columns[1]
                logs_8_units[mnemonic] = units.split('.')[1]
    logs_8_units = np.array(list(logs_8_units.values()))
    logs_8_header = pd.read_csv(filename, skiprows=10, nrows=32, names=['MNEM', 'DATA', 'DESCRIPTION'], usecols=[0,1,2])
    self.well_8 = {'DATA':logs_8_df, 'UNITS':logs_8_units, 'FMTNS':logs_8_fmtns, 'HEADER':logs_8_header}
    las = lasio.LASFile()
    for i, data in enumerate(self.well_8['DATA'].columns):
        las.append_curve(data, self.well_8['DATA'][data], unit=self.well_8['UNITS'][i])
    for mnem in las.header['Well'].keys():
        name = [item.strip() for item in self.well_8['HEADER'][mnem].values]
        data = [item.strip() for item in self.well_8['HEADER'][data].values]
        if mnem in name:
            las.header['Well'][mnem] = data[name.index(mnem)]
    las.header['Well']['STRT'].unit = 'ft'
    las.header['Well']['STOP'].unit = 'ft'
    return las

def read_nmr_16(self, filename='25_10-16_S_NMR.LAS'):
    self.nmr_16_header = pd.read_csv(filename, skiprows=10, nrows=37, names=['MNEM', 'DATA', 'DESCRIPTION'], usecols=[0,1,2])
    self.nmr_16_df = pd.read_csv(filename, skiprows=182, sep=', ', engine='python')
    self.nmr_16_df = self.nmr_16_df.rename(columns={'#DEPTH':'DEPTH'}, inplace=False).replace(-999,np.nan)
    las = lasio.LASFile()
    for c in self.nmr_16_df.columns:
        las.append_curve(c, self.nmr_16_df[c])
    return las

def plot_title(self, fig, df, xloc=0.5, yloc=0.05):
    fld = df.header['Well']['FLD'].value
    wll = df.header['Well']['WELL'].value
    com = df.header['Well']['COMP'].value
    fig.text(xloc, yloc, '{} | {} | {}'.format(fld, com, wll), weight='bold', ha='center', va='center', fontsize=14)
    return None

def plot_formations(self, ax, df, lw=1, alpha=0.5, bounds=[0,1], cmap='tab20', align:str='center', triangle:bool=False):
    for i in range(len(df)):
        data = df.iloc[i]
        top, bot, name = data['Top'], data['Bottom'], data['Name']
        my_hatch = data['Hatch'] if 'Hatch' in df.columns else None

```

```

my_edge = 'gray' if 'Hatch' in df.columns else None
my_color = data['Color'] if 'Color' in df.columns else mpl.colormaps[cmap](i)
if triangle:
    ax.fill_betweenx([top,bot], bounds, color=my_color, hatch=my_hatch, edgecolor=my_edge,
lw=lw, alpha=alpha)
    ax.text(np.mean(bounds), bot-30, name, ha=align, va='center')
else:
    ax.fill_betweenx([top,bot], bounds[0], bounds[1], color=my_color, hatch=my_hatch,
edgecolor=my_edge, lw=lw, alpha=alpha)
    ax.text(np.mean(bounds), np.mean([top,bot]), name, ha=align, va='center')
ax.set_xlim(bounds[0],bounds[1])
my_label = 'Lithology' if 'Lith' in df.columns else 'Formations'
ax.set_xlabel(my_label, weight='bold')
ax.xaxis.set_label_position('top'); ax.xaxis.set_ticks_position('top')
ax.spines['top'].set_linewidth(2); ax.set_xticks([])
return None

def plot_nmrt2(self, ax, df, sampling=75, cutoff=0.25, vscale=200, tmin=0.3, tmax=3000,
               start=None, stop=None, color='darkgreen', alpha=0.5, lw=0.3):
    if start is None:
        start = 6000
    if stop is None:
        stop = self.logs_16.header['Well']['STOP'].value + 50
    t_range = np.log10(tmax) - np.log10(tmin)
    t2_data = np.nan_to_num(np.array(df.df()), nan=0)
    t2_depth, t2_dist = df.index[::-sampling], t2_data[::-sampling, 1:]
    t2_norm = (t2_dist.max() - t2_dist.min())
    bins = np.arange(t2_dist.shape[1])
    bins_time = 10**((bins*t_range/(len(bins)-1))+np.log10(tmin))
    for i in range(len(t2_depth)):
        x = vscale * t2_dist[i] / t2_norm
        ax.semilogx(bins_time, t2_depth[i]-x, color='k', linewidth=lw, alpha=alpha)
        ax.fill_between(bins_time, t2_depth[i]-x, t2_depth[i]-cutoff,
                        where = t2_depth[i]-x < t2_depth[i]-cutoff,
                        color=color, alpha=alpha)
    ax.axvline(3, linestyle='--', c='red', label='3$ms$'); ax.axvline(33, linestyle='-', c='red',
label='33$ms$')
    ax.legend(loc='upper right', facecolor='lightgray', edgecolor='k', fancybox=False)
    ax.set_xlabel('T2_DIST_MRF [ms]', color=color, weight='bold')
    ax.xaxis.set_label_position('top'); ax.xaxis.set_ticks_position('top')
    ax.spines['top'].set_edgecolor(color); ax.spines['top'].set_linewidth(2)
    ax.grid(True, which='both', axis='x')
    ax.set_xlim(bins_time.min(), bins_time.max())
    ax.set_ylim(start, stop)
    return None

def plot_curve(self, ax, df, curve, lb=None, ub=None, color='k', pad=0, s=2, mult=1,
               units:str=None, mask=None, offset:int=0, title:str=None, label:str=None,
               semilog:bool=False, bar:bool=False, fill:bool=None, rightfill:bool=False,
               marker=None, edgecolor=None, ls=None, alpha=None):
    if mask is None:
        x, y = -offset+mult*df[curve], df.index
    else:
        x, y = -offset+mult*df[curve][mask], df.index[mask]
    lb = x[~np.isnan(x)].min() if lb is None else lb
    ub = x[~np.isnan(x)].max() if ub is None else ub
    if semilog:
        ax.semilogx(x, y, c=color, label=curve, alpha=alpha,
                    marker=marker, markersize=s, markeredgecolor=edgecolor, linestyle=ls, linewidth=s)
    else:
        if bar:
            ax.barr(y, x, color=color, label=curve, alpha=alpha)
        else:
            ax.plot(x, y, c=color, label=curve, alpha=alpha,
                    marker=marker, markersize=s, markeredgecolor=edgecolor, linewidth=s, linestyle=ls)
    if fill:
        ax.fill_betweenx(y, x, ub, alpha=alpha, color=color) if rightfill else ax.fill_betweenx(y, lb,
x, alpha=alpha, color=color)
    if units is None:
        if hasattr(df, 'curvesdict'):
            units = df.curvesdict[curve].unit
        else:
            units = ''
    ax.set_xlim(lb, ub)
    ax.grid(True, which='both')
    ax.set_title(title, weight='bold') if title != None else None
    xlabel = label if label is not None else curve

```

```

if offset != 0:
    ax.set_xlabel('{} [{}]\nwith {} offset'.format(xlab, units, offset), color=color, weight='bold')
else:
    ax.set_xlabel('{} [{}]\n'.format(xlab, units), color=color, weight='bold')
ax.xaxis.set_label_position('top'); ax.xaxis.set_ticks_position('top')
ax.xaxis.set_tick_params(color=color, width=s)
ax.spines['top'].set_position(('axes', 1+pad/100))
ax.spines['top'].set_edgecolor(color); ax.spines['top'].set_linewidth(2)
if ls is not None:
    ax.spines['top'].set_linestyle(ls)
return None

def plot_full_log_8(self, figsize=(15,12), start=None, stop=None, apply_mask:bool=True, size=0.75):
    if start is not None and stop is not None and apply_mask is None:
        l_mask = np.logical_and(self.logs_8.index >= start, self.logs_8.index <= stop)
        f_mask = np.logical_and(self.fpres_8.index >= start, self.fpres_8.index <= stop)
    elif start is None and stop is None and apply_mask is True:
        l_mask = np.logical_and(self.logs_8.index >= self.mask_i_8[0],
self.logs_8.index <= self.mask_i_8[-1])
        f_mask = np.logical_and(self.fpres_8.index >= self.mask_i_8[0],
self.fpres_8.index <= self.mask_i_8[-1])
    else:
        l_mask, f_mask = None, None
    fig, axs = plt.subplots(1, 7, figsize=figsize, sharey=True)
    ax1, ax2, ax3, ax4, ax5, ax6, ax7 = axs
    ax11 = ax1.twiny()
    ax21, ax22, ax23, ax24, ax25 = ax2.twiny(), ax2.twiny(), ax2.twiny(), ax2.twiny(), ax2.twiny()
    ax31, ax32 = ax3.twiny(), ax3.twiny()
    ax41 = ax4.twiny()
    ax51, ax52 = ax5.twiny(), ax5.twiny()
    ax1.set_ylabel('Depth [ft]', weight='bold')
    self.plot_curve(ax1, self.logs_8, 'CAL', 0, 100, 'royalblue', s=size, mask=l_mask,
pad=0, alpha=0.5, fill=True)
    self.plot_curve(ax11, self.logs_8, 'GR', 0, 200, 'green', s=size, mask=l_mask,
pad=8)
    self.plot_curve(ax2, self.logs_8, 'M0R9', 0.2, 2000, 'red', s=size, mask=l_mask,
pad=0, semilog=True)
    self.plot_curve(ax21, self.logs_8, 'M0R6', 0.2, 2000, 'magenta', s=size, mask=l_mask,
semilog=True)
    self.plot_curve(ax22, self.logs_8, 'M0R3', 0.2, 2000, 'orange', s=size, mask=l_mask,
pad=16, semilog=True)
    self.plot_curve(ax23, self.logs_8, 'M0R2', 0.2, 2000, 'green', s=size, mask=l_mask,
pad=24, semilog=True)
    self.plot_curve(ax24, self.logs_8, 'M0R1', 0.2, 2000, 'blue', s=size, mask=l_mask,
pad=32, semilog=True)
    self.plot_curve(ax25, self.logs_8, 'M0RX', 0.2, 2000, 'black', s=size, mask=l_mask,
pad=40, semilog=True)
    self.plot_curve(ax3, self.logs_8, 'ZDEN', 1.65, 2.65, 'red', s=size, mask=l_mask,
pad=0)
    self.plot_curve(ax31, self.logs_8, 'ZCOR', -5, 5, 'black', s=size, mask=l_mask,
pad=8, ls='--')
    self.plot_curve(ax32, self.logs_8, 'NPHI', 0.6, 0, 'blue', s=size, mask=l_mask,
pad=16)
    self.plot_curve(ax4, self.logs_8, 'DT', -50, 180, 'tab:orange', s=size, mask=l_mask,
pad=0)
    self.plot_curve(ax41, self.logs_8, 'PE', 0, 25, 'magenta', s=size, mask=l_mask,
pad=8)
    self.plot_curve(ax5, self.logs_8, 'K', 0, 30, 'mediumpurple', s=size, mask=l_mask,
pad=0)
    self.plot_curve(ax51, self.logs_8, 'TH', 0, 30, 'gray', s=size, mask=l_mask,
pad=8)
    self.plot_curve(ax52, self.logs_8, 'U', 0, 30, 'limegreen', s=size, mask=l_mask,
pad=16)
    self.plot_curve(ax6, self.fpres_8, 'FPRES', 200, 260, 'black', s=5.00, mask=f_mask,
pad=0, ls='', marker='s')
    self.plot_formations(ax7, self.well_8['FMTNS'])
    self.plot_title(fig, self.logs_8); plt.gca().invert_yaxis()
    plt.savefig('figures/25_10-8.jpg', dpi=600, bbox_inches='tight', facecolor='w') if self.save_plot
else None
    plt.show()
    return None

def plot_full_log_16(self, figsize=(15,12), start=None, stop=None, apply_mask:bool=True):
    if start is not None and stop is not None and apply_mask is None:
        l_mask = np.logical_and(self.logs_16.index >= start, self.logs_16.index <= stop)
        c_mask = np.logical_and(self.core_16.index >= start, self.core_16.index <= stop)

```

