Project report

TPG4560 – Petroleum Engineering, Specialization Project

Student – Fadhil Berylian

Investigation of the Known Issues and Validation of OPM Flow Reservoir Simulator

Trondheim, 14 December 2019

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Geoscience and Petroleum

NTNU

Summary

As an open source reservoir simulator aimed as both professionals and researchers, OPM Flow should ensure that it is reliable and could deliver satisfactory results with as little error as possible. Unfortunately, in the past versions of Flow, some issues are found that impact simulation results. These include convergence issue for small-scale models and oil injection phenomenon in waterflooding scenario. With the release of new versions of Flow, this study investigates whether the issues have been fixed, and give recommendations if any issues are still found. Furthermore, simulation results from OPM Flow are compared against the widely-used ECLIPSE to see how the open source simulator holds up. A Python script that runs fully-implicit simulation is also used for validating results from Flow.

The OPM Flow release version used in this study is taken directly from OPM's Github repository, to ensure the version investigated is the most recently updated. Several cases are built for this study, including single-phase flow cases and two-phase oil/water flow cases. The cases have different gridblock size, down to centimeter scale, to see how far OPM Flow could still produce reliable results without any issue. For benchmarking purposes, the same set of cases are also run in ECLIPSE and results are compared. In addition to that, a Python code is created from scratch that gives an accurate simulation results using fully-implicit method to solve multiphase flow equations. It is hoped that the Python code would be able to function as a validation tool for developers of reservoir simulators.

After testing the cases on the newest Flow versions, it is found that the problems found before have been mostly fixed. Convergence issues are not found even for the smallest scale case, and no more oil injection when the same waterflooding scenario is re-run in the new Flow release. Results from Flow are also similar to ECLIPSE and the Python script when a same case is tested in all three simulators. However, further improvements are still needed for OPM Flow as some problems are found during the course of this study. When well BHP constraint is the same as reservoir initial pressure, the well would shut down at simulation start. This could be currently mitigated by parsing WTEST keyword, but the issue should be fixed by better defining the BHP constraint. Numerical issues also tend to happen during the beginning of simulation, and could be fixed by implementing model equilibriation technique used in ECLIPSE. Finally, it is found that saturation curve in results from Flow oscillates and less smooth compared to Python. Improving interpolation and trendline model in Flow input tables could help fix this issue.

Table of Contents

List of Tables

List of Figures

Abbreviations

p	=	pressure
S	=	fluid saturation
ϕ	=	porosity
k	=	permeability
l (subscript)	=	fluid phase (can be o for oil, w for water, or g for gas)
ρ	=	fluid density
R_{so}	=	solution-gas/oil ratio
B	=	formation volume factor (FVF)
μ	=	fluid viscosity
c	=	compressibility
P_c	=	capillary pressure
k_r	=	relative permeability
Φ	=	fluid potential
γ	=	fluid gravity
q	=	fluid flowrate
T	=	transmissibility
V_b	=	bulk volume
Δt	=	timestep size
Δx, Δy, Δz	=	gridblock size in x, y, and z direction
sc (subscript)	=	value at standard condition
ψ_i	=	connection with gridblock i
(v) (superscript)	=	iteration index
n (superscript)	=	timestep index

Chapter 1

Introduction

This chapter will discuss some background to this study, what are the problems that made this study necessary to conduct, and the objectives that this study aims to achieve.

OPM Flow is a fully-implicit reservoir simulator and one of the tools created by the Open Porous Media (OPM) initiative (Rasmussen et al., 2019). It is open-source, meaning anyone may have the right to study and change the source code. In the effort to continuously improve the end product, Flow encourages anyone to take part in its development by creating pull requests to propose necessary changes in the OPM GitHub repository (`https://github.com/OPM`). As a result of its active development, over the years Flow has matured to be a robust reservoir simulator with various industry-standard features. In its recent stable release version, Flow is capable of modeling EOR scenarios (with polymer or solvent), modified Black Oil model, end-point scaling and hysterisis, has a knowledge of several grid systems, various well control schemes, flexible output of restart files and summary files, and much more.

The first documented release of OPM Flow was the 2013.10 version released on October 2013 and ever since, a new version is released every six months. In version 2018.10, even after all the continuous and collaborative effort to improve the simulator, Flow is still far from perfect and some issues still could be encountered. This study is aimed to investigate if the issues has persisted in the most recent release of Flow, and whether any changes to its source code are necessary. Furthermore, in order to validate the value of Flow in the industry, this study also compares the results from Flow to other reservoir simulators, such as the widely-used Eclipse simulator built by Schlumberger, and also a fully-implicit simulator developed using Python 2.7 built during the course of this study.

1.1 Problem Description

One of the issues encountered in Flow 2018.10 is a convergence issue. The Flow software could get convergence problems when the physical model is scaled down. If this issue is

ignored, it could lead to some difficulties when working with the software, for example when dealing with models with pinch-out areas that are usually represented by small, irregular sized grid blocks. A simple example using a one-dimensional grid shows that for gridblock size of 2 meter, Flow 2018.10 fails to accurately predict pressure performance in the reservoir. Figure 1.1 shows the average field pressure (FPR) plot for such an example. The injection well in the left boundary was set to 200 bar and the production well in the right was set to 100 bar. Ideally, average pressure would supposedly go to 150 bar. However, here it never goes close to that, but instead stays near 100 bar. When investigating this particular subject, another interesting matter arises, in which wells would unexpectedly shut down in the beginning of simulation. The last issue that would require attention was discovered during the class project in the Reservoir Simulation class of NTNU on Spring 2019. It was found that when optimizing the well placement for a 5-spot water injection scheme, the results achieved was unreliable, as it is found that some wells injected oil into the reservoir to help keep bottom hole pressure (BHP) stable (Figure 1.2). A waterflooding project that involve multiple injectors and/or producers is a huge undertaking, and if well placement simulated by Flow is proved to be faulty, then huge amount of damage in costs would be inevitable. All of the issues discussed here are inspected carefully in this study, in the hope that in the newest versions OPM development team has taken some measures to fix them, and if not then this study could help by investigating the source code to any parts that need improvement and aid the OPM team with the development.

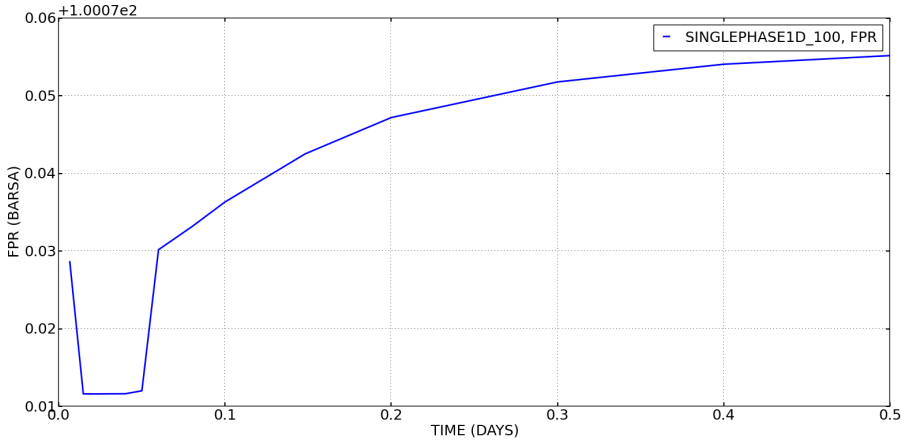


Figure 1.1: One-dimensional reservoir simulation result from Flow 2018.10. The average pressure in reservoir (FPR) ideally would approach 150 bar for this particular case, but it is clearly not what happen here.

The ECLIPSE reservoir simulator has long been an industry-reference simulator, and is widely used around the globe. Flow uses same keywords as Eclipse as input commands for the simulator. Cases (in the form of .DATA file) that can be run on Eclipse then can also be run on Flow with no need to change anything. Benchmarking results from OPM Flow with results from ECLIPSE would then be practical, and the results comparison could give

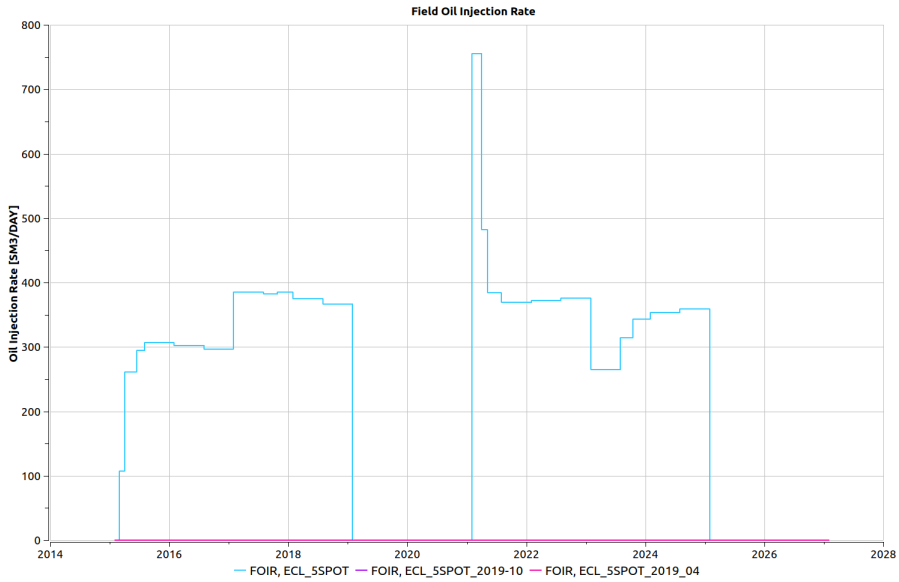


Figure 1.2: Oil injection phenomenon during a water injection scenario.

insights on how Flow might be improved.Another comparison that is conducted here is with a Python-based fully-implicit simulator. Since Flow source code is based on C++ language, it would be interesting to see how it would compare to a fully-implicit simulator built with Python. This Python simulator is built for the purpose of this study and also as an attempt to develop a validation tool for any reservoir simulation software. Python has been known for its practicality, and building a simple validation tool could help any reservoir simulator developers that want to test their results and perfect their software.

1.2 Objective

Based on the background and description of the problem, the objectives of this study can be formulated as:

1. Investigate the known issues in OPM Flow reservoir simulator and determine if recent releases have minimized or eliminated these issues.

2. Conduct comparisons of simulation cases results from Flow against ECLIPSE and fully-implicit Python script as an effort to validate OPM Flow as a robust simulation tool.

3. Present results and recommendations that could help with future developments of OPM Flow.

Chapter 2

Background and Basic Theory

This chapter presents underlying theories and principles used in this study, including literature reviews and how are they relevant to this study.

