COMPUTER IMPLEMENTATION OF THE DYKSTRA-PARSONS METHOD OF WATERFLOOD CALCULATIONS

A Thesis Submitted to the Department of
Petroleum Engineering

African University of Science and Technology Abuja, Nigeria

In Partial Fulfilment of the Requirements for the Degree of

Master of Science in Petroleum Engineering.

By

Lekia Prosper Kiisi

December 2020.

CERTIFICATION

This is to certify that the thesis titled “Computer Implementation of the Dykstra-Parsons Method of Waterflood Calculations” submitted to the school of postgraduate studies, African University of Science and Technology (AUST), Abuja, Nigeria for the award of the Masters’ degree is a record of original research carried out by Lekia Prosper Kiisi in the Department of Petroleum Engineering.

COMPUTER IMPLEMENTATION OF THE DYKSTRA-PARSONS METHOD OF WATERFLOOD CALCULATIONS

By

Lekia Prosper Kiisi

A THESIS APPROVED BY THE PETROLEUM ENGINEERING DEPARTMENT

RECOMMENDED: ----------

Supervisor, Prof. Ikiensikimama Sunday

The name of the first co-supervisor

The second co-supervisor

Head, Department of Petroleum Engineering

October 23, 2021

APPROVED:

Chief Academic Officer

August 24th, 2021

Date

COPYRIGHT PAGE

ABSTRACT

The Dykstra-Parson techniques stands as the most widely used waterflood methods. The authors developed a discrete, analytical solution from which waterflood performance parameters were determined. Reznik et al extended the work of Dykstra and Parson to include exact, analytical, continuous solutions, with explicit solutions for time, constant injection pressure, constant overall injection rate conditions, assuming piston-like displacement.

This work presents a computer implementation to compare the results of Dykstra and Parson method, and the Reznik et al extension. A user-friendly graphical user interface executable application has been developed for both methods using Python 3 and Tkinter, to serve as a hands-on tool for petroleum engineers and the industry.

The results of the program for both methods gave a close match with that obtained from simulation performed with Flow (Open Porous Media). The results provided more insight into the underlying principles and applications of the methods.

Keywords and Phrases: Waterflood, Dykstra-Parson, Computer, Python, Enhanced oil recovery

To the unfailing God

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Prof. Sunday S.Ikiensikimama for the continuous support duringmy research and the writing of my thesis.

Special thanks to the rest of my faculties: Prof. David Ogbe, Dr. Alpheus Igbokoyi the head of department, Prof Djebbar Tiab and Prof Wumi Iledare for their continuous education, and the high level of impact they have painstakingly brought to my life.

Warmest regards to my colleagues for the memorable time spent with them and the academic contributions and support they gave to me during my stay at the African University of Science and Technology, especially my partner and friend Uzoekwe Paul, and my room-mate Bonaventure Odeke for their stimulating discussions, Onyinye Chizobam, and my wonderful class representative, Solomon Israel for his self-less leadership.

Last but not the least, I would like to thank my family: my parents and siblings for supporting me spiritually, financially, and emotionally throughout the period of writing this thesis and my life in general.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

a= parameter defined by Eq 2.47,dimensionless

Av = vertical coverage, dimensionless

b= parameter defined by Eq 2.48, dimensionless

c= parameter defined by Eq 2.49, dimensionless

D= parameter defined by Eq 2.61,dimensionless

f= instantaneous fluid cut, *RB/RB*[res m3 /res m3]

F wop = water-to-oil ratio (WOR), *RB/RB*[res *m 3 /res* m3]

h= bed thickness, ft [m]

k = bed permeability, md

kr = relative permeability, dimensionless

L= bed length, ft [m]

n = number of beds

N = number of calculated points betweenbreakthroughs (Reznik et al, 1984), Total number of beds in system (Dykstra-Parsons)

NPR = cumulative oil recovered, res bbl [res m3]

O= bed-ordering parameter, *cp/md* [Pa' *s/md]*

p = pressure, psia [kPa]

q= injection or production rate, *RB/D*[res *m3 /d]*

S= fluid saturation, dimensionless

Sor = residual oil saturation, dimensionless

S*wi* = irreducible water saturation, dimensionless

t= real or process time, days

v= superficial velocity, *ft/D [mid]*

W= cumulative injected water, **RB** [res m3]

x= streamline coordinate, ft [m]

y= bed width, ft [m]

z= point of flood-front coincidence

{3 = fluid formation volume factor, *RB/STB*[res *m3 /std* m3]

Aj= reservoir fluid mobility, *md/cp (md/Pa'* s)

Ar = reservoir mobility ratio, dimensionless

f1- = fluid viscosity, cp [Pa' s]

r= bed property time, days

Φ = porosity, dimensionless

Subscripts

BT= breakthrough point

i,j,o= bed indices with range *L.n*

n = bed with largest value of O, slowest bed (Reznik et al, 1984), bed just breaking through (Dykstra-Parsons)

o= oil

R= arbitrary reference point

τ = identification of a cumulative quantity attime *t*

T= total

w= water

α= fastest bed not watered-out at time *t*

Chapter 1: Introduction

1.1 Introduction

The oil and gas industry has been phenomena to the determination of the world's economy. World's proven reserve amounts to 1.65 trillion barrels as of 2016. Proven reserve is the amount of oil that can be recovered under current and available technology, economically, at a specified date. Without much ado, it is recovered oil that is valued. And all effort is put forward technologically and economically, in getting the oil out of the subsurface at the highest possible efficiency.

Oil recovery, under available research is classified into three. Basically, one can account them chronologically, but this is not usually the case. Primary recovery being the first on the list, employs the natural energy of the reservoir for oil displacement to producing wells. The natural energy sources include the solution gas drive, gas-cap drive, natural water drive, fluid and rock expansion, and gravity drainage. These are fundamentally termed drive mechanisms. The recovery efficiency of primary recovery is quite limited to about 20% of the original oil in place. In the presence of improved technology and economy, further recovery is achieved by considering secondary recovery mechanisms.

Secondary recovery augments the reservoir natural energy through the injection of water or gas to displace oil towards the producing wells. This is where waterflooding as a reservoir recovery technique stands eminent. And because of its predominant use, waterflooding is majorly in consideration when secondary recovery is mentioned. Its greatest advantage has been its low cost, ease of applicability, accessibility, and availability of water as the displacing fluid. This process should be clearly distinguished from water injection for pressure maintenance, whose objective is to abate the depletion of reservoir energy.

Typically, recovery by waterflooding after primary recovery is between 35 to 50%. Recovering the remaining oil, introduces enhanced oil recovery. The enhanced oil recovery

technique embodies recovery mechanisms and methods that tend to alter rock and fluid properties to improve overall displacement efficiency. This has been classified into; mobility-control such as polymer flooding, chemical, miscible, thermal, and other processes such as microbial enhanced oil recovery. The application of any of these oil recovery techniques is in consideration of several factors. Enhanced oil recovery can take the place of primary recovery at the inception of the recovery process in reservoirs with very dense oil that could not be produced by natural reservoir energy except through thermal recovery. This is also true for secondary recovery, when the need to move from primary recovery to enhanced oil recovery is essential economically, and in getting a satisfactory recovery.

The timeline of the theories and contributions to the development of waterflooding principles and calculations, reveal one of the earliest contributors, Buckley, S. E. et al (1942), who proposed one of the profound foundations of waterflood performance determination based on the frontal advancement theorem., with the consideration of the following assumptions; flow is linear and horizontal, water is injected into an oil reservoir, oil and water are both incompressible, oil and water are immiscible, gravity and capillary pressure effects are negligible, steady state flood conditions, diffuse flow, fluid saturations are uniformly distributed, the areal sweep efficiency at breakthrough is given or estimated from a laboratory-derived correlation, areal sweep efficiency does not change after breakthrough. After this, Stiles W. E. (1949), developed a solution that assumed mobility ratio as unity, piston-like displacement, all beds having the same porosity and the same relative permeabilities to water behind the flood and to oil at the un-swept zone. A year after this, Dykstra H. et al (1950) classical work on stratified reservoirs was published, but unlike Stile’s work, their work was able to incorporate all mobility ratios, and permeabilities were arranged in descending order. Their model was a semi-empirical, and statistical model, that assumed the reservoir to consists of isolated layers of uniform permeability with no cross flow between layers and retained the Piston-like displacement of Stiles W. E (1949); that is only one phase is flowing in any given volume element. It also assumed the flow to be linear, the fluids to be incompressible; that is there are no transient pressure effects, and the pressure drop across every layer to be the same.

In 1955 Craig-Geffen-Morse Method was developed. It Combines the Buckley-Leverett-Welge and Dykstra-Parsons's approach and considers the expanding areal sweep efficiency after breakthrough. Utilizes a modified Welge H. J. (1952) equation to consider the displacement mechanism in the swept area, but heavily depends on laboratory data. The following four stages were proposed by their method.

- Initialization to cusping of the waterfront.
- Cusping to fill-up.
- Fill-up to breakthrough
- Breakthrough to flood-out

Johnson C. E. (1956) went ahead to prepare a graphical representation of Dykstra-Parson analytical solution. This served as a great tool for engineers working with the Dykstra and Parsons method without the aid of computers. Warren J. F et al (1964) also made a great addition, considering the effect of crossflow on oil recovery from stratified reservoirs, with assumptions permitting viscous forces, but neglecting capillary and gravity forces. Reznik et al (1984) extended Dykstra-Parsons method to real-time exact analytically continuous solutions. We shall look at their work in greater detail in subsequent sections. El-Khatib (1985), applied the Dykstra-Parsons analytical solutions to completely communicating reservoir layers, and Tiab D (1986) extended the Dykstra-Parsons method to layered-composite reservoirs. Just like the work of Johnson (1956), Enick et al (1988), presented a graphical representation of the work of Reznik et al (1984)

To commensurate the progress made so far, Mahfoudhl et al (1990), applied the Dykstra-Parsons technique to polymer flooding. The sequence of fluid injection is as follows:

- Waterflood
- Polymer slug
- Drive fluid (brine)

In their work, all assumptions of Dykstra-Parsons were considered, with communication between beds existing only at the injection and production surfaces, reduction of the absolute permeability of the bed behind the front to account for permeability reduction due to polymer adsorption, and no account for polymer

degradation, inaccessible pore volume, visco-elastic effect, polymer loss from retention. Their theory assumed unsteady fluid flow, and no diffusion between polymer and drive water or injected water.

In this work, focus will be on the theories of waterflooding, which could also be applicable in enhanced oil recovery applications such as polymer flooding, or other chemical immiscible displacement technique. The theoretical advances of these theories, and its foregoing applications even in enhanced oil recovery, will be discussed in the adjoining chapter. But this will be with greater focus on Dykstra-Parsons waterflood performance prediction method, decorated with an encompassing literature review.

1.1 Aim and Objectives.

The aim of this study is to develop a computer program to implement the Dykstra-Parsons Waterflood performance technique.

The objectives of the study include:

1. Writing a computer program implementing the Dykstra and Parson waterflood method, and the Reznik et al (1984) extension to exact, continuous solutions using Python 3.0.
2. Integrating the computer program with a graphical user interface.
3. Converting the resultant computer program to a desktop application.
4. Validating the result of the computer program using the Open Porous Media (OPM Flow) simulation software.

Chapter 1: Theories in Waterflood Performance Evaluation

1.2 Introduction

Waterflooding is fundamentally the oldest, most important secondary recovery technique used in the industry. It can be employed as a means of reservoir maintenance and as a secondary recovery technique in a depleted reservoir. There are several considerations to an effective waterflood project design. These factors include the recovery efficiencies, rock and fluid properties, flood patterns, water injection rates, and reservoir pressure. But not necessarily limited to these. Waterflooding is simply the process of injecting water into the reservoir through injection wells. The water drives the oil through the formation towards the production wells. For efficient performance, and sweep efficiency improvement, the water is injected in patterns, which is essentially the arrangement of injectors and producers, in the reservoir.

The common waterflood patterns include:

a. The direct-line drive
b. The five-spot pattern
c. The nine-spot pattern
d. The staggered-line drive
e. Seven and four-spots pattern
f. Skewed four-spot.
g. Peripheral flooding
h. Line flooding.

The five-spot and nine-spot pattern can either be classified as normal or inverted. The ratio of producing wells to injection wells in the four-spot, skewed four-spot, and inverted seven-spot is two. While the five-spot, direct-line drive, and the staggered line drive have a ratio of one. This ratio for the seven-spot, nine-spot, and inverted nine-spot is one-half, one-third, and three, respectively. The corresponding areal efficiency for each flood pattern extracted from Singh, S. P. et al. (1982)is given in Table 2.1.

Table 2.1Common waterflood patterns and their breakthrough efficiencies

S/ N	Pattern Type	Ea, At Breakthrough, Fraction
1	Direct line drive	0.570
2	Direct line drive	0.706
3	Five Spot	0.723
4	Seven Spot	0.740
5	Staggered line-drive	0.800

1.2.1 Sweep Efficiencies.

It has been shown that knowing the oil in place, the areal efficiency, displacement efficiency, and the vertical sweep efficiency, one can determine the determine the cumulative recovery at any time using Equation 2.1.

$N_p = N\left(E¿¿A \times E_D \times E_v\right)¿$(2.1)

Where

N = oil in place in the pore volume

E_A = the areal sweep efficiency.

E_D = displacement efficiency

E_V = Vertical sweep efficiency.

The product of the areal and vertical sweep efficiency gives the **volumetric sweep efficiency**.

While the overall recovery factor or efficiency is the product of the areal sweep efficiency, vertical sweep efficiency and the displacement sweep efficiency as expressed in Equation 2.2.

$$RF = E_A \times E_D \times E_v \qquad (2.2)$$

Areal sweep efficiency is defined as the reservoir area fraction contacted by the displacing fluid during the waterflooding operation. The areal sweep efficiency is a function of the relative flow properties of oil and water, the well pattern, pressure distribution and the directional permeability.

The vertical sweep efficiency is the fraction of the formation in the vertical plane which injected water for waterflood operation will contact. It depends primarily on the vertical stratification of the reservoir. Figure 2.1 is a representation of areal and vertical sweep of waterflood in a stratified reservoir, showing the swept and the un-swept zone.

The displacement sweep efficiency represents that portion of movable oil displaced by the displacing fluid, expressed mathematically in Equation 2.3 - 2.8.

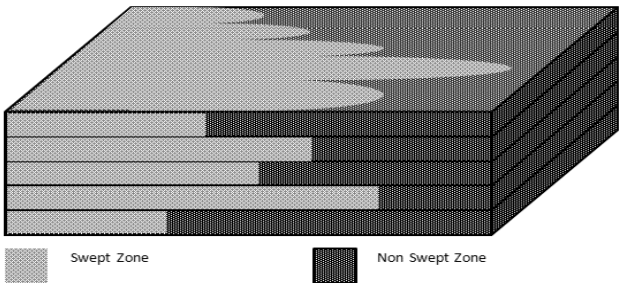

Figure 21: 3-Dimensional view of vertical and areal sweep efficiency

$$E_D = \frac{Change \in oil\ saturation \in the\ water\ swept\ zone}{Initial\ oil\ saturation\ at\ start\ of\ waterflood} \tag{2.3}$$

$$E_D = \frac{volume\ of\ oil\ at\ start\ of\ flood - remaining\ oil\ volume}{volume\ of\ oil\ at\ start\ of\ flood} \tag{2.4}$$

Where

S_{oi} = Initial oil saturation

$$Soi = 1\text{-}Swc - Sgi \tag{2.5}$$

And

$$\acute{S}_o = 1 - \acute{S}_w \tag{2.6}$$

Therefore,

$$E_D = \frac{(\frac{Soi}{Boi}) - (\frac{\acute{S}_o}{Bo})}{(Soi/Boi)} \tag{2.7}$$

The areal sweep efficiency can also be analytically determined without the use of chart from the set of equation shown in Equation 2.8 – 2.11.

$$E_A = E_{ABT} + 0.2749 \ln(\frac{W_{inj}}{W_{iBT}}) \tag{2.8}$$

$$E_{ABT} = 0.54602036 + \frac{0.03170817}{M} + \frac{0.30222997}{e^M} - 0.00509693\,M \tag{2.9}$$

$$W_{iBT} = E_{ABT} \text{ per pore volume} \quad (2.10)$$

$$W_{inj} = \frac{E_{ABT}}{E_D} \text{ per pore volume} \quad (2.11)$$

1.2.2 Mobility Ratio

According to Tiab(1986), it is also found that waterflooding performance in layered composite reservoirs is essentially controlled by the mobility ratio.

Mobility is defined as the ease of flow of fluid. It relates the effective permeability of a fluid with its viscosity.

Mobility is expressed as in Equation 2.12,

$$\lambda_{fluid} = \left(\frac{k}{\mu}\right)_{fluid} \quad (2.12)$$

The mobility ratio is now defined as the ratio of the mobility of the displacing fluid phase to the mobility of the displaced fluid phase.

Therefore, for a waterflood where injected water is displacing oil, the mobility ratio is given in Equation 2.13 and 2.14.

$$\text{M} = \frac{K_w/\mu_w}{K_o/\mu_o} = \frac{K_w \mu_o}{K_o \mu_w} \quad (2.13)$$

In terms of the relative permeability of the fluid, considering that absolute permeability is the same for both fluids, we have the expression in Equation 2.14.

$$\text{M} = \frac{K_{rw} \mu_o}{K_{ro} \mu_w} \quad (2.14)$$

The relative permeability defined above, for water is the relative permeability in the water-contacted zone of the formation, and for oil, the relative permeability is defined for the un-swept oil zone.

A waterflood operation is successful as the areal sweep efficiency is sufficiently high, and this is only possible at favorable mobility ratio (M less than unity). This implies that the mobility of oil is greater than that of water. Where the mobility ratio is less than unity, the velocity of the waterfront in each layer will be impaired as the flood advances, which somewhat stabilizes the flood front. While, the reverse applies to the case where the mobility ratio is greater than unity.

The mobility ratio also influences the fluid injectivity, which is defined as the rate at which fluid can be injected per unit pressure difference between injection and producing wells. At favorable mobility ratios, the fluid injectivity decreases with increase in areal sweep efficiency. While the reverse goes for unfavorable mobility ratios (M greater than unity).

1.3 Theories and Advances in Waterflood Performance Predictions.

Several theories have been proposed to predict waterflood performance. This is quite credited to certain authors and researchers as BuckleyS. E. et al (1942) who were the first to describe and include a saturation distribution using the frontal advancement theorem in immiscible displacement in homogeneous formation. The frontal advance equation incorporates the fractional flow equation in a material balance considering the fraction flow of water at the different sections of the reservoir being invaded by the flooded water. The result is an equation that relates the saturation with position and time at a given fluid injection rate as seen in Equation 2.15. With this equation, one can determine the position of the flood front at any time knowing the fractional flow slope as a function of saturation, which can be obtained from the fractional flow curve.

$$x = \frac{5.615 W_i}{\phi A} \frac{d f_w}{d S_w} \tag{2.15}$$

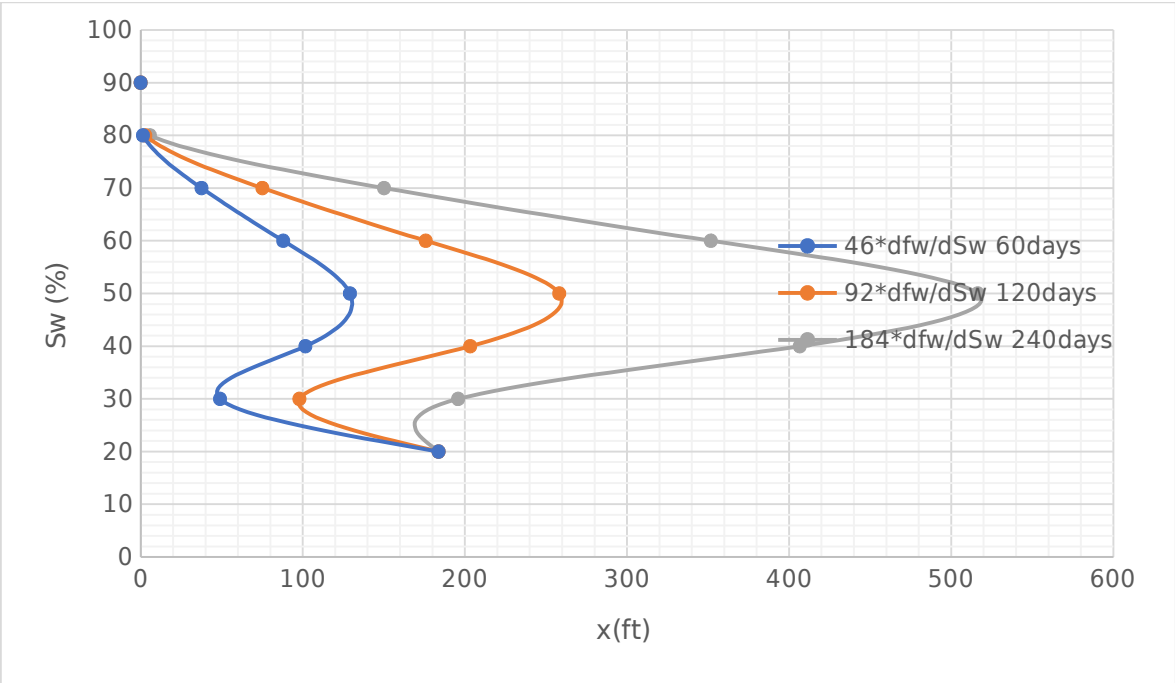

Figure2.2 Waterflood frontal advancement saturation distribution (Source: B. C. CRAFT, M. H. (1991))

As shown from Figure 2.2 it could be observed that more than one saturation can occur at the same location. This is not possible in a practical sense. To resolve this issue, BuckleyS. E. et al. (1942) suggested that a portion of the calculated saturation distribution curve is imaginary, and the real curve contains a discontinuity at the front.

In a waterflood study, the reservoir is sectioned into three zones. Starting at the injection point, is the free water zone, having 100% water saturation, then a transition zone from which oil and water can be produced, and the oil zone with constituent connate water. The oil zone is considered as the un-swept zone, while the water zone is called the swept zone. During water breakthrough at the producer, indicating a complete sweep by the injected water, residual oil is left in the reservoir.

Welge H. J. (1952) proposed an improved method of achieving the same result as BuckleyS. E. et al. (1942), by integrating the saturation distribution from the injection point to the front and obtained the water saturation behind the front expressed in Equation 2.16. But requires that the initial water saturation be uniform. A graphical interpretation of the result show that a line drawn tangent to the fractional flow curve from the initial water saturation (S_{wi}), meets at the flood front (f_{wf}, S_{wf}) as shown in Figure2.3.

$$\left[\frac{df_w}{dS_w}\right]_{S_{wf}} = \frac{\frac{f_w}{S_{wf}} - \frac{f_w}{S_{wi}}}{S_{wf} - S_{wi}} \qquad (2.16)$$

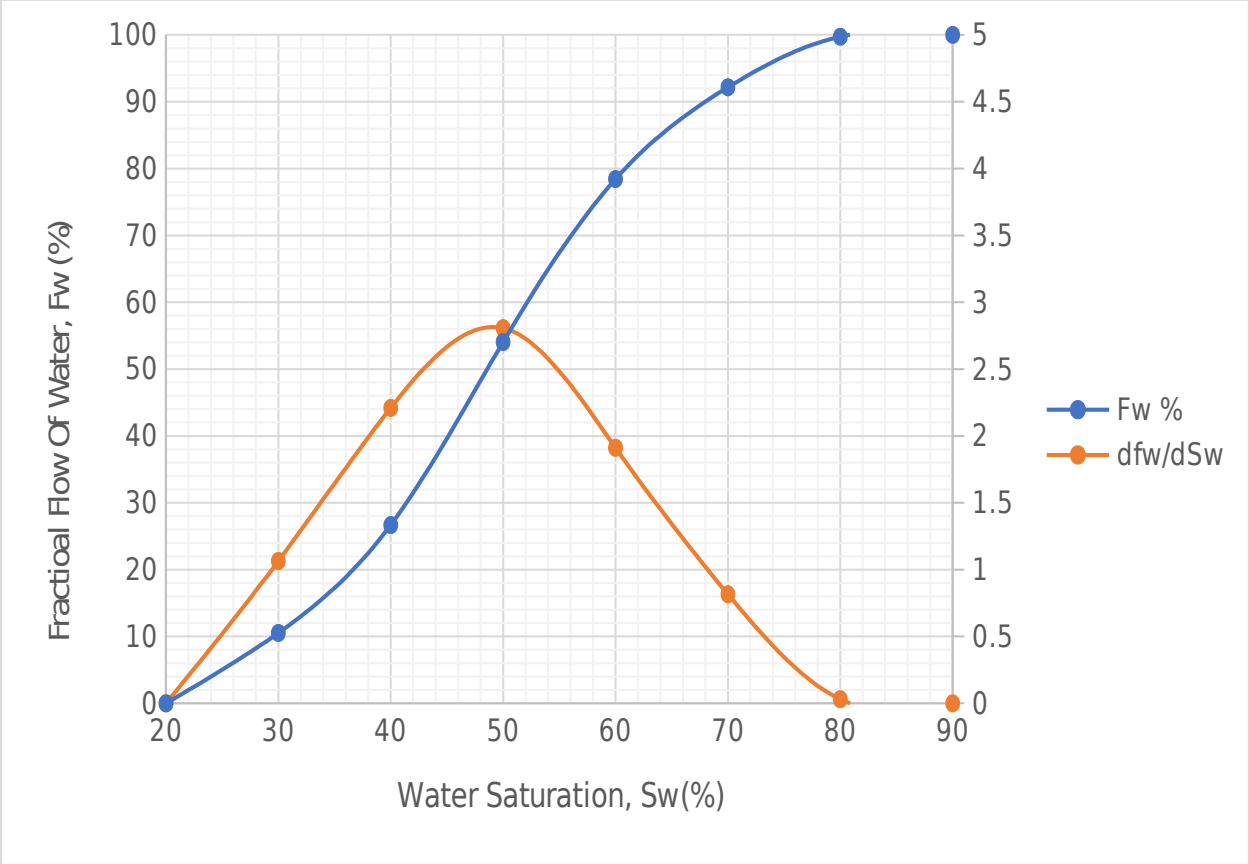

Figure2.3 Fractional flow curve and derivative of fractional flow curve (Source: B. C. Craft, M. H. (1991))

1.4 Dykstra and Parson Water Flood Performance Model

In addition to the analytical solution developed by Dykstra and Parsons, Dykstra and Parsons also formulated a graphical technique for achieving the same result. This method is the best fit for engineers attempting to make these calculations manually. The proposed solution employs a statistical technique, along with a set of coverage charts for several water oil ratios.

In their solution, A vertical permeability variation was determined by making a plot of the percentage of the permeabilities greater than a given value against that permeability on a log-probability paper. The plot results in a straight line. The vertical permeability variation was obtained by dividing the difference between the median permeability and the

permeability at 84.1 cumulative percent by the median permeability. It was found that only this permeability variation is necessary to characterize the permeability distribution. The reason is that the magnitude of the permeability does not count, as the solution is expressed in ratios of permeabilities. By this, it was possible to calculate curves with the coverage expressed as a function of the permeability variation and mobility ratio for a given water-oil ratio. These charts have been successfully extended to water oil ratios of 0.1 to 100.

Johnson. C. E. (1986) predicted oil recovery by water flood using the Dykstra-Parsons method. He presented a graphical approach, shown in Figure 2.4 to determine fractional recovery, given the vertical permeability variation, mobility ratio, and initial water saturation. Vertical permeability variation and initial water saturation, they argued, can be obtained from core studies. And the mobility ratio is a function of fluid properties (relative permeabilities of oil and water, and oil and water viscosities). The oil recovery fraction was determined at different water oil ratios. Four charts were constructed for WOR's of 1,5,25, and 100. The recovery modulus in their work was given by Equation 2.17.

$$\psi = R\left[1 - \frac{S_{wi}}{(WOR)^{0.2}}\right] \qquad (2.17)$$

Where,

R = fractional recovery of the original oil in place.

Swi = initial water saturation.

WOR = producing water-oil ratio.

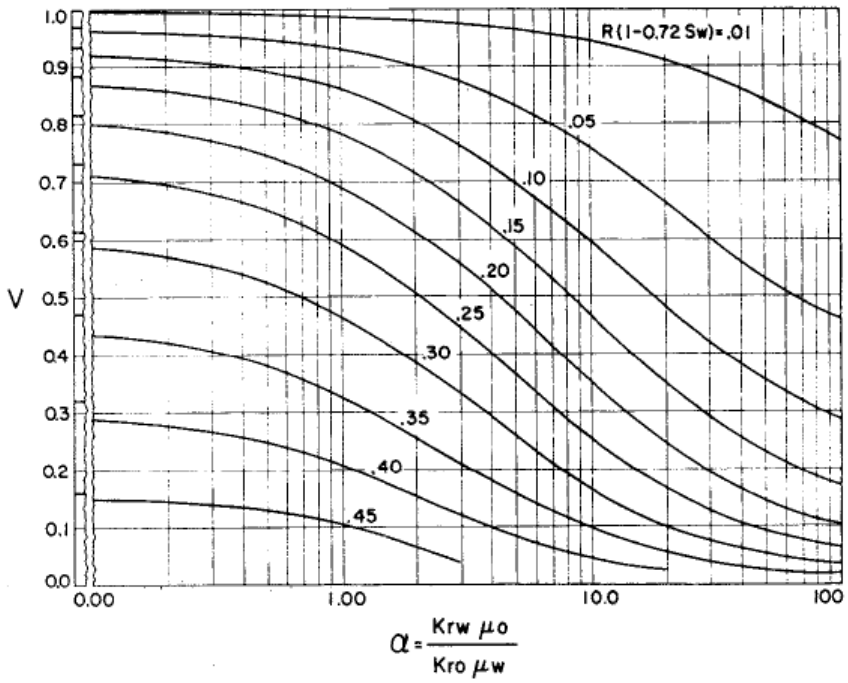

Figure2.4Permeability Variation Plotted Against Mobility Ratio Showing Lines of Constant R(1-0.72Sw) for a Producing Water-Oil Ratio of 5 (Source: Johnson, C. E. (1956))

The Dykstra-Parsons method for calculating oil recovery using coverage charts followed four major steps, outlined below.

