Essential Guide to Data Science for Petroleum Geoscientists and Engineers

Yohanes Nuwara

07.09.2024

Contents

Unit 1. Very Brief Intro to Numpy, Matplotlib, Pandas, and Scipy

This is our Google Colab notebook. A notebook is where we will write codes, stream and import dataset, run them, and see the results. Your local computer doesn't do the work, but your internet does (because Google Colab is a Cloud IDE).

First of all, we will import our GitHub repository, that later on we can stream and import the data from.

```
!git clone 'https://github.com/yohanesnuwara/python-bootcamp-for-geoengineers'
```

```
Cloning into 'python-bootcamp-for-geoengineers'...
remote: Enumerating objects: 3, done.ote: Counting objects: 100% (3/3), done.ote: Compressing objects: 100% (3/3), done.ote: Total 156 (delta 0), reused 0 (delta 0), pack-reused 153
```

This notebook gives a very 3x brief introduction to Numpy, Matplotlib, Pandas, and Scipy.

Now, we will import libraries.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy
```

After this we will go through each library (how to use them).

Numpy

Numpy library is widely used for numerical computations.

The objectives of this section are:

- Handle arrays (1D, 2D, and 3D)
- List comprehension
- Data cleansing
- Element search
- Read (and write) file

Handle arrays (1D, 2D, and 3D)

1D array

Ways to create an array are:

- `np.array`: array consists of several values
- `np.arange`: array of series from a start value to an end value, with a specified increment
- `np.linspace`: array of series between two values, with a specified number o elements

```
# create array with specified values
array1 = np.array([10, 11, 45, 75, 65, 78, 90, 85, 56, 77])
print(array1)

# create an array, say from 1 to 50, with a specified increment, say 5
array2 = np.arange(1, 50, 5)
print(array2)

# create a 1D numpy array consisting of numbers from 1 to 100, divided
uniformly into 100 numbers
array3 = np.linspace(1, 100, 10)
print(array3)
```

```
[10 11 45 75 65 78 90 85 56 77]
[ 1  6 11 16 21 26 31 36 41 46]
[  1.  12.  23.  34.  45.  56.  67.  78.  89. 100.]
```

Print the length of each array

```
print(len(array1))
print(len(array2))
print(len(array3))
```

```
10
10
10
```

Now, instead of repetitive codes, we can use `for` loop

```
arrays = np.array([array1, array2, array3])

for i in range(len(arrays)):
  print(len(arrays[i]))
```

```
10
10
10
```

Now, sort the elements in the `array1` in ascending order (smallest to highest number)

```
print('Sorted from smallest to highest:', np.sort(array1))
```

```
Sorted from smallest to highest: [10 11 45 56 65 75 77 78 85 90]
```

To sort with descending order, we can use alternative `[::-1]`

```
print('Sorted from highest to smallest:', array1[::-1])
```

```
Sorted from highest to smallest: [77 56 85 90 78 65 75 45 11 10]
```

Numpy array is unique because indexing starts from 0, not 1. Try printing the first element of the array.

```
print('First element:', array1[0])
print('Second element:', array1[1])
```

```
First element: 10
Second element: 11
```

Print the very last element N and its preceding element $N-1$

```
print('Last element N:', array1[-1])
print('Element N-1:', array1[-2])
```

```
Last element N: 77
Element N-1: 56
```

We can also print the first three elements of the array

```
print('First five elements:', array1[:3])
```

```
First five elements: [10 11 45]
```

Print the 5th until 8th element

```
print('Fifth until eighth elements:', array1[5:8])
```

```
Fifth until eighth elements: [78 90 85]
```

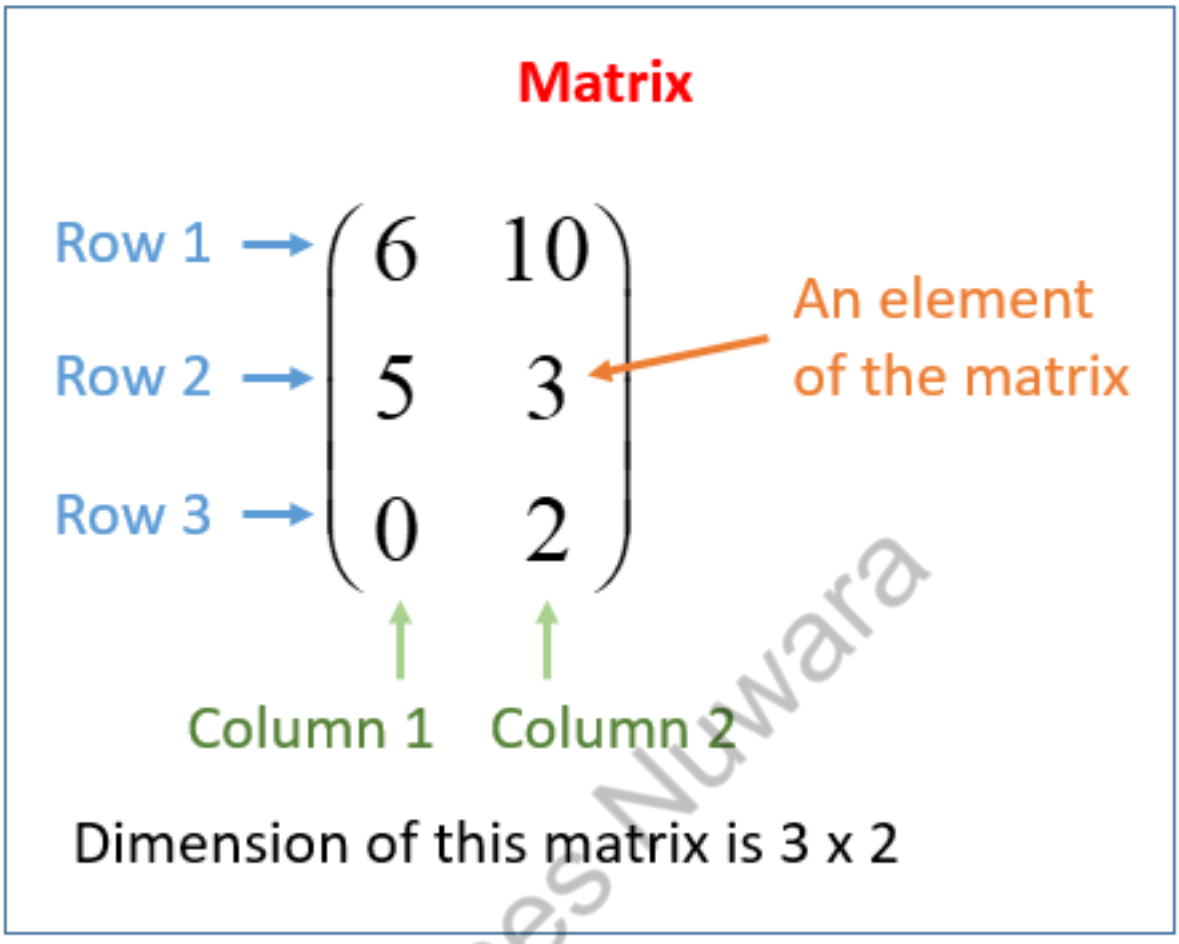
Print the last four elements of the array

```
print('Last four elements:', array1[-4:])
```

```
Last four elements: [90 85 56 77]
```

2D array (Matrix)

In math, we know 2D array as a matrix. We create matrix also using `np.array`



To create a matrix is simple. Imagine you have 3 arrays, each consisting of 3 elements.

Array 1: $[10, 20, 30)$

Array 2: $[50, 70, 90)$

Array 3: $[12, 14, 16)$

Then, you stack them. Now, you will have a (3×3) matrix.

$$\begin{bmatrix} 10 & 20 & 30 \\ 50 & 70 & 90 \\ 12 & 14 & 16 \end{bmatrix}$$

Likewise, using Numpy to stack these arrays are very simple.

You can build it step-by-step, from the 1st row to the 3rd row

```
# each row as 1D array
first = np.array([10, 20, 30])
second = np.array([50, 70, 90])
third = np.array([12, 14, 16])
```

```
# stack them together into 2D array
M = np.array([first,
              second,
              third])
M
```

```
array([[10, 20, 30],
       [50, 70, 90],
       [12, 14, 16]])
```

OR, you can build it directly!

You already know how to make 1D numpy array by `np.array([...])`. There's only one squared bracket. So, for a matrix, which is a 2D numpy array, use `np.array([[...]])` with two squared brackets.

```
M = np.array([[10, 20, 30],
              [50, 70, 90],
              [12, 14, 16]])
M
```

```
array([[10, 20, 30],
       [50, 70, 90],
       [12, 14, 16]])
```

Print the matrix shape

```
M.shape
```

```
(3, 3)
```

Remember that Python indexing starts from 0. So, if you want to print the element $M_{1,1}$, pass `M[0,0]`

```
print('Element 1,1:', M[0,0])
```

```
Element 1,1: 10
```

Likewise, print element $M_{1,3}$

```
print('Element 1,3:', M[0,2])
```

```
Element 1,3: 30
```

Print all elements in row 2, or element $M_{2,n}$

```
print('All elements in second row:', M[1,:])
```

```
All elements in second row: [50 70 90]
```

And print all elements in column 2, or element $M_{n,2}$

```
print('All elements in second column:', M[:,1])
```

```
All elements in second column: [20 70 14]
```

Data cleansing

Create an array that consists of NaN values with `np.nan`

```
my_array = np.array([np.nan, 15, np.nan, 20, np.nan, 34, np.nan,
np.nan, 67, 30, 10, np.nan, 34, np.nan, 50, 25, np.nan])
```

Check if there is `NaN` values in an array. Returns `True` if there's any. Unless, it returns `False` (Boolean argument).

```
np.isnan(my_array).any()
```

```
True
```

Removing `NaN` values from the array (just delete the `NaN` elements)

```
nan_remove = my_array[~np.isnan(my_array)]
nan_remove
```

```
array([15., 20., 34., 67., 30., 10., 34., 50., 25.])
```

Replacing (imputing) `NaN` values with any number. E.g. 0

```
# replace with zeros

# first make a new array to store the imputed data, name it new_array
new_array = np.array(my_array)

# replace with zeros
new_array[np.isnan(new_array)]=0
new_array
```

```
array([ 0., 15.,  0., 20.,  0., 34.,  0.,  0., 67., 30., 10.,  0.,
34.,
        0., 50., 25.,  0.])
```

Imputing `NaN` values with the mean of the data

```
# calculate mean of the non-NaN values
# we already have nan_remove array
```

```
mean = np.mean(nan_remove)

# # first make a new array to store the imputed data, name it
new_array
new_array = np.array(my_array)

# replace with the mean values
new_array[np.isnan(new_array)]=mean
new_array
```

```
array([31.66666667, 15.        , 31.66666667, 20.        ,
31.66666667,
       34.        , 31.66666667, 31.66666667, 67.        ,
30.        ,
       10.        , 31.66666667, 34.        , 31.66666667,
50.        ,
       25.        , 31.66666667])
```

Element Search

Create arbitrary array

```
my_array = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110,
120, 130, 140, 150])
```

Search if there is any element with value 70 of the created array. Returns `True` if there's any. Unless, it returns `False` (Boolean argument).

```
np.any(my_array == 70)
```

```
True
```

Check what index of the element with value 70 is in the array

```
np.where(my_array == 70)[0]
```

```
array([6])
```

Check what index of the element with values **less than** 70

```
np.where(my_array < 70)[0]
```

```
array([0, 1, 2, 3, 4, 5])
```

Data QC: Replace all values that are **less than** 70 with certain value

```
# first make a new array to store the replaced data, name it new_array
new_array = np.array(my_array)
```

```
# replace with value 10
new_array[new_array < 70] = 10
new_array
```

```
array([ 10,  10,  10,  10,  10,  10,  70,  80,  90, 100, 110, 120,
130,
       140, 150])
```

Read text file

We use `np.loadtxt` to read a text file.

An example here is given a `sincos.txt` file inside the GitHub repository. The file contains numerical result of sine and cosine function.

First, we specify the file path.

```
# copy the path in "Table of Contents" tab in Colab, and paste
filepath = '/content/python-bootcamp-for-geoengineers/data/sincos.txt'
```

Open with `np.loadtxt`

```
data = np.loadtxt(filepath)
data
```

```
array([[ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00],
       [ 3.60360360e-01,  6.28943332e-03,  9.99980221e-01],
       [ 7.20720721e-01,  1.25786178e-02,  9.99920886e-01],
       ...,
       [ 3.59279279e+02, -1.25786178e-02,  9.99920886e-01],
       [ 3.59639640e+02, -6.28943332e-03,  9.99980221e-01],
       [ 3.60000000e+02, -2.44929360e-16,  1.00000000e+00]])
```

Check its shape

```
data.shape
```

```
(1000, 3)
```

It has 1,000 rows and 3 columns.

Matplotlib

Plot and its accessories

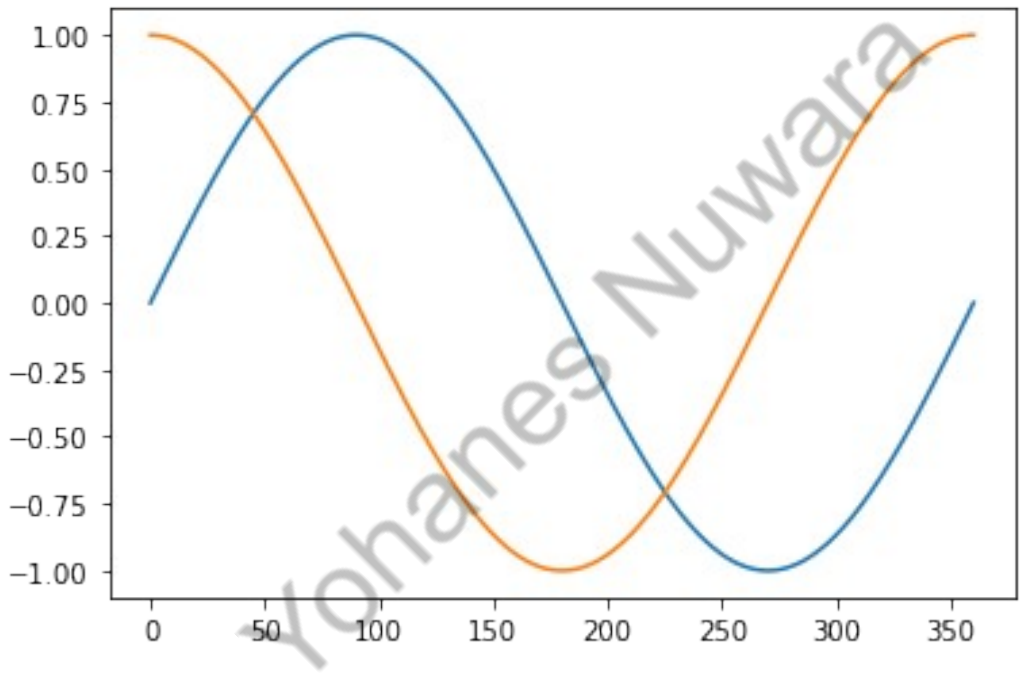
In this session, we will use the data that we have imported using `np.loadtxt` earlier.

It has 3 columns. First column is x values, second column is result of $\sin(x)$, and the third column is result of $\cos(x)$.

```
x = data[:,0]
sinx = data[:,1]
cosx = data[:,2]
```

Next we make a plot using Matplotlib Pyplot (or `plt`).

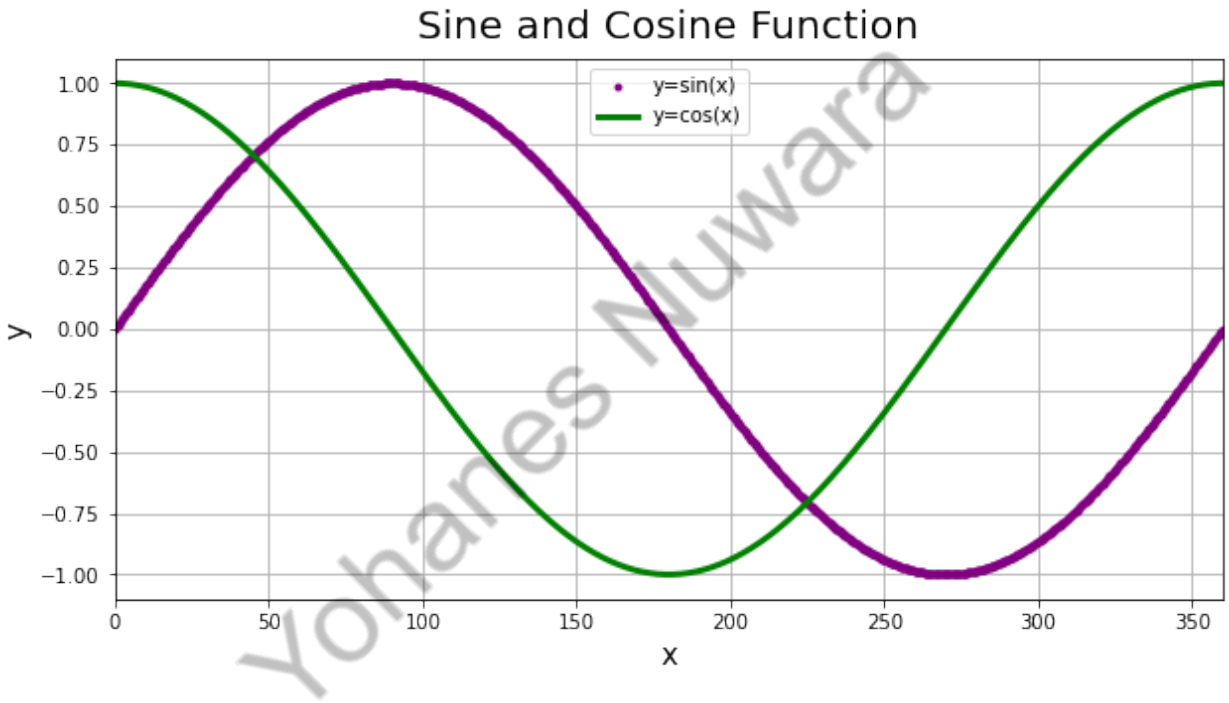
```
# plot
plt.plot(x, sinx)
plt.plot(x, cosx)
plt.show()
```



In every plot, we need to give plot attributes (title, label, legend) and may change the color of curve. We will modify this now.

```
# resize the plot
plt.figure(figsize=(10,5))

# plot, specify color, linetype, linewidth, and give labels
plt.plot(x, sinx, '.', color='purple', label='y=sin(x)')
plt.plot(x, cosx, '-', color='green', linewidth=3, label='y=cos(x)')

# give title, with size and pad
plt.title('Sine and Cosine Function', size=20, pad=10)

# give labels, with size
```

```
plt.xlabel('x', size=15)
plt.ylabel('y', size=15)

# limit the axes
plt.xlim(0, 360)

# show the legends and specify its location in the plot
plt.legend(loc='upper center')

# show the grids
plt.grid()

plt.show()
```



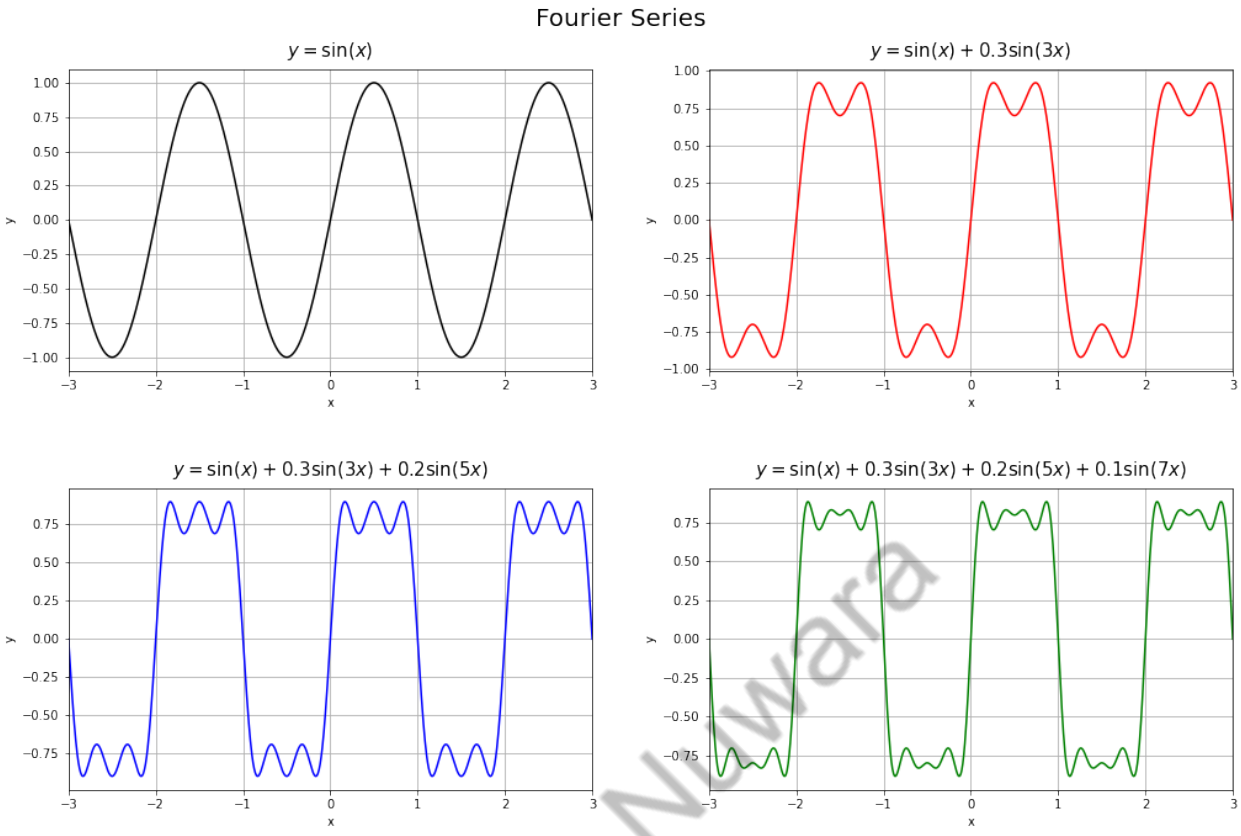
Subplot

What we'll do now is to create our own data using a sine function.

```
x = np.linspace(-3, 3, 1000)
y1 = np.sin(np.pi * x)
y2 = np.sin(np.pi * x) + (0.3 * (np.sin(3 * np.pi * x)))
y3 = np.sin(np.pi * x) + (0.3 * (np.sin(3 * np.pi * x))) + (0.2 *
(np.sin(5 * np.pi * x)))
y4 = np.sin(np.pi * x) + (0.3 * (np.sin(3 * np.pi * x))) + (0.2 *
(np.sin(5 * np.pi * x))) + (0.1 * (np.sin(7 * np.pi * x)))
```

Next, plot all of the results, using `subplots` so you will have all plots side by side.

```
plt.figure(figsize=(15,10))

plt.suptitle('Fourier Series', size=20)

plt.subplot(2,2,1)
plt.plot(x, y1, color='black')
plt.title(r'$y=\sin(x)$', size=15, pad=10)
plt.xlabel('x', size=10)
plt.ylabel('y', size=10)
plt.xlim(min(x), max(x))
plt.grid()

plt.subplot(2,2,2)
plt.plot(x, y2, color='red')
plt.title(r'$y=\sin(x)+0.3\sin(3x)$', size=15, pad=10)
plt.xlabel('x', size=10)
plt.ylabel('y', size=10)
plt.xlim(min(x), max(x))
plt.grid()

plt.subplot(2,2,3)
plt.plot(x, y3, color='blue')
plt.title(r'$y=\sin(x)+0.3\sin(3x)+0.2\sin(5x)$', size=15, pad=10)
plt.xlabel('x', size=10)
plt.ylabel('y', size=10)
plt.xlim(min(x), max(x))
plt.grid()

plt.subplot(2,2,4)
plt.plot(x, y4, color='green')
plt.title(r'$y=\sin(x)+0.3\sin(3x)+0.2\sin(5x)+0.1\sin(7x)$', size=15,
pad=10)
plt.xlabel('x', size=10)
plt.ylabel('y', size=10)
plt.xlim(min(x), max(x))
plt.grid()

# set distance between subplots
plt.tight_layout(4)

plt.show()
```



Pandas

Create a Dataframe

```
company = np.array(['ConocoPhillips', 'Royal Dutch Shell', 'Equinor
ASA', 'Sonatrach','Petronas'])
country = np.array(['USA', 'Netherlands', 'Norway', 'Algeria',
'Malaysia'])
date = np.array(['2002/08/02', '1907/04/23', '1972/06/14',
'1963/12/31', '1974/08/07'])

company_df = pd.DataFrame({'Company': company, 'Country': country,
'Date Founded': date})
company_df
```

```
             Company      Country Date Founded
0     ConocoPhillips          USA   2002/08/02
1  Royal Dutch Shell  Netherlands   1907/04/23
2        Equinor ASA       Norway   1972/06/14
3          Sonatrach      Algeria   1963/12/31
4           Petronas     Malaysia   1974/08/07
```

Basic elements of Dataframe

```
company_df['Company']
```

```
0       ConocoPhillips
1    Royal Dutch Shell
2          Equinor ASA
3            Sonatrach
4             Petronas
Name: Company, dtype: object
```

```
company_df.iloc[2]
```

```
Company          Equinor ASA
Country               Norway
Date Founded      1972/06/14
Name: 2, dtype: object
```

Convert a column data to Array

```
company_df['Company'].values
```

```
array(['ConocoPhillips', 'Royal Dutch Shell', 'Equinor ASA',
'Sonatrach',
       'Petronas'], dtype=object)
```

Adding new column to the dataframe

```
employee = np.array([11400, 82000, 20000, 120000, 51000])

company_df['Employee'] = employee
company_df
```

	Company	Country	Date Founded	Employee
0	ConocoPhillips	USA	2002/08/02	11400
1	Royal Dutch Shell	Netherlands	1907/04/23	82000
2	Equinor ASA	Norway	1972/06/14	20000
3	Sonatrach	Algeria	1963/12/31	120000
4	Petronas	Malaysia	1974/08/07	51000

Display summary statistics

```
company_df.describe()
```

```
            Employee
count        5.000000
mean    56880.000000
std     44939.648419
min     11400.000000
25%     20000.000000
50%     51000.000000
```

```
75%     82000.000000
max    120000.000000
```

Convert to Datetime Format

```
company_df['Date Founded']
```

```
0    2002/08/02
1    1907/04/23
2    1972/06/14
3    1963/12/31
4    1974/08/07
Name: Date Founded, dtype: object
```

Format check web: https://strftime.org/

```
company_df['Date Founded'] =  pd.to_datetime(company_df['Date
Founded'], format='%Y/%m/%d')

company_df
```

	Company	Country	Date Founded	Employee
0	ConocoPhillips	USA	2002-08-02	11400
1	Royal Dutch Shell	Netherlands	1907-04-23	82000
2	Equinor ASA	Norway	1972-06-14	20000
3	Sonatrach	Algeria	1963-12-31	120000
4	Petronas	Malaysia	1974-08-07	51000

```
company_df['Date Founded']
```

```
0   2002-08-02
1   1907-04-23
2   1972-06-14
3   1963-12-31
4   1974-08-07
Name: Date Founded, dtype: datetime64[ns]
```

Accessing Dataframe columns and rows

Display the "Country" column

```
company_df['Country']
```

```
0            USA
1    Netherlands
2         Norway
3        Algeria
4       Malaysia
Name: Country, dtype: object
```

Or alternatively we can search by its column index. Let's display the fourth column

```
company_df.iloc[:,3]
```

```
0     11400
1     82000
2     20000
3    120000
4     51000
Name: Employee, dtype: int64
```

Display the third row

```
company_df.iloc[2,:]
```

```
Company                  Equinor ASA
Country                       Norway
Date Founded    1972-06-14 00:00:00
Employee                       20000
Name: 2, dtype: object
```

Slicing dataframe

In any case, we may want to select only portion of the dataframe. For example, we want to get the **first two colums** only.

```
company_df.iloc[:, 0:2]
```

```
              Company      Country
0     ConocoPhillips          USA
1  Royal Dutch Shell  Netherlands
2         Equinor ASA       Norway
3           Sonatrach      Algeria
4            Petronas     Malaysia
```

Also, we may want to get the **first two columns**, omitting the rest.

```
company_df.iloc[0:2, :]
```

```
              Company      Country Date Founded  Employee
0     ConocoPhillips          USA   2002-08-02     11400
1  Royal Dutch Shell  Netherlands   1907-04-23     82000
```

Data Analysis of PetroWeek 2020 Registrants

Let us analyze the registrant data of this PetroWeek 2020 Python training. Data is in CSV format. First, we specify the file path.

```
filepath =
'/content/python-bootcamp-for-geoengineers/data/registrant_data_petrow
eek2020.csv'
```

Then open the data using Pandas `read_csv`

```
registrant =
pd.read_csv('/content/python-bootcamp-for-geoengineers/data/registrant
_data_petroweek2020.csv',
                           encoding = "ISO-8859-1")
```

Now you can take the overview of data by viewing its `head` and `tail`.

```
registrant.head(10)
```

```
                                  Name  ... Unnamed: 5
0  BOYKE DEO JHON INDRA UTAMA SIHITE  ...         NaN
1                           michelle  ...         NaN
2                 Alfan Khoirul Umam  ...         NaN
3               Theo Rifaldi Siregar  ...         NaN
4              Alfian Gilang Gumelar  ...         NaN
5                   Namira Fitriyani  ...         NaN
6                            Shirley  ...         NaN
7               Evan Fadhil Nurhakim  ...         NaN
8                   Reifandi Redhiza  ...         NaN
9          Ahmat Syukron Haqqulyakin  ...         NaN

[10 rows x 6 columns]
```

```
registrant.tail(10)
```

```
                    Name  ... Unnamed: 5
444        Vikas Kooneti  ...         NaN
445          Prili lauma  ...         NaN
446        Nehal Khetani  ...         NaN
447     Mohammed Gumati   ...         NaN
448  ABOLAJI SAHEED BAYO  ...         NaN
449     Moad Ben Ramadan  ...         NaN
450       Febry roma rio  ...         NaN
451                  NaN  ...         NaN
452                  NaN  ...         NaN
453                  NaN  ...         NaN

[10 rows x 6 columns]
```

Now as you can see, how MESSED UP our data is. Normally, we don't use the `encoding`. However, this raw data requires `encoding`. Also you can see 2 things:

- The last 3 `Unnamed` columns has all `NaN` values
- The last 3 rows has all `NaN` values (look its `tail`!)

Data cleansing 1: Delete Unwanted Columns

We want to remove the **last 3 rows** because they contain NaN values. We can use the `.iloc` technique that we have learnt just before.

```
registrant_edit = registrant.iloc[:,0:-3]
registrant_edit.tail(10)
```

```
                    Name  ...                    Major / Batch
444        Vikas Kooneti  ...          Petroleum Engineering
445          Prili lauma  ...                              Pe 18
446        Nehal Khetani  ...  Btech in Petroleum Engineering
447     Mohammed Gumati   ...                       Geophysics
448  ABOLAJI SAHEED BAYO  ...                         September
449     Moad Ben Ramadan  ...             Reservoir engineer
450       Febry roma rio  ...    Petroleum engineering / 2014
451                  NaN  ...                              NaN
452                  NaN  ...                              NaN
453                  NaN  ...                              NaN

[10 rows x 3 columns]
```

Data cleansing 2: Delete Unwanted Rows

Next, we want to remove the **last 3 columns** because they contain NaN values. Also we use `.iloc`

```
registrant_edit = registrant_edit.iloc[0:-3,:]
registrant_edit.tail(10)
```

```
                    Name  ...                    Major / Batch
441          Sobia Fatima  ...  Petroleum and Gas Engineering
442         Anwar Santoso  ...      Prtroleum Enginerring 2017
443  Rizky Wijaya Saputra  ...   Mechanical Engineering / 2019
444         Vikas Kooneti  ...          Petroleum Engineering
445           Prili lauma  ...                              Pe 18
446         Nehal Khetani  ...  Btech in Petroleum Engineering
447      Mohammed Gumati   ...                       Geophysics
448   ABOLAJI SAHEED BAYO  ...                         September
449      Moad Ben Ramadan  ...             Reservoir engineer
450        Febry roma rio  ...    Petroleum engineering / 2014

[10 rows x 3 columns]
```

How many registrants in this course ?

```
registrant_edit.count()
```

```
Name            449
University      427
```

```
Major / Batch    417
dtype: int64
```

Data analysis: Visualize pie diagram based on registrants' major

Now that our data has been edited, we'd like to visualize the majors of the registrants in this course !

First, do a slicing of the column that contains major, `Major / Batch`

```
major = registrant_edit['Major / Batch']
major
```

```
0               petroleum engineering
1                                pe/18
2             Teknik Perminyakan/2019
3              Geological Engineering
4                Teknik Geologi/ 2016
                     ...
446    Btech in Petroleum Engineering
447                        Geophysics
448                          September
449                Reservoir engineer
450      Petroleum engineering / 2014
Name: Major / Batch, Length: 451, dtype: object
```

Before continuing, we need to REMOVE all rows that contain `NaN` values, so we can sort well.

```
major = major.dropna()
major
```

```
0               petroleum engineering
1                                pe/18
2             Teknik Perminyakan/2019
3              Geological Engineering
4                Teknik Geologi/ 2016
                     ...
446    Btech in Petroleum Engineering
447                        Geophysics
448                          September
449                Reservoir engineer
450      Petroleum engineering / 2014
Name: Major / Batch, Length: 417, dtype: object
```

Now, do the sorting. Here we'd like to count: **How many participants are from major "X" ???**

We do this using `str.contains`, meaning we find in the "Major" colum, which row has string that contains sub-string. Confused?

For example, major Petroleum Engineering. We could find it by searching a substring such as `Petro`.

Next on, we pass `.count()` to count all rows that contains this sub-string.

```
# count all majors (in English)
petroleum_major1 = major[major.str.contains('Petro')].count()
geophysics_major1 = major[major.str.contains('Geoph')].count()
geology_major = major[major.str.contains('Geol')].count()
mech_major = major[major.str.contains('Mech')].count()
electrical_major = major[major.str.contains('Electr')].count()
chemical_major = major[major.str.contains('Chem')].count()
material_major = major[major.str.contains('Material')].count()
metallurgy_major = major[major.str.contains('Metal')].count()
astronomy_major = major[major.str.contains('Astro')].count()
economy_major = major[major.str.contains('Econo')].count()
marine_major = major[major.str.contains('Marine')].count()
```

Because some participants are from Indonesia, they inputted in the registration form in Indonesian language. So, we apply the sorting too.

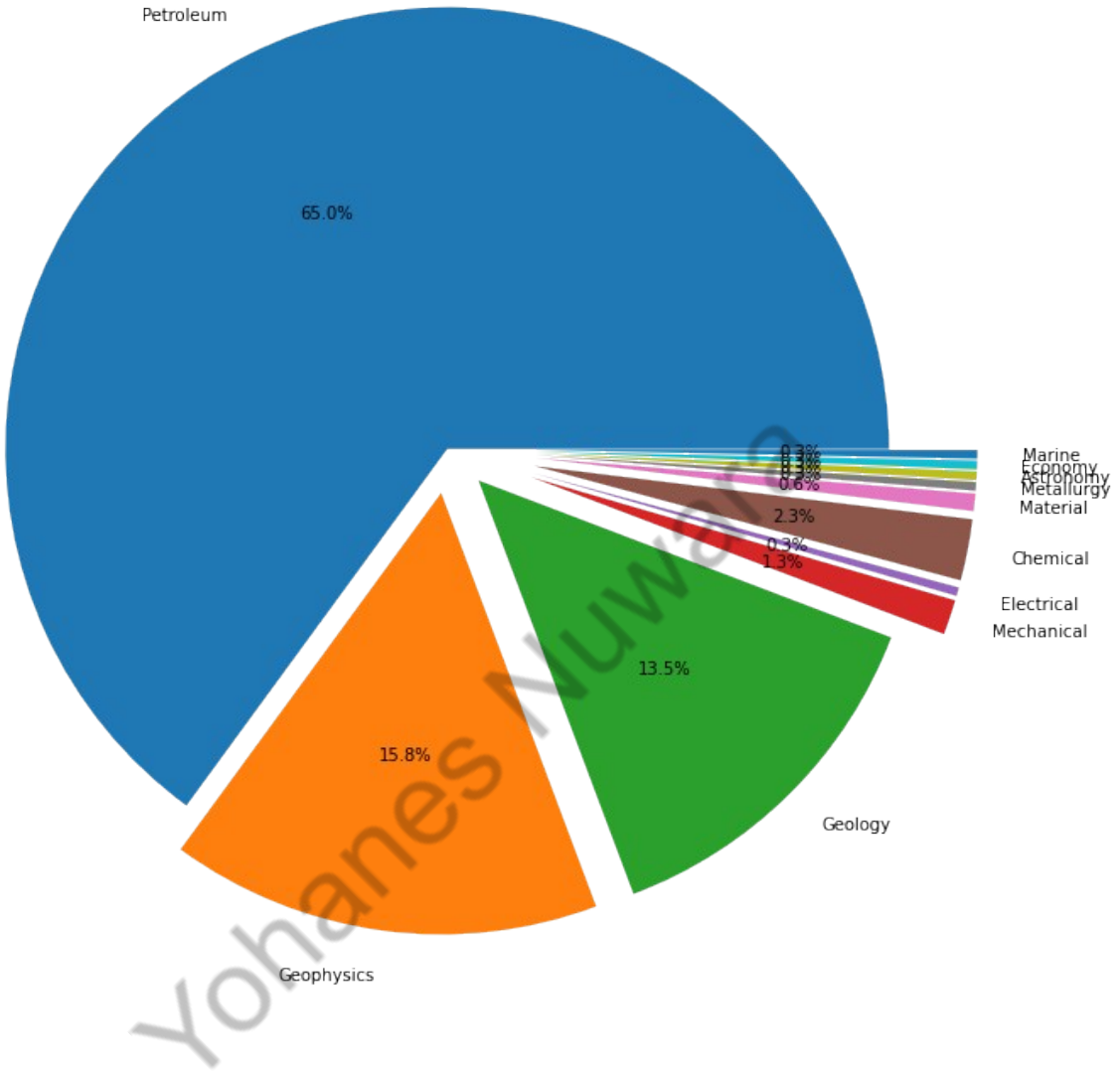
```
# count all majors (in Bahasa)
petroleum_major2 = major[major.str.contains('minyak')].count()
geophysics_major2 = major[major.str.contains('Geof')].count()
```

Next, we sum up the major which in English and Bahasa, into one individual sum.