2.1 Reservoir Simulation

2.1.1 The need for reservoir simulation

Reservoir simulation is an act of utilizing computers to simulate fluid flow in a hydrocarbon reservoir model (Berg, 2019), by solving complex equations through the use of physical, mathematical, computer programming, and reservoir engineering knowledge (Ertekin et al., 2001). Reservoir simulation is necessary because petroleum engineers, especially reservoir engineers, have to obtain an accurate depiction of reservoir performance under various operating conditions. Since hydrocarbon recovery projects pose high risks in costs and safety, these risks have to be assessed thoroughly and minimized as far as possible. By taking into consideration the results from simulating a hydrocarbon reservoir, the engineers could get a proper picture of how the reservoir would undergone recovery, predicting if any unwanted occurrence could happen, and how to mitigate them. Most of the risks would exist simply because the reservoir has specific characteristic that could hinder the recovery process, for example heterogeneous and anisotropic rock properties could lead to greatly varying permeability values across the reservoir that might give recovery much less than what was initially predicted. One should consider using reservoir simulation to acknowledge risks before venturing further into the development phase of hydrocarbon recovery.

Advancements in computing technology lead to the widespread use of reservoir simulation tools in the petroleum industry. Only a few decades ago, only a few high-end computers would be able to run a computationally-heavy reservoir simulator. Today, nearly every personal computer could run a reservoir simulation tool with great speed and accuracy. At its core, reservoir simulator solves partial differential equations (PDE's) that would later

be developed into a set of algebraic mathematical equations. These equations should have appropriate boundary and initial conditions in order to be able to approximate the behavior of the reservoir. The flow of different fluid phases (oil, water, gas) and mass transfers are also represented by the same equations. Another important equation, Darcy's law, depicts effects of different acting forces in the reservoir (gravity, viscous, capillary).

Reservoir simulation is generally performed in the following steps, according to Ertekin et al. (2001):

1. **Set the objectives**. The objectives have to be clear, realistic, and compatible with available data. The objectives are used to set strategy, identify resources, and determine lessons to be learned.

2. **Acquire and validate all reservoir data**. After the objectives are set, reservoir data are gathered and incorporated into the reservoir model.

3. **Construct the reservoir model**. The reservoir model is built in the form of stacked grid blocks or cells. Figure 2.1 shows one example of representing real physical feature in terms of gridblocks. Properties of the reservoir are assigned to each of the cells. Even though in reality reservoir properties differ from one point to another, in the reservoir model the properties are assumed to be constant within a grid block.

4. **History match the reservoir model**. The simulation model could be improved with the help of production data, if available. The tuning process using historical production data is called "history matching".

5. **Run prediction cases.** Finally, various production schemes are evaluated and sensitivity analyses are conducted on different production and reservoir parameters.

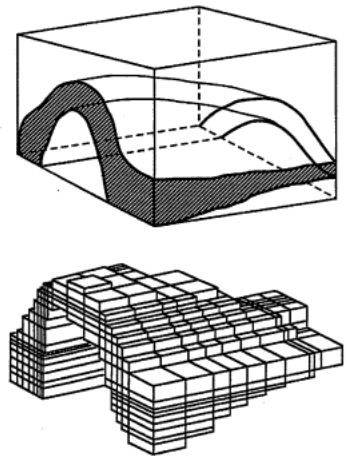


Figure 2.1: Three-dimensional gridblocks representation of an anticline (Ertekin et al., 2001)

2.1.2 Basic reservoir engineering concepts

In order to be able to model flow problems in reservoir, understanding basic concepts of reservoir engineering is important. This subsection will discuss some of these concepts: ock and fluid properties, fluid potential, and finally Darcy's law for fluid flow in porous media.

Rock properties
Basic rock properties discussed here are assumed to be independent of fluid flowing through it, and no chemical reaction occurs between the rock and the fluid.

1. **Porosity**, symbolized as ϕ, is defined as the ratio of pore space in a rock sample to the total volume of the rock sample. In reservoir simulation calculations, only interconnected pore spaces comes into concern.

2. **Permeability**, symbolized as k, is the capacity of a porous medium to transmit fluids through its interconnected pores. Generally, the permeability used in reservoir simulation calculation is the horizontal permeability, k_H, since flow in reservoirs are usually horizontal. However, for many cases vertical permeability k_V would also become important. In general, $k_H > k_V$.

Fluid properties
The properties described here are independent of the rock that it flows through. Fluid properties generally are highly dependent on reservoir pressure and temperature. The three types of fluid used in reservoir simulation calculations are oil, water, and gas (denoted as o, w, and g respectively).

1. **Fluid density**, symbolized as ρ, is defined as mass per unit volume. The density in reservoir conditions can be found from density in standard conditions (sc) and FVF

$$\rho_l = \frac{\rho_{l,sc}}{B_l} \tag{2.1}$$

   For gas, density can be found from real-gas law

$$\rho_g = \frac{pM}{zRT} \tag{2.2}$$

2. **Solution-gas/liquid ratio**, symbolized as R_s, is the volume of gas that must dissolve into a unit volume of liquid (both volumes measured at standard condition), for the liquid and gas system to reach equilibrium at reservoir conditions.

3. **Formation volume factor (FVF)**, symbolized as B_l, where $l = o, w, g$, is the ratio of volume of phase l at reservoir conditions to its volume at standard conditions.

4. **Fluid compressibility**, symbolized as c_l for liquid and c_g for gas, is defined as change in relative volume of a unit mass as a result of change in pressure at constant temperature. Liquid compressibility of liquid l is defined as

$$c_l = \frac{1}{\rho_l}\left(\frac{\partial \rho_l}{\partial p}\right)_T \tag{2.3}$$

While for gas, compressibility is

$$c_g = \frac{1}{p} - \frac{1}{z}\left(\frac{\partial z}{\partial p}\right)_T \tag{2.4}$$

ρ in Equations 2.3 and 2.4 is density, while z in Equation 2.4 is gas-compressibility factor that defines deviation of a real gas from an ideal gas.

5. **Fluid viscosity**, symbolized as μ, is the measure of how easy the fluid would flow under applied pressure. A more viscous fluid would be more difficult to move under the same amount of force. In general, when comparing the three phases, the order from least viscous to most viscous would be: gas, water, oil.

Fluid-Rock properties
This segment discusses two important properties that depends on both fluid and rock in the reservoir.

1. **Fluid saturation**, symbolized as S_l, is the fraction of porosity occupied by phase l, where $l = o, w, g$. For all available phases in pore space, then the total saturation would be 1. In multiphase flow with oil, gas, and water this would be

$$S_w + S_o + S_g = 1 \tag{2.5}$$

2. **Capillary pressure**, symbolized as P_c, is the result of capillary forces acting in small openings in pore systems between two or more phases. Capillary pressure can be defined as the pressure of the non-wetting phase minus the pressure of the wetting phase. For oil/water systems, commonly oil would be the non-wetting phase while water is the wetting phase. The capillary pressure between oil and water is then

$$P_{cow} = p_o - p_w \tag{2.6}$$

Capillary pressure is a function of saturation and also history of saturation (due to hysterisis effects), and differs for various reservoir rocks and fluids.

3. **Relative permeability**, symbolized as k_r, as explained when describing Equation 2.13, is the ratio of effective permeability of a phase flowing in multiphase flow to the absolute permeability of the medium. Relative permeability is generally modeled as a function of saturation between two phases (for example, oil/water and oil/gas relative permeability).

Fluid Potential
Fluid potential, Φ, is defined as the work required to transfer a mass of fluid from a state of atmospheric pressure and reference elevation/datum to the point in question. When comparing fluid potential in different points in space, the fluid would flow from a higher potential point to a lower potential one. Fluid potential in the form of an equation is

$$\Phi = p - \gamma Z \tag{2.7}$$

where p is pressure at that datum, γ is fluid gravity, and Z is the distance from reference elevation (positive value for vertical downward direction). For any arbitrary point with a new datum, fluid potential could be written as

$$\Phi - \Phi^o = (p - p^o) - \gamma Z \tag{2.8}$$

where values with superscipt o denotes those at datum, and values with no superscripts define values at the arbitrary point. The potential gradient, obtained by differentiating Equation 2.8, is

$$\vec{\nabla}\Phi = \vec{\nabla}p - \gamma\vec{\nabla}Z \tag{2.9}$$

While for multiphase flow, the potential gradient for each phase (oil, water, and gas) is

$$\vec{\nabla}\Phi_l = \vec{\nabla}p_l - \gamma_l\vec{\nabla}Z \tag{2.10}$$

where l can be o for oil, w for water, or g for gas.

It is imperative to remember that Z is positive in vertical downward direction, since in the realm of reservoir engineering (and petroleum engineering in general), the term for distance increases with increasing depth.

Darcy's Law
Henry Darcy formulated this law based on his experimental results in 1856. Darcy's law has been widely used to describe the movement of other fluids, including two or more phases, in consolidated rocks and other porous media (Craft et al., 1991). Darcy's law states that the apparent velocity (flow rate over a cross-area perpendicular to flow direction) of a single-phase fluid in a porous medium, u, is proportional to the permeability in the direction of flow, k, and the fluid potential gradient, $\vec{\nabla}\Phi$, while inversely proportional to the fluid viscosity, μ:

$$\vec{u} = -\frac{k}{\mu}\vec{\nabla}\Phi \tag{2.11}$$

Using the definition of velocity as flow rate, q, over a cross-sectional area perpendicular to the flow, A, and the definition of fluid potential gradient from Equation 2.9, Darcy's law can be reformulated to

$$\frac{\vec{q}}{A} = -\frac{k}{\mu}\left(\vec{\nabla}p - \gamma\vec{\nabla}Z\right) \tag{2.12}$$

However, when using Equation 2.12, it is necessary to remember there are assumptions and limitations imposed:

1. The fluid is single-phase, homogeneous, and Newtonian.
2. No chemical reaction occurs between the fluid and the porous medium.
3. Permeability is property of the medium, it is independent of the type of fluid, and the system's pressure and temperature.
4. The flow condition is laminar.

For multiphase flow, another term is added, which is relative permeability (k_r). Relative permeability of a phase can be found by dividing the effective permeability of the phase to the absolute permeability of the medium. Effective permeability is the permeability of a phase in a multiphase flow. The sum of effective permeabilities of phases present is always less than the absolute permeability (Dake, 1978). Equation 2.12 for phase $l = o, w, g$ in multiphase flow would become

$$\frac{\vec{q}_l}{A} = -\frac{kk_{rl}}{\mu_l}\left(\vec{\nabla}p_l - \gamma_l\vec{\nabla}Z\right) \tag{2.13}$$

2.2 Development of Single-Phase Flow Simulation

Before going to multiphase flow, the more basic single-phase flow will be discussed. This subsection will look at the basic equations involved in single-phase flow, and how some mathematical concepts would help model the equations for reservoir simulation.