Step 1: Assemble permeability data in descending order. Calculate percentage equal to or greater than for each entry.

Step 2: Plot percentage against log permeability on a probability paper. Calculate permeability variation using Equation 2.18.

$$v = \frac{k_{50} - k_{84.1}}{k_{50}} \tag{2.18}$$

Step 3: calculate mobility ratio.

Step 4: obtain coverage, C, from the coverage charts.

Step 5: calculate the cumulative recovery using Equation 2.19.

$$N_p = 7758 \frac{Ah\phi C\left(S_{oi} - S_{i}\right)E_a}{B_o} \tag{2.19}$$

Step 6: Plot Np versus F_{wo}

Step 7: Integrate Np – F_{wo} curve graphically to get Wp.

Step 8: Calculate the cumulative water injected from Equation 2.20.

$$W_i = W_F + N_p B_o + W_p \quad (2.20)$$

Step 9: Time

$$t = \frac{W_i}{i_w} \quad (2.21)$$

Step 10: Calculate oil and water rates from the Equation 2.22 – 2.24.

$$q_o = \frac{i_w}{B_o + F_{wo}} \quad (2.22)$$

$$q_w = q_o F_{wo} \quad (2.23)$$

or

$$q_w = i_w - B_o q_o \quad (2.24)$$

According to C.H. Wu(1988), the Dykstra-Parsons graphical technique had a great deal of appeal to petroleum engineers working on waterflood evaluation. However, the Dykstra-Parsons approach is not limited to graphical applications. The equation can be successful programmed into a computer to complete the calculations without reverting to the use of the charts and graphs.

The following steps are to be employed in this procedure using the equations developed by the authors.

Step 1: Permeability arrangement/ordering

This is the very first step in obtaining a solution to the problem. It entails ordering the permeability in descending order.

Step 2: Calculate the mobility ratio M.

Step 3: Determine the distances the flood has advanced in the j^{th} layer (j > n), when the nth layer has just broken through using Equation 2.25.

When the nth layer has just broken through, all the layers with permeability greater than that of the nth layer should also have broken through. Hence the fraction of the reservoir for which the layers have been completely flooded out is n/N. the remaining layers, which have permeabilities less than the nth layer, will be only partially swept out.

$$\frac{x_j}{L} = \frac{M - \sqrt{M^2 + \frac{K_j}{K_n}(1 - M^2)}}{M - 1} \quad (2.25)$$

Step 4: Calculate the coverage, C_n.

From the calculated front locations, the vertical coverage or recovery factor can be calculated using Equation 2.26. The coverage is defined as the fraction of the reservoir which has been invaded by water. While N is the total number of layers in the system.

$$C_n = \frac{n + \frac{(N-n)M}{M-1} - \frac{1}{M-1}\sum_{n+1}^{N}\left(M^2 + \frac{K_j}{K_n}(1-M^2)\right)^{\frac{1}{2}}}{N} \quad (2.26)$$

Step 5: Compute the water-oil ratio using Equation 2.27 – 2.28.

The water-oil ratio equation is developed on three assumptions.

- After breakthrough, only water is producing from the layer.
- Before breakthrough, only oil is producing from the layer.
- The mobility that exists in the oil zone ahead of the waterfront may be different from that which exists in the water zone behind the waterfront.

$$WOR_n = \frac{\sum_{j=1}^{n} K_j}{\sum_{n+1}^{N} \frac{K_j}{\left(M^2 + \frac{K_j}{K_n}(1-M^2)\right)^{\frac{1}{2}}}} \quad (2.27)$$

The producing water-oil ratio is given by,

$$F_{wo} = B_o \times WOR \quad (2.28)$$

Step 6: Compute the Cumulative oil recovery, N_{pn}, using Equation 2.29 and 2.30.

The cumulative oil recovery is calculated from the coverage and the moveable oil in volume in place at the start of waterflooding.

$$N_{pn} = E_{ABT} V_B \frac{7758\phi(S_{oi} - S_{\text{ċ}})}{B_o} C_n \quad (2.29)$$

Where,

$$V_B = Ah \quad (2.30)$$

Step 7: Compute the cumulative water produced, W_p, using Equation 2.31 and 2.32.

If the F_{wo} is plotted against the recovery on a rectangular coordinate. The area under the curve is the water produced up to the given recovery N_p.

$$F_{wo} = \frac{q_w}{q_o} = \frac{\frac{dW_p}{dt}}{\frac{dN_p}{dt}} = \frac{dW_p}{dN_p} \tag{2.31}$$

$$W_p = \int_0^{N_p} (F_{wo})\, dN_p \tag{2.32}$$

Step 8: Compute the volume of water require to fill-up the gas space, W_f, from Equation 2.33.

$$W_f = 7758 A h \phi (S_{gi} - S_{gr}) \tag{2.33}$$

Step 9: Compute the water injected, W_i, using Equation 2.34.

$$W_i = W_p + B_o N_p + W_f \tag{2.34}$$

Step 10: Compute the time to reach a given recovery, t, using Equation 2.35.

Time,

$$t = \frac{W_i}{i_w} \tag{2.35}$$

Where,

i_w is the water injection rate assumed to be constant.

Mobarak S. (1975) was able to compare the Dykstra-Parson method with a numeric model and obtained a close agreement with both models. The comparison of the numerical model and Dykstra-Parson calculation is seen in Figure 2.5. As has been noted by Reznik et al (1984), some of the disadvantage of complex numerical simulation is that they tend to obscure the various distinct aspects of waterflooding by embedding them in numerical aggregations. Which makes it difficult to distinguish the effects of vertical stratification from those of areal pattern coverage? Also, the nonlinearity of the saturation distribution causes instabilities in numerical models for even in non-stratified reservoirs. They were able to extend the Dykstra-Parson analytical, discrete, stratification model to analytically continuous space-time solutions.

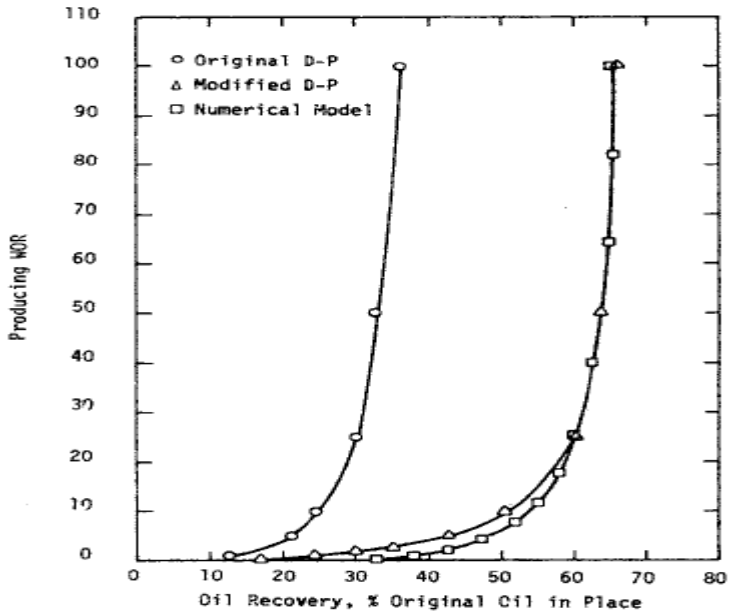

Fig. R-1—Predicted WOR-recovery performance for communicating layers.

Figure2.5– Predicted WOR-recovery Performance for Communicating Layers (Source: Mobarek S. (1975))

1.5 Reznik et al Continuous Solution Model

Reznik et al (1984)presented the following in their work.

- Exact analytical continuous solution.
- Explicit solutions for time.
- Constant injection pressure, and constant overall injection rate conditions.
- Property time
- Real or process time.
- Bed flood-front passing phenomenon
- A trailing zone.
- Piston-like displacement.
- Flow regime of creeping or low Reynold's number.

The following step-by-step procedure can be followed based on the set of equations, conditions and constraints developed by Reznik et al (1984).

Step 1: Determine for each bed the change in water saturation, mobility for water and oil, and mobility ratio using Equation 2.36 – 2.39.

$$(\Delta S_w)_i = 1 - S_{ori} - (S_{wi})_i \quad (2.36)$$

Mobility of water,

$$\lambda_{fwi} = \frac{K_i K_{rwi}}{\mu_{wi}} \quad (2.37)$$

Mobility of oil,

$$\lambda_{foi} = \frac{K_i K_{roi}}{\mu_{oi}} \quad (2.38)$$

Mobility ratio of bed *i*,

$$\lambda_{ri} = \frac{\lambda_{fwi}}{\lambda_{foi}} \quad (2.39)$$

Step 2: Bed ordering from Equation 2.40.

Initiate the bed ordering parameter:

The beds are arranged in increasing/ascending order of O_i

$$O_i = \frac{\phi_i (\Delta S_w)_i (1 + \lambda_{ri})}{\lambda_{fwi}} \quad (2.40)$$

Frequently, all rock and fluid properties, except absolute permeability, are assumed to be independent of bed. Then, Equation (2.40) can be written as

$$O_i = \frac{1}{K_i} \quad (2.41)$$

Step 3: Flood Front Location

- Set point of flood front coincidence between beds *i and j, Z_{ij}, as shown in Equation 2.42*.

  Where *Oi* < *Oj*

  $$Z_{ij} = \frac{2\left[O_j \lambda_{rj}(1 + \lambda_{ri}) - O_i \lambda_{ri}(1 + \lambda_{rj})\right]}{\left[O_i(1 - \lambda_{ri})(1 + \lambda_{rj}) - O_j(1 - \lambda_{rj})(1 + \lambda_{ri})\right]} \quad (2.42)$$

  It should be noted that in each reservoir, there exist n(n-1)/2 Z_{ij} between beds. This implies for instance in a reservoir with five beds, there will exist point of coincidence between bed layers as shown in Figure 2.6.

1 – 2 1 – 3 1 – 4 1 - 5	2 – 3 2 – 4 2 – 5	3 – 4 3 – 5	4 – 5

Figure 2.6 Flood Front Point of Coincidence

Condition:

$$1 < \frac{O_i}{O_j} > \frac{\lambda_{rj}(1+\lambda_{ri})}{\lambda_{ri}(1+\lambda_{rj})} \tag{2.43}$$

- Setting condition limits

  If Equation (2.43)in the condition above is not satisfied, then.

$$\frac{x_i}{L} > \frac{x_j}{L} \tag{2.44}$$

  If Equation (2.43) the condition above is satisfied, then.

  i. *for*

$$\frac{x_i}{L} \le Z_{ij}, \frac{x_i}{L} < \frac{x_j}{L} \tag{2.45}$$

  thereafter,

  ii. $$for \frac{x_i}{L} > Z_{ij}, \frac{x_i}{L} > \frac{x_j}{L} \tag{2.46}$$

- Compute the flood front location.

  i. For $0 < \lambda_{rj} < 1$

$$a_j = \lambda_{rj}^{\ 2} \tag{2.47}$$

$$b_j = \frac{O_n}{O_j}\left(\frac{2\lambda_{rn}}{1+\lambda_{rn}}\right)\left(1-\lambda_{rj}^{\ 2}\right) \tag{2.48}$$

$$c_j = \frac{O_n}{O_j}\left(\frac{1-\lambda_{rn}}{1+\lambda_{rn}}\right)\left(1-\lambda_{rj}^{\ 2}\right) \tag{2.49}$$

$$\frac{x_j}{L} = \frac{\left[-\lambda_{rj} + \sqrt{a_j + b_j\left(\frac{x_n}{L}\right) + c_j\left(\frac{x_n}{L}\right)^2}\right]}{\left(1-\lambda_{rj}\right)}$$

(2.50)

ii. For $\lambda_{rj}=1$

$$\frac{x_j}{L}=\frac{\left\{\frac{O_n}{O_j}\left[\left(1-\lambda_{rn}\right)\left(\frac{x_n}{L}\right)^2+2\lambda_{rn}\left(\frac{x_n}{L}\right)\right]\right\}}{\left(1+\lambda_{rn}\right)} \tag{2.51}$$

Average mobility of the fluids in bed *i* at time t, is given in Equation 2.52.

$$\lambda_{fi}=\frac{\lambda_{fwi}}{\lambda_{ri}+\left(1-\lambda_{ri}\right)\frac{x_i}{L}} \tag{2.52}$$

The use of bed n is more convenient in as much as it is the only bed with a continuously increasing flood-front value. It provides a continuous basis for the generation of all the bed flood-front positions over a finite time.

Since the variable x_n/L is continuous, we can set a continuous distribution over it with range between $0<x_n/L<1$.

Step 4: Determine the property time using Equation 2.53.

The property time is defined as the time required for the advancement of the flood front in a single bed, isolated from the multi-bed system, when under the influence of an overall pressure that is invariant with time.

Property time τ, is identical to real or process time, t, for the case of constant injection pressure only.

$$\tau_i=\frac{L^2}{\Delta P}\frac{\phi_i\left(\Delta S_w\right)_i}{\lambda_{fwi}}\left[\lambda_{ri}\left(\frac{x_i}{L}\right)+\frac{1}{2}\left(1-\lambda_{ri}\right)\left(\frac{x_i}{L}\right)^2\right] \tag{2.53}$$

Step 5: Determine the Real or Process time, following Equation 2.54 – 2.56.

To determine the process time, we consider it differently for both the constant injection pressure (CIP case), and the constant overall injection rate (CIR case).

It should be noted that for the CIR case, the overall pressure gradient, at time t, can be determined by

$$\frac{\Delta P}{L}=\frac{q_T}{y\sum_{j=1}^{n}\left(h_j\lambda_{fj}\right)} \tag{2.54}$$

In the manner, for the CIP case the total injection rate of water, at time t is given by.

$$q_T = y\left(\frac{\Delta P}{L}\right)\sum_{j=1}^{n}\left(h_j \lambda_{fj}\right) \tag{2.55}$$

- CIP case.

  Since the property and process time are identical for the CIP case, we can rewrite Equation (2.53) with just replacing *i* with *n*.

$$t = \frac{L^2}{\Delta P}\frac{\phi_n (\Delta S_w)_n}{\lambda_{fwn}}\left[\lambda_{rn}\left(\frac{x_n}{L}\right) + \frac{1}{2}\left(1-\lambda_{rn}\right)\left(\frac{x_n}{L}\right)^2\right] \tag{2.56}$$

- CIR case.

  i. Determine the linear velocity of the flood front in bed *i*.

$$\frac{dx_i}{dt} = \frac{\Delta P \lambda_{fi}}{L\phi_i\left(\Delta S_w\right)_i} \tag{2.57}$$

  ii. Determine the instantaneous volumetric flow rate of water into bed *i*.

$$q_{wi} = y\left(\frac{\Delta P}{L}\right)h_i \lambda_{fwi} \tag{2.58}$$

  iii. Determine the overall injection rate.

  q_T is constant for all t, which defines the CIR constraint.

$$q_T = \sum_{j=1}^{n} q_j \tag{2.59}$$

  iv. Determine the ultimate recoverable oil in bed.

$$N_{PRj} = yL h_j \phi_j \left(\Delta S_w\right)_j \tag{2.60}$$

  v. Defining the dynamic bed, α

  The dynamic bed α is defined as the bed with the lowest value of O_j that has experienced water breakthrough at time t. Or better still it is last bed to or the bed that just experience water breakthrough. A representation of the dynamic bed concept is shown in Figure 2.7.

  Therefore, xj/L =1, for $1 \le j \le (\alpha - 1)$, and

  xj/L< 1, for for $\alpha \le j \le n$

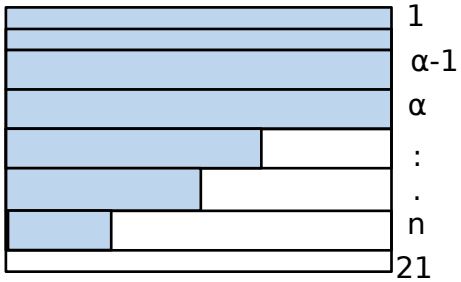

Figure 2.7 Dynamic Bed Representation

For a given value of x_n/L, one first identifies α from values of $(x_n/L)_j$. To do this, one can write the program to return the last $x_j/L = 1$. The corresponding bed will give the value of α. Determine the parameter,

$$D_j = b_j\left[\left(\frac{x_n}{L}\right) - \left(\frac{x_n}{L}\right)_{\alpha-1}\right] + c_j\left[\left(\frac{x_n^{\,2}}{L}\right) - \left(\frac{x_n^{\,2}}{L}\right)_{\alpha-1}\right] \tag{2.61}$$

$$t - t_{BT\alpha-1} = \frac{1}{q_T}\left(\sum_{j=1}^{\alpha-1}\left[\frac{PRj}{(1-\lambda_{rj})}\frac{D_j}{2}\right] + \sum_{j=\alpha}^{n}\left\{N_{PRj}\left[\left(\frac{x_j}{L}\right) - \left(\frac{x_j}{L}\right)_{\alpha-1}\right]\right\}\right) \tag{2.62}$$

Step 6: Determine the instantaneous production values.

- Determine the instantaneous oil production rate in bed *j* at $x_j = L$ at time t

$$q_{oj} = y\,h_i\left(\frac{\Delta P}{L}\right)\frac{\lambda_{fwj}}{\left[(1-\lambda_{rj})\left(\frac{x_j}{L}\right) + \lambda_{rj}\right]} \tag{2.63}$$

- Determine the instantaneous producing WOR, F_{WOP}, at $x_j = L$, for all *j* at time t

$$F_{WOP} = \frac{\sum_{j=1}^{\alpha-1} q_{wj}}{\sum_{j=\alpha}^{n} q_{oj}} \tag{2.64}$$

Step 7: Calculate the cumulative production values, from Equation 2.65 – 2.67.

- Determine the cumulative oil recovered from all beds.

$$N_{PRt} = \sum_{j=1}^{\alpha-1} N_{PRj} + \sum_{j=\alpha}^{n}\left(\frac{x_j}{L}\right) N_{PRj} \tag{2.65}$$

In surface units, multiply N_{PRj} by $1/\beta_{oj}$

- Determine the vertical coverage, at time t denoted by Cv.

$$N_{PRT} = \sum_{j=1}^{n} N_{PRj} \tag{2.66}$$

$$C_v = \frac{N_{PRt}}{N_{PRT}} \tag{2.67}$$

- Determine cumulative WOR, designated by F_{wopt}, from Equation 2.68 -2.69.

  i. For CIR case.

$$F_{wopt} = \frac{q_T t - N_{PRt}}{N_{PRt}} \tag{2.68}$$

  ii. For CIP case.

$$F_{wopt} = \frac{y \frac{\Delta P}{L} \sum_{j=1}^{\alpha-1} \left[t - t_{BTj}\right] h_j \lambda_{fwj}}{N_{PRt}} \tag{2.69}$$

  Implicitly in this equation is the fact that q_j is constant for $t > t_{BTj}$. This is not true for the CIR case.

- Determine the cumulative water injected into bed *i* to time t, following Equation 2.70 – 2.75.

  i. For the CIP case.

  For $t \le t_{BTi}$:

$$W_i = \left(\frac{x_i}{L}\right) N_{PRi} \tag{2.70}$$

  For $t > t_{BTi}$:

$$W_i = N_{PRi} + y \frac{\Delta P}{L} \left(t - t_{BTi}\right) h_i \lambda_{fwi} \tag{2.71}$$

  ii. For the CIR case.

  For $t \le t_{BTi}$:

$$W_i = \left(\frac{x_i}{L}\right) N_{PRi} \tag{2.72}$$

  For $t > t_{BTi}$:

$$W_i = N_{PRi} + h_i \lambda_{fwi} \left\{ \sum_{\delta=1}^{\alpha-1} \left[\frac{q_T \left(t_{BT\delta+1} - t_{BT\delta}\right) - \sum_{j=\delta+1}^{n} \Delta N_{PRj}}{\sum_{j=1}^{\delta} \left(h_j \lambda_{fwj}\right)} \right] - \frac{q_T \left(t - t_{BT\alpha-1}\right) - \sum_{j=1}^{\alpha-1} \Delta N_{PRj}}{\sum_{j=1}^{\alpha} \left(h_j \lambda_{fwj}\right)} \right\} \tag{2.73}$$

  Where,

$$\Delta N_{PRj} = N_{PRj}\left[\left(\frac{x_j}{L}\right)_{i+1} - \left(\frac{x_j}{L}\right)_i\right] \qquad (2.74)$$

and

$$\Delta N_{PRj} = N_{PRj}\left[\left(\frac{x_j}{L}\right)_{\alpha} - \left(\frac{x_j}{L}\right)\right] \qquad (2.75)$$

1.6 Tiab's Layered-composite Reservoirs Model.

Tiab(1986)extended the Dykstra-Parsons method to layered-composite reservoirs and found that the sweep capacity of the displacing fluid is a function of rock characteristics from layer to layer and not just depending on permeability as assumed by Dykstra-Parsons. And the waterflooding performance is controlled by the mobility ratio. Equations for pressure drop, time of breakthrough, water-front location, coverage, WOR and cumulative oil recovery was established and applied for constant injection pressure. The speed and proportion of flowing water varies from layer-to-layer and time to time. Some of the assumptions in the model he proposed, each layer is independently homogeneous, isotropic, and saturated with incompressible fluids. The reservoir was also assumed to be horizontal with a linear geometry. The model assumes no cross flow between layers, and a piston-like displacement. It was assumed that at any time of water injection, the pressure drop in a particular layer is equal to those in the other layers. The layer ordering is in ascending order with respect to their breakthrough times (i.e., the first layer breaks through first, before the last layer), and capillary and gravity forces are negligible. Initial water saturation assumed to be equal to the irreducible saturation. And the waterfront to leave behind it, oil at its irreducible saturation. In his model, Tiab(1986), also assumed relative permeability to water behind its front, and relative permeability to oil to be the same in the swept and un-swept portions of the system, respectively.

Each layer in the Tiab's model consists of two regions. One region represents the swept zone, which shows the waterfront separation from the oil zone at a particular time. While the other region is the un-swept zone, with high oil saturation. This is as shown in Figure 2.8.

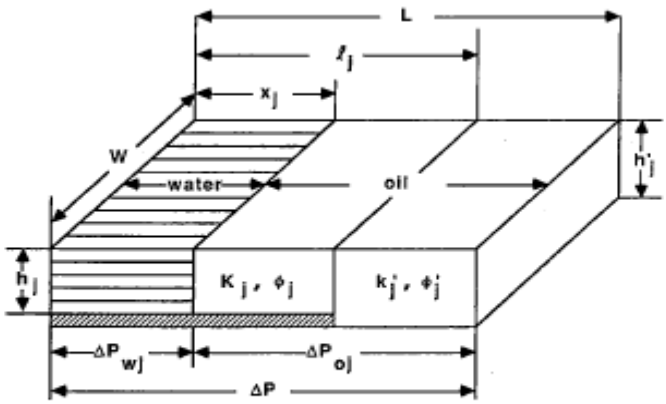

Figure2.8Schematic diagram showing waterflooding in a layered-composite reservoir (Source: Tiab(1986))

Chapter 2: Computer Implementation Process

1.7 Introduction

In this work, the waterflood performance shall be investigated using the traditional Dykstra and Parsons, and the Reznik et al method using Python programming language. [1]

The data for the program was taken from C.H. Wu (1988). The data includes a relative permeability of water and oil, and saturation table, and a table showing bed parameters, thickness, porosity, and permeability for each bed layer. There were no individual bed viscosities, thus an average viscosity for oil and water was used for the program. The saturation distribution was not also on per bed basis. The relative permeability of oil for the determination of the mobility ratio was taken at the initial water saturation (SWI), and that of water at 1-SOR. Where SOR is the residual oil saturation.

The solution and step by step approach for the two cases was implemented with Python 3 software in Anaconda integrated development environment, shown in the schematic in Figure 3.1. The process employs three steps.

- Writing the computer program
- Development of a graphical user interface.
- Conversion of the python file to an executable format.

[1] Python is a high-level, object-oriented programming language, created by by Guido van Rossum in the 1980s.

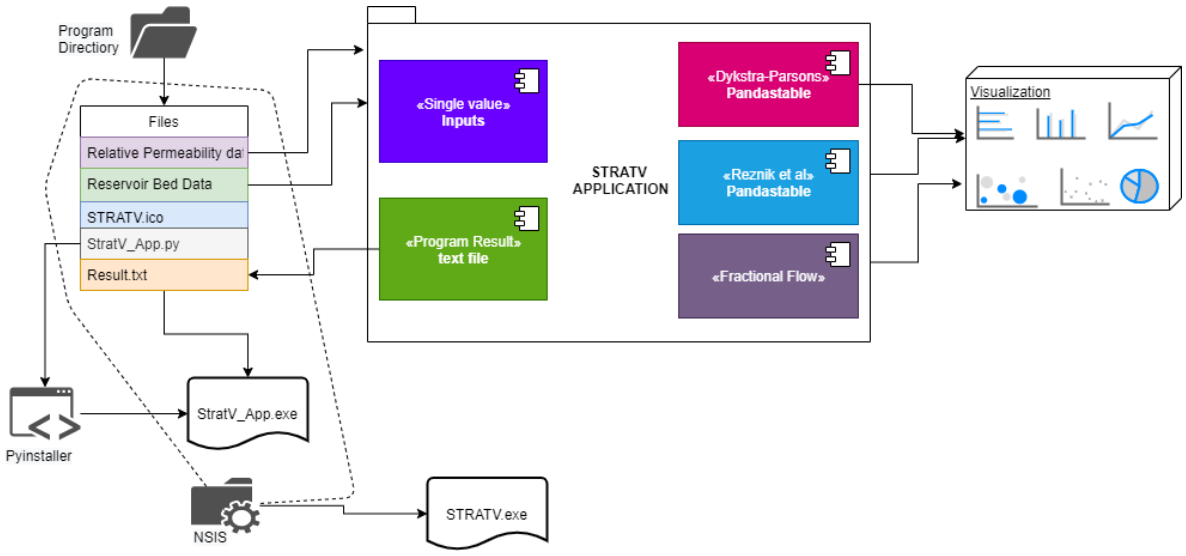

Figure 3.1 – Computer Implementation Process Schematic

1.8 Writing the Computer Program.

The computer program was written in python programming language for both Dykstra-Parson discrete analytical solution and Reznik et al continuous solution, based on the process and waterflood performance parameters proposed by the authors.

1.1.1.1 Defining Inputs

STEP1: The first step for this process is input initialization. All the inputs required for the calculation were first initialized. The input data in Table 3.1 for the initialization step was also obtained from C.H. Wu (1988) paper.

Table 3.2– Defining Program inputs

1	Length_of_bed_ft = 2896
2	width_of_bed_ft = 2000
3	average_porosity = 0.25
4	VISO = 3.6
5	VISW = 0.95
6	OFVF = 1.11
7	WFVF = 1.01
8	SWI = 0.2
9	SGI = 0.16
1	SOI = 0.65
0	SOR = 0.35

1	Residual_gas_saturation_unswept_area = 0.06
1	Residual_gas_saturation_swept_area = 0.02
1	Residual_gas_saturation = Residual_gas_saturation_unswept_area+Residual_gas_saturation_swept_area
2	Constant_injection_rate = 1800
1	Inj_Pressure_differential = 700
3	
1	
4	
1	
5	
1	
6	

1.8.1 Importing Python Modules

STEP2: The next step requires importing all necessary modules from the python library. Some of which requires installation such as the Pandastable. Installing files or libraries into your python path is mostly accomplished with the 'pip' command. By simply typing 'pip install filename' into your command prompt or using windows PowerShell for windows users. Pandastable is a software with graphical user interface, for viewing and working with tables, and making custom plots. It was developed by Farrel. D (2019). The following Python modules and libraries in Table 3.2 were imported for this work.

Table 3.3– Algorithms for Importing Python Module

1	**from tkinter import***
2	from tkinter import filedialog
3	from tkinter import ttk
4	import pandas as pd
5	from tkinter.messagebox import*
6	from tkinter.filedialog import askopenfilename
7	import csv
8	import math
9	from decimal import*
1	import numpy as np
0	from pandastable.core import Table

```
1  from pandastable.data import TableModel
1  import matplotlib.pyplot as plt
1  from scipy import integrate
2
1
3
1
4
```

Tkinter is a Python default application for building a graphical user interface. Pandas is the Python default library for working with tables, while numpy is for arrays. Python uses the matplotlib for making plots.