```

        f_mask = np.logical_and(self.fpres_16.index>=start, self.fpres_16.index<=stop)
    elif start is None and stop is None and apply_mask is True:
        l_mask = np.logical_and(self.logs_16.index>=self.mask_i_16[0],
self.logs_16.index<=self.mask_i_16[-1])
        c_mask = np.logical_and(self.core_16.index>=self.mask_i_16[0],
self.core_16.index<=self.mask_i_16[-1])
        f_mask = np.logical_and(self.fpres_16.index>=self.mask_i_16[0],
self.fpres_16.index<=self.mask_i_16[-1])
    else:
        l_mask, c_mask, f_mask = None, None, None
fig, axs = plt.subplots(1, 8, figsize=figsize, sharey=True)
ax1, ax2, ax3, ax4, ax5, ax6, ax7, ax8 = axs
ax11 = ax1.twiny()
ax21, ax22, ax23, ax24 = ax2.twiny(), ax2.twiny(), ax2.twiny(), ax2.twiny()
ax31 = ax3.twiny()
ax41, ax42, ax43, ax44 = ax4.twiny(), ax4.twiny(), ax4.twiny(), ax4.twiny()
ax51, ax52 = ax5.twiny(), ax5.twiny()
ax61, ax62 = ax6.twiny(), ax6.twiny()
ax71, ax72 = ax7.twiny(), ax7.twiny()
ax1.set_ylabel('Depth [ft]', weight='bold')
self.plot_curve(ax1, self.logs_16, 'HCAL', 0, 100, 'royalblue', mask=l_mask, pad=0,
alpha=0.5, fill=True)
self.plot_curve(ax11, self.logs_16, 'HCGR', -5, 205, 'green', mask=l_mask, pad=8)
self.plot_curve(ax2, self.logs_16, 'AT90', 0.2, 2000, 'red', mask=l_mask, pad=0,
semilog=True)
self.plot_curve(ax21, self.logs_16, 'AT60', 0.2, 2000, 'magenta', mask=l_mask, pad=8,
semilog=True)
self.plot_curve(ax22, self.logs_16, 'AT30', 0.2, 2000, 'orange', mask=l_mask, pad=16,
semilog=True)
self.plot_curve(ax23, self.logs_16, 'AT20', 0.2, 2000, 'green', mask=l_mask, pad=24,
semilog=True)
self.plot_curve(ax24, self.logs_16, 'AT10', 0.2, 2000, 'blue', mask=l_mask, pad=32,
semilog=True)
self.plot_curve(ax3, self.logs_16, 'RH72_1DF', 0.2, 2000, 'tab:blue', mask=l_mask, pad=0,
semilog=True)
self.plot_curve(ax31, self.logs_16, 'RV72_1DF', 0.2, 2000, 'tab:red', mask=l_mask, pad=8,
semilog=True)
self.plot_curve(ax4, self.logs_16, 'RHOZ', 1.65, 2.65, 'red', mask=l_mask, pad=0)
self.plot_curve(ax41, self.logs_16, 'HDRA', -5, 5, 'black', mask=l_mask, pad=8,
ls='--')
self.plot_curve(ax42, self.logs_16, 'TNPH', 0.6, 0, 'blue', mask=l_mask, pad=16)
self.plot_curve(ax43, self.core_16, 'CDEN', 1.65, 2.65, 'darkred', mask=c_mask, pad=24,
ls='', marker='s', s=5)
self.plot_curve(ax44, self.core_16, 'CPOR', 0.6, 0, 'black', mask=c_mask, pad=32,
ls='', marker='o', s=5)
self.plot_curve(ax5, self.logs_16, 'DTCO', 50, 250, 'tab:orange', mask=l_mask, pad=0)
self.plot_curve(ax51, self.logs_16, 'DTSM', 50, 450, 'tab:brown', mask=l_mask, pad=8)
self.plot_curve(ax52, self.logs_16, 'PEFZ', -12, 12, 'magenta', mask=l_mask, pad=16)
self.plot_curve(ax6, self.logs_16, 'HFK', 0, 30, 'mediumpurple', mask=l_mask, pad=0)
self.plot_curve(ax61, self.logs_16, 'HTHO', 0, 30, 'gray', mask=l_mask, pad=8)
self.plot_curve(ax62, self.logs_16, 'HURA', 0, 30, 'limegreen', mask=l_mask, pad=16)
self.plot_curve(ax7, self.fpres_16, 'FPRES', 235, 265, 'black', mask=f_mask, pad=0,
ls='', marker='s', s=5)
self.plot_curve(ax71, self.core_16, 'CSW', 1, 0, 'blue', mask=c_mask, pad=8,
ls='', marker='x', s=5)
self.plot_curve(ax72, self.core_16, 'CPERM', 16e3, 1e0, 'red', mask=c_mask, pad=16,
ls='', marker='o', s=5, semilog=True)
self.plot_nmr2(ax8, self.nmr_16, start=start, stop=stop)
self.plot_title(fig, self.logs_16); plt.gca().invert_yaxis()
plt.savefig('figures/25_10-16S.jpg', dpi=600, bbox_inches='tight', facecolor='w') if self.save_plot
else None
plt.show()
return None

def zonation(self, w8_start=5750, dtw_start=5750, nclusters=5, n_init=50, step_pattern=6,
make_plot:bool=False):
    w16 = self.logs_16.df()
[[ 'HCGR', 'RHOZ', 'PEFZ', 'TNPH', 'AT10', 'AT20', 'AT30', 'AT60', 'AT90', 'DTCO', 'HFK', 'HTHO', 'HURA', 'HDRA' ]].dropna()

    w08 = self.logs_8.df()
[['GR', 'ZDEN', 'PE', 'NPHI', 'M0R1', 'M0R2', 'M0R3', 'M0R6', 'M0R9', 'DT', 'K', 'TH', 'U', 'ZCOR']].dropna()

    def make_DTW(w8_dtw_start=dtw_start, step_pattern=step_pattern):
        gr_16, gr_08 = self.logs_16['HCGR'], self.logs_8['GR']
        w1 = gr_08[self.logs_8.index>w8_dtw_start][~np.isnan(gr_08[self.logs_8.index>w8_dtw_start])]
        w2 = gr_16[~np.isnan(gr_16)]
        alignment = dtw(w1, w2, keep_internals=True, step_pattern=rabinerJuangStepPattern(step_pattern,
"c"))

```

```

    return alignment
def make_kmeans(w08=w08, w16=w16, w8_start=w8_start, nclusters=nclusters, n_init=n_init):
    w08 = w08[w08.index >= w8_start]
    pipeline = make_pipeline(MinMaxScaler(), KMeans(n_clusters=nclusters, n_init=n_init))
    pipeline.fit(np.array(w16))
    labels16 = pipeline.predict(np.array(w16))
    w16 = pd.DataFrame(w16, columns=w16.columns)
    w16['labels'] = labels16
    labels08 = pipeline.predict(np.array(w08))
    w08 = pd.DataFrame(w08, columns=w08.columns)
    w08['labels'] = labels08
    return w08, w16
w08, w16 = make_kmeans()
alignment = make_DTW()
self.zonation_results = {'w08':w08, 'w16':w16, 'alignment':alignment}
self.plot_zonation() if make_plot else None
return self.zonation_results if self.return_d else None

def plot_zonation(self, lb=0, ub=250, ystart=None, yend=9150, figsize=(15,10), buffer=250,
                  cmap='brg_r', fmntn_cmap='Pastel2', h_ratio=[5,1], w_ratio=[1.5, 1.5, 0.5], lw=1.5):
    w08, w16, alignment = self.zonation_results.values()
    w8_start = w08.index[0]
    ystart = w8_start-buffer if ystart is None else ystart
    nclusters = len(np.unique(w16['labels']))
    fig = plt.figure(figsize=figsize)
    gs = GridSpec(2, 3, figure=fig, height_ratios=h_ratio, width_ratios=w_ratio)
    ax1, ax2, ax3 = fig.add_subplot(gs[0, 0]), fig.add_subplot(gs[0, 1]), fig.add_subplot(gs[0, 2])
    ax4, ax5 = fig.add_subplot(gs[1,:-1]), fig.add_subplot(gs[1,-1])
    axs = [ax1, ax2, ax3, ax4, ax5]
    my_cmap = LinearSegmentedColormap.from_list(cmap, mpl.colormaps[cmap](np.linspace(0,1,nclusters)),
N=nclusters)
    for i in range(nclusters):
        im1 = ax1.fill_betweenx(w16.index, lb, ub, where=w16['labels']==i, cmap=my_cmap,
color=my_cmap(i), alpha=0.5)
        im2 = ax2.fill_betweenx(w08.index, lb, ub, where=w08['labels']==i, cmap=my_cmap,
color=my_cmap(i), alpha=0.5)
        self.plot_curve(ax1, self.logs_16, 'HGR', lb, ub, 'k', title='Well 16-S')
        self.plot_curve(ax2, self.logs_8, 'GR', lb, ub, 'k', title='Well 8')
        cb = plt.colorbar(im1, ax=ax1); cb.set_ticks(np.arange(0,1,1/nclusters))
        cb.set_ticklabels(['c{}'.format(i+1) for i in range(nclusters)])
        cb = plt.colorbar(im2, ax=ax2); cb.set_ticks(np.arange(0,1,1/nclusters))
        cb.set_ticklabels(['c{}'.format(i+1) for i in range(nclusters)])
        self.plot_formations(ax3, self.well_8['FMTNS'], cmap=fmntn_cmap)
    for i in range(len(self.well_8['FMTNS'])):
        mid = (self.well_8['FMTNS'][['Top']][i] + self.well_8['FMTNS'][['Bottom']][i])/2
        ax2.axhline(self.well_8['FMTNS'][['Top']][i], c='k', lw=lw, ls='--')
        ax3.axhline(self.well_8['FMTNS'][['Top']][i], c='k', lw=1, ls='--')
        ax2.text(180, mid, self.well_8['FMTNS'][['Name']][i])
    ax1.invert_yaxis(); ax1.set_ylabel('Depth [ft]', weight='bold')
    for i, ax in enumerate([ax1, ax2, ax3, ax4]):
        ax.set_xlim(yend, ystart)
        if i==1 or i==2:
            ax.set_yticklabels([])
    ax4.plot(w16.index, w16['labels']+1, '-', label='Well 16-S')
    ax4.plot(w08.index, w08['labels']-nclusters, '-', label='Well 8')
    ax4.hlines(0, w8_start, yend, color='k', lw=2, ls='--')
    ax4.set_xlabel('Depth [ft]', weight='bold'); ax4.set_ylabel('DTW Class', weight='bold')
    ax4.set_ylim(nclusters+0.5,-(nclusters+0.5)); ax4.set_yticks(np.arange(-nclusters,nclusters+1,1))
    ax4.legend(facecolor='lightgray', edgecolor='k', fancybox=False); ax4.invert_yaxis()
    ax5.plot(alignment.index1, alignment.index2, c='tab:red')
    ax5.set_xticks([]); ax5.set_yticks([]); ax5.set_xlabel('DTW Path', weight='bold')
    [ax.grid(True, which='both') for ax in axs]
    plt.tight_layout()
    plt.savefig('figures/DTW_classes.jpg', dpi=600, bbox_inches='tight', facecolor='w') if
self.save_plot else None
    plt.show()
    return None

def plot_GRvAT90(self, figsize=(15,8), c1='k', c2='g', m='o', s=2, alpha1=0.2, alpha2=0.5, bins=20):
    fig = plt.figure(figsize=figsize)
    gs = GridSpec(2, 2, figure=fig)
    ax1 = fig.add_subplot(gs[0, 0])
    ax2 = fig.add_subplot(gs[0, 1], sharex=ax1, sharey=ax1)
    ax3 = fig.add_subplot(gs[1, 0])
    ax4 = fig.add_subplot(gs[1, 1], sharex=ax3, sharey=ax3)
    axs = [ax1, ax2, ax3, ax4]

```

```

def make_GR_dist(ax, df, gr='GR', c=c2, bins=bins, alpha=alpha2):
    d_ = df[gr]
    d = d_[~np.isnan(d_)]
    mu, std = stats.norm.fit(d)
    x = np.linspace(mu-3*std, mu+3*std, 100)
    pdf, cdf = stats.norm.pdf(x, mu, std), stats.norm.cdf(x, mu, std)
    ax.hist(df[gr], bins=bins, density=True, color=c, edgecolor='gray', alpha=alpha, label='whole
interval')
    ax.plot(x, pdf, c='r', linewidth=2, label='PDF')
    ax.twinx().plot(x, cdf, c='b', linewidth=2, label='CDF')
    ax.set_xlabel('GR [GAPI]', weight='bold')
mask08 = np.logical_and(self.logs_8.index >= self.mask_i_8[0], self.logs_8.index <=
self.mask_i_8[1])
mask16 = np.logical_and(self.logs_16.index >= self.mask_i_16[0], self.logs_16.index <=
self.mask_i_16[1])
ax1.scatter(self.logs_8['GR'][mask08], self.logs_8['M0R9'][mask08], c=c1, alpha=0.2, label='whole
interval')
ax2.scatter(self.logs_16['HCGR'][mask16], self.logs_16['AT90'][mask16], c=c1, alpha=0.2,
label='whole interval')
keys, titles = list(self.zonation_results.keys()), ['Well 8', 'Well 16-S']
for k in range(2):
    m = [self.mask_i_8, self.mask_i_16]
    mask = np.logical_and(self.zonation_results[keys[k]].index >= m[k][0],
self.zonation_results[keys[k]].index <= m[k][1])
    for i in range(1,6):
        zone = self.zonation_results[keys[k]]['labels'] == i
        data = self.zonation_results[keys[k]].iloc[:,[0,8]][mask]
        x, y = data.iloc[:,0], data.iloc[:,1]
        axs[k].scatter(x[zone], y[zone], label=f'zone {i}', alpha=alpha1)
        axs[k].set_title(titles[k], weight='bold')
    axs[k].set_xlabel('GR [GAPI]', weight='bold'); axs[k].set_ylabel('AT90 [$\Omega$cdot m$]', weight='bold')
make_GR_dist(ax3, self.logs_8, 'GR')
make_GR_dist(ax4, self.logs_16, 'HCGR')
for i, ax in enumerate(axs):
    ax.grid(True, which='both')
    ax.legend(facecolor='lightgray', edgecolor='k', fancybox=False)
    if i < 2:
        ax.set_yscale('log')
plt.tight_layout()
plt.savefig('figures/GRvAT90.jpg', dpi=600, bbox_inches='tight', facecolor='w') if self.save_plot
else None
plt.show()
return None