```
# summing majors in English and in Bahasa
petroleum_major = np.sum([petroleum_major1, petroleum_major2])
geophysics_major = np.sum([geophysics_major1, geophysics_major2])
```

Finally, we create a pie diagram using Matplotlib that we learnt before. Use: `plt.pie`

```
major_name = ['Petroleum', 'Geophysics', 'Geology', 'Mechanical',
'Electrical',
              'Chemical', 'Material', 'Metallurgy', 'Astronomy',
'Economy', 'Marine']
major_count = [petroleum_major, geophysics_major, geology_major,
               mech_major, electrical_major, chemical_major,
material_major,
               metallurgy_major, astronomy_major, economy_major,
marine_major]
explode = [0, 0.1, 0.1, 0.2, 0.2 , 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]

plt.figure(figsize=(20,12))
plt.pie(major_count, labels=major_name, explode=explode,
autopct='%1.1f%%')
plt.show()
```



```
import numpy as np
import matplotlib.pyplot as plt # plotting
import pandas as pd
import scipy

# Making a list using numpy array method
x = np.array([1, 2, 3, 4, 5, 6])
print(x)
```

```
[1 2 3 4 5 6]
```

```
# Making a list using LIST
x = [1, 2, 3, 4, 5]
print(x)
```

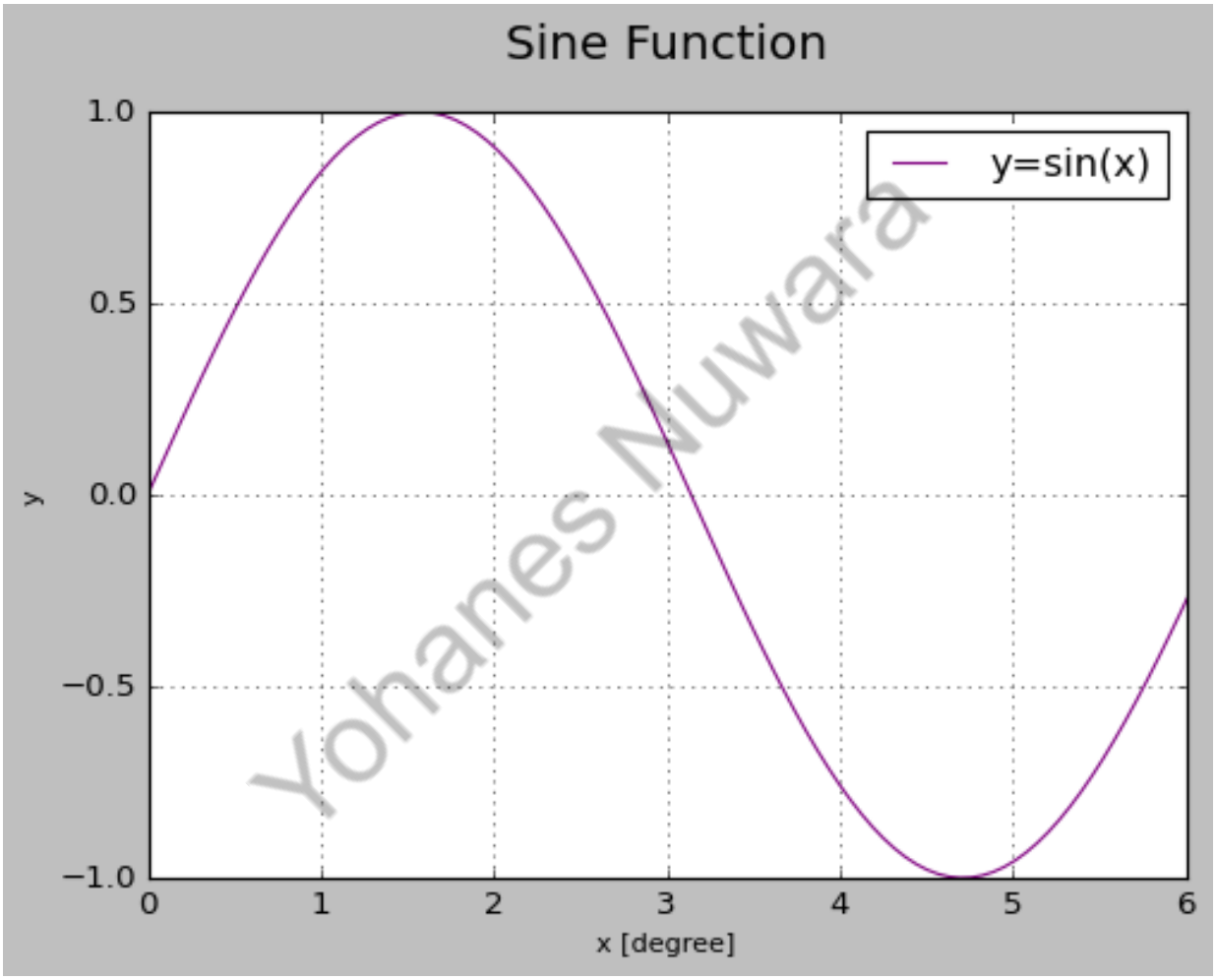
```
[1, 2, 3, 4, 5]
```

```
# produce an array from 1 to 100 divided into 50 numbers
y = np.linspace(1, 100, 50)

# produce an array from 1 to 100, the increment 2
y = np.arange(1, 100, 2)
print(y)
```

```
[ 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45
47
 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93
95
 97 99]
```

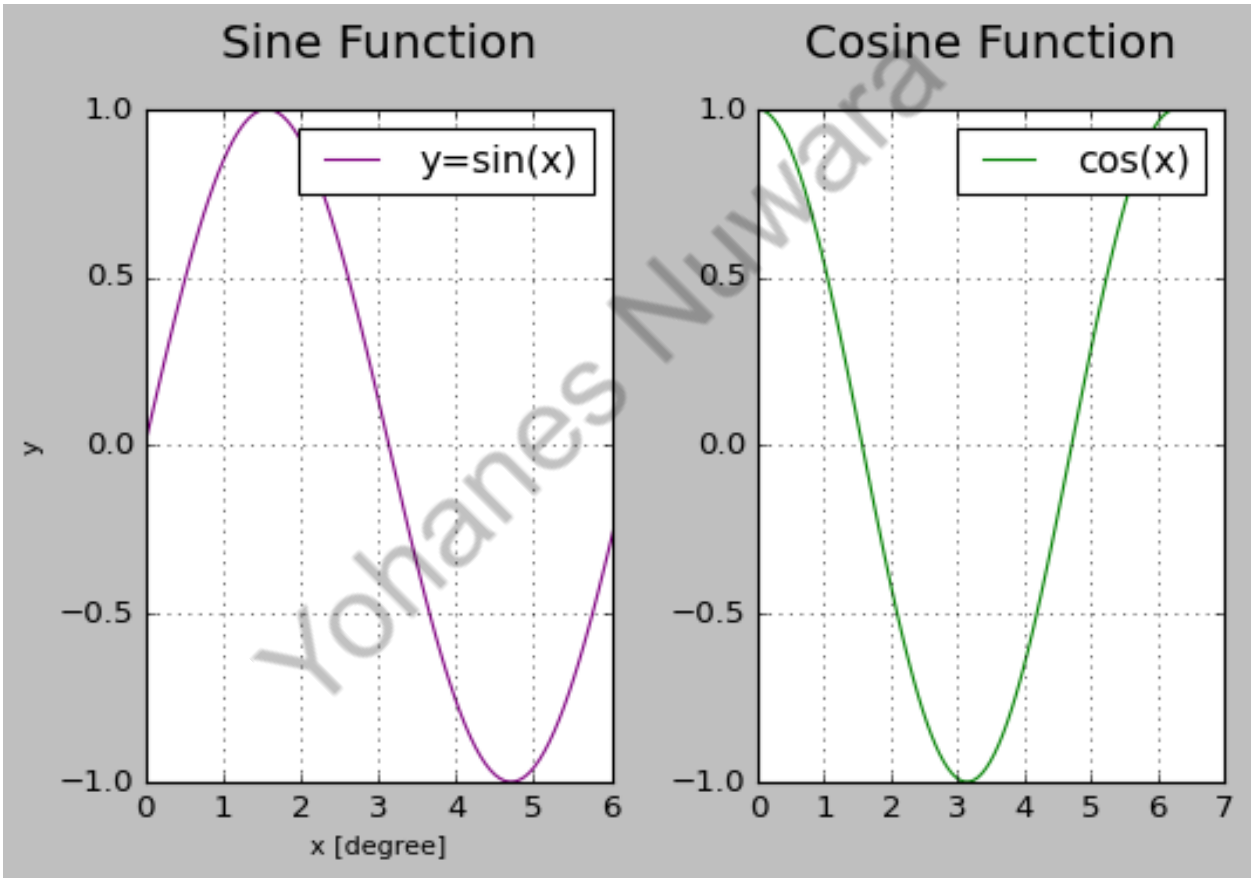
```
def sine(x):
  y = np.sin(x)
  return y

def cosine(x):
  y = np.cos(x)
  return y

c = np.linspace(0, 360, 100) # degree
c = np.deg2rad(c)

y = sine(c)
y1 = cosine(c)

plt.style.use("classic")

plt.figure(figsize=(7,5))

plt.plot(c, y, color="purple", label="y=sin(x)")
```

```
# plt.plot(c, y1, color="green", label="y=cos(x)")
plt.xlabel("x [degree]", size=10)
plt.ylabel("y", size=10)
plt.title("Sine Function", size=18, pad=20)
plt.xlim(0,6)

plt.legend()
plt.grid()
plt.show()
```




```
plt.style.use("classic")

plt.figure(figsize=(7,5))

plt.subplot(1,2,1)
plt.plot(c, y, color="purple", label="y=sin(x)")
# plt.plot(c, y1, color="green", label="y=cos(x)")
plt.xlabel("x [degree]", size=10)
plt.ylabel("y", size=10)
```

```
plt.title("Sine Function", size=18, pad=20)
plt.xlim(0,6)
plt.grid()
plt.legend()

plt.subplot(1,2,2)
plt.plot(c, y1, color="green", label="cos(x)")
plt.title("Cosine Function", size=18, pad=20)
plt.legend()

plt.tight_layout(1.6)
plt.grid()
plt.show()
```




```
# generate an array consisting random numbers from 0 to 1, 50 numbers
noise = np.random.random(100)

print(noise)
```

```
[0.00388175 0.6337363  0.74297108 0.57642325 0.53652743 0.53005074
 0.77971157 0.36034314 0.13098436 0.61136187 0.8021041  0.03542865
 0.50114152 0.55765362 0.2278868  0.21930513 0.78217925 0.7251346
 0.24591637 0.92464065 0.4817952  0.96587002 0.15934233 0.90749606
```

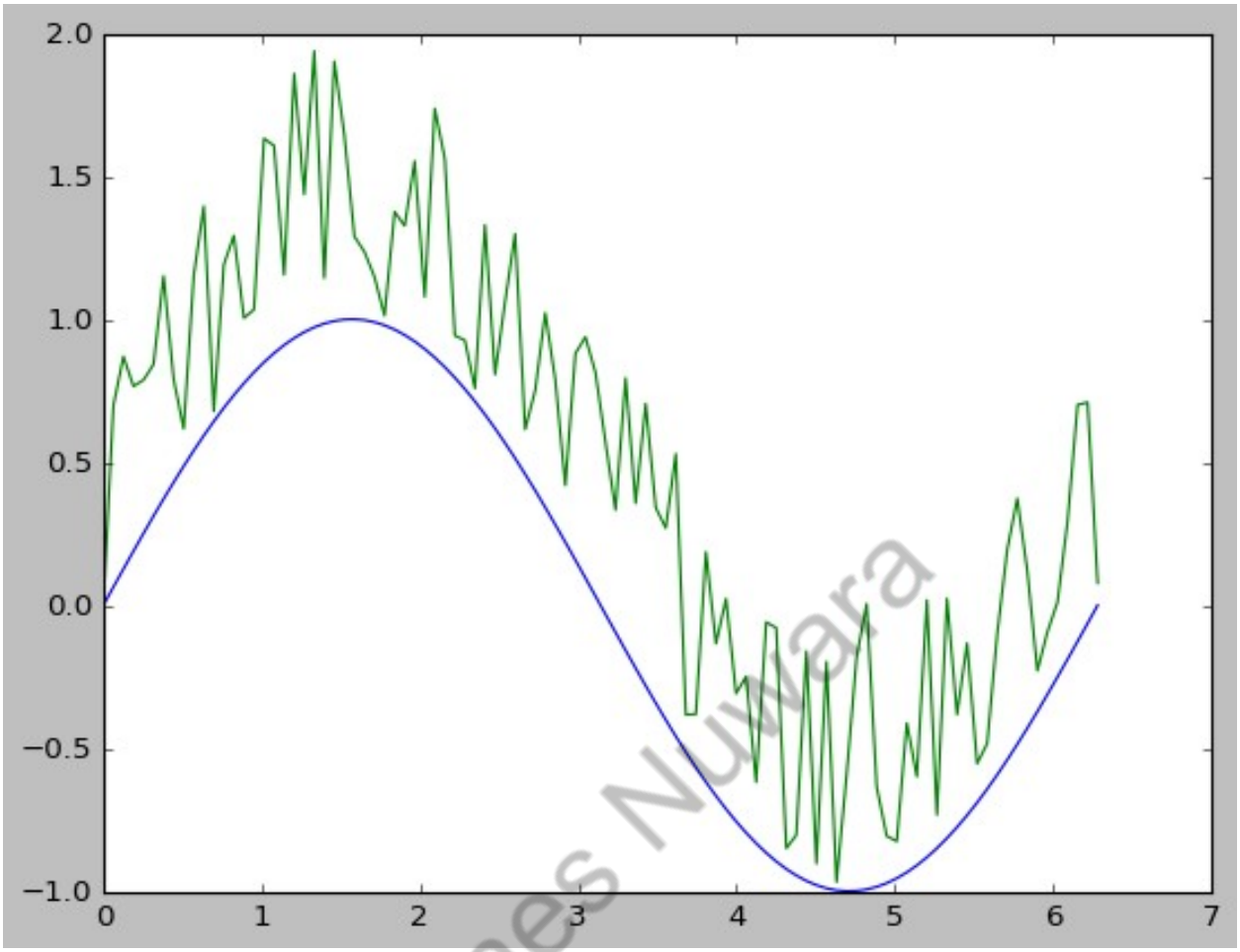
```
 0.64455348 0.28903922 0.239019   0.15768869 0.03363798 0.41177474
 0.38156535 0.63055013 0.18298955 0.8703928  0.72908173 0.14670695
 0.1703229  0.04563562 0.66283558 0.18864358 0.49438137 0.78482035
 0.15662146 0.3503087  0.67981549 0.50677176 0.19883566 0.72216175
 0.84290212 0.78226133 0.60157195 0.42816842 0.95213682 0.57691252
 0.98655042 0.68312571 0.67095985 0.98739004 0.13091967 0.18633272
 0.80412161 0.53277157 0.734845   0.44723313 0.5464199  0.21403991
 0.80687368 0.81668074 0.07240444 0.14151155 0.80285143 0.07550133
 0.79205886 0.02903956 0.40402434 0.81723216 0.99840585 0.35000831
 0.1643609  0.12948069 0.52205683 0.31042893 0.89922626 0.11699367
 0.83773847 0.39435236 0.60142315 0.1363308  0.15908434 0.49180779
 0.73585847 0.85949065 0.54944344 0.14253548 0.21895331 0.26144345
 0.48099999 0.82727996 0.77238599 0.07718859]
```

```
print(y)
```

```
[ 0.00000000e+00  6.34239197e-02  1.26592454e-01  1.89251244e-01
  2.51147987e-01  3.12033446e-01  3.71662456e-01  4.29794912e-01
  4.86196736e-01  5.40640817e-01  5.92907929e-01  6.42787610e-01
  6.90079011e-01  7.34591709e-01  7.76146464e-01  8.14575952e-01
  8.49725430e-01  8.81453363e-01  9.09631995e-01  9.34147860e-01
  9.54902241e-01  9.71811568e-01  9.84807753e-01  9.93838464e-01
  9.98867339e-01  9.99874128e-01  9.96854776e-01  9.89821442e-01
  9.78802446e-01  9.63842159e-01  9.45000819e-01  9.22354294e-01
  8.95993774e-01  8.66025404e-01  8.32569855e-01  7.95761841e-01
  7.55749574e-01  7.12694171e-01  6.66769001e-01  6.18158986e-01
  5.67059864e-01  5.13677392e-01  4.58226522e-01  4.00930535e-01
  3.42020143e-01  2.81732557e-01  2.20310533e-01  1.58001396e-01
  9.50560433e-02  3.17279335e-02 -3.17279335e-02 -9.50560433e-02
 -1.58001396e-01 -2.20310533e-01 -2.81732557e-01 -3.42020143e-01
 -4.00930535e-01 -4.58226522e-01 -5.13677392e-01 -5.67059864e-01
 -6.18158986e-01 -6.66769001e-01 -7.12694171e-01 -7.55749574e-01
 -7.95761841e-01 -8.32569855e-01 -8.66025404e-01 -8.95993774e-01
 -9.22354294e-01 -9.45000819e-01 -9.63842159e-01 -9.78802446e-01
 -9.89821442e-01 -9.96854776e-01 -9.99874128e-01 -9.98867339e-01
 -9.93838464e-01 -9.84807753e-01 -9.71811568e-01 -9.54902241e-01
 -9.34147860e-01 -9.09631995e-01 -8.81453363e-01 -8.49725430e-01
 -8.14575952e-01 -7.76146464e-01 -7.34591709e-01 -6.90079011e-01
 -6.42787610e-01 -5.92907929e-01 -5.40640817e-01 -4.86196736e-01
 -4.29794912e-01 -3.71662456e-01 -3.12033446e-01 -2.51147987e-01
 -1.89251244e-01 -1.26592454e-01 -6.34239197e-02 -2.44929360e-16]
```

```
plt.plot(c, y)
plt.plot(c, y+noise)
```

```
[<matplotlib.lines.Line2D at 0x7fa6dcec5978>]
```



Pandas

```
x = ["John", "Peter", "Ashley", "Bob", "Ronald"]
y = [13, 14, 20, 25, 78]
z = ["Germany", "France", "Egypt", "USA", "Norway"]

data = pd.DataFrame({"Name": x, "Ages": y, "Nationality": z})

data
```

```
     Name  Ages Nationality
0    John    13     Germany
1   Peter    14      France
2  Ashley    20       Egypt
3     Bob    25         USA
4  Ronald    78      Norway
```

```
data["Occupation"] = ["Carpenter", "Mudlogger", "Petrophysicist",
"Businessman", "Dentist"]
data["DOB"] = [1990, 1991, 1967, 1978, 2002]
```

```
data
```

```
     Name  Ages Nationality      Occupation   DOB
0    John    13     Germany       Carpenter  1990
1   Peter    14      France       Mudlogger  1991
2  Ashley    20       Egypt  Petrophysicist  1967
3     Bob    25         USA     Businessman  1978
4  Ronald    78      Norway         Dentist  2002
```

```
data["Company"] = ["Chevron", "McDonalds", "KFC", "Shell", "Baker
Hughes"]

data
```

```
     Name  Ages Nationality      Occupation   DOB       Company
0    John    13     Germany       Carpenter  1990       Chevron
1   Peter    14      France       Mudlogger  1991     McDonalds
2  Ashley    20       Egypt  Petrophysicist  1967           KFC
3     Bob    25         USA     Businessman  1978         Shell
4  Ronald    78      Norway         Dentist  2002  Baker Hughes
```

```
# Print who is working for Chevron
mask = data["Company"]=="Chevron"

data[mask]
```

```
   Name  Ages Nationality Occupation   DOB  Company
0  John    13     Germany  Carpenter  1990  Chevron
```

```
# Print the names who was born after 1990
mask = data["DOB"] < 1980

data[mask]
```

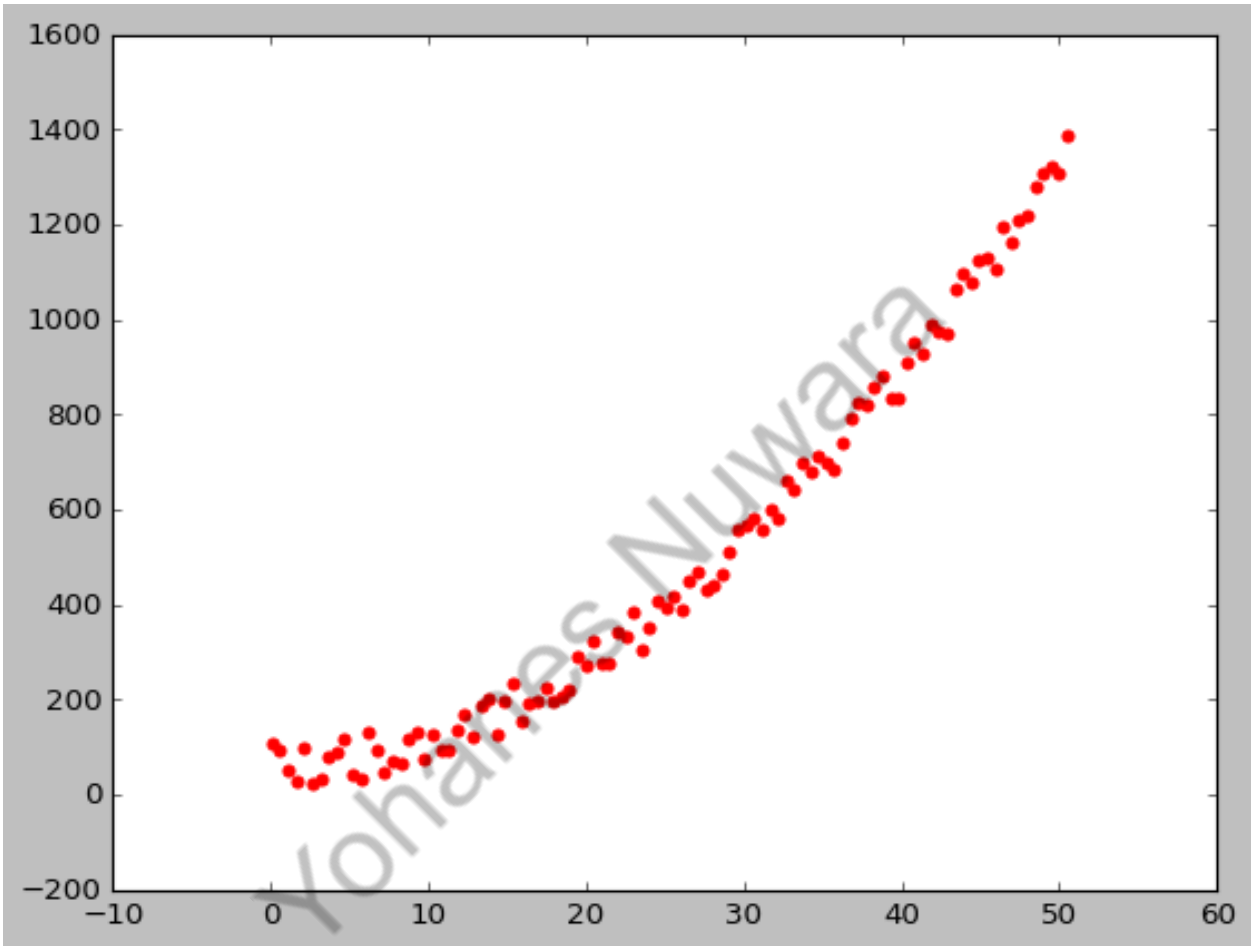
```
     Name  Ages Nationality      Occupation   DOB Company
2  Ashley    20       Egypt  Petrophysicist  1967     KFC
3     Bob    25         USA     Businessman  1978   Shell
```


SciPy

Curve-fitting

```
x = np.linspace(0.1, 50.5, 100)

# def polynomial(x):
#   return 0.5 * (x**2) + 0.7 * x + 10

y = polynomial(x)
```

```
noise = np.random.random(100) * 100

ydata = y + noise

plt.scatter(x, ydata, color="red")
plt.show()
```

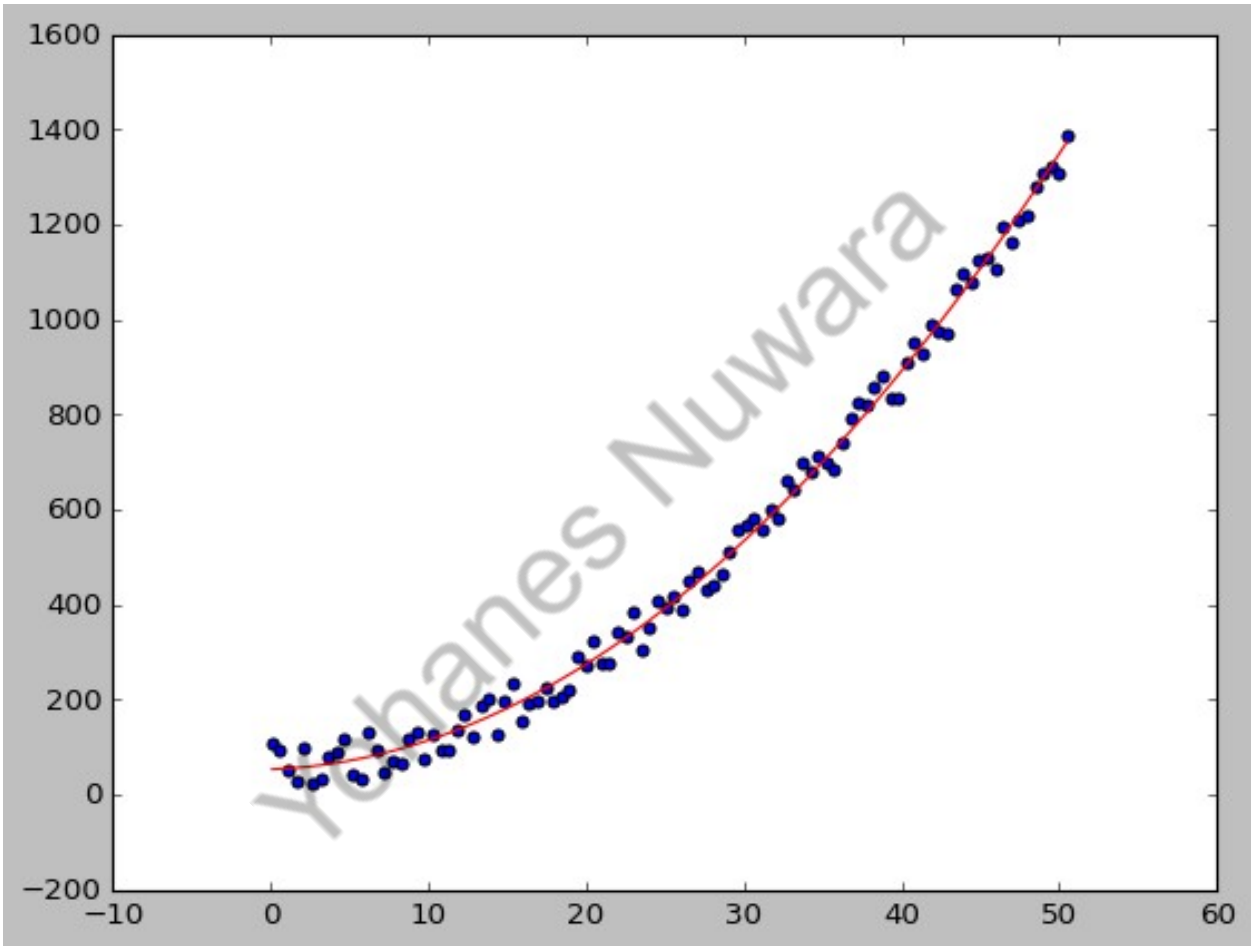


$$y = A x^2 + B x + C$$

```
from scipy.optimize import curve_fit

def polynomial2(x, a, b, c):
  y = a * x**2 + b * x + c
  return y

# x and y data are our data, that hasn't been fitted
curve_fit(polynomial2, x, ydata)
```

```
(array([ 0.49507611,  1.16682362, 52.62428369]),
 array([[ 2.37941817e-04, -1.20398561e-02,  1.00919131e-01],
```

```
        [-1.20398561e-02,  6.50312427e-01, -6.14622929e+00],
        [ 1.00919131e-01, -6.14622929e+00,  7.79830222e+01]]))

yfit = polynomial2(x, a=0.49507611, b=1.16682362, c=52.62428369)

plt.scatter(x, ydata)
plt.plot(x, yfit, color="red")

plt.show()
```



np.loadtxt > loading TXT file pd.read_csv > loading CSV file

LAS file > lasio library SEGY file > segyio library

```
def r_squared(ydata, yfit):
  R2 = np.sqrt(np.sum((yfit-ydata)**2))
  return R2

r_squared(ydata, yfit)
```

```
293.4046892360701
```

```
print(x, ydata) # data
print(x, yfit)
```

```
[ 0.1         0.60909091  1.11818182  1.62727273  2.13636364
2.64545455
  3.15454545  3.66363636  4.17272727  4.68181818  5.19090909  5.7
  6.20909091  6.71818182  7.22727273  7.73636364  8.24545455
8.75454545
  9.26363636  9.77272727 10.28181818 10.79090909 11.3
11.80909091
 12.31818182 12.82727273 13.33636364 13.84545455 14.35454545
14.86363636
 15.37272727 15.88181818 16.39090909 16.9        17.40909091
17.91818182
 18.42727273 18.93636364 19.44545455 19.95454545 20.46363636
20.97272727
 21.48181818 21.99090909 22.5        23.00909091 23.51818182
24.02727273
 24.53636364 25.04545455 25.55454545 26.06363636 26.57272727
27.08181818
 27.59090909 28.1        28.60909091 29.11818182 29.62727273
30.13636364
 30.64545455 31.15454545 31.66363636 32.17272727 32.68181818
33.19090909
 33.7        34.20909091 34.71818182 35.22727273 35.73636364
36.24545455
 36.75454545 37.26363636 37.77272727 38.28181818 38.79090909 39.3
 39.80909091 40.31818182 40.82727273 41.33636364 41.84545455
42.35454545
 42.86363636 43.37272727 43.88181818 44.39090909 44.9
45.40909091
 45.91818182 46.42727273 46.93636364 47.44545455 47.95454545
48.46363636
 48.97272727 49.48181818 49.99090909 50.5       ] [ 105.05820522
95.08267025   50.01757811   27.55951876   99.86766773
   22.87866594   33.64165658   77.2922472    88.29958988  115.59550272
   39.35896709   32.64353332  129.27704762   95.09019731   44.69830428
   67.89084685   66.70325541  114.91325271  130.78993156   76.12517058
  127.85527318   92.41284423   91.7184986   133.70938881  168.41190888
  119.18646669  188.18299461  202.71756204  125.62406983  196.56907044
  235.00295537  154.28890738  190.63681067  198.46482577  225.88369615
  196.52029456  206.0583588   217.57069466  289.93575488  273.42988598
  323.91739293  276.11882201  276.10132784  339.34720375  334.54542875
  383.00881263  304.65063181  349.15333025  407.22441701  393.39355979
  415.02086037  389.55776747  448.4512752   470.21347781  429.18021441
  439.22192545  463.68713782  511.1113981   555.20228969  565.00789133
  579.89710596  558.55512172  600.88142978  578.6605533   658.42985167
  641.66977815  697.25143617  679.37855678  712.31991478  696.52069019
  681.67279966  740.31412921  793.48034695  824.88639263  817.45231641
  857.9718803   878.47294824  834.50654888  831.56377959  909.42285443
```

```
  952.29831114  926.3979042   988.86307691  973.93636294  971.71310516
 1065.68438983 1095.6967456  1079.6278742  1126.20294802 1127.53699514
 1105.65917882 1193.98342075 1161.8944034  1208.29149432 1216.99532167
 1280.19587805 1308.86492088 1323.57523042 1306.75824727
1386.73903851]
[ 0.1         0.60909091  1.11818182  1.62727273  2.13636364
2.64545455
  3.15454545  3.66363636  4.17272727  4.68181818  5.19090909  5.7
  6.20909091  6.71818182  7.22727273  7.73636364  8.24545455
8.75454545
  9.26363636  9.77272727 10.28181818 10.79090909 11.3
11.80909091
 12.31818182 12.82727273 13.33636364 13.84545455 14.35454545
14.86363636
 15.37272727 15.88181818 16.39090909 16.9        17.40909091
17.91818182
 18.42727273 18.93636364 19.44545455 19.95454545 20.46363636
20.97272727
 21.48181818 21.99090909 22.5        23.00909091 23.51818182
24.02727273
 24.53636364 25.04545455 25.55454545 26.06363636 26.57272727
27.08181818
 27.59090909 28.1        28.60909091 29.11818182 29.62727273
30.13636364
 30.64545455 31.15454545 31.66363636 32.17272727 32.68181818
33.19090909
 33.7        34.20909091 34.71818182 35.22727273 35.73636364
36.24545455
 36.75454545 37.26363636 37.77272727 38.28181818 38.79090909 39.3
 39.80909091 40.31818182 40.82727273 41.33636364 41.84545455
42.35454545
 42.86363636 43.37272727 43.88181818 44.39090909 44.9
45.40909091
 45.91818182 46.42727273 46.93636364 47.44545455 47.95454545
48.46363636
 48.97272727 49.48181818 49.99090909 50.5       ] [  52.74591681
53.51865449   54.54801345   55.83399367   57.37659516
   59.17581792   61.23166195   63.54412725   66.11321381   68.93892165
   72.02125076   75.36020114   78.95577278   82.8079657    86.91677989
   91.28221534   95.90427207  100.78295006  105.91824933  111.31016986
  116.95871167  122.86387474  129.02565908  135.44406469  142.11909158
  149.05073973  156.23900915  163.68389984  171.3854118   179.34354503
  187.55829953  196.0296753   204.75767234  213.74229065  222.98353022
  232.48139107  242.23587319  252.24697657  262.51470123  273.03904716
  283.82001435  294.85760282  306.15181255  317.70264355  329.51009583
  341.57416937  353.89486418  366.47218027  379.30611762  392.39667624
  405.74385613  419.34765729  433.20807972  447.32512342  461.69878839
  476.32907463  491.21598214  506.35951092  521.75966096  537.41643228
  553.32982487  569.49983872  585.92647385  602.60973025  619.54960791
```

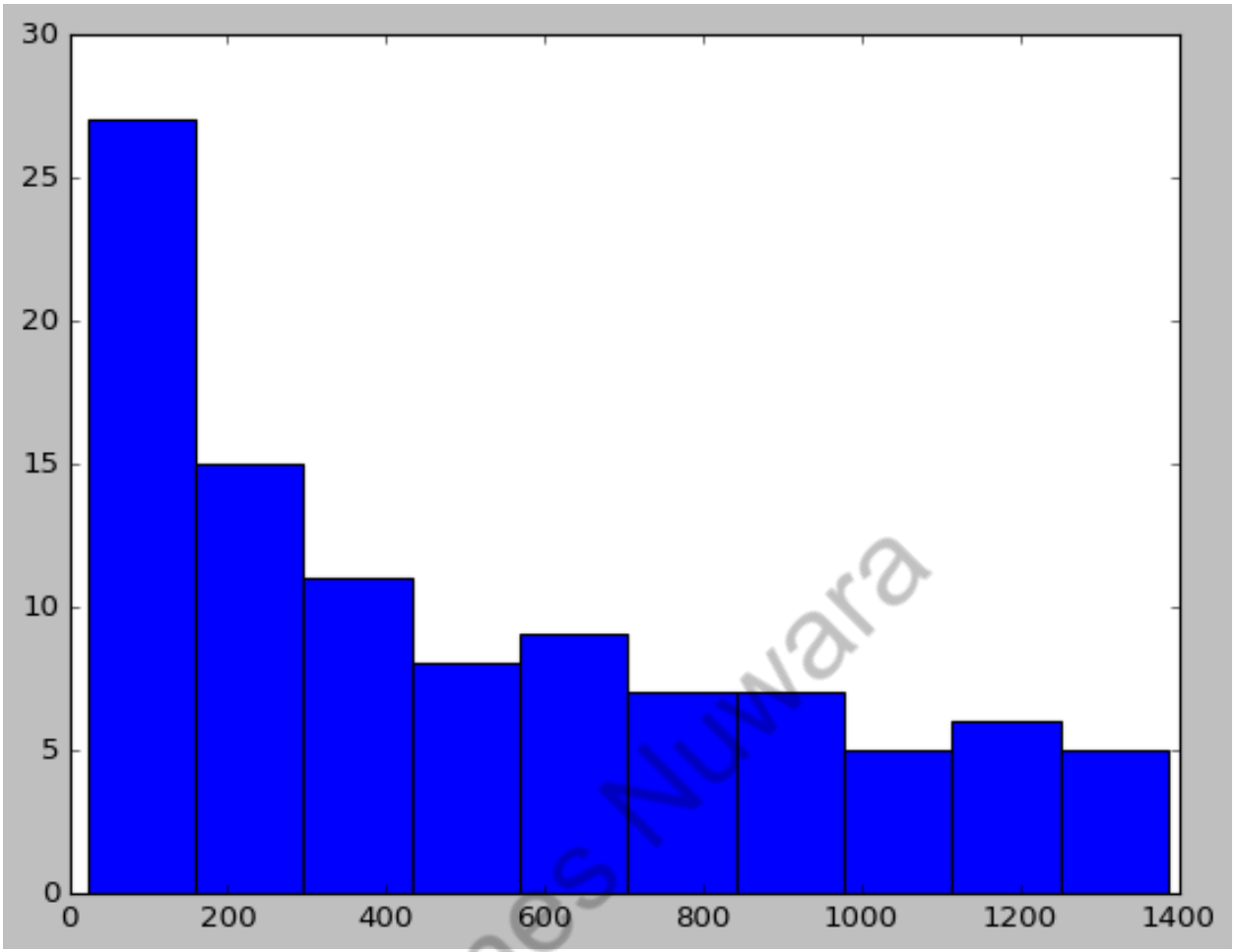
```
  636.74610685  654.19922705  671.90896852  689.87533127  708.09831528
  726.57792056  745.31414712  764.30699494  783.55646403  803.06255439
  822.82526602  842.84459892  863.12055309  883.65312853  904.44232524
  925.48814322  946.79058246  968.34964298  990.16532477 1012.23762783
 1034.56655215 1057.15209775 1079.99426461 1103.09305275 1126.44846215
 1150.06049283 1173.92914477 1198.05441799 1222.43631247 1247.07482822
 1271.96996524 1297.12172353 1322.5301031  1348.19510393
1374.11672603]
```

Root-finding, interpolation-extrapolation

Plotly

```
import plotly.express as px
import plotly.graph_objects as go

fig = px.scatter(x=x, y=ydata, color=ydata)
# fig = px.line(x=x, y=yfit)

fig.show()

plt.hist(ydata)
```

```
(array([27., 15., 11.,  8.,  9.,  7.,  7.,  5.,  6.,  5.]),
 array([  22.87866594,  159.26470319,  295.65074045,  432.03677771,
         568.42281496,  704.80885222,  841.19488948,  977.58092674,
        1113.96696399, 1250.35300125, 1386.73903851]),
 <a list of 10 Patch objects>)
```



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns

# Make a dataframe
name = ['Nuwara', 'Nha', 'Tien', 'Andres']
company = ['OYO', 'PTTEP', 'PTTEP', 'Petrobras']
city = ['Jakarta', 'Ho Chi Minh', 'Ho Chi Minh', 'Brazil']
experience = [2, 10, 10, 5]

df = pd.DataFrame({'Name': name, 'company': company, 'City': city,
                   'Exp': experience})
df
```

	Name	company	City	Exp
0	Nuwara	OYO	Jakarta	2
1	Nha	PTTEP	Ho Chi Minh	10
2	Tien	PTTEP	Ho Chi Minh	10
3	Andres	Petrobras	Brazil	5

```
# To print the header
df.tail(2) # or head(2)
```

	Name	company	City	Exp
2	Tien	PTTEP	Ho Chi Minh	10
3	Andres	Petrobras	Brazil	5

```
# To find out name of columns
df.columns
```

```
Index(['Name', 'company', 'City', 'Exp'], dtype='object')
```

```
# To find out the how numbers of columns
len(df.columns)
```

```
4
```

```
# Size of dataframe
df.shape
```

```
(4, 4)
```

```
# How to access the first column or first row
# First column
df.iloc[:,0]

# Third row
df.iloc[2,:]
```

```
# Second to third row
# df.iloc[1:3,:]
```

```
Name              Tien
company          PTTEP
City       Ho Chi Minh
Exp                 10
Name: 2, dtype: object
```

```
# How to print any column
df['company']
```

```
0          OYO
1        PTTEP
2        PTTEP
3    Petrobras
Name: company, dtype: object
```

```
# How to find out which person is in PTTEP
# To do masking
df['City']=='Jakarta'
```

```
# Assign to a new variable
cond = df['City']=='Jakarta'
```

```
# Assign the conditional var to our DF
df[cond]
```

```
     Name company     City  Exp
0  Nuwara     OYO  Jakarta    2
```

```
# How to find out which person is in PTTEP
# To do masking
df['company']=='PTTEP'
```

```
# Assign to a new variable
cond = df['company']=='PTTEP'
```

```
# Assign the conditional var to our DF
df_update = df[cond]
```

```
df_update.reset_index(drop=True)
```

```
   Name company         City  Exp
0   Nha   PTTEP  Ho Chi Minh   10
1  Tien   PTTEP  Ho Chi Minh   10
```

```
a = 4
```

```
if a==4:
  print('Not 4')
```

```
else:
  print('Is 4')
```

```
Not 4
```

```
# Add new data
df['Year_start'] = np.array([1960, 1970, 1980, 2000])
df['Number_of_pets'] = np.array([9, 3, 4, 1])

df
```

	Name	company	City	Exp	Year_start	Number_of_pets
0	Nuwara	OYO	Jakarta	2	1960	9
1	Nha	PTTEP	Ho Chi Minh	10	1970	3
2	Tien	PTTEP	Ho Chi Minh	10	1980	4
3	Andres	Petrobras	Brazil	5	2000	1

```
# Summary statistics: mean, variance, std, percentiles
df.describe()
```

	Exp	Year_start	Number_of_pets
count	4.000000	4.000000	4.00000
mean	6.750000	1977.500000	4.25000
std	3.947573	17.078251	3.40343
min	2.000000	1960.000000	1.00000
25%	4.250000	1967.500000	2.50000
50%	7.500000	1975.000000	3.50000
75%	10.000000	1985.000000	5.25000
max	10.000000	2000.000000	9.00000

```
# Find out unique values
df['company'].unique()
```

```
array(['OYO', 'PTTEP', 'Petrobras'], dtype=object)
```

```
df.columns
```

```
Index(['Name', 'company', 'City', 'Exp', 'Year_start',
'Number_of_pets'], dtype='object')
```

```
# a new dataframe
new_names = ['John', 'Vlad']
new_companies = ['BP', 'Total']
new_city = ['UK', 'Russia']
new_exp = [10, 2]

df2 = pd.DataFrame({'Name': new_names, 'company': new_companies,
'City': new_city, 'Exp': new_exp})
df2
```

```
   Name company    City  Exp
0  John      BP      UK   10
1  Vlad   Total  Russia    2
```

```
# Merge new data to old one
df_combine = pd.concat((df, df2)).reset_index(drop=True)

df_combine
```

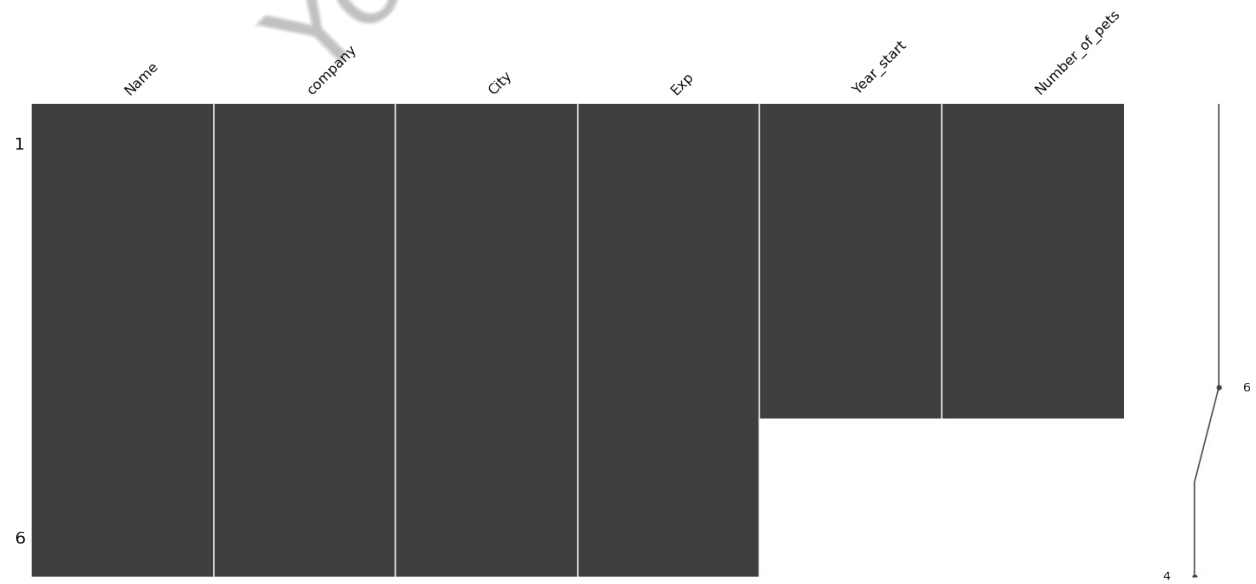
```
     Name    company         City  Exp  Year_start  Number_of_pets
0  Nuwara        OYO      Jakarta    2      1960.0             9.0
1     Nha      PTTEP  Ho Chi Minh   10      1970.0             3.0
2    Tien      PTTEP  Ho Chi Minh   10      1980.0             4.0
3  Andres  Petrobras       Brazil    5      2000.0             1.0
4    John         BP           UK   10         NaN             NaN
5    Vlad      Total       Russia    2         NaN             NaN
```

```
# Select multiple columns
df_combine[['Name', 'company', 'City', 'Exp']]
```

```
     Name    company         City  Exp
0  Nuwara        OYO      Jakarta    2
1     Nha      PTTEP  Ho Chi Minh   10
2    Tien      PTTEP  Ho Chi Minh   10
3  Andres  Petrobras       Brazil    5
4    John         BP           UK   10
5    Vlad      Total       Russia    2
```

```
import missingno as msno

msno.matrix(df_combine)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f86fa2e59d0>
```



```
# drop the columns with missing values
df_combine.dropna(subset=['Exp', 'City'])
```

```
     Name    company         City  Exp  Year_start  Number_of_pets
0  Nuwara        OYO      Jakarta    2      1960.0             9.0
1     Nha      PTTEP  Ho Chi Minh   10      1970.0             3.0
2    Tien      PTTEP  Ho Chi Minh   10      1980.0             4.0
3  Andres  Petrobras       Brazil    5      2000.0             1.0
4    John         BP           UK   10         NaN             NaN
5    Vlad      Total       Russia    2         NaN             NaN
```

```
df_combine
```

```
     Name    company         City  Exp  Year_start  Number_of_pets
0  Nuwara        OYO      Jakarta    2      1960.0             9.0
1     Nha      PTTEP  Ho Chi Minh   10      1970.0             3.0
2    Tien      PTTEP  Ho Chi Minh   10      1980.0             4.0
3  Andres  Petrobras       Brazil    5      2000.0             1.0
4    John         BP           UK   10         NaN             NaN
5    Vlad      Total       Russia    2         NaN             NaN
```

```
df_combine[df_combine['City']=='Ho Chi Minh']
```

```
   Name company         City  Exp  Year_start  Number_of_pets
1   Nha   PTTEP  Ho Chi Minh   10      1970.0             3.0
2  Tien   PTTEP  Ho Chi Minh   10      1980.0             4.0
```

```
# Change the value in Year_start where the value in City equals Ho Chi Minh
# The new value will be 20, instead of 10

df_combine.loc[df_combine['City']=='Ho Chi Minh', 'Exp'] = 20

df_combine
```

```
     Name    company         City  Exp  Year_start  Number_of_pets
0  Nuwara        OYO      Jakarta    2      1960.0             9.0
1     Nha      PTTEP  Ho Chi Minh   20      1970.0             3.0
2    Tien      PTTEP  Ho Chi Minh   20      1980.0             4.0
3  Andres  Petrobras       Brazil    5      2000.0             1.0
4    John         BP           UK   10         NaN             NaN
5    Vlad      Total       Russia    2         NaN             NaN
```

Data Analytics

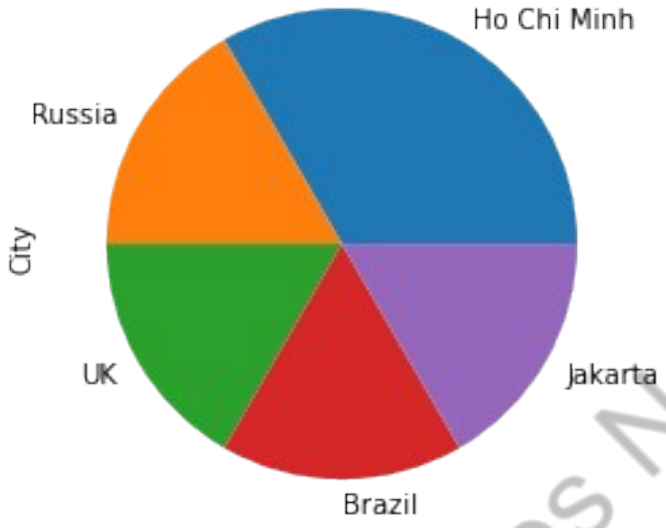
```
# To find the distribution of city
df_combine['City'].value_counts()
```

```
Ho Chi Minh    2
Russia         1
UK             1
```

```
Brazil         1
Jakarta        1
Name: City, dtype: int64
```