STEP 3: The tabular data for the saturation distribution, relative permeability of water and oil, and the reservoir bed parameters were arranged in a csv file stored in the same directory or folder as the Python file or executable file. This file serves as a template for working with the application. When trying to input new data, the column/ table header title must correspond with the csv file template. The data is as shown in Table 3.3:

Table 3.4- Oil relative permeability data

SW	KRW	KRO
0.2	0	1
0.25	0.003	0.68
0.3	0.008	0.46
0.35	0.018	0.32
0.4	0.035	0.2
0.45	0.054	0.124
0.5	0.08	0.071
0.55	0.105	0.038
0.6	0.14	0.017
0.65	0.18	0

As a matter of rule, the oil and water relative permeability must be (1) saved in the same folder as the executable file (2) must be saved with the name **'Oil_Water_Relative_Permeability_data.csv'**. (3) The file must be saved with the same column heading as shown in Table (3.3) and Table 3.4

Same rule also applies for the reservoir bed data. This should be saved with the name **'Permeability_Porosity_distribution_data.csv'**, and with same column heading as shown in Table 3.4:

Table 3.5 - Bed data permeability distribution

LAYER	DEPTH	THICKNESS	PERMEABILITY	POROSITY
1	4359.5	1	593	0.301
2	4361	1	500	0.288
3	4363.5	1	366	0.283
4	4365.5	1	1464	0.309
5	4367.5	1	2790	0.311
6	4369.5	1	5940	0.305
7	4371.5	1	9230	0.322
8	4373.5	1	2860	0.306
9	4375.5	1	3080	0.322
10	4377.5	1	594	0.297
11	4379.5	1	2370	0.323
12	4381.5	1	526	0.286

The csv (comma-separated values) files were imported into the into the python file with the following algorithms in Table 3.5.

Table 3.6 – Algorithm for reading input data csv files

1	bed_data = pd.read_csv('Permeability_Porosity_distribution_data.csv')
2	RPERM_data = pd.read_csv('Oil_Water_Relative_Permeability_data.csv')

STEP 4: After the above steps, the program is written based on the step by process outlined by Dykstra-Parson and Reznik et al. The following were determined from the calculations:

General Calculations:

1) Average Porosity
2) Relative Permeability at 1-SOR
3) Relative Permeability at SWI
4) Gross Rock Volume
5) Displacement Efficiency
6) Mobility Ratio

7) Areal Sweep Efficiency at Breakthrough
8) Areal of Reservoir Bed
9) Areal Sweep Efficiency

Dykstra-Parsons Calculation: For the Dykstra-Parsons calculation, the following were determined.

1) Sorted Bed Layers
2) Oil Mobility
3) Water Mobility
4) Injected water flowrates
5) Oil flow rate in each bed
6) Vertical coverage
7) Water oil ratio
8) Cumulative oil recovery
9) Water volume for gas space fill-up
10) Producing water-oil ratio
11) Cumulative water produced.
12) Cumulative water injected.
13) Time (days)
14) Time (years)
15) Flood front location of each bed.

For the Reznik et al extension to continuous exact analytical solution, the following below were computed.

1) Sorted bed layers.
2) Breakthrough time
3) Flood front location of the last bed as each bed breaks through.
4) Ultimate recovery
5) Flood front location of each bed at breakthrough of other beds
6) Flood front location of last bed at each real time step.
7) Real time for constant injection pressure case
8) Dynamic bed
9) Sum of water flowrate before breakthrough of dynamic bed

10) Sum of oil flowrate before breakthrough of dynamic bed
11) Instantaneous producing water oil ratio before breakthrough of dynamic bed.
12) Instantaneous producing water cut.
13) Cumulative oil recovered.
14) Cumulative oil recovered from all beds.
15) Vertical coverage
16) Cumulative water oil ratio for constant injection rate case
17) Cumulative water oil ratio for constant injection pressure case
18) Flood front location of other beds at time t.
19) Flood location of other beds with indication of beyond breakthrough location
20) Average mobility
21) Superficial filter velocity
22) Actual linear velocity
23) Instantaneous volumetric flowrate of water
24) Instantaneous volumetric flowrate of oil.
25) Property time
26) Cumulative water injected.

It is worth noting that the column and row numbering are based on their index and not necessarily on the reservoir bed layers. The program, numbers the columns and rows based on index rather than the bed layer number. Care must be taken to match these indexes to the corresponding layer, to identify the given bed of interest. Table 3.6 shows the bed layer after the bed ordering. Dykstra-Parson ordered the bed according to decreasing order of permeability, while Reznik et al proposed a bed ordering parameter given in Equation (2.40)

Table 3.7 – Index versus Layer definition

Index	Layers
0	7
1	6
2	9
3	8
4	5
5	11

6	4
7	10
8	1
9	12
10	2
11	3

The codes written for the implementation of the solution is found in the Appendix A.

1.8.2 The Fractional Flow Curve

A fractional flow curve menu tab was incorporated into the program. Generating this curve followed four encompassing steps. Which are highlighted under this section:

To begin this calculation, the following Python modules in Table 3.7 are imported:

Table 3.8– Python modules for making fractional flow curve.

```
1  import numpy as np
2  from scipy.optimize import*
3  import math
4  import random
5  import matplotlib
6  import matplotlib.pyplot as plt
7
```

1.1.1.2 Generating the fractional flow equation.

The fractional flow equation is generally given by:

$$f_w = \frac{1}{1 + \frac{K_{ro}}{K_{rw}} \frac{\mu_w}{\mu_o}} \tag{3.1}$$

Where,

$$\frac{K_{ro}}{K_{rw}} = a\, e^{-b S_w} \tag{3.2}$$

This imply that the coefficients and constants *'a'* and *'b'*, can be expressed as:

$$a = K_1 e^{b S_w} \tag{3.3}$$

$$b = \frac{\ln K_1 - \ln K_2}{S_{w2} - S_{w1}} \quad (3.4)$$

The Python program for the determination of *a* and *b* was written as shown in Table 3.8:

Table 3.9– Defining function for fractional flow calculation.

1	b = (np.log((KRO/KRW)[2]) - np.log((KRO/KRW)[3]))/(SW[3] -SW[2])
2	a = (KRO/KRW)[2]*math.exp(b*SW[2])
3	def fw(SW):
4	fw = 1/(1+a*(VISW/VISO)*np.exp(-b*SW))
5	return(fw)

Note: The extreme points are not chosen because log of the relative permeability ratio does not form straight lines at the extremes, which will give erroneous result for the relative permeability ratio correlation in Equation (3.2)

After successful determination of *'a'* and *'b'*, the fractional flow data can be generation by substituting *'a'*, and *'b'* into the fractional flow equation.

1.1.1.3 Generating the Tangent to the fractional flow curve.

Generating the tangent to the fractional flow curve is the tricky part of plotting the fractional flow curve. Customarily, to generating a tangent is easier when the point of tangency is given or known. In this case the only point that is known is the initial water saturation (S_{wi}, 0) from where, the tangent line is drawn.

With the point (S_{wi}, 0) known, the tangent equation can be expressed as:

$$y = m(S_w - S_{wi}) \quad (3.5)$$

where,

m is the slope of the tangent line,

and S_w the water saturation

The concept involves generating several tangent lines that will intercept the fractional flow curve at several points. The line (drawn from point (S_{wi}, 0)) with the maximum slope touching the fractional flow curve will give the suitable tangent line.

This necessitated equating the tangent line equation, and the fractional flow equation.

$$m\,(S_w - S_{wi}) = \frac{1}{1 + a\,e^{-bS_w}\dfrac{\mu_w}{\mu_o}} \tag{3.6}$$

which gave m to be:

$$m = \frac{1}{(S_w - S_{wi})\left(1 + a\,e^{-bS_w}\dfrac{\mu_w}{\mu_o}\right)} \tag{3.7}$$

For this program, ten thousand of uniformly distributed random points were generated to the required slope. Using the algorithm in Table 3.9.

Table 3.10 – Algorithm for drawing tangent to the fractional flow curve

```
1  ''' To calculate a suitable slope for the tangent to the fractional flow curve
2  Drawn from the initial water saturation'''
3  # STEP1: Generate a list of uniformly distributed random numbers from a water saturation
4  # greater than the initial water saturation to 1
5  xList = []
6  for i in range(0, 10000):
7  x = random.uniform(SWI+0.1, 1)
8  xList.append(x)
9  xs = np.array(xList)
1  # STEP2: Calculate different slopes of tangents or lines intersecting the fractional
0  # flow curve using the array generated in step 1 as the water saturation.
1  m = 1/((xs-SWI)*(1+(VISW/VISO)*a*np.exp(-b*xs)))
1  # STEP3: Calculate the maximum slope from different slopes generated in step 2.
1  # The value of this slope will be the slope of the tangent to the fractional flowcurve
2  tangent_slope = max(m)
1
3
1
4
1
5
```

Upon calculating the slope of the tangent line, the water saturation can be calculated from tangent equation by substituting the point (S_{wBT}, 1) into the tangent equation.

Where S_{wBT} is the water saturation at breakthrough.

The flood front water saturation is also determined by substituting the point (Swf, Fwf) into the tangent equation and the fractional flow equation. The resulting non-linear equation is then solve using Python *fsolve* by importing the *scipy* module. The algorithms are as shown in Table 3.10.

Table 3.11– Algorithm for Water saturation and fractional flow at flood front

1	Saturation_at_Breakthrough = SWI + 1/tangent_slope
2	def funct(SWF):
3	swf = SWF[0]
4	F[0] = ((tangent_slope*(swf-SWI)*(1+(VISW/VISO)*a*math.exp(-b*swf)))-1)
5	return F
6	SWF_Guess = np.array([SWI+0.1])
7	SWF = fsolve(funct, SWF_Guess)[0]
8	# Fractional flow at the flood front
9	Fwf = fw(SWF)

1.1.1.4 Differential of the fractional flow equation

The differential of the fractional flow equation was also plotted on the fractional flow curve. Differentiating the fractional flow equation with respect to water saturation gives the expression below:

$$\left(\frac{df_w}{dS_w}\right)_{S_w} = \frac{ab\,e^{-b\,S_w}\dfrac{\mu_w}{\mu_o}}{\left(1+a\,e^{-b\,S_w}\dfrac{\mu_w}{\mu_o}\right)^2} \tag{3.8}$$

The Python program for this is given as in Table 3.11.

Table 3.12 – Algorithm for fractional flow derivative

1	# Calculating the differential of the fractional flow equation
2	dfw_dSw = (VISW/VISO)*a*b*np.exp(-SW*b)/(1+(VISW/VISO)*a*np.exp(-SW*b))**2
3	# Generating the data for the tangent plot
4	tangent = (SW-SWI)*tangent_slope

1.1.1.5 Making the fractional flow curve

The fractional flow curve comprises the plot of the fractional flow equation, the derivative of the fractional flow equation, and the tangent on the same plot.

The Python algorithm in Table 3.12 was used to set up the plot.

Table 3.13– Algorithm for making the fractional flow curve.

1	# Making the plots
2	fig, ax = plt.subplots(constrained_layout=True)
3	fig.set_figheight(8)
4	fig.set_figwidth(12)
5	fractional_flow_curve = ax.plot(SW, fw(SW), 'b', label = 'Fractional Flow (Fw)')
6	tangent_curve = ax.plot(SW, tangent, 'k--')
7	ax.set_ylabel("Fractional Flow (fw)",fontsize=14)
8	ax.set_xlabel("Water Saturation (Sw)",fontsize=14)
9	ax.set_ylim([0,1])
1	ax.set_xlim([0,1])
0	# twin object for two different y-axis on the same plot
1	ax2=ax.twinx()
1	# make a plot with different y-axis using second axis object
1	dfw_dSw_curve = ax2.plot(SW, dfw_dSw, 'r', label ='dFw/dSw')
2	ax2.set_ylabel("dfw/dSw",fontsize=14)
1	ax.grid(True)
3	ax2.legend()
1	ax.legend(loc='upper left')
4	ax.annotate(" (Swf, Fwf)", (SWF, Fwf))
1	ax.annotate(" SwBT", (Saturation_at_Breakthrough, 1))
5	plt.show()
1	
6	
1	
7	
1	
8	
1	
9	
2	
0	
2	
1	

The resultant plot of the fractional flow curve is as shown in Figure 3.2.

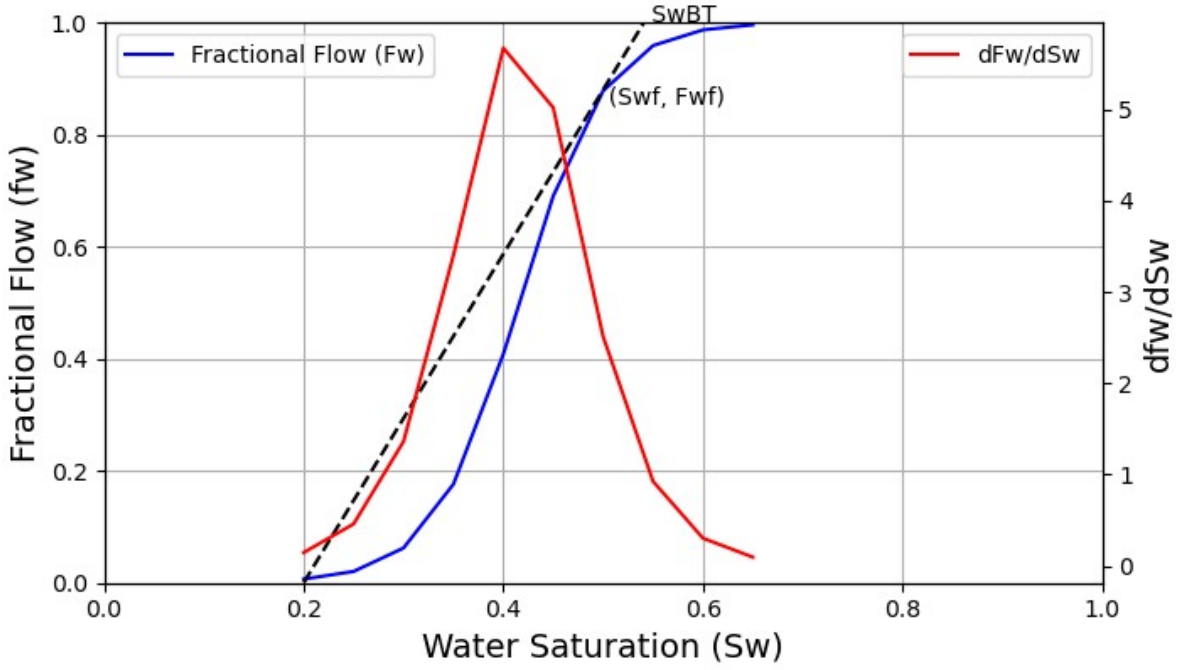

Figure 3.2 – The Fractional Flow Curve

1.9 Development of The Graphical User Interface.

The graphical user interface for the program was created with Tkinter[TM2]. Tkinter makes use of widgets such as labels, buttons, treeviews, entries, frames, canvas, and lots more to build graphical user interface.

The graphical user interface is section as seen in Figure 3.3 to Figure (3.9).

1.9.1 The Application Window.

When the program is turned on or opened the window in Figure 3.2 comes up. This is the application from which you can navigate to other part of the application.

[2] The tkinter package is the standard Python interface to the Tk GUI toolkit.

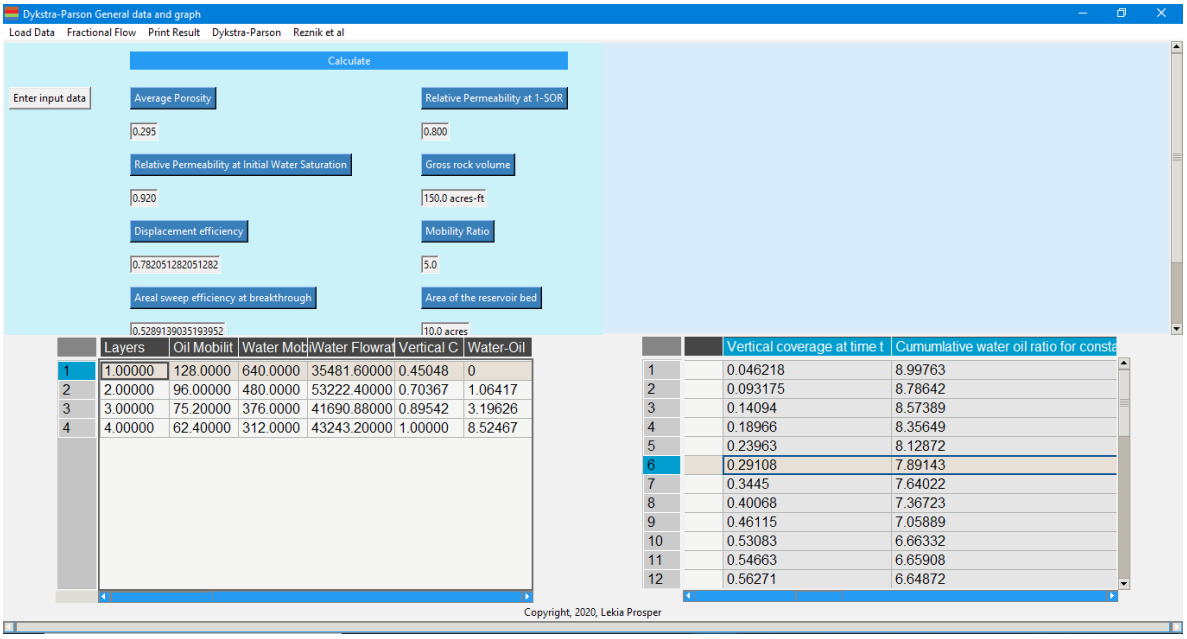

Figure 3.3– Application window

1.9.2 The Menu Bar

The application consists of five menu bars.

- Load Data
- Fractional flow
- Print Result
- Dykstra-Parsons: clicking on this menu brings up the table (*the pandastable*) on the left.
- Reznik et al: clicking on this menu brings up the table (*the pandastable*) on the right.

1.9.3 Enter Input Data Button

There is an 'Enter input data' button, that opens a window for entering the input data. The pop-up input window is as shown in Figure 3.4.

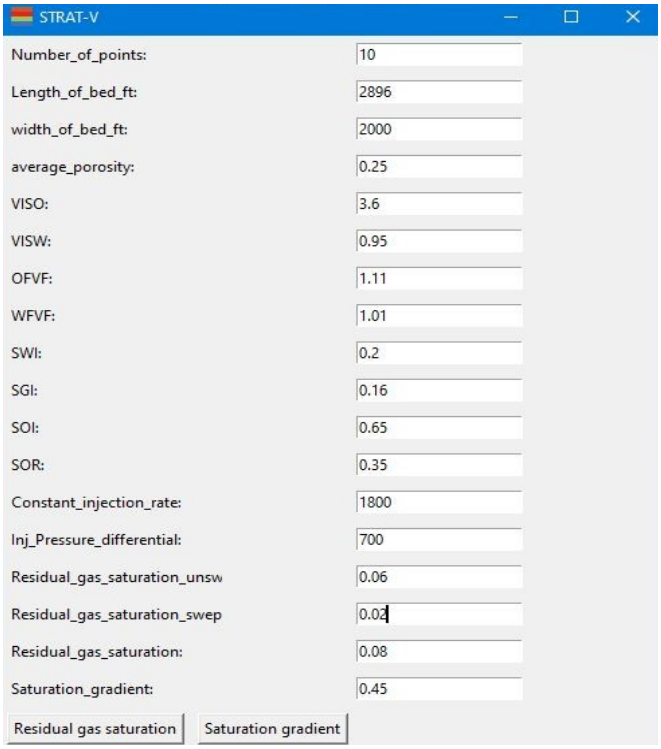

Figure 3.4 – Input window

The first input in the input window, is the '*Number of points*. This is the number of points between flood front location of the last bed at breakthrough of other beds for the Reznik et al continuous solution method. Reznik et al method proposes the use of the flood front position of the last bed at a given time to determine the flood front location of other beds at the given time. To achieve this, they divided the flood location of the last bed between breakthrough time into time steps. The number of divisions between time steps is represented by the number of points.

The Residual gas saturation button adds up the Residual gas saturation of the un-swept zone and the Residual gas saturation of the swept zone and the returns the solution in the entry widget. While the Saturation gradient returns the result of Equation (2.36)

The abbreviation for the input labels is defined as follows.

VISO – the viscosity of oil

VISW – the viscosity of water.

OFVF – the oil formation volume factor

WFVF – the water formation volume factor

SWI – the initial water saturation

SOI – the initial oil saturation

SGI – the initial gas saturation

SOR – the residual oil saturation.

1.9.4 The Calculate Frame

The ‘Calculate’ frame comprise calculation for the general calculations. The average porosity, mobility ratio and the different sweep efficiencies, and relative permeabilities at initial water saturation and one minus the residual oil saturation. The gross rock volume and the reservoir area is also determined. The calculation is performed by clicking on each button.

1.9.5 Fractional Flow Window

The fractional flow window is as shown in Figure 3.5. The window consists of two frames. One to display the result and the other to display the fractional flow data, the derivative of the fractional flow with respect to water saturation, and the tangent to the fractional flow curve data.

By the top right is a view plot button to display the fractional flow curve.

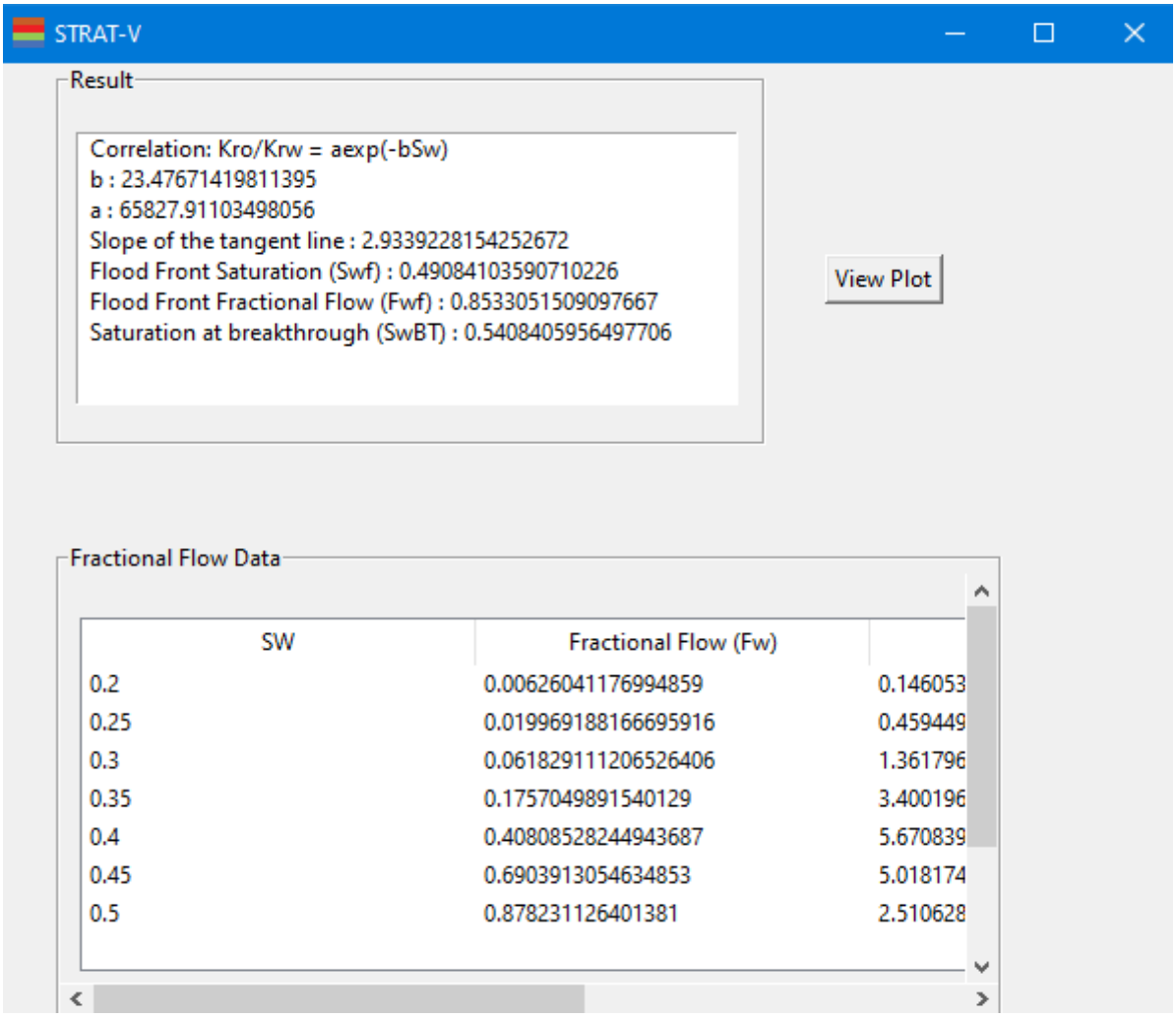

Figure 3.5– Fractional flow window

1.9.6 The Pandastable

The Pandastable[3] can perform numerous operations to work on tables. This menu can be accessed by 'right clicking' on the table.

For example**, right clicking on the column header** shows the pop-up menu in Figure 3.6.

[3] The pandastable library provides a table widget for Tkinter with plotting and data manipulation functionality. It uses the pandas DataFrame class to store table data. Created by Damien Farrell in 2014. https://pandastable.readthedocs.io/en/latest/description.html

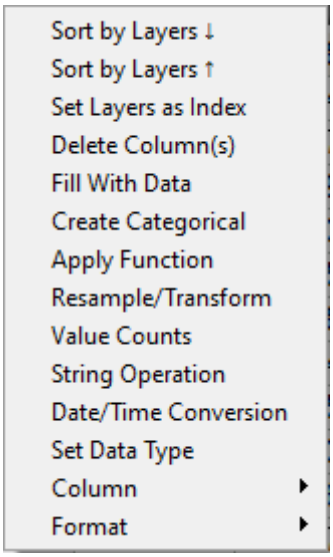

Figure3.6 – Manipulating Pandastable (a)

Also**, right clicking on the row index** shows the pop-up menu in Figure 3.7.

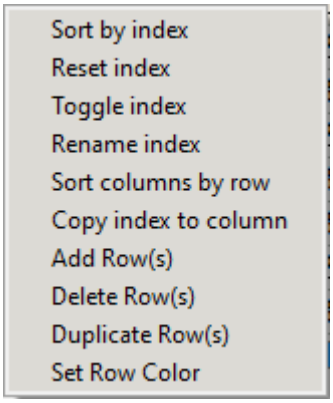

Figure3.7 – Manipulating Pandastable (b)

To **make plots**, the data to be plotted is selected, beginning with the one to be made the abscissa. **Right clicking on any of the cells** in the selected region brings up the pop-up menu in Figure 3.8. If the pop-up does not show up when the plot data is selected, simply drag the column to be made the abscissa to the first column position.

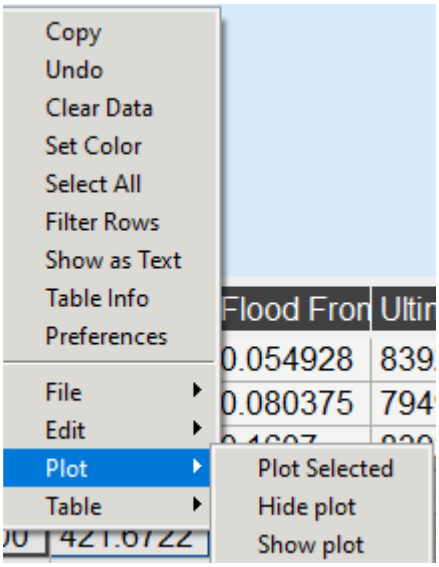

Figure 3.8 – Making Plots with Pandastable

Choosing the Plot Selected option brings up the plot window as shown in Figure 3.9.

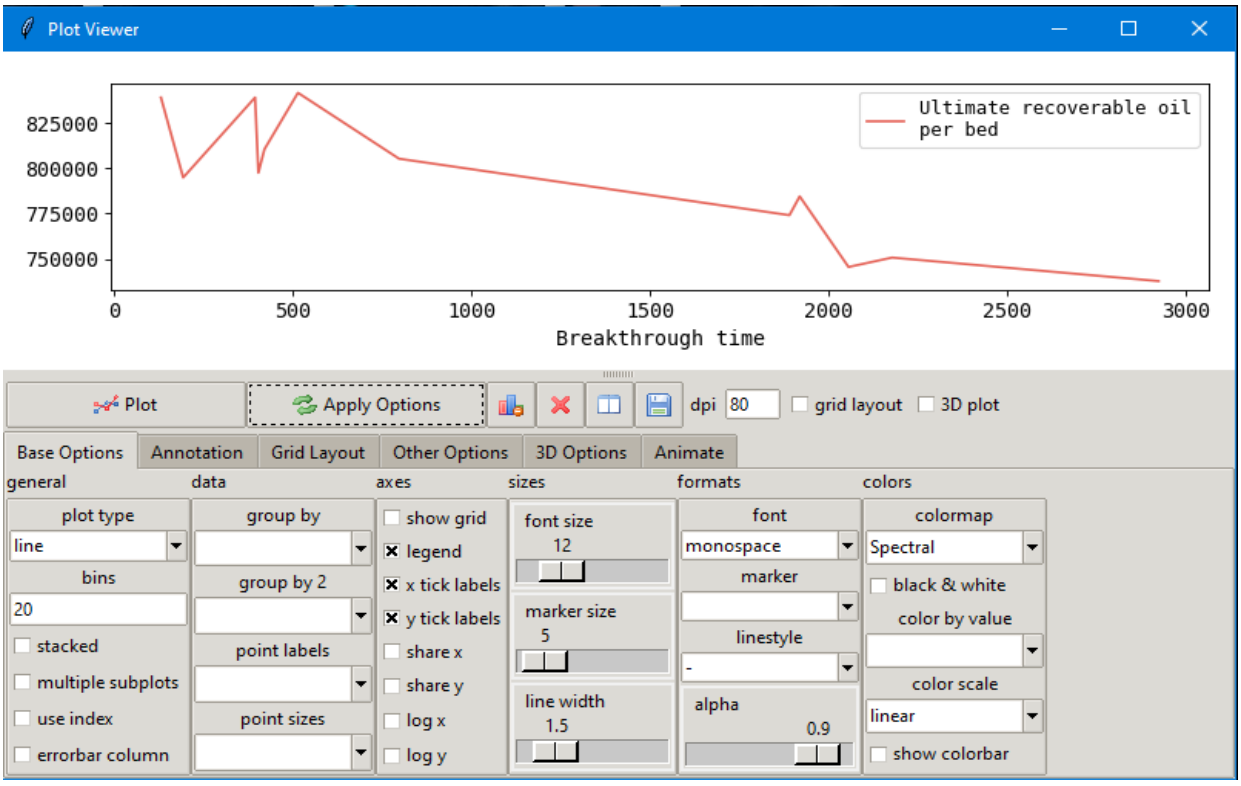

Figure 3.9 – Pandastable plot viewer

The ‘use index’ is usually ticked by default. You may need to uncheck the option to get the required plots. With plot viewer different plot type can be made, and quickly customized.