def plot_GrvPEFvRHOB(self, first_hist:str='RHOB', second_hist:str='PEF',
figsize=(15,10), c1='k', m='.', s=2, alpha1=0.5, alpha2=0.5, bins=20):
    hist_mapping = {'GR': (0, 'darkgreen'), 'HCGR': (0, 'darkgreen'), 'ECGR': (0, 'darkgreen'),
    'PE': (1, 'magenta'), 'PEF': (1, 'magenta'), 'PEFZ': (1, 'magenta'),
    'RHOB': (2, 'red'), 'RHOZ': (2, 'red'), 'ZDEN': (2, 'red')}
    fig = plt.figure(figsize=figsize)
    gs = GridSpec(4, 4, figure=fig)
    ax1, ax2, ax3 = fig.add_subplot(gs[0, :2]), fig.add_subplot(gs[1, :2]), fig.add_subplot(gs[2, :2])
    ax4 = fig.add_subplot(gs[0, 2:], sharex=ax1, sharey=ax1)
    ax5 = fig.add_subplot(gs[1, 2:], sharex=ax2, sharey=ax2)
    ax6 = fig.add_subplot(gs[2, 2:], sharex=ax3, sharey=ax3)
    ax7, ax8 = fig.add_subplot(gs[3, 0]), fig.add_subplot(gs[3, 1])
    ax9 = fig.add_subplot(gs[3, 2], sharex=ax7, sharey=ax7)
    ax10 = fig.add_subplot(gs[3, 3], sharex=ax8, sharey=ax8)
    dfs, n1, n2 = [self.logs_8, self.logs_16], ['GR', 'PE', 'ZDEN', 'GR'], ['HCGR', 'PEFZ', 'RHOZ', 'HCGR']
    ms, ns, keys, k = [self.mask_i_8, self.mask_i_16], [n1,n2], list(self.zonation_results.keys()), 0
    if first_hist in hist_mapping.keys():
        p1, c31 = hist_mapping[first_hist]
    else:
        raise ValueError('first histogram must be one of {}'.format(list(hist_mapping.keys())))
    if second_hist in hist_mapping.keys():
        p2, c32 = hist_mapping[second_hist]
    else:
        raise ValueError('second histogram must be one of {}'.format(list(hist_mapping.keys())))
    cs = [c31, c32]
    hist_axes = [ax7, ax8, ax9, ax10]
    hist_data = [dfs[0][n1[p1]], dfs[0][n1[p2]], dfs[1][n2[p1]], dfs[1][n2[p2]]]
    hist_labs = [n1[p1], n1[p2], n2[p1], n2[p2]]
    hist_unit = [dfs[0].curvesdict[n1[p1]].unit, dfs[0].curvesdict[n1[p2]].unit,
                dfs[1].curvesdict[n2[p1]].unit, dfs[1].curvesdict[n2[p2]].unit]

    for j in range(2):

```

```

mask = np.logical_and(dfs[j].index >= ms[j][0], dfs[j].index <= ms[j][1])
for i in range(3):
    units = [dfs[j].curvesdict[ns[j][i]].unit, dfs[j].curvesdict[ns[j][i+1]].unit]
    x, y = dfs[j][ns[j][i]][mask], dfs[j][ns[j][i+1]][mask]
    ax, k = fig.axes[k], k+1
    ax.plot(x, y, c=c1, marker=m, linestyle='', ms=s, alpha=alpha1, label='whole interval')
    ax.set_xlabel('{} {}'.format(ns[j][i], units[0]), weight='bold')
    ax.set_ylabel('{} {}'.format(ns[j][i+1], units[1]), weight='bold')
    for p in range(1,6):
        zone = self.zonation_results[keys[j]]['labels'] == p
        x = self.zonation_results[keys[j]][ns[j][i]]
        y = self.zonation_results[keys[j]][ns[j][i+1]]
        ax.plot(x[zone], y[zone], marker=m, linestyle='', ms=s*2, alpha=alpha2, label='zone
{}'.format(p))
for i in range(4):
    ax, d_ = hist_axes[i], hist_data[i]
    d = d_[~np.isnan(d_)]
    mu, std = stats.norm.fit(d)
    x = np.linspace(mu-3*std, mu+3*std, 100)
    pdf, cdf = stats.norm.pdf(x, mu, std), stats.norm.cdf(x, mu, std)
    ax.hist(hist_data[i], bins=bins, density=True, color=cs[i%2], edgecolor='gray', alpha=alpha2,
label='whole interval')
    ax.plot(x, pdf, c='r', linewidth=2, label='PDF')
    ax.twinx().plot(x, cdf, c='b', linewidth=2)
    ax.set_xlabel('{} {}'.format(hist_labs[i], hist_unit[i]), weight='bold')
for i in range(2):
    fld, com, wll = dfs[i].header['Well']['FLD'].value, dfs[i].header['Well']['COMP'].value,
dfs[i].header['Well']['WELL'].value
    fig.axes[i*3].set_title('{} | {} | {}'.format(fld, com, wll), weight='bold')
for i in range(len(fig.axes[:-4])):
    fig.axes[i].legend(facecolor='lightgray', edgecolor='k')
    fig.axes[i].grid(True, which='both')
plt.tight_layout()
plt.savefig('figures/GRvPEFvRHOB.jpg', dpi=600, bbox_inches='tight', facecolor='w') if
self.save_plot else None
plt.show()
return None

```

```

def plot_KvsTh(self, figsize=(12,4)):
    fig, axs = plt.subplots(1, 3, figsize=figsize, sharex=True, sharey=True)
    ax1, ax2, ax3 = axs
    titles = ['K-vs-Th - Clay Types', 'K-vs-Th - Sandstones', 'K-vs-Th - Shales']
    for ax in axs:
        l = np.linspace(0,5,50)
        ax.plot(l, l*25, label='Th/K=25'); ax.plot(l, l*12, label='Th/K=12')
        ax.plot(l, l*3.5, label='Th/K=3.5'); ax.plot(l, l*2, label='Th/K=2')
        ax.axline([0,16.6],[5,20], c='k', linestyle='--'); ax.text(1.7, 18.5, 'Clay Line')
        ax.plot([0.282, 3.95], [7.5, 14], c='k', linestyle='-.'); ax.text(1.1, 9.2, 'Mixed Layer Clay',
rotation=9.5)
        ax.grid(True, which='both'); ax.legend(facecolor='lightgray', edgecolor='k', fancybox=False)
        ax.set_xlabel('{} {}'.format('HFK', self.logs_16.curvesdict['HFK'].unit), weight='bold')
        ax.set_title(titles[axs.tolist().index(ax)], weight='bold')
    data = self.logs_16.df().loc[:,['HFK','HTHO','HCGR']]
    mask = np.logical_and(data.index >= self.mask_i_16[0], data.index <= self.mask_i_16[1])
    x0, y0, z = data.iloc[:,0][mask], data.iloc[:,1][mask], data.iloc[:,2][mask]
    im = ax1.scatter(x0, y0, c=z, s=2, cmap='jet', alpha=0.5, vmin=0, vmax=200)
    ax1.set_ylabel('{} {}'.format('HTHO', self.logs_16.curvesdict['HTHO'].unit), weight='bold')
    ax1.set_xlim(0, 5); ax1.set_ylim(0, 35)
    shales = self.zones16[self.zones16['Lith'] == 1]
    sands = self.zones16[self.zones16['Lith'] == 0]
    ax2.scatter(x0, y0, s=1, c='k', alpha=0.1)
    for i in range(len(sands)):
        mask = np.logical_and(data.index >= sands.iloc[i,0], data.index <= sands.iloc[i,1])
        x, y, z = data.iloc[:,0][mask], data.iloc[:,1][mask], data.iloc[:,2][mask]
        ax2.scatter(x, y, c=z, s=2, cmap='jet', alpha=0.5, vmin=0, vmax=200)
    ax3.scatter(x0, y0, s=1, c='k', alpha=0.1)
    for i in range(len(shales)):
        mask = np.logical_and(data.index >= shales.iloc[i,0], data.index <= shales.iloc[i,1])
        x, y, z = data.iloc[:,0][mask], data.iloc[:,1][mask], data.iloc[:,2][mask]
        ax3.scatter(x, y, c=z, s=2, cmap='jet', alpha=0.5, vmin=0, vmax=200)
    cb = plt.colorbar(im, ax=ax)
    cb.set_label('{} {}'.format('HCGR', self.logs_16.curvesdict['HCGR'].unit), weight='bold',
rotation=270, labelpad=15)
    plt.tight_layout(); plt.show()

```

```

def plot_res_crossplot(self, figsize=(12,5.5)):
```

```

fig, axs = plt.subplots(1, 2, figsize=figsize)
data = [['HCGR', 'RHOZ', 'PEFZ'], ['RH72_1DF', 'RV72_1DF', 'HCGR']]
for i, ax in enumerate(axs):
    mask = np.logical_and(self.logs_16.index >= self.mask_i_16[0], self.logs_16.index <=
self.mask_i_16[1])
    x, y, z = self.logs_16[data[i][0]][mask], self.logs_16[data[i][1]][mask], self.logs_16[data[i]
[2]][mask]
    im = ax.scatter(x, y, c=z, cmap='jet', alpha=0.5)
    cb = plt.colorbar(im, ax=ax, orientation='horizontal')
    cb.set_label('{}_{}'.format(data[i][2], self.logs_16.curvesdict[data[i][2]].unit),
weight='bold', labelpad=10)
    ax.grid(True, which='both')
    ax.set_xlabel('{}_{}'.format(data[i][0], self.logs_16.curvesdict[data[i][0]].unit),
weight='bold')
    ax.set_ylabel('{}_{}'.format(data[i][1], self.logs_16.curvesdict[data[i][1]].unit),
weight='bold')
    ax.axline([0,0], [1,1], color='r', linestyle='--'); ax.set_xscale('log'); ax.set_yscale('log')
plt.tight_layout(); plt.show()

def plot_RHOB_NPHI(self, figsize=(15, 10), w_ratios=None):
    fig, axs = plt.subplots(1, 5, figsize=figsize, sharey=True, width_ratios=w_ratios)
    ax1, ax2, ax3, ax4, ax5 = axs
    data, pts = self.corr_vals['w16_calcvalues'], [self.sand_pt, self.shale_pt, self.lime_pt]
    ax11, ax21, ax22, ax23, ax31, ax41 = ax1.twiny(), ax2.twiny(), ax2.twiny(),
ax2.twiny(), ax3.twiny(), ax4.twiny()
    self.plot_curve(ax1, self.logs_16, 'HCGR', 0, 200, 'g')
    self.plot_curve(ax11, self.corr_vals['w16_calcvalues'], 'Csh', 0, 1, 'k', units='v/v', pad=8)
    self.plot_curve(ax2, self.logs_16, 'RHOZ', 1.65, 2.65, 'r')
    self.plot_curve(ax21, data, 'RHOB', 1.65, 2.65, 'firebrick', ls='', marker='.', pad=8)
    self.plot_curve(ax22, self.logs_16, 'TNPH', 1, 0, 'b', pad=16)
    self.plot_curve(ax23, data, 'TNPH', 1, 0, 'royalblue', ls='', marker='.', pad=24)
    self.plot_curve(ax3, data, 'RHOB_SS_LS', 1, 0, 'darkred')
    self.plot_curve(ax31, data, 'NPHI_SS_LS', 1, 0, 'darkblue', pad=8)
    self.plot_curve(ax4, data, 'RHOB_Corr', 1, 0, 'tab:red')
    self.plot_curve(ax41, data, 'NPHI_Corr', 1, 0, 'tab:blue', pad=8)
    self.plot_formations(ax5, df=self.zones16)
    self.plot_title(fig, self.logs_16)
    for i, ax in enumerate(axs):
        ax.grid(True, which='both')
        ax.hlines(self.zones16['Top'], 0, 200, ls='--', color=['k' if i!=0 else 'r'])
        ax.hlines([p for p in pts], 0, 200, lw=2.5, color=['gold', 'g', 'c'])
    plt.gca().invert_yaxis(); ax1.set_ylabel('Depth [ft]', weight='bold')
    plt.savefig('figures/RHOB_NPHI.jpg', dpi=600, bbox_inches='tight', facecolor='w') if self.save_plot
else None
    plt.show()
    return None

def resistivity_inversion(self, Rvsh=None, Rhsh=None, lambda_reg=1e-4, Wd_matrix:bool=True,
x0=[0.5, 1.5], method='L-BFGS-B', tol=1e-6, maxiter=1e3):
    if Rvsh is None:
        Rvsh = self.logs_16.df()['RV72_1DF'].loc[self.shale_pt] #hw1=2.8133
    if Rhsh is None:
        Rhsh = self.logs_16.df()['RH72_1DF'].loc[self.shale_pt] #hw1=0.7746
    def objective(variables, *args):
        Csh, Rs = variables
        Rv, Rh = args[0], args[1]
        eq1 = (Csh*Rvsh + (1-Csh)*Rs) - Rv
        eq2 = (Csh/Rhsh + (1-Csh)/Rs) - (1/Rh)
        eqs = [eq1/Rv, eq2*Rh] if Wd_matrix else [eq1, eq2]
        return linalg.norm(eqs) + lambda_reg*linalg.norm(variables)
    def inversion():
        res_aniso = self.logs_16.df()[['RV72_1DF', 'RH72_1DF']].dropna()
        sol, fun, jac, nfev = [], [], [], []
        for _, row in res_aniso.iterrows():
            Rv_value, Rh_value = row['RV72_1DF'], row['RH72_1DF']
            solution = optimize.minimize(objective,
                x0 = x0,
                args = (Rv_value, Rh_value),
                bounds = [(0,1), (None, None)],
                method = method,
                tol = tol,
                options = {'maxiter':maxiter})
            fun.append(solution.fun); jac.append(solution.jac); nfev.append(solution.nfev)
            jac1, jac2 = np.array(jac)[:,0], np.array(jac)[:,1]
            sol.append({'Csh':solution.x[0], 'Rs':solution.x[1]})
        sol = pd.DataFrame(sol, index=res_aniso.index)
        sol['func'], sol['nfev'], sol['jac1'], sol['jac2'], sol['norm_jac'] = fun, nfev, jac1, jac2,

```