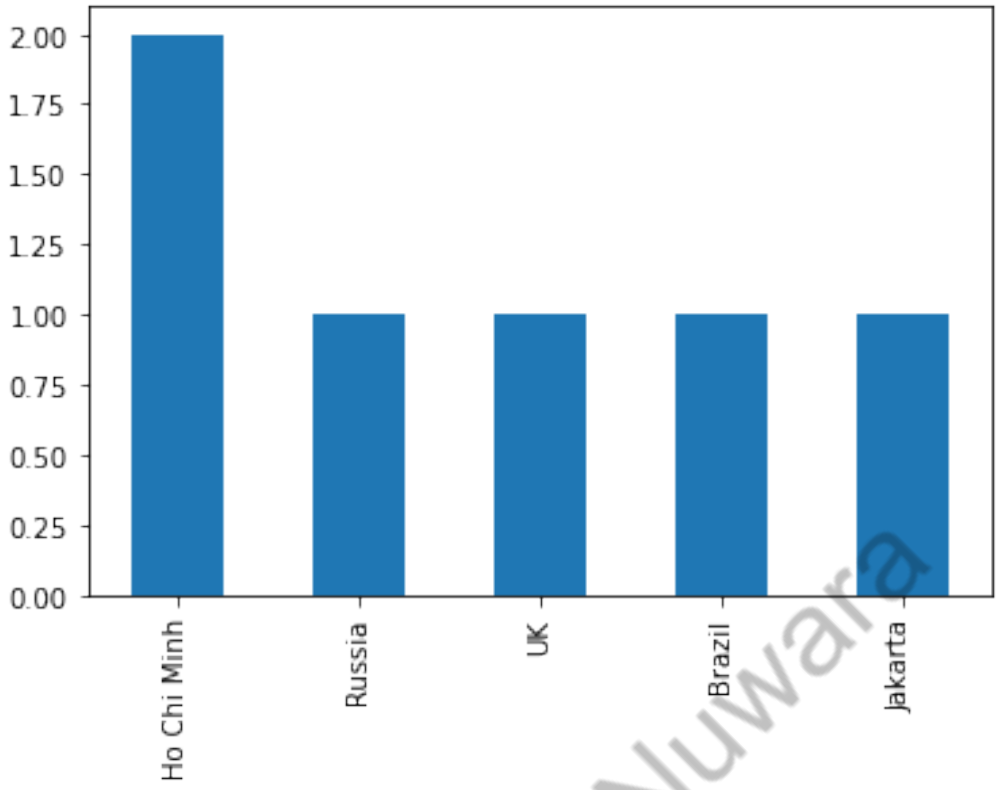
```
# Make pie diagram
df_combine['City'].value_counts().plot(kind='pie')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f86f1429e10>
```




```
# Make bar diagram
df_combine['City'].value_counts().plot(kind='bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f86f1584650>
```




```
df_combine.describe()
```

	Exp	Year_start	Number_of_pets
count	6.000000	4.000000	4.00000
mean	9.833333	1977.500000	4.25000
std	8.400397	17.078251	3.40343
min	2.000000	1960.000000	1.00000
25%	2.750000	1967.500000	2.50000
50%	7.500000	1975.000000	3.50000
75%	17.500000	1985.000000	5.25000
max	20.000000	2000.000000	9.00000

```
# Convert np to df
a = np.array([['Honda', 1, 3],
              ['Chevro', 2, 4],
              ['Hyundai', 5, 6],
              ['Tesla', 10, 20]])

car = pd.DataFrame(a, columns=['Brand', 'Mileage', 'Price'])

car
```

	Brand	Mileage	Price
0	Honda	1	3
1	Chevro	2	4

```
2  Hyundai       5     6
3    Tesla      10    20

# Convert df to np
car.values

array([['Honda', '1', '3'],
       ['Chevro', '2', '4'],
       ['Hyundai', '5', '6'],
       ['Tesla', '10', '20']], dtype=object)

# How to find out which person is in PTTEP
# To do masking
df['company']=='PTTEP'

# Assign to a new variable
cond = df['company']=='PTTEP'

# Assign the conditional var to our DF
df_update = df[cond]
```

Case Study 1

```
# Load from CSV
df = pd.read_csv('/content/SPE_Lease_Data.csv')

df.head(5)

    Lease  ...                                        Description
0  OSPREY  ...   Well began producing and was later recomplete...
1  FALCON  ...   Interior well on a 3-well pad that was shut-i...
2    HAWK  ...   Exterior well on a 3-well pad that was shut-i...
3   EAGLE  ...   Single unbounded well with modern completion ...
4    KITE  ...   Child well with modern completion design offs...

[5 rows x 95 columns]

df.columns

Index(['Lease', 'Well Number', 'State', 'Formation/Reservoir',
       ' Initial Pressure Estimate (psi) ', ' Reservoir Temperature
(deg F) ',
       ' Net Pay (ft) ', ' Wellbore Diameter (ft) ', 'Porosity',
       ' Water Saturation ', ' Oil Saturation ', ' Gas Saturation ',
       ' Gas Specific Gravity ', 'CO2', ' H2S ', 'N2',
       'Condensate Yield (Bc/MMcf)', 'Condensate Gravity (API)',
       ' Dew Point Pressure (psi) ', 'Sep. Temperature (deg F)',
       'Sep. Pressure (psi)', ' Oil Gravity (API) ', ' Initial GOR
(scf/bbl) ',
       ' Bubble Point Pressure (psi) ', ' TVD (ft) ', ' Spacing ', '#
Stages',
```

```
       '# Clusters ', '# Clusters per Stage', 'Pre-Refrac Completion #
Stages',
       'Pre-Refrac Completion # Clusters ',
       'Initian Completion # Clusters per Stage', '# of Total Proppant
(Lbs)',
       'Total Fluid (Bbls)', ' Lateral Length (ft) ', ' Top Perf (ft)
',
       ' Bottom Perf (ft) ', ' Sandface Temp (deg F) ',
       ' Static Wellhead Temp (deg F) ', ' Configuration ',
'Production Path',
       'Pressure Loss Correlation', ' Tubing ID (in) ', ' Tubing OD
(in) ',
       ' Tubing Depth (ft) ', ' Casing ID 1 (in) ', ' Casing Footage 1
(ft) ',
       ' Casing ID 2 (in) ', ' Casing Footage 2 (ft) ', ' Casing Depth
(ft) ',
       ' Configuration Change (Days Since First Prod) ', 'Production
Path.1',
       'Pressure Loss Correlation.1', ' Tubing ID (in) .1',
       ' Tubing OD (in) .1', ' Tubing Depth (ft) .1', ' Casing ID 1
(in) .1',
       ' Casing Footage 1 (ft) .1', ' Casing ID 2 (in) .1',
       ' Casing Footage 2 (ft) .1', ' Casing Depth (ft) .1',
       ' Configuration Change (Days Since First Prod) .1', 'Production
Path.2',
       'Pressure Loss Correlation.2', ' Tubing ID (in) .2',
       ' Tubing OD (in) .2', ' Tubing Depth (ft) .2', ' Casing ID 1
(in) .2',
       ' Casing Footage 1 (ft) .2', ' Casing ID 2 (in) .2',
       ' Casing Footage 2 (ft) .2', ' Casing Depth (ft) .2',
       ' Configuration Change (Days Since First Prod) .2', 'Production
Path.3',
       'Pressure Loss Correlation.3', ' Tubing ID (in) .3',
       ' Tubing OD (in) .3', ' Tubing Depth (ft) .3', ' Casing ID 1
(in) .3',
       ' Casing Footage 1 (ft) .3', ' Casing ID 2 (in) .3',
       ' Casing Footage 2 (ft) .3', ' Casing Depth (ft) .3',
       ' Configuration Change (Days Since First Prod) .3', 'Production
Path.4',
       'Pressure Loss Correlation.4', ' Tubing ID (in) .4',
       ' Tubing OD (in) .4', ' Tubing Depth (ft) .4', ' Casing ID 1
(in) .4',
       ' Casing Footage 1 (ft) .4', ' Casing ID 2 (in) .4',
       ' Casing Footage 2 (ft) .4', ' Casing Depth (ft) .4', '
Description '],
      dtype='object')

df['Formation/Reservoir']
```

```
0             EAGLE FORD
1             EAGLE FORD
2             EAGLE FORD
3             EAGLE FORD
4             EAGLE FORD
5             EAGLE FORD
6             EAGLE FORD
7             EAGLE FORD
8             EAGLE FORD
9             EAGLE FORD
10            EAGLE FORD
11    HAYNESVILLE SHALE
12    HAYNESVILLE SHALE
13    HAYNESVILLE SHALE
14    HAYNESVILLE SHALE
15    HAYNESVILLE SHALE
16    HAYNESVILLE SHALE
17    HAYNESVILLE SHALE
18    HAYNESVILLE SHALE
19    HAYNESVILLE SHALE
20       BOSSIER SHALE
21    HAYNESVILLE SHALE
22    HAYNESVILLE SHALE
23    HAYNESVILLE SHALE
24    HAYNESVILLE SHALE
25    HAYNESVILLE SHALE
26             MARCELLUS
27             MARCELLUS
28             MARCELLUS
29             MARCELLUS
30             MARCELLUS
31             MARCELLUS
32             MARCELLUS
33             MARCELLUS
34             MARCELLUS
35             MARCELLUS
36             MARCELLUS
37    MARCELLUS - UPPER
38    MARCELLUS - UPPER
39    MARCELLUS - UPPER
40    MARCELLUS - UPPER
41    MARCELLUS - UPPER
42    MARCELLUS - UPPER
43    MARCELLUS - UPPER
44    MARCELLUS - UPPER
45    MARCELLUS - UPPER
46    MARCELLUS - UPPER
47    MARCELLUS - UPPER
48    MARCELLUS - UPPER
49    MARCELLUS - UPPER
```

```
50    MARCELLUS - UPPER
51    MARCELLUS - UPPER
52    MARCELLUS - UPPER
Name: Formation/Reservoir, dtype: object
```

```
df.describe().T
```

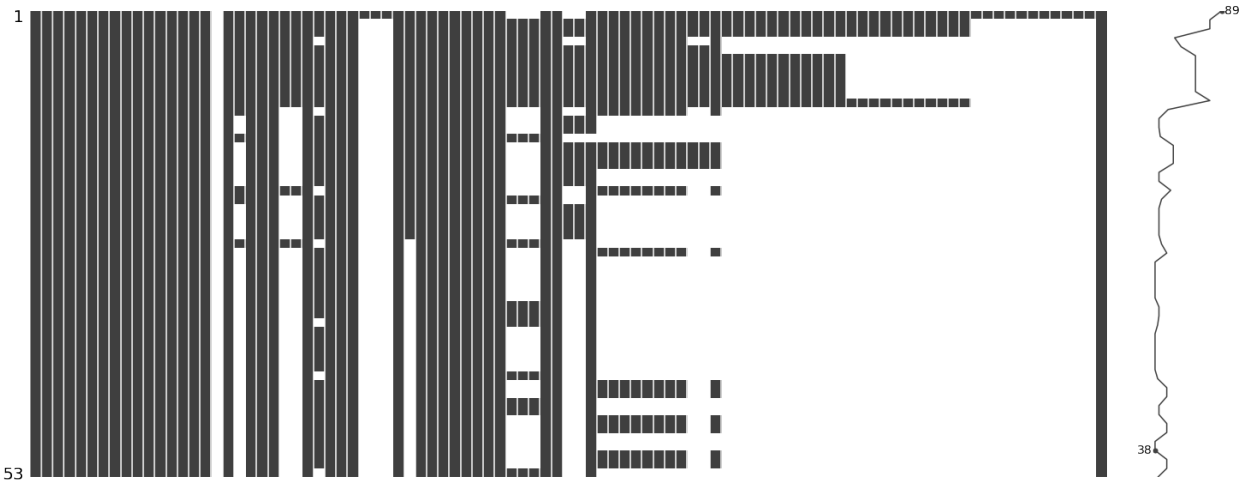
```
                                   count          mean  ...         75%
max
Well Number                         53.0     39.113208  ...      63.000
76.000
 Initial Pressure Estimate (psi)    53.0   6274.320755  ...    9900.000
12223.000
 Reservoir Temperature (deg F)      53.0    211.867925  ...     305.000
379.000
 Net Pay (ft)                       53.0    157.830189  ...     208.000
268.000
 Wellbore Diameter (ft)             53.0      0.700000  ...       0.700
0.700
...                                  ...           ...  ...         ...
...
 Casing ID 1 (in) .4                 1.0      4.670000  ...       4.670
4.670
 Casing Footage 1 (ft) .4            1.0   6941.000000  ...    6941.000
6941.000
 Casing ID 2 (in) .4                 1.0      2.992000  ...       2.992
2.992
 Casing Footage 2 (ft) .4            1.0   6307.000000  ...    6307.000
6307.000
 Casing Depth (ft) .4                1.0  13248.000000  ...   13248.000
13248.000

[80 rows x 8 columns]
```

```
df['Formation/Reservoir'].unique()
```

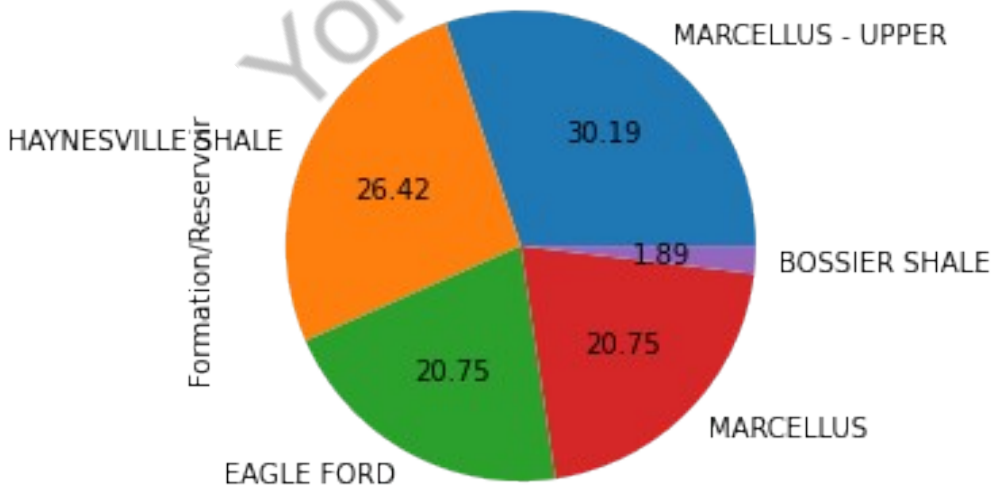
```
array(['EAGLE FORD', 'HAYNESVILLE SHALE', 'BOSSIER SHALE ',
'MARCELLUS',
       'MARCELLUS - UPPER'], dtype=object)
```

```
msno.matrix(df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f86f0b707d0>
```

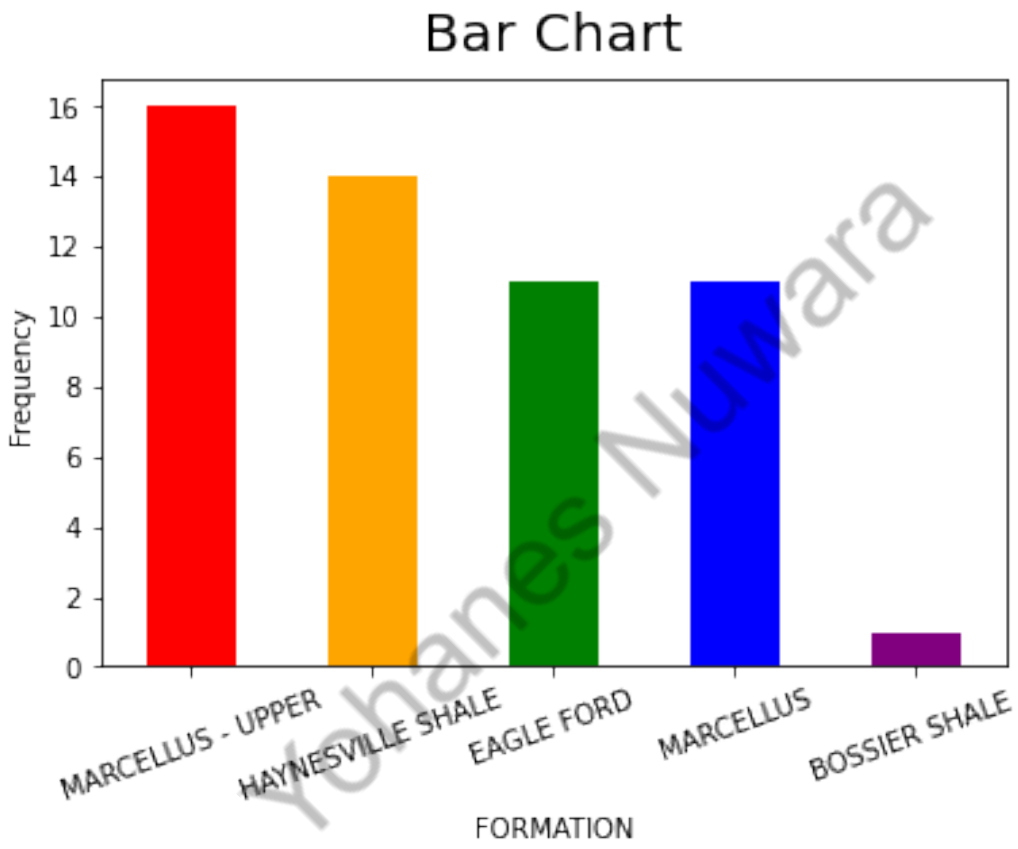



```
# Pie diagram for respective states
df['Formation/Reservoir'].value_counts()

MARCELLUS - UPPER    16
HAYNESVILLE SHALE    14
EAGLE FORD           11
MARCELLUS            11
BOSSIER SHALE         1
Name: Formation/Reservoir, dtype: int64

df['Formation/Reservoir'].value_counts().plot(kind='pie',
autopct='%.2f')

<matplotlib.axes._subplots.AxesSubplot at 0x7f86ef195590>
```

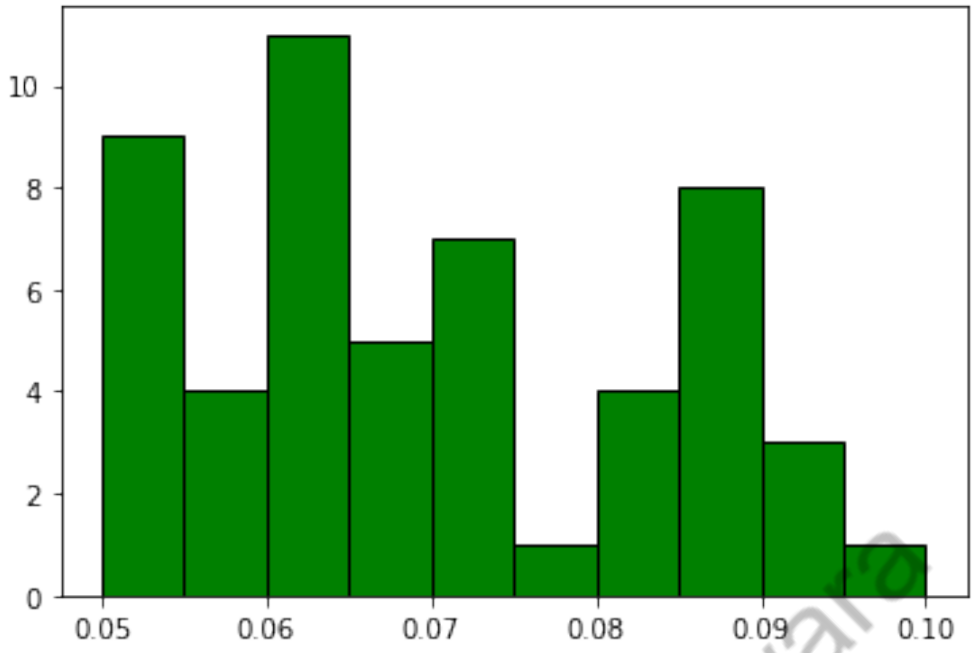


```
colors = ['red', 'orange', 'green', 'blue', 'purple']

df['Formation/Reservoir'].value_counts().plot(kind='bar',
color=colors, rot=20)
plt.xlabel('FORMATION')
plt.ylabel('Frequency')
plt.title('Bar Chart', size=20, pad=10)
```

```
Text(0.5, 1.0, 'Bar Chart')
```

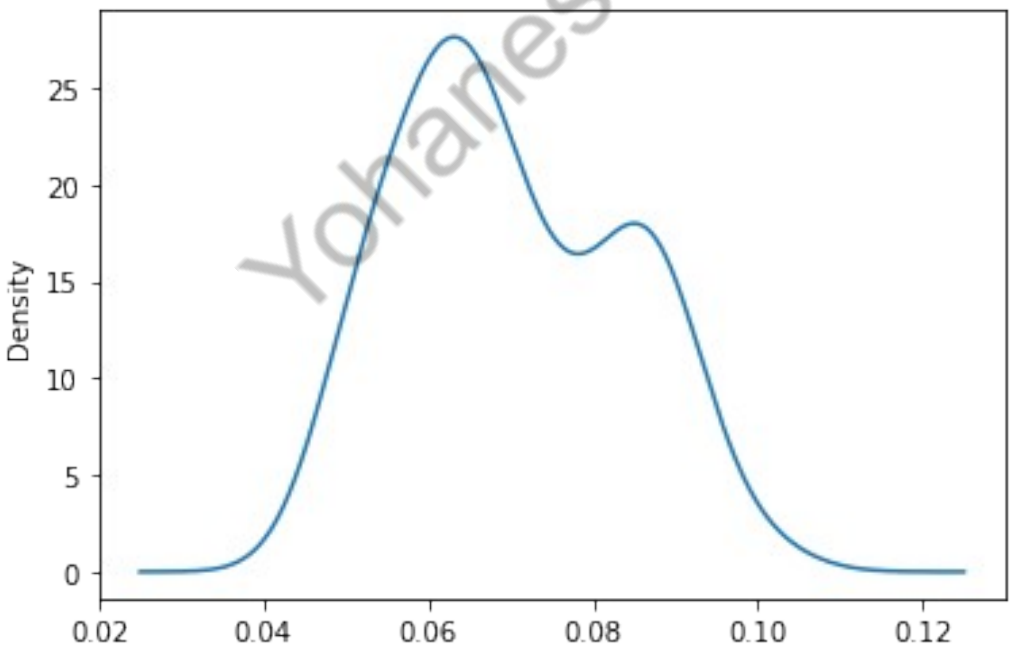



```
# Distribution of porosity
plt.hist(df['Porosity'], bins=10, lw=1, edgecolor='k', color='green')
# plt.grid()
plt.show()
```



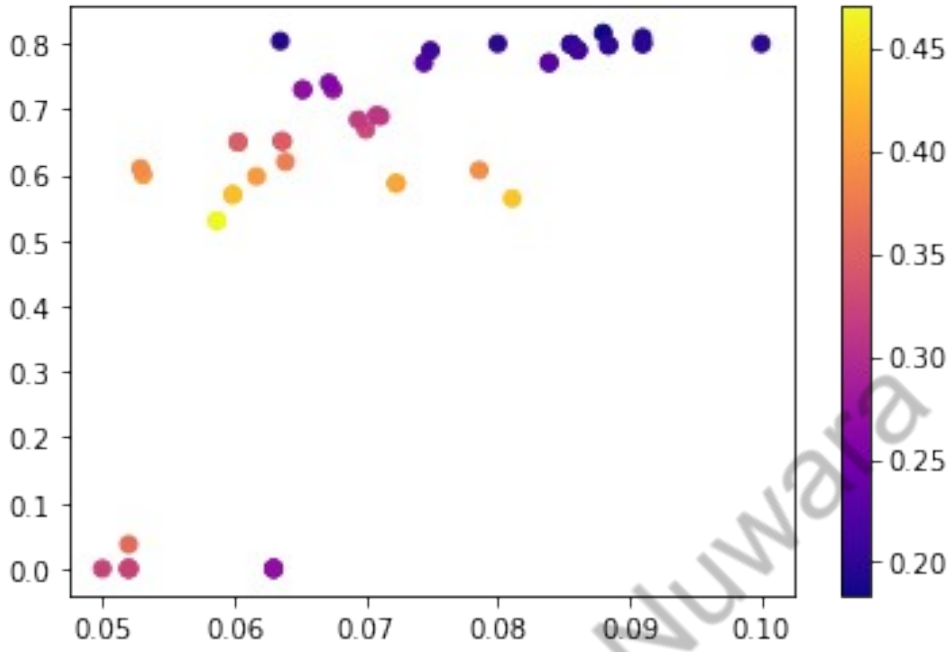

```
# Kernel density estimation (KDE)
df['Porosity'].plot(kind='kde')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f86ee5f3d90>
```




```
# Crossplot porosity and Sg
plt.scatter(df['Porosity'], df[' Gas Saturation '], c=df[' Water
Saturation '], cmap='plasma')
```

```
# plt.yscale('log')
plt.colorbar()
plt.show()
```




```
# Correlation
df.corr().round(2).head(20)
```

	Well Number	...	Casing Depth (ft) .4
Well Number	1.00	...	NaN
Initial Pressure Estimate (psi)	0.70	...	NaN
Reservoir Temperature (deg F)	0.54	...	NaN
Net Pay (ft)	0.66	...	NaN
Wellbore Diameter (ft)	0.00	...	NaN
Porosity	0.77	...	NaN
Water Saturation	-0.39	...	NaN
Oil Saturation	-0.74	...	NaN
Gas Saturation	0.81	...	NaN
Gas Specific Gravity	-0.60	...	NaN

```
CO2                                          0.64  ...
NaN
 H2S                                          NaN  ...
NaN
N2                                          -0.27  ...
NaN
Condensate Yield (Bc/MMcf)                   NaN  ...
NaN
Condensate Gravity (API)                    -0.11  ...
NaN
 Dew Point Pressure (psi)                    0.45  ...
NaN
Sep. Temperature (deg F)                     NaN  ...
NaN
Sep. Pressure (psi)                          NaN  ...
NaN
 Oil Gravity (API)                           0.21  ...
NaN
 Initial GOR (scf/bbl)                      -0.51  ...
NaN

[20 rows x 80 columns]
```

Task 1: List top 10 lease which has the highest porosity

```
# Sort values
top5 = df.sort_values('Porosity', ascending=False).head(10)

top5
```

```
            Lease  ...
Description
20         CUCKOO ...                                    SNG; Bossier
well
19           LOON  ...     CLS; Wider CS & increase #/cl completion
test
18      BEE-EATER  ...   CCD; Modern completion comparison for twin
pa...
14      CASSOWARY  ...   PDB; CLS parent later offset by modern
comple...
13      PARTRIDGE  ...   CCD; Modern completions infilled between
lega...
12          MYNAH  ...   CCD; Modern completions infilled between
lega...
11       LORIKEET  ...                         SNG; Unbounded legacy
well
22  ARCHAEOPTERYX  ...   CCD; Modern completions infilled between
lega...
23           TERN  ...   CCD; Modern completions infilled between
```

```
lega...
17          HARPY  ...   CCD; Modern 3 well development infilled
betwe...

[10 rows x 95 columns]
```

```
top5['Lease'].unique()
```

```
array(['CUCKOO', 'LOON', 'BEE-EATER', 'CASSOWARY', 'PARTRIDGE'],
      dtype=object)
```

Task 2: List 10 most oil saturated lease

```
# Sort values
df.sort_values(' Oil Saturation ', ascending=False).head(10)
```

```
       Lease  ...                                        Description
0     OSPREY  ...   Well began producing and was later recomplete...
1     FALCON  ...   Interior well on a 3-well pad that was shut-i...
3      EAGLE  ...   Single unbounded well with modern completion ...
2       HAWK  ...   Exterior well on a 3-well pad that was shut-i...
4       KITE  ...   Child well with modern completion design offs...
10      CROW  ...   Child well that was codeveloped with adjacent...
5      SWIFT  ...   Exterior well on a 2-well pad with modern com...
6    SPARROW  ...   Exterior well on a 2-well pad with modern com...
8   CARDINAL  ...   Exterior well on a 3-well pad with modern com...
7       LARK  ...   Exterior well on a 3-well pad with modern com...

[10 rows x 95 columns]
```

```
# Sort values
df.sort_values(' Gas Saturation ', ascending=False).head(10)
```

```
        Lease  ...                                        Description
11   LORIKEET  ...                          SNG; Unbounded legacy well
14  CASSOWARY  ...   PDB; CLS parent later offset by modern comple...
27        EMU  ...           CLUSTER BOUND; Modern Completion design
21    ROOSTER  ...                CLS; Increase #/cl completion test
20     CUCKOO  ...                                SNG; Bossier well
19       LOON  ...     CLS; Wider CS & increase #/cl completion test
18  BEE-EATER  ...   CCD; Modern completion comparison for twin pa...
17      HARPY  ...   CCD; Modern 3 well development infilled betwe...
16    WARBLER  ...   CLD; Modern 3 well development infilled betwe...
15     ORIOLE  ...   CCD; Modern 3 well development infilled betwe...

[10 rows x 95 columns]
```

Task 3: List top 5 thickest net pay in Pennsylvania (PA)

```
# Do masking
PA_df = df[df['State']=='PA']

# Sort values
PA_df.sort_values(' Net Pay (ft) ', ascending=False).head(5)
```

```
       Lease  ...                              Description
43       OWL  ...   1200' innerwell unbounded exterior
42   PELICAN  ...   1200' innerwell unbounded exterior
47    CANARY  ...   1500' innerwell unbounded exterior
46  BLUEBIRD  ...   1500' innerwell unbounded exterior
40   PENGUIN  ...   1500' innerwell unbounded exterior

[5 rows x 95 columns]
```

Groupby and pivoting

```
df_combine
```

```
     Name    company         City  Exp  Year_start  Number_of_pets
0  Nuwara        OYO      Jakarta    2      1960.0             9.0
1     Nha      PTTEP  Ho Chi Minh   10      1970.0             3.0
2    Tien      PTTEP  Ho Chi Minh   10      1980.0             4.0
3  Andres  Petrobras       Brazil    5      2000.0             1.0
4    John         BP           UK   10         NaN             NaN
5    Vlad      Total       Russia    2         NaN             NaN
```

```
df_combine.groupby('City')[['Number_of_pets', 'Exp']].mean()
```

```
             Number_of_pets  Exp
City
Brazil                  1.0    5
Ho Chi Minh             3.5   10
Jakarta                 9.0    2
Russia                  NaN    2
UK                      NaN   10
```

```
df = pd.read_csv('/content/SPE_Lease_Data.csv')
```

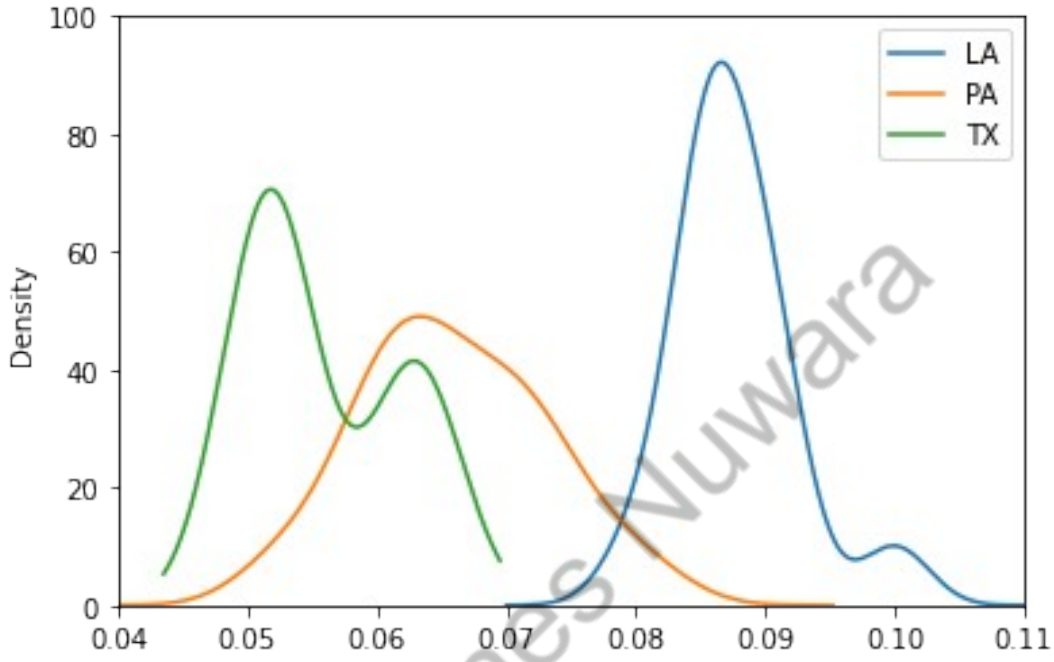
Task 4: Group by each formation, calculate the average of Porosity and Oil Saturation

```
df.groupby('Formation/Reservoir')[['Porosity', ' Oil Saturation
']].mean()
```

```
                     Porosity   Oil Saturation
Formation/Reservoir
BOSSIER SHALE        0.100000          0.000000
EAGLE FORD           0.055818          0.682091
HAYNESVILLE SHALE    0.086750          0.000000
```

```
MARCELLUS             0.071009          0.000118
MARCELLUS - UPPER     0.062444          0.000000
```

```
df.groupby('State')['Porosity'].plot(kind='kde', legend='State')
plt.xlim(0.04, 0.11)
plt.ylim(0, 100)
plt.show()
```



Task 4: Group by each formation, print the whole summary stats

```
df.groupby('Formation/Reservoir')[['Porosity', ' Oil Saturation
']].describe().T
```

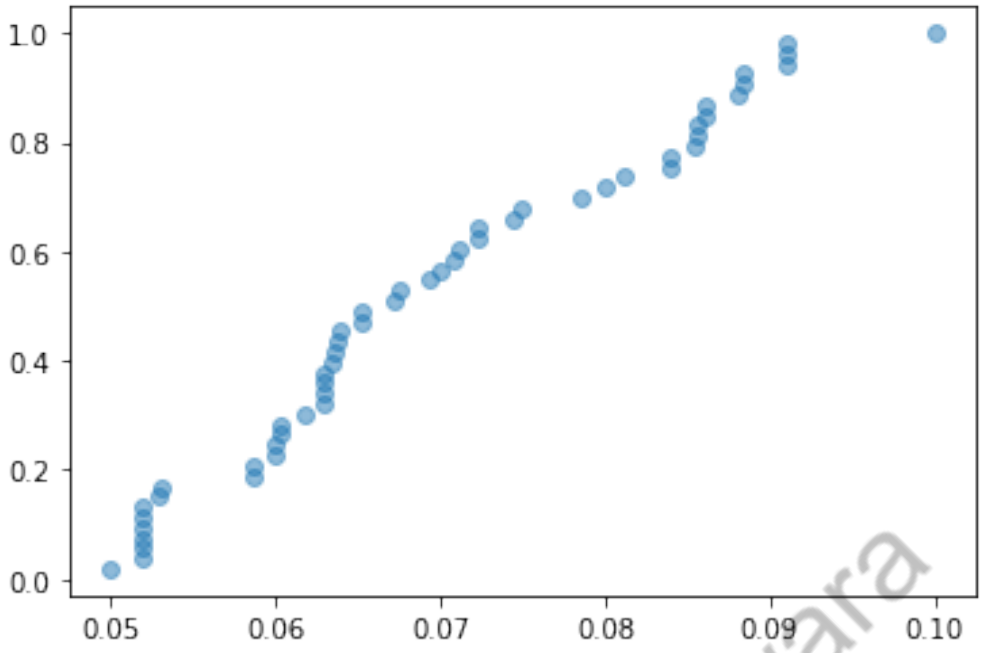
```
Formation/Reservoir     BOSSIER SHALE   ...  MARCELLUS - UPPER
Porosity         count              1.0  ...          16.000000
                 mean               0.1  ...           0.062444
                 std                NaN  ...           0.005709
                 min                0.1  ...           0.052900
                 25%                0.1  ...           0.059600
                 50%                0.1  ...           0.061000
                 75%                0.1  ...           0.065700
                 max                0.1  ...           0.072300
 Oil Saturation  count              1.0  ...          16.000000
                 mean               0.0  ...           0.000000
                 std                NaN  ...           0.000000
                 min                0.0  ...           0.000000
                 25%                0.0  ...           0.000000
                 50%                0.0  ...           0.000000
                 75%                0.0  ...           0.000000
```

```
                  max                0.0  ...           0.000000

[16 rows x 5 columns]
```

```
# Pivoting
pd.pivot_table(df, index=['State', 'Formation/Reservoir'],
values=['Porosity', ' Net Pay (ft) '],
               aggfunc=[np.mean, np.std])
```

```
                                    mean                       std

                             Net Pay (ft)   Porosity  Net Pay (ft)
Porosity
State Formation/Reservoir

LA    BOSSIER SHALE           150.000000  0.100000            NaN
NaN
      HAYNESVILLE SHALE       180.714286  0.086750      31.303548
0.003139
PA    MARCELLUS               138.090909  0.071009      13.974652
0.005954
      MARCELLUS - UPPER       214.687500  0.062444      31.584213
0.005709
TX    EAGLE FORD               66.454545  0.055818       5.888355
0.005724
```

```
def ecdf(data):
    """
    Plot ECDF (Empirical Cumulative Distribution Function)
    """
    n = len(data)
    x = np.sort(data)
    y = np.arange(1, n+1) / n

    plt.scatter(x, y, alpha=0.5)

ecdf(df['Porosity'])
```



Time-series analysis

29 October 2021 -> 2021-10-29, 2021/10/29, 29-Oct-21, 29/10/2021

Pandas -> 2021-10-29

```
volve = pd.read_csv('/content/Volve_Production_Data.csv')

volve.head(10)
```

```
     DATEPRD NPD_WELL_BORE_NAME  ...  BORE_WI_VOL   FLOW_KIND
0  07-Apr-14          15/9-F-1 C  ...          NaN  production
1  08-Apr-14          15/9-F-1 C  ...          NaN  production
2  09-Apr-14          15/9-F-1 C  ...          NaN  production
3  10-Apr-14          15/9-F-1 C  ...          NaN  production
4  11-Apr-14          15/9-F-1 C  ...          NaN  production
5  12-Apr-14          15/9-F-1 C  ...          NaN  production
6  13-Apr-14          15/9-F-1 C  ...          NaN  production
7  14-Apr-14          15/9-F-1 C  ...          NaN  production
8  15-Apr-14          15/9-F-1 C  ...          NaN  production
9  16-Apr-14          15/9-F-1 C  ...          NaN  production

[10 rows x 16 columns]
```

```
# Convert to Pandas datetime
volve['DATEPRD'] = pd.to_datetime(volve['DATEPRD'], format='%d-%b-%y')

volve.head()
```

```
     DATEPRD NPD_WELL_BORE_NAME  ...  BORE_WI_VOL   FLOW_KIND
0 2014-04-07         15/9-F-1 C  ...          NaN  production
1 2014-04-08         15/9-F-1 C  ...          NaN  production
2 2014-04-09         15/9-F-1 C  ...          NaN  production
3 2014-04-10         15/9-F-1 C  ...          NaN  production
4 2014-04-11         15/9-F-1 C  ...          NaN  production

[5 rows x 16 columns]
```

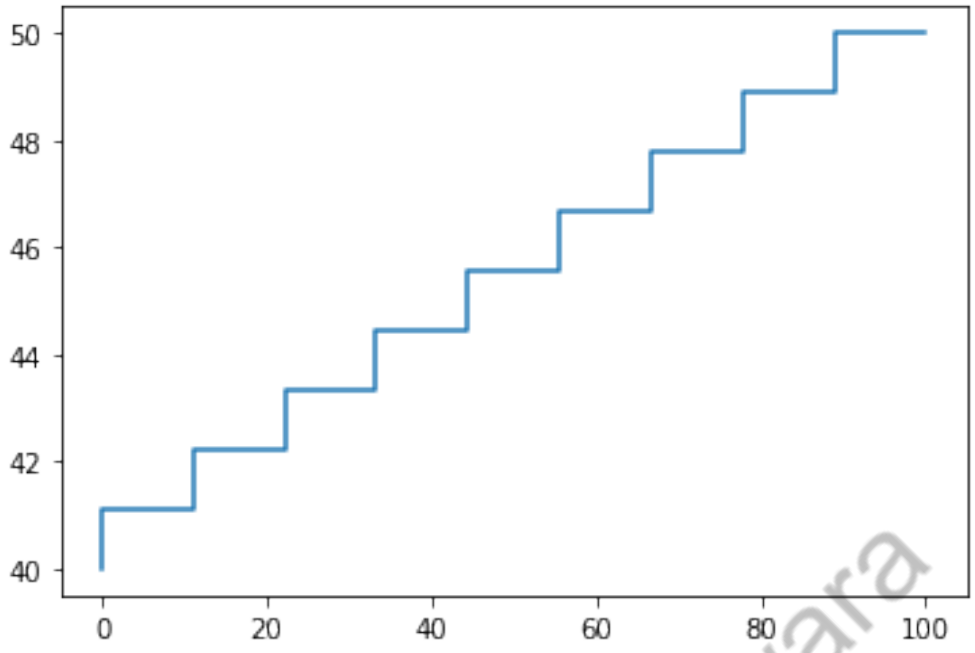
Task: Isolate the dataframe from only well 15/9-F-14

```
# Do masking
f14 = volve[volve['NPD_WELL_BORE_NAME']=='15/9-F-
14'].reset_index(drop=True)

f14
```

```
        DATEPRD NPD_WELL_BORE_NAME  ...  BORE_WI_VOL   FLOW_KIND
0    2008-02-12          15/9-F-14  ...          NaN  production
1    2008-02-13          15/9-F-14  ...          NaN  production
2    2008-02-14          15/9-F-14  ...          NaN  production
3    2008-02-15          15/9-F-14  ...          NaN  production
4    2008-02-16          15/9-F-14  ...          NaN  production
...         ...                ...  ...          ...         ...
3051 2016-09-13          15/9-F-14  ...          NaN  production
3052 2016-09-14          15/9-F-14  ...          NaN  production
3053 2016-09-15          15/9-F-14  ...          NaN  production
3054 2016-09-16          15/9-F-14  ...          NaN  production
3055 2016-09-17          15/9-F-14  ...          NaN  production

[3056 rows x 16 columns]
```

```
x = np.linspace(0, 100, 10)
y = np.linspace(40, 50, 10)

plt.step(x, y, where='pre')
```

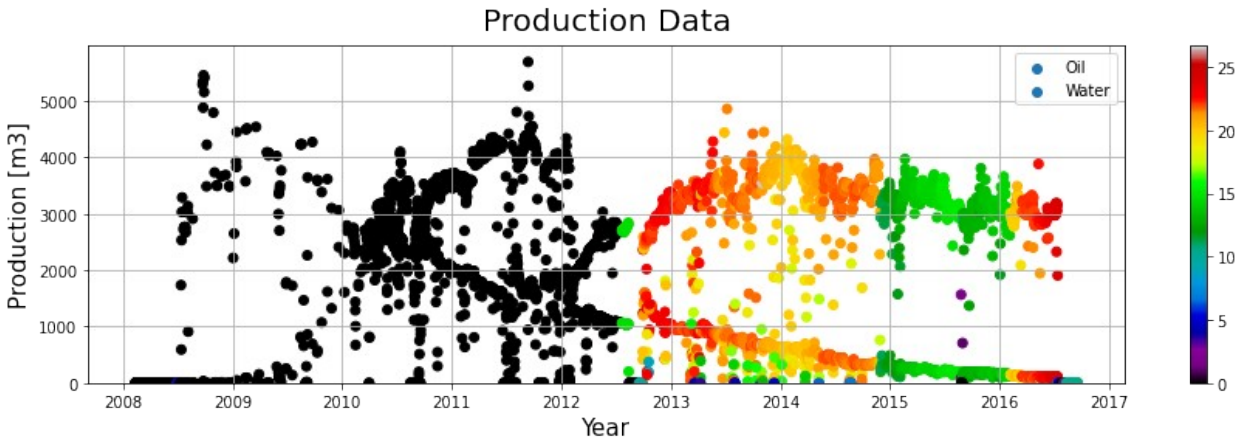
```
[<matplotlib.lines.Line2D at 0x7f1157b9d450>]
```




```
f14.columns
```

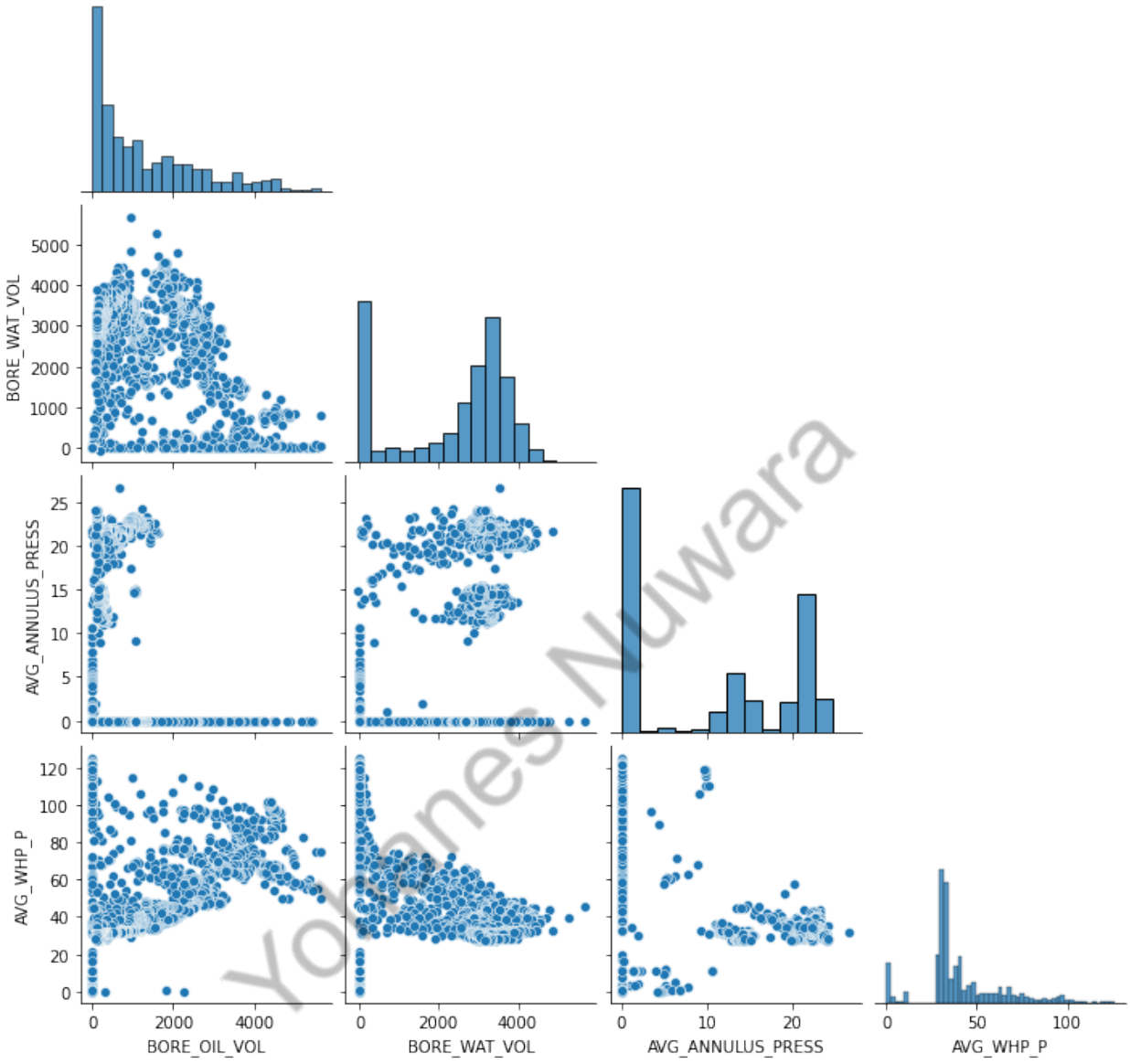
```
Index(['DATEPRD', 'NPD_WELL_BORE_NAME', 'ON_STREAM_HRS',
       'AVG_DOWNHOLE_PRESSURE', 'AVG_DOWNHOLE_TEMPERATURE',
'AVG_DP_TUBING',
       'AVG_ANNULUS_PRESS', 'AVG_CHOKE_SIZE_P', 'AVG_WHP_P',
'AVG_WHT_P',
       'DP_CHOKE_SIZE', 'BORE_OIL_VOL', 'BORE_GAS_VOL',
'BORE_WAT_VOL',
       'BORE_WI_VOL', 'FLOW_KIND'],
      dtype='object')
```

```
x = f14['DATEPRD'].values
y = f14['BORE_OIL_VOL'].values
y2 = f14['BORE_WAT_VOL'].values
color = f14['AVG_ANNULUS_PRESS'].values

plt.figure(figsize=(15,4))
plt.scatter(x, y, label='Oil', c=color, cmap='nipy_spectral')
plt.scatter(x, y2, label='Water', c=color, cmap='nipy_spectral')
plt.colorbar()
plt.xlabel('Year', size=15)
plt.ylabel('Production [m3]', size=15)
plt.title('Production Data', size=20, pad=10)
plt.ylim(ymin=0)
plt.legend()
plt.grid()
plt.show()
```




```
# Make it interactive
fig = px.scatter(f14, 'DATEPRD', 'BORE_OIL_VOL',
color='AVG_ANNULUS_PRESS')
fig.show()

# Pairplot
variables = ['BORE_OIL_VOL', 'BORE_WAT_VOL', 'AVG_ANNULUS_PRESS',
'AVG_WHP_P']

sns.pairplot(f14, vars=variables, corner=True)
```

```
<seaborn.axisgrid.PairGrid at 0x7f1154c482d0>
```



Session 2 Python for Exploration: Petrophysical Analysis of Well Log Data

As usual, import the three core libraries of Python

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

Stream data directly into Google Colab: Dutch F3 logs

Data is often stored in a website. In Colab, we can stream directly from the website **without downloading it into our PC**. We use `!wget`.

```
!wget 'https://github.com/yohanesnuwara/python-bootcamp-for-geoengineers/blob/master/data/Dutch_F3_Logs.zip?raw=true'
```

```
--2020-08-30 07:32:54--  https://github.com/yohanesnuwara/python-bootcamp-for-geoengineers/blob/master/data/Dutch_F3_Logs.zip?raw=true
Resolving github.com (github.com)... 140.82.112.4
Connecting to github.com (github.com)|140.82.112.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://github.com/yohanesnuwara/python-bootcamp-for-geoengineers/raw/master/data/Dutch_F3_Logs.zip [following]
--2020-08-30 07:32:55--  https://github.com/yohanesnuwara/python-bootcamp-for-geoengineers/raw/master/data/Dutch_F3_Logs.zip
Reusing existing connection to github.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/yohanesnuwara/python-bootcamp-for-geoengineers/master/data/Dutch_F3_Logs.zip [following]
--2020-08-30 07:32:55--
https://raw.githubusercontent.com/yohanesnuwara/python-bootcamp-for-geoengineers/master/data/Dutch_F3_Logs.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
151.101.0.133, 151.101.64.133, 151.101.128.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|
151.101.0.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1100414 (1.0M) [application/zip]
Saving to: ‘Dutch_F3_Logs.zip?raw=true’

Dutch_F3_Logs.zip?r 100%[===================>]   1.05M  --.-KB/s    in
0.1s

2020-08-30 07:32:56 (7.59 MB/s) - ‘Dutch_F3_Logs.zip?raw=true’ saved
```

```
[1100414/1100414]
```

A new file will then be downloaded under `/content`. The name of the file is still `.zip?raw=true`. We must rename it first to `.zip`.