1.9.7 Print Result Menu Bar

After all calculations, you can get all your result in a text file that will be automatically saved as ‘Result.txt’ in the folder of the software application by clicking on the ‘Print Result’ menu bar. This is as shown in Figure 3.10.

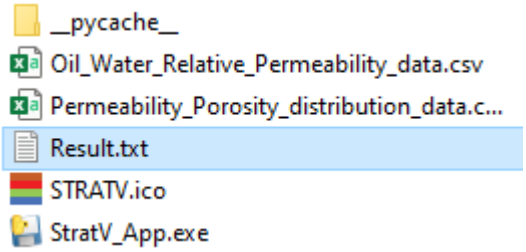

Figure 3.10 Result stored in the same folder as the Application.

1.10 Conversion of The Python File to An Executable Format.

To convert the python file to an installable executable file, the following process were taken.

- Run command prompt in the directory with the python file and all other related file. Install python installer, by typing the command ‘*pip install pyinstaller*[4]’. This command will be success if python is installed the system path. With pyinstaller installed, the command ‘*pyinstaller --onefile -w filename.py’* is issued. If some dependencies (such as pandas, pandastable, etc.) are absent, they can be installed with the command *“pyinstaller -F –hidden-import ‘name of module absent’ filename.py”*. The possibility of pyinstaller not finding a dependency already used in the development of the application is traceable to the fact that these dependencies were installed in an IDE and not in the path where Python is in the computer.

[4] PyInstaller bundles a Python application and all its dependencies into a single package. The user can run the packaged app without installing a Python interpreter or any modules.

- The NSIS[5] (nullsoft scriptable install system) is used to bundle any additional document or file the application may require, into an executable zip file.

[5] Nullsoft Scriptable Install System is a script-driven installer authoring tool for Microsoft Windows backed by Nullsoft, the creators of Winamp. NSIS is released under a combination of free software licenses, primarily the zlib license. Wikipedia

Chapter 3: Result Analysis

The example used to test the application/program with saturation and relative permeability data as shown in Table 3.3, and bed parameters as shown in Table 3.4 was gotten from the work of C.H. Wu (1988). Tabular results were obtained separately for Dykstra-Parsons's method and Reznik et al method, and the necessary waterflood descriptive and performance plots were made.

4.1 Flood Front Location

Both authors presented different concept of deriving the flood front location. This is observed from the interpretation of the equations they presented poses.

1.10.1 Dykstra-Parsons Flood Front Location

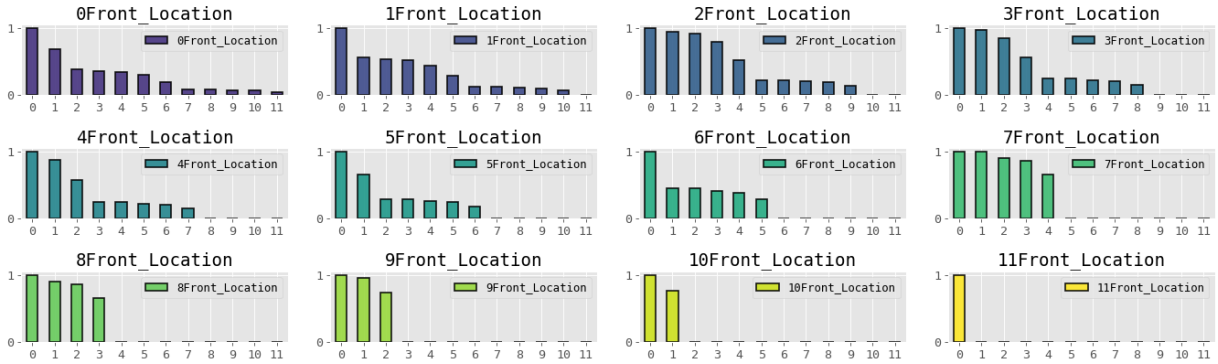

Figure 4.11– Bar plot of Dykstra-Parsons flood front location versus bed index

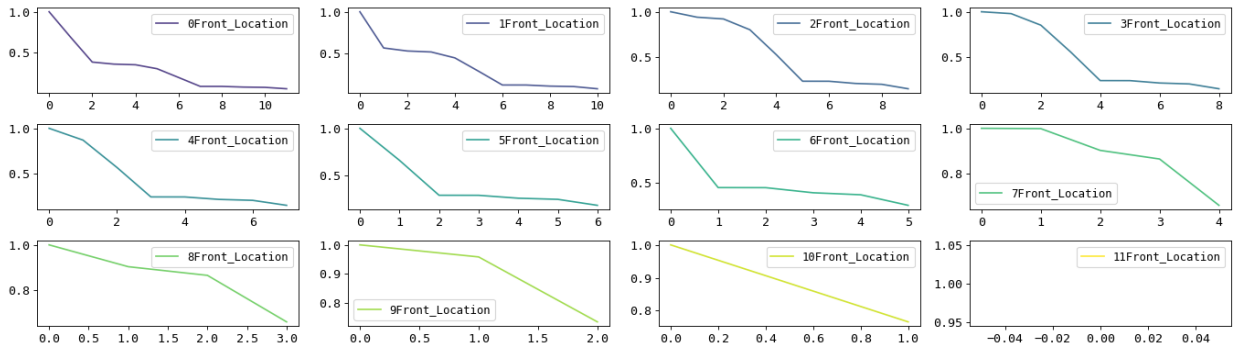

Figure 4.12– Line plot of Dykstra-Parsons flood front location versus bed index

In the Dykstra-Parsons flood front location as shown in Figure 4.1 and 4.2, the flood position of other beds when a given bed breaks through is determined. This is true for other similar data like the oil flowrate of each bed. In the first subplot in Figure 4.1, the first bar starts at 1, indicating the breakthrough of the first ordered bed. The other bars show the position of other beds at the breakthrough of the first bed. The Dykstra-Parson flood front or equations can capture waterflood parameters at breakthrough of each bed. The second subplot in Figure 4.1 and Figure 4.2 represent the flood front position of each bed at the breakthrough of the second ordered bed. In the same manner, subplot 3 is for the breakthrough of the third ordered bed and the position of the beds after it. It goes on and on until the breakthrough of the last bed, as shown in the last subplot in Figure 4.1. As can be seen from the legend on the plot, the numbering of the flood front is done by index not by the layer number. The actual representation of the index to layer number is shown in Table 3.6. Layer 7 is represented by the index 0 as it becomes the first bed after bed ordering. Therefore, the legend with the tag ‘0front_Location’ represents the first bed (layer 7). This implies that the first sub-plot in Figure 4.1 and Figure 4.2 represents the flood front location of other beds as the first bed (Layer 7) breaks through. As can be clearly observed, the Dykstra-Parsons flood front equation only presents the flood front location of beds below (lower permeability) the breakthrough bed. Also, from the plot, the flood front of other beds when the last bed (Layer 3 with index 11) breaks through is not visible because it is just a point 1.00, representing the breakthrough of only the last bed since there is no other bed below it.

1.10.2 Reznik et al Flood Front Location

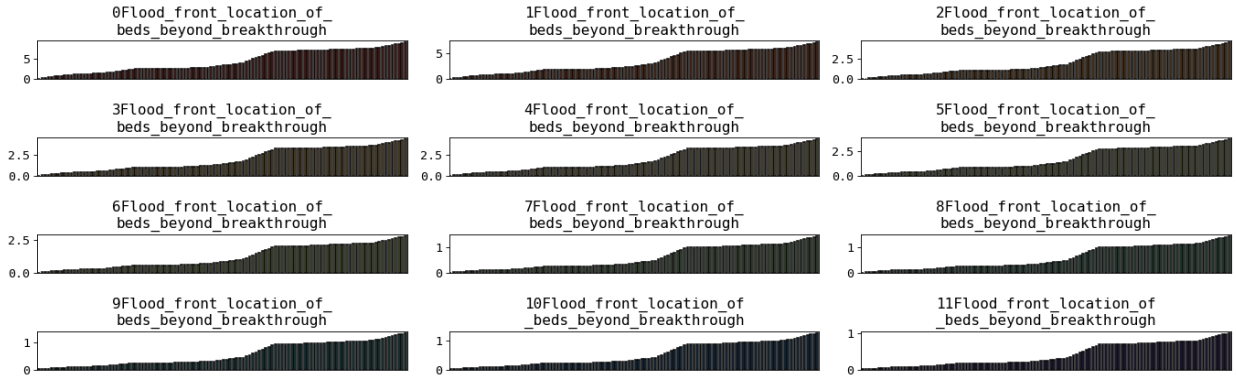

Figure 4.13– Bar plot of Reznik et al flood front location versus real time

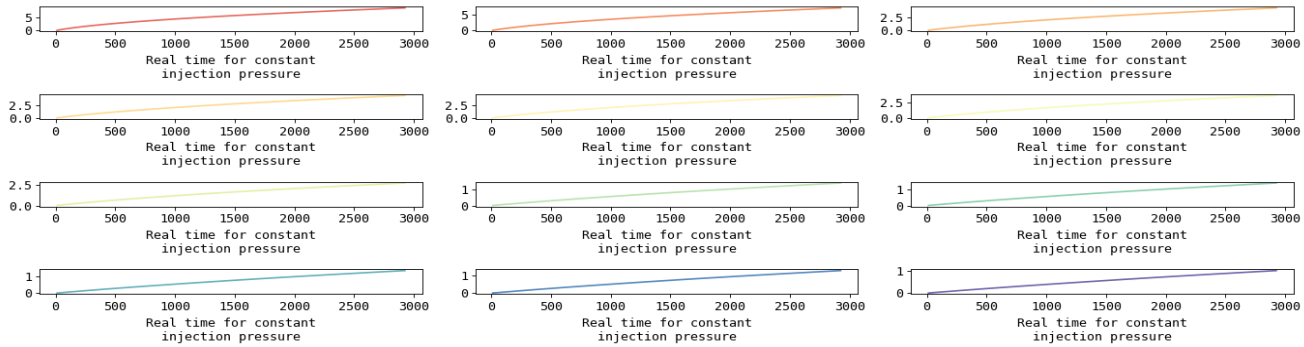

Figure 4.14- Line plot of Reznik et al flood front location versus real time

Reznik et al strived to present the flood location of each bed as a function of a continuous time step. By this, they were able to capture the position of each bed even beyond breakthrough. The last bed was use as the basis for the continuous spectrum since it is the slowest bed and will continue to advance until all other beds breaks through. To achieve this, the position of the last bed was divided into discrete steps between each of its flood front location at breakthrough. This provided more time step beyond that provided by the discrete breakthrough time of each bed.

Figure 4.3 and Figure 4.4shows the advancement of the flood front of each bed as the last bed moves to breakthrough. The first subplot in Figure 4.3 and 4.4 traces the continuous movement of the flood front of the first ordered bed. The movement is captured until the last bed breaks through. Likewise, the flood front position of each bed is captured until the

last bed breaks through. This is clearly shown in the last subplot with the flood front position ending at 1, representing its breakthrough.

1.11 Vertical Coverage

1.11.1 Dykstra-Parsons Vertical Coverage

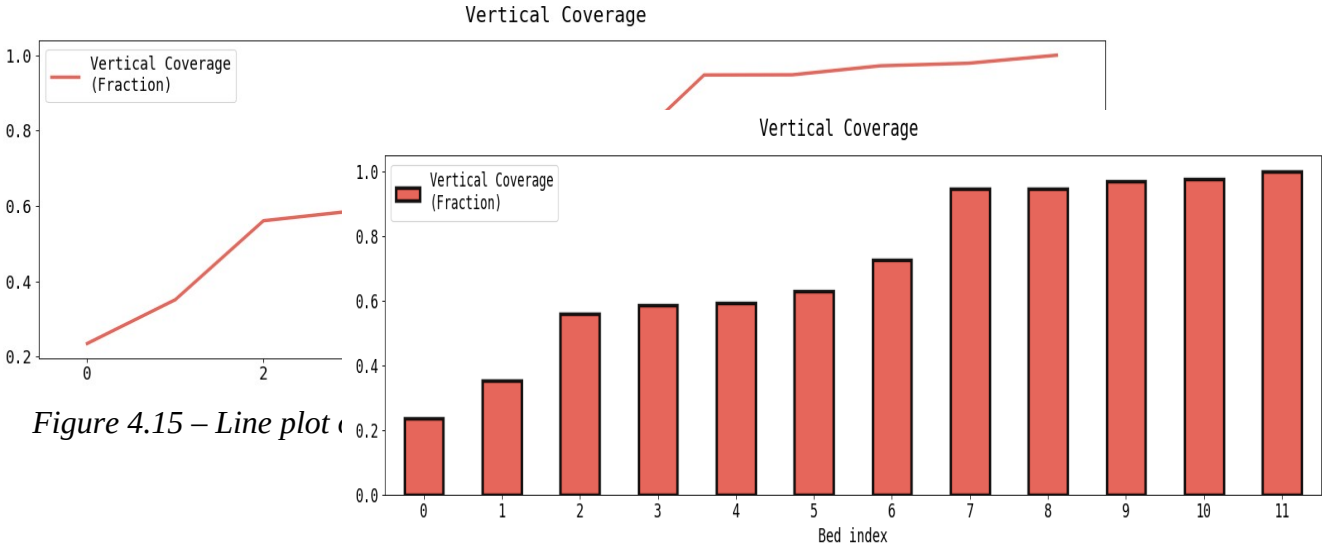

Figure 4.15 – Line plot

Figure 4.16 – Bar plot of Dykstra-Parsons vertical coverage versus bed index

The Dykstra-Parsons vertical coverage describes the fraction of the reservoir flooded by water as each bed breaks through. In Figure 4.6, the vertical coverage of the reservoir at the breakthrough of bed three and four is somewhat constant. This implies that both beds advance almost at the same speed. This is a direct effect of the closeness of their permeability values. Bed three has a permeability of 2790md while bed four has a permeability of 2860md. A difference of 70md.

The line Plot is seen in Figure 4.5 and the bar graph in Figure 4.6. From Figure 4.6, as the first bed breaks through, the fraction of the reservoir that should have been waterflooded is about 0.23. And as the last bed break through, the entire reservoir bed should be waterflooded, which is represented with a vertical coverage of 1.00. As already been discussed, the vertical coverage presented by Dykstra-Parson, is not a function of some continuous time, but just a discrete function of the individual bed breakthrough.

1.11.2 Reznik et al Vertical Coverage

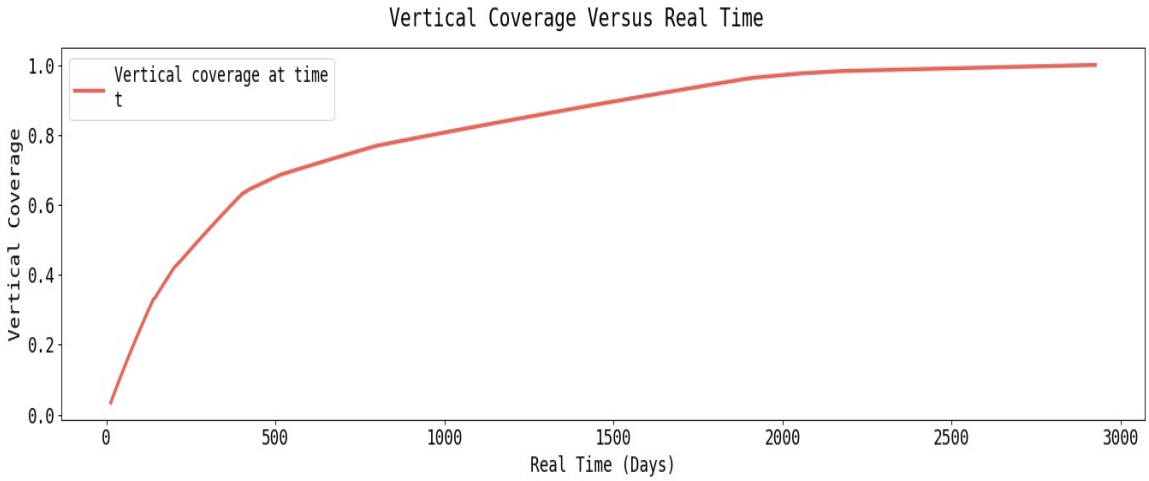

Figure 4.17– Reznik et al plot of vertical coverage versus real time

Reznik et al presented their vertical coverage as a continuous function of real time rather than on the concept of the individual bed breakthrough. Figure 4.7 looks more smoother as compared to Figure 4.5 and will be smoother as the number of points is increased. But it retains the same shape as Figure 4.5. The curve in Figure 4.7 is steep at the beginning and flattens at the end. In Figure 4.7, the vertical coverage of the reservoir is captured beginning at time zero until the entire reservoir breaks through.

1.12 Cumulative Oil Recovery

1.12.1 Dykstra-Parsons Cumulative Oil Recovery Versus Producing Water-Oil Ratio

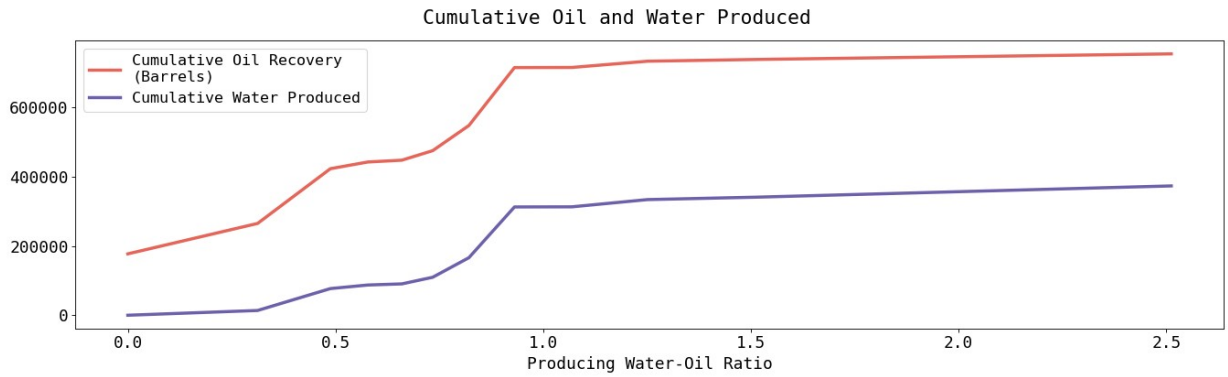

Figure 4.18– Dykstra-Parson plot of cumulative oil recovery and water produced versus producing WOR

Water production rate is only obtained from beds already broken through. While the oil production rate is chiefly from beds with permeability lower than the beds which has broken through. The cumulative water produced is obtained by taking the integral of the producing water-oil ratio against the cumulative oil recovery. The plot in Figure 4.8 shows a representation of the plot of cumulative oil recovery and water produced against the water oil ratio.

1.12.2 Reznik et al Cumulative Oil Recovery Versus Producing Water-Oil Ratio

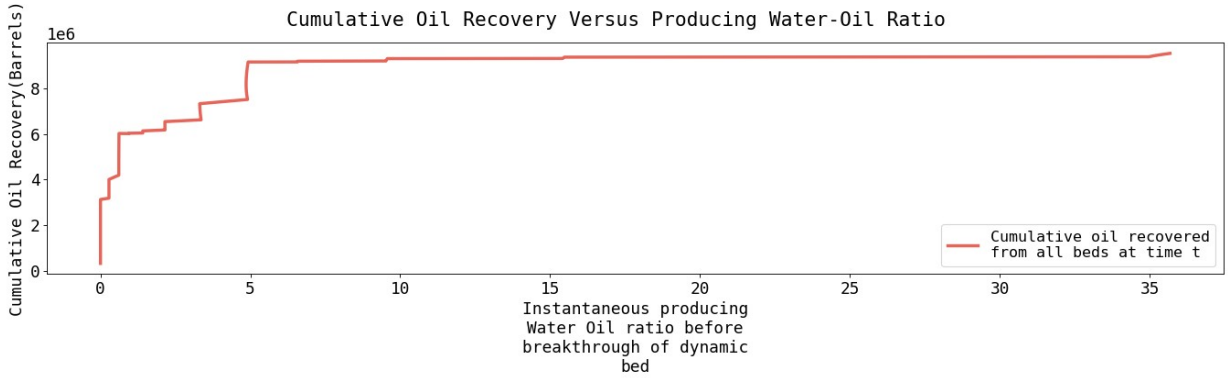

Figure 4.19– Reznik et al plot of cumulative oil recovery and water produced versus producing WOR

Reznik et al calculated their instantaneous producing water-oil ratio by considering that the water produced at a given time is only the sum of water production rate of each bed which has broken through. The bed which just breaks through, he called the dynamic bed. And the oil production at a given time is restricted to beds which have not broken through. Which are the beds below the dynamic bed at a given time. For instance, when the dynamic bed is

the first bed, no water is expected to be produced, as the bed will just be broken through. But it is regarded and expected, that other beds below the first bed should have had oil production. This explains why there is oil production despite no instantaneous producing water oil ratio from Figure 4.9. The Reznik et al in Figure 4.9gives a larger range (0 – 35) of producing water-oil ratio, than the result of Dykstra-Parsons (0 – 2.5). As can be observed from the plot, within the same range of producing water-oil ratio as that of Dykstra-Parsons the maximum cumulative oil produced is still about 700,000 barrels of oil, authenticating both solutions.

1.13 Oil Flow Rate

1.13.1 Dykstra-Parsons Oil Flow Rate

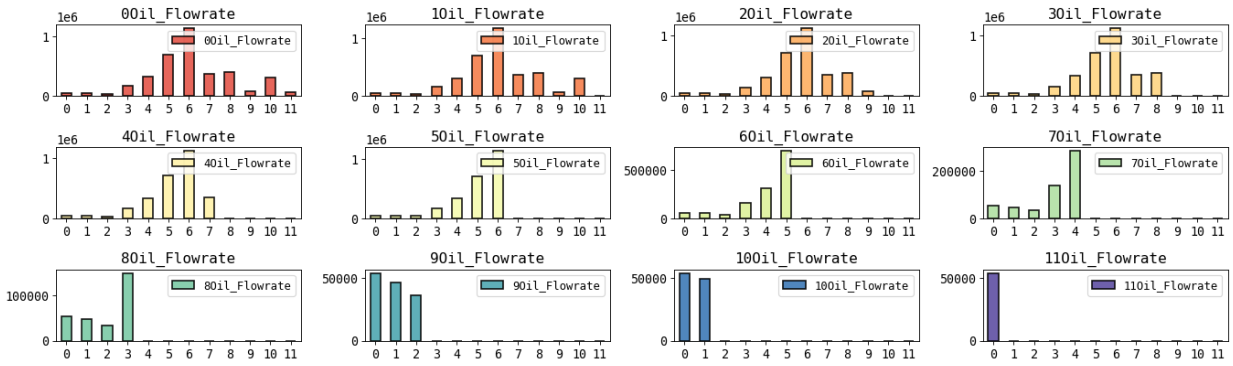

Figure 4.20 – Dykstra -Parson plot of oil flowrate versus bed index

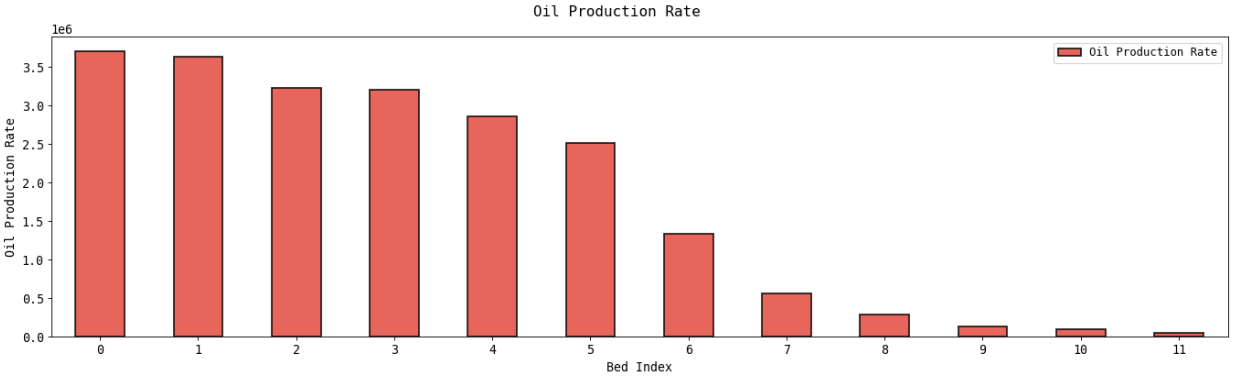

Figure 4.21 - Dykstra-Parsons's plot of Oil Production rate versus bed index

The oil flow rate in Figure (4.10) is the flow rate per bed as a particular bed break through. The oil flow rate in this figure do not follow a regular pattern. This variation may be due to

the porosity distribution from bed to bed. A summation of this oil rate for beds with permeability lower than the beds that just break through or what Reznik et al calls the dynamic bed, gives the representation in Figure (4.11). This oil flowrate, and the sum of the water flow rate for all beds which has broken through (beds above the dynamic bed), is used for the calculation of the water oil ratio. The oil flow rate pattern in Figure (4.11) is more regular as it decreases as we go down the bed.

1.13.2 Reznik et al Oil Flow Rate

Reznik et al takes each of the discrete bed oil flow rate and analyses it with a progressive time. In accordance with the continuous movement of the of the flood front of the last bed. Generally, the oil flow rate in each bed is decreasing with time horizontally along a bed and reducing vertically as we move from bed to bed downwards. This inarguably follows the same pattern as the instantaneous water flow rate in Figure (4.12). **But the model did not provide information on what happens when there is a variation in the location or bed in which the water is injected**.

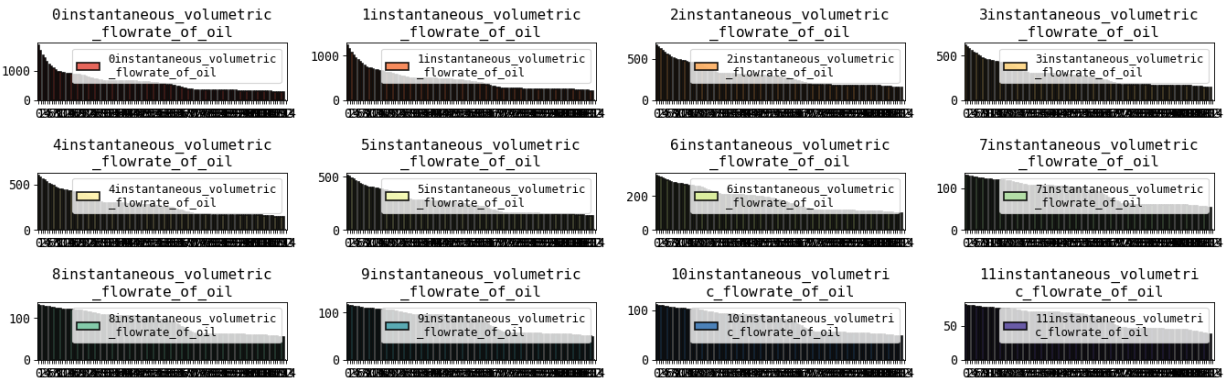

Figure 4.22 – Reznik et al plot of instantaneous volumetric oil flow rate versus real time

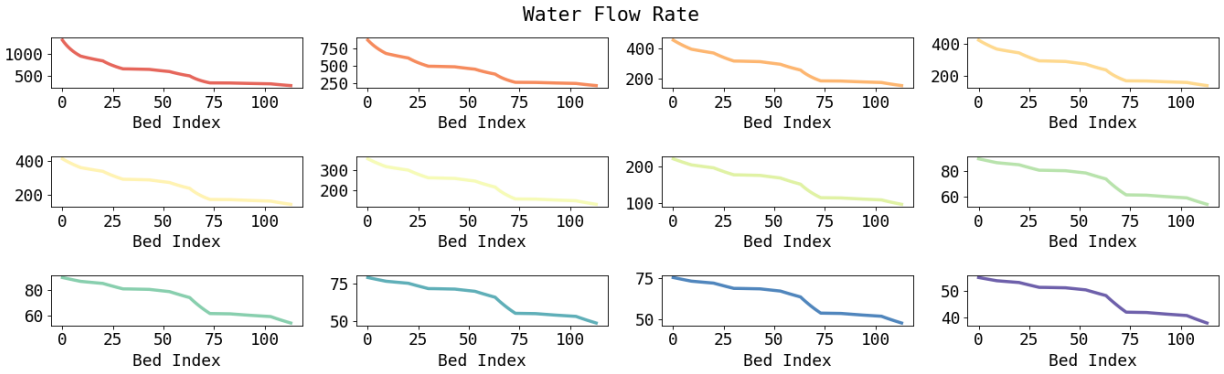

Figure 4.23– Reznik et al plot of water flow rate versus bed index

1.14 Result Validation

The result from the program for both Dykstra-Parson and Reznik et al calculations were validated with simulation results. The simulation was performed using the Open Porous Media (OPM) Flow[6] simulator. With the simulation runs conducted for ten years, with consideration of a default PVT data used in the SPE1.data file, the following results were obtained under a 20 by 20 by 12 grid blocks. The simulation specification is as shown in Appendix B.