```

linalg.norm(jac, axis=1)
    return sol
def simulate(sol):
    Csh, Rs = sol['Csh'], sol['Rs']
    Rv = Csh*Rvsh + (1-Csh)*Rs
    Rh = Csh/Rhsh + (1-Csh)/Rs
    sim = pd.DataFrame({'Rv_sim':Rv, 'Rh_sim':1/Rh}, index=sol.index)
    return sim
sol = inversion()
self.res_inv = sol.join(simulate(sol))
gr_16 = self.logs_16.df()['HCGR'].dropna()
self.res_inv['Csh_linear'] = (gr_16.gr_16.min())/(gr_16.max()-gr_16.min())
return self.res_inv if self.return_d else None

def plot_resistivity_inversion(self, figsize=(15,10), gr_lim=[0,200], res_lim=[0.2,2000],
                                nfev_lim=[20,300], jac_lim=[-0.05,250], fun_lim=[-0.05,0.5],
                                gr_c='g', at10_c='r', at90_c='b', resh_c='tab:blue', resv_c='tab:red',
sim_c='k',
                                res_units='$\Omega\cdot m$', inv_cs=['navy','darkred','k']):
    fig, axs = plt.subplots(1, 5, figsize=figsize, sharey=True)
    ax1, ax2, ax3, ax4, ax5 = axs
    [ax.grid(True, which='both') for ax in axs]
    c1, c2, c3 = inv_cs
    ax11, ax12 = ax1.twiny(), ax1.twiny()
    ax21, ax22 = ax2.twiny(), ax2.twiny()
    ax31, ax32 = ax3.twiny(), ax3.twiny()
    ax41 = ax4.twiny(); ax51 = ax5.twiny()
    self.plot_curve(ax1, self.logs_16, 'HCGR', gr_lim[0], gr_lim[1], gr_c, pad=16,
    self.plot_curve(ax11, self.res_inv, 'Csh_linear', 0, 1, 'gray', pad=8,
units='v/v')
    self.plot_curve(ax12, self.res_inv, 'Csh', 0, 1, sim_c, pad=0,
units='v/v', ls='--')
    self.plot_curve(ax2, self.logs_16, 'AT10', res_lim[0], res_lim[1], at10_c, pad=16,
semilog=True)
    self.plot_curve(ax21, self.logs_16, 'AT90', res_lim[0], res_lim[1], at90_c, pad=8,
semilog=True)
    self.plot_curve(ax22, self.res_inv, 'Rs', res_lim[0], res_lim[1], sim_c, pad=0,
semilog=True, units=res_units)
    self.plot_curve(ax3, self.res_inv, 'nfev', nfev_lim[0], nfev_lim[1], c1, pad=8, label='# Func Evals')
    self.plot_curve(ax31, self.res_inv, 'norm_jac', jac_lim[0], jac_lim[1], c2, pad=16,
label='$\ell_2(\mathbf{Jac})$')
    self.plot_curve(ax32, self.res_inv, 'func', fun_lim[0], fun_lim[1], c3, pad=0,
label='Objective function')
    self.plot_curve(ax4, self.logs_16, 'RH72_1DF', res_lim[0], res_lim[1], resh_c, pad=8,
semilog=True)
    self.plot_curve(ax41, self.res_inv, 'Rh_sim', res_lim[0], res_lim[1], sim_c, pad=0,
semilog=True, units=res_units, ls='--')
    self.plot_curve(ax5, self.logs_16, 'RV72_1DF', res_lim[0], res_lim[1], resv_c, pad=8,
semilog=True)
    self.plot_curve(ax51, self.res_inv, 'Rv_sim', res_lim[0], res_lim[1], sim_c, pad=0,
semilog=True, units=res_units, ls='--')
    plt.gca().invert_yaxis(); ax1.set_ylabel('Depth [ft]', weight='bold')
    self.plot_title(fig, self.logs_16)
    plt.savefig('figures/resistivity_inversion.jpg', dpi=600, bbox_inches='tight', facecolor='w') if
self.save_plot else None
    plt.show()
    return None

def plot_csh(self, figsize=(15,15)):
    gr8 = self.logs_8.df()['GR']
    self.csh8 = pd.DataFrame({'Csh_linear':(gr8 - gr8.min()) / (gr8.max() - gr8.min())})
    fig, axs = plt.subplots(1, 2, figsize=figsize)
    ax1, ax2 = axs
    ax11, ax21, ax22 = ax1.twiny(), ax2.twiny(), ax2.twiny()
    self.plot_curve(ax1, self.logs_8, 'GR', 0, 200, color='green')
    self.plot_curve(ax2, self.logs_16, 'HCGR', 0, 200, color='green')
    self.plot_curve(ax11, self.csh8, 'Csh_linear', 0, 1, 'k', units='v/v', pad=8)
    self.plot_curve(ax21, self.res_inv, 'Csh_linear', 0, 1, 'black', alpha=0.8, units='v/v', ls='--',
pad=8)
    self.plot_curve(ax22, self.res_inv, 'Csh', 0, 1, 'k', units='v/v', pad=16)
    ax1.set_title('Well 8', weight='bold'); ax2.set_title('Well 16S', weight='bold')
    ax1.set_ylabel('Depth [ft]', weight='bold')
    [ax.invert_yaxis() for ax in axs]
    plt.show()

```

```

def plot_fpres(self, figsize=(10,8), mybox={'facecolor':'wheat', 'edgecolor':'k'}):
    fig, axs = plt.subplots(1, 2, figsize=figsize)
    ax1, ax2 = axs
    ax11, ax12, ax13, ax21, ax22, ax23 = ax1.twiny(), ax1.twiny(), ax1.twiny(), ax2.twiny(),
    ax2.twiny(), ax2.twiny()
    axf = [ax13, ax23]
    titles = ['Well 8', 'Well 16S']
    fluids = ['gas', 'oil', 'water', 'other?']
    cs = ['r', 'g', 'b', 'tab:orange']
    masks08 = [[6800,7000], [7650,8000], [8150,9000]]
    masks16 = [[7850,8050], [8125,8250], [8400,8550], [8525,9000]]
    masks = [masks08, masks16]
    pres08, pres16 = self.fpres_8.df().dropna(), self.fpres_16.df().dropna()
    pres08['FPRES2'] = pres08['FPRES']*14.7
    pres16['FPRES2'] = pres16['FPRES']*14.7
    dfs = [pres08, pres16]
    self.plot_curve(ax13, pres08, 'FPRES2', 2900, 4000, s=5, ls='', marker='v', units='psia')
    self.plot_curve(ax1, self.logs_8, 'GR', 0, 200, 'darkgreen', alpha=0.5, pad=8)
    self.plot_curve(ax11, self.logs_8, 'M0R1', 0.2, 2000, 'darkblue', semilog=True, alpha=0.5,
    pad=16)
    self.plot_curve(ax12, self.logs_8, 'M0R9', 0.2, 2000, 'darkred', semilog=True, alpha=0.5,
    pad=24)
    self.plot_curve(ax23, pres16, 'FPRES2', 3400, 4000, s=5, ls='', marker='v', units='psia')
    self.plot_curve(ax2, self.logs_16, 'HCGR', 0, 200, 'darkgreen', alpha=0.5, pad=8)
    self.plot_curve(ax21, self.logs_16, 'AT10', 0.2, 2000, 'darkblue', semilog=True, alpha=0.5,
    pad=16)
    self.plot_curve(ax22, self.logs_16, 'AT90', 0.2, 2000, 'darkred', semilog=True, alpha=0.5,
    pad=24)
    for k in range(2):
        for j in range(len(masks[k])):
            data = dfs[k].loc[masks[k][j][0]:masks[k][j][1]]
            z = data.index.values.reshape(-1,1)
            lr = LinearRegression()
            lr.fit(z, data['FPRES2'])
            y = lr.predict(z)
            axf[k].plot(y, z, ls='-', c=cs[j])
            axf[k].text(data['FPRES2'].min(), masks[k][j][0], '{} $\nabla$={:.3f}'.format(fluids[j],
            lr.coef_[0]), bbox=mybox)
    [ax.set_title(t, weight='bold') for ax, t in zip(axs, titles)]
    [ax.grid(True, which='both') for ax in axs]
    ax1.set_ylim(6550,8400); ax2.set_ylim(7750, 8600)
    [ax.invert_yaxis() for ax in axs]
    plt.show()

def plot_phi(self, figsize=(15,10)):
    csh16 = self.res_inv['Csh'].loc[6300:9002]
    phin_16 = self.corr_vals['w16_calcvalues']['NPHI_Corr']
    phid_16 = self.corr_vals['w16_calcvalues']['RHOB_Corr']
    phish_16 = np.mean([0.1745, 0.4221])
    phis_16 = np.sqrt(0.5*(phid_16**2 + phin_16**2))
    phit_16 = (1-csh16)*phis_16 + csh16*phish_16
    self.phi_16 = pd.DataFrame({'Csh':csh16,
    'phiN':phin_16, 'phiD':phid_16, 'phiS':phis_16, 'phiT':phit_16})
    csh08 = self.csh8.loc[6500:8632.25].values.squeeze()
    phin_08 = self.corr_vals['w08_neutron']['NPHI_Corr (v/v)'].loc[6500:8632.25]
    rhob_08 = self.logs_16.df()['RHOZ'].loc[6500:8632.25]
    phid_08 = (rhob_08 - 2.65) / (1-2.65)
    phish_08 = np.mean([0.1745, 0.4221])
    phis_08 = np.sqrt(0.5*(phid_08**2 + phin_08**2))
    phit_08 = (1-csh08)*phis_08 + csh08*phish_08
    self.phi_08 = pd.DataFrame({'Csh':csh08,
    'phiN':phin_08, 'phiD':phid_08, 'phiS':phis_08, 'phiT':phit_08})
    fig, axs = plt.subplots(1, 4, figsize=figsize)
    ax1, ax2, ax3, ax4 = axs
    titles = ['Well 8', 'Well 8', 'Well 16S', 'Well 16S']
    ax11, ax12 = ax1.twiny(), ax1.twiny()
    self.plot_curve(ax1, self.logs_8, 'GR', 0, 200, 'g')
    self.plot_curve(ax11, self.logs_8, 'M0R1', 0.2, 2000, 'b', semilog=True, pad=8)
    self.plot_curve(ax12, self.logs_8, 'M0R9', 0.2, 2000, 'r', semilog=True, pad=16)
    ax21, ax22 = ax2.twiny(), ax2.twiny()
    self.plot_curve(ax2, self.phi_08, 'phiN', 1, 0, 'darkblue', units='frac')
    self.plot_curve(ax21, self.phi_08, 'phiD', 1, 0, 'darkred', units='frac', pad=8)
    self.plot_curve(ax22, self.phi_08, 'phiT', 1, 0, 'k', units='frac', pad=16)
    ax31, ax32 = ax3.twiny(), ax3.twiny()
    self.plot_curve(ax3, self.logs_16, 'HCGR', 0, 200, 'g')
    self.plot_curve(ax31, self.logs_16, 'AT10', 0.2, 2000, 'b', semilog=True, pad=8)
    self.plot_curve(ax32, self.logs_16, 'AT90', 0.2, 2000, 'r', semilog=True, pad=16)

```

```

ax41, ax42 = ax4.twiny(), ax4.twiny()
self.plot_curve(ax4, self.phi_16, 'phiN', 1, 0, 'darkblue', units='frac')
self.plot_curve(ax41, self.phi_16, 'phiD', 1, 0, 'darkred', units='frac', pad=8)
self.plot_curve(ax42, self.phi_16, 'phiT', 1, 0, 'k', units='frac', pad=16)
ax2.sharey(ax1); ax4.sharey(ax3); ax1.invert_yaxis(); ax3.invert_yaxis()
for i, ax in enumerate(axes):
    ax.set_title(titles[i], weight='bold')

def plot_phi_perm(self, figsize=(15,10), kw=6000, kappa=8, eta=1.75):
    fig, axs = plt.subplots(1, 2, figsize=figsize)
    ax1, ax2 = axs
    titles = ['Well 8', 'Well 16S']
    self.phi_08['k_wr'] = kw * (self.phi_08['phiT']**kappa / 0.25**eta)
    self.phi_16['k_wr'] = kw * (self.phi_16['phiT']**kappa / 0.25**eta)
    ax11 = ax1.twiny()
    self.plot_curve(ax1, self.phi_08, 'k_wr', 1e-3, 1e3, units='mD', semilog=True)
    self.plot_curve(ax11, self.phi_08, 'phiT', 1, 0, color='darkblue', units='frac', pad=8)
    ax21, ax22, ax23 = ax2.twiny(), ax2.twiny(), ax2.twiny()
    self.plot_curve(ax2, self.phi_16, 'k_wr', 1e-3, 1e3, units='mD', semilog=True)
    self.plot_curve(ax21, self.core_16, 'CPERM', 1e-3, 1e3, marker='d', color='r', s=5, semilog=True,
pad=8)
    self.plot_curve(ax22, self.phi_16, 'phiT', 1, 0, color='darkblue', units='frac', pad=16)
    self.plot_curve(ax23, self.core_16, 'CPOR', 1, 0, marker='o', color='b', s=5, pad=24)
    for i, ax in enumerate(axes):
        ax.invert_yaxis()
        ax.set_title(titles[i], weight='bold')
    plt.show()
    return None