Then do unzipping.

```
!unzip '/content/Dutch_F3_Logs.zip'

Archive:  /content/Dutch_F3_Logs.zip
   creating: Dutch_F3_Logs/
  inflating: Dutch_F3_Logs/F02-1_logs.las
  inflating: Dutch_F3_Logs/F02-1_markers.txt
  inflating: Dutch_F3_Logs/F03-2_logs.las
  inflating: Dutch_F3_Logs/F03-2_markers.txt
  inflating: Dutch_F3_Logs/F03-4_logs.las
  inflating: Dutch_F3_Logs/F03-4_markers.txt
  inflating: Dutch_F3_Logs/F06-1_logs.las
  inflating: Dutch_F3_Logs/F06-1_markers.txt
```

Copy, paste, and define the file path of each log (there are 4 logs)

```
F021_path = '/content/Dutch_F3_Logs/F02-1_logs.las'
F032_path = '/content/Dutch_F3_Logs/F03-2_logs.las'
F034_path = '/content/Dutch_F3_Logs/F03-4_logs.las'
F061_path = '/content/Dutch_F3_Logs/F06-1_logs.las'
```

Read log data

First, we will install a specific Python library for well-log data, called `lasio`. Google Colab doesn't have `lasio`, so we need to do `!pip install`

```
!pip install lasio

Requirement already satisfied: lasio in /usr/local/lib/python3.6/dist-
packages (0.25.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-
packages (from lasio) (1.18.5)
```

Then import the library (as usual)

```
import lasio
```

You already have the filepaths, then you can directly read them using `lasio`. Use `lasio.read`

```
F021 = lasio.read(F021_path)
F032 = lasio.read(F032_path)
```

```
F034 = lasio.read(F034_path)
F061 = lasio.read(F061_path)
```

```
Header section Parameter regexp=~P was not found.
Header section Parameter regexp=~P was not found.
Header section Parameter regexp=~P was not found.
Header section Parameter regexp=~P was not found.
```

Don't worry if you get messages about "header section", you're all set now.

Next, take a look at what logs are available in the log, by passing `.keys()`

```
F021.keys()
```

```
['DEPTH', 'RHOB', 'DT', 'GR', 'AI', 'AI_REL', 'PHIE']
```

What units are used by each log? Pass: `.curves`

```
F021.curves
```

```
[CurveItem(mnemonic=DEPTH, unit=M, value=, descr=1     DEPTH,
original_mnemonic=DEPTH, data.shape=(9680,)),
 CurveItem(mnemonic=RHOB, unit=kg/m3, value=, descr=2     Density,
original_mnemonic=RHOB, data.shape=(9680,)),
 CurveItem(mnemonic=DT, unit=us/m, value=, descr=3     Sonic,
original_mnemonic=DT, data.shape=(9680,)),
 CurveItem(mnemonic=GR, unit=API, value=, descr=4     Gamma Ray,
original_mnemonic=GR, data.shape=(9680,)),
 CurveItem(mnemonic=AI, unit=m/s)*(kg/m3, value=, descr=5     P-
Impedance, original_mnemonic=AI, data.shape=(9680,)),
 CurveItem(mnemonic=AI_REL, unit=m/s)*(kg/m3, value=, descr=6     P-
Impedance_rel, original_mnemonic=AI_REL, data.shape=(9680,)),
 CurveItem(mnemonic=PHIE, unit=fraction, value=, descr=7     Porosity,
original_mnemonic=PHIE, data.shape=(9680,))]
```


Visualize well log curve (Example: F021 log)

Do you still remember how to get a data from one row in Pandas? For example we have a dataframe called `df` and column name we want to retrieve is `name`. To get the data, we pass: `df['name']`

Same thing applies here. Our well name is `F021`, as our dataframe. The column name that we want to get is `RHOB`, so we pass: `F021['RHOB']`. So, we get RHOB data.

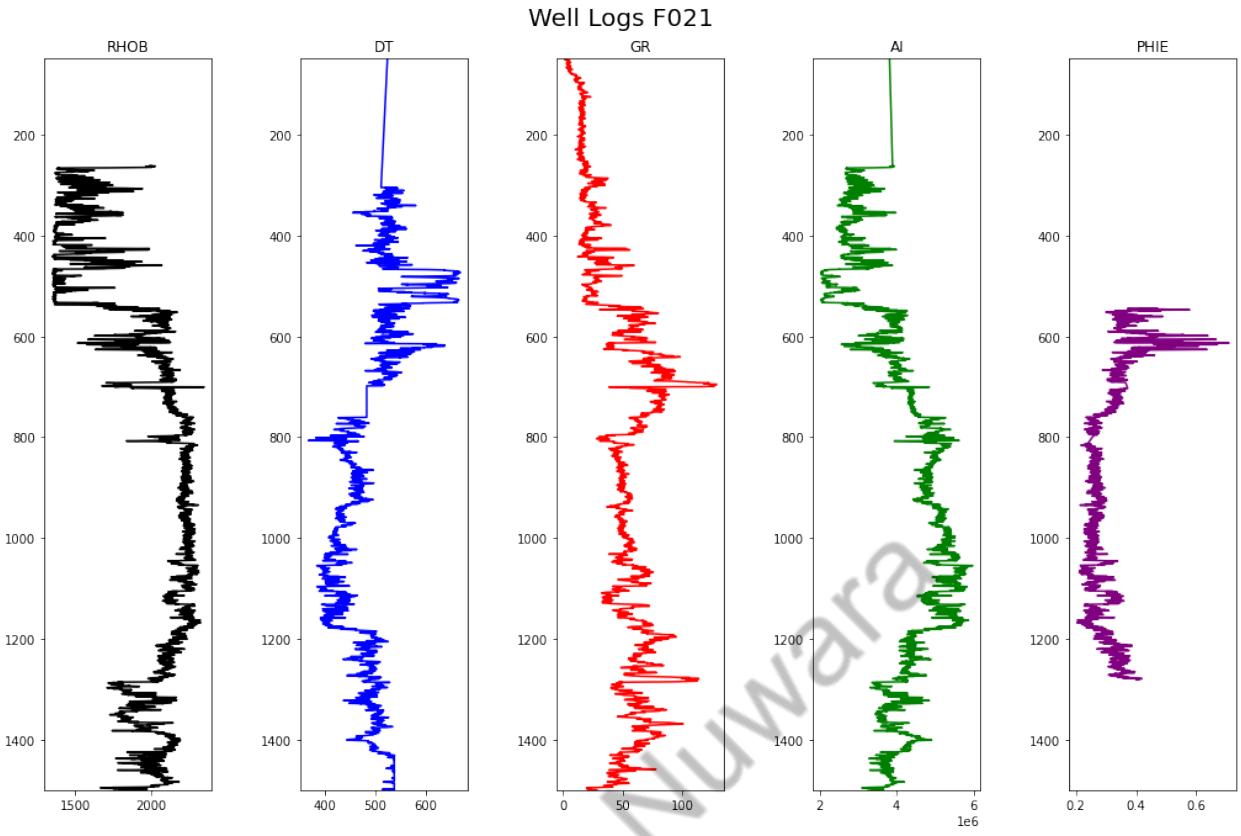
Now, get all the data.

```
depth = F021['DEPTH']
rhob = F021['RHOB']
dt = F021['DT']
```

```
gr = F021['GR']
ai = F021['AI']
phie = F021['PHIE']
```

Next, using matplotlib `plt.subplots` that we have learnt, we'll visualize the logs side-to-side.

```
plt.figure(figsize=(15,10))

plt.suptitle('Well Logs F021', size=20)

plt.subplot(1,5,1)
plt.plot(rhob, depth, color='black')
plt.ylim(max(depth), min(depth))
plt.title('RHOB')

plt.subplot(1,5,2)
plt.plot(dt, depth, color='blue')
plt.ylim(max(depth), min(depth))
plt.title('DT')

plt.subplot(1,5,3)
plt.plot(gr, depth, color='red')
plt.ylim(max(depth), min(depth))
plt.title('GR')

plt.subplot(1,5,4)
plt.plot(ai, depth, color='green')
plt.ylim(max(depth), min(depth))
plt.title('AI')

plt.subplot(1,5,5)
plt.plot(phie, depth, color='purple')
plt.ylim(max(depth), min(depth))
plt.title('PHIE')

# set space between logs
plt.tight_layout(4)
plt.show()
```

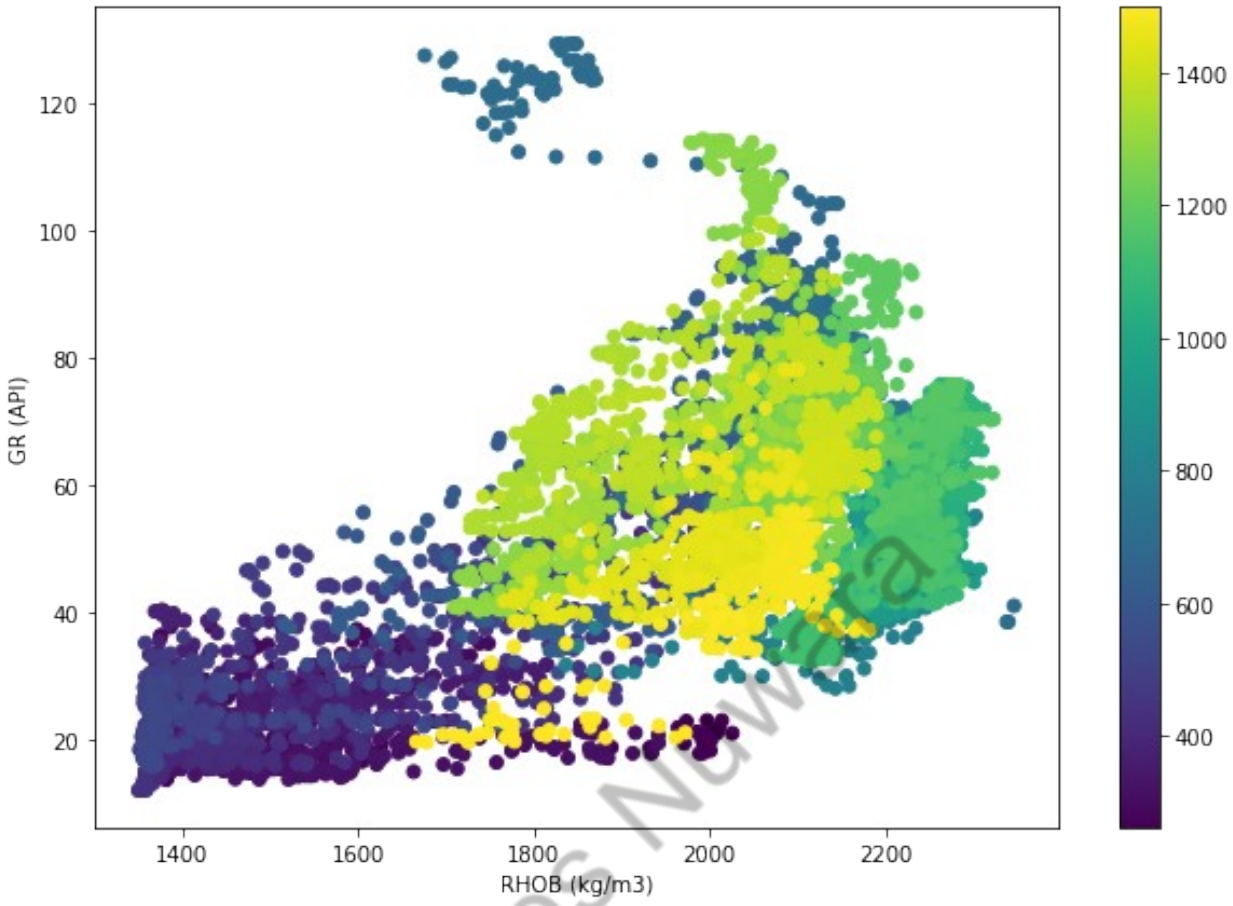


Basic Petrophysical Analysis

Crossploting Logs

Let's do crossplot of logs. Here we use the "scatter method" in Matplotlib: `plt.scatter`

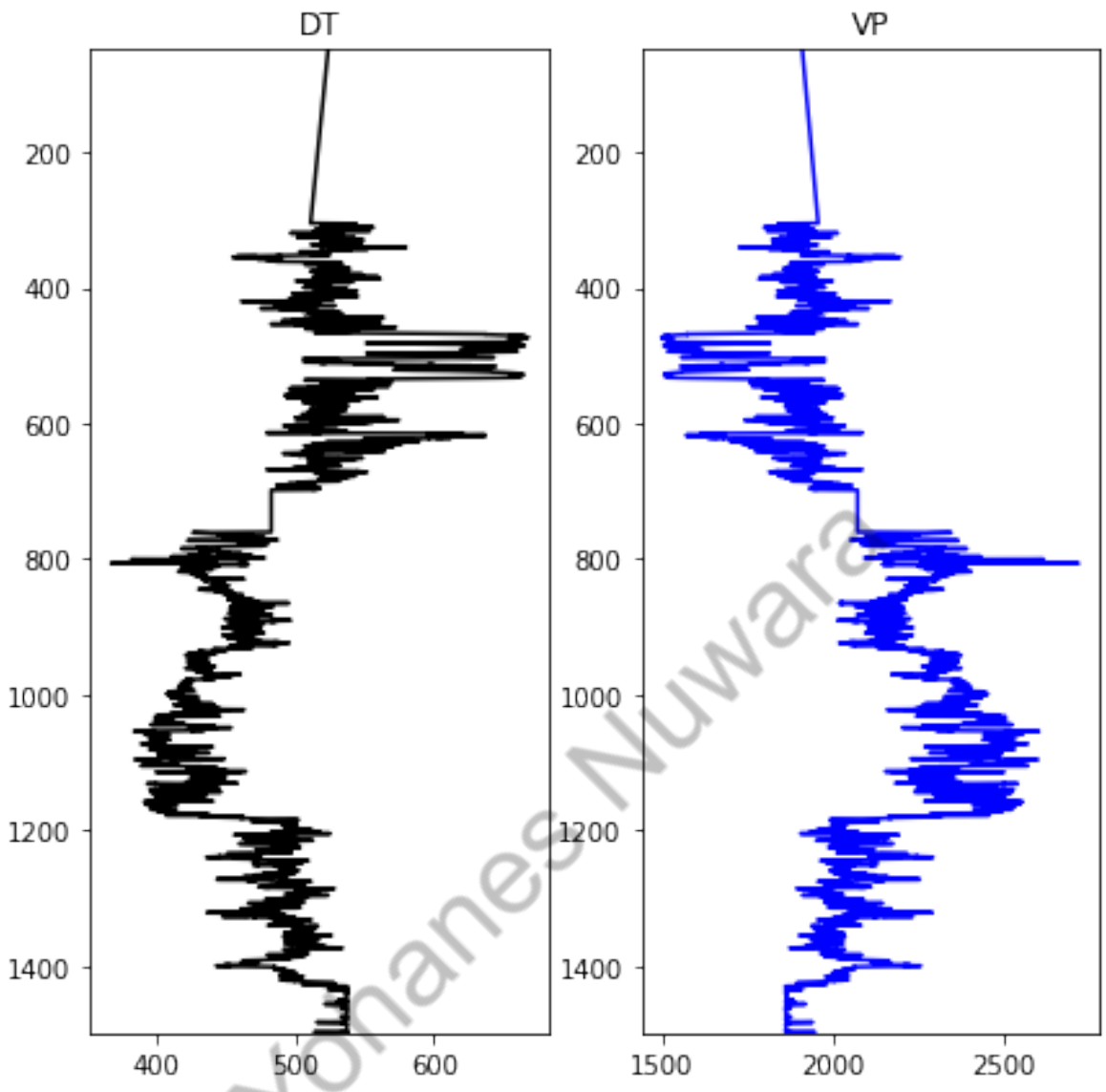
```
plt.figure(figsize=(10,7))

plt.scatter(rhob, gr, c=depth)

plt.xlabel('RHOB (kg/m3)')
plt.ylabel('GR (API)')
plt.colorbar()
plt.show()
```



Convert DT log to VP log

Our DT data has unit of $\mu s/m$. We'd like to convert it to a velocity unit, thus VP log, with unit of m/s. Then display the converted log

```
# convert DT to VP
vp = 1E+06 / dt

# display the converted log
plt.figure(figsize=(7,7))

plt.subplot(1,2,1)
plt.plot(dt, depth, color='black')
plt.ylim(max(depth), min(depth))
plt.title('DT')

plt.subplot(1,2,2)
plt.plot(vp, depth, color='blue')
plt.ylim(max(depth), min(depth))
plt.title('VP')

plt.show()
```



Produce VSHALE log from GR

We calculate shale volume (VSH) using Gamma Ray (GR), by this formula:

$$VSH = \frac{GR - GR_{min}}{GR_{max} - GR_{min}}$$

First, we need to detect if there is NaN in our data. In this calculation, we need the min and max value of GR.

```
np.min(gr), np.max(gr)
```

```
(nan, nan)
```

The result above shows there's NaN in our data, meaning that we need to handle them first. **Removing NaN is not preferrable**, so we could replace the NaN value with the average value of the data. This is what we call **IMPUTATION**.

The imputation function has been provided to you as follows.

```
def fillna(log):
  """
  Replace (Impute) NaN values with Average Values
  """
  log_nan_remove = log[~np.isnan(log)]
  mean = np.mean(log_nan_remove)
  log[np.isnan(log)] = mean
  return log
```

Let's impute or data

```
gr_imputed = fillna(gr)
```

So, when we compute the min and max value, it won't return NaN anymore, and all set to go.

```
np.min(gr), np.max(gr)
```

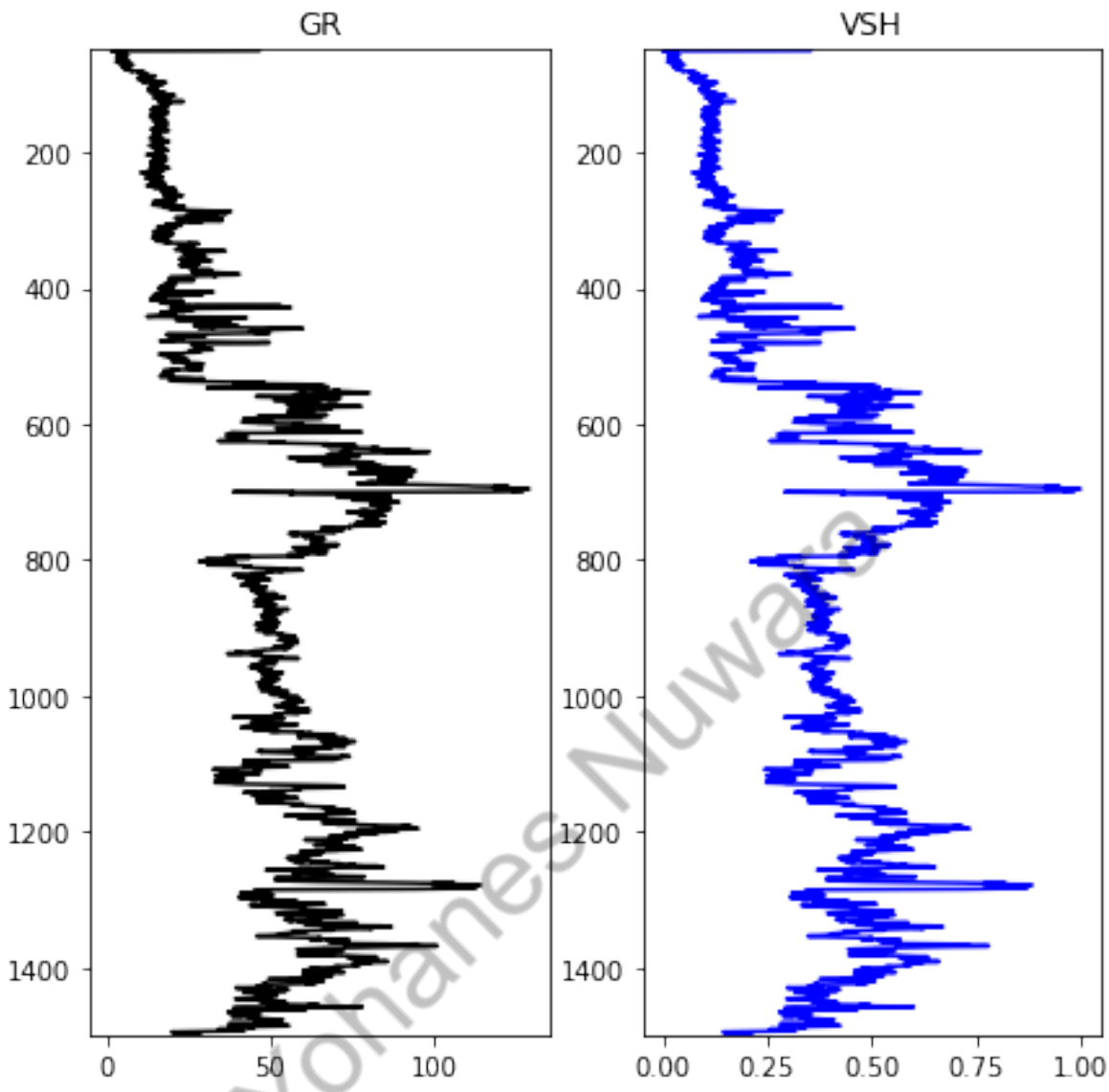
```
(1.3392, 129.1773)
```

Now we can produce VSH log using the above formula

```
gr_min = np.min(gr_imputed)
gr_max = np.max(gr_imputed)

vsh = (gr_imputed - gr_min) / (gr_max - gr_min)
```

Let's display the created VSH log

```
# display the converted log
plt.figure(figsize=(7,7))

plt.subplot(1,2,1)
plt.plot(gr, depth, color='black')
plt.ylim(max(depth), min(depth))
plt.title('GR')

plt.subplot(1,2,2)
plt.plot(vsh, depth, color='blue')
plt.ylim(max(depth), min(depth))
plt.title('VSH')

plt.show()
```



Produce DPHI (Density Porosity) from RHOB

Here we assume that the entire log represents the same lithology composed of sandstone matrix and filled by brine. Calculate DPHI from RHOB using this formula:

$$DPHI = \frac{\rho_{ma} - RHOB}{\rho_{ma} - \rho_f}$$

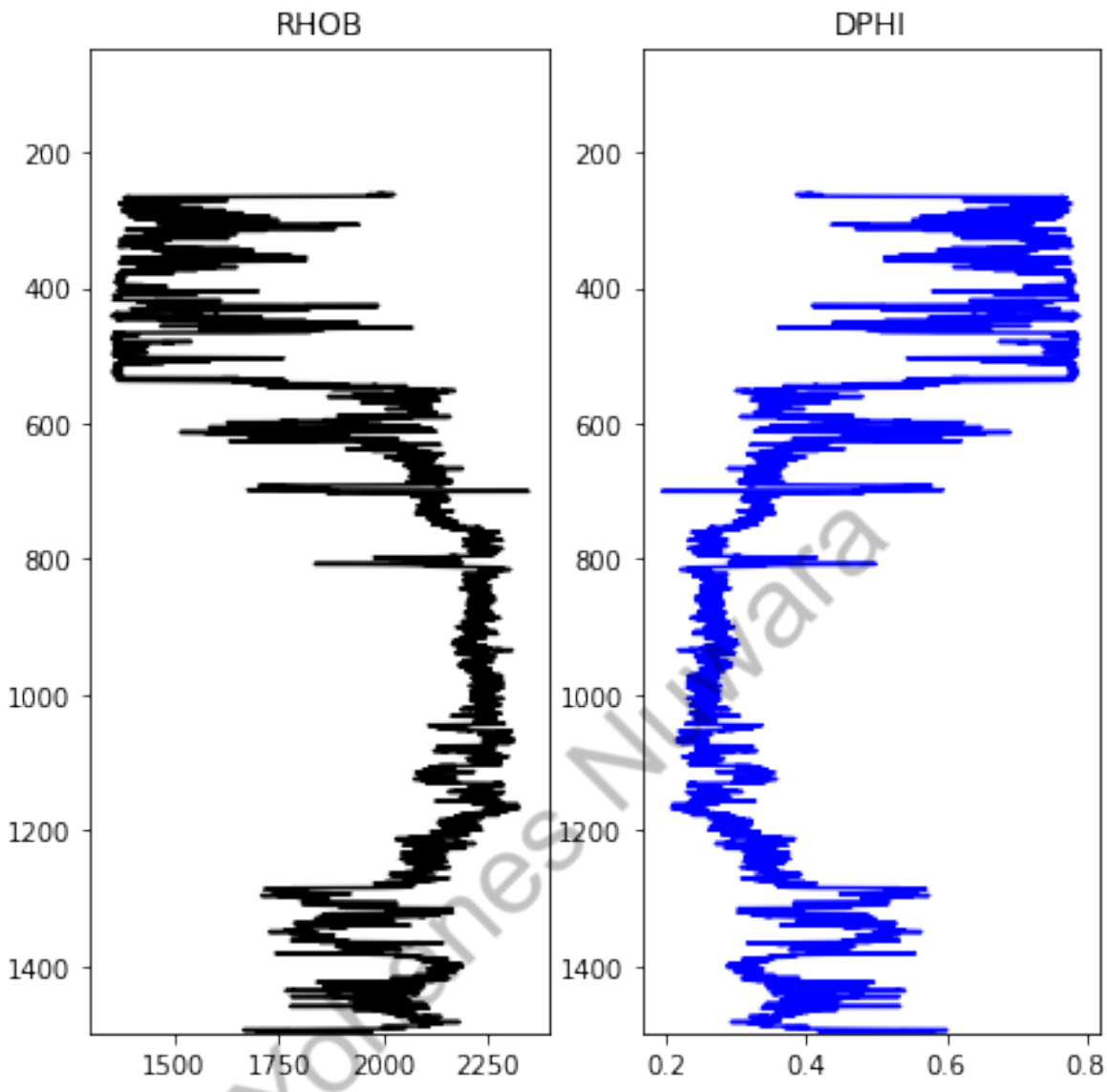
Hence, matrix density is assumed 2.67 g/cc and fluid density is assumed 1 g/cc. In addition, we should convert our RHOB log unit from kg/m3 to g/cc.

```
denma = 2.67 # g/cc
rhof = 1

# convert RHOB from kg/m3 to g/cc unit
```

```
rhob_converted = rhob * 0.001

dphi = (denma - rhob_converted) / (denma - rhof)
```

Let's display the created DPHI log.

```
# display the converted log
plt.figure(figsize=(7,7))

plt.subplot(1,2,1)
plt.plot(rhob, depth, color='black')
plt.ylim(max(depth), min(depth))
plt.title('RHOB')

plt.subplot(1,2,2)
plt.plot(dphi, depth, color='blue')
plt.ylim(max(depth), min(depth))
plt.title('DPHI')

plt.show()
```

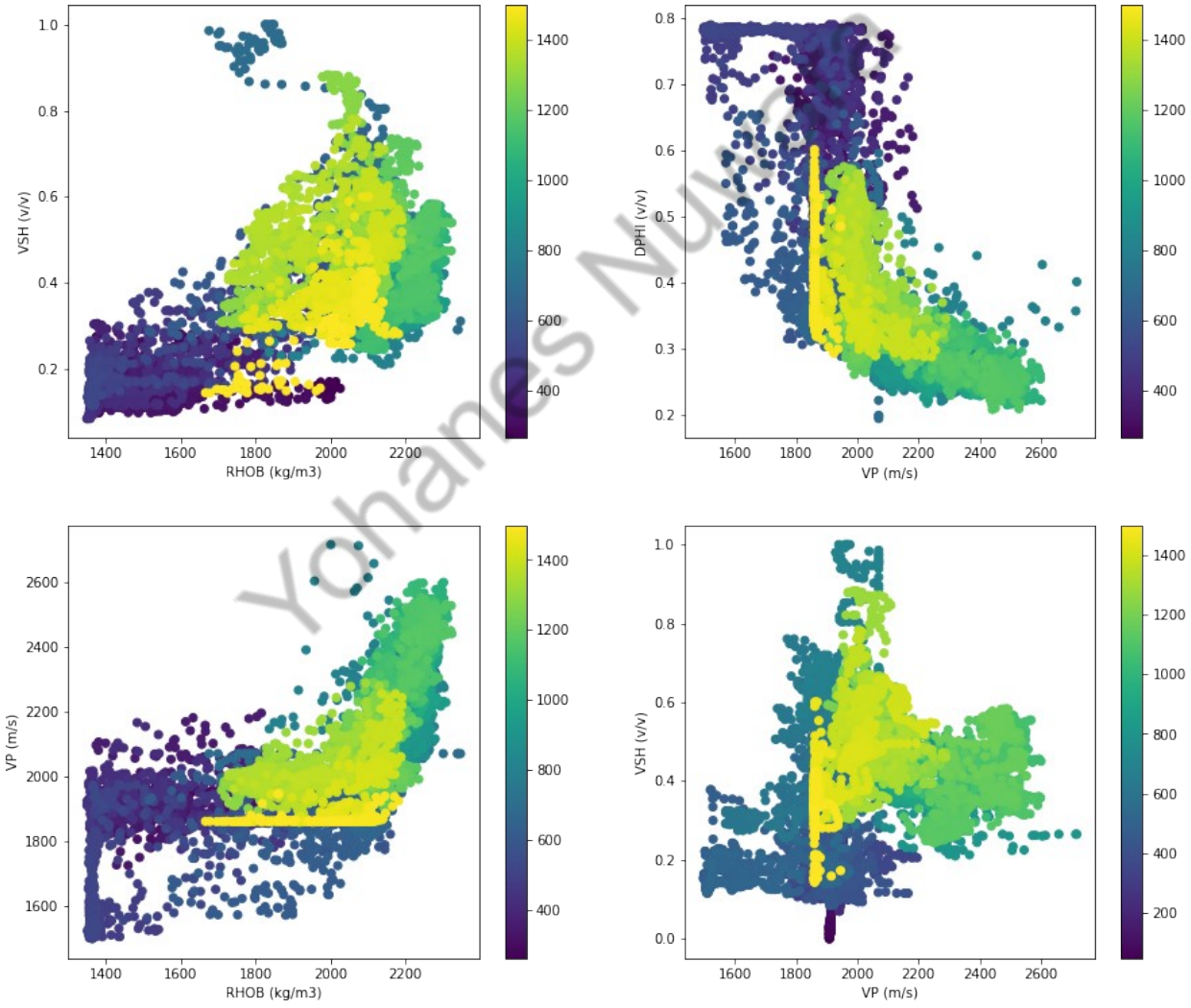


Look at the above DPHI log. The upper and lower part of the log has porosity higher than 45%. Still remember that porosity never exceed 45% (ideal porosity)? Therefore, our assumption can't be used for that zones, and hence, lithological and fluid correction must be done (beyond the scope of this training).

Multiple Crossplot of Logs

```
plt.figure(figsize=(15,13))

plt.subplot(2,2,1)
plt.scatter(rhob, vsh, c=depth)
plt.xlabel('RHOB (kg/m3)'); plt.ylabel('VSH (v/v)')
plt.colorbar()

plt.subplot(2,2,2)
plt.scatter(vp, dphi, c=depth)
```

```
plt.xlabel('VP (m/s)'); plt.ylabel('DPHI (v/v)')
plt.colorbar()

plt.subplot(2,2,3)
plt.scatter(rhob, vp, c=depth)
plt.xlabel('RHOB (kg/m3)'); plt.ylabel('VP (m/s)')
plt.colorbar()

plt.subplot(2,2,4)
plt.scatter(vp, vsh, c=depth)
plt.xlabel('VP (m/s)'); plt.ylabel('VSH (v/v)')
plt.colorbar()

plt.show()
```



Optional: Pairplots

Pairplot is a powerful technique to look at how the log data is **distributed** and **correlated**. We use `seaborn` library to do so.

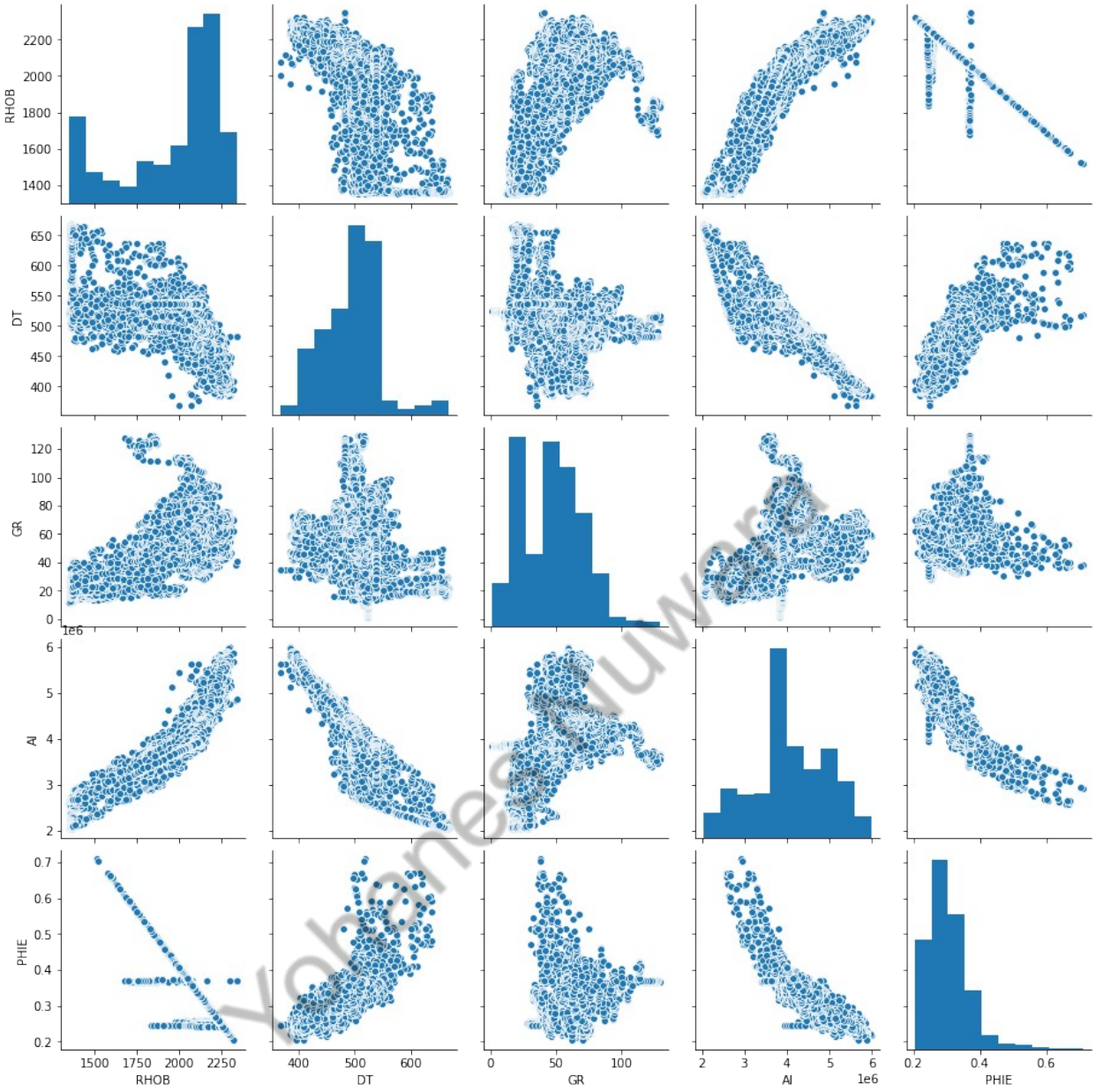
Because Google Colab already had `seaborn`, we can directly import it.

```
import seaborn as sns

well_df = pd.DataFrame({'RHOB': rhob, 'DT': dt, 'GR': gr, 'AI': ai,
'PHIE': phie})
depth_df = pd.DataFrame({'DEPTH': depth})
well_df
```

	RHOB	DT	GR	AI	PHIE
0	NaN	524.0485	46.613531	3820257.00	NaN
1	NaN	524.0410	46.613531	3820311.75	NaN
2	NaN	524.0336	46.613531	3820366.25	NaN
3	NaN	524.0262	46.613531	3820419.75	NaN
4	NaN	524.0190	46.613531	3820472.25	NaN
...	...	...	...	...	...
9675	1787.5444	532.1727	27.627300	3359011.00	NaN
9676	1814.1691	534.8610	28.528100	3391772.00	NaN
9677	1861.6497	537.1902	28.581300	3465524.00	NaN
9678	1880.3978	537.2630	28.581300	3499957.75	NaN
9679	1856.3350	537.3831	27.975900	3454404.00	NaN

[9680 rows x 5 columns]

```
sns.pairplot(well_df)
plt.show()
```



Formation Evaluation with Python

```
# import numpy, matplotlib, and pandas
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# install and import lasio
!pip install lasio
import lasio
```

```
Collecting lasio
  Downloading
https://files.pythonhosted.org/packages/5e/8e/ce58a22ec8454a12f92333a5
0f2add5f6131218c4815952d6ca7cbd578f0/lasio-0.28-py3-none-any.whl
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-
packages (from lasio) (1.18.5)
Installing collected packages: lasio
Successfully installed lasio-0.28
```

```
# cloning github repository of formation-evaluation
!git clone https://github.com/yohanesnuwara/formation-evaluation
```

```
Cloning into 'formation-evaluation'...
remote: Enumerating objects: 102, done.ote: Counting objects: 100%
(102/102), done.ote: Compressing objects: 100% (95/95), done.ote:
Total 102 (delta 43), reused 0 (delta 0), pack-reused 0
```

```
# import all functions from formation-evaluation
import sys
sys.path.append('/content/formation-evaluation')

from well_log_display import well_log_display
from triple_combo import triple_combo
from ND_plot import ND_plot
from label_generator import label_generator
```


Functions

```
def calculate_klogh(formation_name, phif, vsh):
  # there is no available equation for Heather Fm.
  if formation_name == 'hugin':
    return 10 ** (2 + (8 * phif) - (9 * vsh))
  if formation_name == 'sleipner':
    return 10 ** (-3 + (32 * phif) - (2 * vsh))
  if formation_name == 'skagerak':
    return 10 ** (-1.85 + (17.4 * phif) - (3 * vsh))
```