The result shows a close match with what was obtained from the simulation. From Figure 4.14 to 4.15, the Reznik et al solution gave result with close match to that obtained from the simulation model. The continuous nature of Reznik et al proposal also provided more accuracy to the result. Increasing the number of points between breakthroughs will give a smoother curve. The Dykstra-Parson time is not based on real time but on the time, a given recovery is achieved. Their model proposes a discrete solution dependent on the established number of reservoir beds.

The time for a given recovery tend to occur at an earlier time for the simulation than for Dykstra and Parson. At the earlier stage of waterflooding Reznik et al result gave an earlier recovery time for cumulative oil recovery with the reverse for oil production rate. At a later period of the waterflood, the Reznik et al solution leads the simulation result for cumulative oil produced, with the reverse for oil production rate. The Simulation attends a constant cumulative oil production than both the Reznik et al and the Dykstra-Parson solution.

One reason for this lag is because Dykstra-Parsons model assumes that at breakthrough, all moveable oil has been swept from the layer that has just broken through, whereas in the simulation model at breakthrough, there is still moveable oil behind the front.

[6]**Flow** is a reservoir simulator for three-phase black-oil problems using a fully implicit formulation. There are also specialized variants for solvent and polymer problems. https://opm-project.org/

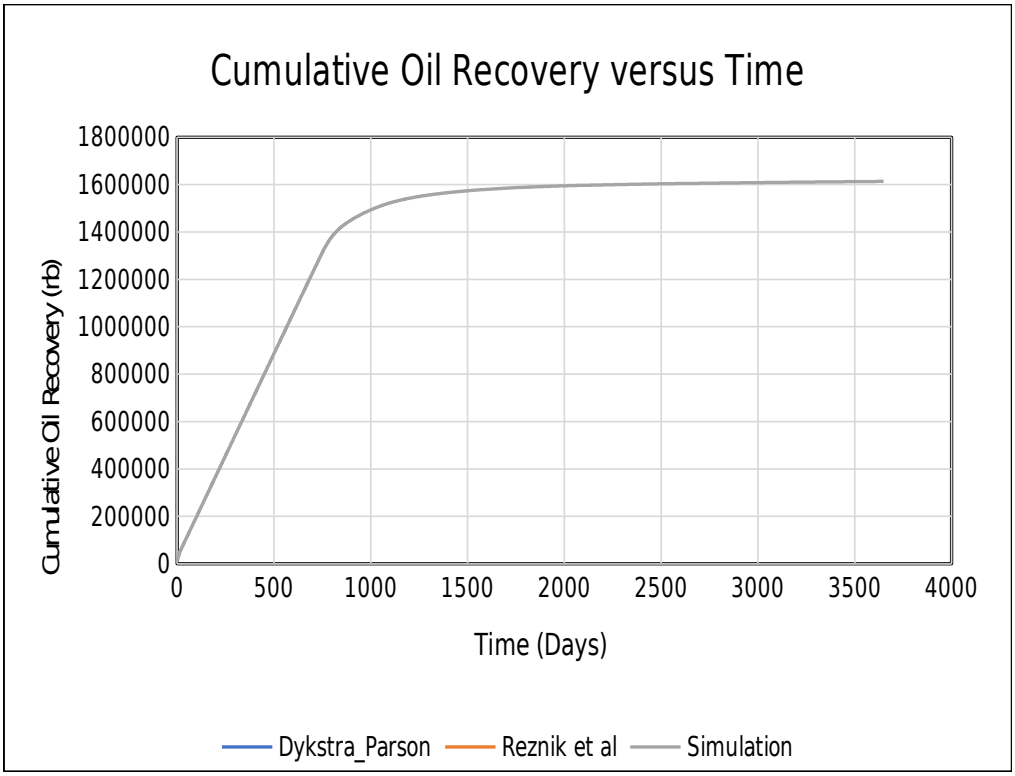

Figure 4.24 – Comparison of Cumulative oil recovery result with simulation result.

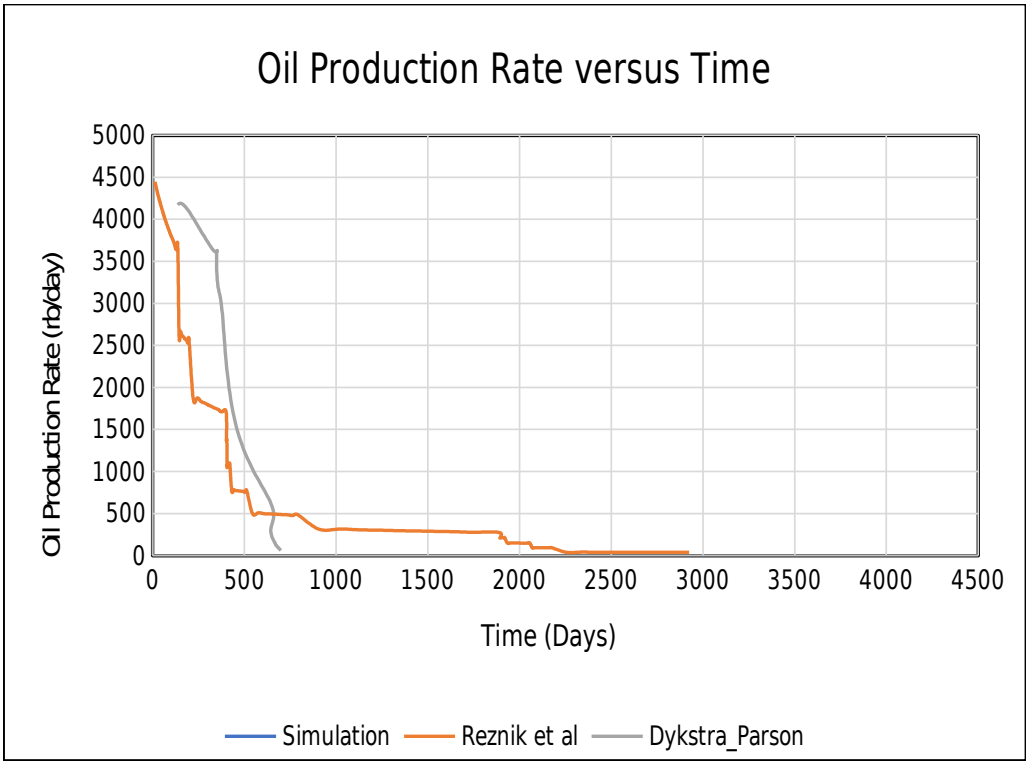

Figure 4.25 – Comparison of Oil production rate result with simulation result

Chapter 4: Conclusion and Recommendation

1.15 Conclusion

Dykstra and Parson waterflood calculation is one of the widely used waterflood performance prediction methods, especially in stratified reservoirs. It provides an analytical and discrete method for waterflood calculations. Several contributions and application of the method have been made. But the Reznik et al extension of the Dykstra-Parsons method to continuous real time analytical solution is a notable addition to the traditional idea of Dykstra and Parsons.

A computer application was successfully developed with Python to implement the calculation of the classical Dykstra-Parsons waterflood method and the Reznik et al method. The application also has menu bar for fractional flow results and curve.

To validate the result of the program, results were compared with that obtained from simulation, and was found to show a close agreement. Reznik et al result gave a closer match with simulation result than the result of Dykstra and Parson.

From Figure 4.14, at a time of 500 days, the cumulative oil recovery for Dykstra-Parson is about 600,000rb, while for Reznik et al is 850,000rb and for the simulation is 1,180,000rb. At time 700 days and 1750 days there is a coincidence in the Reznik and Simulation result. At a further time beyond 1750 days the two methods, Reznik et al and Simulation tends to maintain a constant cumulative oil recovery with about an average margin of 80,000rb.

1.16 Recommendation

The solutions of Reznik et al (1984), serves as an improvement to the classical Dykstra and Parsons solution. It is preferrable, as it provides a continuous time step, and agrees more with the numerical simulation model. Computer program can also be developed for D. Tiab (1986) extension of Dykstra-Parsons waterflood recovery method in composite reservoirs.

Bed mobilities, rather than end point mobilities as used in this study can also be incorporated in as a consideration.