def plot_core_log_poro_perm(self, figsize=(12,4.5), lims1=[0,0.5], lims2=[1e-3,1e5]):
    por_perm_df = self.core_16.df()[['CPOR','CPERM']].dropna()
    por_perm_df['LPOR'] = self.phi_16['phiT']
    por_perm_df['LPERM'] = self.phi_16['k_wr']
    fig, axs = plt.subplots(1, 2, figsize=figsize)
    ax1, ax2 = axs
    ax1.scatter(por_perm_df['LPOR'], por_perm_df['CPOR'])
    ax2.loglog(por_perm_df['LPERM'], por_perm_df['CPERM'], marker='o', ls='')
    ax1.set_xlabel('Log Porosity [v/v]', weight='bold'); ax1.set_ylabel('Core Porosity [v/v]',
weight='bold')
    ax2.set_xlabel('Log Permeability [mD]', weight='bold'); ax2.set_ylabel('Core Permeability [mD]',
weight='bold')
    ax1.set(xlim=lims1, ylim=lims1); ax2.set(xlim=lims2, ylim=lims2)
    for ax in axs:
        ax.grid(True, which='both')
        ax.axline([0,0],[1,1], color='r')
    fig.suptitle('Well 16S', weight='bold')
    plt.tight_layout(); plt.show()
    return None

def plot_thomas_stieber(self, figsize=(12,4), color='cornflowerblue'):
    fig, axs = plt.subplots(1, 2, figsize=figsize)
    ax1, ax2 = axs
    titles = ['Well 8', 'Well 16S']
    data = [self.phi_08, self.phi_16]
    mask8, mask16 = [8150, 8290], [8400, 8550]
    masks = [mask8, mask16]
    for i, ax in enumerate(axes):
        df = data[i][data[i]['phiT']<=1].dropna()
        x, y, = df['Csh'], df['phiT']
        ax.scatter(x, y, c='k', alpha=0.15, label='whole interval')
        ax.scatter(x.loc[masks[i][0]:masks[i][1]], y.loc[masks[i][0]:masks[i][1]], c=color, alpha=0.5,
label='aquifer zone')
        xt, yt = x.loc[masks[i][0]:masks[i][1]], y.loc[masks[i][0]:masks[i][1]]
        lr = LinearRegression()
        lr.fit(xt.values.reshape(-1,1), yt.values.reshape(-1,1))
        t = np.linspace(0,1,100).reshape(-1,1)
        z = lr.predict(t)
        ax.plot(t, z, c='r', lw=3)
        line = f'$\phi_T$ = {lr.coef_[0][0] :.2f} $C_{sh}$ + {lr.intercept_[0] :.2f}'
        ax.text(0.05, 0.95, line, transform=ax.transAxes, color='r')
        ax.set_title(titles[i], weight='bold')
        ax.grid(True, which='both')
        ax.legend(facecolor='wheat', edgecolor='k', fancybox=False)
        ax.set_xlabel('$C_{sh}$ [v/v]', weight='bold')
        ax.set_ylabel('$\phi_T$ [v/v]', weight='bold')
        ax.set(xlim=(0,1), ylim=(0,1))

```

```

plt.tight_layout(); plt.show()
return None

def plot_pickett(self, figsize=(12,8)):
    fig, axs = plt.subplots(2, 2, figsize=figsize)
    titles = ['Well 8', 'Well 16S']
    reses = [self.logs_8.df()['MOR9'], self.logs_16.df()['AT90']]
    phis = [self.phi_08['phiT'], self.phi_16['phiT']]
    masks = [[8150, 8290], [8450, 8500]]
    lims = [[[0.15,5], [1,0.1]], [[0.15,3], [1,0.1]]]
    for j in range(2):
        x, y = phis[j], reses[j].loc[phis[j].index]
        axs[0,j].scatter(x, y, c='k', alpha=0.1, label='whole interval')
        for i in range(2):
            xt, yt = phis[j].loc[masks[i][0]:masks[i][1]], reses[j].loc[masks[i][0]:masks[i][1]]
            axs[i,j].scatter(xt, yt, label='aquifer zone', c='cornflowerblue', alpha=0.5)
            axs[i,j].sharey(axs[0,j])
            axs[i,j].set_yscale('log')
            axs[i,j].set_title(titles[i], weight='bold')
            axs[i,j].set_xlabel('$\phi_T$ [v/v], weight='bold')
            axs[i,j].set_ylabel('AT90 [$\Omega_m$]', weight='bold')
            axs[i,j].grid(True, which='both')
            axs[i,j].legend(facecolor='wheat', edgecolor='k', fancybox=False)
            axs[i,j].set_xlim=(0,1))
        axs[1,j].axline(lims[j][0], lims[j][1], c='r')
    plt.tight_layout(); plt.show()
    return None

def archie_sw(self, a=1, m=1.8, n=1.7, rw8=0.0295, rs8=0.4183, rw16=0.0295, rs16=0.4183):
    phis08 = self.phi_08['phis']
    self.phi_08['Sw'] = ((a*rw8) / (rs8 * phis08**m))**(1/n)
    self.phi_08['Shc'] = 1-self.phi_08['Sw']
    self.phi_08['HPV'] = self.phi_08['Shc'] * self.phi_08['phiT']
    phis16 = self.phi_16['phis']
    self.phi_16['Sw'] = ((a*rw16) / (rs16 * phis16**m))**(1/n)
    self.phi_16['Shc'] = 1-self.phi_16['Sw']
    self.phi_16['HPV'] = self.phi_16['Shc'] * self.phi_16['phiT']
    return None

def plot_archie_sw(self, figsize=(15,13)):
    fig = plt.figure(figsize=figsize)
    gs = GridSpec(1, 2)
    gs_l = gs[0].subgridspec(1,3)
    gs_r = gs[1].subgridspec(1,3)
    ax1 = plt.subplot(gs_l[0])
    ax2 = plt.subplot(gs_l[1], sharey=ax1)
    ax3 = plt.subplot(gs_l[2], sharey=ax1)
    ax4 = plt.subplot(gs_r[0])
    ax5 = plt.subplot(gs_r[1], sharey=ax4)
    ax6 = plt.subplot(gs_r[2], sharey=ax4)
    ax11, ax12 = ax1.twiny(), ax1.twiny()
    self.plot_curve(ax1, self.logs_8, 'GR', 0, 200, 'g')
    self.plot_curve(ax11, self.logs_8, 'MOR1', 0.2, 2000, 'b', semilog=True, pad=8)
    self.plot_curve(ax12, self.logs_8, 'MOR9', 0.2, 2000, 'r', semilog=True, pad=16)
    ax21, ax22 = ax2.twiny(), ax2.twiny()
    phin_08 = self.corr_vals['w08_neutron']['NPHI_Corr (v/v)'].loc[6500:8632.25]
    rhob_08 = self.logs_16.df()['RHOZ'].loc[6500:8632.25]
    phid_08 = (rhob_08 - 2.65) / (1-2.65)
    df = pd.DataFrame({'RHOB_Corr': rhob_08, 'NPHI_Corr': phin_08, 'PHID': phid_08})
    self.plot_curve(ax2, df, 'PHID', 1, 0, 'r')
    self.plot_curve(ax21, df, 'NPHI_Corr', 1, 0, 'b', pad=8)
    self.plot_curve(ax22, self.phi_08, 'phiT', 1, 0, 'k', units='v/v', pad=16)
    ax31, ax32 = ax3.twiny(), ax3.twiny()
    self.plot_curve(ax3, self.phi_08, 'Sw', 1, 0, 'dodgerblue', bar=True, units='v/v')
    self.plot_curve(ax31, self.phi_08, 'Shc', 0, 1, 'darkred', bar=True, units='v/v', pad=8)
    self.plot_curve(ax32, self.phi_08, 'HPV', 0, 1, 'k', pad=16)
    ax41, ax42 = ax4.twiny(), ax4.twiny()
    self.plot_curve(ax4, self.logs_16, 'HCGR', 0, 200, 'g')
    self.plot_curve(ax41, self.logs_16, 'AT10', 0.2, 2000, 'b', semilog=True, pad=8)
    self.plot_curve(ax42, self.logs_16, 'AT90', 0.2, 2000, 'r', semilog=True, pad=16)
    ax51, ax52, ax53 = ax5.twiny(), ax5.twiny(), ax5.twiny()
    self.plot_curve(ax5, self.corr_vals['w16_calcvalues'], 'RHOB_Corr', 1, 0, 'r')
    self.plot_curve(ax51, self.corr_vals['w16_calcvalues'], 'NPHI_Corr', 1, 0, 'b', pad=8)
    self.plot_curve(ax52, self.phi_16, 'phiT', 1, 0, 'k', units='v/v', pad=16)
    self.plot_curve(ax53, self.core_16, 'CPOR', 1, 0, 'm', marker='s', ls='', s=5, pad=24)
    ax61, ax62 = ax6.twiny(), ax6.twiny()
    self.plot_curve(ax6, self.phi_16, 'Sw', 1, 0, 'dodgerblue', bar=True, units='v/v')

```

```

self.plot_curve(ax61, self.phi_16, 'Shc', 0, 1, 'darkred', bar=True, units='v/v', pad=8)
self.plot_curve(ax62, self.phi_16, 'HPV', 0, 1, 'k', pad=16)
ax1.invert_yaxis(); ax4.invert_yaxis(); ax1.set_ylabel('Depth [ft]', weight='bold')
fig.text(0.11, 0.075, '|+'-'*25+'Well 8'+'-*25+'|', weight='bold', fontsize=16)
fig.text(0.525, 0.075, '|+'-'*25+'Well 16S'+'-*25+'|', weight='bold', fontsize=16)
plt.show()
return None

def leverett_class(self, figsize=(15,10), cutoffs=[0, 0.55, 1.25, 3.5, 12, 220], maxphi=0.4, alpha=0.15,
fs=11.5,
           colors=['r','gold','g','b','k']):
df = self.core_16.df()[['CPOR','CPERM']].dropna()
x, y = df['CPOR'], df['CPERM']
data = np.sqrt(df['CPERM'] / (df['CPOR']))
leverett_masks = []
for i in range(len(cutoffs)-1):
    mask = np.logical_and(data>=cutoffs[i], data<=cutoffs[i+1])
    leverett_masks.append(mask)
color_centers = []
for i in range(len(cutoffs)-1):
    color_centers.append(np.mean([cutoffs[i], cutoffs[i+1]]))
lin_poro = np.linspace(0, maxphi, 100)
lin_perm_low, lin_perm_med, lin_perm_high = [], [], []
for i in range(len(color_centers)):
    lin_perm_low.append(cutoffs[i]**2 * lin_poro)
    lin_perm_med.append(color_centers[i]**2 * lin_poro)
    lin_perm_high.append(cutoffs[i+1]**2 * lin_poro)
df['CLASS'] = np.zeros(len(df))
for i, m in enumerate(leverett_masks):
    df.loc[m, 'CLASS'] = i+1
def plotter():
    fig = plt.figure(figsize=figsize)
    gs = GridSpec(5, 2, figure=fig, width_ratios=[1.5, 1])
    ax1 = fig.add_subplot(gs[1:4, 0])
    ax2 = fig.add_subplot(gs[:, 1])
    axs = [ax1, ax2]
    for i, m in enumerate(leverett_masks):
        ax1.scatter(x[m], y[m], c=colors[i], label=f'$\sqrt{k/\phi})$ = {color_centers[i]:.2f}')
        ax1.plot(lin_poro, lin_perm_med[i], c=colors[i])
        ax1.fill_between(lin_poro, lin_perm_low[i], lin_perm_high[i], color=colors[i], alpha=alpha)
        ax1.scatter(x, y, c='lightgray', marker='.', s=1, label='Core Data')
        ax1.set_title('Leverett Rock Classification', weight='bold')
        ax1.set_yscale('log')
        ax1.set_xlabel('Core Porosity [v/v]', weight='bold'); ax1.set_ylabel('Core Permeability [mD]', weight='bold')
        ax1.legend(facecolor='wheat', edgecolor='k', bbox_to_anchor=(0.5,-0.33), loc='lower center',
ncol=3, fontsize=fs)
        ax21, ax22 = ax2.twiny(), ax2.twiny()
        self.plot_curve(ax21, self.core_16, 'CPOR', 0, 1, 'k', marker='o', ls='', s=4, pad=8)
        self.plot_curve(ax22, self.core_16, 'CPERM', 1e-3, 1e3, 'r', semilog=True, marker='d', ls='',
s=5, pad=16)
        ax2.barrh(df.index, df['CLASS'], height=1.5, color=[colors[int(cls)-1] for cls in df['CLASS']])
        ax2.set(xlim=(0,5))
        z = np.arange(df.index[0], df.index[-1]+15, step=0.25)
        t = np.zeros(len(z))
        class_values = df['CLASS'].values
        for i in range(len(t)):
            t[i] = class_values[np.argmin(np.abs(df.index.values - z[i]))]
        for i in range(len(t)):
            t[i] = t[i-1] if t[i] == 0 else t[i]
        ax2.plot(t, z, 'k', lw=3)
        for i in range(len(colors)):
            ax2.fill_betweenx(z, 0, i+1, where=t==i+1, color=colors[i])
        ax2.invert_yaxis(); ax2.set_ylabel('Core Depth [ft]', weight='bold')
        ax2.set_xlabel('Leverett Rock Class', weight='bold')
        ax2.xaxis.set_label_position('top'); ax2.xaxis.set_ticks_position('top')
        ax2.spines['top'].set_linewidth(2)
        for ax in axs:
            ax.grid(True, which='both')
    plt.tight_layout(); plt.show()
    self.rock_cls = pd.DataFrame({'Index':z, 'Leverett':t})
    return None
plotter()
return None

def winland_class(self, figsize=(15,10), cutoffs=[0, 10, 100, 750, 2500, 4000], kexp=0.588, texp=0.732,

```