Load well log data

```
# specify file path (well 15/9-F-11A)
filepath = '/content/formation-evaluation/data/volve/15_9-F-11A.LAS'

# read with lasio
well = lasio.read(filepath)

# check the available logs with .keys()
well.keys()
```

```
['DEPTH',
 'ABDCQF01',
 'ABDCQF02',
 'ABDCQF03',
 'ABDCQF04',
 'BS',
 'CALI',
 'DRHO',
 'DT',
 'DTS',
 'GR',
 'NPHI',
 'PEF',
 'RACEHM',
 'RACELM',
 'RD',
 'RHOB',
 'RM',
 'ROP',
 'RPCEHM',
 'RPCELM',
 'RT']
```



```
# look for more detail with .curves
well.curves
```

```
[CurveItem(mnemonic="DEPTH", unit="M", value="00 001 00 00", descr="0
Depth", original_mnemonic="DEPTH", data.shape=(35735,)),
 CurveItem(mnemonic="ABDCQF01", unit="g/cm3", value="00 000 00 00:   1
ABDCQF01:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="ABDCQF01",
data.shape=(35735,)),
 CurveItem(mnemonic="ABDCQF02", unit="g/cm3", value="00 000 00 00:   2
ABDCQF02:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="ABDCQF02",
data.shape=(35735,)),
 CurveItem(mnemonic="ABDCQF03", unit="g/cm3", value="00 000 00 00:   3
ABDCQF03:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="ABDCQF03",
data.shape=(35735,)),
 CurveItem(mnemonic="ABDCQF04", unit="g/cm3", value="00 000 00 00:   4
ABDCQF04:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="ABDCQF04",
data.shape=(35735,)),
```

```
 CurveItem(mnemonic="BS", unit="inches", value="00 000 00 00:   5
BS:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="BS",
data.shape=(35735,)),
 CurveItem(mnemonic="CALI", unit="inches", value="70 280 00 01:   6
CALI:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="CALI",
data.shape=(35735,)),
 CurveItem(mnemonic="DRHO", unit="g/cm3", value="45 356 01 01:   7
DRHO:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="DRHO",
data.shape=(35735,)),
 CurveItem(mnemonic="DT", unit="us/ft", value="60 520 32 01:   8
DT:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="DT",
data.shape=(35735,)),
 CurveItem(mnemonic="DTS", unit="us/ft", value="00 000 00 00:   9
DTS:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="DTS",
data.shape=(35735,)),
 CurveItem(mnemonic="GR", unit="API", value="30 150 01 01:  10
GR:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="GR",
data.shape=(35735,)),
 CurveItem(mnemonic="NPHI", unit="v/v", value="42 890 01 01:  11
NPHI:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="NPHI",
data.shape=(35735,)),
 CurveItem(mnemonic="PEF", unit="b/elec", value="45 358 01 01:  12
PEF:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="PEF",
data.shape=(35735,)),
 CurveItem(mnemonic="RACEHM", unit="ohm.m", value="95 220 00 01:  13
RACEHM:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="RACEHM",
data.shape=(35735,)),
 CurveItem(mnemonic="RACELM", unit="ohm.m", value="00 524 01 00:  14
RACELM:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="RACELM",
data.shape=(35735,)),
 CurveItem(mnemonic="RD", unit="ohm.m", value="00 000 00 00:  15
RD:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="RD",
data.shape=(35735,)),
 CurveItem(mnemonic="RHOB", unit="g/cm3", value="45 350 01 01:  16
RHOB:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="RHOB",
data.shape=(35735,)),
 CurveItem(mnemonic="RM", unit="ohm.m", value="00 000 00 00:  17
RM:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="RM",
data.shape=(35735,)),
 CurveItem(mnemonic="ROP", unit="m/hr", value="00 000 00 00:  18
ROP:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="ROP",
data.shape=(35735,)),
 CurveItem(mnemonic="RPCEHM", unit="ohm.m", value="00 000 00 00:  19
RPCEHM:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="RPCEHM",
data.shape=(35735,)),
 CurveItem(mnemonic="RPCELM", unit="ohm.m", value="00 000 00 00:  20
RPCELM:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="RPCELM",
data.shape=(35735,)),
 CurveItem(mnemonic="RT", unit="ohm.m", value="00 000 00 00:  21
```

```
RT:COMPOSITE:rC:NONE", descr="v1", original_mnemonic="RT",
data.shape=(35735,))]

# see the data using .data
well.data

array([[ 188.5,    nan,    nan, ...,    nan,    nan,    nan],
       [ 188.6,    nan,    nan, ...,    nan,    nan,    nan],
       [ 188.7,    nan,    nan, ...,    nan,    nan,    nan],
       ...,
       [3761.7,    nan,    nan, ...,    nan,    nan,    nan],
       [3761.8,    nan,    nan, ...,    nan,    nan,    nan],
       [3761.9,    nan,    nan, ...,    nan,    nan,    nan]])

# convert it to Pandas dataframe using .df() and then reset index
well = well.df().reset_index()

# then show the dataframe
well.head(10)
```

	DEPTH	ABDCQF01	ABDCQF02	ABDCQF03	ABDCQF04	...	RM	ROP	RPCEHM	RPCELM	RT
0	188.5	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
1	188.6	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
2	188.7	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
3	188.8	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
4	188.9	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
5	189.0	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
6	189.1	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
7	189.2	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
8	189.3	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
9	189.4	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN

```
[10 rows x 22 columns]
```


Generate formation labels from formation tops file

We have formation tops file. What we are going to do now is to give formation top labels to the logs. We'll use function `label_generator`.

```
# specify file path
filepath = '/content/formation-evaluation/data/volve/Fmtops_15_9-F-
11A.csv'

# read with Pandas
tops = pd.read_csv(filepath)

tops
```

	PICKS	DEPTH
0	HORDALAND GP. Top	2600.0
1	Ty Fm. Top	2624.0
2	Ekofisk Fm. Top	2794.5
3	Hod Fm. Top	3118.0
4	Draupne Fm. Top	3525.8
5	Heather Fm. Top	3574.9
6	Heather Fm. Sand VOLVE Top	3585.2
7	Hugin Fm. VOLVE Top	3594.6
8	Sleipner Fm. Top	3702.0

```
# see help to find what visualization inputs are required
help(label_generator)

Help on function label_generator in module label_generator:

label_generator(df_well, df_tops, column_depth, label_name)
    Generate Formation (or other) Labels to Well Dataframe
    (useful for machine learning and EDA purpose)

    Input:

    df_well is your well dataframe (that originally doesn't have the
intended label)
    df_tops is your label dataframe (this dataframe should ONLY have 2
columns)
      1st column is the label name (e.g. formation top names)
      2nd column is the depth of each label name

    column_depth is the name of depth column on your df_well dataframe
    label_name is the name of label that you want to produce (e.g. FM.
LABEL)

    Output:

    df_well is your dataframe that now has the labels (e.g. FM. LABEL)


# use "label_generator" to generate formation labels
# then show the dataframe
well = label_generator(well, tops, 'DEPTH', 'FM. LABEL')
```

```
well
```

```
        DEPTH  ABDCQF01  ABDCQF02  ...  RPCELM    RT          FM. LABEL
0       188.5       NaN       NaN  ...     NaN   NaN                NaN
1       188.6       NaN       NaN  ...     NaN   NaN                NaN
2       188.7       NaN       NaN  ...     NaN   NaN                NaN
3       188.8       NaN       NaN  ...     NaN   NaN                NaN
4       188.9       NaN       NaN  ...     NaN   NaN                NaN
...       ...       ...       ...  ...     ...   ...                ...
35730  3761.5     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top
35731  3761.6     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top
35732  3761.7     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top
35733  3761.8     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top
35734  3761.9     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top

[35735 rows x 23 columns]
```

As we see here we have NaN in the `FM. LABEL` column simply because the depths are not corresponding to any Fm. tops. So we can replace NaNs with a string named as `Unknown`

```
# Replace NaNs in formation label column with string called "Unknown"
well['FM. LABEL'] = well['FM. LABEL'].fillna('Unknown')

well
```

```
        DEPTH  ABDCQF01  ABDCQF02  ...  RPCELM    RT          FM. LABEL
0       188.5       NaN       NaN  ...     NaN   NaN            Unknown
1       188.6       NaN       NaN  ...     NaN   NaN            Unknown
2       188.7       NaN       NaN  ...     NaN   NaN            Unknown
3       188.8       NaN       NaN  ...     NaN   NaN            Unknown
4       188.9       NaN       NaN  ...     NaN   NaN            Unknown
...       ...       ...       ...  ...     ...   ...                ...
35730  3761.5     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top
35731  3761.6     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top
35732  3761.7     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top
35733  3761.8     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top
35734  3761.9     2.306     2.303  ...   0.737  0.77  Sleipner Fm. Top

[35735 rows x 23 columns]
```


Visualize

Well-log visualization

We will use function `well_log_display`

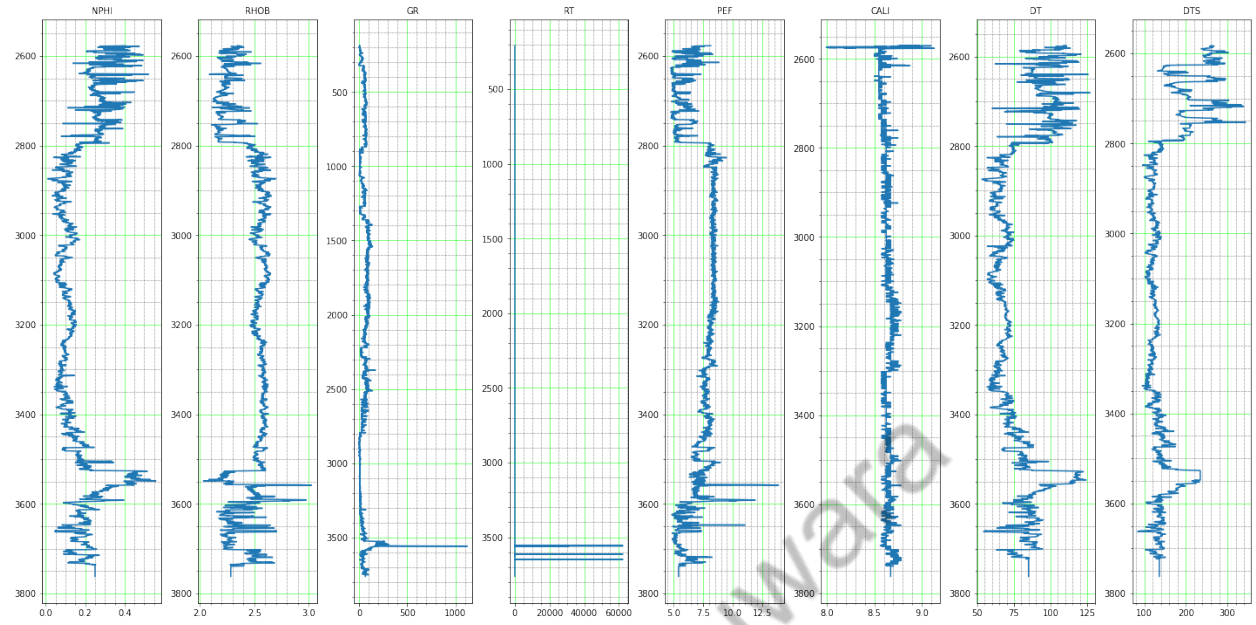
```
# see help to find what visualization inputs are required
# identify which input is OBLIGATORY, which input is OPTIONAL
help(well_log_display)
```

```
Help on function well_log_display in module well_log_display:

well_log_display(df, column_depth, column_list, column_semilog=None,
min_depth=None, max_depth=None, column_min=None, column_max=None,
colors=None, fm_tops=None, fm_depths=None, tight_layout=1,
title_size=10)
    Display log side-by-side style

    Input:

    df is your dataframe
    specify min_depth and max_depth as the upper and lower depth limit
    column_depth is the column name of your depth
    column_list is the LIST of column names that you will display

    column_semilog is specific for resistivity column; if your
resistivity is
      in column 3, specify as: column_semilog=2. Default is None, so
if
      you don't specify, the resistivity will be plotted in normal
axis instead

    column_min is list of minimum values for the x-axes.
    column_max is list of maximum values for the x-axes.

    colors is the list of colors specified for each log names. Default
is None,
      so if don't specify, the colors will be Matplotlib default
(blue)

    fm_tops and fm_depths are the list of formation top names and
depths.
      Default is None, so no tops are shown. Specify both lists, if
you want
      to show the tops
```

Use default

```
# Logs we gonna visualize are: 'NPHI', 'RHOB', 'GR', 'RT', 'PEF',
'CALI', 'DT', 'DTS'
df_well = well
column_depth = 'DEPTH'
column_list = ['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT', 'DTS']
```

```
well_log_display(df_well, column_depth, column_list)
```



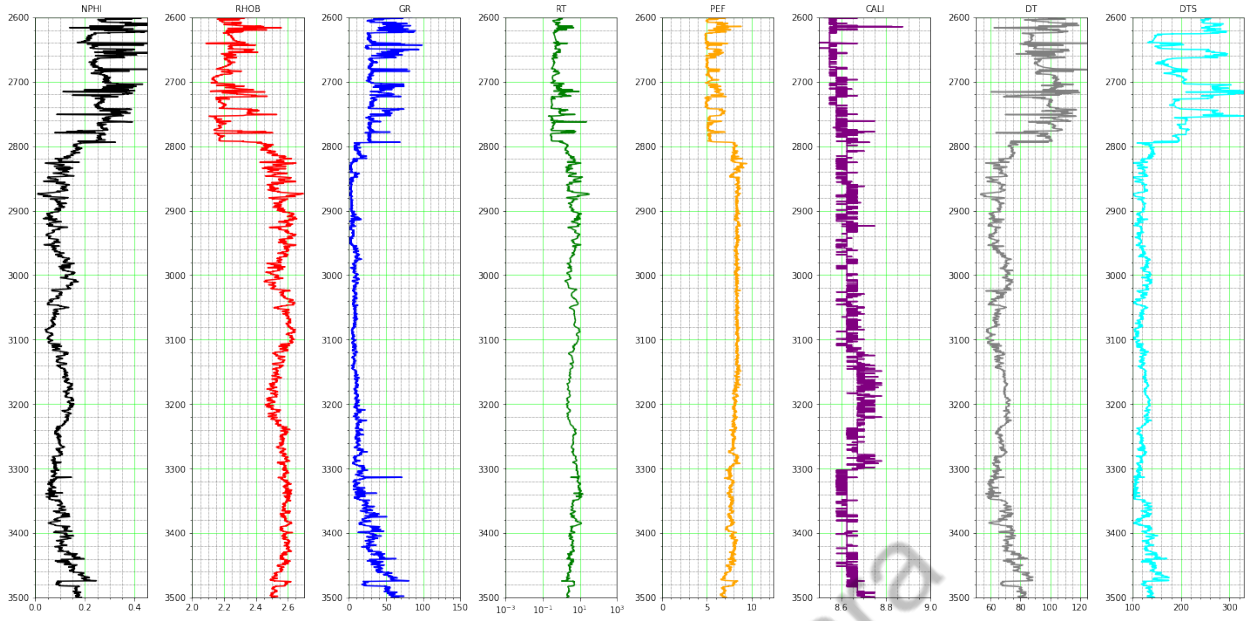
Make the visualization looks better (adding the optional variables)

- Transform the resistivity axis from Cartesian axis to Semilog axis
- Give limits to the x and depth axes (2,600 to 3,500 m)
- Give different colors to each log

```
column_list
```

```
['NPHI', 'RHOB', 'GR', 'RT', 'PEF', 'CALI', 'DT', 'DTS']
```

```
# adding optional variables like "min_depth", "max_depth", etc.
column_semilog=3
min_depth=2600
max_depth=3500
column_min=[0, 2, 0, 0.001, 0, 8.5, 50, 100]
column_max=[0.45, 2.7, 150, 1000, 12.5, 9, 125, 330]
colors=['black', 'red', 'blue', 'green', 'orange', 'purple', 'gray',
'cyan']

well_log_display(df_well, column_depth, column_list,
                 column_semilog, min_depth, max_depth,
                 column_min, column_max, colors)
```



Triple combo

We are seeking hydrocarbon occurence in depth between 3,580 to 3,702 m. So, we'll visualize the crossover in a triple combo. We will use function `triple_combo`

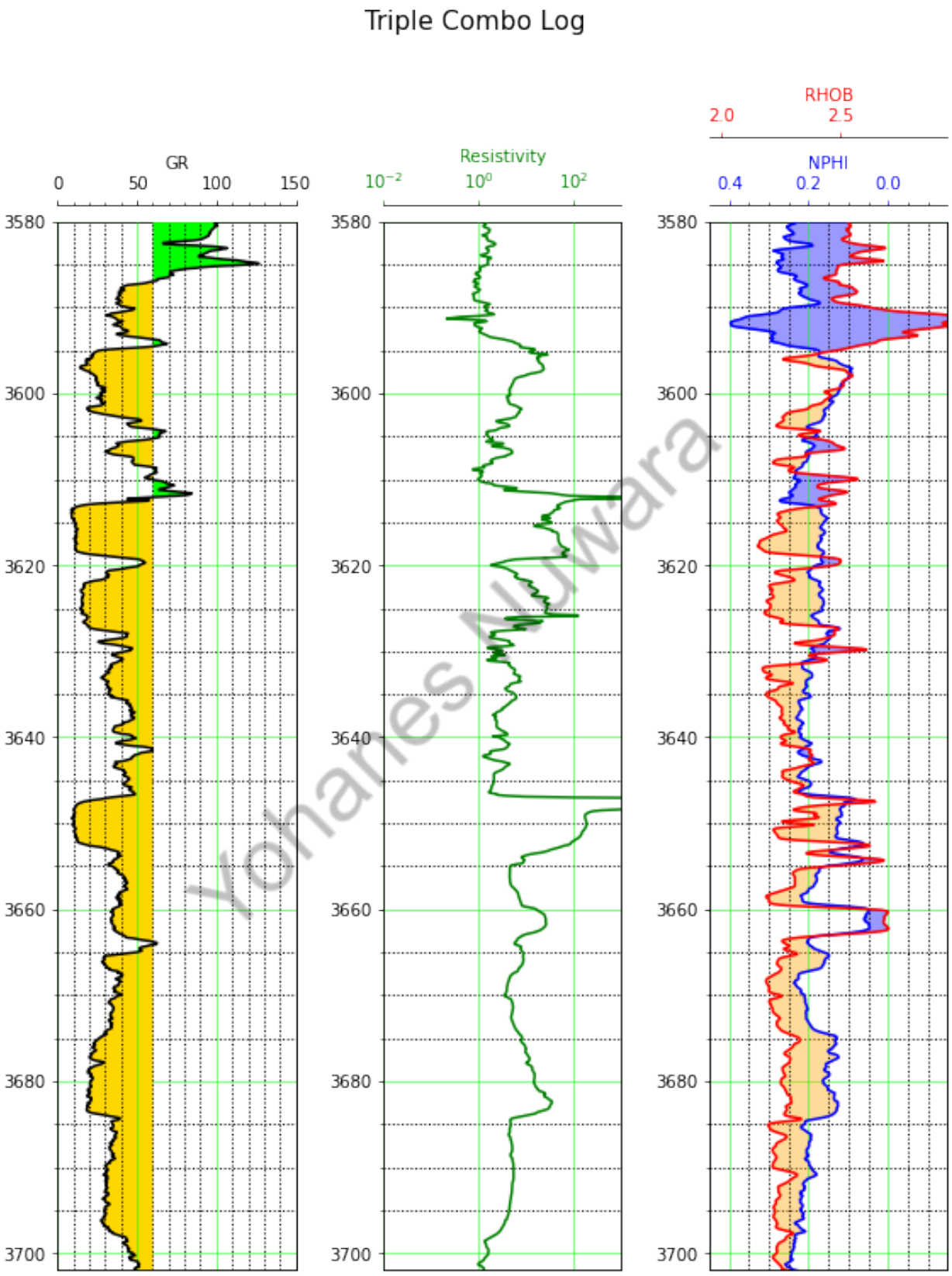
```
# see help to find what visualization inputs are required
# identify which input is OBLIGATORY, which input is OPTIONAL
help(triple_combo)
```

```
Help on function triple_combo in module triple_combo:

triple_combo(df, column_depth, column_GR, column_resistivity,
column_NPHI, column_RHOB, min_depth, max_depth, min_GR=0, max_GR=150,
sand_GR_line=60, min_resistivity=0.01, max_resistivity=1000,
color_GR='black', color_resistivity='green', color_RHOB='red',
color_NPHI='blue', figsize=(6, 10), tight_layout=1, title_size=15,
title_height=1.05)
    Producing Triple Combo log

    Input:

    df is your dataframe
    column_depth, column_GR, column_resistivity, column_NPHI,
column_RHOB
    are column names that appear in your dataframe (originally from
the LAS file)

    specify your depth limits; min_depth and max_depth

    input variables other than above are default. You can specify
    the values yourselves.
```

```
    Output:

    Fill colors; gold (sand), lime green (non-sand), blue (water-
zone), orange (HC-zone)
```

Use default

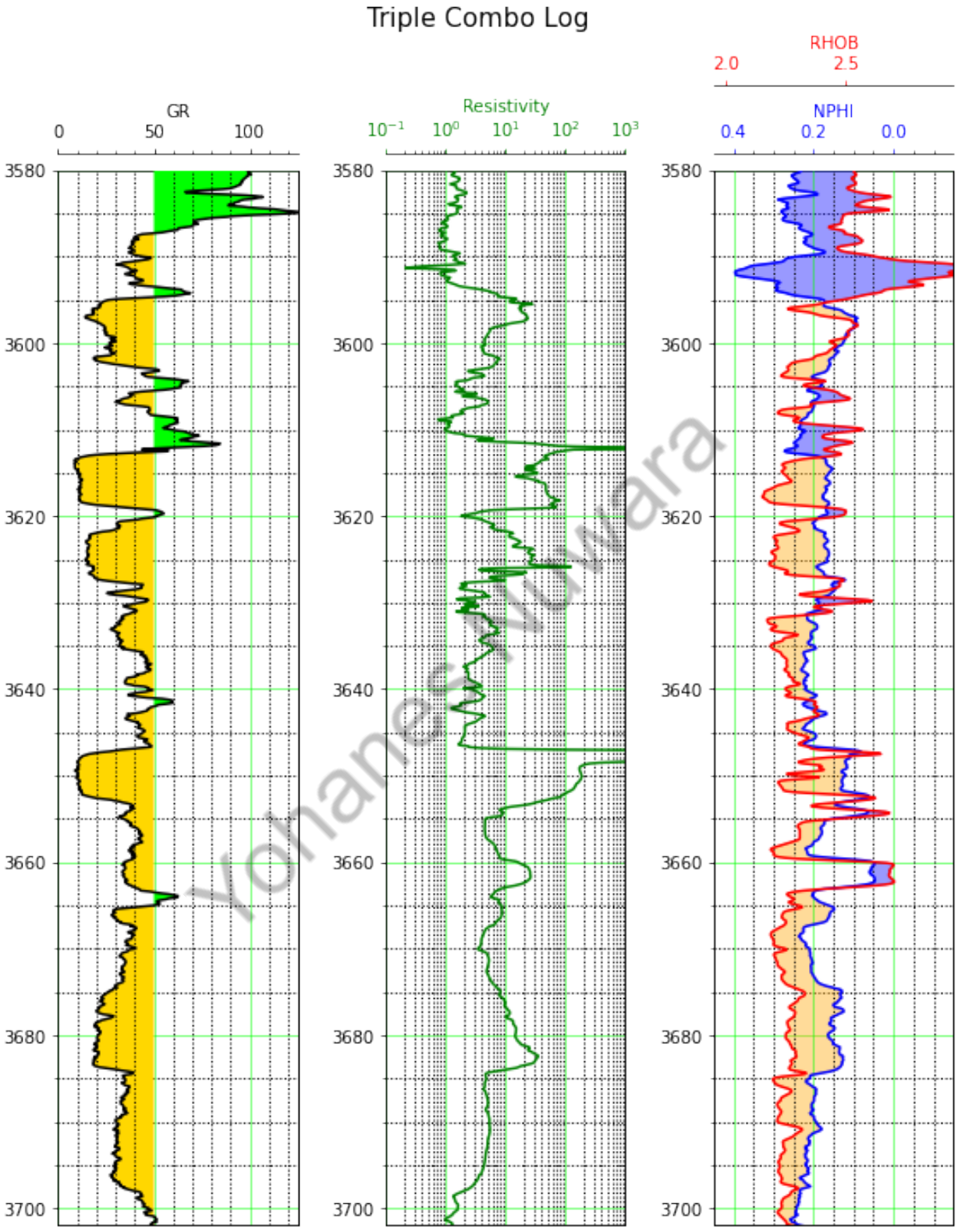
```
# triple combo at depth from 3,580 to 3,702 m
df = well
column_depth = 'DEPTH'
column_GR = 'GR'
column_resistivity = 'RT'
column_RHOB = 'RHOB'
column_NPHI = 'NPHI'
min_depth = 3580
max_depth = 3702

triple_combo(df, column_depth, column_GR, column_resistivity,
             column_NPHI, column_RHOB, min_depth, max_depth)
```



Make the visualization looks better (adding the optional variables)

- Give limits to the x axes (specifically, GR and resistivity)
- Give a sand GR line, e.g. 50 API
- Adjust figure size and title height (if needed)

```
# adding optional variables like "sand_GR_line", "figsize", etc.
min_GR=0
max_GR=125
sand_GR_line=50
min_resistivity=0.1
max_resistivity=1000
title_height=1.01

triple_combo(df, column_depth, column_GR, column_resistivity,
             column_NPHI, column_RHOB, min_depth, max_depth,
             min_GR, max_GR, sand_GR_line, min_resistivity,
             max_resistivity, title_height=title_height)
```


Triple Combo Log
GR
0
50
100
Resistivity
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
RHOB
2.0
2.5
NPHI
0.4
0.2
0.0
3580
3600
3620
3640
3660
3680
3700
Yohanes Nuwara

Neutron Density Plot

We wish to identify the lithologies of each formation (whether sandstone, limestone, or dolomite) using ND plot. We will use function `ND_plot`.

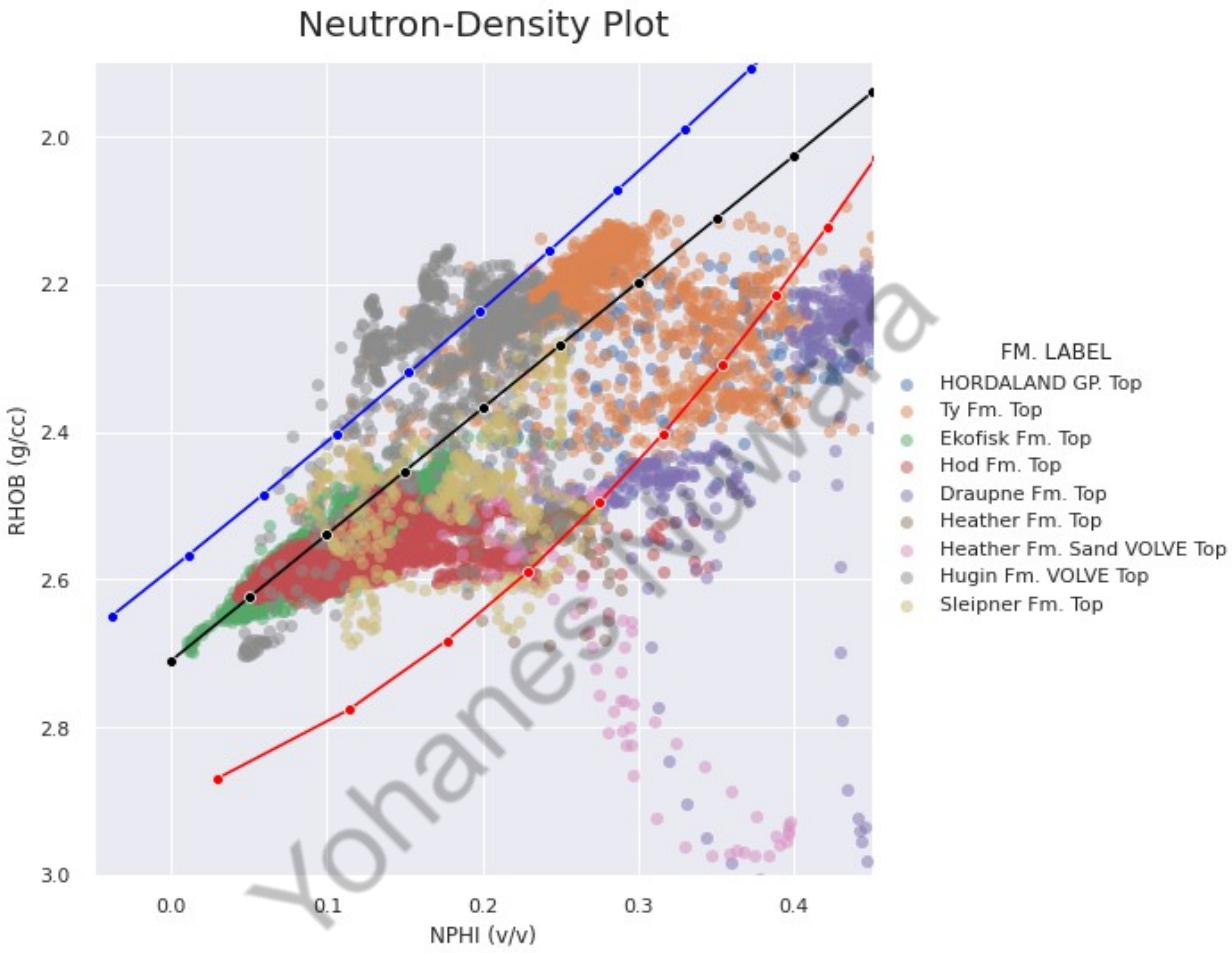
```
# see help to find what visualization inputs are required
# identify which input is OBLIGATORY, which input is OPTIONAL
help(ND_plot)
```

```
Help on function ND_plot in module ND_plot:

ND_plot(denfl, df, column_nphi, column_rhob, column_hue, color_by,
figsize=(7, 7), scatter_size=50, scatter_alpha=0.5)
    Producing Neutron-Density (Cross)plot

    Input:

    denfl is your fluid density
    df is your dataframe
    column_nphi and column_rhob are the column name of your NPHI and
RHOB
    column_hue is the column name that you want for the color of the
points
      e.g. depth, vshale, formation labels, etc.

    color_by depends on the column_hue that you're giving
      * if you're giving a continuous hue (numerical) like depth or
vshale
        define color_by='continuous'
      * if you're giving a categorical hue (labels) like formation
names
        define color_by='categorical'

    figsize, scatter_size, scatter_alpha are by default. You can also
specify
      by yourselves.

    Output:

    3 lines. Blue is sandstone, black is limestone, red is dolomite
    Each line has dots representing porosity value from 0 to 0.5
      by increment of 0.05
```


Plot the NPHI and RHOB points differentiated by each formation names. (categorical)

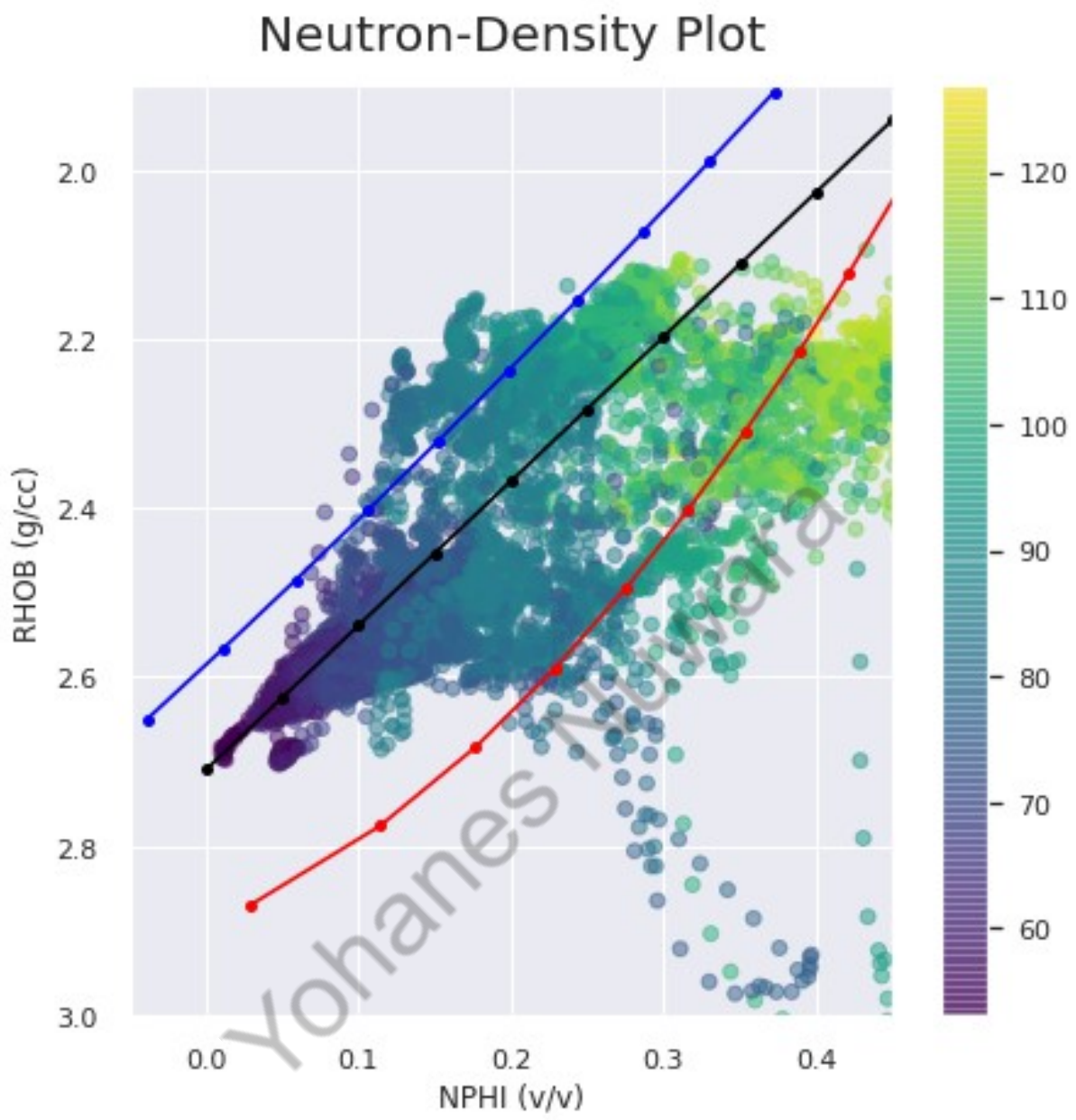
```
# use "ND_plot" to make a plot with colors based on formation names
denfl = 1
df = well
```

```
column_nphi = 'NPHI'
column_rhob = 'RHOB'
column_hue = 'FM. LABEL'
color_by = 'categorical'

ND_plot(denfl, df, column_nphi, column_rhob, column_hue, color_by)
```



Plot NPHI and RHOB points differentiated by depths/other continuous variables

```
# use "ND_plot" to make a plot with colors based on continuous
variables
column_hue = 'DT'
color_by = 'continuous'

ND_plot(denfl, df, column_nphi, column_rhob, column_hue, color_by)
```



Petrophysical Calculation per Formation Top

You already have the data with formation labels. Next we will calculate new petrophysical variables PER FORMATION LABEL (only Heather and Hugin).

Sleipner will be your assignment.

First thing we need to do is to "mask the subset of dataframe in FM. LABEL column that contains each of the above names".

```
# list all formation labels using df['x'].unique()
well['FM. LABEL'].unique()
```

```
array(['Unknown', 'HORDALAND GP. Top', 'Ty Fm. Top', 'Ekofisk Fm.
Top',
       'Hod Fm. Top', 'Draupne Fm. Top', 'Heather Fm. Top',
       'Heather Fm. Sand VOLVE Top', 'Hugin Fm. VOLVE Top',
       'Sleipner Fm. Top'], dtype=object)
```

```
# create masks for Heather and Hugin using string contains:
str.contains('...')
mask_heather = well['FM. LABEL'].str.contains('Heather')
mask_hugin = well['FM. LABEL'].str.contains('Hugin')

# see inside the mask
mask_heather
```

```
0        False
1        False
2        False
3        False
4        False
         ...
35730    False
35731    False
35732    False
35733    False
35734    False
Name: FM. LABEL, Length: 35735, dtype: bool
```

Now isolate Heather and Hugin into individual subsets of Dataframe, using the masks that we made before.

```
# create dataframe subsets of Heather and Hugin: df[mask]
# heather = well[mask_heather]
heather = well[mask_heather]
hugin = well[mask_hugin]

# print the Heather dataframe
heather.head(10)
```

	DEPTH	ABDCQF01	ABDCQF02	...	RPCELM	RT	FM. LABEL
33864	3574.9	2.555	2.519	...	1.743	1.881	Heather Fm. Top
33865	3575.0	2.585	2.508	...	1.778	1.960	Heather Fm. Top
33866	3575.1	2.585	2.504	...	1.773	1.914	Heather Fm. Top
33867	3575.2	2.562	2.504	...	1.700	1.775	Heather Fm. Top
33868	3575.3	2.532	2.505	...	1.654	1.766	Heather Fm. Top
33869	3575.4	2.503	2.507	...	1.640	1.750	Heather Fm. Top
33870	3575.5	2.486	2.513	...	1.638	1.745	Heather Fm. Top
33871	3575.6	2.484	2.520	...	1.636	1.741	Heather Fm. Top
33872	3575.7	2.491	2.526	...	1.642	1.768	Heather Fm. Top
33873	3575.8	2.495	2.528	...	1.646	1.783	Heather Fm. Top

```
[10 rows x 23 columns]
```

We calculate new petrophysical outputs on these subsets. Based on petrophysical report, each formation has different variables like fluid density (ρ_{fl}), etc. The variables have been already listed below.

```
# List of individual variables of each formation (DO NOT CHANGE THESE
VALUES)
#            Heath  Hugin   Sleip
rho_ma     = [2.66,  2.65,   2.65]
rho_fl     = [1,     0.9,    0.9]
gr_min     = [7,     7,      7]
gr_max     = [120,   150,    105]
A          = [0,     0.4,    0.4]  # regression coeff for PHID
B          = [0,     0.01,   0.01] # regression coeff for PHID

# from report, m of Hugin and Sleipner is calculated using specific
formula (p. 15, 16)
# now we just assume both has the following values (taken from
Asquith, p. 5)
# Hugin is "estimated" as consolidated sst, Sleipner as carbonates
m          = [2,     2,      2.14]
a          = [1,     1,      1]
n          = [2,     2.45,   2.45]
# water resistivity, already extrapolated at depth from Rw @ 20C =
0.07 ohm-m
Rw         = [0.022, 0.022,  0.022]
```

1. Calculate density porosity (PHID)

$$\phi_F = \phi_D + A \times (NPHI - \phi_D) + B$$

where:

$$\phi_D = \frac{\rho_{ma} - \rho_b}{\rho_{ma} - \rho_{fl}} \text{ [fraction]}$$

ρ_{ma} is the matrix density [g/cm^3]
ρ_b is the measured bulk density (RHOB), [g/cm^3]
ρ_{fl} is the pore fluid density [g/cm^3].

A and B are regression coefficients.

NPHI: Neutron log in limestone units [fraction]

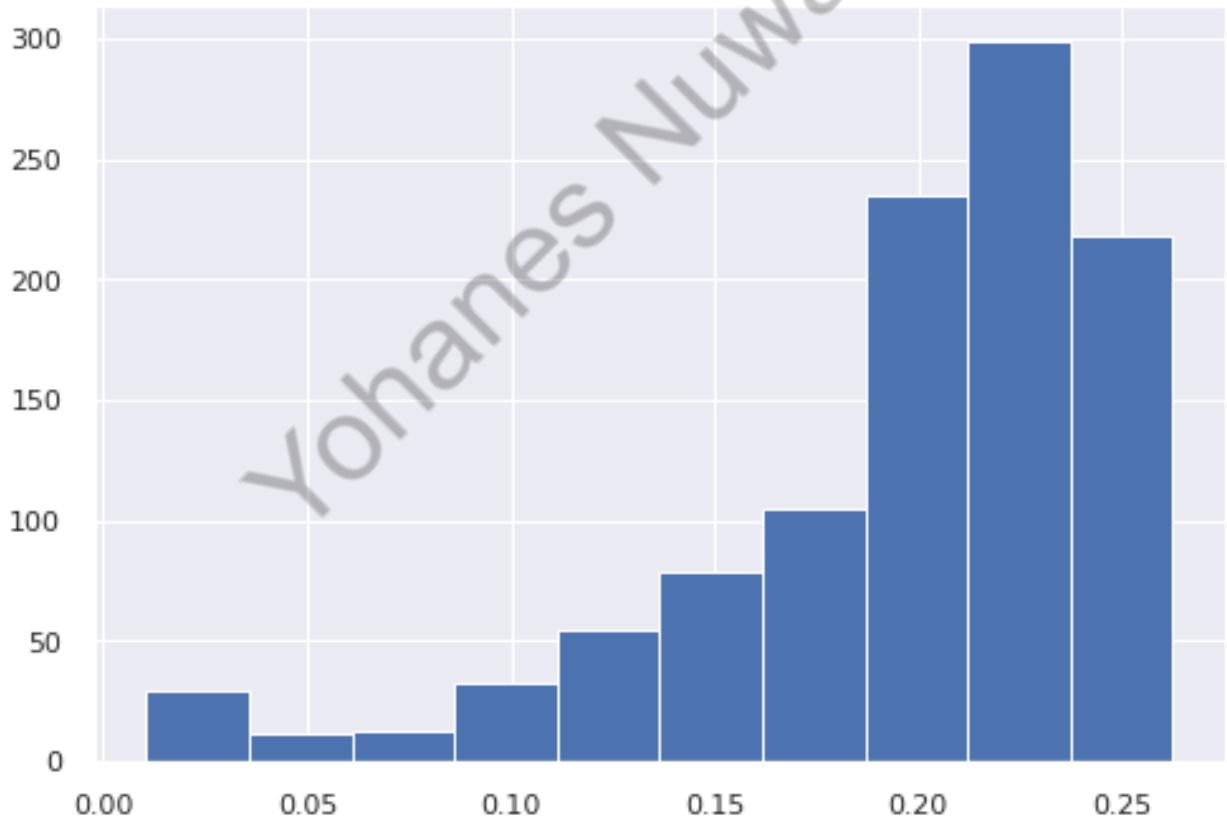

```
# calculate PHID for Heather and Hugin
rhob_heather, rhob_hugin = heather['RHOB'], hugin['RHOB']

phid_heather = (rho_ma[0] - rhob_heather) / (rho_ma[0] - rho_fl[0])
phid_hugin = (rho_ma[1] - rhob_hugin) / (rho_ma[1] - rho_fl[1])

phid_hugin
```

```
34061    0.055429
34062    0.064000
34063    0.073714
34064    0.083429
34065    0.095429
           ...
35130    0.228571
35131    0.227429
35132    0.226286
35133    0.226286
```

```
35134    0.223429
Name: RHOB, Length: 1074, dtype: float64
```

2. Calculate total porosity (PHIF)

```
# calculate PHIF for Heather and Hugin
# phid + A * (NPHI - phid) + B
nphi_heather, nphi_hugin = heather['NPHI'], hugin['NPHI']

phif_heather = phid_heather + A[0] * (nphi_heather - phid_heather) +
B[0]
phif_hugin = phid_hugin + A[1] * (nphi_hugin - phid_hugin) + B[1]

phif_hugin

# display histogram with bins 10
plt.hist(phif_hugin, bins=10)
plt.show()
```



Comment: some porosities are negative. This is normal because there are densities larger than calcite (> 2.71), while the measurement is in lst. unit check: `max(...)`

```
# maximum value of Heather's RHOB
max(rhob_heather)
```

```
2.975
```

3. Calculate shale volume (VSH)

$$VSH = VSH_{GR} = \frac{GR - GR_{min}}{GR_{max} - GR_{min}}$$

where:

GR = gamma ray log reading [API]
GR_{min} = GR reading in clean sand [API]


```
# calculate VSH for Heather and Hugin
gr_heather, gr_hugin = heather['GR'], hugin['GR']

vsh_heather = (gr_heather - gr_min[0]) / (gr_max[0] - gr_min[0])
vsh_hugin = (gr_hugin - gr_min[1]) / (gr_max[1] - gr_min[1])

vsh_heather

# print summary stats: df.describe()
vsh_heather.describe()
```

```
count    197.000000
mean       0.592781
std        0.234732
min        0.207053
25%        0.311726
50%        0.714283
75%        0.785841
max        1.061062
Name: GR, dtype: float64
```

4. Calculate water saturation (SW)

$$Sw_t = \left(\frac{a \times R_w}{\phi_F^m \times R_t} \right)^{\frac{1}{n}}$$

where:

a = Archie (tortuosity) factor
R_w = resistivity of formation water [Ohmm]
ϕ_F = Total porosity [fraction]
m = cementation exponent
R_t = true resistivity [Ohmm]
n = saturation exponent


```
# calculate SW for Heather and Hugin
rt_heather, rt_hugin = heather['RT'], hugin['RT']

sw_heather = ((a[0] * Rw[0]) / (phif_heather * rt_heather))**(1 /
n[0])
sw_hugin = ((a[1] * Rw[1]) / (phif_hugin * rt_hugin))**(1 / n[1])

sw_heather
```

```
33864    0.362193
33865    0.358470
33866    0.361508
33867    0.370357
33868    0.367641
           ...
34056    0.706821
34057    0.438811
34058    0.307335
34059    0.253053
34060    0.228789
Length: 197, dtype: float64
```

```
sw_heather.describe()
```

```
count    155.000000
mean       0.509408
std        0.278306
min        0.228789
25%        0.395769
50%        0.444801
75%        0.505650
max        2.418432
dtype: float64
```

Comment: In Heather most SW result is larger than 1! However this is true because they're in fact water zone. So, we can mask all SW values larger than 1, to be changed to 1

```
## create mask
mask_sw_heather = sw_heather <= 1

## change anyting larger than 1 with 1 using the mask
sw_heather = sw_heather[mask_sw_heather]

sw_heather
```

```
33864    0.362193
33865    0.358470
33866    0.361508
33867    0.370357
33868    0.367641
           ...
34056    0.706821
34057    0.438811
34058    0.307335
34059    0.253053
34060    0.228789
Length: 148, dtype: float64
```

```
sw_heather.describe()
```

```
count    148.000000
mean       0.457721
std        0.096288
min        0.228789
25%        0.393667
50%        0.439114
75%        0.490111
max        0.889357
dtype: float64
```

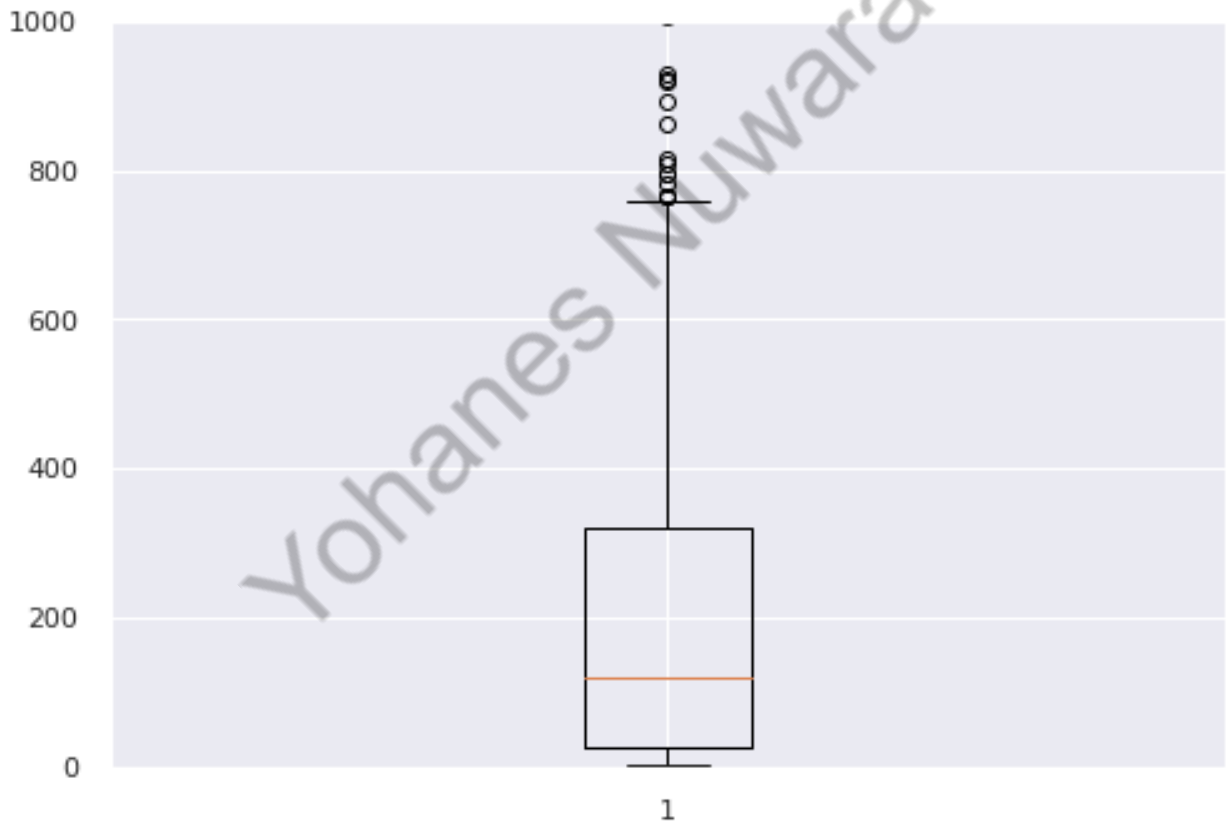

Calculate permeability (KLOGH)

The functions have been prepared for you. The equation used are from the report (p. 13), different for each formation.