APPENDIX A

Python Program Codes

```
from tkinter import*
from tkinter import filedialog
from tkinter import ttk
import pandas as pd
from tkinter.messagebox import*
from tkinter.filedialog import askopenfilename
import csv
import math
from decimal import*
import numpy as np
import random
import matplotlib
from scipy.optimize import*
import matplotlib.pyplot as plt
from pandastable.core import Table
from pandastable.data import TableModel
from datetime import datetime
#=============================================================================================
=====================================
# datetime object containing current date and time
now = datetime.now()

# dd/mm/YY H:M:S
date_time = now.strftime("%B %d, %Y | %H:%M:%S")
#print("date and time =", date_time)
#=============================================================================================
=======================================
#import os

#from idlelib.editor import EditorWindow
#def new(filename=None):
#    return EditorWindow(filename)
#=============================================================================================
=====================================
#Reservoir and Process Data initialization and definiton.

Number_of_points = 0

Length_of_bed_ft = 0

width_of_bed_ft = 0

average_porosity = 0

VISO = 0

VISW = 0

OFVF = 0

WFVF = 0
```

```
SWI = 0

SGI = 0

SOI = 0

SOR = 0

#Constant Injection Rate in STB/D
Constant_injection_rate = 0

#Injection Pressure Differential in PSI
Inj_Pressure_differential = 0

Residual_gas_saturation_unswept_area = 0

Residual_gas_saturation_swept_area = 0

Residual_gas_saturation = Residual_gas_saturation_unswept_area+Residual_gas_saturation_swept_area

Saturation_gradient = 1-SOR-SWI

#=================================================================================
==================================
root = Tk()
root.iconbitmap('STRATV.ico')
root.title('STRAT-V')

style = ttk.Style()
style.theme_use('clam')

# list the options of the style
# (Argument should be an element of TScrollbar, eg. "thumb", "trough", ...)
style.element_options("Horizontal.TScrollbar.thumb")

# configure the style
style.configure("Horizontal.TScrollbar", gripcount=0,
                background="#2196f3", darkcolor="#2196f3", lightcolor="grey",
                troughcolor="#2196f3", bordercolor="grey", arrowcolor="white")

#Create a main frame
main_frame = Frame(root,bg='#d6ebfb')
main_frame.pack(fill = BOTH, expand =1)

#Create a canvas

my_canvas = Canvas(main_frame,bg='#d6ebfb')
my_canvas.pack(side = LEFT, fill=BOTH, expand = 1)

#Add a scrollbar to the canvas
my_scrollbar = ttk.Scrollbar(main_frame, orient = VERTICAL, command = my_canvas.yview)
my_scrollbar.pack(side=RIGHT, fill = Y)
my_scrollbarx = ttk.Scrollbar(root, orient=HORIZONTAL, command=my_canvas.xview)
my_scrollbarx.pack(side=BOTTOM, fill = X)
#Configure the canvas
my_canvas.configure(xscrollcommand=my_scrollbarx.set, yscrollcommand = my_scrollbar.set)
my_canvas.bind('<Configure>', lambda e: my_canvas.configure(scrollregion = my_canvas.bbox('all')))

#Create ANOTHER frame inside the canvas
second_frame = Frame(my_canvas, bg = '#c8f1f7')
Label(root, text = 'Copyright, 2020, Lekia Prosper').pack(side = BOTTOM)
```

```
#Add that frame to a windows in the canvas
my_canvas.create_window((0,0), window = second_frame, anchor = 'nw')
#=============================================================================================
=====================================
def Load_File():
    import_file=Tk()
    import_file.iconbitmap('STRATV.ico')
    import_file.title('STRAT-V')
    import_file.geometry('500x500')
    import_file.pack_propagate(False)
    #import_file.resizable(0,0)

    #frame for Treeview
    frame3=LabelFrame(import_file,text='Data File')
    #frame3.place(height=250, width=500)
    frame3.place(relheight=0.5, relwidth=1)

    #Frame for open filedialog
    file_frame=LabelFrame(import_file, text='Open File')
    file_frame.place(height=100, width=500, rely=0.65, relx=0)

    #Buttons
    button1=Button(file_frame, text = 'Browse A File',command=lambda:file_dialog())
    button1.place(rely=0.65, relx=0.8)

    button2= Button(file_frame, text = 'Load Permeability-Porosity Data', command=lambda:
Load_Permeability_Porosity_Data())
    button2.place(rely=0.65,relx=0.4)
    button3=Button(file_frame, text = 'Load Relative Permeability Data', command=lambda:
Load_Relative_Permeability_Data())
    button3.place(rely=0.65, relx=0)

    label_file = ttk.Label(file_frame, text = 'No File Selected')
    label_file.place(rely=0,relx=0)

    #Treeview Widget
    tv1=ttk.Treeview(frame3)
    tv1.place(relheight=1, relwidth=1)

    treescrolly = Scrollbar(frame3, orient='vertical',command=tv1.yview)
    treescrollx = Scrollbar(frame3, orient='horizontal',command=tv1.xview)
    tv1.configure(xscrollcommand=treescrollx.set)
    tv1.configure(yscrollcommand=treescrolly.set)
    treescrollx.pack(side='bottom', fill='x')
    treescrolly.pack(side='right', fill='y')

    def file_dialog():
        filename= filedialog.askopenfilename(initialdir="/", title = "Select A File", filetype=(("csvfiles","*.csv"),("All Files",
"*.*")))
        label_file["text"]=filename

    def Load_Permeability_Porosity_Data():
        file_path=label_file['text']
        try:
            excel_filename=r"{}".format(file_path)
            bed_data=pd.read_csv(excel_filename)
        except ValueError:
            messagebox.showerror("Information","The file you have chosen is invalid")
            return None
        except FileNotFoundError:
            messagebox.showerror("Information", f"No such file as {file_path}")
            return None
```

```
        clear_data()
        tv1["column"]=list(bed_data.columns)
        tv1['show']='headings'
        for column in tv1['columns']:
            tv1.heading(column,text=column)
        bed_data_rows = bed_data.to_numpy().tolist()
        for row in bed_data_rows:
            tv1.insert("","end", values=row)

    def Load_Relative_Permeability_Data():
        file_path=label_file['text']
        try:
            excel_filename=r"{}".format(file_path)
            RPERM_data=pd.read_csv(excel_filename)
        except ValueError:
            messagebox.showerror("Information","The file you have chosen is invalid")
            return None
        except FileNotFoundError:
            messagebox.showerror("Information", f"No such file as {file_path}")
            return None

        clear_data()
        tv1["column"]=list(RPERM_data.columns)
        tv1['show']='headings'
        for column in tv1['columns']:
            tv1.heading(column,text=column)
        RPERM_data_rows = RPERM_data.to_numpy().tolist()
        for row in RPERM_data_rows:
            tv1.insert("","end", values=row)
        return None
    def clear_data():
        tv1.delete(*tv1.get_children())
    import_file.mainloop()
#============================================================================================
====================================
def openfile():
    filename = askopenfilename(parent=root)
    f = open(filename)
    f.read()
#============================================================================================
====================================
import pandas as pd
#============================================================================================
====================================

def fractional_flow():
    Fw_root = Tk()
    Fw_root.iconbitmap('STRATV.ico')
    Fw_root.title('STRAT-V')
    Fw_root.pack_propagate(False)
    Frame1 = LabelFrame(Fw_root, text='Result')
    Frame1.place(relheight=0.4, relwidth=0.6,rely=0, relx=0.05)
    listbox = Listbox(Frame1)
    listbox.place(relheight=0.8, relwidth=0.95,rely=0.1, relx=0.02)

    # Creating a Treeview
    Frame2 = LabelFrame(Fw_root, text='Fractional Flow Data')
    Frame2.place(relheight=0.5, relwidth=0.8,rely=0.5, relx=0.05)
    tree = ttk.Treeview(Frame2)
    tree.place(relheight=0.8, relwidth=0.95,rely=0.1, relx=0.02)
```

```
treescrolly = Scrollbar(Frame2, orient='vertical',command=tree.yview)
treescrollx = Scrollbar(Frame2, orient='horizontal',command=tree.xview)
tree.configure(xscrollcommand=treescrollx.set)
tree.configure(yscrollcommand=treescrolly.set)
treescrollx.pack(side='bottom', fill='x')
treescrolly.pack(side='right', fill='y')


SW_table = pd.DataFrame(SW, columns = ['SW'])


# Using the correlation between relative permeability ratio and water saturation

# Calculating the coefficient b
b = (np.log((KRO/KRW)[2])-np.log((KRO/KRW)[3]))/(SW[3]-SW[2])

#=========================================================================

# Calculating the coefficient a
a = (KRO/KRW)[2]*math.exp(b*SW[2])

#=========================================================================
# Calculating the fractional flow
def fw(SW):
   fw = 1/(1+a*(VISW/VISO)*np.exp(-b*SW))
   return(fw)
#=========================================================================
"" To calculate a suitable slope for the tangent to the fractional flow curve
Drawn from the initial water saturation"'

# STEP1: Generate a list of uniformly distributed random numbers from a water saturation
# greater than the initial water saturation to 1
xList = []
for i in range(0, 10000):
   x = random.uniform(SWI+0.1, 1)
   xList.append(x)
xs = np.array(xList)

# STEP2: Calculate different slopes of tangents or lines intersecting the fractional
# flow curve using the array generated in step 1 as the water saturation.
m = 1/((xs-SWI)*(1+(VISW/VISO)*a*np.exp(-b*xs)))

# STEP3: Calculate the maximum slope from different slopes generated in step 2.
# The value of this slope will be the slope of the tangent to the fractional flow
# curve.
tangent_slope=max(m)
#print('slope of the tangent line is:\n ',tangent_slope)
#=========================================================================
# Calculate the breakthrough saturation.
Saturation_at_Breakthrough = SWI + 1/tangent_slope
#print('saturation at breakthrough is:\n ', Saturation_at_Breakthrough)
#=========================================================================
# Calculating the saturation at the flood front

def funct(SWF):
   swf = SWF[0]
   F = np.empty((1))
   F[0] = ((tangent_slope*(swf-SWI)*(1+(VISW/VISO)*a*math.exp(-b*swf)))-1)
   return F
SWF_Guess = np.array([SWI+0.1])
SWF = fsolve(funct, SWF_Guess)[0]
SWF
```

```
    #=============================================================================
    # Fractional flow at the flood front
    Fwf = fw(SWF)
    Fwf
    #=============================================================================
    # Fractional flow
    Fw = fw(SW)
    Fw_table = pd.DataFrame(Fw, columns = ['Fractional Flow (Fw)'])
    #print(Fw_table)
    #=============================================================================
    # Calculating the differential of the fractional flow equation
    dfw_dSw = (VISW/VISO)*a*b*np.exp(-SW*b)/(1+(VISW/VISO)*a*np.exp(-SW*b))**2
    dfw_dSw_table = pd.DataFrame(dfw_dSw, columns = ['dFw/dSw'])
    #print(dfw_dSw_table)
    #=============================================================================
    # Generating the data for the tangent plot
    tangent = (SW-SWI)*tangent_slope
    tangent_table = pd.DataFrame(tangent, columns = ['Tangent'])
    #print(tangent_table)
    #=============================================================================
    Fractional_flow_table = pd.concat([SW_table, Fw_table, dfw_dSw_table, tangent_table], axis=1)
    #print(Fractional_flow_table)
    #=============================================================================
    # Making the plots
    def plot():
        get_ipython().run_line_magic('matplotlib', '')
        fig, ax = plt.subplots(constrained_layout=True)
        fig.set_figheight(4)
        fig.set_figwidth(7)
        fractional_flow_curve = ax.plot(SW, fw(SW), 'b', label = 'Fractional Flow (Fw)')
        tangent_curve = ax.plot(SW, tangent, 'k--')
        ax.set_ylabel("Fractional Flow (fw)",fontsize=14)
        ax.set_xlabel("Water Saturation (Sw)",fontsize=14)
        ax.set_ylim([0,1])
        ax.set_xlim([0,1])
        # twin object for two different y-axis on the same plot
        ax2=ax.twinx()
        # make a plot with different y-axis using second axis object
        dfw_dSw_curve = ax2.plot(SW, dfw_dSw, 'r', label ='dFw/dSw')
        ax2.set_ylabel("dfw/dSw",fontsize=14)
        ax.grid(True)
        ax2.legend()
        ax.legend(loc='upper left')
        ax.annotate("  (Swf, Fwf)", (SWF, Fwf))
        ax.annotate(" SwBT", (Saturation_at_Breakthrough, 1))
        plt.show()
        plt.ion()

#=============================================================================================
====================
    listbox.insert(1,"  Correlation: Kro/Krw = aexp(-bSw)")
    listbox.insert(2, '  b : ' +str(b))
    listbox.insert(3, '  a : '+ str(a))
    listbox.insert(4, '  Slope of the tangent line : ' + str(tangent_slope))
    listbox.insert(5, '  Flood Front Saturation (Swf) : '+ str(SWF))
    listbox.insert(6, '  Flood Front Fractional Flow (Fwf) : '+ str(Fwf))
    listbox.insert(7, '  Saturation at breakthrough (SwBT) : '+ str(Saturation_at_Breakthrough))

    tree["column"]=list(Fractional_flow_table.columns)
    tree['show']='headings'
    for column in tree['columns']:
        tree.heading(column,text=column)
```

```
    Fractional_flow_table_rows = Fractional_flow_table.to_numpy().tolist()
    for row in Fractional_flow_table_rows:
        tree.insert("","end", values=row)

    View_plot = Button(Fw_root, text = 'View Plot',justify = LEFT,relief= RAISED,cursor='hand2',command = plot)
    View_plot.place(rely=0.2, relx=0.7)
    Fw_root.geometry("600x500")

    Fw_root.mainloop()

def enter_inputs():
    global entries
    global Number_of_points
    global Length_of_bed_ft
    global width_of_bed_ft
    global average_porosity
    global VISO
    global VISW
    global OFVF
    global WFVF
    global SWI
    global SGI
    global SOI
    global SOR
    global Constant_injection_rate
    global Inj_Pressure_differential
    global Residual_gas_saturation_unswept_area
    global Residual_gas_saturation_swept_area
    global Residual_gas_saturation

    fields = ('Number_of_points','Length_of_bed_ft', 'width_of_bed_ft', 'average_porosity','VISO',
'VISW','OFVF','WFVF','SWI', 'SGI','SOI','SOR','Constant_injection_rate',

'Inj_Pressure_differential','Residual_gas_saturation_unswept_area','Residual_gas_saturation_swept_area','Residual_gas_s
aturation','Saturation_gradient')

    def residual_gas_saturation(entries):
        global RGSU
        global RGSS
        global RGS
        global Number_of_points
        global Length_of_bed_ft
        global width_of_bed_ft
        global average_porosity
        global VISO
        global VISW
        global OFVF
        global WFVF
        global SWI
        global SGI
        global SOI
        global SOR
        global Constant_injection_rate
        global Inj_Pressure_differential
        global Residual_gas_saturation_unswept_area
        global Residual_gas_saturation_swept_area
        global Residual_gas_saturation
        Number_of_points = float(entries['Number_of_points'].get())
        Length_of_bed_ft = float(entries['Length_of_bed_ft'].get())
        width_of_bed_ft = float(entries['width_of_bed_ft'].get())
        SWI = float(entries['SWI'].get())
        SOR = float(entries['SOR'].get())
```

```
    SGI = float(entries['SGI'].get())
    VISW = float(entries['VISW'].get())
    VISO = float(entries['VISO'].get())
    Saturation_gradient = float(entries['Saturation_gradient'].get())
    Constant_injection_rate = float(entries['Constant_injection_rate'].get())
    Inj_Pressure_differential = float(entries['Inj_Pressure_differential'].get())

    RGSU = float(entries['Residual_gas_saturation_unswept_area'].get())
    RGSS =  float(entries['Residual_gas_saturation_swept_area'].get())
    Residual_gas_saturation = float(entries['Residual_gas_saturation'].get())
    RGS = RGSU+RGSS
    RGS = ("%8.4f" % RGS).strip()
    entries['Residual_gas_saturation'].delete(0, END)
    entries['Residual_gas_saturation'].insert(0, RGS)

def Saturation_gradient(entries):
    Saturation_gradient = float(entries['Saturation_gradient'].get())
    Saturation_gradient = 1-SOR-SWI
    Saturation_gradient = ("%8.4f" % Saturation_gradient).strip()
    entries['Saturation_gradient'].delete(0, END)
    entries['Saturation_gradient'].insert(0, Saturation_gradient)

def makeform(inputs, fields):
    global entries
    entries = {}
    for field in fields:
      # print(field)
       row = Frame(inputs)
       lab = Label(row, width=22, text=field+": ", anchor='w')
       ent = Entry(row)
       ent.insert(0, "0")
       row.pack(side=TOP,
             fill=X,
             padx=5,
             pady=5)
       lab.pack(side=LEFT)
       ent.pack(side=RIGHT,
             expand=YES,padx = 100,
             fill=X)
       entries[field] = ent
    return entries

if __name__ == '__main__':
    inputs = Tk()
    inputs.iconbitmap('STRATV.ico')
    inputs.title('STRAT-V')
    ents = makeform(inputs, fields)
    b1 = Button(inputs, text='Residual gas saturation',
         command=(lambda e=ents:residual_gas_saturation(e)))
    b1.pack(side=LEFT, padx=5, pady=5)

    b2 = Button(inputs, text='Saturation gradient',
           command=(lambda e1=ents:Saturation_gradient(e1)))
    b2.pack(side=LEFT, padx=5, pady=5)
    inputs.mainloop()
Enter_input_button = Button(second_frame, text = 'Enter input data',justify = LEFT,relief=
RAISED,cursor='hand2',command = enter_inputs).grid(row=2,column =0,padx=5,pady=10)

#=================================================================================
=======================================
Label(second_frame, text='Calculate',fg = 'white', bg = '#2196f3',justify = CENTER,relief= FLAT).grid(row=0,
column=4, columnspan = 8,padx = 40,pady=10, sticky = W+E+N+S)
```

```
#Importing the Permeability Porosity distribution data
bed_data = pd.read_csv('Permeability_Porosity_distribution_data.csv')
#=============================================================================================
=====================================
#=============================================================================================
=====================================
#Importing the Relative permeability Data
import pandas as pd
RPERM_data = pd.read_csv('Oil_Water_Relative_Permeability_data.csv')

SW = np.array(RPERM_data['SW'])
KRW = np.array(RPERM_data['KRW'])
KRO = np.array(RPERM_data['KRO'])

#=============================================================================================
=====================================
def Reznik(entries):
   try:
      global RGSU
      global RGSS
      global RGS
      global Number_of_points
      global Length_of_bed_ft
      global width_of_bed_ft
      global average_porosity
      global VISO
      global VISW
      global OFVF
      global WFVF
      global SWI
      global SGI
      global SOI
      global SOR
      global Constant_injection_rate
      global Inj_Pressure_differential
      global Residual_gas_saturation_unswept_area
      global Residual_gas_saturation_swept_area
      global Residual_gas_saturation
      #Number_of_points = float(entries['Number_of_points'].get())
      Length_of_bed_ft = float(entries['Length_of_bed_ft'].get())
      width_of_bed_ft = float(entries['width_of_bed_ft'].get())
      SWI = float(entries['SWI'].get())
      SOI = float(entries['SOI'].get())
      SOR = float(entries['SOR'].get())
      SGI = float(entries['SGI'].get())
      VISW = float(entries['VISW'].get())
      VISO = float(entries['VISO'].get())
      OFVF = float(entries['OFVF'].get())
      WFVF = float(entries['WFVF'].get())
      Saturation_gradient = float(entries['Saturation_gradient'].get())
      Constant_injection_rate = float(entries['Constant_injection_rate'].get())
      Inj_Pressure_differential = float(entries['Inj_Pressure_differential'].get())

      #RGSU = float(entries['Residual_gas_saturation_unswept_area'].get())
      #RGSS =  float(entries['Residual_gas_saturation_swept_area'].get())

      import pandas as pd
      import math
      import numpy as np

      application_window = Tk()
      application_window.iconbitmap('STRATV.ico')
```

```
    application_window.title('STRAT-V')


    KRW_1_SOR = np.interp(1-SOR, SW, KRW)
    KRO_SWI = np.interp(SWI, SW, KRO)

    Water_Mobility = bed_data.PERMEABILITY*KRW_1_SOR/VISW
    Water_Mobility_table= pd.DataFrame(Water_Mobility).rename(columns={'PERMEABILITY': 'Water Mobility'})
    #Water_Mobility_table

#=============================================================================================
=====================================
    #CALCULATING THE WATER MOBILITY

    Oil_Mobility = bed_data.PERMEABILITY*KRO_SWI/VISO
    Oil_Mobility_table= pd.DataFrame(Oil_Mobility).rename(columns={'PERMEABILITY': 'Oil Mobility'})
    #Oil_Mobility_table

#=============================================================================================
==================================
    # CALCULATING THE MOBILITY RATIO, M.
    #import math
    Mobility_Ratio = Water_Mobility/Oil_Mobility
    Mobility_Ratio_table= pd.DataFrame(Mobility_Ratio).rename(columns={'PERMEABILITY': 'MOBILITY
RATIO'})
    #Mobility_Ratio_table


#=============================================================================================
====================================


    # ARRANGING THE DATA IN ORDER OF DECREASING PERMEABILITY.
    Bed_ordering_parameter=np.array(bed_data.POROSITY)*Saturation_gradient*(1+Mobility_Ratio)/Water_Mobility

    # ARRANGING THE DATA IN ORDER OF DECREASING PERMEABILITY.
    #Bed Ordering
    Bed_ordering_parameter = np.array(bed_data.POROSITY)*Saturation_gradient*(1+Mobility_Ratio)/
Water_Mobility
    Bed_ordering_parameter_table = pd.DataFrame(Bed_ordering_parameter).rename(columns={'PERMEABILITY':
'BED ORDERING PARAMETER'})
    bed_data_combine = pd.concat([bed_data,
Bed_ordering_parameter_table,Water_Mobility_table,Oil_Mobility_table,Mobility_Ratio_table], axis = 1)

    bed_data_sort = bed_data_combine.sort_values(by='BED ORDERING PARAMETER',ignore_index=True,
ascending=True)
    Average_porosity = '%.3f' % np.mean(bed_data_sort.POROSITY)

#=============================================================================================
====================================
    # Extracting input variables from data table.
    import numpy as np
    Layers = np.array(bed_data_sort['LAYER'])
    Layer_table1 = pd.DataFrame(Layers, columns=['Layers'])
    Bed_ordering_parameter = np.array(bed_data_sort['BED ORDERING PARAMETER'])
    Bed_ordering_parameter_sort_table = pd.DataFrame(Bed_ordering_parameter)
    PORO = np.array(bed_data_sort['POROSITY'])
    Porosity_sort_table = pd.DataFrame(PORO)
    permeability_array = np.array(bed_data_sort['PERMEABILITY'])

    Water_mobility_array = np.array(bed_data_sort['Water Mobility'])
    Water_mobility_sort_table = pd.DataFrame(Water_mobility_array)
```

```
    Oil_mobility_array = np.array(bed_data_sort['Oil Mobility'])
    Oil_mobility_sort_table = pd.DataFrame(Oil_mobility_array)

    Permeability_sort_table = pd.DataFrame(permeability_array)
    bed_thickness = np.array(bed_data_sort['THICKNESS'])


#=============================================================================================
====================================

    #Bed order parameter ratio of each bed to the last bed
    bed_order_ratio_list = []
    for j in range(len(Layers)):
        bed_order_ratio_to_lastbed = bed_data_sort['BED ORDERING PARAMETER'][j]/bed_data_sort['BED
ORDERING PARAMETER'].iat[-1]
        bed_order_ratio_list.append(bed_order_ratio_to_lastbed)
    bed_order_ratio=pd.DataFrame(bed_order_ratio_list)


#=============================================================================================
====================================

    #Bed order parameter ratio of each bed to the last bed
    bed_order_ratio_to_other_beds_list = []
    for j in range(len(Layers)):
        bed_order_ratio_to_otherbeds = bed_data_sort['BED ORDERING PARAMETER'].iat[-1]/bed_data_sort['BED
ORDERING PARAMETER'][j]
        bed_order_ratio_to_other_beds_list.append(bed_order_ratio_to_otherbeds)
    bed_order_ratio_to_otherbeds=pd.DataFrame(bed_order_ratio_to_other_beds_list)


#=============================================================================================
====================================
    #Flood front position of bed n when bed j has just broken through.

    last_mobility_ratio = bed_data_sort['MOBILITY RATIO'].iat[-1]

    Flood_front_position_of_bed_n_j = (-last_mobility_ratio+np.sqrt(last_mobility_ratio**2+(bed_order_ratio)*(1-
last_mobility_ratio**2)))/(1-last_mobility_ratio)
    Flood_front_position_of_bed_n_j = pd.DataFrame(Flood_front_position_of_bed_n_j).rename(columns = {0:'Flood
Front Position of the last bed at breakthrough of other beds'})


#=============================================================================================
====================================
    global Number_of_points
    #Flood front location of the last bed.
    Number_of_points = float(entries['Number_of_points'].get())
    #converting the flood front table to a list
    flood_front_of_last_bed = 0
    flood_front_of_last_bed_list = []
    if Mobility_Ratio_table.iloc[0,0] == 1:
       Flood_front_position_of_bed_n_j_list=bed_order_ratio.to_list()
       for index, position in list(enumerate(Flood_front_position_of_bed_n_j_list)):
          while flood_front_of_last_bed < Flood_front_position_of_bed_n_j_list[0]:
            # if flood_front_of_last_bed < Flood_front_position_of_bed_n_j_list[0]:
             flood_front_of_last_bed = flood_front_of_last_bed +
Flood_front_position_of_bed_n_j_list[0]/Number_of_points
             flood_front_of_last_bed_list.append(flood_front_of_last_bed)
```

```
            if(index > 0):
                while flood_front_of_last_bed >=Flood_front_position_of_bed_n_j_list[index-1] and
flood_front_of_last_bed <= Flood_front_position_of_bed_n_j_list[index]:
                    flood_front_of_last_bed = flood_front_of_last_bed + (Flood_front_position_of_bed_n_j_list[index]-
Flood_front_position_of_bed_n_j_list[index-1])/Number_of_points
                    flood_front_of_last_bed_list.append(flood_front_of_last_bed)

        else:
            Flood_front_position_of_bed_n_j_list=Flood_front_position_of_bed_n_j['Flood Front Position of the last bed at
breakthrough of other beds'].to_list()
            for index, position in list(enumerate(Flood_front_position_of_bed_n_j_list)):
                while flood_front_of_last_bed < Flood_front_position_of_bed_n_j_list[0]:
                  # if flood_front_of_last_bed < Flood_front_position_of_bed_n_j_list[0]:
                    flood_front_of_last_bed = flood_front_of_last_bed +
Flood_front_position_of_bed_n_j_list[0]/Number_of_points
                    flood_front_of_last_bed_list.append(flood_front_of_last_bed)


            if(index > 0):
                while flood_front_of_last_bed >=Flood_front_position_of_bed_n_j_list[index-1] and
flood_front_of_last_bed <= Flood_front_position_of_bed_n_j_list[index]:
                    flood_front_of_last_bed = flood_front_of_last_bed + (Flood_front_position_of_bed_n_j_list[index]-
Flood_front_position_of_bed_n_j_list[index-1])/Number_of_points
                    flood_front_of_last_bed_list.append(flood_front_of_last_bed)


        flood_front_of_last_bed_table = pd.DataFrame(flood_front_of_last_bed_list).rename(columns = {0:'Flood Front
Position of the last bed at time t'})


#=============================================================================================
=====================================

        #Calculating Real or Process time for the CIP case
        porosity_of_last_bed = bed_data_sort['POROSITY'].iat[-1]
        water_mobility_of_last_bed = bed_data_sort['Water Mobility'].iat[-1]
        Real_time_CIP =
158.064*((Length_of_bed_ft**2/Inj_Pressure_differential)*porosity_of_last_bed*Saturation_gradient/
water_mobility_of_last_bed)*(last_mobility_ratio*np.array(flood_front_of_last_bed_list) + 0.5*(1-
last_mobility_ratio)*np.array(flood_front_of_last_bed_list)**2)
        Real_time_CIP_table = pd.DataFrame(Real_time_CIP).rename(columns = {0:'Real time for constant injection
pressure'})
        #Real_time_CIP_table


#=============================================================================================
=====================================
        # Calculating breakthrough time of each bed.
        porosity_of_last_bed = bed_data_sort['POROSITY'].iat[-1]
        water_mobility_of_last_bed = bed_data_sort['Water Mobility'].iat[-1]
        breakthrough_time =
158.064*((Length_of_bed_ft**2/Inj_Pressure_differential)*porosity_of_last_bed*Saturation_gradient/
water_mobility_of_last_bed)*(last_mobility_ratio*np.array(Flood_front_position_of_bed_n_j['Flood Front Position of the
last bed at breakthrough of other beds'].to_list()) + 0.5*(1-
last_mobility_ratio)*np.array(Flood_front_position_of_bed_n_j['Flood Front Position of the last bed at breakthrough of
other beds'].to_list())**2)
        breakthrough_time_table = pd.DataFrame(breakthrough_time).rename(columns = {0:'Breakthrough time'})
        #breakthrough_time_table
```

```
#=====================================================================================
=====================================
    # Flood front position of other beds with respect to bed n
    Flood_front_location_of_other_beds_list = []
    for j in range(len(Layers)):
        aj = Mobility_Ratio[j]**2
        bed_order_of_last_bed = bed_data_sort['BED ORDERING PARAMETER'].iat[-1]
        bj = (bed_order_ratio_to_other_beds_list[j])*(2*last_mobility_ratio/(1+last_mobility_ratio))*(1-
Mobility_Ratio[j]**2)
        cj = (bed_order_ratio_to_other_beds_list[j])*((1-last_mobility_ratio)/(1+last_mobility_ratio))*(1-
Mobility_Ratio[j]**2)

        Flood_front_location_of_other_beds = (-Mobility_Ratio[j]+np.sqrt(aj+bj*np.array(flood_front_of_last_bed_list)
+cj*np.array(flood_front_of_last_bed_list)**2))/(1-Mobility_Ratio[j])

        Flood_front_location_of_other_beds_list.append(Flood_front_location_of_other_beds)
        for i in range(len(Flood_front_location_of_other_beds_list[j])):
            if Flood_front_location_of_other_beds_list[j][i] > 1:
                Flood_front_location_of_other_beds_list[j][i] = 1
    Flood_front_location_of_other_beds_table = pd.DataFrame(Flood_front_location_of_other_beds_list).transpose()
    #Flood_front_location_of_other_beds_table

#=====================================================================================
=====================================
    # Front position of other beds at breakthrough.
    Front_position_of_other_beds_at_breakthrough_list = []
    for j in range(len(Layers)):
        aj = Mobility_Ratio[j]**2
        bed_order_of_last_bed = bed_data_sort['BED ORDERING PARAMETER'].iat[-1]
        bj = (bed_order_ratio_to_other_beds_list[j])*(2*last_mobility_ratio/(1+last_mobility_ratio))*(1-
Mobility_Ratio[j]**2)
        cj = (bed_order_ratio_to_other_beds_list[j])*((1-last_mobility_ratio)/(1+last_mobility_ratio))*(1-
Mobility_Ratio[j]**2)

        Front_position_of_other_beds_at_breakthrough = (-Mobility_Ratio[j]
+np.sqrt(aj+bj*np.array(Flood_front_position_of_bed_n_j['Flood Front Position of the last bed at breakthrough of other
beds'].to_list())+cj*np.array(Flood_front_position_of_bed_n_j['Flood Front Position of the last bed at breakthrough of
other beds'].to_list())**2))/(1-Mobility_Ratio[j])

        Front_position_of_other_beds_at_breakthrough_list.append(Front_position_of_other_beds_at_breakthrough)

    Front_position_of_other_beds_at_breakthrough_table =
pd.DataFrame(Front_position_of_other_beds_at_breakthrough_list)
    #Front_position_of_other_beds_at_breakthrough_table

#=====================================================================================
=====================================
    # Flood front position of other beds with resect to bed n. This is to know how far each front has advanced beyond the
bed
    from decimal import Decimal
    Flood_front_location_of_other_beds_beyond_breakthrough_list = []
    for j in range(len(Layers)):
        aj = Mobility_Ratio[j]**2
        bed_order_of_last_bed = bed_data_sort['BED ORDERING PARAMETER'].iat[-1]
        bj = (bed_order_ratio_to_other_beds_list[j])*(2*last_mobility_ratio/(1+last_mobility_ratio))*(1-
Mobility_Ratio[j]**2)
        cj = (bed_order_ratio_to_other_beds_list[j])*((1-last_mobility_ratio)/(1+last_mobility_ratio))*(1-
Mobility_Ratio[j]**2)
```

```
        Flood_front_location_of_other_beds_beyond_breakthrough = (-Mobility_Ratio[j]
+np.sqrt(aj+bj*np.array(flood_front_of_last_bed_list)+cj*np.array(flood_front_of_last_bed_list)**2))/(1-
Mobility_Ratio[j])


Flood_front_location_of_other_beds_beyond_breakthrough_list.append(Flood_front_location_of_other_beds_beyond_br
eakthrough)
    Flood_front_location_of_other_beds_beyond_breakthrough_table =
pd.DataFrame(Flood_front_location_of_other_beds_beyond_breakthrough_list).transpose().round(4)
    #Flood_front_location_of_other_beds_beyond_breakthrough_table


#=============================================================================================
=========================================
    Property_time_list = []
    for i in range(len(Layers)):
        Property_time = 158.064*((Length_of_bed_ft**2/Inj_Pressure_differential)*bed_data_sort['POROSITY']
[i]*Saturation_gradient/bed_data_sort['Water Mobility']
[i])*(Mobility_Ratio[i]*Flood_front_location_of_other_beds_beyond_breakthrough_table[i]+0.5*(1-
Mobility_Ratio[i])*Flood_front_location_of_other_beds_beyond_breakthrough_table[i]**2)
        Property_time_list.append(Property_time)
    Property_time_table= pd.DataFrame(Property_time_list).T
    #Property_time_table


#=============================================================================================
===================================
    #Average mobility of the fluids in each bed at time t
    average_mobility_at_time_t_list = []
    for i in range(len(Layers)):
        average_mobility_at_time_t = Water_mobility_array[i]/(Mobility_Ratio[i]+(1-
Mobility_Ratio[i])*Flood_front_location_of_other_beds_beyond_breakthrough_table[i])
        average_mobility_at_time_t_list.append(average_mobility_at_time_t)
    average_mobility_at_time_t_table = pd.DataFrame(average_mobility_at_time_t_list).transpose()


#=============================================================================================
===================================
    #Superficial filter velocity of Darcy's law at time t
    Superficial_filter_velocity_list = []
    for i in range(len(Layers)):
        Superficial_filter_velocity = (Inj_Pressure_differential/Length_of_bed_ft)*average_mobility_at_time_t_table[i]
        Superficial_filter_velocity_list.append(Superficial_filter_velocity)
    Superficial_filter_velocity_table = pd.DataFrame(Superficial_filter_velocity_list).transpose()


#=============================================================================================
===================================
    #Real time actual linear velocity of the flood front.
    actual_linear_velocity_list = []
    for i in range(len(Layers)):
        actual_linear_velocity  = Superficial_filter_velocity_table[i]/(bed_data_sort['POROSITY'][i]*Saturation_gradient)
        actual_linear_velocity_list.append(actual_linear_velocity)
    actual_linear_velocity_table = pd.DataFrame(actual_linear_velocity_list).transpose()


#=============================================================================================
===================================
    # Instantaneous volumetric flow rate of water into bed.
    instantaneous_volumetric_flowrate_of_water_list = []
    for i in range(len(Layers)):
```

```
        instantaneous_volumetric_flowrate_of_water =
0.0011267*width_of_bed_ft*bed_thickness[i]*Superficial_filter_velocity_table[i]
        instantaneous_volumetric_flowrate_of_water_list.append(instantaneous_volumetric_flowrate_of_water)
    instantaneous_volumetric_flowrate_of_water_table =
pd.DataFrame(instantaneous_volumetric_flowrate_of_water_list).transpose()
    #instantaneous_volumetric_flowrate_of_water_table


#=============================================================================================
====================================
    # Instantaneous volumetric flow rate of oil into bed.
    instantaneous_volumetric_flowrate_of_oil_list = []
    for i in range(len(Layers)):
        instantaneous_volumetric_flowrate_of_oil =
0.0011267*width_of_bed_ft*bed_thickness[i]*Superficial_filter_velocity_table[i]/((1-bed_data_sort['MOBILITY
RATIO'][i])*Flood_front_location_of_other_beds_table[i]+bed_data_sort['MOBILITY RATIO'][i])
        instantaneous_volumetric_flowrate_of_oil_list.append(instantaneous_volumetric_flowrate_of_oil)
    instantaneous_volumetric_flowrate_of_oil_table =
pd.DataFrame(instantaneous_volumetric_flowrate_of_oil_list).transpose()


#=============================================================================================
====================================
    # Total flow rate for each bed.
    Constant_total_injection_rate_list = []
    for i in range(len(Layers)):
        Constant_total_injection_rate = np.sum(instantaneous_volumetric_flowrate_of_water_table[i])
        Constant_total_injection_rate_list.append(Constant_total_injection_rate)
    Constant_total_injection_rate_table = pd.DataFrame(Constant_total_injection_rate_list)
    Constant_total_injection_rate_for_all_beds = Constant_total_injection_rate_table.sum(axis=0).values[0]
    #Constant_total_injection_rate_for_all_beds

#=============================================================================================
====================================
    # Get the count of ones in each column at a given time
    number_of_ones_list = {}
    for i in range(len(Real_time_CIP_table)+1):
        number_of_ones_list[i] = Flood_front_location_of_other_beds_table[0:i].isin([1]).sum().to_frame().T.iloc[0,:]
        #number_of_ones_list.append(number_of_ones)
    number_of_ones_table = pd.DataFrame.from_dict(number_of_ones_list).T

    # returns the column with lowest count of 1 at a given time period. this represents the dynamic bed.
    number_of_ones_table['Dynamic_bed'] = number_of_ones_table.idxmin(axis=1)
    pd.set_option("display.max_rows", None, "display.max_columns", None)


#=============================================================================================
====================================
    dynamic_bed = number_of_ones_table['Dynamic_bed']
    dynamic_bed_table = pd.DataFrame(dynamic_bed).rename(columns = {'Dynamic_bed':'Dynamic Bed'})
    #dynamic_bed_table = pd.DataFrame(number_of_ones_table['Dynamic_bed'], columns = ['Dynamic bed'])
    water_flow_rate_and_Dynamic_bed=pd.concat([instantaneous_volumetric_flowrate_of_water_table,dynamic_bed],
axis = 1)
    #print(dynamic_bed_table)

#=============================================================================================
====================================
    #just before breakthrough of the dynamic bed
    sum_water_flowrate_before_breakthrough_of_dynamic_bed_list = []
    for i in range(len(Real_time_CIP_table)):
        for j in range(len(Layers)):
```

```
            if dynamic_bed[i] == j:
                sum_water_flowrate_before_breakthrough_of_dynamic_bed =
instantaneous_volumetric_flowrate_of_water_table.iloc[i,0:j].sum(axis = 0)

sum_water_flowrate_before_breakthrough_of_dynamic_bed_list.append(sum_water_flowrate_before_breakthrough_of_d
ynamic_bed)
    sum_water_flowrate_before_breakthrough_of_dynamic_bed_table =
pd.DataFrame(sum_water_flowrate_before_breakthrough_of_dynamic_bed_list).rename(columns = {0:'Sum water
flowrate before breakthrough of dynamic bed'})
    #sum_water_flowrate_before_breakthrough_of_dynamic_bed_table


#=================================================================================================
====================================
    #just before breakthrough of the dynamic bed
    sum_oil_flowrate_before_breakthrough_of_dynamic_bed_list = []
    for i in range(len(Real_time_CIP_table)):
        for j in range(len(Layers)):
            if dynamic_bed[i] == j:
                sum_oil_flowrate_before_breakthrough_of_dynamic_bed =
instantaneous_volumetric_flowrate_of_oil_table.iloc[i,j:len(Layers)+1].sum(axis = 0)

sum_oil_flowrate_before_breakthrough_of_dynamic_bed_list.append(sum_oil_flowrate_before_breakthrough_of_dynam
ic_bed)
    sum_oil_flowrate_before_breakthrough_of_dynamic_bed_table =
pd.DataFrame(sum_oil_flowrate_before_breakthrough_of_dynamic_bed_list).rename(columns = {0:'Sum oil flowrate
before breakthrough of dynamic bed'})
    sum_oil_flowrate_before_breakthrough_of_dynamic_bed_table


#=================================================================================================
====================================
    # Instantaneous producing WOR, defined at xj = l, for all j, at time t just before breakthrough of the dynamic bed
    Instantaneous_producing_Water_Oil_Ratio_before_breakthrough_of_dynamic_bed =
(np.array(sum_water_flowrate_before_breakthrough_of_dynamic_bed_list)/
np.array(sum_oil_flowrate_before_breakthrough_of_dynamic_bed_list))
    Instantaneous_producing_Water_Oil_Ratio_before_breakthrough_of_dynamic_bed_table =
pd.DataFrame(Instantaneous_producing_Water_Oil_Ratio_before_breakthrough_of_dynamic_bed).rename(columns =
{0:'Instantaneous producing Water Oil ratio before breakthrough of dynamic bed'})
    #Instantaneous_producing_Water_Oil_Ratio_before_breakthrough_of_dynamic_bed_table

    # Instantaneous producing Water cut, defined at xj = l, for all j, at time t just before breakthrough of the dynamic bed
    Instantaneous_producing_Water_cut = np.array(sum_water_flowrate_before_breakthrough_of_dynamic_bed_list)/
(np.array(sum_oil_flowrate_before_breakthrough_of_dynamic_bed_list)
+np.array(sum_water_flowrate_before_breakthrough_of_dynamic_bed_list))
    Instantaneous_producing_Water_cut_table = pd.DataFrame(Instantaneous_producing_Water_cut).rename(columns =
{0:'Instantaneous producing Water cut'})
    #Instantaneous_producing_Water_cut_table


#=================================================================================================
====================================
    # Ultimate recoverable oil per bed
    global Ultimate_recoverable_oil_per_bed_table
    Ultimate_recoverable_oil_per_bed_list = []
    for i in range(len(Layers)):
        Ultimate_recoverable_oil_per_bed = Length_of_bed_ft*width_of_bed_ft*bed_data_sort['THICKNESS']
[i]*bed_data_sort['POROSITY'][i]*Saturation_gradient
        Ultimate_recoverable_oil_per_bed_list.append(Ultimate_recoverable_oil_per_bed)
    Ultimate_recoverable_oil_per_bed_table = pd.DataFrame(Ultimate_recoverable_oil_per_bed_list).rename(columns
= {0:'Ultimate recoverable oil per bed'})
```

```
    #Ultimate_recoverable_oil_per_bed_table

#=====================================================================================
=====================================
    # Total recoverable oil in place for the entire system of n beds.

    Total_recoverable_oil_in_place = Ultimate_recoverable_oil_per_bed_table.sum(axis = 0).values[0]

    #Total_recoverable_oil_in_place


#=====================================================================================
=====================================
    # Product of flood front location and ultimate recovery at per bed.