2.2.1 Basic single-phase flow equation

The flow equation for single-phase flow in porous media is obtained by combining Darcy's law and the law of mass conservation. Darcy's law have been explained previously in Subsection 2.1.2. The law of mass conservation describes the material balance in a control volume. For any component c, the material balance is expressed as

$$(m_i - m_o)_c + (m_s)_c = (m_a)_c \tag{2.14}$$

In Equation 2.14, m_i denotes mass entering the control volume (mass in), m_o denotes mass leaving the control volume (mass out), m_s denotes mass entering/leaving the control volume via external pathways, in petroleum reservoir systems usually by wells (sink/source), and m_a is accumulated mass in the control volume. For single-phase three-dimensional flow, using a rectangular coordinate systems (x, y, z), the mass-conservation equation is

$$\begin{aligned} &-\frac{\partial}{\partial x}(\dot{m}_x A_x)\Delta x - \frac{\partial}{\partial y}(\dot{m}_y A_y)\Delta y - \frac{\partial}{\partial z}(\dot{m}_z A_z)\Delta z \\ &= V_b\frac{\partial}{\partial t}(m_v) - q_m \end{aligned} \tag{2.15}$$

In Equation 2.15, $\dot{m}$ is mass flux, A is the cross-section perpendicular to the dimension direction, V_b is the volume of the system, m_v is the total mass in the system, and q_m is mass entering/exiting via sink/source.
Combining Equation 2.12 (Darcy's law) and Equation 2.15 (Mass conservation law), the basic equation for single-phase flow can be developed

$$\begin{aligned} &\frac{\partial}{\partial x}\left[\frac{k_x A_x}{\mu_l B_l}\left(\frac{\partial p}{\partial x} - \gamma_l\frac{\partial Z}{\partial x}\right)\right]\Delta x + \frac{\partial}{\partial y}\left[\frac{k_y A_y}{\mu_l B_l}\left(\frac{\partial p}{\partial y} - \gamma_l\frac{\partial Z}{\partial y}\right)\right]\Delta y \\ &+ \frac{\partial}{\partial z}\left[\frac{k_z A_z}{\mu_l B_l}\left(\frac{\partial p}{\partial z} - \gamma_l\frac{\partial Z}{\partial z}\right)\right]\Delta z = V_b\frac{\partial}{\partial t}\left(\frac{\phi}{B_l}\right) - q_{l,sc} \end{aligned} \tag{2.16}$$

where $l = o, w, g$. Equation 2.16 is the fundamental equation in the development of reservoir simulation. For a slightly compressible system usually found in reservoir systems, the accumulated mass term becomes

$$V_b \frac{\partial}{\partial t}\left(\frac{\phi}{B_l}\right) = \frac{V_b \phi c_l}{B_l^o}\frac{\partial p}{\partial t} \tag{2.17}$$

where B^o denotes FVF at a reference pressure.

2.2.2 The concept of finite-difference

The basis of finite-difference concept is a sequence of mathematical operations that occurs at discrete points, whether in space or in time. Based on the location of the reference point in the calculation, finite-difference operations can be forward-difference, backward-difference, or central-difference. The most widely used application of finite-difference is in derivative analysis. The forward-difference definition of first derivative of $f(x)$ at point x_i is

$$\frac{df}{dx} \approx \frac{f(x_{i+1}) - f(x_i)}{h} \tag{2.18}$$

The approximation on Equation 2.18 will be better as h approaches zero.

2.2.3 Finite-difference for space and time derivative

Industry-standard reservoir simulation tools in general use a rectangular coordinate system that discretizes the system into block-shaped grids, called gridblocks/cells. Figure 2.2 illustrates the rectangular coordinate system for three-dimensional flow. Taking for example

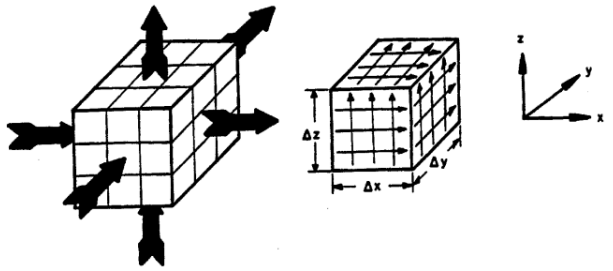


Figure 2.2: Rectangular coordinate system for 3D flow (x,y,z) (Ertekin et al., 2001)

one-dimensional flow in x direction from Equation 2.16, and knowing that $\partial Z/\partial x = 0$, then the basic flow equation can be discretized in space using central-difference into

$$\begin{aligned}\left(\frac{A_x k_x}{\mu_l B_l \Delta x}\right)_{i+1/2}(p_{i+1} - p_i) - \left(\frac{A_x k_x}{\mu_l B_l \Delta x}\right)_{i-1/2}(p_i - p_{i-1}) \\ + (q_{l,sc})_i = \left(\frac{V_b \phi c_l}{B_l^o}\right)_i \frac{\partial p_i}{\partial t}\end{aligned} \tag{2.19}$$

This can then be simplified into

$$T_{lx,i+1/2}(p_{i+1} - p_i) - T_{lx,i-1/2}(p_i - p_{i-1}) + (q_{l,sc})_i = \left(\frac{V_b \phi c_l}{B_l^o}\right)_i \frac{\partial p_i}{\partial t} \tag{2.20}$$

The coefficient T is called "transmissibility", and is a property of the porous medium and the flowing fluid.
For the time derivative, let us discretize at different time levels $n+1$ for next time step and n for the current time. Continuing from Equation 2.20, by applying time discretization the equation becomes

$$\begin{aligned} T_{lx,i+1/2}(p_{i+1}^n - p_i^n) - T_{lx,i-1/2}(p_i^n - p_{i-1}^n) \\ + (q_{l,sc})_i = \left(\frac{V_b \phi c_l}{B_l^o \Delta t}\right)_i (p_i^{n+1} - p_i^n) \end{aligned} \tag{2.21}$$

2.2.4 Explicit and implicit formulations

In reservoir simulation, the flow equations are to be solved from an appointed starting time until an appointed ending. Time is discretized into multiple "time steps". Depending on the finite-difference method used to discretize time, the calculation of the flow equations could be either explicit or implicit.

Explicit formulations use forward-difference in time, so the pressure for the next time step $(n + 1)$ can be directly calculated from known terms from the current time step (n). Equation 2.21 is explicit, and p_i^{n+1} can be calculated directly. On the other hand, implicit formulation uses backward-difference and cannot be computed directly. Equation 2.21 would become implicit if all time level at the left-hand side is changed to $n + 1$. To solve an implicit formulation for N time steps, the equation in each time step from $n = 1$ until $n = N$ have to be solved simultaneously using matrix computation. Even though explicit formulation is easier to solve, but it has been proven that implicit formulation is more stable and reliable (LeVeque, 2007). In order to create a robust and reliable reservoir simulation tool, implicit formulation have to be employed.

2.3 Development of Multiphase Flow Simulation

So far, the discussion has only concerned itself with single-phase flow. From here, the flow condition in which multiple phase flow together will be discussed. There are a lot of differences that arise from adding more phases to the flow system, but the underlying equations and concepts used are similar, and they only need to be extended accordingly.

2.3.1 Flow equations in multiphase flow

The goal here is to find the flow equation for each of the phases (oil,water,gas) based on the basic single-phase flow equation, which is Equation 2.16. For oil and water, simply

add relative permeability and saturation terms for each phase.

$$\begin{aligned}\frac{\partial}{\partial x}\left[\frac{k_x k_{ro} A_x}{\mu_o B_o}\left(\frac{\partial p_o}{\partial x}-\gamma_o\frac{\partial Z}{\partial x}\right)\right]\Delta x+\frac{\partial}{\partial y}\left[\frac{k_y k_{ro} A_y}{\mu_o B_o}\left(\frac{\partial p_o}{\partial y}-\gamma_o\frac{\partial Z}{\partial y}\right)\right]\Delta y\\ +\frac{\partial}{\partial z}\left[\frac{k_z k_{ro} A_z}{\mu_o B_o}\left(\frac{\partial p_o}{\partial z}-\gamma_o\frac{\partial Z}{\partial z}\right)\right]\Delta z=V_b\frac{\partial}{\partial t}\left(\frac{\phi S_o}{B_o}\right)-q_{o,sc}\end{aligned} \tag{2.22}$$

$$\begin{aligned}\frac{\partial}{\partial x}\left[\frac{k_x k_{rw} A_x}{\mu_w B_w}\left(\frac{\partial p_w}{\partial x}-\gamma_w\frac{\partial Z}{\partial x}\right)\right]\Delta x+\frac{\partial}{\partial y}\left[\frac{k_y k_{rw} A_y}{\mu_w B_w}\left(\frac{\partial p_w}{\partial y}-\gamma_w\frac{\partial Z}{\partial y}\right)\right]\Delta y\\ +\frac{\partial}{\partial z}\left[\frac{k_z k_{rw} A_z}{\mu_w B_w}\left(\frac{\partial p_w}{\partial z}-\gamma_w\frac{\partial Z}{\partial z}\right)\right]\Delta z=V_b\frac{\partial}{\partial t}\left(\frac{\phi S_w}{B_w}\right)-q_{w,sc}\end{aligned} \tag{2.23}$$

For gas, however, another thing have to be considered. Recall the discussion regarding solution gas-oil ratio (R_s) in Subsection 2.1.2. According to the black-oil model, mass transfer is possible between the oil and gas phases. Gas components can dissolve in oil to some extent, depending on reservoir conditions. Therefore, the flow equation should represent not only gas components in the gas phase, but also gas components in the oil phase.

$$\begin{aligned}&\frac{\partial}{\partial x}\left[\frac{k_x k_{rg} A_x}{\mu_g B_g}\left(\frac{\partial p_g}{\partial x}-\gamma_g\frac{\partial Z}{\partial x}\right)+R_s\frac{k_x k_{ro} A_x}{\mu_o B_o}\left(\frac{\partial p_o}{\partial x}-\gamma_o\frac{\partial Z}{\partial x}\right)\right]\Delta x\\ &+\frac{\partial}{\partial y}\left[\frac{k_y k_{rg} A_y}{\mu_g B_g}\left(\frac{\partial p_g}{\partial y}-\gamma_g\frac{\partial Z}{\partial y}\right)+R_s\frac{k_y k_{ro} A_y}{\mu_o B_o}\left(\frac{\partial p_o}{\partial y}-\gamma_o\frac{\partial Z}{\partial y}\right)\right]\Delta y\\ &+\frac{\partial}{\partial z}\left[\frac{k_z k_{rg} A_z}{\mu_g B_g}\left(\frac{\partial p_g}{\partial z}-\gamma_g\frac{\partial Z}{\partial z}\right)+R_s\frac{k_z k_{ro} A_z}{\mu_o B_o}\left(\frac{\partial p_o}{\partial z}-\gamma_o\frac{\partial Z}{\partial z}\right)\right]\Delta z\\ &=V_b\frac{\partial}{\partial t}\left(\frac{\phi S_g}{B_g}+\frac{\phi R_s S_o}{B_o}\right)-q_{g,sc}\end{aligned} \tag{2.24}$$

Equations 2.22, 2.23, and 2.24 are the fundamental equations for each phase in multiphase flow. These equations can be further extended if the reservoir simulation concern more situations that could happen in real world application, such as capillary pressure between phases, reaction between rock and fluid, etc.