Hugin Fm.: $KLOGH = 10^{(2+8\times PHIF-9\times VSH))}$

Sleipner Fm.: $KLOGH = 10^{(-3+32\times PHIF-2\times VSH))}$

```
# There is no equation for Heather Fm., so Heather is not calculated
k_hugin = 10**(2 + 8 * phif_hugin - 9 * vsh_hugin)

# display boxplot
plt.boxplot(k_hugin)
plt.ylim(-1,1000)
plt.show()
```

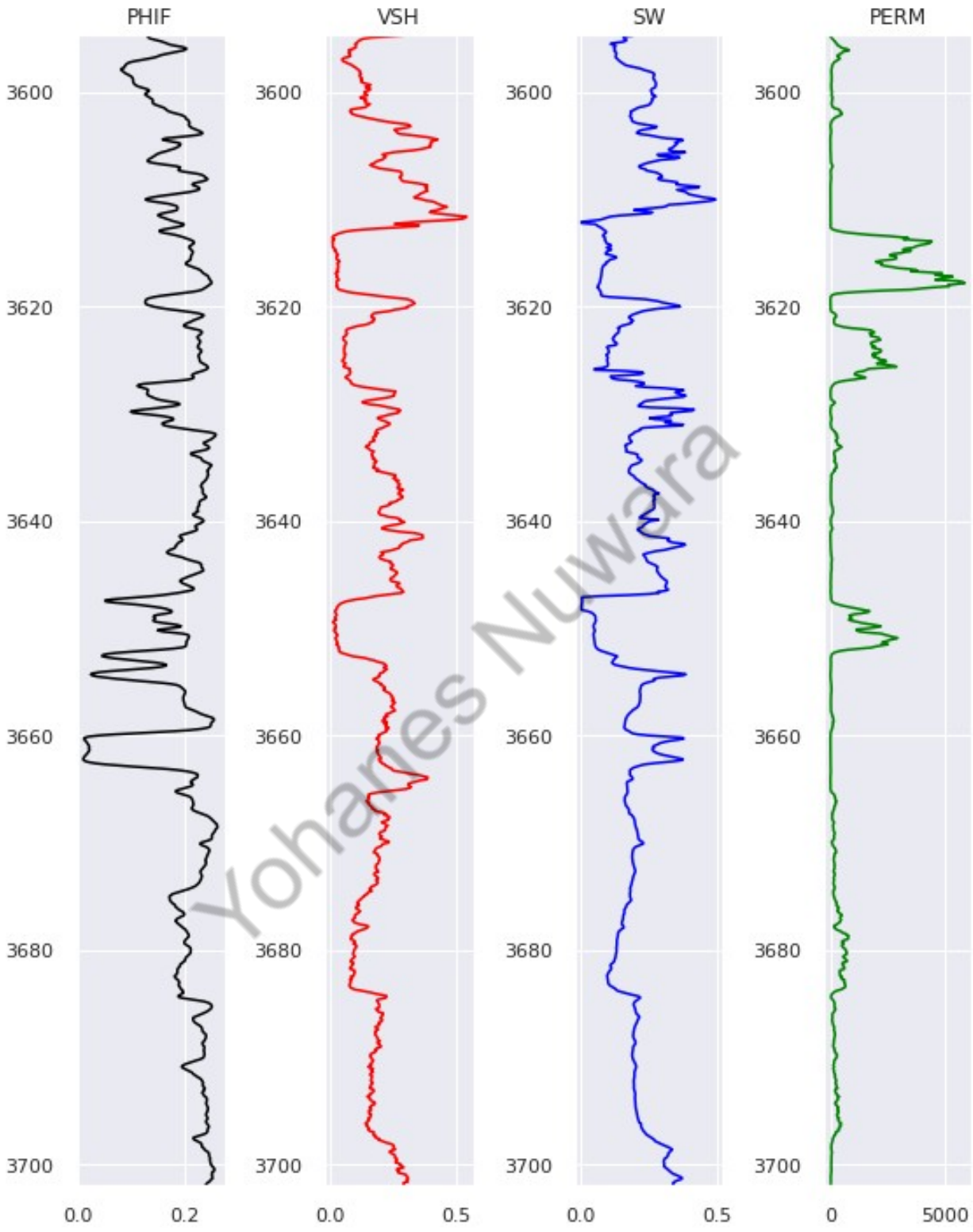


Comment: there are lots of outliers of Hugin permeability

Visualize calculations result

We have done with calculations, let's visualize them into logs. We will now practice using bare Matplotlib.

```
# visualize calculation result as derived logs (Hugin)

plt.figure(figsize=(8,10))

depth_hugin = hugin['DEPTH']
logs = [phif_hugin, vsh_hugin, sw_hugin, k_hugin]
titles = ['PHIF', 'VSH', 'SW', 'PERM']
color = ['black', 'red', 'blue', 'green']

for i in range(len(logs)):
  plt.subplot(1,4,i+1)
  plt.plot(logs[i], depth_hugin, color=color[i])
  plt.ylim(max(depth_hugin), min(depth_hugin))
  plt.title(titles[i])

plt.tight_layout(1)
plt.show()
```


PHIF
VSH
SW
PERM
3600
3620
3640
3660
3680
3700
0.0
0.2
0.0
0.5
0.0
0.5
0
5000
Yohanes Nuwara

[BONUS] Merge the calculated results to original dataframe

Our data doesn't have columns for PHIF, VSH, SW, and K yet. We first create them.

```
# create new columns for PHIF, VSH, SW, and K, and initiate with NaNs
# example: df['PHIF'] = np.full(x, np.nan) # where x is length of
dataframe
```

Before we created masks for Heather, Hugin, and Sleipner. We'll use it again now to insert all the calculated results to replace the NaNs in the dataframe

```
# apply mask
df.loc[mask_hugin, 'PHIF'] = phif_hugin
df.iloc[9946,:]
```

```
DEPTH                       3594.6
NPHI                          0.23
RHOB                         2.553
GR                          56.027
RT                           7.286
PEF                           7.81
CALI                         8.672
DT                          75.703
FM. LABEL    Hugin Fm. VOLVE Top
PHIF                     0.135257
VSH                           NaN
SW                            NaN
KLOGH                         NaN
Name: 9946, dtype: object
```


End of the training!

Now we know how to:

- Read .LAS file using `lasio` and convert to Pandas dataframe
- Generate formation top labels from formation top file .CSV
- Visualize well-logs
- Visualize triple combo
- Visualize Neutron-Density plot (with categorical and continuous variables)
- Making subsets of dataframes based on each formation name
- Compute petrophysical variables (PHIF, SW, VSH, K)
- Visualize the results
- Add the computed variables back to our original well log dataframe

Copyright

Session 3. Production Data Visualization and Decline Curve Analysis

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

!git clone 'https://github.com/yohanesnuwara/python-bootcamp-for-geoengineers'
```

```
Cloning into 'python-bootcamp-for-geoengineers'...
remote: Enumerating objects: 9, done.ote: Counting objects: 100% (9/9), done.ote: Compressing objects: 100% (9/9), done.ote: Total 162 (delta 3), reused 0 (delta 0), pack-reused 153
```

Loading BHP data

The BHP (borehole pressure) that we'll use here is BHP `Data.csv`. It's available under directory `data`. Using the same way that we've learnt in previous session, now we load the data.

```
bhp_csv = '/content/python-bootcamp-for-geoengineers/data/BHP Data.csv'

bhp_data = pd.read_csv(bhp_csv)
bhp_data.head(10)
```

```
         Date   A    B   C    D       E  ...    K   L   M   N   O  Unnamed: 16
0  1997-11-06 NaN  0.0 NaN  0.0  236.13  ...  0.0 NaN NaN NaN NaN         NaN
1  1997-11-07 NaN  0.0 NaN  0.0  220.18  ...  0.0 NaN NaN NaN NaN         NaN
2  1997-11-08 NaN  0.0 NaN  0.0  210.78  ...  0.0 NaN NaN NaN NaN         NaN
3  1997-11-09 NaN  0.0 NaN  0.0  208.19  ...  0.0 NaN NaN NaN NaN         NaN
4  1997-11-10 NaN  0.0 NaN  0.0  205.80  ...  0.0 NaN NaN NaN NaN         NaN
5  1997-11-11 NaN  0.0 NaN  0.0  205.74  ...  0.0 NaN NaN NaN NaN         NaN
6  1997-11-12 NaN  0.0 NaN  0.0  205.58  ...  0.0 NaN NaN NaN NaN         NaN
7  1997-11-13 NaN  0.0 NaN  0.0  205.18  ...  0.0 NaN NaN NaN NaN         NaN
8  1997-11-14 NaN  0.0 NaN  0.0  198.00  ...  0.0 NaN NaN NaN NaN         NaN
```

```
NaN
9  1997-11-15 NaN  0.0 NaN  0.0  212.68  ...  0.0 NaN NaN NaN NaN
NaN

[10 rows x 17 columns]
```

Data QC: Convert to Datetime Format

Check the date column first, check its `dtype`. Here `dtype` is `object`.

```
bhp_data['Date']
```

```
0       1997-11-06
1       1997-11-07
2       1997-11-08
3       1997-11-09
4       1997-11-10
           ...
4665           NaN
4666           NaN
4667           NaN
4668           NaN
4669           NaN
Name: Date, Length: 4670, dtype: object
```

We need to convert `object` to `datetime`, so that it will be recognized as date time.

Also, we will delete the `Unnamed` column at the very end.

```
# convert date string to Panda datetime format
bhp_data['Date'] =  pd.to_datetime(bhp_data['Date'], format='%Y-%m-
%d') # format check web: https://strftime.org/

bhp_data.head(10)
```

```
        Date   A    B   C    D       E  ...    K   L   M   N   O
Unnamed: 16
0 1997-11-06 NaN  0.0 NaN  0.0  236.13  ...  0.0 NaN NaN NaN NaN
NaN
1 1997-11-07 NaN  0.0 NaN  0.0  220.18  ...  0.0 NaN NaN NaN NaN
NaN
2 1997-11-08 NaN  0.0 NaN  0.0  210.78  ...  0.0 NaN NaN NaN NaN
NaN
3 1997-11-09 NaN  0.0 NaN  0.0  208.19  ...  0.0 NaN NaN NaN NaN
NaN
4 1997-11-10 NaN  0.0 NaN  0.0  205.80  ...  0.0 NaN NaN NaN NaN
NaN
5 1997-11-11 NaN  0.0 NaN  0.0  205.74  ...  0.0 NaN NaN NaN NaN
NaN
```

```
6 1997-11-12 NaN  0.0 NaN  0.0  205.58  ...  0.0 NaN NaN NaN NaN
NaN
7 1997-11-13 NaN  0.0 NaN  0.0  205.18  ...  0.0 NaN NaN NaN NaN
NaN
8 1997-11-14 NaN  0.0 NaN  0.0  198.00  ...  0.0 NaN NaN NaN NaN
NaN
9 1997-11-15 NaN  0.0 NaN  0.0  212.68  ...  0.0 NaN NaN NaN NaN
NaN

[10 rows x 17 columns]
```

Now, the `Unnamed` has been deleted, and the date column has been in `datetime`

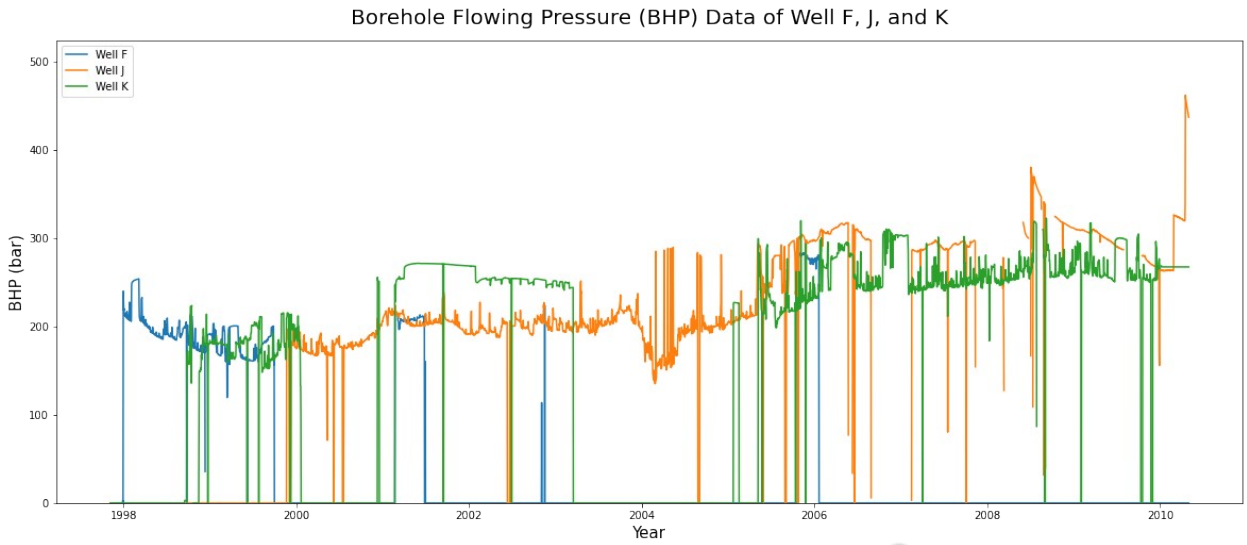
```
bhp_data['Date']
```

```
0      1997-11-06
1      1997-11-07
2      1997-11-08
3      1997-11-09
4      1997-11-10
          ...
4665          NaT
4666          NaT
4667          NaT
4668          NaT
4669          NaT
Name: Date, Length: 4670, dtype: datetime64[ns]
```

Display BHP data

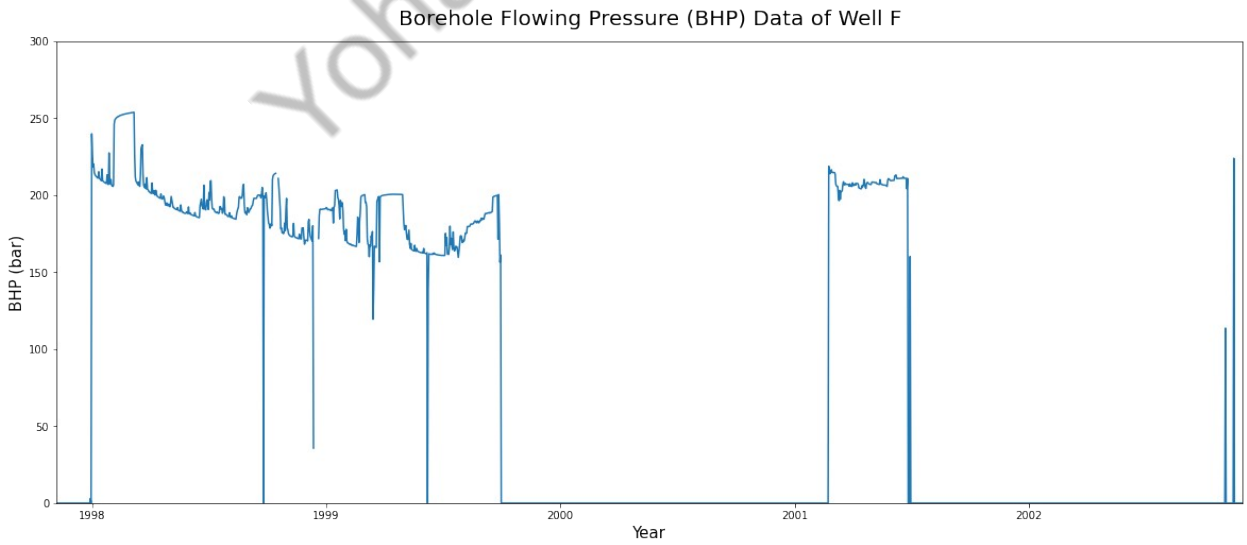
Now let us visualize the BHP of three wells: F, J, K.

```
plt.figure(figsize=(20,8))

plt.plot(bhp_data['Date'], bhp_data['F'], label='Well F')
plt.plot(bhp_data['Date'], bhp_data['J'], label='Well J')
plt.plot(bhp_data['Date'], bhp_data['K'], label='Well K')

plt.title('Borehole Flowing Pressure (BHP) Data of Well F, J, and K',
size=20, pad=15)
plt.xlabel('Year', size=15)
plt.ylabel('BHP (bar)', size=15)
plt.ylim(ymin=0)
plt.legend()
plt.show()
```



In this training, we'll use one well only, Well F.

```
plt.figure(figsize=(20,8))

plt.plot(bhp_data['Date'], bhp_data['F'])

plt.title('Borehole Flowing Pressure (BHP) Data of Well F', size=20,
pad=15)
plt.xlabel('Year', size=15)
plt.ylabel('BHP (bar)', size=15)
plt.xlim(min(bhp_data['Date']), np.datetime64('2002-12-01'))
plt.ylim(0, 300)
plt.show()
```

Loading Production rate data

Now we'll upload another data, the production rate data. It contains data of well from E to H, each contains water, oil, and gas production rate.

```
rate_csv = '/content/python-bootcamp-for-geoengineers/data/Production
Rate Well E to H.csv'

rate_data = pd.read_csv(rate_csv)
rate_data.head(10)
```

```
         DATE  Gas Rate E  Oil Rate E  ...  Gas Rate H  Oil Rate H
Water Rate H
0  06/11/1997        0.00        0.00  ...         0.0         0.0
0.0
1  07/11/1997   482594.69     4347.70  ...         0.0         0.0
0.0
2  22/11/1997   634722.75     5601.95  ...         0.0         0.0
0.0
3  09/12/1997   651415.00     5433.42  ...         0.0         0.0
0.0
4  24/12/1997   693727.75     5481.02  ...         0.0         0.0
0.0
5  11/01/1998   703808.31     5169.75  ...         0.0         0.0
0.0
6  11/02/1998    53044.25      370.62  ...         0.0         0.0
0.0
7  07/03/1998   563920.38     3487.88  ...         0.0         0.0
0.0
8  30/03/1998   772098.50     4638.90  ...         0.0         0.0
0.0
9  31/03/1998   906735.00     5427.25  ...         0.0         0.0
0.0

[10 rows x 13 columns]
```


Data QC: Convert to Datetime Format

Now we do the same thing, converting to datetime. Recognize the date type, and convert.

```
# convert date string to Panda datetime format
rate_data['DATE'] =  pd.to_datetime(rate_data['DATE'],
format='%d/%m/%Y') # format check web: https://strftime.org/

rate_data.head(10)
```

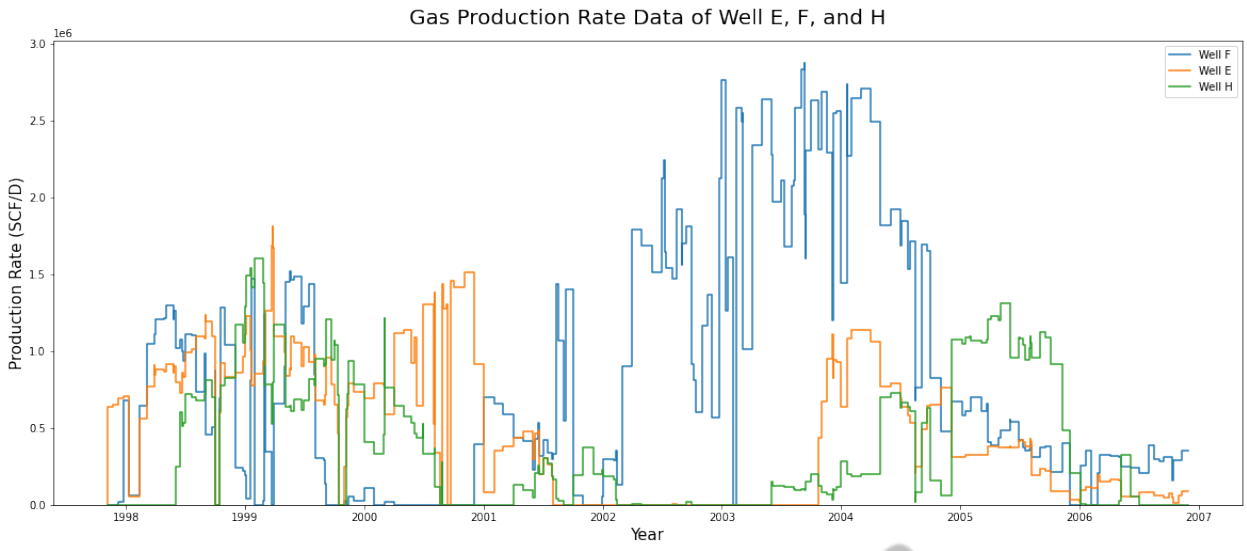
```
        DATE  Gas Rate E  Oil Rate E  ...  Gas Rate H  Oil Rate H
Water Rate H
0 1997-11-06        0.00        0.00  ...         0.0         0.0
```

```
0.0
1 1997-11-07   482594.69     4347.70  ...          0.0          0.0
0.0
2 1997-11-22   634722.75     5601.95  ...          0.0          0.0
0.0
3 1997-12-09   651415.00     5433.42  ...          0.0          0.0
0.0
4 1997-12-24   693727.75     5481.02  ...          0.0          0.0
0.0
5 1998-01-11   703808.31     5169.75  ...          0.0          0.0
0.0
6 1998-02-11    53044.25      370.62  ...          0.0          0.0
0.0
7 1998-03-07   563920.38     3487.88  ...          0.0          0.0
0.0
8 1998-03-30   772098.50     4638.90  ...          0.0          0.0
0.0
9 1998-03-31   906735.00     5427.25  ...          0.0          0.0
0.0

[10 rows x 13 columns]
```

Display Production rate data

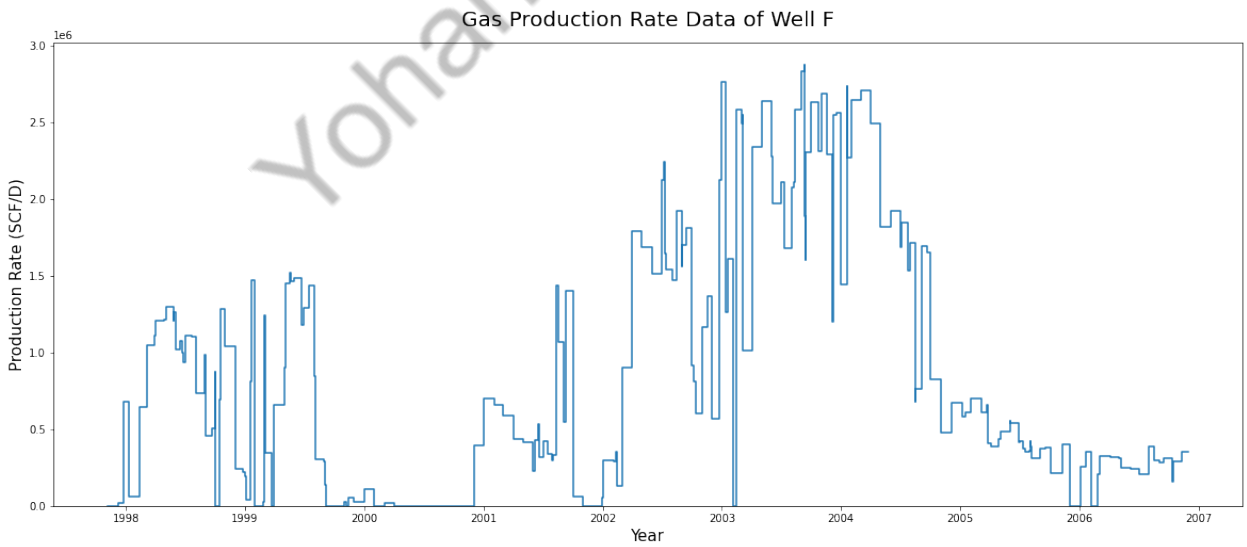
Different from the display of BHP data, we commonly display production rate data in a **step-like manner**. Instead of `plt.plot`, we use `plt.step`

```
plt.figure(figsize=(20,8))

plt.step(rate_data['DATE'], rate_data['Gas Rate F'], label='Well F')
plt.step(rate_data['DATE'], rate_data['Gas Rate E'], label='Well E')
plt.step(rate_data['DATE'], rate_data['Gas Rate H'], label='Well H')

plt.title('Gas Production Rate Data of Well E, F, and H', size=20,
pad=15)
plt.xlabel('Year', size=15)
plt.ylabel('Production Rate (SCF/D)', size=15)
plt.legend()
plt.ylim(ymin=0)

plt.show()
```



For DCA, in this training also we'll only use well F.

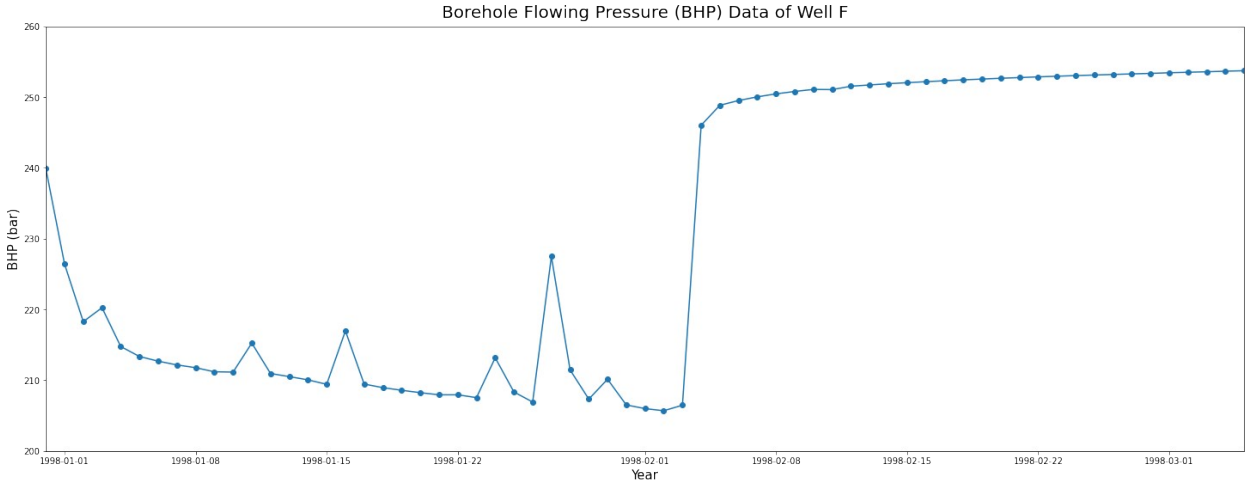
```
plt.figure(figsize=(20,8))

plt.step(rate_data['DATE'], rate_data['Gas Rate F'], label='Well F')

plt.title('Gas Production Rate Data of Well F', size=20, pad=15)
plt.xlabel('Year', size=15)
plt.ylabel('Production Rate (SCF/D)', size=15)
plt.ylim(ymin=0)
plt.show()
```

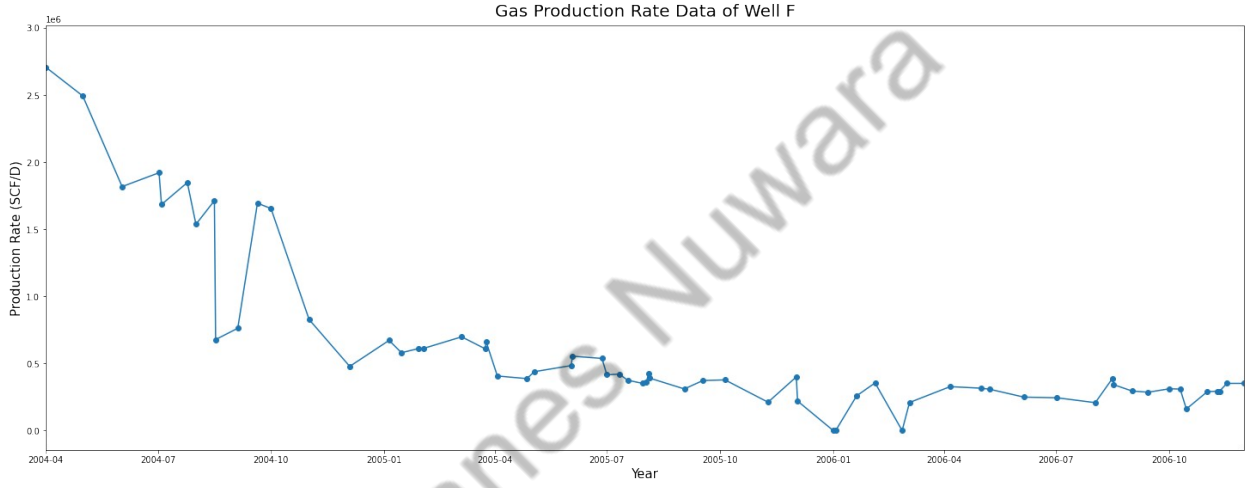


Zoom into data of interesting target for analysis (PTA and DCA)

We can zoom in into an interesting target in our data (in the display), simply by configuring `plt.xlim` and specify the date.

```
plt.figure(figsize=(25,20))

plt.subplot(2,1,1)
plt.plot(bhp_data['Date'], bhp_data['F'], 'o-')
plt.title('Borehole Flowing Pressure (BHP) Data of Well F', size=20,
pad=10)
plt.xlabel('Year', size=15)
plt.ylabel('BHP (bar)', size=15)
plt.xlim(np.datetime64('1997-12-31'), np.datetime64('1998-03-05'))
plt.ylim(200, 260)

plt.subplot(2,1,2)
plt.plot(rate_data['DATE'], rate_data['Gas Rate F'], 'o-')
plt.title('Gas Production Rate Data of Well F', size=20, pad=10)
plt.xlabel('Year', size=15)
plt.ylabel('Production Rate (SCF/D)', size=15)
plt.xlim(np.datetime64('2004-04-01'), max(rate_data['DATE'].values))
# plt.xlim(np.datetime64('1997-12-31'), np.datetime64('1998-03-05'))
# plt.ylim(200, 260)

plt.show()
```





Decline Curve Analysis (without removing outliers)

From the curve above, we'll select date for DCA, started from 1 April 2004 to date (end of the data).

Originally, we slice the data. But as for now, the sliced data has been prepared for you. Let's directly load it.

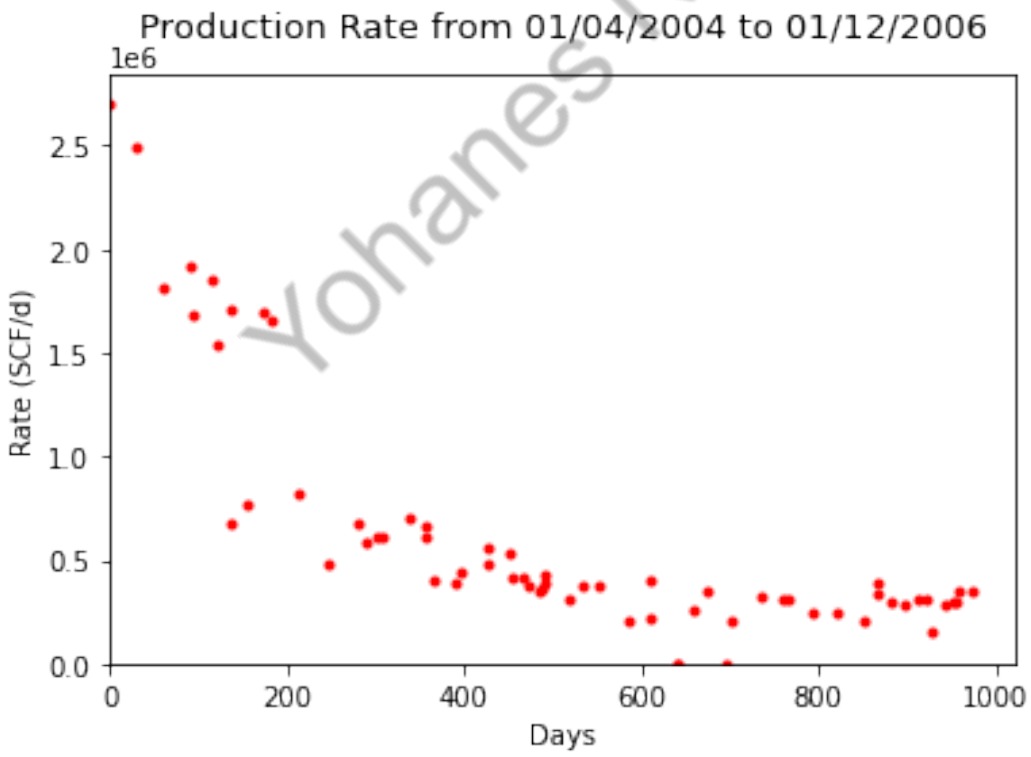
```
df =
pd.read_csv('/content/python-bootcamp-for-geoengineers/data/norne_prod
uction_rate_sample.csv')

# convert date string to Panda datetime format
df['Date'] =  pd.to_datetime(df['Date'], format='%Y-%m-%d')

t = df['Date']
q = df['Rate (SCF/d)']
```

Because in DCA we need the time axis displayed as days from 0 to N, we need to convert it. Use the following script.

```
import datetime

# subtract one datetime to another datetime
timedelta = [j-i for i, j in zip(t[:-1], t[1:])]
timedelta = np.array(timedelta)
timedelta = timedelta / datetime.timedelta(days=1)

# take cumulative sum over timedeltas
t = np.cumsum(timedelta)
t = np.append(0, t)
t = t.astype(float)
```

Now plot the production rate data

```
plt.plot(t, q, '.', color='red')
plt.title('Production Rate from 01/04/2004 to 01/12/2006', size=13,
pad=15)
plt.xlabel('Days')
plt.ylabel('Rate (SCF/d)')
plt.xlim(xmin=0); plt.ylim(ymin=0)

plt.show()
```



Next, let's do curve fitting. In curve fitting, it's always recommended to normalize our data. So we normalize our data by dividing each data by its max values.

```
# normalize the time and rate data
t_normalized = t / max(t)
q_normalized = q / max(q)
```

Next, we make the hyperbolic function for DCA.

$$q=\frac{q_i}{\left(1+b\cdot d_i\cdot t\right)^{1/b}}$$

```
# function for hyperbolic decline
def hyperbolic(t, qi, di, b):
  return qi / (np.abs((1 + b * di * t))**(1/b))
```

Let's start fitting. In curve-fitting, we can use `Scipy` package, from that we import `curve_fit`.

```
from scipy.optimize import curve_fit

popt, pcov = curve_fit(hyperbolic, t_normalized, q_normalized)
popt
```

```
array([1.05921107, 6.51659042, 0.50002664])
```

Because we had fitted on the normalized data, we need to denormalize our fitted parameters. Here's the equation that we'll use:

$$q=\frac{q_i\cdot q_{max}}{\left(1+b\cdot\frac{d_i}{t_{max}}\cdot t\right)^{1/b}}$$

```
qi, di, b = popt

# de-normalize qi and di
qi = qi * max(q)
di = di / max(t)

print('Initial production rate:', np.round(qi, 3), 'SCF')
print('Initial decline rate:', np.round(di, 3), 'SCF/D')
print('Decline coefficient:', np.round(b, 3))
```

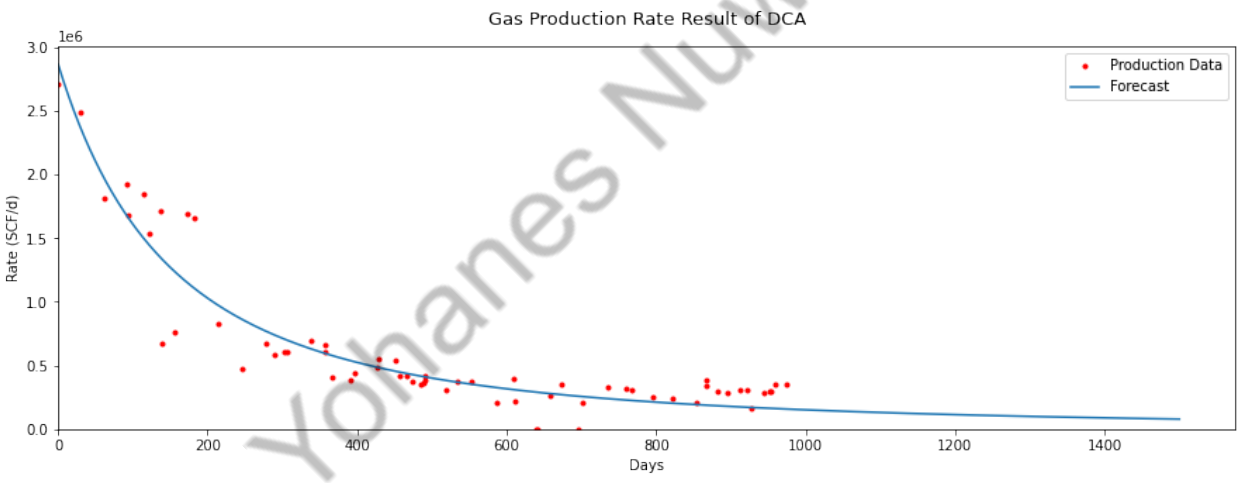
```
Initial production rate: 2866266.453 SCF
Initial decline rate: 0.007 SCF/D
Decline coefficient: 0.5
```

Let's now forecast what will be our production rate until 1,500 days!

```
# forecast gas rate until 1,500 days
t_forecast = np.arange(1501)
q_forecast = hyperbolic(t_forecast, qi, di, b)
```

Finally, we plot our DCA result.