```

pexp=0.864,
    maxphi=0.4, alpha=0.15, fs=11.5, colors=['r','gold','g','b','k']):
df = self.core_16.df()[['CPOR','CPERM']].dropna()
x, y = df['CPOR'], df['CPERM']
data = y**kexp * 10**texp / x**pexp
winland_masks = []
for i in range(len(cutoffs)-1):
    mask = np.logical_and(data>=cutoffs[i], data<=cutoffs[i+1])
    winland_masks.append(mask)
color_centers = []
for i in range(len(cutoffs)-1):
    color_centers.append(np.mean([cutoffs[i], cutoffs[i+1]]))
lin_poro = np.linspace(0, maxphi, 100)
lin_perm_low, lin_perm_med, lin_perm_high = [], [], []
def winland(r35, p=lin_poro):
    return ((r35 * p**pexp) / 10**texp)**(1/kexp)
for i in range(len(color_centers)):
    lin_perm_low.append(winland(cutoffs[i]))
    lin_perm_med.append(winland(color_centers[i]))
    lin_perm_high.append(winland(cutoffs[i+1]))
df['CLASS'] = np.zeros(len(df))
for i, m in enumerate(winland_masks):
    df.loc[m, 'CLASS'] = i+1
def plotter():
    fig = plt.figure(figsize=figsize)
    gs = GridSpec(5, 2, figure=fig, width_ratios=[1.5, 1])
    ax1 = fig.add_subplot(gs[1:4, 0])
    ax2 = fig.add_subplot(gs[:, 1])
    axs = [ax1, ax2]
    for i, m in enumerate(winland_masks):
        ax1.scatter(x[m], y[m], c=colors[i], label='$R35$={:.1f}'.format(color_centers[i]))
        ax1.plot(lin_poro, lin_perm_med[i], c=colors[i])
        ax1.fill_between(lin_poro, lin_perm_low[i], lin_perm_high[i], color=colors[i], alpha=alpha)
    ax1.scatter(x, y, c='lightgray', marker='.', s=1, label='Core Data')
    ax1.set_title('Winland Rock Classification', weight='bold')
    ax1.set_yscale('log')
    ax1.set_xlabel('Core Porosity [v/v]', weight='bold'); ax1.set_ylabel('Core Permeability [mD]', weight='bold')
    ax1.legend(facecolor='wheat', edgecolor='k', bbox_to_anchor=(0.5, -0.33), loc='lower center', ncol=3, fontsize=fs)
    ax21, ax22 = ax2.twiny(), ax2.twiny()
    self.plot_curve(ax21, self.core_16, 'CPOR', 0, 1, 'k', marker='o', ls='', s=4, pad=8)
    self.plot_curve(ax22, self.core_16, 'CPERM', 1e-3, 1e3, 'r', semilog=True, marker='d', ls='', s=4, pad=8)
    ax2.bart(df.index, df['CLASS'], height=1.5, color=[colors[int(cls)-1] for cls in df['CLASS']])
    ax2.set_xlim=(0,5)
    z = np.arange(df.index[0], df.index[-1]+15, step=0.25)
    t = np.zeros(len(z))
    class_values = df['CLASS'].values
    for i in range(len(t)):
        t[i] = class_values[np.argmin(np.abs(df.index.values - z[i]))]
    for i in range(len(t)):
        t[i] = t[i-1] if t[i] == 0 else t[i]
    ax2.plot(t, z, 'k', lw=3)
    for i in range(len(colors)):
        ax2.fill_between(z, 0, i+1, where=t==i+1, color=colors[i])
    ax2.invert_yaxis(); ax2.set_ylabel('Core Depth [ft]', weight='bold')
    ax2.set_xlabel('Winland Rock Class', weight='bold')
    ax2.xaxis.set_label_position('top'); ax2.xaxis.set_ticks_position('top')
    ax2.spines['top'].set_linewidth(2)
    for ax in axs:
        ax.grid(True, which='both')
    plt.tight_layout(); plt.show()
    self.rock_cls['Winland'] = t
    return None
plotter()
return None

def lorenz_class(self, figsize=(15,10), cutoffs=[0, 0.05, 0.85, 1.5, 2, 2.75],
colors=['r','gold','g','b','k'], fs=11.5):
df = self.core_16.df()[['CPOR','CPERM']].dropna()
poro, perm = df['CPOR'], df['CPERM']
cums = df.sum(0)
cporo = np.cumsum(poro)/cums.iloc[0]
cperm = np.cumsum(perm)/cums.iloc[1]
slope = np.concatenate([[0], np.diff(cperm)/np.diff(cporo)])
z = np.cumsum(np.sort(cperm))/np.cumsum(np.sort(cporo)).max()

```

```

masks = []
for i in range(len(cutoffs)-1):
    masks.append((slope>=cutoffs[i])&(slope<=cutoffs[i+1]))
df['CLASS'] = np.zeros(len(df))
for i, m in enumerate(masks):
    df.loc[m, 'CLASS'] = i+1
w = np.arange(df.index[0], df.index[-1]+15, step=0.25)
t = np.zeros(len(w))
class_values = df['CLASS'].values
for i in range(len(t)):
    t[i] = class_values[np.argmin(np.abs(df.index-w[i]))]
for i in range(len(t)):
    t[i] = t[i-1] if t[i]==0 else t[i]
def plotter():
    fig = plt.figure(figsize=figsize)
    gs = GridSpec(5, 2, figure=fig, width_ratios=[1.5, 1])
    ax1 = fig.add_subplot(gs[0:2, 0])
    ax2 = fig.add_subplot(gs[3:5, 0])
    ax3 = fig.add_subplot(gs[:, 1])
    cmap = ListedColormap(colors)
    ax11 = ax1.twinx()
    ax1.scatter(cporo, cperm, c=slope, cmap=cmap)
    ax11.plot(cporo, slope, c='k', ls='--'); ax11.set_ylabel('Slope | Cutoffs', weight='bold',
rotation=270, labelpad=15)
    ax11.hlines(cutoffs, 0, 1, colors='k', ls=':')
    ax1.set_title('Stratigraphic modified Lorenz coefficients', weight='bold')
    ax1.set_xlabel('Cumulative Porosity []', weight='bold'); ax1.set_ylabel('Cumulative Permeability []', weight='bold')
    im2 = ax2.scatter(z, np.sort(cperm), c=z, cmap=cmap)
    cb = plt.colorbar(im2, pad=0.04, fraction=0.046); cb.set_label('Lorenz Classes', weight='bold',
labelpad=15, rotation=270)
    cb.set_ticks(np.arange(len(colors))/len(colors)); cb.set_ticklabels(range(1, len(colors)+1))
    ax2.axline([0,0],[1,1], ls='--', c='k')
    ax2.set_title('Lorenz Rock Classes', weight='bold')
    ax2.set_xlabel('Cumulative Storage Capacity', weight='bold'); ax2.set_ylabel('Cumulative Flow Capacity', weight='bold')
    ax31, ax32 = ax3.twiny(), ax3.twiny()
    self.plot_curve(ax31, self.core_16, 'CPOR', 0, 1, 'k', marker='o', ls='', s=4, pad=8)
    self.plot_curve(ax32, self.core_16, 'CPERM', 1e-3, 1e3, 'r', semilog=True, marker='d', ls='', s=5, pad=16)
    ax3.bah(df.index, df['CLASS'], height=1.5, color=[colors[int(cls)-1] for cls in df['CLASS']])
    ax3.set(xlim=(0,5))
    ax3.invert_yaxis(); ax3.set_ylabel('Depth [ft]', weight='bold')
    ax3.xaxis.set_label_position('top'); ax3.xaxis.set_ticks_position('top')
    ax3.spines['top'].set_linewidth(2)
    ax3.set_xlabel('Lorenz Rock Class', weight='bold')
    ax3.plot(t,w, 'k', lw=3)
    for i in range(len(colors)):
        ax3.fill_betweenx(w, 0, i+1, where=t==i+1, color=colors[i])
    for ax in [ax1, ax2]:
        ax.grid(True, which='both')
        ax.set(xlim=(-0.01,1.025), ylim=(-0.025,1.025))
plotter()
self.rock_cls['Lorenz'] = t
return None

def plot_w16_classes(self, figsize=(15,9), colors=['r','gold','g','b','k']):
    classes = self.rock_cls.set_index('Index')
    fig, axs = plt.subplots(1, 3, figsize=figsize, sharey=True)
    for i, ax in enumerate(axs):
        x, y = classes.iloc[:,i], classes.index
        ax.plot(x, y, color='k', lw=2)
        ax.bah(y, x, height=1.5, color=[colors[int(cls)-1] for cls in x])
        ax.set_xlabel(classes.columns[i]+ ' Classes', weight='bold')
        ax.xaxis.set_label_position('top'); ax.xaxis.set_ticks_position('top')
        ax.spines['top'].set_linewidth(2)
        ax.grid(True, which='both')
    axs[0].invert_yaxis(); axs[0].set_ylabel('Depth [ft]', weight='bold')
    plt.show()
    return None

def quantitative_por_perm_sw(self, figsize=(15,13), plim=(0,0.4), klim=(1e-3,1e3), slim=(0.4,1)):
    fig, axs = plt.subplots(2, 3, figsize=figsize, height_ratios=[4,1])
    fig.suptitle('Well 16S', weight='bold')
    ax1, ax2, ax3, ax4, ax5, ax6 = axs.flatten()
    ax2.sharey(ax1); ax3.sharey(ax1)

```

```

ax1.invert_yaxis(); ax1.set_ylabel('Depth [ft]', weight='bold')
ax11, ax21, ax31 = ax1.twiny(), ax2.twiny(), ax3.twiny()
self.plot_curve(ax1, self.phi_16, 'phiT', 1, 0, 'darkblue', units='v/v')
self.plot_curve(ax11, self.core_16, 'CPOR', 1, 0, 'r', s=5, marker='o', pad=8)
self.plot_curve(ax2, self.phi_16, 'k_wr', 1e-3, 1e3, semilog=True, units='mD')
self.plot_curve(ax21, self.core_16, 'CPERM', 1e-3, 1e3, 'r', semilog=True, s=5, marker='o', pad=8)
self.plot_curve(ax3, self.phi_16, 'Sw', 1, 0, 'b')
self.plot_curve(ax31, self.core_16, 'CSW', 1, 0, 'r', s=5, marker='o', pad=8)
x = self.phi_16
y = self.core_16.df()
y = y[y.index.isin(x.index)]
lims = [plim, klim, slim]
titles = ['Porosity', 'Permeability', 'Water Saturation']
for i, ax in enumerate([ax4, ax5, ax6]):
    ax.grid(True, which='both', alpha=0.5)
    ax.set(xlim=lims[i], ylim=lims[i])
    ax.set_xlabel('Log Values', weight='bold')
    ax.axline([0,0], [1,1], color='r')
    ax.set_title(titles[i], weight='bold')
ax4.set_ylabel('Core Values', weight='bold')
ax4.scatter(x['phiT'], y['CPOR'])
ax5.scatter(x['k_wr'], y['CPERM'])
ax5.set_xscale('log'); ax5.set_yscale('log')
ax6.scatter(x['Sw'], y['CSW'])
plt.show()
return None

def leverett_j_function(self, perm_oil=0.3, perm_gas=0.05, lambda_reg=1e-3, swirr=0.05,
                      sigma_cos_oil=26, sigma_cos_gas=50, method='CG', figsize=(4,10)):
    jfunc = self.pc_data['Lev "J(SHG)"]'
    def objective(variables, *args):
        a, b = variables
        swfunc, jfunc = args[0], args[1]
        error = swfunc - a * jfunc**b + swirr
        return linalg.norm(error) + lambda_reg*linalg.norm(variables)
    df = self.pc_data[['Equiv Brine Sat. (Frac.)','Lev "J(SHG)"]']
    solutions = []
    sol, jac, nfev = [], [], []
    for index, row in df.iterrows():
        swfunc, jfunc = df['Equiv Brine Sat. (Frac.)'], df['Lev "J(SHG)"]'
        solution = optimize.minimize(objective,
                                      x0      = [1,1],
                                      args    = (swfunc, jfunc),
                                      method  = method,
                                      tol     = 1e-6,
                                      options = {'maxiter': 1000})
        sol.append(solution.fun); jac.append(solution.jac); nfev.append(solution.nfev)
        solutions.append({'a':solution.x[0], 'b':solution.x[1]})

    solutions = pd.DataFrame(solutions)
    phi_mean = self.core_16.df()['CPOR'].mean()
    a, b = solutions['a'].iloc[-1], solutions['b'].iloc[-1]
    def plotter():
        _, ax = plt.subplots(1, 1, figsize=figsize)
        self.plot_curve(ax, self.phi_16, 'Sw', 1, 0, 'dodgerblue', bar=True, alpha=0.8)
        self.plot_curve(ax, self.phi_16, 'Sw', 1, 0, 'darkblue', s=0.5)
        oil = a*((self.pc_data['Hg Injection Pressure (psia)'].values * np.sqrt(perm_oil/phi_mean)) /
(2*sigma_cos_oil))**b
        gas = a*((self.pc_data['Hg Injection Pressure (psia)'].values * np.sqrt(perm_gas/phi_mean)) /
(2*sigma_cos_gas))**b
        tel = a*((self.pc_data['Hg Injection Pressure (psia)'].values * np.sqrt(10.0/phi_mean)) /
(2*sigma_cos_oil))**b
        te2 = a*((self.pc_data['Hg Injection Pressure (psia)'].values * np.sqrt(1.00/phi_mean)) /
(2*sigma_cos_oil))**b
        te3 = a*((self.pc_data['Hg Injection Pressure (psia)'].values * np.sqrt(0.10/phi_mean)) /
(2*sigma_cos_oil))**b
        te4 = a*((self.pc_data['Hg Injection Pressure (psia)'].values * np.sqrt(0.01/phi_mean)) /
(2*sigma_cos_oil))**b
        z = np.linspace(self.phi_16.index[0], self.phi_16.index[-1], len(oil))
        ax.plot(oil[::-1], z, 'g', lw=3, label='Oil Leg')
        ax.plot(gas[::-1], z, 'r', lw=3, label='Gas Leg')
        ax.plot(tel[::-1], z, 'k--', lw=2, label='k=10'); ax.plot(te2[::-1], z, 'k--', lw=2,
label='k=0.1')
        ax.plot(te3[::-1], z, 'k--', lw=2, label='k=0.01'); ax.plot(te4[::-1], z, 'k--', lw=2,
label='k=0.001')
        ax.legend(facecolor='wheat', edgecolor='k'); ax.invert_yaxis()
        plt.show()
    return None

```