2.3.2 Finite-difference approximations in multiphase flow

Using the same thought process as before when applying finite-difference to change Equation 2.16 to Equation 2.20, the multiphase flow equations can be discretized. Taking for example one of the dimensions, the equations for oil, water, and gas would respectively become

$$\Delta\Big[T_o\big(\Delta p_o-\gamma_o\Delta Z\big)\Big]=\frac{V_b}{\Delta t}\Delta_t\left[\frac{\phi(1-S_w-S_g)}{B_o}\right]-q_{o,sc} \tag{2.25}$$

$$\Delta\Big[T_w\big(\Delta p_w - \gamma_w \Delta Z\big)\Big] = \frac{V_b}{\Delta t}\Delta_t\left[\frac{\phi S_w}{B_w}\right] - q_{w,sc} \tag{2.26}$$

$$\begin{aligned}&\Delta\Big[T_g\big(\Delta p_g - \gamma_g \Delta Z\big)\Big] + \Delta\Big[T_o R_s\big(\Delta p_o - \gamma_o \Delta Z\big)\Big] \\ &= \frac{V_b}{\Delta t}\left[\Delta_t\Big[\frac{\phi S_w}{B_w}\Big] + \Delta_t\Big[\frac{\phi R_s(1 - S_w - S_g)}{B_o}\Big]\right] - q_{g,sc}\end{aligned} \tag{2.27}$$

The plain Δ symbol denotes discretization in space, while Δ_t denotes discretization in time. While time increment Δt is clearly present on the equations, the space increments reside within the transmissibility terms.

During reservoir simulation, as reservoir conditions constantly change, a lot of properties included in flow equations will also change. For example, phase density changes depending on pressure, and relative permeability changes depending on saturation. In order to develop a reliable simulation, it is highly important to fully understand the characteristics of rock and fluid properties of the reservoir to be simulated. This could be done by taking rock and fluid samples from the reservoir and conduct multiple tests on them.

2.3.3 Methods for solving multiphase flow equations

Multiphase flow simulation yields multiple finite-difference equations for each gridblock that accounts for each phase. The goal of simulating the multiphase flow generally is to find the pressures and saturations at each time step in each grid cell. This subsection will discuss three commonly used methods for solving multiphase flow equations: Fully-Implicit method (also known as the Simultaneous Solution method), IMPES method, and SEQ method.

Fully-Implicit method
The fully-implicit method aims to solve all the multiphase flow equations simultaneously, hence the alternative name Simultaneous Solution. This method is considered the most powerful, albeit the most computational heavy for a single time step. The different terms in the flow equations, including the objective values (pressures and saturations) are represented by matrices and vectors. A matrix form equation is then developed that relates the matrices and vectors, and Newton's iteration method can then be applied in order to find the objective values for the next time step. The connection among the grid blocks are also important. In a reservoir model, determining the connections are a must before going to the simulation. In a three-dimensional space, one grid cell is usually connected to six cells around it: the cells directly above, below, west, east, north, and south of it. These connections would then be accounted for when building the matrix for each gridblock. Figure 2.3 shows how the flow matrix would look like for a cell with six connections as described before. The matrix form equation to solve in the Fully-Implicit method is

$$\Big([F]^{(v)} - [C]^{(v)} + [Q]^{(v)}\Big)\Big(\vec{X}^{(v+1)} - \vec{X}^{(v)}\Big) = -\left\{\vec{F}^{(v)} - \Big[\vec{C}^{(v)} - \vec{C}^{n}\Big] + \vec{Q}^{(v)}\right\} \tag{2.28}$$

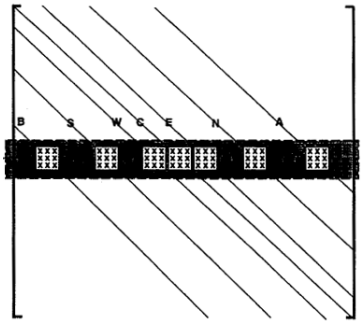


Figure 2.3: Structure of a multidiagonal flow matrix representing connections of a particular "central (C)" gridblock. Pointers for neighboring gridblocks: below (B), south (S), west (W), east (E), north (N), and above (A) (Ertekin et al., 2001)

where v denotes the index in the Newton's iteration. For the matrices in the left-hand side and the vectors in the right-hand side, F is the flow term which includes transmissibility, C is the accumulation term which includes compressibility, Q is the sink/source term which includes wells, and X is the objective vector which include pressures and saturations. The index v would start from 0 until $\vec{X}^{(v+1)} - \vec{X}^{(v)}$ converges. The matrices represent dependence of flow parameters to pressure and saturation, and the vectors represent the value of those parameters under pressures and saturations of current iteration.

Implicit-Pressure Explicit-Saturation (IMPES) method
The main objective of IMPES is to eliminate all saturation unknowns by combining flow equations to get a single pressure equation for each gridblock. To be able to do this, transmissibilities and production/injection are evaluated at the current time step while pressures are evaluated at the next time step (Berg, 2019). Once pressure for the next time step is obtained, saturations are calculated explicitly. Even though the calculation is simpler than Fully-Implicit method, IMPES is less accurate for general use. It is recommended to only use IMPES if saturations in the system change slowly over time.

Sequential Fully-Implicit (SEQ) method
SEQ method aims to be more accurate and stable than IMPES, but without the need to solve pressures and saturations simultaneously like Fully-Implicit method. To do this, SEQ divides the workflow into two sequences, solving pressures implicitly in one step and then solving saturations implicitly in the next step.
In the implicit-pressure step, transmissibilities and production/injection are evaluated at current time step, while pressures are evaluated at the next time step just like in IMPES. However, there are no eliminations of saturation unknowns. In the implicit-saturation step, the saturations obtained in the first step is used to solve the saturations implicitly. Even though SEQ is more reliable than IMPES, the independent solving of pressure and saturation would result in material balance not satisfied for every phase.

2.4 OPM Flow as a Reservoir Simulator

This section will discuss the OPM Flow reservoir simulator, and will take a look at the inner workings of the software, and how it handles the equations associated with reservoir and well properties as the OPM initiative team have described in detail in Rasmussen et al. (2019).

2.4.1 Reservoir and well model

OPM Flow uses the black-oil model, in which the reservoir fluid consists of three phases (oleic, aqueous, gaseous) and three components (oil, water, gas). Oil and gas components can be found on oleic and gaseous phases, and component transfer between the two phases are allowed. For the objective variables, OPM Flow chooses oil pressure (p_o), water saturation (s_w), and a variable third variable. The third variable could be gas saturation (s_g), solution gas-oil ratio (r_{go}), or solution oil-gas ratio (r_{og}), depending on the presence of oleic and/or gaseous phase. The choice of the third variable is made for each grid cell. For initialization, OPM Flow uses initial values of pressure and saturation specified by the user to define initial conditions, while boundary conditions use no-flow boundaries (Neumann boundary condition) as default. More recent versions have added some aquifer models as constant-pressure boundaries. Rock and fluid properties could be defined by including tables in the input file. From the tables, OPM Flow would interpolate or extrapolate to some degree the dependence of the properties to pressure or saturation.

OPM Flow has two different models for wells: standard well and multi-segment well. The standard well model describes flow in each well with one set of objective variables. The variables for one well in a three-phase system includes total flow rate Q_t, water F_w and gas F_g fractions, and bottom-hole pressure p_{bh}. The well and the reservoir are coupled to keep the system closed, by coupling mass conservation equations for each component in the reservoir and in the well. More limitations can also be imposed via well control, in which wells are only allowed to operate above a certain Q_t and/or p_{bh}. The multi-segment well model could be used to model more advanced wells such as multilateral wells or horizontal wells. In this model, the well is split into different segments, each having inlet and outlet nodes. The pressure of the nodes replace p_{bh} as a part of the objective variables. Each node can connect to two or more nodes from different segments, but each node can only be a part of one segment. For a node with multiple connections, it is important to make sure pressure of that node would result in the same value when calculating well equations for each connected segment.

2.4.2 Problem solving strategy

The reservoir and well equations described in the reservoir and well model are subject to a fully-implicit solution method. Newton's iterations are used to iterate the same matrix form equation as Equation 2.28. The equation can be simplified into

$$J(y_n)(y_{n+1} - y_n) = -R(y_n) \tag{2.29}$$

The objective variables are located within y in which $y = (p_o, s_w, x)$, which would be the objective vector $\vec{X}$ in Equation 2.28. Recall that the third variable is x since it could be different for each grid cell in each time step. All the matrices in left-hand side of Equation 2.28 are combined into a single Jacobian matrix J, and all the vectors on the right-hand side are combined into a single residual vector R. Each term in Equation 2.29 includes reservoir and well equations, as well as the coupling equations.

Developing the Jacobian matrix J requires the simulator to compute various partial derivatives in each iteration. Traditionally, the partial derivatives would be computed analytically and would sometimes produce unreliable results as a result of manual calculations of partial derivatives, while also being time-consuming because hard-coding each partial derivative is inefficient. OPM Flow has employed a way to mitigate this problem, by the use of Automatic Differentiation (AD). The basic idea of AD is for a complicated derivation that needs to utilize the chain rule, AD would compute all sequence of values in the chain rule and the derivations of the sequences automatically (Neidinger, 2010). A simple example would be calculating the derivative of $y = \cos(x^2 + 1)$ at $x = 4$. Using AD, the software would not only calculate the left column in Table 2.1, but also the right column automatically.

Variables	Derivatives
$x = 4$	
$y_1 = x^2 + 1 = 17$	$y_1^{'} = 2x = 8$
$y = \cos(y_1) = -0.275$	$y^{'} = -\sin(y_1)y_1^{'} = 7.691$

Table 2.1: Example of Automatic Differentiation (AD) in action.