```
# plot the production data with the forecasts (rate and cum.
production)
plt.figure(figsize=(15,5))

plt.plot(t, q, '.', color='red', label='Production Data')
plt.plot(t_forecast, q_forecast, label='Forecast')
plt.title('Gas Production Rate Result of DCA', size=13, pad=15)
plt.xlabel('Days')
plt.ylabel('Rate (SCF/d)')
plt.xlim(xmin=0); plt.ylim(ymin=0)
plt.legend()

plt.show()
```



```
# Import Numpy, Pandas, Matplotlib, Seaborn, and Missingno
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import missingno as msno
import seaborn as sns

filepath = "http://bit.ly/piopetro-data1"

df = pd.read_csv(filepath)

df.head()
```

```
     DATEPRD NPD_WELL_BORE_NAME  ...  BORE_WI_VOL   FLOW_KIND
0  07-Apr-14         15/9-F-1 C  ...          NaN  production
1  08-Apr-14         15/9-F-1 C  ...          NaN  production
2  09-Apr-14         15/9-F-1 C  ...          NaN  production
3  10-Apr-14         15/9-F-1 C  ...          NaN  production
4  11-Apr-14         15/9-F-1 C  ...          NaN  production

[5 rows x 16 columns]
```

```
# Convert datetime column to Pandas datetime format (%d-%b-%y)
df["DATEPRD"] = pd.to_datetime(df["DATEPRD"], format="%d-%b-%y")

df.head()
```

```
     DATEPRD NPD_WELL_BORE_NAME  ...  BORE_WI_VOL   FLOW_KIND
0 2014-04-07         15/9-F-1 C  ...          NaN  production
1 2014-04-08         15/9-F-1 C  ...          NaN  production
2 2014-04-09         15/9-F-1 C  ...          NaN  production
3 2014-04-10         15/9-F-1 C  ...          NaN  production
4 2014-04-11         15/9-F-1 C  ...          NaN  production

[5 rows x 16 columns]
```

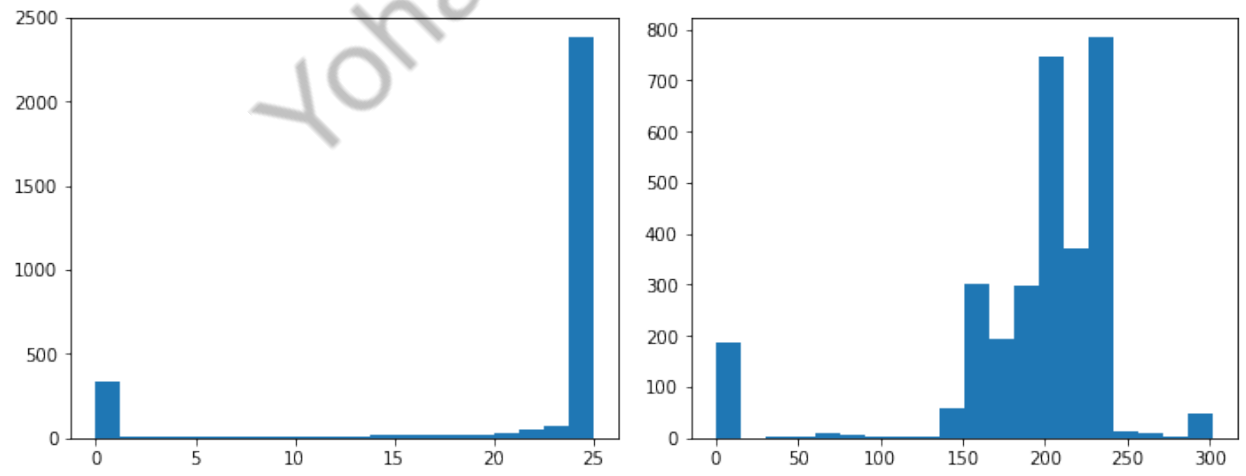
```
# Print all column names / features
df.columns
```

```
Index(['DATEPRD', 'NPD_WELL_BORE_NAME', 'ON_STREAM_HRS',
       'AVG_DOWNHOLE_PRESSURE', 'AVG_DOWNHOLE_TEMPERATURE',
'AVG_DP_TUBING',
       'AVG_ANNULUS_PRESS', 'AVG_CHOKE_SIZE_P', 'AVG_WHP_P',
'AVG_WHT_P',
       'DP_CHOKE_SIZE', 'BORE_OIL_VOL', 'BORE_GAS_VOL',
'BORE_WAT_VOL',
```

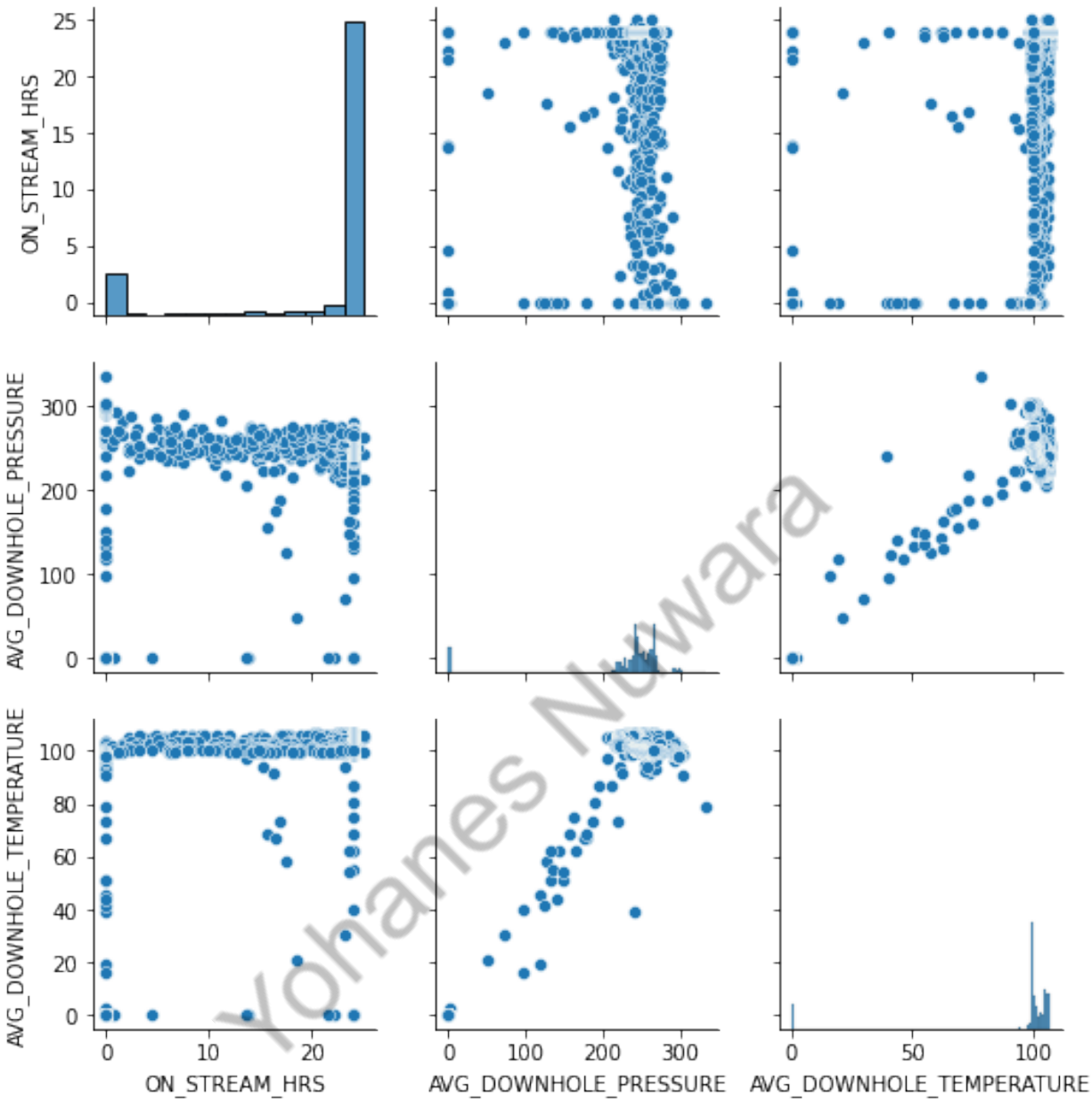
```
       'BORE_WI_VOL', 'FLOW_KIND'],
      dtype='object')

# Print all well names (unique)
df["NPD_WELL_BORE_NAME"].unique()

array(['15/9-F-1 C', '15/9-F-11', '15/9-F-12', '15/9-F-14', '15/9-F-15
D',
       '15/9-F-4', '15/9-F-5'], dtype=object)

# Separate well 15/9-F-14 from dataframe
mask = df["NPD_WELL_BORE_NAME"] == "15/9-F-14"

well14_df = df[mask].reset_index()

well14_df.head()

   index    DATEPRD NPD_WELL_BORE_NAME  ...  BORE_WAT_VOL  BORE_WI_VOL
FLOW_KIND
0   4967 2008-02-12          15/9-F-14  ...           0.0          NaN
production
1   4968 2008-02-13          15/9-F-14  ...           0.0          NaN
production
2   4969 2008-02-14          15/9-F-14  ...           0.0          NaN
production
3   4970 2008-02-15          15/9-F-14  ...           0.0          NaN
production
4   4971 2008-02-16          15/9-F-14  ...           0.0          NaN
production

[5 rows x 17 columns]

# Display summary statistics of data
well14_df.describe()

             index  ON_STREAM_HRS  ...  BORE_WAT_VOL  BORE_WI_VOL
count  3056.000000    3056.000000  ...   3056.000000          0.0
mean   6494.500000      20.541194  ...   2330.245746          NaN
std     882.335537       7.881136  ...   1462.922870          NaN
min    4967.000000       0.000000  ...    -59.000000          NaN
25%    5730.750000      24.000000  ...    695.500000          NaN
50%    6494.500000      24.000000  ...   2965.500000          NaN
75%    7258.250000      24.000000  ...   3444.250000          NaN
max    8022.000000      25.000000  ...   5692.000000          NaN

[8 rows x 14 columns]

# Plot water and oil production (use step)
plt.figure(figsize=(15,5))

plt.step(df["DATEPRD"], df["BORE_WAT_VOL"])
```

```
plt.step(df["DATEPRD"], df["BORE_OIL_VOL"])

plt.show()
```

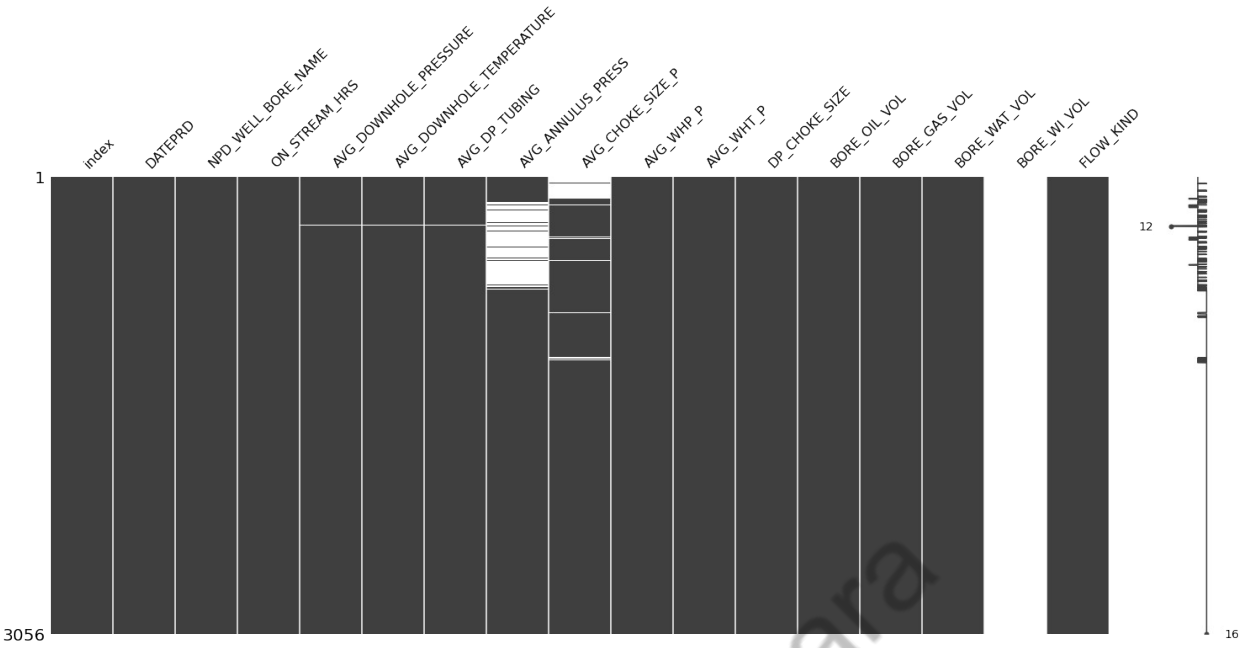

```
# Produce histogram of data (use subplots!)
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.hist(well14_df["ON_STREAM_HRS"], bins=20)

plt.subplot(1,2,2)
plt.hist(well14_df["AVG_DP_TUBING"], bins=20)

plt.tight_layout(1.5)
plt.show()
```




```
# Make pairplot between features using Seaborn
sns.pairplot(well14_df[["ON_STREAM_HRS", "AVG_DOWNHOLE_PRESSURE",
"AVG_DOWNHOLE_TEMPERATURE"]])
plt.show()
```



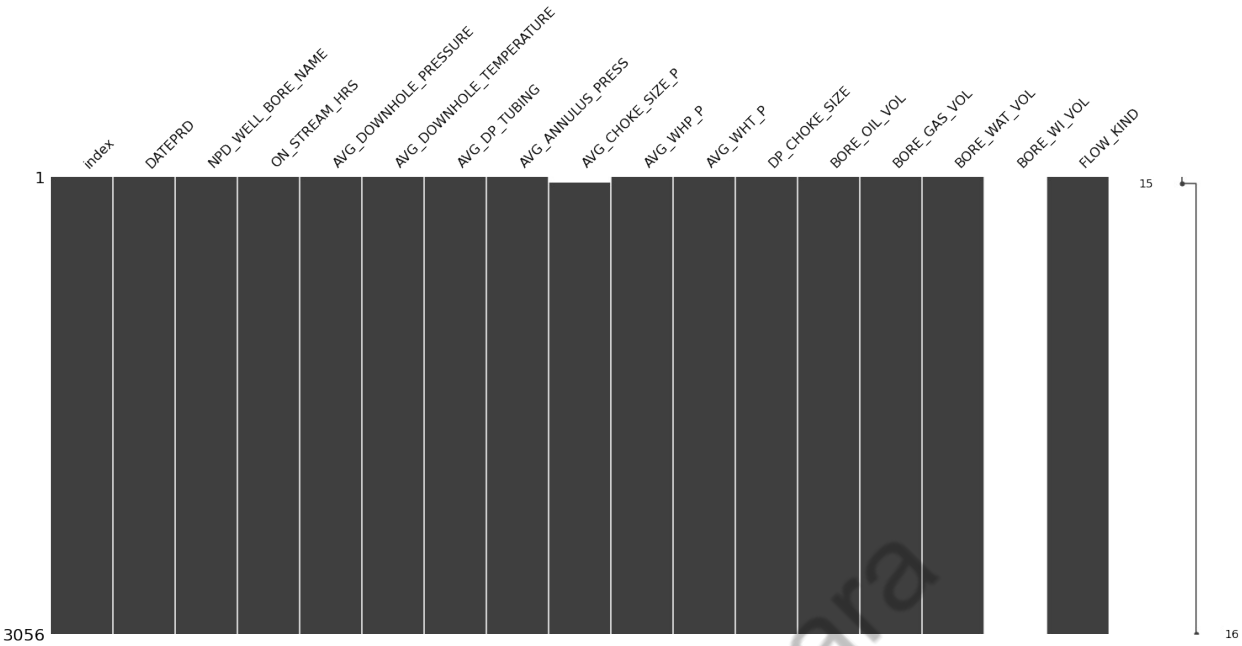

```
# Visualize missing values (non-numeric) in data using Missingno
msno.matrix(well14_df)
plt.show()
```




```
# Interpolation to fill missing values
well14_df = well14_df.interpolate(method="linear", axis=0, extr)

well14_df.head()
```

```
   index    DATEPRD NPD_WELL_BORE_NAME  ...  BORE_WAT_VOL  BORE_WI_VOL
FLOW_KIND
0   4967 2008-02-12          15/9-F-14  ...           0.0          NaN
production
1   4968 2008-02-13          15/9-F-14  ...           0.0          NaN
production
2   4969 2008-02-14          15/9-F-14  ...           0.0          NaN
production
3   4970 2008-02-15          15/9-F-14  ...           0.0          NaN
production
4   4971 2008-02-16          15/9-F-14  ...           0.0          NaN
production

[5 rows x 17 columns]
```

```
# Visualize Missigno after missing values filled w/ interpolated data
msno.matrix(well14_df)
plt.show()
```



```
# Get DCA utility
!wget
https://raw.githubusercontent.com/yohanesnuwara/pyreservoir/master/
dca/dca.py
```

```
--2021-03-02 02:59:51--
https://raw.githubusercontent.com/yohanesnuwara/pyreservoir/master/
dca/dca.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|
185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7857 (7.7K) [text/plain]
Saving to: ‘dca.py’

 dca.py                0%[                    ]       0  --.-KB/s
dca.py              100%[===================>]   7.67K  --.-KB/s    in
0s

2021-03-02 02:59:51 (85.2 MB/s) - ‘dca.py’ saved [7857/7857]
```

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.optimize import curve_fit

from dca import remove_outlier, arps_fit

# Production data link (same as session 2)
path = "http://bit.ly/piopetro-data1"

# Read data
df = pd.read_csv(path)

df.head()
```

	DATEPRD	NPD_WELL_BORE_NAME	...	BORE_WI_VOL	FLOW_KIND
0	07-Apr-14	15/9-F-1 C	...	NaN	production
1	08-Apr-14	15/9-F-1 C	...	NaN	production
2	09-Apr-14	15/9-F-1 C	...	NaN	production
3	10-Apr-14	15/9-F-1 C	...	NaN	production
4	11-Apr-14	15/9-F-1 C	...	NaN	production

[5 rows x 16 columns]

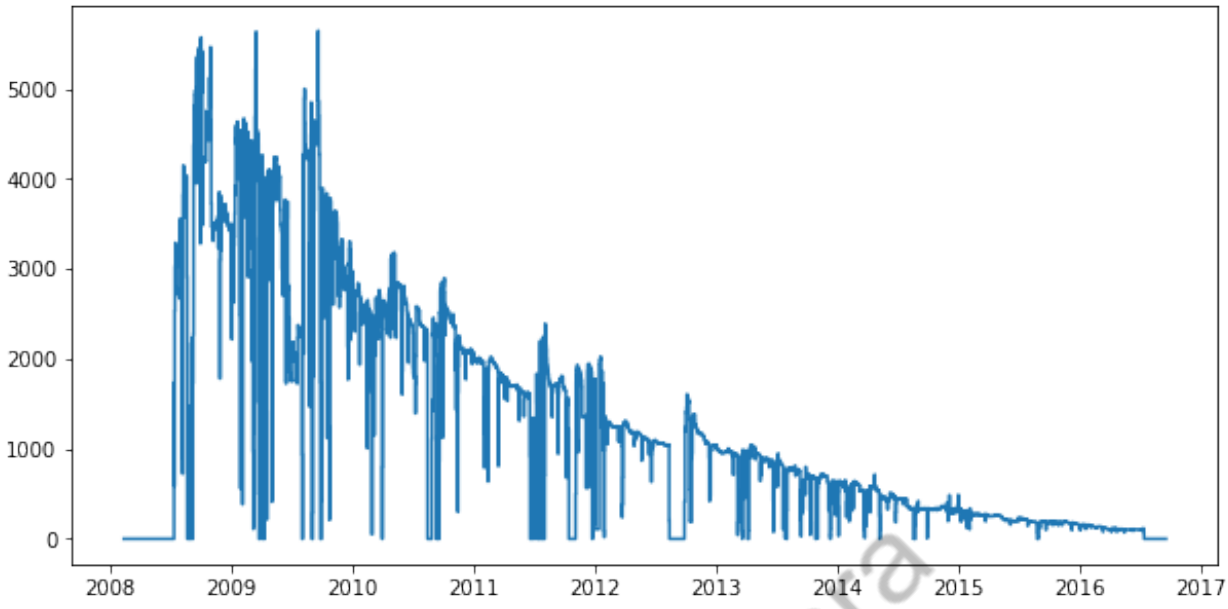
```
# Convert date column to pandas datetime format
df["DATEPRD"] = pd.to_datetime(df["DATEPRD"], format="%d-%b-%y")

df.head()
```

```
     DATEPRD NPD_WELL_BORE_NAME  ...  BORE_WI_VOL   FLOW_KIND
0 2014-04-07         15/9-F-1 C  ...          NaN  production
1 2014-04-08         15/9-F-1 C  ...          NaN  production
2 2014-04-09         15/9-F-1 C  ...          NaN  production
3 2014-04-10         15/9-F-1 C  ...          NaN  production
4 2014-04-11         15/9-F-1 C  ...          NaN  production

[5 rows x 16 columns]
```

```
# Select well 15/9-F-14 (Don't forget to reset index)
df = df[df["NPD_WELL_BORE_NAME"]=="15/9-F-14"].reset_index()

# Dataframe only has 2 columns of interest: time and rate
df = df[["DATEPRD", "BORE_OIL_VOL"]]

df.head()
```

```
     DATEPRD  BORE_OIL_VOL
0 2008-02-12           0.0
1 2008-02-13           0.0
2 2008-02-14           0.0
3 2008-02-15           0.0
4 2008-02-16           0.0
```

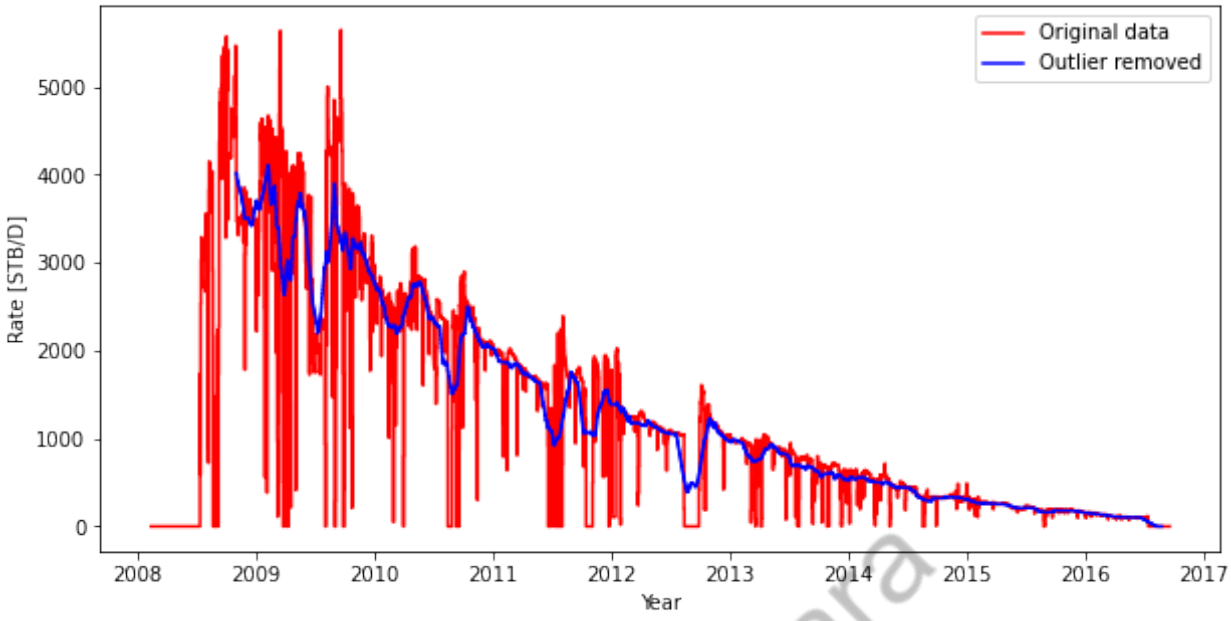
```
# Plot oil production rate
plt.figure(figsize=(10,5))
plt.step(df["DATEPRD"], df["BORE_OIL_VOL"])
plt.show()
```




```
# Removing outliers (window=150, num_stdev=50)
df2 = remove_outlier(df, "BORE_OIL_VOL", 50, 50, trim=True)

df2.head()
```

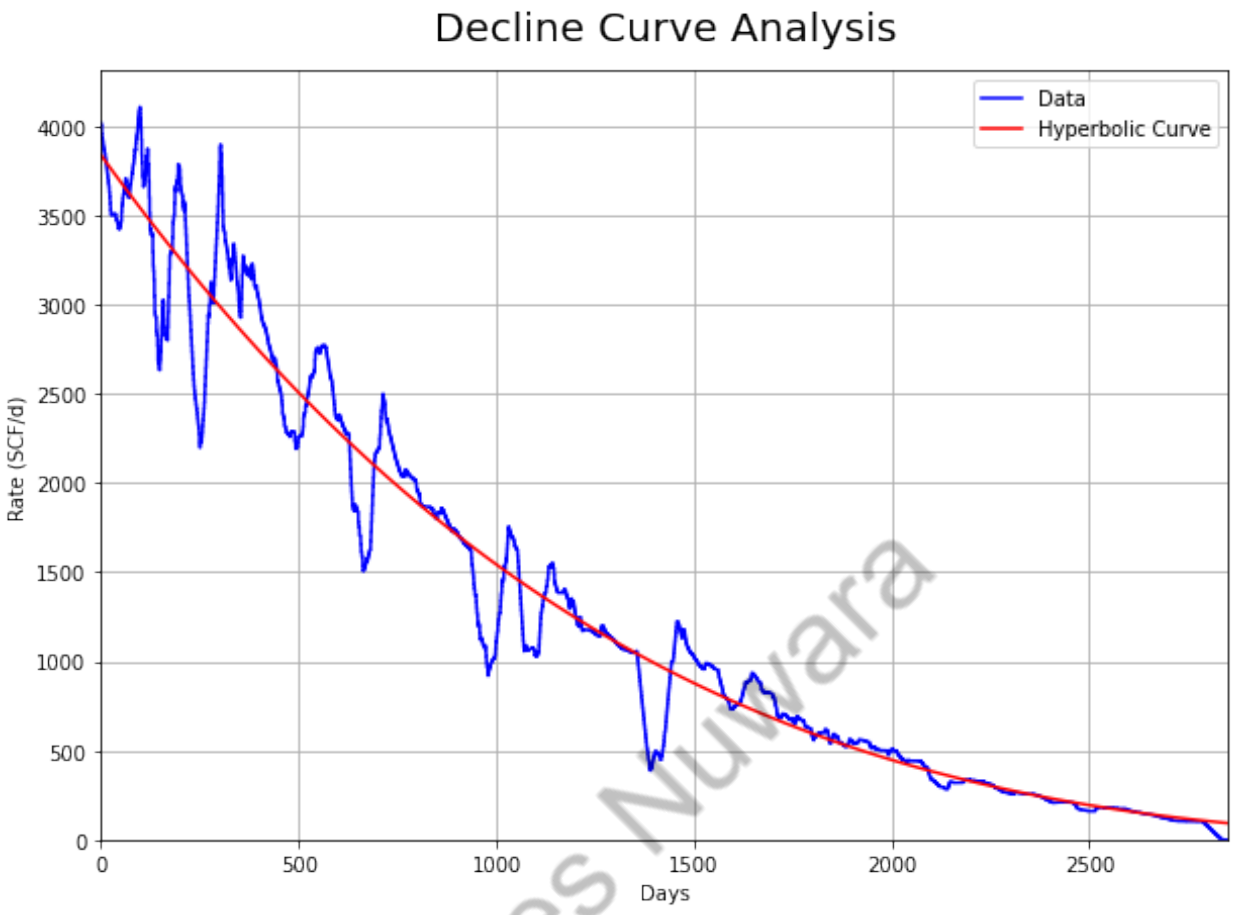
```
     DATEPRD  BORE_OIL_VOL  ...  BORE_OIL_VOL_rol_Std
BORE_OIL_VOL_is_Outlier
0 2008-11-01        4645.0  ...            593.445732
False
1 2008-11-02        3732.0  ...            593.123727
False
2 2008-11-03        3476.0  ...            592.919798
False
3 2008-11-04        3461.0  ...            670.047802
False
4 2008-11-05        3458.0  ...            679.680219
False

[5 rows x 5 columns]
```

```
# Plot outlier-removed oil production rate, compare w/ original
plt.figure(figsize=(10,5))

plt.step(df["DATEPRD"], df["BORE_OIL_VOL"], color="red",
label="Original data")
plt.step(df2["DATEPRD"], df2["BORE_OIL_VOL_rol_Av"], color="blue",
label="Outlier removed")
plt.xlabel("Year"); plt.ylabel("Rate [STB/D]")

plt.legend()
plt.show()
```




```
# Define time and production rate (from outlier-removed dataframe)
t = df2["DATEPRD"]
q = df2["BORE_OIL_VOL_rol_Av"]

# Decline curve analysis
arps_fit(t, q, plot=True)
```

```
Initial production rate (qi)  : 3835.95682 VOL/D
Initial decline rate (di)     : 0.00080 VOL/D
Decline coefficient (b)       : -0.28729
RMSE of regression            : 0.00753
```




```
(3835.956818560814,
 0.0008015095854182389,
 -0.2872863626589032,
 0.007526803769319763)

# x = np.array([-0.5, -0.25, -1, 0, 1, 0.25, 0.5])

# def f(x, a, b, c):
#   return (a*x**2) + (b*x) + c

# y = f(x, 0.5, 0.8, 10)
# noise = np.random.random(7) * 0.1
# y = np.round((y+noise), 2)

# [a,b,c], pcov = curve_fit(f, x, y)
# print(a, b, c)

# plt.scatter(x,y)
# plt.show()
```

Material Balance Analysis with Python

```
# import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# clone "pyreservoir" repository
!git clone https://github.com/yohanesnuwara/pyreservoir
```

```
Cloning into 'pyreservoir'...
remote: Enumerating objects: 122, done.ote: Counting objects: 100%
(122/122), done.ote: Compressing objects: 100% (122/122), done.ote:
Total 780 (delta 63), reused 0 (delta 0), pack-reused 658
```

```
# import system and define path
import sys
sys.path.append('/content/pyreservoir/matbal')

# import "mbal" module and from that import the functions
from mbal import drygas, gascondensate, oil
```

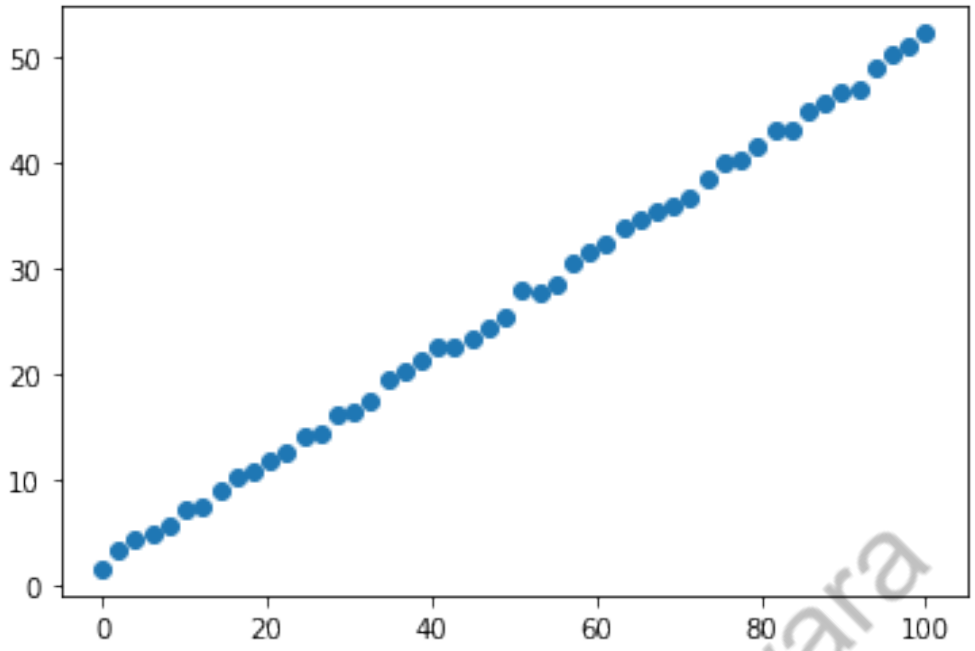
Intro to Regression (Curve-fitting)

Linear regression is the basis of material balance plot analysis. To be well-versed with our subsequent training, we will do a simple linear regression with `Scipy`.

```
# import curve_fit from Scipy optimize
from scipy.optimize import curve_fit
```

First, make a dummy data. Add with random noise.

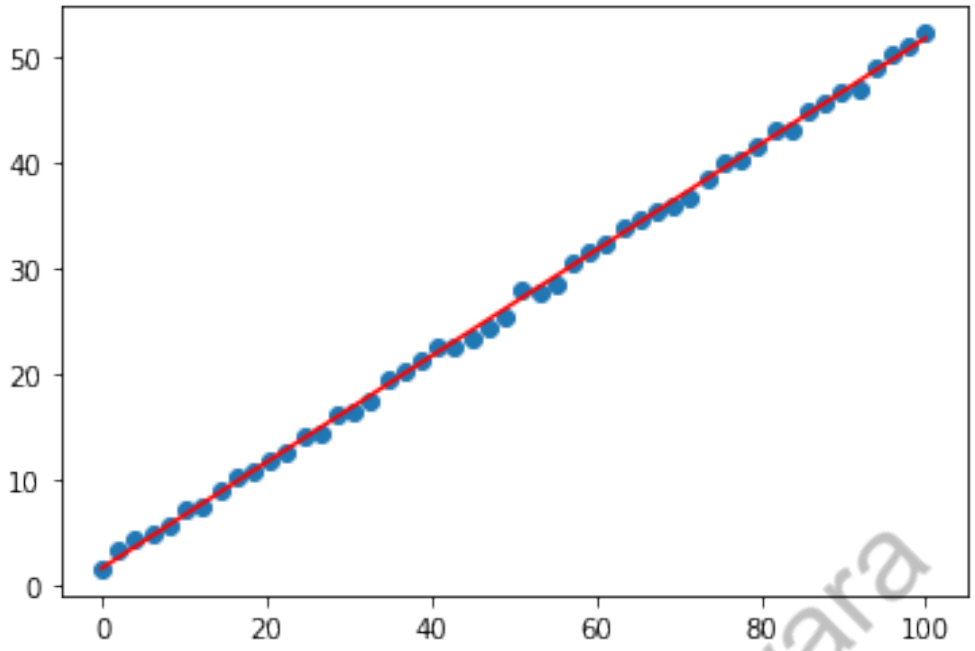
```
# create function
def linear(x, a, b):
  y = a * x + b
  return y

# create dummy data with a=0.5 and b=1.
# add with random noise, multiplied by 1.5.
x = np.linspace(0, 100, 50)
y = linear(x, 0.5, 1)
noise = np.random.random(50) * 1.5
y = y + noise

plt.plot(x,y,'o')
plt.show()
```




```
# do linear regression on x and y dummy data
popt, pcov = curve_fit(linear, x, y)

a, b = popt
print(a, b)
```

```
0.5014676464752442 1.6817879092707164
```

```
# plot the linear fit line with dummy data
yreg = a * x + b

plt.plot(x,y,'o')
plt.plot(x, yreg, 'r-')
plt.show()
```



Dry-gas material balance

We will do material balance plot analysis on a dry-gas reservoir; the data is from the U.S. Gulf Coast gas reservoir (Source: Brian F. Towler book).

First we will load the production data.

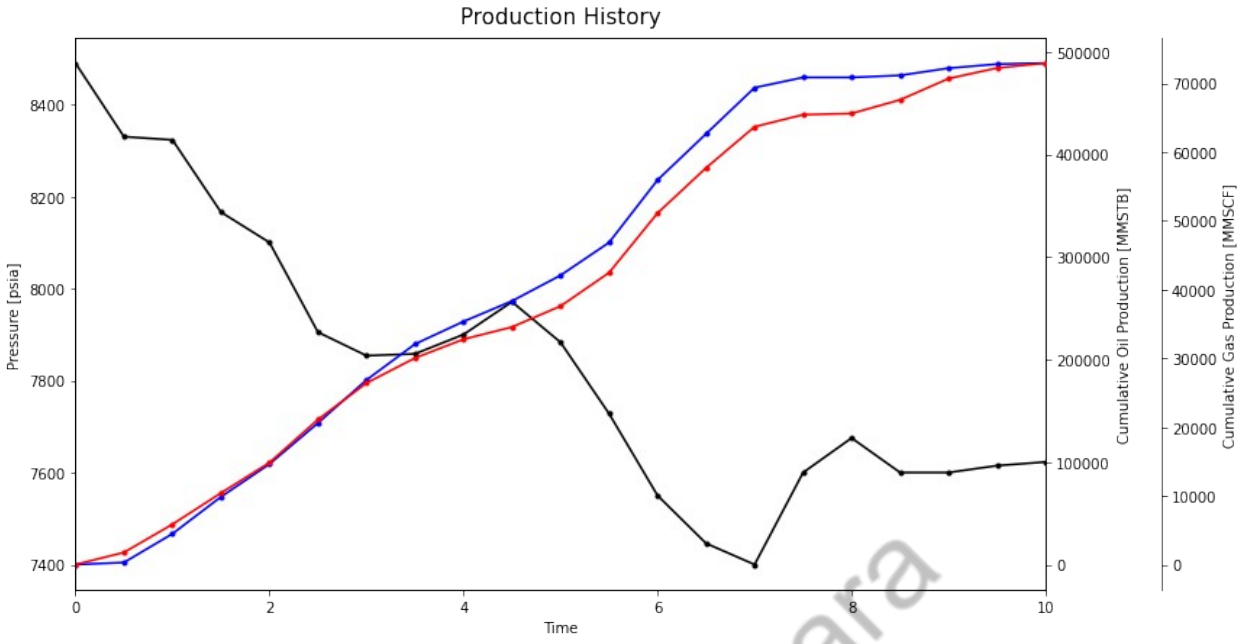
```
# load production data
df =
pd.read_csv('/content/pyreservoir/data/dry_gas_with_waterdrive.csv')

df.head(10)
```

	t	p	Gp	Np	total_Gp	Wp	Bg	Bw	z
0	0.0	8490.0	0.0	0.0	0.00	0	0.5404	1.0518	1.271392
1	0.5	8330.0	1758.0	2000.0	1759.42	0	0.5458	1.0520	1.259897
2	1.0	8323.0	5852.0	30000.0	5873.24	1000	0.5460	1.0520	1.259299
3	1.5	8166.0	10410.0	66000.0	10456.73	3000	0.5516	1.0522	1.248217
4	2.0	8100.0	14828.0	98000.0	14897.38	4000	0.5540	1.0522	1.243516
5	2.5	7905.0	21097.0	138000.0	21194.70	7000	0.5614	1.0524	1.229789
6	3.0	7854.0	26399.0	180000.0	26526.44	9000	0.5634	1.0525	1.226208

```
7  3.5  7858.0  30042.0  215000.0  30194.22  10000  0.5632  1.0525
1.226397
8  4.0  7900.0  32766.0  237000.0  32933.80  11000  0.5616  1.0524
1.229449
9  4.5  7971.0  34548.0  257000.0  34729.96  11000  0.5588  1.0524
1.234314
```

We want to plot the reservoir pressure, cum. oil production, and cum. gas production into one plot.

```
# Define variables for to visualize
t = df['t'].values
Gp = df['Gp'].values
Np = df['Np'].values
p = df['p'].values

# Plot pressure and cum production in one plot: use .twinx()
fig = plt.figure(figsize=(12,7))
host = fig.add_subplot(111)

par1 = host.twinx()
par2 = host.twinx()

host.plot(t, p, '.-', color='black')
par1.plot(t, Np, '.-', color='blue')
par2.plot(t, Gp, '.-', color='red')

host.set_title('Production History', size=15, pad=10)
host.set_xlim(0, max(t))
host.set_xlabel('Time')
host.set_ylabel('Pressure [psia]')
par1.set_ylabel('Cumulative Oil Production [MMSTB]')
par2.set_ylabel('Cumulative Gas Production [MMSCF]')

par2.spines['right'].set_position(('outward', 80))

plt.show()
```



Now, the material balance plot analysis.

The reservoir has some properties as listed:

- Formation compressibility 3.5 microsip
- Water compressibility 2.9 microsip
- Initial water saturation 0.21

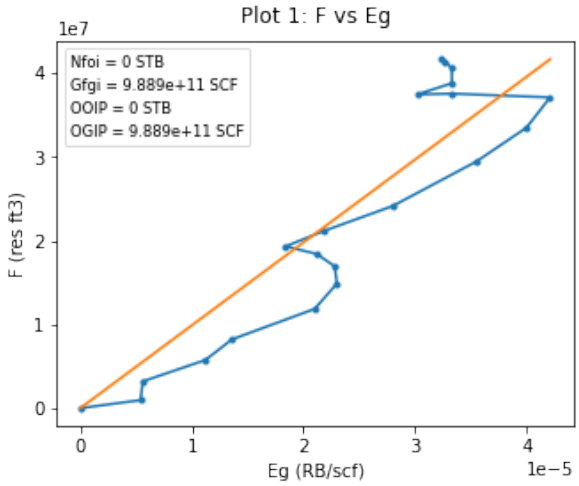
And also, some variables in the production data are NOT recorded in FIELD UNITS. They are:

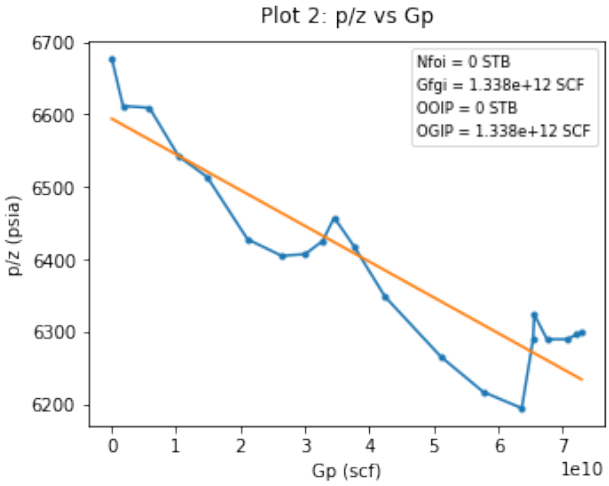
- Cum. gas production [MMSCF]
- Gas FVF [RB / MSCF]

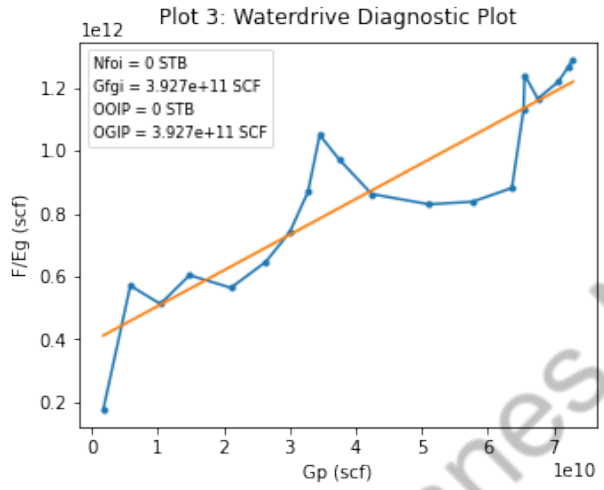
```
# see help to find out the required inputs
help(drygas)
```

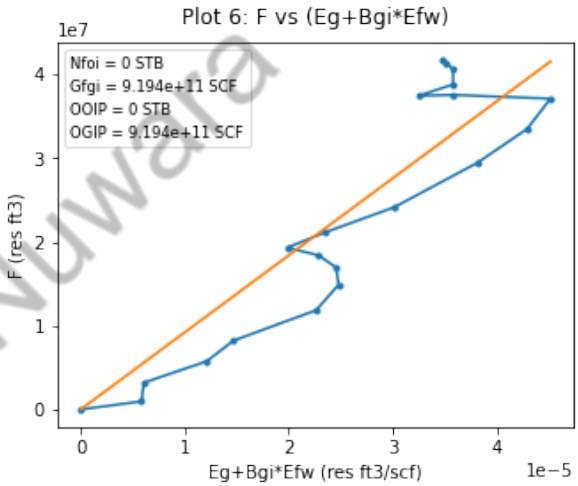
```
Help on class drygas in module mbal:

class drygas(builtins.object)
 |  Dry-Gas Material Balance Plot
 |
 |  Methods defined here:
 |
 |  calculate_params(self, p, Bg, Gp, cf, cw, swi)
 |      Calculate Material Balance Paramaters for Dry-Gas Reservoir
 |
 |      Output: F, Btg, Efw, Eg
 |
 |  plot(self, p, z, Gp, F, Btg, Efw, Eg)
 |      Create Material Balance Plots for Dry-Gas Reservoir
 |
```

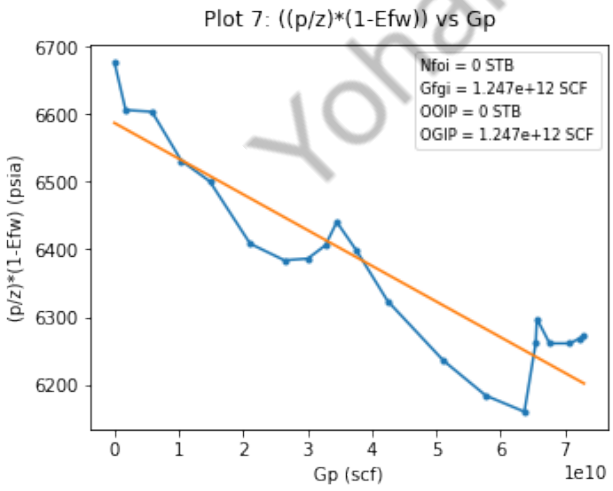
```
 |
----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)


# input all the required ones.
# variables need conversion: Bg, Gp

Bg = df['Bg'].values * (1 / 1E+3) # convert to RB/scf
Gp = df['Gp'].values * (1E+6) # convert to scf
cf = 3.5E-6 # sip
cw = 2.9E-6 # sip
swi = 0.21
z = df['z'].values

# calculate parameters for plotting: use function "calculate_params"
x = drygas()
F, Btg, Efw, Eg = x.calculate_params(p, Bg, Gp, cf, cw, swi)

# create MBAL plots and automatically give in place results
plt.figure(figsize=(10,12)) # define the figure size
x.plot(p, z, Gp, F, Btg, Efw, Eg)

/content/pyreservoir/matbal/mbal.py:143: RuntimeWarning: invalid value
encountered in true_divide
  x3, y3 = Gp, (F / Eg)
```












```
(array([       0.        ,   959516.4       ,  3195192.        ,
        5742156.        ,  8214712.        , 11843855.8       ,
       14873196.6       , 16919654.4       , 18401385.6       ,
       19305422.4       , 21133098.        , 24126306.4       ,
       29438280.3       , 33487338.8       , 37092435.        ,
```

```
       37538338.4       , 37438777.8       , 38778104.1       ,
       40553705.6       , 41392720.60000001, 41781750.4       ]),
 array([0.00e+00, 5.40e-06, 5.60e-06, 1.12e-05, 1.36e-05, 2.10e-05,
       2.30e-05, 2.28e-05, 2.12e-05, 1.84e-05, 2.18e-05, 2.80e-05,
       3.55e-05, 4.00e-05, 4.21e-05, 3.33e-05, 3.02e-05, 3.33e-05,
       3.33e-05, 3.27e-05, 3.24e-05]),
 array([0.        , 0.0008322 , 0.00086861, 0.00168521, 0.00202849,
       0.00304274, 0.00330801, 0.0032872 , 0.00306875, 0.00269946,
       0.00315717, 0.00396336, 0.00488919, 0.00543012, 0.00566938,
       0.00462913, 0.00423903, 0.00462913, 0.00462913, 0.00455111,
       0.0045095 ]))
```

Gas-condensate reservoir

We will do material balance plot analysis on a gas-condensate reservoir; the data is from the Bacon Lime gas reservoir (Source: Brian F. Towler book).

```
# load production data
df =
pd.read_csv('/content/pyreservoir/data/baconlime_gas_condensate.csv')

df.head(10)
```

	p	Np	Gp	Bg	Bo	Rs	Rv	z
0	3700.0	0.0	0.00	0.87	10.058	11560.7	86.5	0.93177
1	3650.0	28600.0	0.34	0.88	2.417	2378.0	81.5	0.92941
2	3400.0	93000.0	1.20	0.92	2.192	2010.0	70.5	0.91859
3	3100.0	231000.0	3.30	0.99	1.916	1569.0	56.2	0.90791
4	2800.0	270000.0	4.30	1.08	1.736	1272.0	46.5	0.90013
5	2500.0	379000.0	6.60	1.20	1.617	1067.0	39.5	0.89566
6	2200.0	481000.0	9.10	1.35	1.504	873.0	33.8	0.89485
7	1900.0	517200.0	10.50	1.56	1.416	719.0	29.9	0.89795
8	1600.0	549000.0	12.00	1.85	1.326	565.0	27.3	0.90507
9	1300.0	580000.0	12.80	2.28	1.268	461.0	25.5	0.91611

Now, the material balance plot analysis.