```

plotter()
return None

def mineral_inversion_08(self, components=['Quartz','Calcite','Porosity'], colors=['r','g','darkblue'],
                      lambda_reg=1e-6, x0=[0.75, 0.1, 0.15], maxiter=1000, tol=1e-10,
figsize=(12,8)):
    data = self.logs_8.df()[['ZDEN','PE','NPHI','DT']].dropna()
    def objective(variables, *args):
        c1, c2, por = variables
        rhob, pef, nphi, dt = args
        eq1 = 2.65*c1 + 2.71*c2 + 1.00*por - rhob
        eq2 = 1.80*c1 + 5.10*c2 + 0.00*por - pef
        eq3 = -0.04*c1 + 0.00*c2 + 1.00*por - nphi
        eq4 = 51.3*c1 + 47.6*c2 + 189*por - dt
        eq5 = c1 + c2 + por - 1
        sol = np.array([eq1/2.71, eq2/5.10, eq3/1, eq4/189, eq5/1])
        return linalg.norm(sol) + lambda_reg*linalg.norm(variables)
    def constraint(variables):
        return np.sum(variables) - 1
    solutions = []
    for index, row in data.iterrows():
        rhob, pef, nphi, dt = row['ZDEN'], row['PE'], row['NPHI'], row['DT']
        solution = optimize.minimize(fun = objective,
                                      x0 = x0,
                                      args = (rhob, pef, nphi, dt),
                                      method = 'SLSQP',
                                      constraints = {'type':'eq', 'fun':constraint},
                                      bounds = [(0,1),(0,1),(0,1)],
                                      jac = '3-point',
                                      tol = tol,
                                      options = {'maxiter':maxiter})
        solutions.append({'Quartz':solution.x[0], 'Calcite':solution.x[1],
'Porosity':solution.x[2],
'Fun':solution.fun, 'Jac':linalg.norm(solution.jac),
'Nfev':solution.nfev})
    self.mineral_inv_08 = pd.DataFrame(solutions, index=data.index)
    def plotter():
        fig, axs = plt.subplots(1, 5, figsize=figsize, sharey=True)
        prev_sum = 0
        for i in range(3):
            axs[i].plot(self.mineral_inv_08.iloc[:,i], self.mineral_inv_08.index, color=colors[i])
            axs[i].set_title(components[i], weight='bold')
            axs[-2].barh(self.mineral_inv_08.index, self.mineral_inv_08.iloc[:,i], left=prev_sum,
color=colors[i])
            prev_sum += self.mineral_inv_08.iloc[:,i]
        for ax in axs:
            ax.set_xlim(0,1)
            ax.grid(True, which='both')
        ax = axs[-1]
        ax1, ax2, ax3 = ax.twiny(), ax.twiny(), ax.twiny()
        axn = [ax, ax1, ax2, ax3]
        ax.plot(self.mineral_inv_08['Fun'], self.mineral_inv_08.index, color='k')
        ax1.plot(self.mineral_inv_08['Jac'], self.mineral_inv_08.index, color='darkred')
        ax2.plot(self.mineral_inv_08['Nfev'], self.mineral_inv_08.index, color='darkblue')
        ax3.plot(self.mineral_inv_08[components].sum(1), self.mineral_inv_08.index, color='m')
        ax3.set(xlim=(0.95,1.05))
        colors2 = ['k','darkred','darkblue','m']
        for i, ax in enumerate(axn):
            ax.xaxis.set_label_position('top')
            ax.xaxis.set_ticks_position('top')
            ax.xaxis.set_tick_params(color=colors2[i], labelcolor=colors2[i])
            ax.spines['top'].set_position((('axes',1+0.08*i)))
            ax.set_xlabel(['Objective Function','Jacobian Norm','NFev','Material Balance'][i],
weight='bold', color=colors2[i])
            ax[0].set_ylabel('Depth [ft]', weight='bold'); ax[0].invert_yaxis()
            ax[-2].set_title('Composition', weight='bold')
            fig.suptitle('Well 8', weight='bold')
            plt.tight_layout(); plt.show()
        return None
    plotter()
    return None

def mineral_inversion_16(self, components=['Quartz','Calcite','Porosity'], colors=['r','g','darkblue'],
                      lambda_reg=1e-6, x0=[0.75, 0.1, 0.15], maxiter=1000, tol=1e-10,
figsize=(12,8)):
    data = self.logs_16.df()[['RHOZ','PEFZ','DTCO']].dropna().loc[6300:9002]

```

```

data = data.join(self.corr_vals['w16_calcvalues']['NPHI_Corr']).dropna()
def objective(variables, *args):
    c1, c2, por = variables
    rhob, pef, nphi, dt = args
    eq1 = 2.65*c1 + 2.71*c2 + 1.00*por - rhob
    eq2 = 1.80*c1 + 5.10*c2 + 0.00*por - pef
    eq3 = -0.04*c1 + 0.00*c2 + 1.00*por - nphi
    eq4 = 51.3*c1 + 47.6*c2 + 189*por - dt
    eq5 = c1 + c2 + por - 1
    sol = np.array([eq1/2.71, eq2/5.10, eq3/1, eq4/189, eq5/1])
    return linalg.norm(sol) + lambda_reg*linalg.norm(variables)
def constraint(variables):
    return np.sum(variables) -1
solutions = []
for index, row in data.iterrows():
    rhob, pef, nphi, dt = row['RHOZ'], row['PEFZ'], row['NPHI_Corr'], row['DTCO']
    solution = optimize.minimize(fun = objective,
                                  x0 = x0,
                                  args = (rhob, pef, nphi, dt),
                                  method = 'SLSQP',
                                  constraints = {'type':'eq', 'fun':constraint},
                                  bounds = [(0,1),(0,1),(0,1)],
                                  jac = '3-point',
                                  tol = tol,
                                  options = {'maxiter':maxiter})
    solutions.append({'Quartz':solution.x[0], 'Calcite':solution.x[1],
                      'Porosity':solution.x[2],
                      'Fun':solution.fun, 'Jac':linalg.norm(solution.jac),
                      'Nfev':solution.nfev})
self.mineral_inv_16 = pd.DataFrame(solutions, index=data.index)
def plotter():
    fig, axs = plt.subplots(1, 5, figsize=figsize, sharey=True)
    prev_sum = 0
    for i in range(3):
        axs[i].plot(self.mineral_inv_16.iloc[:,i], self.mineral_inv_16.index, color=colors[i])
        axs[i].set_title(components[i], weight='bold')
        axs[-2].barh(self.mineral_inv_16.index, self.mineral_inv_16.iloc[:,i], left=prev_sum,
                     color=colors[i])
        prev_sum += self.mineral_inv_16.iloc[:,i]
    for ax in axs:
        ax.set_xlim(0,1)
        ax.grid(True, which='both')
    ax = axs[-1]
    ax1, ax2, ax3 = ax.twiny(), ax.twiny(), ax.twiny()
    axn = [ax, ax1, ax2, ax3]
    ax.plot(self.mineral_inv_16['Fun'], self.mineral_inv_16.index, color='k')
    ax1.plot(self.mineral_inv_16['Jac'], self.mineral_inv_16.index, color='darkred')
    ax2.plot(self.mineral_inv_16['Nfev'], self.mineral_inv_16.index, color='darkblue')
    ax3.plot(self.mineral_inv_16[components].sum(1), self.mineral_inv_16.index, color='m')
    ax3.set(xlim=(0.95,1.05))
    colors2 = ['k','darkred','darkblue','m']
    for i, ax in enumerate(axn):
        ax.xaxis.set_label_position('top')
        ax.xaxis.set_ticks_position('top')
        ax.xaxis.set_tick_params(color=colors2[i], labelcolor=colors2[i])
        ax.spines['top'].set_position((('axes',1+0.08*i)))
        ax.set_xlabel(['Objective Function', 'Jacobian Norm', 'NFev', 'Material Balance'][i],
                     weight='bold', color=colors2[i])
    axs[0].set_ylabel('Depth [ft]', weight='bold'); axs[0].invert_yaxis()
    axs[-2].set_title('Composition', weight='bold')
    fig.suptitle('Well 16S', weight='bold')
    plt.tight_layout(); plt.show()
    return None
plotter()
return None

def well_to_well_correlation(self, figsize=(15,15)):
    pipeline = make_pipeline(MinMaxScaler(), KMeans(5, n_init=50))
    d08 = self.mineral_inv_08[['Quartz','Calcite','Porosity']].loc[5750:]
    d16 = self.mineral_inv_16[['Quartz','Calcite','Porosity']]
    pipeline.fit(d08)
    labels08 = pipeline.predict(d08)
    labels16 = pipeline.predict(d16)
    labs08 = pd.DataFrame(labels08, columns=['Cluster'], index=d08.index)
    labs16 = pd.DataFrame(labels16, columns=['Cluster'], index=d16.index)
    fig, axs = plt.subplots(1, 6, figsize=figsize, sharey=True)
    ax1, ax2, ax3, ax4, ax5, ax6 = axs

```

```

colors = ['r', 'gold', 'g', 'b', 'k']
z = np.arange(labs08.index[0], labs08.index[-1], step=0.25)
t = np.zeros(len(z))
class_values = labs08['Cluster'].values
for i in range(len(t)):
    t[i] = class_values[np.argmin(np.abs(labs08.index.values - z[i]))]
for i in range(len(t)):
    t[i] = t[i-1] if t[i]==0 else t[i]
for i in range(len(colors)):
    ax1.fill_betweenx(z, 0, i+1, where=t==i+1, color=colors[i], alpha=0.5)
    ax2.fill_betweenx(z, 0, i+1, where=t==i+1, color=colors[i], alpha=0.5)
ax11, ax12, ax13 = ax1.twiny(), ax1.twiny(), ax1.twiny()
self.plot_curve(ax11, self.logs_8, 'GR', 0, 200, 'k')
self.plot_curve(ax12, self.logs_8, 'M0R1', 0.2, 2000, 'b', semilog=True, pad=8)
self.plot_curve(ax13, self.logs_8, 'M0R9', 0.2, 2000, 'r', semilog=True, pad=16)
ax21, ax22 = ax2.twiny(), ax2.twiny()
self.plot_curve(ax2, self.phi_08, 'phiT', 1, 0, 'r')
self.plot_curve(ax21, self.phi_08, 'k_wr', 1e-3, 1e3, 'k', semilog=True, pad=8)
self.plot_curve(ax22, self.phi_08, 'Sw', 1, 0, 'b', pad=16)
ax31, ax32 = ax3.twiny(), ax3.twiny()
self.plot_curve(ax3, self.mineral_inv_08, 'Quartz', 0, 1, 'r')
self.plot_curve(ax31, self.mineral_inv_08, 'Calcite', 0, 1, 'g', pad=8)
self.plot_curve(ax32, self.mineral_inv_08, 'Porosity', 0, 1, 'darkblue', pad=16)
z = np.arange(labs16.index[0], labs16.index[-1], step=0.25)
t = np.zeros(len(z))
class_values = labs16['Cluster'].values
for i in range(len(t)):
    t[i] = class_values[np.argmin(np.abs(labs16.index.values - z[i]))]
for i in range(len(t)):
    t[i] = t[i-1] if t[i]==0 else t[i]
for i in range(len(colors)):
    ax4.fill_betweenx(z, 0, i+1, where=t==i+1, color=colors[i], alpha=0.5)
    ax5.fill_betweenx(z, 0, i+1, where=t==i+1, color=colors[i], alpha=0.5)
ax41, ax42, ax43 = ax4.twiny(), ax4.twiny(), ax4.twiny()
self.plot_curve(ax41, self.logs_16, 'HGR', 0, 200, 'k')
self.plot_curve(ax42, self.logs_16, 'AT10', 0.2, 2000, 'b', semilog=True, pad=8)
self.plot_curve(ax43, self.logs_16, 'AT90', 0.2, 2000, 'r', semilog=True, pad=16)
ax51, ax52 = ax5.twiny(), ax5.twiny()
self.plot_curve(ax5, self.phi_16, 'phiT', 1, 0, 'r')
self.plot_curve(ax51, self.phi_16, 'k_wr', 1e-3, 1e3, 'k', semilog=True, pad=8)
self.plot_curve(ax52, self.phi_16, 'Sw', 1, 0, 'b', pad=16)
ax61, ax62 = ax6.twiny(), ax6.twiny()
self.plot_curve(ax6, self.mineral_inv_16, 'Quartz', 0, 1, 'r')
self.plot_curve(ax61, self.mineral_inv_16, 'Calcite', 0, 1, 'g', pad=8)
self.plot_curve(ax62, self.mineral_inv_16, 'Porosity', 0, 1, 'darkblue', pad=16)
axs[0].invert_yaxis(); axs[0].set_ylabel('Depth [ft]', weight='bold')
plt.tight_layout(); plt.show()
return None

```