2.4.3 Input and output handling

In order to be relevant and easily recognizable in the industry, OPM Flow uses a widely-used format when reading and writing files. The ECLIPSE simulator from Schlumberger is the dominant simulator in the industry, and OPM Flow has decided to support the I/O formats used by ECLIPSE. For input, OPM Flow reads from a `.DATA` file that contains various keywords that have different commands and/or specifications for the simulation. The details of each keyword and how to use them have been listed in the OPM Flow manual (Baxendale). The `.DATA` file is organized into several sections, each dealing with a specific part of the simulation. The sections are listed in Table 2.2.
OPM Flow has a range of different output files. Summary files (`.UNSMRY` and `.SMSPEC`) contain well and reservoir data in time-series format that then can be viewed using visualization tools such as *ResInsight* or *Eclipse Office*. There is also a restart file (`.UNRST`) that can be used for restarting the simulation from an arbitrary checkpoint, alongside `.INIT` and `.EGRID` files that provide the initialization and grid model. There are also documentation files such as `.PRT` and `.DBG` that let users see the step-by-step process of the simulation and lists of errors or warnings that occured throughout the run. All of these

Section	Function
RUNSPEC	Overall simulation settings (grid dimensions, phases present, etc.)
GRID	Grid geometry and petrophysical properties.
EDIT	Modifications and multipliers to the GRID section.
PROPS	Flow parameters, fluid properties (PVT), and rock properties.
REGIONS	Define different regions in the model that have different properties.
SOLUTION	Model initialization (Water saturation, pressure, fluid contacts, etc.)
SUMMARY	Choose variables to save as output.
SCHEDULE	Well control and parameters, time step definition.

Table 2.2: The sections in a .DATA file and their functions.

files are formatted to be similar to the ones used in Eclipse, so users already familiar with Eclipse would have no problem adjusting when they start working with OPM Flow.

2.5 Known Issues in OPM Flow

As explained in Section 1.2, this study will investigate the known issues encountered before in older versions of OPM Flow. This section will describe the issues in more detail.

2.5.1 Convergence issues

This issue was caught into attention when working on Exercise in Reservoir Simulation course in NTNU Petroleum Engineering department of Spring 2019 semester. Three one-dimensional single-phase cases were simulated using Flow 2018.10, each having different grid size. **SinglePhase_Ori_2** has 2 meters grid size, **SinglePhase_Ori_4** has 4 meters grid size, and **SinglePhase_Ori_20** has 20 meters grid size. Each of them has one injector well at the left boundary set at 100 bar BHP, and one producer well at the right boundary set at 200 bar BHP. After the simulation has finished, the average field pressure (FPR) of each case is plotted. Figure 2.4 shows the FPR of the cases compared to each other.

The resulting FPR from Figure 2.4 was unexpected. Since the injector was operating at 200 bar and the producer was at 100 bar, the FPR should go to around 150 bar, but that is not what happened here. The effect seems to be worse for the smallest grid size case, where the FPR went down and then up again during the beginning of the simulation. This could be attributed to some errors in calculating FPR, or this could be a bigger error in calculating pressure inside the reservoir. However, when plotting the pressure values in each gridblock it was found that the pressure was close to expected value. Figure 2.5 shows the pressure distribution plot for case SinglePhase_Ori_2. The inaccuracy of FPR plot in Figure 2.4 then could be a result of false reporting or false formulation of FPR calculation, but it is worth investigating to see if the new Flow version has fixed it. Another issue that could be clearly seen in Figure 2.5 is the numerical errors at the early timesteps of the simulation. Berg (2019) has explained the cause of this error in his Lecture Notes, that the

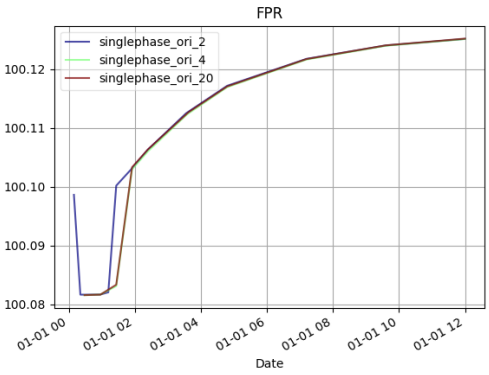


Figure 2.4: Average field pressure (FPR) of the single-phase cases simulated using Flow 2018.10.

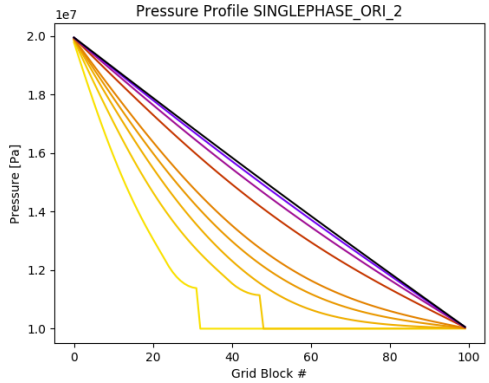


Figure 2.5: Pressure distribution plot in each gridblock of the original case tested using Flow 2018.10 with grid size of 2 meters.

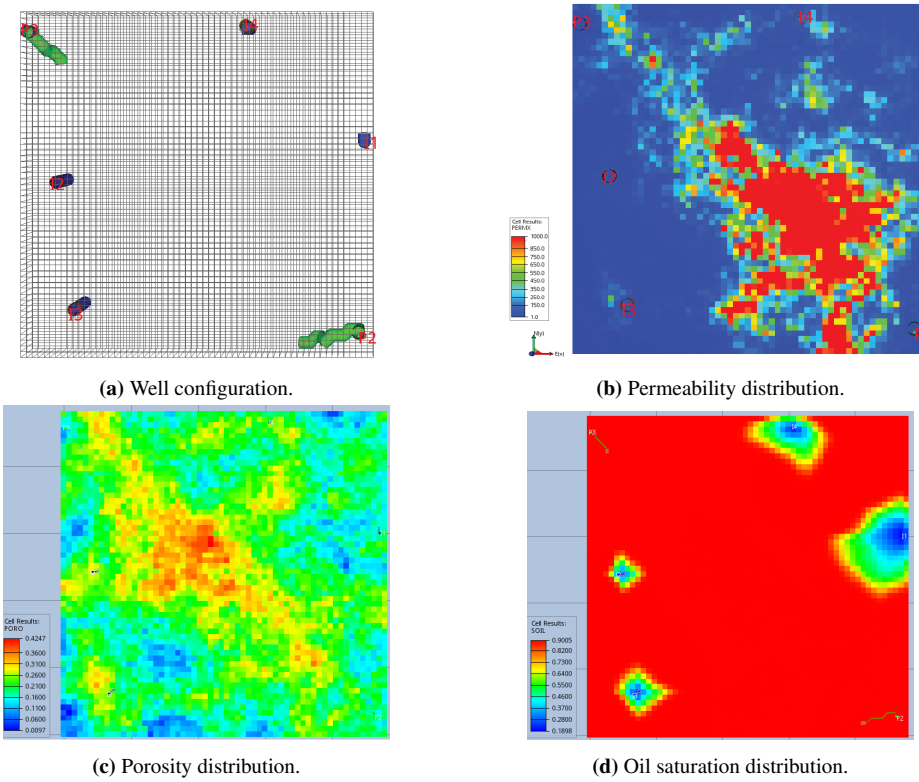
timestep used is too large for the pressure gradient specified in the model. This study then would also see if Flow has fixed this numerical issue.

2.5.2 ”Oil injection” issues in waterflooding simulation

The next issue to check originated from the course project that was also in Reservoir Simulation course in NTNU Petroleum Engineering department of Spring 2019 semester. The purpose of the project was to benchmark optimization algorithms for well control optimization. The reservoir used is a two-dimensional model populated with oil and water. The simulation scenario is a waterflooding process using four injector wells and two producer wells. The optimization algorithms was supposed to give the best placement locations for the wells in order to achieve NPV as high as possible.

OPM Flow version 2018.10 was used to simulate the case, and the results that came out was satisfactory enough. At first, the problem was not noticed until one of the project groups checked the field injection rate. It turns out that some wells inject oil to keep some wells in operation when it was supposed to shut down (see Figure 1.2). This is a serious problem because then the well placement results become invalidated and the purpose of the project is not satisfied. This study will check whether this problem persists in the newer versions of OPM Flow, using the same case and scenario.

Figure 2.6 shows the well configuration, as well as permeability, porosity, and oil saturation distribution in the model. To incorporate the optimization algorithm, **FieldOpt** software from NTNU's Petroleum Cybernetics Group is used. The software is already installed at `pet.geo.ntnu.no` server. A `.JSON` file is used to bridge the results of running the model simulation in OPM Flow and optimizing the well placement in FieldOpt. A snippet of the `.JSON` file is presented as Figure 2.7.



(a) Well configuration.

(b) Permeability distribution.

(c) Porosity distribution.

(d) Oil saturation distribution.

Figure 2.6: Characteristics of reservoir model used in Reservoir Simulation course project Spring 2019.

Figure 2.7: Screenshot of a `.JSON` file used to run optimization algorithms. The file contains optimization parameters and constraints imposed in each well at different timesteps.

Chapter 3

Methodology

This chapter presents in detail the steps taken to conduct this study, beginning from compiling OPM Flow modules to be able to run Flow, then creating the cases for testing the Flow reservoir simulator, and finally comparing results from Flow to other simulators as a mean to validate it.

3.1 Compilation of OPM Flow Modules

Before testing any cases, OPM Flow have to be installed first. It is recommended to use Ubuntu or MacOS to run OPM Flow since the installation process is made with Linux in mind. Updating the simulator is also much easier on Linux platforms. Installing OPM Flow can be done either by downloading and installing from binary packages, or by building from source. This section will discuss both methods of installing to a device using Ubuntu.

3.1.1 Installing OPM Flow from binary packages

Installing from packages are simple and lets users stay updated to the most recent stable release version. OPM Flow usually releases a new version of their software twice a year, in May and November. To install from binary packages, a user could use the `bash` script included in Appendix A.1 that then can be executed in terminal.

The script adds OPM repository available online as a PPA (Personal Package Archive) to the device running Ubuntu, and then installs the software included in the repository.