The reservoir has some properties as listed:

- Formation and water are incompressible (0 sip)
- Initial water saturation 0.2
- Dewpoint pressure 3,691 psia
- No gas injection

And also, some variables in the production data are NOT recorded in FIELD UNITS. They are:

- Cum. gas production [BSCF]
- Gas FVF [RB / MSCF]
- Volatile oil-gas ratio [STB / MMSCF]

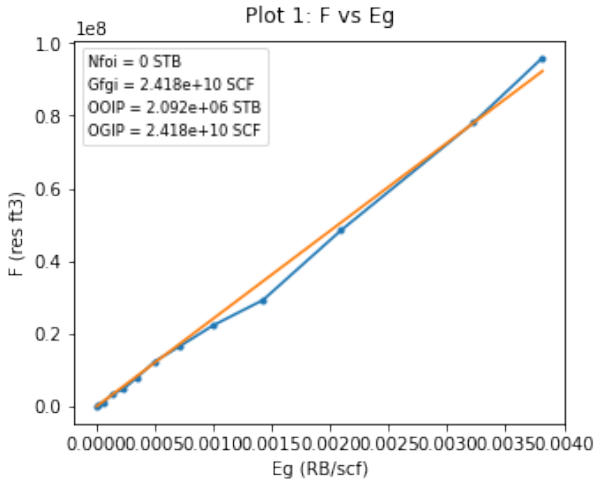
```
# see help to find out the required inputs
help(gascondensate)
```

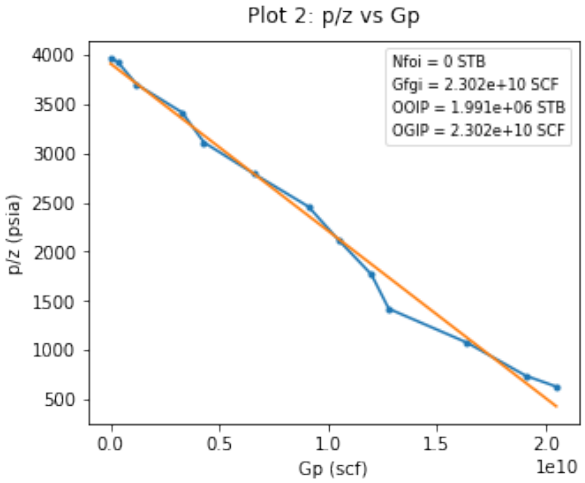
```
Help on class gascondensate in module mbal:

class gascondensate(builtins.object)
 |  Gas-Condensate Material Balance Plot
 |
 |  Methods defined here:
 |
 |  calculate_params(self, p, pdew, Bg, Bo, Np, Gp, Gi, cf, cw, swi,
Rs, Rv)
 |      Calculate Material Balance Paramaters for Gas-Condensate
Reservoir
 |
 |      Output: F, Btg, Efw, Eg
 |
 |  plot(self, p, z, Gp, F, Btg, Efw, Eg, Rv)
 |      Create Material Balance Plots for Dry-Gas Reservoir
 |
 |
----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

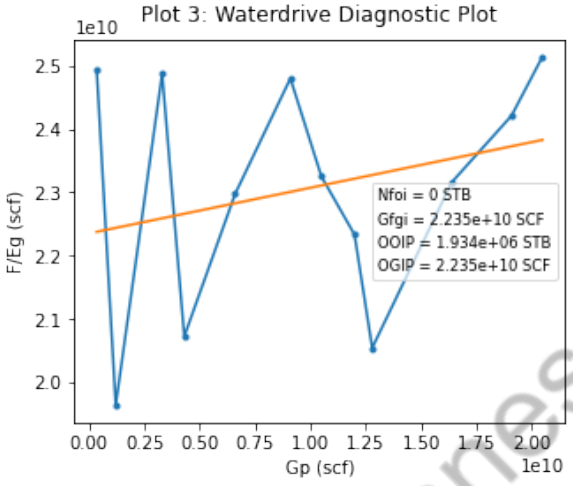
```
# input all the required ones. Gi = 0 (no gas injection)
# variables need conversion: Bg, Gp, Rv.

p = df['p'].values
pdew = 3691 # dewpoint pressure, psia
Bg = df['Bg'].values * (1 / 1E+3) # convert to RB/scf
Bo = df['Bo'].values
Np = df['Np'].values
Gp = df['Gp'].values * 1E+9 # convert to SCF
Gi = np.zeros(len(df)) # no gas injection
cf = 0
cw = 0
swi = 0.1
Rs = df['Rs'].values
Rv = df['Rv'].values * (1 / 1E+6) # convert to STB/SCF
z = df['z'].values

# calculate parameters for plotting: use function "calculate_params"
x = gascondensate()
```

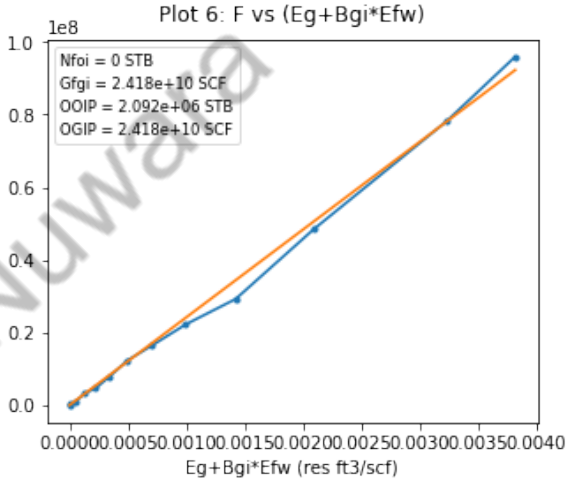
```
F, Btg, Efw, Eg = x.calculate_params(p, pdew, Bg, Bo, Np, Gp, Gi, cf,
cw, swi, Rs, Rv)

# create MBAL plots and automatically give in place results
plt.figure(figsize=(10,12))
x.plot(p, z, Gp, F, Btg, Efw, Eg, Rv)
```

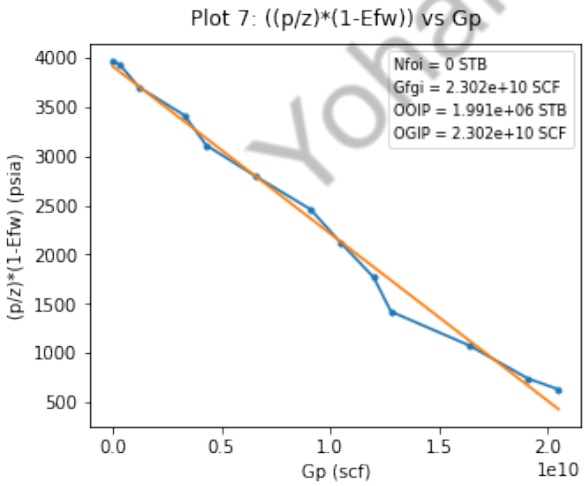
```
/content/pyreservoir/matbal/mbal.py:362: RuntimeWarning: invalid value
encountered in true_divide
  x3, y3 = Gp, (F / Eg)
```












```
(array([       0.        ,   299558.07852462,  1107354.93041437,
        3285114.17028451,  4670970.14195644,  7961571.88964701,
       12343155.55832162, 16441143.13114773, 22263131.82686345,
       29239665.28526088, 48424305.0861038 , 78147206.39933419,
       95961416.12370604]),
```

```
 array([0.00000000e+00, 1.20116771e-05, 5.63903436e-05, 1.32052248e-
04,
        2.25400509e-04, 3.46516304e-04, 4.97672690e-04, 7.07026820e-
04,
        9.96880778e-04, 1.42338952e-03, 2.09072925e-03, 3.22765428e-
03,
        3.81632116e-03]),
 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]))
```

Saturated Volatile Oil Reservoir

We will do material balance plot analysis on a saturated volatile oil reservoir; the data is from the Prudhoe Bay oil reservoir (Source: Brian F. Towler book).

```
# load production data
columns = ['p', 'Bo', 'Bg', 'Rs', 'Rv', 'Np', 'Gp']
df = pd.read_csv('/content/pyreservoir/data/Table 12.16 (edited).csv',
names=columns)

df.head(10)
```

	p	Bo	Bg	Rs	Rv	Np	Gp
0	4658.0	2.70727	0.830	2834	116	0	0.000000e+00
1	4598.0	2.63143	0.835	2711	111	1345320	3.360340e+09
2	4398.0	2.33771	0.853	2247	106	5847480	1.462202e+10
3	4198.0	2.20391	0.874	2019	94	10069920	2.592911e+10
4	3998.0	2.09309	0.901	1828	84	14145840	3.850768e+10
5	3798.0	1.99116	0.933	1651	74	17862120	5.272101e+10
6	3598.0	1.90524	0.970	1500	66	20952360	6.766090e+10
7	3398.0	1.82832	1.015	1364	60	23563080	8.400850e+10
8	3198.0	1.75726	1.066	1237	54	25814160	1.012640e+11
9	2998.0	1.68592	1.125	1111	49	27825480	1.194740e+11

Now, the material balance plot analysis.

The reservoir has some properties as listed:

- Formation and water are incompressible (0 sip)
- Initial water saturation 0.2
- No gas injection

And also, some variables in the production data are NOT recorded in FIELD UNITS. They are:

- Cum. gas production [100 * SCF]
- Cum. oil production [100 * STB]
- Gas FVF [RB / MSCF]
- Volatile oil-gas ratio [STB / MMSCF]

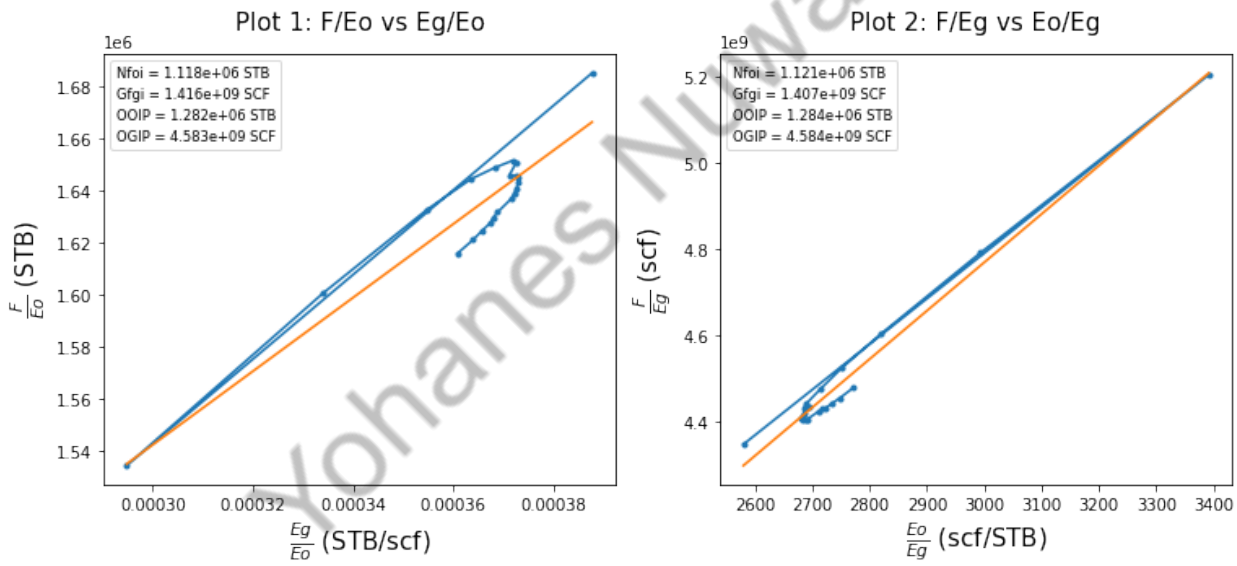
```
# see help to find out the required inputs
help(oil)
```

```
Help on class oil in module mbal:

class oil(builtins.object)
 |  Oil (Undersaturated and saturated; Volatile and Non-volatile)
Material Balance Plot
 |
 |  Methods defined here:
 |
 |  calculate_params(self, p, Bo, Bg, Rv, Rs, Np, Gp, Gi, cf, cw, swi)
 |      Calculate Material Balance Paramaters for Oil Reservoir
 |
 |      Output: F, Bto, Btg, Efw, Eo, Eg
 |
 |  gascap(self, Gfgi, Nfoi, Bg, Bo)
 |      Calculate Total Oil+Gas Expansion Factor from known Gas Cap
ratio
 |      Gfgi and Nfoi known from volumetrics
 |
 |  plot(self, oil_type, F, Bto, Btg, Efw, Eo, Eg, Np, Bo, Rs, Rv,
start=0, end=-1, figsize=(10, 5))
 |      Create Material Balance Plots for Oil Reservoir
 |
 |      Input:
 |      oil_type: 'undersaturated' or 'saturated'
 |
 |
----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

```
# input all the required ones. Gi = 0 (no gas injection)
# variables need conversion: Bg, Gp, Np, Rv.
p = df['p'].values
Bo = df['Bo'].values
Bg = df['Bg'].values * (1 / 1E+3) # convert RB/MSCF to RB/SCF
Rv = df['Rv'].values * (1 / 1E+6) # convert STB/MMSCF to STB/SCF
Rs = df['Rs'].values
Np = df['Np'].values / 100
Gp = df['Gp'].values / 100
Gi = np.zeros(len(df))
cf = 0
```

```
cw = 0
swi = 0.2

# calculate parameters for plotting: use function "calculate_params"
x = oil()
F, Bto, Btg, Efw, Eo, Eg = x.calculate_params(p, Bo, Bg, Rv, Rs, Np,
Gp, Gi, cf, cw, swi)

# create MBAL plots and automatically give in place results
x.plot('saturated', F, Bto, Btg, Efw, Eo, Eg, Np, Bo, Rs, Rv,
figsize=(16,5))
```

```
/content/pyreservoir/matbal/mbal.py:613: RuntimeWarning: invalid value
encountered in true_divide
  x1, y1 = (Eg / Eo), (F / Eo)
/content/pyreservoir/matbal/mbal.py:648: RuntimeWarning: invalid value
encountered in true_divide
  x2, y2 =  (Eo / Eg), (F / Eg)
```



Undersaturated Non-Volatile Oil Reservoir

We will do material balance plot analysis on an undersaturated non-volatile oil reservoir; the data is from Hugin reservoir in the Volve Field (Source: Equinor Data Village).

```
# load production data
df =
pd.read_csv('/content/pyreservoir/data/volve/volve_production.csv')

df.head(10)
```

```
         Date  p (psia)  Np (STB)  ...  Bg (RB/SCF)  Rs (SCF/STB)  Rv
(STB/SCF)
```

```
0  2008-01-01   4780.59         0  ...     0.000962    627.711085
0.0
1  2008-02-01   4725.76    308749  ...     0.000970    627.711085
0.0
2  2008-03-01   4521.70    833031  ...     0.001003    627.711085
0.0
3  2008-04-01   4297.32   1301790  ...     0.001043    627.711085
0.0
4  2008-05-01   4353.89   2090960  ...     0.001032    627.711085
0.0
5  2008-06-01   4359.69   2995280  ...     0.001031    627.711085
0.0
6  2008-07-01   4362.30   4041070  ...     0.001031    627.711085
0.0
7  2008-08-01   4309.95   5081590  ...     0.001041    627.711085
0.0
8  2008-09-01   4231.05   6290800  ...     0.001056    627.711085
0.0
9  2008-10-01   4085.44   7782460  ...     0.001086    627.711085
0.0

[10 rows x 12 columns]
```

Now, the material balance plot analysis.

The reservoir has some properties as listed:

- Formation compressibility is 48.4 microsip
- Water compressibility is 46.7 microsip
- Initial water saturation 0.5
- There is gas injection

All variables are already in FIELD UNITS. No need for conversion.

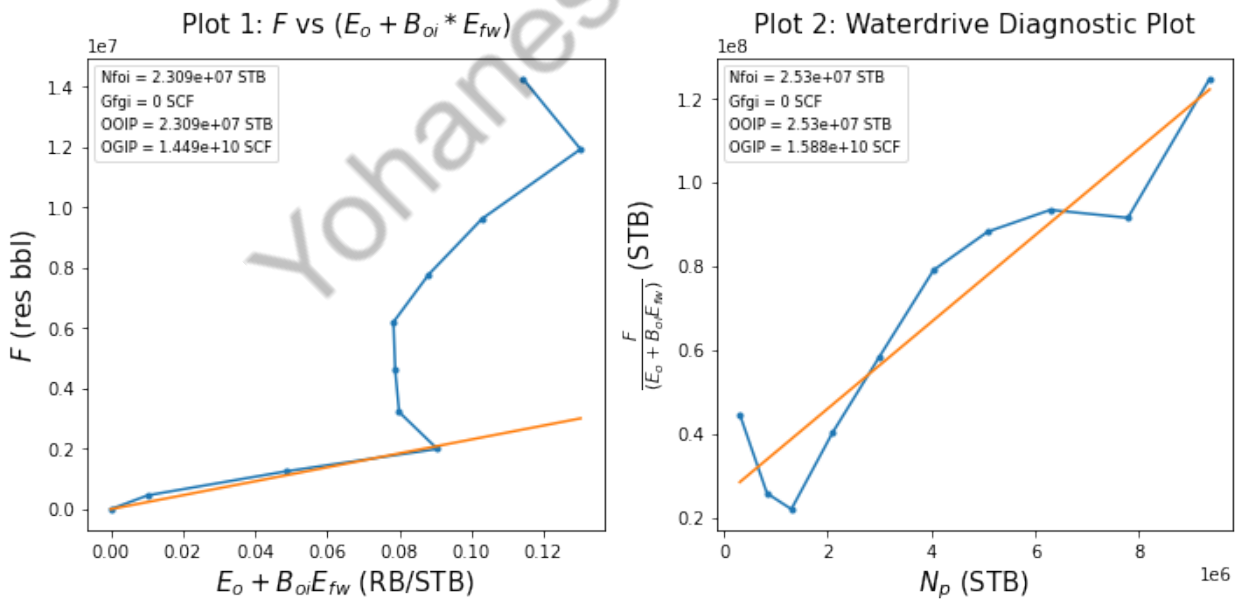
```
df.columns
```

```
Index(['Date', 'p (psia)', 'Np (STB)', 'Gp (SCF)', 'Wp (STB)', 'Gi
(SCF)',
       'Wi (STB)', 'Rp (SCF/STB)', 'Bo (RB/STB)', 'Bg (RB/SCF)',
       'Rs (SCF/STB)', 'Rv (STB/SCF)'],
      dtype='object')
```

```
# input all the required ones
p = df['p (psia)'].values
Bo = df['Bo (RB/STB)'].values
Bg = df['Bg (RB/SCF)'].values
Rv = df['Rv (STB/SCF)'].values
Rs = df['Rs (SCF/STB)'].values
Np = df['Np (STB)'].values
Gp = df['Gp (SCF)'].values
```

```
Gi = df['Gi (SCF)'].values
cf = 48.4E-6 # sip
cw = 46.7E-6 # sip
swi = 0.5

# calculate parameters for plotting: use function "calculate_params"
x = oil()
F, Bto, Btg, Efw, Eo, Eg = x.calculate_params(p, Bo, Bg, Rv, Rs, Np,
Gp, Gi, cf, cw, swi)

# create MBAL plots and automatically give in place results
x.plot('undersaturated', F, Bto, Btg, Efw, Eo, Eg, Np, Bo, Rs, Rv,
end=4)


## Comment: Why end=4? It seems that this reservoir is water-driven
(after
## the first 4 points). Passing end=4 restricts the linear line, and
it gives
## OOIP very near to the computed OOIP by someone else: 22 MMSTB. Ours
is 23 MMSTB.
```

```
/content/pyreservoir/matbal/mbal.py:570: RuntimeWarning: invalid value
encountered in true_divide
  x2, y2 = Np, F / (Eg + Boi * Efw)
```



Aquifer Influx Calculation from the Volve Field

The water drive is evident in the Hugin reservoir. We will calculate how many barrels of water encroach the reservoir using one of the models, the Van Everdingen and Hurst model.

```
# import veh function from aquifer
from aquifer import veh

# see help to find the required inputs
help(veh)
```

```
Help on class veh in module aquifer:

class veh(builtins.object)
 |  Methods defined here:
 |
 |  calculate_aquifer(self, datetime, pressure, cf, cw, perm, poro,
mu_w, r_R, B_star)
 |
 |  calculate_aquifer_constant(self, r_R, h, cf, cw, poro)
 |      Calculate theoretical aquifer constant for VEH (assuming
cylindrical reservoir)
 |
 |      Input:
 |      r_R = reservoir radius
 |
 |
----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```


It requires temporal data `datetime`, which is in the Date column. We need to convert it to Pandas datetime.

```
# convert to datetime format
df['Date'] =  pd.to_datetime(df['Date'], format='%Y-%m-%d')
```

First we need to calculate what it's called as the aquifer constant. We will use `calculate_aquifer_constant` function.

```
# input the required ones
r_R = 1.5 * 3281 # reservoir radius, convert km to ft
h = 30 # reservoir thickness, ft
poro = 0.2 # porosity

# calculate aquifer constant (B') in unit RB/D-psi
y = veh()
```

```
B = y.calculate_aquifer_constant(r_R, h, cf, cw, poro)
B
```

```
15465.24600625215
```

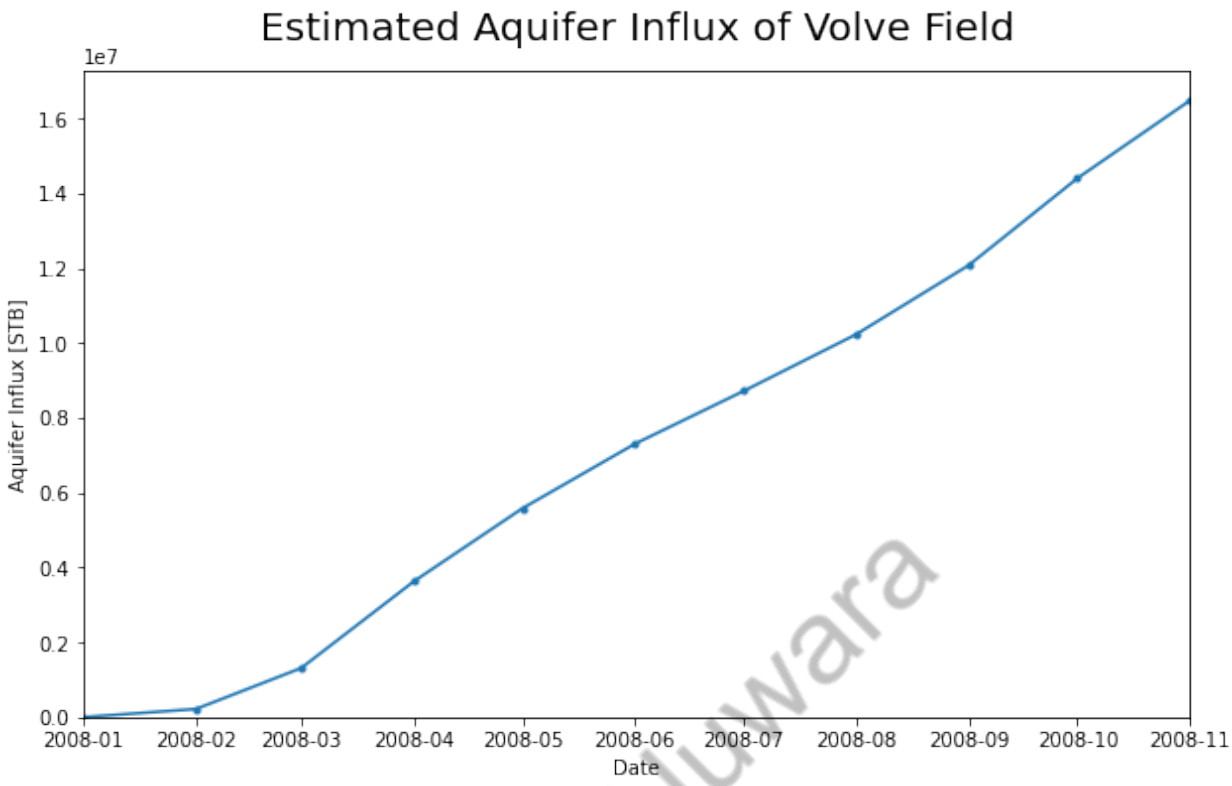
Once we know the aquifer constant, we can calculate the aquifer influx. We will use `calculate_aquifer` function.

```
# input the required ones
datetime = df['Date'].values
perm = 115 # permeability, md
mu_w = 0.318 # water viscosity, cp
B_star = B # aquifer constant, RB/D-psi

We_veh = y.calculate_aquifer(datetime, p, cf, cw, perm, poro, mu_w,
r_R, B_star)
We_veh
```

```
[0.0,
 218423.6750121649,
 1312602.9832122535,
 3623991.36344805,
 5579843.410989863,
 7301926.773786984,
 8713892.343774613,
 10226633.880901858,
 12070882.910508154,
 14396972.092790337,
 16472063.833253667]
```


Finally we plot the aquifer influx.

```
# plot aquifer influx
plt.figure(figsize=(10,6))
plt.plot(datetime, We_veh, '.-')
plt.xlim(min(datetime), max(datetime))
plt.ylim(ymin=0)
plt.xlabel('Date'); plt.ylabel('Aquifer Influx [STB]')
plt.title('Estimated Aquifer Influx of Volve Field', size=20, pad=15)
plt.show()
```



End of the training!

Now we know how to:

- Do simple linear regression
- Read production data
- Visualize production data (pressure, cumulative oil and gas productions) in one plot
- Create material balance plots for different types of reservoirs and estimate for Initial Hydrocarbon in Place:
  - Total oil original in place (OOIP)
  - Total gas original in place (OGIP)
  - Free-phase oil original in place (Nfoi)
  - Free-phase gas original in place (Gfgi)
- Calculate aquifer influx using VEH method

Copyright

Well Test Modeling and Analysis with Python

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

!git clone https://github.com/yohanesnuwara/pyreservoir
```

```
Cloning into 'pyreservoir'...
remote: Enumerating objects: 122, done.ote: Counting objects: 100%
(122/122), done.ote: Compressing objects: 100% (122/122), done.ote:
Total 780 (delta 63), reused 0 (delta 0), pack-reused 658
```

```
import sys
sys.path.append('/content/pyreservoir/welltest')

# import functions for well test modeling and analysis
from wellflo import * # import all functions
from wellanalysis import constant_rate_drawdown_test,
constant_rate_buildup_test
```


Modeling a Multirate Test

We want to model how the pressure transient response of a well is if we conduct a series of rate-changing test (or we call as: multirate test). The following is the series of rate we want to model.

Hours	Rate (STB/D)
$0 < t \leq 10$	1,000
$10 < t \leq 20$	2,000
$20 < t \leq 30$	3,000
$30 < t \leq 45$	1,500
$45 < t \leq 65$	0
$65 < t \leq 70$	1,000

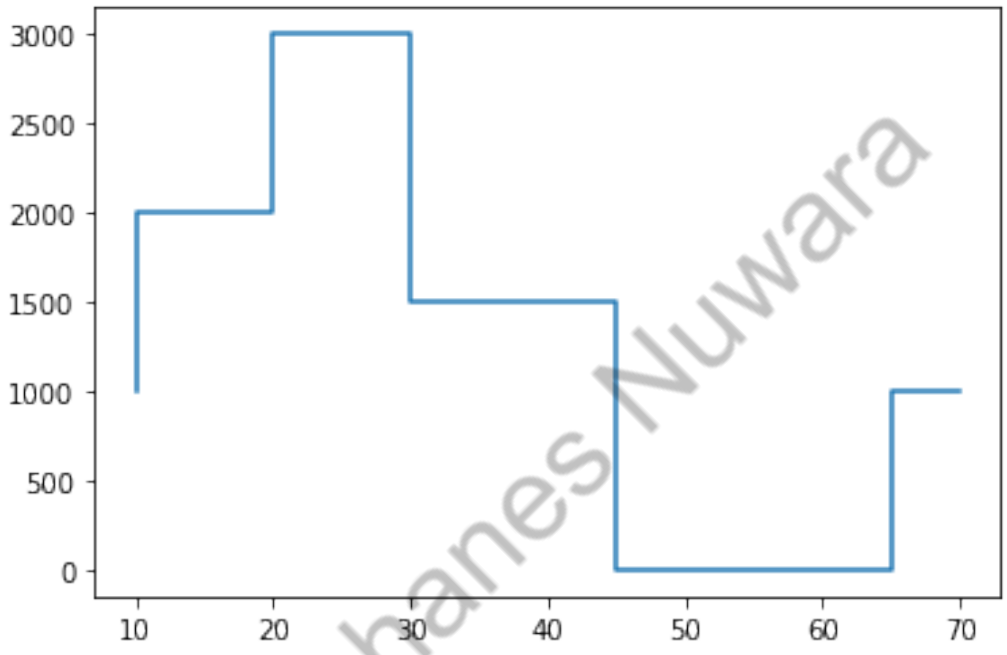
Reservoir rock and fluid properties:

- Reservoir initial pressure 2,500 psia
- Porosity 15%
- Total compressibility 12 microsip
- Permeability 600 md
- Radius of wellbore 4 inch (1 inch = 0.0833 ft)
- Reservoir thickness 32 ft
- Oil viscosity 2 cp

- Distance from wellbore to reservoir outer boundary 3,000 ft
- Oil FVF 1.33 RB/STB

First we want to plot the rates.

```
# plot the rate: use plt.step
t_change = np.array([10, 20, 30, 45, 65, 70])
q_change = np.array([1000, 2000, 3000, 1500, 0, 1000])

plt.step(t_change, q_change)
plt.show()
```




```
# Inputs
poro = 0.15 # Porosity
ct = 12E-6 # Total compressibility, sip
perm = 600 # Permeability, md
rw = 4 * .08333 # Radius of wellbore, convert inch to ft
h = 32 # Reservoir thickness, ft
mu = 2 # Oil viscosity, cp
re = 3000 # Distance from centre of wellbore to outer reservoir
boundary, ft
Bo = 1.333 # Oil FVF, RB/STB

p_initial = 2500 # Initial pressure, psia
```

Before we do modeling, we need to know how much time until the flow behaves finite-acting (or we call as finite-acting time)

```
# we use "time_finite_acting" function. See help first
help(time_finite_acting)
```

```
Help on function time_finite_acting in module wellflo:

time_finite_acting(perm, poro, mu, ct, rw, re)
    Calculate time at flow starts behaving infinite-acting
```

```
# calculate finite-acting time
t_finite = time_finite_acting(perm, poro, mu, ct, rw, re)
t_finite = np.round(t_finite, 3) # round to 3 decimal places

print('Finite-acting time is {} hours'.format(t_finite))
```

```
Finite-acting time is 51.195 hours
```

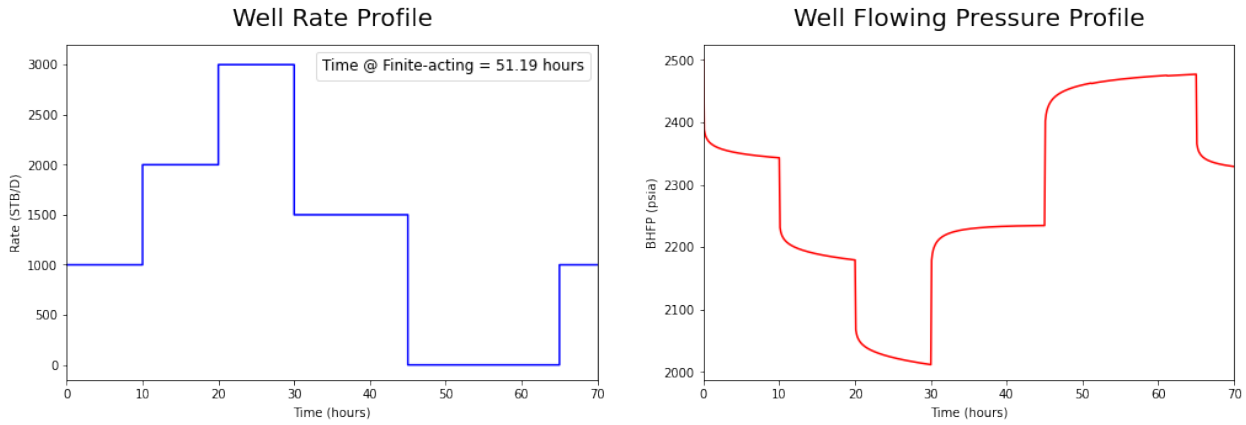
Finally, we do modeling.

```
# we use "simulate_multirate_test" function. See help first
help(simulate_multirate_test)
```

```
Help on function simulate_multirate_test in module wellflo:

simulate_multirate_test(p_initial, t_step, t_change, q_change, re, rw,
perm, poro, mu, ct, Bo, h)
    Simulate the Multiple Constant Rate Test Started from 0th Hour
    Based on Superposition Principle
```

We need to specify the time steps. Smaller timestep will result a better result, but requires more time to compute. Here we'll use timestep of 0.1 hour, that is enough.

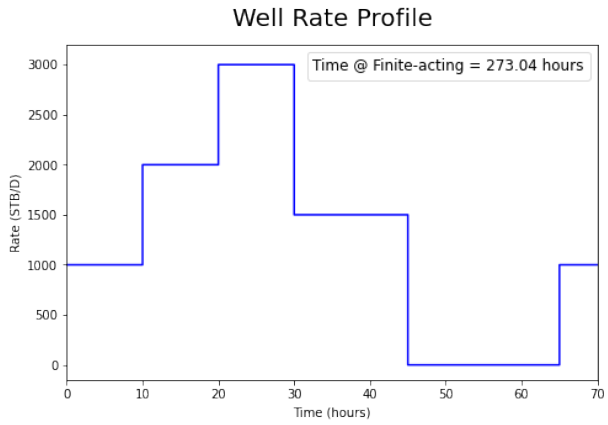
```
# specify time step
t_step = 0.1

simulate_multirate_test(p_initial, t_step, t_change, q_change,
                        re, rw, perm, poro, mu, ct, Bo, h)
```

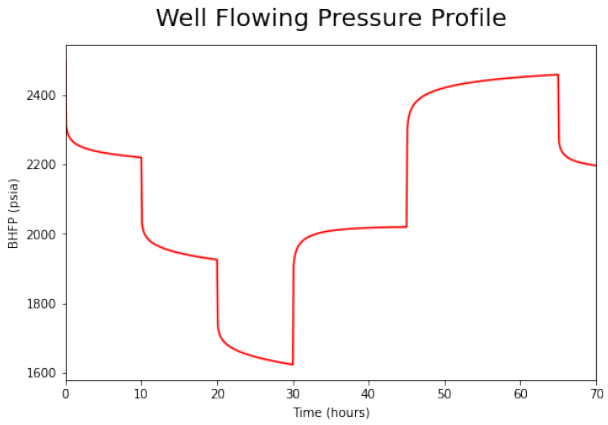


Now, change the porosity, permeability, compressibility, viscosity, reservoir size, and oil FVF. See their effects on the finite-acting time and the pressure transient profile. Make the following cell your playground!

```
# Inputs
poro = 0.4 # Porosity
ct = 12E-6 # Total compressibility, sip
perm = 300 # Permeability, md
rw = 4 * .08333 # Radius of wellbore, convert inch to ft
h = 32 # Reservoir thickness, ft
mu = 2 # Oil viscosity, cp
re = 3000 # Distance from centre of wellbore to outer reservoir
boundary, ft
Bo = 1.33 # Oil FVF, RB/STB

p_initial = 2500 # Initial pressure, psia

# calculate finite-acting time
t_finite = time_finite_acting(perm, poro, mu, ct, rw, re)
t_finite = np.round(t_finite, 3) # round to 3 decimal places

print('Finite-acting time is {} hours'.format(t_finite))

# specify time step
t_step = 0.1

simulate_multirate_test(p_initial, t_step, t_change, q_change,
                        re, rw, perm, poro, mu, ct, Bo, h)
```

```
Finite-acting time is 273.038 hours
```





Modeling a Multiple Pressure Test

We want to model how the rate transient response of a well is if we conduct a series of pressure-changing test (or we call as: multiple pressure test). The following is the series of rate we want to model.

Hours	BHFP (psia)
$0 < t \leq 20$	1,500
$10 < t \leq 40$	1,200
$20 < t \leq 60$	1,000
$30 < t \leq 80$	1,400
$45 < t \leq 100$	1,600
$65 < t \leq 120$	1,000

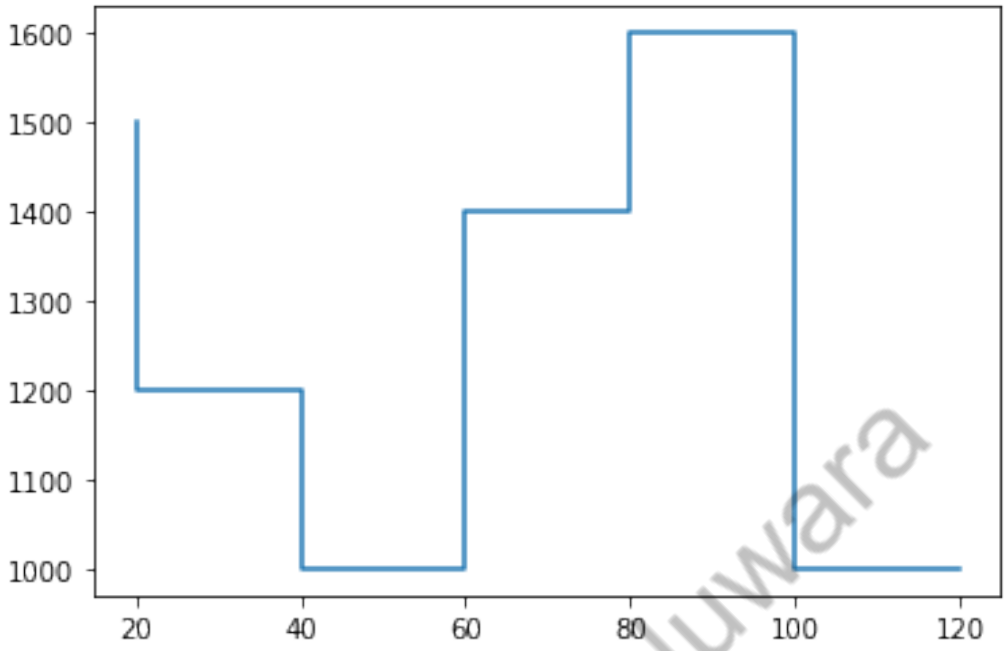
We use the same property as before. Reservoir rock and fluid properties:

- Reservoir initial pressure 2,500 psia
- Porosity 15%
- Total compressibility 12 microsip
- Permeability 600 md
- Radius of wellbore 4 inch (1 inch = 0.0833 ft)
- Reservoir thickness 32 ft
- Oil viscosity 2 cp
- Distance from wellbore to reservoir outer boundary 3,000 ft
- Oil FVF 1.33 RB/STB

First we want to plot the BHFP (borehole flowing pressure)

```
# plot the pressure: use plt.step
t_change = np.array([20, 40, 60, 80, 100, 120])
p_change = np.array([1500, 1200, 1000, 1400, 1600, 1000])
```

```
plt.step(t_change, p_change)
plt.show()
```




```
# Inputs
poro = 0.15 # Porosity
ct = 12E-6 # Total compressibility, sip
perm = 600 # Permeability, md
rw = 4 * .08333 # Radius of wellbore, convert inch to ft
h = 32 # Reservoir thickness, ft
mu = 2 # Oil viscosity, cp
re = 3000 # Distance from centre of wellbore to outer reservoir
boundary, ft
Bo = 1.333 # Oil FVF, RB/STB

p_initial = 2500 # Initial pressure, psia
```

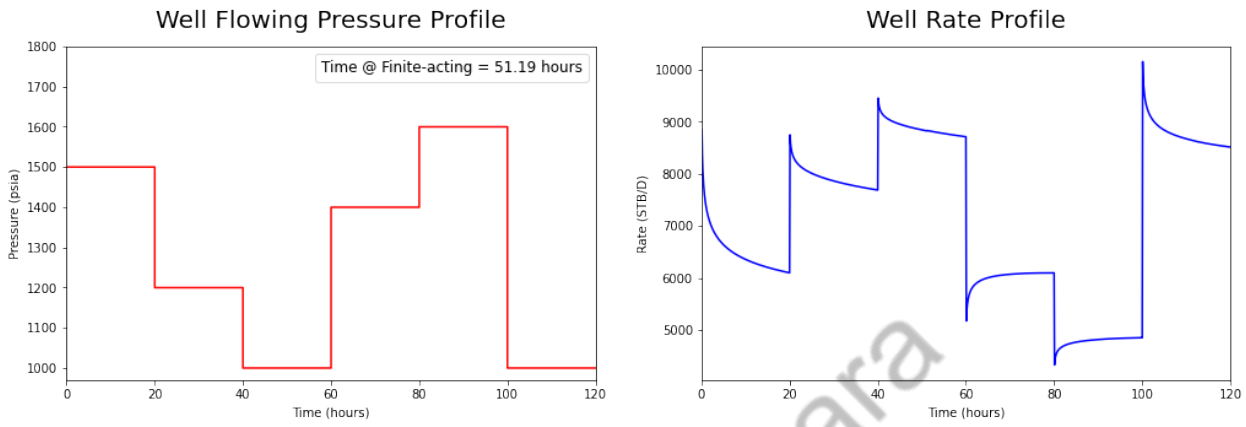
Calculate finite-acting time.

```
# calculate finite-acting time
t_finite = time_finite_acting(perm, poro, mu, ct, rw, re)
t_finite = np.round(t_finite, 3) # round to 3 decimal places

print('Finite-acting time is {} hours'.format(t_finite))
```

```
Finite-acting time is 51.195 hours
```

Specify the timestep and do modeling.

```
# specify time step
t_step = 0.1

simulate_multipressure_test(p_initial, t_step, t_change, p_change,
                            re, rw, perm, poro, mu, ct, Bo, h)
```



Now, change the porosity, permeability, compressibility, viscosity, reservoir size, and oil FVF. See their effects on the finite-acting time and the rate transient profile. Make the following cell your playground!

```
# Inputs
poro = 0.4 # Porosity
ct = 12E-6 # Total compressibility, sip
perm = 300 # Permeability, md
rw = 4 * .08333 # Radius of wellbore, convert inch to ft
h = 32 # Reservoir thickness, ft
mu = 2 # Oil viscosity, cp
re = 3000 # Distance from centre of wellbore to outer reservoir
boundary, ft
Bo = 1.33 # Oil FVF, RB/STB

p_initial = 2500 # Initial pressure, psia

# calculate finite-acting time
t_finite = time_finite_acting(perm, poro, mu, ct, rw, re)
t_finite = np.round(t_finite, 3) # round to 3 decimal places

print('Finite-acting time is {} hours'.format(t_finite))

# specify time step
t_step = 0.1

simulate_multipressure_test(p_initial, t_step, t_change, p_change,
                            re, rw, perm, poro, mu, ct, Bo, h)

Finite-acting time is 273.038 hours
```