    Product_of_flood_front_location_and_ultimate_recovery_list = []
    for j in range(len(Layers)):
        Product_of_flood_front_location_and_ultimate_recovery =
Flood_front_location_of_other_beds_beyond_breakthrough_table[j]*Ultimate_recoverable_oil_per_bed_table.iloc[j,0]

Product_of_flood_front_location_and_ultimate_recovery_list.append(Product_of_flood_front_location_and_ultimate_rec
overy)
    Product_of_flood_front_location_and_ultimate_recovery_table =
pd.DataFrame(Product_of_flood_front_location_and_ultimate_recovery_list).T
    #Product_of_flood_front_location_and_ultimate_recovery_table


#=====================================================================================
=====================================
    # cumulative oil recovered from all beds at time t .
    # Term 1
    cumulative_oil_recovered_at_time_t_list = []
    for i in range(len(Real_time_CIP_table)):
        for j in range(len(Layers)):
            if dynamic_bed[i] == j:
                cumulative_oil_recovered_at_time_t = np.array(Ultimate_recoverable_oil_per_bed_list)[0:j].sum(axis = 0)
                cumulative_oil_recovered_at_time_t_list.append(cumulative_oil_recovered_at_time_t)
    cumulative_oil_recovered_at_time_t_table = pd.DataFrame(cumulative_oil_recovered_at_time_t_list)
    #cumulative_oil_recovered_at_time_t_table

    # Term 2
    cumulative_oil_recovered_and_flood_front_location_at_time_t_list = []
    for k in range(len(Real_time_CIP_table)):
        for l in range(len(Layers)):
            if dynamic_bed[k] == l:
                cumulative_oil_recovered_and_flood_front_location_at_time_t =
Product_of_flood_front_location_and_ultimate_recovery_table.iloc[k,l:len(Layers)+1].sum(axis = 0)

cumulative_oil_recovered_and_flood_front_location_at_time_t_list.append(cumulative_oil_recovered_and_flood_front_l
ocation_at_time_t)
    cumulative_oil_recovered_and_flood_front_location_at_time_t_table =
pd.DataFrame(cumulative_oil_recovered_and_flood_front_location_at_time_t_list)
    #cumulative_oil_recovered_and_flood_front_location_at_time_t_table

    # Cumulative oil recovered from all beds at time t
    Cumulative_oil_recovered_from_all_beds = cumulative_oil_recovered_at_time_t_table +
cumulative_oil_recovered_and_flood_front_location_at_time_t_table
    Cumulative_oil_recovered_from_all_beds_table =
pd.DataFrame(Cumulative_oil_recovered_from_all_beds).rename(columns = {0:'Cumulative oil recovered from all beds
at time t'})

    #Cumulative_oil_recovered_from_all_beds_table
```

```
#=================================================================================================
    # Vertical coverage at time t
    Vertical_coverage_at_time_t = Cumulative_oil_recovered_from_all_beds/Total_recoverable_oil_in_place
    Vertical_coverage_at_time_t_table = pd.DataFrame(Vertical_coverage_at_time_t).rename(columns = {0:'Vertical coverage at time t'})
    #Vertical_coverage_at_time_t_table


#=================================================================================================
    # Cumumlative water oil ratio for constant injecction rate case.
    Cumumlative_water_oil_ratio_for_CIR = ((Constant_total_injection_rate_for_all_beds*Real_time_CIP_table['Real time for constant injection pressure']) - Cumulative_oil_recovered_from_all_beds_table['Cumulative oil recovered from all beds at time t'])/Cumulative_oil_recovered_from_all_beds_table['Cumulative oil recovered from all beds at time t']
    Cumumlative_water_oil_ratio_for_CIR_table = pd.DataFrame(Cumumlative_water_oil_ratio_for_CIR).rename(columns = {0:'Cumumlative water oil ratio for constant injection rate'})
    #Cumumlative_water_oil_ratio_for_CIR_table

#=================================================================================================
    # Cumumlative water oil ratio for constant injecction Pressure case.
    # First get the product of difference between the real time and the breakthrough time, the bed thickness and water mobility.
    product_1_list  = []
    for j in range(len(Layers)):
        product_1 = (Real_time_CIP_table['Real time for constant injection pressure'].to_numpy() - breakthrough_time_table['Breakthrough time'][j])*bed_data_sort['THICKNESS'][j]*instantaneous_volumetric_flowrate_of_water_table[j].to_numpy()
        product_1_list.append(product_1)
    product_1_table = pd.DataFrame(product_1_list).T
    #product_1_table

    Cumumlative_water_oil_ratio_for_CIP_list = []
    for i in range(len(Real_time_CIP_table)):
        for j in range(len(Layers)):
            if dynamic_bed[i] == j:
                Cumumlative_water_oil_ratio_for_CIP = ((width_of_bed_ft*Inj_Pressure_differential/Length_of_bed_ft)*product_1_table.iloc[i, 0:j].sum(axis = 0))/Cumulative_oil_recovered_from_all_beds_table['Cumulative oil recovered from all beds at time t'][i]
                Cumumlative_water_oil_ratio_for_CIP_list.append(Cumumlative_water_oil_ratio_for_CIP)
    Cumumlative_water_oil_ratio_for_CIP_table = pd.DataFrame(Cumumlative_water_oil_ratio_for_CIP_list).rename(columns = {0:'Cumumlative water oil ratio for constant injection pressure'})
    #Cumumlative_water_oil_ratio_for_CIP_table

#=================================================================================================
    # The cumulative water injected into bed i to time t, is given for the Constant injection pressure case by;
    cumulative_water_injected_list = []
    #for i in range(len(Real_time_CIP_table)):
    for j in range(len(Layers)):
        cumulative_water_injected_1 = Flood_front_location_of_other_beds_beyond_breakthrough_table[j]*Ultimate_recoverable_oil_per_bed_table['Ultimate recoverable oil per bed'][j]
        cumulative_water_injected_2 = Ultimate_recoverable_oil_per_bed_table['Ultimate recoverable oil per bed'][j] + (width_of_bed_ft*Inj_Pressure_differential/Length_of_bed_ft)*product_1_table[j]
        #cumulative_water_injected_list_1.append(cumulative_water_injected_1)
        for i in range(len(Real_time_CIP_table)):
```

```
            if Real_time_CIP_table['Real time for constant injection pressure'][i] <= breakthrough_time[j]:

                cumulative_water_injected_list.append(cumulative_water_injected_1)
                #cumulative_water_injected =
Flood_front_location_of_other_beds_beyond_breakthrough_table.iloc[:,j]*Ultimate_recoverable_oil_per_bed_table[0][j]

            else:
                cumulative_water_injected_list.append(cumulative_water_injected_2)
            break
           # cumulative_water_injected = Ultimate_recoverable_oil_per_bed_table[0][j] +
(width_of_bed_ft*Inj_Pressure_differential/Length_of_bed_ft)*product_1_table[j]

    cumulative_water_injected_table = pd.DataFrame(cumulative_water_injected_list).T
    cumulative_water_injected_table



#=======================================================================================
=====================================

    global General
    global Flood_front_location_of_other_beds_table_time
    global Front_position_of_other_beds_at_breakthrough_table_time
    global Flood_front_location_of_other_beds_beyond_breakthrough_table_time
    global average_mobility_at_time_t_table_time
    global Superficial_filter_velocity_table_time
    global actual_linear_velocity_table_time
    global instantaneous_volumetric_flowrate_of_water_table_time
    global instantaneous_volumetric_flowrate_of_oil_table_time

    # TABLE OF ALL OBTAINED VALUES.
    General
=pd.concat([Layer_table1,breakthrough_time_table,Flood_front_position_of_bed_n_j,Ultimate_recoverable_oil_per_bed
_table,
                Front_position_of_other_beds_at_breakthrough_table.rename(columns=lambda x: str(x)+'
Flood_Front_position_of_beds_at_breakthrough'),

flood_front_of_last_bed_table,Real_time_CIP_table,dynamic_bed_table,sum_water_flowrate_before_breakthrough_of_d
ynamic_bed_table,

sum_oil_flowrate_before_breakthrough_of_dynamic_bed_table,Instantaneous_producing_Water_Oil_Ratio_before_break
through_of_dynamic_bed_table,

Instantaneous_producing_Water_cut_table,cumulative_oil_recovered_at_time_t_table,Cumulative_oil_recovered_from_a
ll_beds_table,

Vertical_coverage_at_time_t_table,Cumumlative_water_oil_ratio_for_CIR_table,Cumumlative_water_oil_ratio_for_CIP
_table,
                Flood_front_location_of_other_beds_table.rename(columns=lambda x: str(x)+'
Flood_front_location_of_beds'),
                Flood_front_location_of_other_beds_beyond_breakthrough_table.rename(columns=lambda x: str(x)+'
Flood_front_location_of_beds_beyond_breakthrough'),
                Property_time_table.rename(columns=lambda x: str(x)+' Property Time'),
                average_mobility_at_time_t_table.rename(columns=lambda x: str(x)+' Average_mobility'),
                Superficial_filter_velocity_table.rename(columns=lambda x: str(x)+' Superficial_filter_velocity'),
                actual_linear_velocity_table.rename(columns=lambda x: str(x)+' Actual_linear_velocity'),
                instantaneous_volumetric_flowrate_of_water_table.rename(columns=lambda x: str(x)+'
Instantaneous_volumetric_flowrate_of_water'),
                instantaneous_volumetric_flowrate_of_oil_table.rename(columns=lambda x: str(x)+'
Instantaneous_volumetric_flowrate_of_oil'),
                cumulative_water_injected_table.rename(columns=lambda x: str(x)+' Cumulative Water Injected')
```

```
                    ],axis=1)
    except ZeroDivisionError:
        messagebox.showerror("Omission", "Enter a value of Number of points other than zero")
        return None
    except _tkinter.TclError:
        return None

#=============================================================================================
============================================================

    def general():

        class MyTable(Table):

            def __init__(self, parent=None, **kwargs):
                Table.__init__(self, parent, **kwargs)
                return

        class MyApp(Frame):

            def __init__(self, parent=None):
                self.parent = parent
                Frame.__init__(self)
                #self.main = self.master
                self.application_window = self.master
                #self.main.geometry('800x600+200+100')
                self.application_window.title('Reznik et al General data and graph')
                f = Frame(self.application_window)
                f.pack(side=RIGHT,expand=1)
                pt = make_table(f)
                bp = Frame(self.application_window)
                bp.pack(side=TOP)

                return

        def make_table(table_frame1, **kwds):

            df = General
            pt = MyTable(table_frame1, dataframe=df, **kwds )
            pt.show()
            return pt
        def test1():
            """just make a table"""

            t = Toplevel()
            fr = Frame(t)
            fr.pack(fill=BOTH, expand=1)
            pt = make_table(fr)
            return

        def select_test():
            """cell selection and coloring"""

            #t = Toplevel()
            #fr = Frame(t)
            fr = table_frame1
            #fr.pack(fill=BOTH, expand=1)
            pt = Table(fr)
            pt.show()
            pt.General
            pt.resetIndex(ask=False)
            pt.columncolors = {'c':'#fbf1b8'}
```

```
        df = pt.model.df


        mask_1 = df.a<7
        pt.setColorByMask('a', mask_1, '#337ab7')
        colors = {'red':'#f34130','blue':'blue'}
        for l in df.label.unique():
            mask = df['label']==l
            pt.setColorByMask('label', mask, l)
        pt.redraw()
        return

    def multiple_tables():
        """make many tables in one frame"""

        t = Toplevel(height=800)
        r=0;c=0
        for i in range(6):
            fr = Frame(t)
            fr.grid(row=r,column=c)
            pt = make_table(fr, showtoolbar=True, showstatusbar=True)
            c+=1
            if c>2:
                c=0
                r+=1
        return

    app = MyApp()
    app.mainloop()


#=============================================================================================
====================================
   table_frame1 = Frame(application_window)
   table_frame1.place(relheight = 1, relwidth =1)
   Button(table_frame1, text = 'View data',bg = '#337ab7',fg = 'white',justify = LEFT,relief=
RAISED,cursor='hand2',command = general).grid(row=2,column =2,padx=5,pady=5,stick = W)


#=============================================================================================
========================================
   application_window.geometry("200x150")
   application_window.mainloop()
#=============================================================================================
==========


#=============================================================================================
========================================

# Extracting input variables from data table.

# ARRANGING THE DATA IN ORDER OF DECREASING PERMEABILITY.
bed_data_sort = bed_data.sort_values(by='PERMEABILITY', ascending=False)

PORO = np.array(bed_data_sort['POROSITY'])
permeability_array = np.array(bed_data_sort['PERMEABILITY'])
h = np.array(bed_data_sort['THICKNESS'])
SW = np.array(RPERM_data['SW'])
KRW = np.array(RPERM_data['KRW'])
KRO = np.array(RPERM_data['KRO'])
```

```
#=============================================================================================
=====================================
#This code calculates the permeability ratio, ki/kn
List_of_permeability_ratio = []
for permeability_index in range(len(permeability_array)):
    List_of_permeability_ratio_subset = [][:-permeability_index]
    for index,permeability in enumerate(permeability_array):
        if permeability_index <= index:
            permaebility_ratio = permeability/permeability_array[permeability_index]
            List_of_permeability_ratio_subset.append(permaebility_ratio)
    List_of_permeability_ratio.append(List_of_permeability_ratio_subset)

List_of_permeability_ratio_DataTable = pd.DataFrame(List_of_permeability_ratio).transpose()
#=============================================================================================
=====================================
KRW_1_SOR = np.interp(1-SOR, SW, KRW)
KRO_SWI = np.interp(SWI, SW, KRO)

# Calculating the average porosity
def average_porosity():
    Average_porosity = '%.3f' % np.mean(bed_data_sort.POROSITY)
    Label(second_frame, text= str(Average_porosity),justify = LEFT, relief = SUNKEN).grid(row = 3, column = 6,padx =
40,pady=5, sticky =W)
Button(second_frame,text='Average Porosity',bg = '#337ab7',fg = 'white',cursor='hand2',
command=average_porosity).grid(row=2,column=6,padx = 40,pady=10, sticky =W)

def relative_perm_1_SOR(entries):
    SOR = float(entries['SOR'].get())
    KRW_1_SOR = '%.3f' % np.interp(1-SOR, SW, KRW)
    Label(second_frame, text=  str(KRW_1_SOR),justify = LEFT, relief = SUNKEN).grid(row = 3, column = 7,padx =
40,pady=5, sticky =W)
    #return KRW_1_SOR
Button(second_frame,text='Relative Permeability at 1-SOR',bg = '#337ab7',fg =
'white',cursor='hand2',command=(lambda: relative_perm_1_SOR(entries))).grid(row=2,column=7,padx = 40,pady=10,
sticky =W)

def relative_perm_SWI(entries):
    SWI = float(entries['SWI'].get())
    KRO_SWI = '%.3f' % np.interp(SWI, SW, KRO)
    Label(second_frame, text= str(KRO_SWI),justify = LEFT, relief = SUNKEN).grid(row = 5, column = 6,padx =
40,pady=5, sticky =W)
    #return KRO_SWI
Button(second_frame,text='Relative Permeability at Initial Water Saturation',bg = '#337ab7',fg =
'white',cursor='hand2',command=(lambda: relative_perm_SWI(entries))).grid(row=4,column=6,padx = 40,pady=10,
sticky =W)

def mobility_ratio(entries):
    SWI = float(entries['SWI'].get())
    VISW = float(entries['VISW'].get())
    VISO = float(entries['VISO'].get())
    SOR = float(entries['SOR'].get())
    KRW_1_SOR = np.interp(1-SOR, SW, KRW)
    KRO_SWI = np.interp(SWI, SW, KRO)
    Mobility_Ratio =  KRW_1_SOR*VISO/(KRO_SWI*VISW)
    Label(second_frame, text= str(Mobility_Ratio),justify = LEFT, relief = SUNKEN).grid(row = 7, column = 7,padx =
40,pady=5, sticky =W)
    #return Mobility_Ratio
Button(second_frame,text='Mobility Ratio',bg = '#337ab7',fg = 'white',cursor='hand2',command=(lambda:
mobility_ratio(entries))).grid(row=6,column=7,padx = 40,pady=10, sticky =W)

def areal_sweep_efficiency_at_breakthrough(entries):
    SWI = float(entries['SWI'].get())
```

```
    VISW = float(entries['VISW'].get())
    VISO = float(entries['VISO'].get())
    SOR = float(entries['SOR'].get())
    KRW_1_SOR = np.interp(1-SOR, SW, KRW)
    KRO_SWI = np.interp(SWI, SW, KRO)
    Mobility_Ratio = KRW_1_SOR*VISO/(KRO_SWI*VISW)
    Areal_sweep_efficiency_at_breakthrough =
0.54602036+(0.03170817/Mobility_Ratio)+(0.30222997/math.exp(Mobility_Ratio)-0.0050969*Mobility_Ratio)
    Label(second_frame, text= str(Areal_sweep_efficiency_at_breakthrough),justify = LEFT,relief = SUNKEN).grid(row
= 9, column = 6,padx = 40,pady=5, sticky =W)
    #return Areal_sweep_efficiency_at_breakthrough
Button(second_frame,text='Areal sweep efficiency at breakthrough',bg = '#337ab7',fg =
'white',cursor='hand2',command=(lambda: areal_sweep_efficiency_at_breakthrough(entries))).grid(row=8,column=6,padx
= 40,pady=10, sticky =W)

def area_acres(entries):
    global Length_of_bed_ft
    global width_of_bed_ft
    Length_of_bed_ft = float(entries['Length_of_bed_ft'].get())
    width_of_bed_ft = float(entries['width_of_bed_ft'].get())
    Area_acres =  Length_of_bed_ft*width_of_bed_ft/43560
    Label(second_frame, text= str(Area_acres)+ ' acres',justify = LEFT,relief = SUNKEN).grid(row = 9, column = 7,padx
= 40,pady=5, sticky =W)
    #return Area_acres
Button(second_frame,text='Area of the reservoir bed',bg = '#337ab7',fg = 'white',cursor='hand2',command=(lambda:
area_acres(entries))).grid(row=8,column=7,padx = 40,pady=10, sticky =W)

def gross_rock_volume(entries):
    global Length_of_bed_ft
    global width_of_bed_ft
    Length_of_bed_ft = float(entries['Length_of_bed_ft'].get())
    width_of_bed_ft = float(entries['width_of_bed_ft'].get())
    Area_acres = Length_of_bed_ft*width_of_bed_ft/43560
    Gross_rock_volume_acre_ft =  Area_acres*bed_data_sort.THICKNESS.sum()
    Label(second_frame, text= str(Gross_rock_volume_acre_ft)+ ' acres-ft',justify = LEFT,relief = SUNKEN).grid(row =
5, column = 7,padx = 40,pady=5, sticky =W)
    #return Gross_rock_volume_acre_ft
Button(second_frame,text='Gross rock volume',bg = '#337ab7',fg = 'white',cursor='hand2',command=(lambda:
gross_rock_volume(entries))).grid(row=4,column=7,padx = 40,pady=10, sticky =W)

def displacement_efficiency(entries):
    global SGI
    global SWI
    global SOR
    global Displacement_efficiency
    SWI = float(entries['SWI'].get())
    SOR = float(entries['SOR'].get())
    SGI = float(entries['SGI'].get())
    Displacement_efficiency =  (1-SWI-SGI-SOR)/(1-SWI-SGI)
    Label(second_frame, text= str(Displacement_efficiency),justify = LEFT,relief = SUNKEN).grid(row = 7, column =
6,padx = 40,pady=5, sticky =W)
    #return Displacement_efficiency
Button(second_frame,text='Displacement efficiency',bg = '#337ab7',fg = 'white',cursor='hand2',command=(lambda:
displacement_efficiency(entries))).grid(row=6,column=6,padx = 40,pady=10, sticky =W)

def areal_sweep_efficiency(entries):
    global SGI
    global SWI
    global SOR
    global SGI
    global Constant_injection_rate
    global Inj_Pressure_differential
```

```
    SWI = float(entries['SWI'].get())
    SOR = float(entries['SOR'].get())
    SGI = float(entries['SGI'].get())
    VISW = float(entries['VISW'].get())
    VISO = float(entries['VISO'].get())
    Constant_injection_rate = float(entries['Constant_injection_rate'].get())
    Inj_Pressure_differential = float(entries['Inj_Pressure_differential'].get())
    KRW_1_SOR = np.interp(1-SOR, SW, KRW)
    KRO_SWI = np.interp(SWI, SW, KRO)
    Mobility_Ratio = KRW_1_SOR*VISO/(KRO_SWI*VISW)
    Areal_sweep_efficiency_at_breakthrough =
0.54602036+(0.03170817/Mobility_Ratio)+(0.30222997/math.exp(Mobility_Ratio)-0.0050969*Mobility_Ratio)
    Displacement_efficiency = (1-SWI-SGI-SOR)/(1-SWI-SGI)
    Areal_sweep_efficiency =  Areal_sweep_efficiency_at_breakthrough+0.2749*np.log((1/Displacement_efficiency))
    Label(second_frame, text= str(Areal_sweep_efficiency),justify = LEFT,relief = SUNKEN).grid(row = 13, column =
6,padx = 40,pady=10, sticky =W)
    #return Areal_sweep_efficiency
Button(second_frame,text='Areal sweep efficiency',bg = '#337ab7',fg = 'white',cursor='hand2',command=(lambda:
areal_sweep_efficiency(entries))).grid(row=12,column=6,padx = 40,pady=10, sticky =W)
#=============================================================================================
=============================================================


# EXTRACTING THE SORTED LAYER COLUMN
Layer_column = bed_data_sort['LAYER'].to_numpy()
Layer_table =  pd.DataFrame(Layer_column, columns = ['Layers'])
#=============================================================================================
=============================================================================
#Calculating the oil mobility ratio
#=============================================================================================
====================================
#This code calculates the list of waterflood front location as each layer breaksthrough
def D_tables(entries):
    global All_tables
    global RGSU
    global RGSS
    global RGS
    global Number_of_points
    global Length_of_bed_ft
    global width_of_bed_ft
    global average_porosity
    global VISO
    global VISW
    global OFVF
    global WFVF
    global SWI
    global SGI
    global SOI
    global SOR
    global Constant_injection_rate
    global Inj_Pressure_differential
    global Residual_gas_saturation_unswept_area
    global Residual_gas_saturation_swept_area
    global Residual_gas_saturation
    Number_of_points = float(entries['Number_of_points'].get())
    Length_of_bed_ft = float(entries['Length_of_bed_ft'].get())
    width_of_bed_ft = float(entries['width_of_bed_ft'].get())
    SWI = float(entries['SWI'].get())
    SOR = float(entries['SOR'].get())
    SOI = float(entries['SOI'].get())
    SGI = float(entries['SGI'].get())
```

```
    VISW = float(entries['VISW'].get())
    VISO = float(entries['VISO'].get())
    OFVF = float(entries['OFVF'].get())
    WFVF = float(entries['WFVF'].get())
    Saturation_gradient = float(entries['Saturation_gradient'].get())
    Constant_injection_rate = float(entries['Constant_injection_rate'].get())
    Inj_Pressure_differential = float(entries['Inj_Pressure_differential'].get())

    import pandas as pd
    import math
    import numpy as np

    D_window = Tk()
    D_window.iconbitmap('STRATV.ico')
    D_window.title('STRAT-V')

    KRW_1_SOR = np.interp(1-SOR, SW, KRW)
    KRO_SWI = np.interp(SWI, SW, KRO)
    Oil_Mobility =  permeability_array*KRO_SWI/VISO
    Oil_Mobility_table = pd.DataFrame(Oil_Mobility, columns = ['Oil Mobility'])

    Water_Mobility =  permeability_array*KRW_1_SOR/VISW
    Water_Mobility_table = pd.DataFrame(Water_Mobility, columns = ['Water Mobility'])

    Front_Location_list = []
    Mobility_Ratio = KRW_1_SOR*VISO/(KRO_SWI*VISW)
    Areal_sweep_efficiency_at_breakthrough =
0.54602036+(0.03170817/Mobility_Ratio)+(0.30222997/math.exp(Mobility_Ratio)-0.0050969*Mobility_Ratio)
    Displacement_efficiency = (1-SWI-SGI-SOR)/(1-SWI-SGI)
    import numpy as np
    Areal_sweep_efficiency =  Areal_sweep_efficiency_at_breakthrough+0.2749*np.log((1/Displacement_efficiency))

    #Average_porosity = np.mean(bed_data_sort.POROSITY)
    Average_porosity = np.mean(bed_data_sort.POROSITY)
    Area_acres = Length_of_bed_ft*width_of_bed_ft/43560
    Gross_rock_volume_acre_ft =  Area_acres*bed_data_sort.THICKNESS.sum()

    for permeability_index1 in range(len(permeability_array)):
        Front_Location = (Mobility_Ratio -
np.sqrt(Mobility_Ratio**2+List_of_permeability_ratio_DataTable[permeability_index1]*(1-Mobility_Ratio**2)))/
(Mobility_Ratio-1)
        Front_Location_list.append(Front_Location)
    #This code generates table of flood front location as the layers breakthrough
    Front_Location_list_DataTable = pd.DataFrame(Front_Location_list).transpose()

#=================================================================================================
=====================================
    # CALCULATING THE OIL FLOW RATE IN EACH BED AS EACH BED BREAKS THROUGH
    Water_Flowrate_per_bed = (width_of_bed_ft*bed_data_sort['THICKNESS']*Inj_Pressure_differential/
Length_of_bed_ft)*Water_Mobility

    Water_Flowrate_per_bed_table = pd.DataFrame(Water_Flowrate_per_bed).rename(columns={'THICKNESS':'Water
Flowrate Per Bed (Barrels/D)'})

#=================================================================================================
=======================================
    Water_Flowrate_list = []
    for n in range(len(permeability_array)):
        Water_Flowrate = Water_Flowrate_per_bed_table.iloc[0:n].sum()
        Water_Flowrate_list.append(Water_Flowrate)
    Water_Flowrate_table = pd.DataFrame(Water_Flowrate_list).rename(columns={'Water Flowrate Per Bed
(Barrels/D)':'Water Production Rate (Barrels/D)'})
```

```
#=====================================================================================
=====================================
   # CALCULATING THE OIL FLOW RATE IN EACH BED AS EACH BED BREAKS THROUGH
   Oil_Flowrate_per_bed_list = []
   for bed in Front_Location_list_DataTable.columns:
      Oil_Flowrate_per_bed = (width_of_bed_ft*bed_data_sort['THICKNESS']*Inj_Pressure_differential/
Length_of_bed_ft)*Water_Mobility/((1-Mobility_Ratio)*Front_Location_list_DataTable[bed]+Mobility_Ratio)
      Oil_Flowrate_per_bed_list.append(Oil_Flowrate_per_bed)
   Oil_Flowrate_per_bed_table = pd.DataFrame(Oil_Flowrate_per_bed_list).transpose()

#=====================================================================================
=====================================
   Oil_Flowrate_list = []
   for n in range(len(permeability_array)):
      Oil_Flowrate = Oil_Flowrate_per_bed_table[n].sum()
      Oil_Flowrate_list.append(Oil_Flowrate)
   Oil_Flowrate_table = pd.DataFrame(Oil_Flowrate_list).rename(columns={0:'Oil Production Rate'})

#=====================================================================================
=====================================
   # CALCULATING THE VERTICAL COVERAGE
   coverage_list = []
   Total_Number_of_layers = len(permeability_array)-1
   for number_layer_breakthrough in range(len(permeability_array)):
      coverage_individual = (number_layer_breakthrough+((Total_Number_of_layers-
number_layer_breakthrough)*Mobility_Ratio/(Mobility_Ratio-1))-(1/(Mobility_Ratio-
1))*np.sqrt(Mobility_Ratio**2+List_of_permeability_ratio_DataTable[number_layer_breakthrough][1:]*(1-
Mobility_Ratio**2)).sum())/Total_Number_of_layers
      coverage_list.append(coverage_individual)
   #Table of vertical coverage of the reservoir when a given layer just broke through.
   coverage_table = pd.DataFrame(coverage_list, columns=['Vertical Coverage (Fraction)'])

#=====================================================================================
=====================================
   WOR_denominator_ratio_list = []
   for denominator_index in range(len(permeability_array)):
      WOR_denominator_ratio =
permeability_array[denominator_index]/np.sqrt(Mobility_Ratio**2+List_of_permeability_ratio_DataTable[denominator_
index]*(1-Mobility_Ratio**2))
      WOR_denominator_ratio_list.append(WOR_denominator_ratio)
   WOR_denominator_ratio_table = pd.DataFrame(WOR_denominator_ratio_list)
      #WOR_denominator_ratio_table
   WOR_list = []
   for n in range(len(permeability_array)):
      # CALCULATING THE WATER OIL RATIO, WORn and generate table
      sum_of_permeability = bed_data_sort.PERMEABILITY.iloc[0:n].sum()
      #for number_layer_breakthrough in range(len(permeability_array)):
      WOR = sum_of_permeability/(WOR_denominator_ratio_table[n].sum())
      WOR_list.append(WOR)
   WOR_table = pd.DataFrame(WOR_list).rename(columns={0:'Water-Oil Ratio'})#,columns=['WATER-OIL RATIO'])

#=====================================================================================
=====================================
   #CALCULATING THE CUMULATIVE OIL RECOVERY AS EACH BED BREAKSTHROUGH.
   Cumulative_oil_recovery =
(7758*Areal_sweep_efficiency_at_breakthrough*Gross_rock_volume_acre_ft*Average_porosity*(SOI-
SOR)*coverage_table/OFVF).rename(columns={'Vertical Coverage (Fraction)':'Cumulative Oil Recovery (Barrels)'})

#=====================================================================================
=====================================
```

```
    #CALCULATING THE VOLUME OF WATER REQUIRED TO FILL-UP THE GAS SPACE.
    Water_volume_to_fillup_gas_space =
7758*Area_acres*bed_data_sort.THICKNESS*bed_data_sort.POROSITY*(SGI-Residual_gas_saturation)
    Water_volume_to_fillup_gas_space_table=pd.DataFrame(Water_volume_to_fillup_gas_space, columns = ['Water
Volume For Gas Space Fill-Up'])

#=================================================================================================
=====================================
    #CALCULATING THE PRODUCING WATER-OIL RATIO
    Producing_water_oil_ratio = (WOR_table*OFVF).rename(columns={'Water-Oil Ratio':'Producing Water-Oil Ratio'})
    Producing_water_oil_ratio

#=================================================================================================
=====================================
    # Note that the integration for the calculation of the cumulative oil produced starts from 0
    # Hence, a new row will have to be inserted at the first row with element 0
    # this is done for both the producing water-oil ratio and the cumulative oil produced.

    # for the cumulative oil recovery
    Cumulative_oil_recovery.loc[-1] = [0]  # adding a row
    #Cumulative_oil_recovery.index = Cumulative_oil_recovery.index + 1  # shifting index
    Cumulative_oil_recovery_Starting_from_0 = Cumulative_oil_recovery.sort_index()  # sorting by index

    # for the producing water-oil ratio
    Producing_water_oil_ratio.loc[-1] = [0]  # adding a row
    #Producing_water_oil_ratio.index = Producing_water_oil_ratio.index + 1  # shifting index
    Producing_water_oil_ratio_Starting_from_0 = Producing_water_oil_ratio.sort_index()  # sorting by index

    # CALCULATING THE CUMULATIVE WATER PRODUCTION
    # To determine the cumulative water production, the produced water oil ratio is ingreated against the cumulative oil
recovery.
    # The integration uses a cumulative trapezoidal row by row integration.
    # import numpy and scipy.integrate.cumtrapz
    import numpy as np
    from scipy import integrate
    # Preparing the Integration variables y, x.
    # the to.numpy() method converts from dataframe to numpy array which appears in the form of list of lists in the array.
    #The concatenate function helps to bring the list of lists together.
    x = np.concatenate(Cumulative_oil_recovery_Starting_from_0.to_numpy(),axis=0)
    y = np.concatenate(Producing_water_oil_ratio_Starting_from_0.to_numpy(),axis=0)
    # using scipy.integrate.cumtrapz() method
    Cumulative_water_produced = pd.DataFrame(integrate.cumtrapz(y, x), columns = ['Cumulative Water Produced'])

#=================================================================================================
=========================================
    # CALCULATING THE CUMULATIVE WATER INJECTED, Wi
    Cumulative_water_injected = (Cumulative_water_produced['Cumulative Water Produced'] +
OFVF*Cumulative_oil_recovery['Cumulative Oil Recovery (Barrels)'] +
Water_volume_to_fillup_gas_space_table['Water Volume For Gas Space Fill-Up']).drop([-1])
    Cumulative_water_injected_table = pd.DataFrame(Cumulative_water_injected,columns = ['Cumulative Water Injected
(Barrels)'])

#=================================================================================================
=========================================
    # CALCULATING THE TIME REQUIRED FOR INJECTION TO REACH A GIVEN RECOVERY.
    Time_days = Cumulative_water_injected_table['Cumulative Water Injected (Barrels)']/Constant_injection_rate
    Time_days_table = pd.DataFrame(Time_days).rename(columns ={'Cumulative Water Injected (Barrels)': 'Time
(Days)'}, inplace = False)
    #print(Time_days_table)
    Time_years = Time_days_table/365
    Time_years_table = Time_years.rename(columns ={'Time (Days)': 'Time (Years)'}, inplace = False)
```

```
#=============================================================================================
=============================================
   # TABLE OF ALL OBTAINED VALUES.
   All_tables =pd.concat([Layer_table,Oil_Mobility_table,Water_Mobility_table,
                     Water_Flowrate_per_bed_table, coverage_table, WOR_table,
                     Cumulative_oil_recovery, Water_volume_to_fillup_gas_space_table,
                     Producing_water_oil_ratio, Cumulative_water_produced,
                     Cumulative_water_injected_table, Time_days_table,
                     Time_years_table,Water_Flowrate_table,Oil_Flowrate_table,
                     Oil_Flowrate_per_bed_table.rename(columns=lambda x: str(x)+'Oil_Flowrate'),
                     Front_Location_list_DataTable.rename(columns=lambda x: str(x)+'Front_Location')
                     ], axis = 1).drop([-1])


   def results_and_graph_gui():
      global All_tables
      class MyTable(Table):

         def __init__(self, parent=None, **kwargs):
            Table.__init__(self, parent, **kwargs)
            return

      class MyApp1(Frame):

         def __init__(self, parent=None):
            self.parent = parent
            Frame.__init__(self)
            #self.main = self.master
            self.D_window = self.master
            #self.main.geometry('800x600+200+100')
            self.D_window.title('Dykstra-Parson General data and graph')
            f = Frame(self.D_window)
            f.pack(side = LEFT, expand=1)
            pt = make_table(f)
            bp = Frame(self.D_window)
            bp.pack(side=TOP)

            return

      def make_table(table_frame2, **kwds):

         df = All_tables
         pt = MyTable(table_frame2, dataframe=df, **kwds )
         pt.show()
         return pt
      def test1():
         """just make a table"""

         t = Toplevel()
         fr = Frame(t)
         fr.pack(fill=BOTH, expand=1)
         pt = make_table(fr)
         return

      def select_test():
         """cell selection and coloring"""

         #t = Toplevel()
         #fr = Frame(t)
         fr = table_frame2
         #fr.pack(fill=BOTH, expand=1)
```

```
            pt = Table(fr)
            pt.show()
            pt.All_tables
            pt.resetIndex(ask=False)
            pt.columncolors = {'c':'#337ab7'}
            df = pt.model.df

            mask_1 = df.a<7
            pt.setColorByMask('a', mask_1, '#337ab7')
            colors = {'red':'#f34130','blue':'blue'}
            for l in df.label.unique():
                mask = df['label']==l
                pt.setColorByMask('label', mask, l)
            pt.redraw()
            return

        def multiple_tables():
            """make many tables in one frame"""

            t = Toplevel(height=800)
            r=0;c=0
            for i in range(6):
                fr = Frame(t)
                fr.grid(row=r,column=c)
                pt = make_table(fr, showtoolbar=True, showstatusbar=True)
                c+=1
                if c>2:
                    c=0
                    r+=1
            return
        app1 = MyApp1()
        app1.mainloop()

#=============================================================================================
====================================
    table_frame2 = Frame(D_window)
    table_frame2.place(relheight = 1, relwidth =1)
    Button(table_frame2, text = 'View data',bg = '#337ab7',fg = 'white',justify = LEFT,relief=
RAISED,cursor='hand2',command = results_and_graph_gui).grid(row=2,column =2,padx=5,pady=5,stick = W)


#=============================================================================================
========================================
    D_window.geometry("200x150")
    D_window.mainloop()


menubar = Menu(root)
#Create a load menu
loadmenu = Menu(menubar, tearoff=0)
loadmenu.add_command(label="Load Data",command =  Load_File)
menubar.add_cascade(label="Load Data", menu=loadmenu)

#newmenu = Menu(menubar, tearoff=0)
#newmenu.add_command(label="New",command = new())
#menubar.add_cascade(label="New", menu=newmenu)

Fractional_flowmenu = Menu(menubar, tearoff=0)
Fractional_flowmenu.add_command(label="Data and Plot",command =  fractional_flow)
menubar.add_cascade(label="Fractional Flow", menu=Fractional_flowmenu)

printmenu = Menu(menubar, tearoff=0)
```

```
printmenu.add_command(label="Print Result",command = (lambda: Resultfile(entries)))
menubar.add_cascade(label="Print Result", menu=printmenu)

# create the Output menu
#output = Menu(menu)
outputmenu = Menu(menubar, tearoff=0)
outputmenu.add_command(label="Tabular results and graphs",command =(lambda: D_tables(entries)))
#outputmenu.add_command(label="Fractional Flow", command = fractional_flow)
#added "file" to our menu
menubar.add_cascade(label="Dykstra-Parson", menu=outputmenu)

reznikmenu = Menu(menubar, tearoff=0)
reznikmenu.add_command(label="Reznik et al continuous solution", command=(lambda: Reznik(entries)))
#helpmenu.add_command(label="About...", command=donothing)
menubar.add_cascade(label="Reznik et al", menu=reznikmenu)

root.config(menu=menubar)
#*************************************************************************************************
**********************
def Resultfile(entries):
   Number_of_points = float(entries['Number_of_points'].get())
   Length_of_bed_ft = float(entries['Length_of_bed_ft'].get())
   width_of_bed_ft = float(entries['width_of_bed_ft'].get())
   SWI = float(entries['SWI'].get())
   SOR = float(entries['SOR'].get())
   SOI = float(entries['SOI'].get())
   OFVF = float(entries['OFVF'].get())
   WFVF = float(entries['WFVF'].get())
   SGI = float(entries['SGI'].get())
   VISW = float(entries['VISW'].get())
   VISO = float(entries['VISO'].get())
   Saturation_gradient = float(entries['Saturation_gradient'].get())
   Constant_injection_rate = float(entries['Constant_injection_rate'].get())
   Inj_Pressure_differential = float(entries['Inj_Pressure_differential'].get())
   RGSU = float(entries['Residual_gas_saturation_unswept_area'].get())
   RGSS =  float(entries['Residual_gas_saturation_swept_area'].get())
   RGS = RGSU+RGSS

   KRW_1_SOR = np.interp(1-SOR, SW, KRW)
   KRO_SWI = np.interp(SWI, SW, KRO)
   Oil_Mobility =  permeability_array*KRO_SWI/VISO

   Water_Mobility =  permeability_array*KRW_1_SOR/VISW

   Mobility_Ratio = KRW_1_SOR*VISO/(KRO_SWI*VISW)
   Areal_sweep_efficiency_at_breakthrough =
0.54602036+(0.03170817/Mobility_Ratio)+(0.30222997/math.exp(Mobility_Ratio)-0.0050969*Mobility_Ratio)
   Displacement_efficiency = (1-SWI-SGI-SOR)/(1-SWI-SGI)
   import numpy as np
   Areal_sweep_efficiency =  Areal_sweep_efficiency_at_breakthrough+0.2749*np.log((1/Displacement_efficiency))

   #Average_porosity = np.mean(bed_data_sort.POROSITY)
   Average_porosity = np.mean(bed_data_sort.POROSITY)
   Area_acres = Length_of_bed_ft*width_of_bed_ft/43560
   Gross_rock_volume_acre_ft =  Area_acres*bed_data_sort.THICKNESS.sum()

   SW_table = pd.DataFrame(SW, columns = ['SW'])
   b = (np.log((KRO/KRW)[2])-np.log((KRO/KRW)[3]))/(SW[3]-SW[2])
   a = (KRO/KRW)[2]*math.exp(b*SW[2])
   def fw(SW):
      fw = 1/(1+a*(VISW/VISO)*np.exp(-b*SW))
      return(fw)
```

```
    xList = []
    for i in range(0, 10000):
        x = random.uniform(SWI+0.1, 1)
        xList.append(x)
    xs = np.array(xList)
    m = 1/((xs-SWI)*(1+(VISW/VISO)*a*np.exp(-b*xs)))
    tangent_slope=max(m)
    Saturation_at_Breakthrough = SWI + 1/tangent_slope
    def funct(SWF):
        swf = SWF[0]
        F = np.empty((1))
        F[0] = ((tangent_slope*(swf-SWI)*(1+(VISW/VISO)*a*math.exp(-b*swf)))-1)
        return F
    SWF_Guess = np.array([SWI+0.1])
    SWF = fsolve(funct, SWF_Guess)[0]
    Fwf = fw(SWF)
    Fw = fw(SW)
    Fw_table = pd.DataFrame(Fw, columns = ['Fractional Flow(Fw)'])
    dfw_dSw = (VISW/VISO)*a*b*np.exp(-SW*b)/(1+(VISW/VISO)*a*np.exp(-SW*b))**2
    dfw_dSw_table = pd.DataFrame(dfw_dSw, columns = ['dFw/dSw'])

    tangent = (SW-SWI)*tangent_slope
    tangent_table = pd.DataFrame(tangent, columns = ['Tangent'])
    Fractional_flow_table = pd.concat([SW_table, Fw_table, dfw_dSw_table, tangent_table], axis=1)

    f = open('Result.txt', 'w')
    f.write('RUN DATE AND TIME: '+ str(date_time)+'\n')
    f.write('\n')
    f.write('INPUTS \n')
    f.write('Number of points: '+ str(Number_of_points)+'\n')
    f.write('Lenght of bed in feet: '+ str(Length_of_bed_ft)+'\n')
    f.write('Width of bed in feet: '+ str(width_of_bed_ft)+'\n')
    f.write('Average porosity: '+ str(average_porosity)+'\n')
    f.write('Viscosity of oil in centipoise: '+ str(VISO)+'\n')
    f.write('Viscosity of water in centipoise: '+ str(VISW)+'\n')
    f.write('Formation volume factor of oil: '+ str(OFVF)+'\n')
    f.write('Formation volume factor of water: '+ str(WFVF)+'\n')
    f.write('Initial water saturation: '+ str(SWI)+'\n')
    f.write('Initial gas saturation: '+ str(SGI)+'\n')
    f.write('Initial oil saturation: '+ str(SOI)+'\n')
    f.write('Residual oil saturation: '+ str(SOR)+'\n')
    f.write('Constant injection rate: '+ str(Constant_injection_rate)+'\n')
    f.write('Injection pressure: '+ str(Inj_Pressure_differential)+'\n')
    f.write('Residual gas saturation of unswept area: '+ str(RGSU)+'\n')
    f.write('Residual gas saturation of swept area: '+ str(RGSS)+'\n')
    f.write('Residual gas saturation: '+ str(RGS)+'\n')
    f.write('\n')

f.write('**********************************************************************************************
**********\n ')
    f.write('\n')
    f.write('GENERAL OUTPUTS \n')
    f.write('Average porosity: '+ str(Average_porosity)+'\n')
    f.write('Water relative permeability at 1-SOR: '+ str(KRW_1_SOR)+'\n')
    f.write('Oil relative permeability at initial water saturation: '+ str(KRO_SWI)+'\n')
    f.write('Mobility ratio: '+ str(Mobility_Ratio)+'\n')
    f.write('Areal sweep efficiency at breakthrough: '+ str(Areal_sweep_efficiency_at_breakthrough)+'\n')
    f.write('Area of reservoir bed in acres: '+ str(Area_acres)+'\n')
    f.write('Gross rock volume in acres-feet: '+ str(Gross_rock_volume_acre_ft)+'\n')
    f.write('Displacement efficiency: '+ str(Displacement_efficiency)+'\n')
    f.write('Areal sweep efficiency: '+ str(Areal_sweep_efficiency)+'\n')
    f.write('Saturation gradient: '+ str(Saturation_gradient)+'\n')
```

```
    f.write('\n')
    f.write('FRACTIONAL FLOW OUTPUTS \n')
    f.write('Correlation: Kro/Krw = aexp(-bSw) \n')
    f.write('b: '+ str(b)+'\n')
    f.write('a: '+ str(a)+'\n')
    f.write('Slope of the tangent line: '+ str(tangent_slope)+'\n')
    f.write('Flood Front Saturation(Swf): '+ str(SWF)+'\n')
    f.write('Flood Front Fractional Flow(Fwf): '+ str(Fwf)+'\n')
    f.write('Saturation at breakthrough(SwBT): '+ str(Saturation_at_Breakthrough)+'\n')
    f.write('\n')
    f.write(str(Fractional_flow_table)+'\n')
    f.write('\n')
    f.write('DYKSTRA-PARSONS OUTPUTS \n')
    f.write(str(All_tables)+'\n')
    f.write('\n')
    f.write('REZNIK ET AL OUTPUTS \n')
    f.write(str(General)+'\n')
    f.close()
    return f
#================================================================================
============================================================
root.geometry("900x500")
root.mainloop()
```