3.1.2 Building OPM Flow from source

A more manual and involved method of installing OPM Flow is by building it from source. Simply put, it means taking the OPM Flow source code that is still actively under development, and install it to your device. Before even attempting to fetch the source code,

one must make sure their device satisfies the prerequisites. OPM uses various third-party libraries and frameworks, such as **Boost** that supports tasks such as linear algebra and pseudorandom number generation, **Dune** for solving partial differential equations in grid-based models, and **Zoltan** for grid partitioning and load balancing. Other than those libraries, some tools also have to be installed first:

- git: for cloning the repo to get the source code
- C++ compiler and cmake: for building the software
- doxygen and latex: for building the docs

After all prerequisites are ready, it is time to clone the source code from the OPM github repository at `https://github.com/OPM`. Opening the repository page will reveal the different modules available to clone and build. Each module could be cloned by passing the command `git clone <module url>` in the terminal. After cloning, each module is then built using a C++ compiler and cmake. In total, building all OPM modules take around 30-45 minutes on Ubuntu version 18.04. It is **very important** to build the modules in the following order:

1. opm-common
2. opm-material
3. opm-grid
4. opm-models
5. opm-simulators
6. opm-upscaling

The order above is required because of the structure OPM Flow is built upon. The modules have interdependencies on each other, so a module could not work it depends on another module that has not been built yet. Figure 3.1 shows the module structure in OPM and the interdependencies among them, indicated by arrows. To make the building process easier,

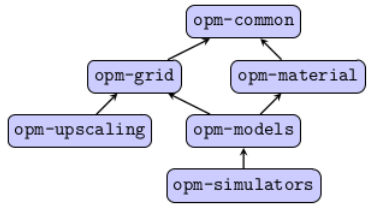


Figure 3.1: The OPM module structure at 2019.10 release version. Interdependencies among modules are indicated by arrows. (Rasmussen et al., 2019)

one can use the `bash` script included in Appendix A.2 that lets users build OPM from scratch in one execution.

3.2 Building Cases for Testing

Several cases are built to test whether some known issues still persist in the newest version of OPM Flow. The first issue to be checked is the convergence issue. As mentioned in Section 1.1, a simple single-phase, one-dimensional grid case was the one that confirms the existence of the issue in OPM Flow. The cases that will be built will be based on that case, but the scale of the model will be reduced to test how far until the simulator get convergence issues. The original case have grid size of 2 meters in every dimension, and three new cases are built with grid size of 2 meters, 0.1 meter, and 0.01 meter. Reservoir properties and grid parameters for the cases is listed in Table 3.1.

Grid Parameters		
Grid Blocks	100	
Rock Properties		
ϕ	0.20	
k	100	mD
Water Properties		
ρ_w	1025	kg/m^3
B_w	1.01	m^3/m^3
c_w	10^{-4}	1/bar
μ_w	1.0	cp
Initialization		
S_w	1.0	(water only)
p_i	100.0	bar

Table 3.1: Grid parameters and reservoir properties used in single-phase cases to test convergence issues.

For the well specifications, each case has two wells: an injector at the left boundary and a producer at the right boundary. Both wells are Bottom Hole Pressure (BHP)-controlled, meaning a constant pressure is applied to each well until the reservoir conditions do not satisfy it anymore. The production well is set to operate at BHP of 100 bar, while the injector pressure is different for each case. The three cases have different model size, so as a fixed value we have to set something that should have a same value in all cases. It is decided that the pressure gradient in each case should be the same, at 0.5 bar/m. Pressure gradient is chosen as a fixed variable because it will create a same flow velocity in all cases, according to the Darcy's Law (Equation 2.12. The timestep applied to each case also depends on the model size. Table 3.2 presents the case identification, alongside with well and timestep specifications in each of the three single-phase cases.

Apart from the single-phase cases, three two-phase cases are also built with the same specifications as in Tables 3.1 and 3.2. The difference is now there are two phases that exists within the system: water and oil. The reason to test two-phase cases is because OPM Flow is intended for multiphase flow, so it would be interesting if any convergence issues

Case ID	SinglePhase_2	SinglePhase_01	SinglePhase_001
Gridblock size (m)	2	0.1	0.01
Injector BHP (bar)	200	105	100.5
Timestep (days)	0.05	0.0001	0.00002
Timestep (s)	4320	8	1

Table 3.2: Case ID, well and timestep specifications for the single-phase cases.

would also occur at its intended usage. While rock properties and water properties remain the same, oil properties and water-oil relative permeabilities have to be introduced for the two-phase cases. Table 3.3 lists oil properties and initialization parameters, while relative permeability curve of the water-oil system is presented as Figure 3.2.

Oil Properties		
ρ_o	849	kg/m^3
B_o @ 100 bar	1.02	m^3/m^3
c_o	8x10^{-4}	1/bar
μ_o @ 100 bar	2.99	cp
Initialization		
S_w	0.2	
p_i	100.0	bar

Table 3.3: Oil properties and initialization parameters in two-phase cases to test convergence issues.

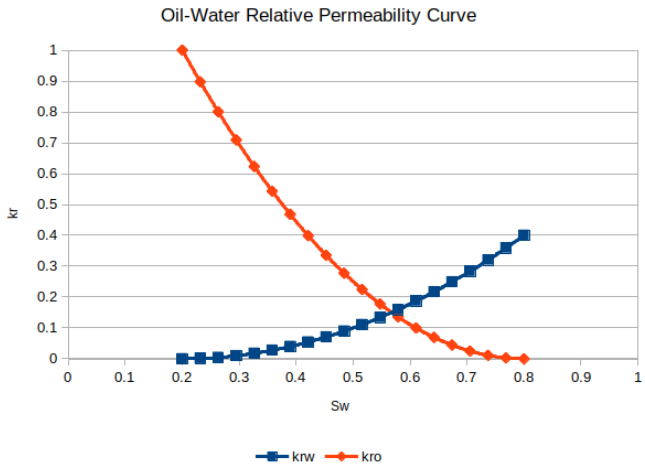


Figure 3.2: Oil-water relative permeability curve for the two-phase cases.

The well specifications in two-phase cases are the same as in the single-phase cases. The timestep, however, is made different. It would be preferable to be able to observe saturation front advancement in the reservoir throughout the simulation run, so the timestep for each case is made accordingly so the end results will let us see a significant saturation change across the model. The number of time steps are set to 100, to make the simulation runtime not too long. Table 3.4 presents the case identification, alongside with well and timestep specifications in each of the three two-phase cases.

Case ID	**TwoPhase_2**	**TwoPhase_01**	**TwoPhase_001**
Gridblock size (m)	2	0.1	0.01
Injector BHP (bar)	200	105	100.5
Timestep (days)	1.0	0.008	0.0008
Timestep (s)	86400	691	69

Table 3.4: Case ID, well and timestep specifications for the two-phase cases.

For the oil injection issue described in Subsection 2.5.2, the same case used in the project course in Spring 2019 will be tested again, this time using the newer versions of Flow: 2019.04 and 2019.10. Then, the results will be plotted to see if any oil injections occur, and how the new findings of this study affects the validity of the project results from Spring 2019.

3.3 Benchmarking with Schlumberger ECLIPSE

Answering the need to validate OPM Flow in Section 1.2, this study have done comparisons of OPM Flow with Sclumberger ECLIPSE reservoir simulator, which is the most widely-used simulation tool in the reservoir engineering discipline. Figure 3.3 shows the appearance of the simulator.

The comparison using ECLIPSE uses the same set of cases as explained in Section 3.2. The cases are run and then plots of results of interest are taken to be directly compared to resulting plots from OPM Flow. Ideally, the results should be similar or very close to each other, but it is expected that ECLIPSE gives a more accurate result since it has been developed for a longer time. Nevertheless, it would be interesting to see how OPM Flow would compare to ECLIPSE.

3.4 Building a Python-Based Fully-Implicit Simulator

A Python-based fully-implicit reservoir simulator is built from scratch. It is hoped that by doing this, not only it could help validate OPM Flow, but also could be a starting point to build a validation tool for any reservoir simulator that uses fully-implicit method. Python is chosen because the language is increasingly popular for its simplicity and intuitiveness. The model used in the Python simulator is also the same as cases in Section 3.2, however,

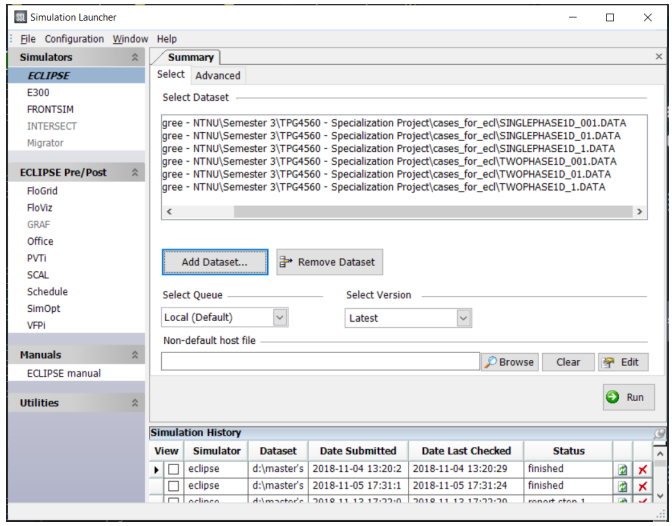


Figure 3.3: The menu interface of Schlumberger ECLIPSE reservoir simulator.

only the two-phase cases are tested with the Python simulator. The full Python script of the self-made reservoir simulator is presented as Appendix B.

The Python script is built using Equation 2.28, the fully-implicit matrix form equation, as the foundation. Each of the matrices and vectors in the equation are described for the oil/water two-phase system with N amount of gridblocks. The vectors contain subvectors for each gridblock in the system, and the Equations 3.1 to 3.4 show how the subvectors are arranged in each vector:

$$\vec{X} = \begin{pmatrix} \vec{X_0} & \vec{X_1} & \dots & \vec{X_{N-1}} \end{pmatrix}^T \tag{3.1}$$

$$\vec{F} = \begin{pmatrix} \vec{F_0} & \vec{F_1} & \dots & \vec{F_{N-1}} \end{pmatrix}^T \tag{3.2}$$

$$\vec{C} = \begin{pmatrix} \vec{C_0} & \vec{C_1} & \dots & \vec{C_{N-1}} \end{pmatrix}^T \tag{3.3}$$

$$\vec{Q} = \begin{pmatrix} \vec{Q_0} & \vec{Q_1} & \dots & \vec{Q_{N-1}} \end{pmatrix}^T \tag{3.4}$$

where $\vec{X}$ is the unknown vector, $\vec{F}$ is the flow vector, $\vec{C}$ is the accumulation vector, and $\vec{Q}$ is the sink/source vector. The subscript in each subvector denotes the grid index i, going from $i = 0$ to $i = N - 1$ for N gridblocks (indexing in Python starts from 0).