```

def plot_sonic_logs(self, figsize=(15,8)):
    fig, axs = plt.subplots(1, 6, figsize=figsize, sharey=True)
    ax1, ax2, ax3, ax4, ax5, ax6 = axs
    titles = ['Well 08', 'Well 16S']
    d08 = self.logs_8.df()
    d16 = self.logs_16.df()
    dt08 = pd.DataFrame({'phi':(d08['DT']-47.6) / (189-47.6),
                         'mat':(d08['DT'] - self.phi_08['phiT']*189) / (1-self.phi_08['phiT']),
                         'flu':(47.6 + (1/self.phi_08['phiT'])*(d08['DT']-47.6))})
    dt16 = pd.DataFrame({'phi':(d16['DTCO']-47.6) / (189-47.6),
                         'mat':(d16['DTCO'] - self.phi_16['phiT']*189) / (1-self.phi_16['phiT']),
                         'flu':(47.6 + (1/self.phi_16['phiT'])*(d16['DTCO']-47.6))})
    ax11, ax12 = ax1.twiny(), ax1.twiny()
    self.plot_curve(ax1, self.logs_8, 'DT', 40, 175)
    self.plot_curve(ax11, dt08, 'phi', 1, 0, 'tab:orange', units='sonic', pad=8)
    self.plot_curve(ax12, self.phi_08, 'phiT', 1, 0, 'r', units='inversion', pad=16)
    self.plot_curve(ax2, dt08, 'mat', units='matrix sonic')
    ax2.vlines(47.6, 0, 8500, 'r', label='calcite'); ax2.vlines(51.3, 0, 8500, 'b', label='quartz')
    ax2.legend(facecolor='lightgray', edgecolor='k')
    self.plot_curve(ax3, dt08, 'flu', units='fluid sonic'); ax3.set(xlim=(0,1000))
    ax3.vlines(189, 0, 8500, 'r', label='water'); ax3.legend(facecolor='lightgray', edgecolor='k')
    ax41, ax42 = ax4.twiny(), ax4.twiny()
    self.plot_curve(ax4, self.logs_16, 'DTCO', 40, 175)
    self.plot_curve(ax41, dt16, 'phi', 1, 0, 'tab:orange', units='sonic', pad=8)
    self.plot_curve(ax42, self.phi_16, 'phiT', 1, 0, 'r', units='inversion', pad=16)
    self.plot_curve(ax5, dt16, 'mat', units='matrix sonic')
    ax5.vlines(47.6, 0, 8500, 'r', label='calcite'); ax5.vlines(51.3, 0, 8500, 'b', label='quartz')

```

```

ax5.legend(facecolor='lightgray', edgecolor='k'); ax5.set(xlim=(0,100))
self.plot_curve(ax6, dt16, 'flu', units='fluid sonic'); ax6.set(xlim=(0,1000))
ax6.vlines(189, 0, 8500, 'r', label='water'); ax6.legend(facecolor='lightgray', edgecolor='k')
for i, ax in enumerate([ax1, ax4]):
    ax.set_title(titles[i], weight='bold')
ax1.invert_yaxis()
plt.show()
return None

def biot_gassmann(self, shear=32, bulk=77, figsize=(15,10)):
    Vs08 = np.sqrt(shear/self.logs_8.df()['ZDEN'].dropna())
    Vp08 = np.sqrt((bulk + (4/3)*shear)/self.logs_8.df()['ZDEN'].dropna())
    gass08 = pd.DataFrame({'Vp':Vp08, 'Vs':Vs08})
    gass08.index = self.logs_8.df()['ZDEN'].dropna().index
    Vs16 = np.sqrt(shear/self.logs_16.df()['RHOZ'].dropna())
    Vp16 = np.sqrt((bulk + (4/3)*shear)/self.logs_16.df()['RHOZ'].dropna())
    gass16 = pd.DataFrame({'Vp':Vp16, 'Vs':Vs16})
    gass16.index = self.logs_16.df()['RHOZ'].dropna().index
    fig, axs = plt.subplots(1, 4, figsize=figsize, sharey=True)
    ax1, ax2, ax3, ax4 = axs
    self.plot_curve(ax1, self.logs_8, 'DT', 0, 180, 'tab:orange')
    ax21 = ax2.twiny()
    self.plot_curve(ax2, gass08, 'Vp', 6, 10, 'tab:red')
    self.plot_curve(ax21, gass08, 'Vs', 2, 6, 'tab:green', pad=8)
    ax31 = ax3.twiny()
    self.plot_curve(ax3, self.logs_16, 'DTCO', 0, 180, 'tab:orange')
    self.plot_curve(ax31, self.logs_16, 'DTSM', 0, 480, 'tab:blue', pad=8)
    ax41 = ax4.twiny()
    self.plot_curve(ax4, gass16, 'Vp', 6, 9, 'tab:red')
    self.plot_curve(ax41, gass16, 'Vs', 3, 5, 'tab:green', pad=8)
    ax1.set_title('Well 8', weight='bold'); ax3.set_title('Well 16S', weight='bold')
    ax1.invert_yaxis()
    plt.show()

#####
##### MAIN #####
#####

if __name__ == '__main__':
    petro = Petrophysics()
    petro.plot_full_log_8(apply_mask=True)
    petro.plot_full_log_16(apply_mask=True)
    petro.zonation()
    petro.plot_zonation()
    petro.plot_GRvAT90()
    petro.plot_KvsTh()
    petro.plot_GRvPEFvRHOB()
    petro.plot_RHOB_NPHI()
    petro.resistivity_inversion()
    petro.plot_resistivity_inversion()
    petro.plot_csh()
    petro.plot_fpres()
    petro.plot_phi()
    petro.plot_phi_perm()
    petro.plot_core_log_poro_perm()
    petro.plot_thomas_stieber()
    petro.plot_pickett()
    petro.archie_sw()
    petro.plot_archie_sw()
    petro.leverett_class()
    petro.winland_class()
    petro.lorenz_class()
    petro.plot_w16_classes()
    petro.quantitative_por_perm_sw()
    petro.leverett_j_function()
    petro.mineral_inversion_08()
    petro.mineral_inversion_16()
    petro.well_to_well_correlation()
    petro.plot_sonic_logs()
    petro.biot_gassmann()

#####
##### END #####
#####

def yomi_co_krig():
    from gstoools import Gaussian, SRF, Spherical
    from matplotlib.pyplot import figure
    from pykrige import SimpleKriging

```

```

### Fit Variogram
x = np.array(LN1_core["DEPTH"])
y = np.array(LN1_core["lnk"])

model = Exponential(dim=2, var=2, len_scale=7)
srf = SRF(model, mean=0, seed=19970221)
field = srf((x, y))
bins = np.arange(20)
bin_center, gamma = vario_estimate((x, y), field, bins)
models = {
    "Gaussian": gs.Gaussian,
    "Exponential": gs.Exponential,
    "Stable": gs.Stable,
    "Rational": gs.Rational,
    "Circular": gs.Circular,
    "Spherical": gs.Spherical,
    "SuperSpherical": gs.SuperSpherical}
scores = {}
figure(figsize=(10, 7), dpi=100)
# plot the estimated variogram
plt.scatter(bin_center, gamma, color="k", label="data")
ax = plt.gca()
# fit all models to the estimated variogram
for model in models:
    fit_model = models[model](dim=2)
    para, pcov, r2 = fit_model.fit_variogram(bin_center, gamma, return_r2=True)
    fit_model.plot(x_max=20, ax=ax)
    scores[model] = r2
# plt.plot(xx,yy,color='red',linewidth=4)
plt.xlabel("Lag Distance (m)", fontweight='bold', fontsize=15)
plt.ylabel("Variogram", fontweight='bold', fontsize=15)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.title("Permeability (lnk)", fontweight='bold', fontsize=25)
print(fit_model)
plt.legend()
ranking = sorted(scores.items(), key=lambda item: item[1], reverse=True)
print("RANKING by Pseudo-r2 score")
for i, (model, score) in enumerate(ranking, 1):
    print(f"{i}>6}. {model}>15}: {score:.5}")
plt.show()

### Detecting Stationarity
from statsmodels.tsa.stattools import acf, pacf
nlags = 20
lag_acf = acf(y, nlags=nlags)
lag_pacf = pacf(y, nlags=nlags, method='ols')
lags = np.linspace(0, (nlags+1)*10, nlags+1)
#Plot ACF:
plt.subplot(121)
plt.plot(lags, lag_acf, color='red')
plt.xlabel("Lags", fontweight='bold', fontsize=15)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.title('Autocorrelation Function', fontweight='bold', fontsize=25)
#Plot PACF:
plt.subplot(122)
plt.plot(lags, lag_pacf, color='red')
plt.xlabel("Lags", fontweight='bold', fontsize=15)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.title('Partial Autocorrelation Function', fontweight='bold', fontsize=25)
plt.tight_layout()
plt.subplots_adjust(left=0.0, bottom=0.0, right=3.0, top=1.0, wspace=0.2, hspace=0.2)
print('Dickey-Fuller Test Results:')
dftest = adfuller(y, autolag='AIC')
dfoutput = pd.Series(dftest[0:4], index=['Test Statistic', 'p-value', '#Lags Used', 'Number of Observations Used'])
for key, value in dftest[4].items():
    dfoutput['Critical Value (%s)' % key] = value
print(dfoutput[0:7])

### Kriging Permeability
plt.style.use("default")
X_D = np.array(LN1_core["DEPTH"])
X_pred_D = np.array(LN1_log["DEPTH"])
X = np.array(LN1_core["CRHOB"])

```

```

y = np.array(LN1_core["lnk"])
X_pred = np.array(LN1_log["RHOZ"])
#uk = SimpleKriging(X, np.zeros(X.shape), y, variogram_model="linear",variogram_parameters = {'slope': 2, 'nugget': 0})
uk = SimpleKriging(X, np.zeros(X.shape), y, variogram_model="gaussian",variogram_parameters = {'psill': 0.4, 'range': 0.4, 'nugget': 0})
uk_d = SimpleKriging(X, np.zeros(X.shape), y, variogram_model="gaussian",variogram_parameters = {'psill': 0.7, 'range': 0.7, 'nugget': 0})
uk_s = SimpleKriging(X, np.zeros(X.shape), y, variogram_model="exponential")
y_pred, y_std = uk.execute("grid", X_pred, np.array([0.0]))
y_pred_d, y_std_d = uk_d.execute("grid", X_pred, np.array([0.0]))
y_pred_s, y_std_s = uk_s.execute("grid", X_pred, np.array([0.0]))
y_pred = np.squeeze(y_pred)
y_std = np.squeeze(y_std)
y_pred_d = np.squeeze(y_pred_d)
y_std_d = np.squeeze(y_std_d)
y_pred_s = np.squeeze(y_pred_s)
y_std_s = np.squeeze(y_std_s)

fig, ax = plt.subplots(1, 1, figsize=(4, 10))
ax.scatter(y, X_D, s=20,label="Core Permeability", color = 'green')
ax.plot(y_pred_s,X_pred_D, label="Exponential", color = 'blue')
ax.legend(loc=4)
ax.set_xlabel("log Permeability")
ax.set_ylabel("Depth (m)")
ax.set_ylim(6300,9002)
ax.xaxis.set_label_position('top')
ax.xaxis.tick_top()
ax.invert_yaxis()
ax.set_xlim(-8, 10)
plt.show()

### Evaluating Results
window_size = 2 # assume window size of 10 days
#Determining rolling statistics
rolling_mean = df.rolling(window = window_size, center = True).mean()
rolling_std = df.rolling(window = window_size, center = True).var()
rolling_P025 = df.rolling(window = window_size, center = True).quantile(.025)
rolling_P975 = df.rolling(window = window_size, center = True).quantile(.975)
#Determining rolling statistics
window_size2 = 30
rolling_mean2 = df2.rolling(window = window_size2, center = True).mean()
rolling_std2 = df2.rolling(window = window_size2, center = True).var()
rolling_P0252 = df2.rolling(window = window_size2, center = True).quantile(.025)
rolling_P9752 = df2.rolling(window = window_size2, center = True).quantile(.975)
#Plot rolling statistics:
plt.subplot(311)
orig = plt.plot(df["lnk"], color='green', linewidth = 1, label='Original')
mean = plt.plot(rolling_mean["lnk"], color='blue', linewidth = 1, label='Rolling Mean')
P025 = plt.plot(rolling_P025, color='grey', linewidth = 1, label='Rolling P025')
P975 = plt.plot(rolling_P975, color='grey', linewidth = 1, label='Rolling P975')
plt.title('Rolling Statistics_Core', fontweight='bold', fontsize=20); plt.legend(loc='best')
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.subplot(312)
orig2 = plt.plot(df2["Pred"], color='green', linewidth = 1, label='Original')
mean2 = plt.plot(rolling_mean2["Pred"], color='blue', linewidth = 1, label='Rolling Mean')
P0252 = plt.plot(rolling_P0252, color='grey', linewidth = 1, label='Rolling P025')
P9752 = plt.plot(rolling_P9752, color='grey', linewidth = 1, label='Rolling P975')
plt.title('Rolling Statistics_Cokrig', fontweight='bold', fontsize=20); plt.legend(loc='best')
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.subplot(313)
std = plt.plot(rolling_std["lnk"], color='red', linewidth = 1, label = 'Rolling Var_Core')
std = plt.plot(rolling_std2["Pred"], color='blue', linewidth = 1, label = 'Rolling Var_Krig_Exp')
plt.legend(loc='best'); plt.title('Rolling Variance_Core vs Krig', fontweight='bold', fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.subplots_adjust(left=0.0, bottom=0.0, right=2.0, top=2.0, wspace=0.5, hspace=0.4)

```