APPENDIX B

Simulation File

```
RUNSPEC

TITLE

----------COMPUTER IMPLEMENTATION OF DYKSTRA-PARSON WATERFLOOD CALCULATION----------

DIMENS

  20 20 12 /

-- The number of equilibration regions is inferred from the EQLDIMS

-- keyword.

EQLDIMS

/

-- The number of PVTW tables is inferred from the TABDIMS keyword;

-- when no data is included in the keyword the default values are used.

TABDIMS

/

OIL

--GAS

WATER

--DISGAS

FIELD

START

  1 'JAN' 2021 /

WELLDIMS

-- Item 1: maximum number of wells in the model

--          - there are two wells in the problem; injector and producer

-- Item 2: maximum number of grid blocks connected to any one well
```

```
--          - must be one as the wells are located at specific grid blocks
-- Item 3: maximum number of groups in the model
--          - we are dealing with only one 'group'
-- Item 4: maximum number of wells in any one group
--          - there must be two wells in a group as there are two wells in total
  2 12 12 2 /
UNIFOUT
GRID
-- The INIT keyword is used to request an .INIT file. The .INIT file
-- is written before the simulation actually starts, and contains grid
-- properties and saturation tables as inferred from the input
-- deck. There are no other keywords which can be used to configure
-- exactly what is written to the .INIT file.
INIT
NOECHO
DX
-- There are in total 4800 cells with length 144.8ft in x-direction
        4800*144.8 /
DY
-- There are in total 4800 cells with length 100ft in y-direction
        4800*100 /
DZ
-- The layers are 1ft thick, in each layer there are 400 cells
        400*1
        400*1
        400*1
        400*1
```

400*1

400*1

400*1

400*1

400*1

400*1

400*1

400*1 /

TOPS

-- The depth of the top of each grid block

400*4359.5 /

PORO

-- Constant porosity of 0.3 throughout all 300 grid cells

-- 300*0.3 /

400*0.301

400*0.288

400*0.283

400*0.309

400*0.311

400*0.305

400*0.322

400*0.306

400*0.322

400*0.297

400*0.323

400*0.286 /

PERMX

-- The layers have perm. 500mD, 50mD and 200mD, respectively.

-- 100*500 100*50 100*200 /

400*593

400*500

400*366

400*1464

400*2790

400*5940

400*9230

400*2860

400*3080

400*594

400*2370

400*526 /

PERMY

-- Equal to PERMX

-- 100*500 100*50 100*200 /

400*593

400*500

400*366

400*1464

400*2790

400*5940

400*9230

400*2860

400*3080

400*594

400*2370

400*526 /

PERMZ

--Equal to PERMX

-- Cannot find perm. in z-direction in Odeh's paper

-- For the time being, we will assume PERMZ equal to PERMX and PERMY:

400*500

400*366

400*1464

400*2790

400*5940

400*9230

400*2860

400*3080

400*594

400*2370

400*526 /

ECHO

PROPS

PVTW

-- Item 1: pressure reference (psia)

-- Item 2: water FVF (rb per bbl or rb per stb)

-- Item 3: water compressibility (psi^{-1})

-- Item 4: water viscosity (cp)

-- Item 5: water 'viscosibility' (psi^{-1})

-- Using values from Norne:

```
-- In METRIC units:
--        277.0 1.038 4.67E-5 0.318 0.0 /
-- In FIELD units:
        700 1.011 3.22E-6 0.95 0.0 /

ROCK
-- Item 1: reference pressure (psia)
-- Item 2: rock compressibility (psi^{-1})
-- Using values from table 1 in Odeh:
        14.7 3E-6 /
SWOF
-- Column 1: water saturation
--            - this has been set to (almost) equally spaced values from 0.12 to 1
-- Column 2: water relative permeability
--            - generated from the Corey-type approx. formula
--              the coeffisient is set to 10e-5, S_{orw}=0 and S_{wi}=0.12
-- Column 3: oil relative permeability when only oil and water are present
--            - we will use the same values as in column 3 in SGOF.
--              This is not really correct, but since only the first
--              two values are of importance, this does not really matter
-- Column 4: water-oil capillary pressure (psi)
0.2     0       1       0
0.25    0.003   0.68    0
0.3     0.008   0.46    0
0.35    0.018   0.32    0
0.4     0.035   0.2     0
0.45    0.054   0.124   0
```

```
0.5     0.08    0.071   0
0.55    0.105   0.038   0
0.6     0.14    0.017   0
0.65    0.18    0       0 /
--SGOF
-- Column 1: gas saturation
-- Column 2: gas relative permeability
-- Column 3: oil relative permeability when oil, gas and connate water are present
-- Column 4: oil-gas capillary pressure (psi)
--              - stated to be zero in Odeh's paper
-- Values in column 1-3 are taken from table 3 in Odeh's paper:
--0      0       1       0
--0.0010       1       0
--0.02  0       0.997   0
--0.05  0.005   0.980   0
--0.12  0.025   0.700   0
--0.2   0.075   0.350   0
--0.25  0.125   0.200   0
--0.3   0.190   0.090   0
--0.4   0.410   0.021   0
--0.45  0.60    0.010   0
--0.5   0.72    0.001   0
--0.6   0.87    0.0001  0
--0.7   0.94    0.000   0
--0.85  0.98    0.000   0
--0.88  0.984   0.000   0 /
--1.00  1.0     0.000   0 /
```

```
-- Warning from Eclipse: first sat. value in SWOF + last sat. value in SGOF
--                       must not be greater than 1, but Eclipse still runs
-- Flow needs the sum to be excactly 1 so I added a row with gas sat. =  0.88
-- The corresponding krg value was estimated by assuming linear rel. between
-- gas sat. and krw. between gas sat. 0.85 and 1.00 (the last two values given)
DENSITY
-- Density (lb per ft3) at surface cond. of
-- oil, water and gas, respectively (in that order)
-- Using values from Norne:
-- In METRIC units:
--      859.5 1033.0 0.854 /
-- In FIELD units:
         53.66 64.49 0.0533 /
--PVDG
-- Column 1: gas phase pressure (psia)
-- Column 2: gas formation volume factor (rb per Mscf)
--             - in Odeh's paper the units are said to be given in rb per bbl,
--               but this is assumed to be a mistake: FVF-values in Odeh's paper
--               are given in rb per scf, not rb per bbl. This will be in
--               agreement with conventions
-- Column 3: gas viscosity (cP)
-- Using values from lower right table in Odeh's table 2:
--14.700        166.666        0.008000
--264.70        12.0930        0.009600
--514.70        6.27400        0.011200
--1014.7        3.19700        0.014000
--2014.7        1.61400        0.018900
```

```
--2514.7        1.29400         0.020800
--3014.7        1.08000         0.022800
--4014.7        0.81100         0.026800
--5014.7        0.64900         0.030900
--9014.7        0.38600         0.047000 /
PVTO
-- Column 1: dissolved gas-oil ratio (Mscf per stb)
-- Column 2: bubble point pressure (psia)
-- Column 3: oil FVF for saturated oil (rb per stb)
-- Column 4: oil viscosity for saturated oil (cP)
-- Use values from top left table in Odeh's table 2:
0.0010 14.7     1.0620 1.0400 /
0.0905 264.7    1.1500 0.9750 /
0.1800 514.7    1.2070 0.9100 /
0.3710 1014.7 1.2950 0.8300 /
0.6360 2014.7 1.4350 0.6950 /
0.7750 2514.7 1.5000 0.6410 /
0.9300 3014.7 1.5650 0.5940 /
1.2700 4014.7 1.6950 0.5100
          9014.7 1.5790 0.7400 /
1.6180 5014.7 1.8270 0.4490
          9014.7 1.7370 0.6310 /
-- It is required to enter data for undersaturated oil for the highest GOR
-- (i.e. the last row) in the PVTO table.
-- In order to fulfill this requirement, values for oil FVF and viscosity
-- at 9014.7psia and GOR=1.618 for undersaturated oil have been approximated:
-- It has been assumed that there is a linear relation between the GOR
```

```
-- and the FVF when keeping the pressure constant at 9014.7psia.
-- From Odeh we know that (at 9014.7psia) the FVF is 2.357 at GOR=2.984
-- for saturated oil and that the FVF is 1.579 at GOR=1.27 for undersaturated oil,
-- so it is possible to use the assumption described above.
-- An equivalent approximation for the viscosity has been used.
/
SOLUTION
-- ------------------------------------------------------------------------
EQUIL
-- Item 1: datum depth (ft)
-- Item 2: pressure at datum depth (psia)
--           - Odeh's table 1 says that initial reservoir pressure is
--             4800 psi at 8400ft, which explains choice of item 1 and 2
-- Item 3: depth of water-oil contact (ft)
--           - chosen to be directly under the reservoir
-- Item 4: oil-water capillary pressure at the water oil contact (psi)
--           - given to be 0 in Odeh's paper
-- Item 5: depth of gas-oil contact (ft)
--           - chosen to be directly above the reservoir
-- Item 6: gas-oil capillary pressure at gas-oil contact (psi)
-- Item 7: RSVD-table
-- Item 8: RVVD-table
-- Item 9: Set to 0 as this is the only value supported by OPM
-- Item #: 1 2    3    4 5    6 7 8 9
          4365.5 700 4381.5 0 4300 0 0 0 0 /
--RSVD
-- Dissolved GOR is initially constant with depth through the reservoir.
```

```
-- The reason is that the initial reservoir pressure given is higher
---than the bubble point presssure of 4014.7psia, meaning that there is no
-- free gas initially present.
--8300 1.270
--8450 1.270 /
SUMMARY
-- ------------------------------------------------------------------------
-- 1a) Oil rate vs time
FOPR
-- Field Oil Production Rate
FWCT
-- Field water cut
-- 1b) GOR vs time
--WWOR
-- Well Gas-Oil Ratio
--   'PROD'
--/
WWCT
-- Well Water Cut
   'INJ'
  'PROD'
/
--WCT
-- Water Cut
--   'INJ'
--  'PROD'
--/
```

```
WWPT
-- Well Water Production Total
  'PROD'
/
-- Using FGOR instead of WGOR:PROD results in the same graph
--FWOR
-- 2a) Pressures of the cell where the injector and producer are located
BPR
1       1       1 /
1       1       2 /
1       1       3 /
1       1       4 /
1       1       5 /
1       1       6 /
1       1       7 /
1       1       8 /
1       1       9 /
1       1       10 /
1       1       11 /
1       1       12 /
20      20      1 /
20      20      2 /
20      20      3 /
20      20      4 /
20      20      5 /
20      20      6 /
20      20      7 /
```

20 20 8 /

20 20 9 /

20 20 10 /

20 20 11 /

20 20 12 /

/

-- 2b) Oil saturation at grid points given in Odeh's paper

BOSAT

1 1 1 /

1 1 2 /

1 1 3 /

1 1 4 /

1 1 5 /

1 1 6 /

1 1 7 /

1 1 8 /

1 1 9 /

1 1 10 /

1 1 11 /

1 1 12 /

20 20 1 /

20 20 2 /

20 20 3 /

20 20 4 /

20 20 5 /

20 20 6 /

20 20 7 /

20 20 8 /

20 20 9 /

20 20 10 /

20 20 11 /

20 20 12 /

/

-- 2c) Water saturation at grid points given in Odeh's paper

BWSAT

1 1 1 /

1 1 2 /

1 1 3 /

1 1 4 /

1 1 5 /

1 1 6 /

1 1 7 /

1 1 8 /

1 1 9 /

1 1 10 /

1 1 11 /

1 1 12 /

20 20 1 /

20 20 2 /

20 20 3 /

20 20 4 /

20 20 5 /

20 20 6 /

20 20 7 /

```
20       20       8 /
20       20       9 /
20       20       10 /
20       20       11 /
20       20       12 /
/
-- In order to compare Eclipse with Flow:
WBHP
  'INJ'
  'PROD'
/
--WGIR
--  'INJ'
--  'PROD'
--/
--WGIT
--  'INJ'
--  'PROD'
--/
--WGPR
--  'INJ'
--  'PROD'
--/
--WGPT
--  'INJ'
--  'PROD'
/
```

WOIR
'INJ'
'PROD'
/
WOIT
'INJ'
'PROD'
/
WOPR
'INJ'
'PROD'
/
WOPT
'INJ'
'PROD'
/
WWIR
'INJ'
'PROD'
/
WWIT
'INJ'
'PROD'
/
WWPR
'INJ'
'PROD'

```
/
WWPT
  'INJ'
  'PROD'
/
--WIR
--  'INJ'
--  'PROD'
--/
--WIT
--  'INJ'
--  'PROD'
--/
--WPR
--  'INJ'
--  'PROD'
--/
--WPT
--  'INJ'
--  'PROD'
--/
--OPR
--  'INJ'
--  'PROD'
--/
--OPT
--  'INJ'
```

```
--  'PROD'
--/
SCHEDULE
-- ------------------------------------------------------------------------
RPTSCHED
        'PRES' 'SGAS' 'RS' 'WELLS' /

RPTRST
        'BASIC=1' /
-- If no resolution (i.e. case 1), the two following lines must be added:
DRSDT
 0 /
-- if DRSDT is set to 0, GOR cannot rise and free gas does not
-- dissolve in undersaturated oil -> constant bubble point pressure
WELSPECS
-- Item #: 1      2       3       4       5        6
        'PROD'          'G1'    20      20      4365.5 'OIL' /
        'INJ'   'G1'    1       1       4365.5 'WATER' /
/
-- Coordinates in item 3-4 are retrieved from Odeh's figure 1 and 2
-- Note that the depth at the midpoint of the well grid blocks
-- has been used as reference depth for bottom hole pressure in item 5
COMPDAT
-- Item #: 1    2       3       4       5       6       7       8       9
        'PROD'          20      20      1       12      'OPEN'1*      1*      0.5 /
        'INJ'   1       1       1       12      'OPEN'1*      1*      0.5 /
/
```

```
-- Coordinates in item 2-5 are retreived from Odeh's figure 1 and 2
-- Item 9 is the well bore internal diameter,
-- the radius is given to be 0.25ft in Odeh's paper
WCONPROD
-- Item #:1      2     3    4         5  9
--         'PROD' 'OPEN' 'ORAT' 20000 4* 1000 /
          'PROD' 'OPEN' 'ORAT' 20000 4* 100 /
/
-- It is stated in Odeh's paper that the maximum oil prod. rate
-- is 20 00000stb per day which explains the choice of value in item 4.
-- The items > 4 are defaulted with the exception of item  9,
-- the BHP lower limit, which is given to be 1000psia in Odeh's paper
WCONINJE
-- Item #:1       2         3         4        5     6  7
          'INJ'    'WATER'       'OPEN''RATE'1800 1* 9014 /
/
-- Stated in Odeh that gas inj. rate (item 5) is 100MMscf per day
-- BHP upper limit (item 7) should not be exceeding the highest
-- pressure in the PVT table=9014.7psia (default is 100 000psia)
TSTEP
--Advance the simulater once a month for TEN years:
31 28 31 30 31 30 31 31 30 31 30 31
31 28 31 30 31 30 31 31 30 31 30 31
31 28 31 30 31 30 31 31 30 31 30 31
31 28 31 30 31 30 31 31 30 31 30 31
31 28 31 30 31 30 31 31 30 31 30 31
31 28 31 30 31 30 31 31 30 31 30 31
```

31 28 31 30 31 30 31 31 30 31 30 31

31 28 31 30 31 30 31 31 30 31 30 31

31 28 31 30 31 30 31 31 30 31 30 31

31 28 31 30 31 30 31 31 30 31 30 31 /

--Advance the simulator once a year for TEN years:

--10*365 /

END

REFERENCES

B. C. Craft, M. H. (1991). *Applied Petroleum Reservoir Engineering.pdf*.

Buckley, S. E. and Leverett, M. C. (1942) Mechanism of Fluid Displacement in Sands. Trans., AIME 146,107-16.

C.H Wu. (1988). Waterflood Performance Projection Using Classical Waterflood Models Compared with Numerical Model Performance jpt forum.

D. Tiab, (1986). SPE 15020 Extension of the Dykstra-Parsons Method to Layered-Composite Reservoirs.

Dykstra, H. and Parsons, H.: "The Prediction of Oil Recovery by Waterflooding, " Secondary Recovery of Oil in the United States, API, New York City (1950) 160-74.

Enick, R. M., Pittsburgh, U., Reznik, A. A., Pittsburgh, U., Miller, R. A., & Pittsburgh, U. (1988). The Statistical and Dimensionless- Time Analog to the Generalized Dykstra-Parsons Method. February 313–319.

Farrell, D. (2019). *pandastable Documentation*.

Field, P., Saavedra, N. F., Peralta, R. C., Ltda, D. T. H., Cobb, W., & Cobb, W. M. (2003). SPE 81042 Distribution of Injected Water by Using CGM Method: A Case History in Palogrande-Cebu Field.

Gasimov, R. R. (2005), Modification of the Dykstra-Parsons Method to Incorporate Buckley-Leverett displacement Theory for Water Floods. Thesis Submitted to the Office of Graduate Studies of Texas A & M University in partial fulfilment of the requirements for the degree of MASTER OF SCIENCE August 2005 Major Subject: Petroleum Engineering.

Gomez, V., Gomez, A., & Duran, J. (2009). SPE 121854 Analytical Simulation of the Injection / Production System of La Cira East and North Areas Using CGM Method.

Jensen, J. L., & Currie, D. (1990). A New Method for Estimating the Dykstra · Parsons Coefficient to Characterize Reservoir Heterogeneity. August, 369–374.

Johnson, C. E. (1956). Prediction of Oil Recovery by Water Flood - A Simplified Graphical Treatment of the Dykstra-Parsons Method. 2–3.

Lewis, E., Dao, E. K., & Mohanty, K. K. (2008). Sweep Efficiency of Miscible Floods in a High-Pressure Quarter-Five-Spot Model. May, 24–27.

Li, D. (2004). SPE 88459 Comparative Simulation Study of Water Flood. 1–10.

Mahfoudhl, J. E., Enick, R. M., & Pittsburgh, U. (1990). Extension of the Generalized Dykstra- Parsons Technique to Polymer Flooding in Stratified Porous Media. August, 339–345.

Morrison, G. R. (n.d.). SPE 97645 Dykstra Parsons Water Flood Theory Adapted to Chemical Flood Modelling.

Nyga, J. I. (2020). Simulation of Immiscible Water-Alternating-Gas Injection in a Stratified Reservoir: Performance Characterization Using a New Dimensionless Number. November 2019, 1711–1728.

Omar, A. I., Chen, Z., & Khalifa, A. E. (2017). Predicating Water-flooding Performance into Stratified Reservoirs Using a data driven proxy model. 8(7), 60–78. https://doi.org/10.5897/JPGE2016.0240

Popa, A. S., Sivakumar, K., & Cassidy, S. (2012). SPE 154302 Associative Data Modeling and Ant Colony Optimization Approach for Waterflood Analysis. 1–14.

Reznik, A. A., Robert M. Enick, & Sudhir B. Panvelker. (1984). An Analytical Extension of the Dykstra -Parsons Vertical Stratification Discrete Solution to a Continuous, Real-Time Basis.

Richardson, J. G. (1957). The Calculation of Waterflood Recovery from Steady-State Relative Permeability Data. 64–66.

Salem Mobarak (1975). Waterflooding Performance Using Dykstra-Parsons as
Singh, S. P., & Kiel, O. G. (1982). Waterflood Design (Pattern, Rate, and Timing).

Stiles, W.E. (1949). Use of Permeability Distribution in Water-flood Calculations. Trans., AIME 186,9-13.

Warren, J.E., (1964), and Cosgrove, J.J.: Prediction of Waterflood Behavior in a Stratified System. Soc. Pet. Eng. · J., 149-15 7.

Welge, H. J. (1952). A Simplified Method for Computing Oil Recoveries by Gas or Water Drive. Trans. AIME, 195, 91-98.

Zhao, L., Li, L., Wu, Z., & Zhang, C. (2016). Analytical Model of Waterflood Sweep Efficiency in Vertical Heterogeneous Reservoirs under Constant Pressure. 2016.