Next, let us see the elements of each subvector of each vector. Starting from the unknown subvector $\vec{X}_i$, it is populated by the objective values that are to be solved by this script: the oil pressure P_o and the water saturation S_w of each gridblock i. The flow subvector $\vec{F}_i$ contains transmissibilities and pressure difference for each phase. The accumulation subvector $\vec{C}_i$ contains the pore volume of each grid and the timestep size used. Finally, the

sink/source subvector $\vec{Q}_i$ contains production/injection terms in each grid.

$$\vec{X}_i = \begin{pmatrix} S_{w_i} \\ P_{o_i} \end{pmatrix} \tag{3.5}$$

$$\vec{F}_i = \begin{pmatrix} \sum_{m\in\psi_i} T_{w_{i,m}}(P_{o,m} - P_{o,i}) \\ \sum_{m\in\psi_i} T_{wi,m}(P_{o,m} - P_{o,n}) \end{pmatrix} \tag{3.6}$$

$$\vec{C}_i = \begin{pmatrix} \frac{V_b}{\Delta t}\left[\frac{\phi S_w}{B_w}\right] \\ \frac{V_b}{\Delta t}\left[\frac{\phi(1-S_w)}{B_w}\right] \end{pmatrix}_i \tag{3.7}$$

$$\vec{Q}_i = \begin{pmatrix} q_{w,sc_i} \\ q_{o,sc_i} \end{pmatrix} \tag{3.8}$$

In Equation 3.6, the summation $\sum_{m\in\psi_n}$ denotes the sum of the transmissibilites and pressure change between gridblock i and gridblocks m, which is every gridblock in connection with gridblock i. For the one-dimensional model used in building this script, gridblock i is connected with gridblocks $i-1$ and $i+1$, except for the left gridblock 0 that only connects with gridblock 1, and right gridblock $N-1$ that only connects with gridblock $N-2$. The subscript sc in flowrates in Equation 3.8 means it is measured in standard condition, but it is not too significant here because the water and oil FVF is close to 1.0 (see Tables 3.1 and 3.3).

The transmissibility is a parameter that needs to be carefully conditioned, because it is defined for $i+1/2$ and $i-1/2$ gridblocks when calculating on grid i, while being dependent on both pressure and saturation change (recall Equation 2.20). For this study, upstream weighting is used, meaning the transmissibility on $i+1/2$ and $i-1/2$ gridblocks are approximated to be equal to transmissibility of the nearest upstream gridblock. In this case, the flow goes from grid $i-1$ to i to $i+1$, so the transmissibility on $i+1/2$ is set to be equal to transmissibility on i, and the transmissibility on $i-1/2$ is set to be equal to transmissibility on $i-1$.

For the Jacobian matrices on the left-hand side of Equation 2.28, each of the matrices contains submatrices associated with each gridblock. Equations 3.9 to 3.11 show how the

submatrices are arranged in each matrix:

$$[F] = \begin{pmatrix} F_{0,0} & F_{0,1} & & & & \\ F_{1,0} & F_{1,} & F_{1,2} & & & \\ & F_{2,1} & F_{2,2} & F_{2,3} & & \\ & & \ddots & \ddots & & \\ & & & F_{N-2,N-1} & F_{N-1,N} \end{pmatrix} \tag{3.9}$$

$$[C] = \begin{pmatrix} C_0 & & & \\ & C_1 & & \\ & & \ddots & \\ & & & C_{N-1} \end{pmatrix} \tag{3.10}$$

$$[Q] = \begin{pmatrix} Q_0 & & & \\ & Q_1 & & \\ & & \ddots & \\ & & & Q_{N-1} \end{pmatrix} \tag{3.11}$$

The empty regions in the matrices would be populated by zeros. The arrangement of non-zero rows and columns shows how a grid is dependent to other grids. For parameters inside the accumulation matrix $[C]$ and the sink-source matrix $[Q]$, a grid is only dependent to itself. However, the flow terms for a grid i in the flow matrix $[F]$ are dependent on flow terms of the connected gridblocks $m \in \psi_i$.

The Jacobian submatrices consist of the dependencies of the flow, accumulation, and sink/-source terms to the unknowns (P_o and S_w) in terms of partial derivatives. The flow submatrix $F_{i,m}$, consists of

$$F_{i,m} = \begin{pmatrix} \frac{\partial F_{w_i}}{\partial S_{w_m}} & \frac{\partial F_{w_i}}{\partial P_{o_m}} \\ \frac{\partial F_{o_i}}{\partial S_{w_m}} & \frac{\partial F_{o_i}}{\partial P_{o_m}} \end{pmatrix} \tag{3.12}$$

where for $i = m$:

$$\frac{\partial F_{w_i}}{\partial S_{w_m}} = \sum_{m \in \psi_i} \left(\frac{\partial T_{w_{i,m}}}{\partial S_{w_i}} (P_{o_m} - P_{o_i}) \right) \tag{3.13}$$

$$\frac{\partial F_{w_i}}{\partial P_{o_m}} = \sum_{m \in \psi_i} \left(-T_{w_{i,m}} + \frac{\partial T_{w_{i,m}}}{\partial P_{o_i}} (P_{o_m} - P_{o_i}) \right) \tag{3.14}$$

$$\frac{\partial F_{o_i}}{\partial S_{w_m}} = \sum_{m \in \psi_i} \left(\frac{\partial T_{o_{i,m}}}{\partial S_{w_i}} (P_{o_m} - P_{o_i}) \right) \tag{3.15}$$

$$\frac{\partial F_{o_i}}{\partial P_{o_m}} = \sum_{m \in \psi_i} \left(-T_{o_{i,m}} + \frac{\partial T_{o_{i,m}}}{\partial P_{o_i}} (P_{o_m} - P_{o_i}) \right) \tag{3.16}$$

and for $i \neq m$:

$$\frac{\partial F_{w_i}}{\partial S_{w_m}} = \frac{\partial T_{w_{i,m}}}{\partial S_{w_m}}(P_{o_m} - P_{o_i}) \tag{3.17}$$

$$\frac{\partial F_{w_i}}{\partial P_{o_m}} = T_{w_{i,m}} + \frac{\partial T_{w_{i,m}}}{\partial P_{o_i}}(P_{o_m} - P_{o_i}) \tag{3.18}$$

$$\frac{\partial F_{o_i}}{\partial S_{w_m}} = \frac{\partial T_{o_{i,m}}}{\partial S_{w_i}}(P_{o_m} - P_{o_i}) \tag{3.19}$$

$$\frac{\partial F_{o_i}}{\partial P_{o_m}} = T_{o_{i,m}} + \frac{\partial T_{o_{i,m}}}{\partial P_{o_i}}(P_{o_m} - P_{o_i}) \tag{3.20}$$

The partial derivatives of transmissibility to P_o and S_w will be zero for the following conditions:

- Partial derivative of transmissibility at i to neighboring gridblock's P_{o_m} and S_{w_m} is zero if i is upstream to m.
- Partial derivative of transmissibility at i to its own P_{o_i} and S_{w_i} is zero if m is upstream to i.

The accumulation submatrix C_i consists of:

$$C_i = \begin{pmatrix} C_{ww_i} & C_{wp_i} \\ C_{ow_i} & C_{op_i} \end{pmatrix} \tag{3.21}$$

where

$$C_{ww_i} = \left\{ \frac{V_b}{\Delta t}\left(\frac{\phi}{B_w}\right) \right\}_i \tag{3.22}$$

$$C_{wp_i} = \left\{ \frac{V_b}{\Delta t}\left(\frac{\phi^{'}}{B_w^n} + \phi(\frac{1}{B_w})^{'}\right) S_w^n \right\}_i \tag{3.23}$$

$$C_{ow_i} = \left\{ -\frac{V_b}{\Delta t}\left(\frac{\phi}{B_o}\right) \right\}_i \tag{3.24}$$

$$C_{op_i} = \left\{ \frac{V_b}{\Delta t}\left(\frac{\phi^{'}}{B_o^n} + \phi(\frac{1}{B_o})^{'}\right)(1 - S_w)^n \right\}_i \tag{3.25}$$

Superscript n denotes the value at current timestep, and does not change during the iteration process until the iteration is completed. Superscript $'$ on porosity and FVF denotes derivative to pressure. In this case, rock compressibility is assumed zero and $\phi^{'}$ will be zero. However, oil and water compressibility are defined and FVF derivative to pressure can be obtained from compressibility.

The sink/source submatrix Q_i consists of:

$$Q_i = \begin{pmatrix} \frac{\partial q_{w,sc_i}}{\partial S_{w_i}} & \frac{\partial q_{w,sc_i}}{\partial P_{o_i}} \\ \frac{\partial q_{o,sc_i}}{\partial S_{w_i}} & \frac{\partial q_{o,sc_i}}{\partial P_{o_i}} \end{pmatrix} \tag{3.26}$$

In the case used for building this Python script, Q_i is zero for all i, except for $i = 0$ because of the injection well, and $i = N - 1$ because of the production well. Since the wells are BHP-controlled for this case, the partial derivatives are governed by Darcy's law for multiphase flow(Equation 2.13):

- Partial derivatives of q to P_o is controlled by change in pressure drop between the well and the boundary gridblock ($i = 0$ for injector well and $i = N-1$ for producer well).
- Partial derivatives of q to S_w is controlled by gradient of relative permeability in k_r vs S_w curves.

The Python script itself consists of various functions, each having a specific task that contributes to the fully-implicit method to solve multiphase flow equations:

1. `relPerm` calculates water and oil relative permeability based on S_w in a gridblock. The relative permeability curve is developed using Corey's method, using Corey exponent of 2.0 for both oil and water. The result is close to the plot in Figure 3.2.
2. `derivRelPerm` calculates gradient of water and oil relative permeability based on S_w in a gridblock.
3. `determineConnection` determine connecting gridblocks for every grid.
4. `darcyFlow` calculates flow rate according to Darcy's equation for multiphase flow (Equation 2.13).
5. `calculateTransmissibility` calculates water and oil transmissibilities from a specific gridblock i to the connecting gridblocks m.
6. `fillFlowVectorElement` fills each element of the flow vector $\vec{F}$.
7. `fillFlowMatrixDiagElement` fills each element of the flow matrix $[F]$ for the submatrices $F_{i,m}$ where $i = m$.
8. `fillFlowMatrixNondiagElement` fills each element of the flow matrix $[F]$ for the submatrices $F_{i,m}$ where $i \neq m$.
9. `fillUnknownVector` builds the unknown vector $\vec{X}$.
10. `fillResidualVector` builds the residual vector that consists of the flow vector $\vec{F}$, the accumulation vector $\vec{C}$, and the sink/source vector $\vec{Q}$. This function also fills the elements of $\vec{C}$ and $\vec{Q}$, while the elements of the more complicated $\vec{F}$ are filled by `fillFlowVectorElement`.
11. `fillJacobianMatrix` builds the Jacobian matrix that consists of the flow matrix $[F]$, the accumulation matrix $[C]$, and the sink/source matrix $[Q]$. This function also fills the elements of $[C]$ and $[Q]$, while the elements of the more complicated $[F]$ are filled by `fillFlowMatrixDiagElement` and `fillFlowMatrixNondiagElement`.

12. `NextIterDeltaUnknown` calculates the change in unknown vector, $(\vec{X}^{(v+1)} - \vec{X}^{(v)})$, in one iteration of Newton's method.

13. `NextTimestepUnknown` calculates the objective values/unknowns (P_o and S_w) for the next timestep, using a *while loop* that implements `NextIterDeltaUnknown` until $(\vec{X}^{(v+1)} - \vec{X}^{(v)})$ converges. The criteria for convergence is set to be either "error" is below 10^{-4} or number of iterations have exceeded 20. The "error" is defined as the square root of sum of $(S_w^{(v+1)} - S_w^{(v)})^2$ in every gridblock.

The setup before running the simulation includes setting the properties to build relative permeability curves, reservoir initialization (rock and fluid parameters), well initialization (well control parameters), and setting the size and amount of timesteps to be used. The simulation is then could be started, which will use all the functions defined above. The workflow of the simulation is illustrated in Figure 3.4.

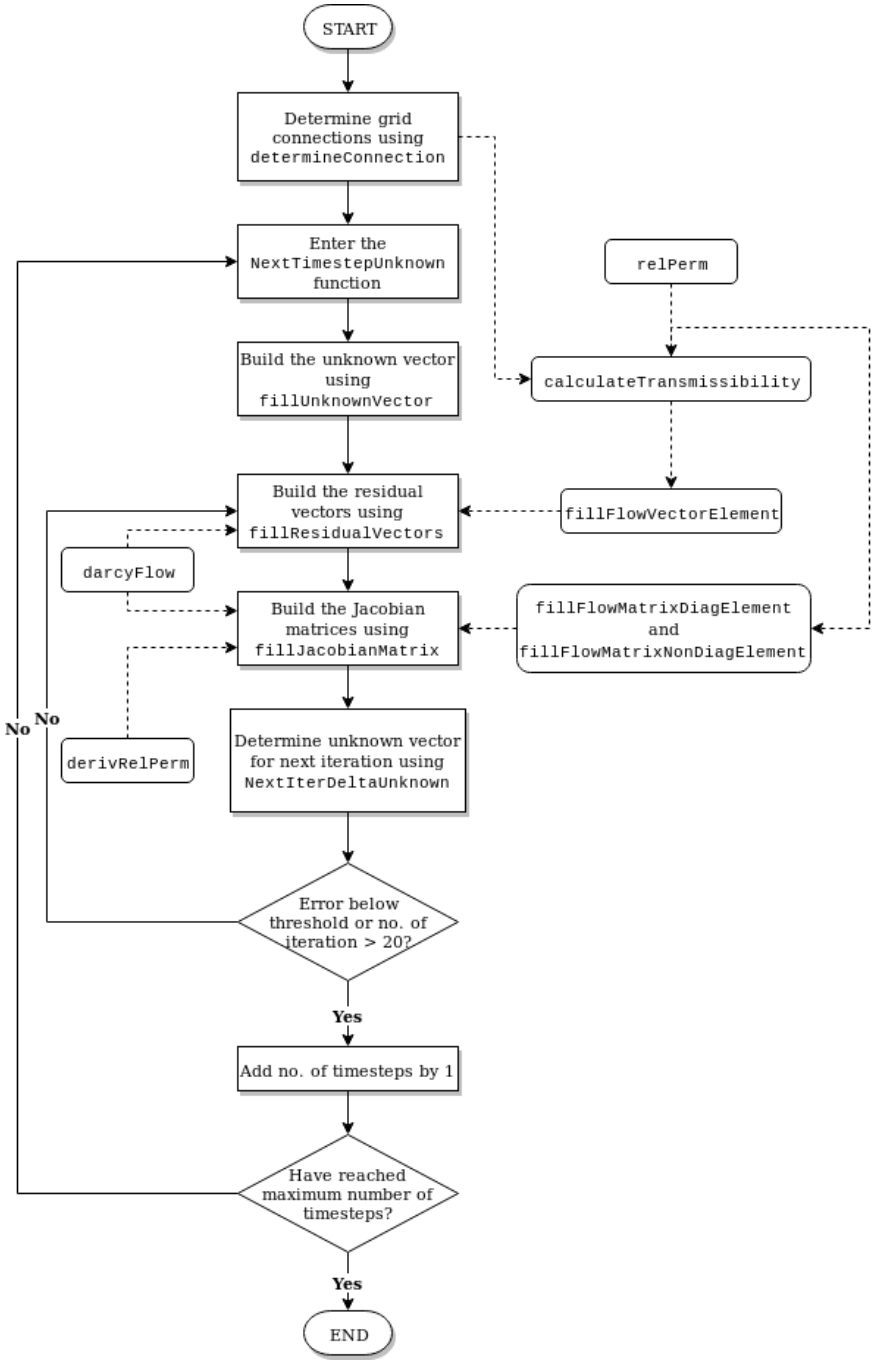


Figure 3.4: The flowchart detailing how the Python script works from the beginning until the end of a simulation. The main workflow follows the solid lines, while the dashed lines show how the supporting functions help the main workflow.

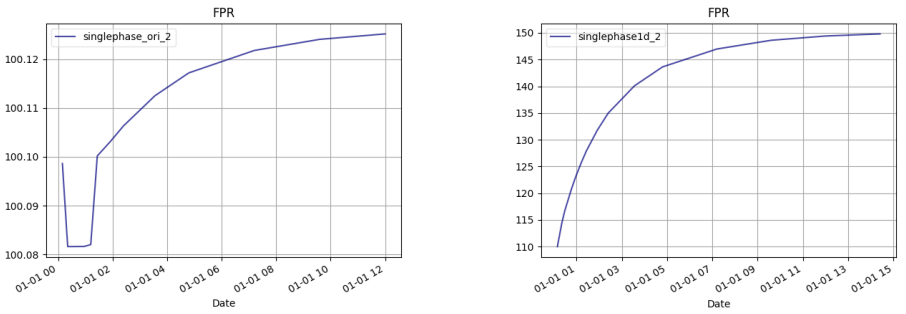
Chapter 4

Results and Discussion

In this chapter, results from testing and comparing cases in Flow and other reservoir simulators are presented. The results are then discussed to provide insights required in order to achieve the objectives of this study.

4.1 Convergence Issues Investigation

The first case to test is the SinglePhase_2 case. This is the case similar to the original case that got the pressure plot wrong in Flow 2018.10. Figure 4.1 shows the comparison of average field pressure (FPR) plot of the original case and SinglePhase_2 case. Observing



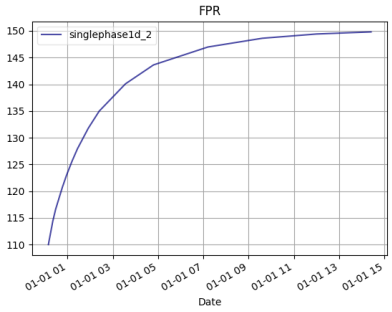
(a) Original case.

(b) SinglePhase_2 case.

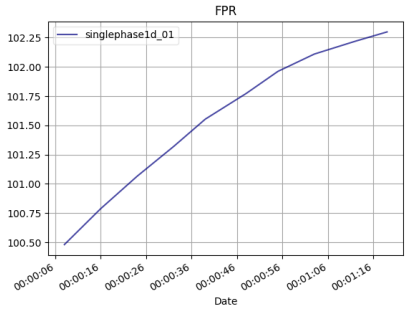
Figure 4.1: Comparison of average field pressure (FPR) of the original case tested using Flow 2018.10 and SinglePhase_2 case tested using Flow 2019.10.

Figure 4.1, it seems that the issue no longer persists in OPM Flow version 2019.10. Since the injection well is set to operate at 200 bar and the production well operates at 100 bar,

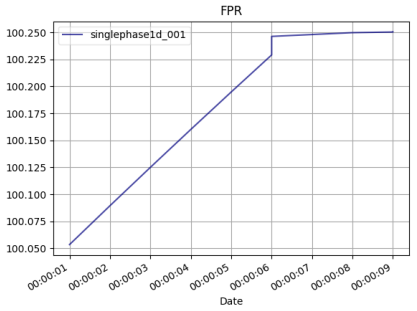
it is expected that the average field pressure would eventually goes to 150 bar as shown in Figure 4.1a, meanwhile there are clearly some issues in the original case that only allows the pressure to rise a bit above the initial value of 100 bar. This might mean that the convergence issue is solved by the newest version of OPM Flow, but in order to make sure other cases with more extreme grid block size are tested. Figure 4.2 shows the comparison of FPR plots in three single-phase cases, using different block sizes of 2 m (SinglePhase_2 case), 0.1 m (SinglePhase_01 case), and 0.01 m (SinglePhase_001 case).



(a) SinglePhase_2 case.



(b) SinglePhase_01 case.



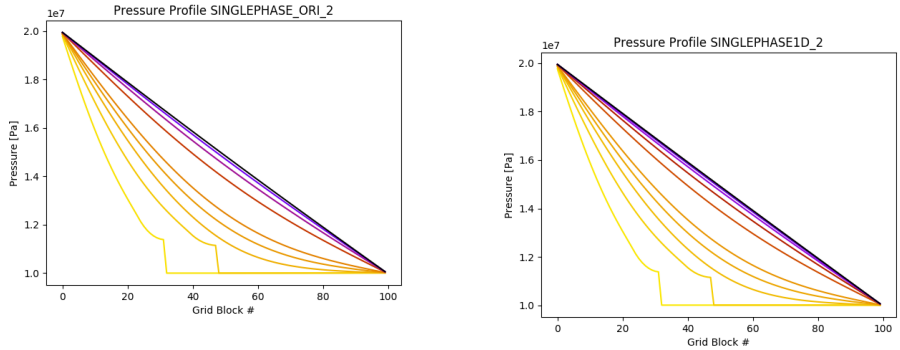
(c) SinglePhase_001 case.

Figure 4.2: Comparison of average field pressure (FPR) plot of the three single-phase cases tested using OPM Flow version 2019.10

All the plots in Figure 4.2 show nice field pressure buildup to reach the average pressure between the producer and the injector wells (the steady-state average pressure). The larger the timestep used, the smoother the buildup will be on the plot. For the cases with very small grid size such as SinglePhase_001, using the same timestep as SinglePhase_2 is not recommended because the buildup will not be seen as the pressure will seem to instantly reach the steady-state average pressure. Even after scaling the reservoir down to centimeters scale, the simulation works as expected and there are no noticeable errors

whatsoever.

After checking other plots of the original case, however, they tell a different story. Using Python to plot the pressure distribution for the original case and SinglePhase_2 case results in Figure 4.3. It shows that the pressures for the original case are already correct and very similar to the SinglePhase_2 case, even though the FPR plot presents an inaccurate result. If the plots on Figure 4.3 are stacked with each other, the difference is indistinguishable, as can be seen in Figure 4.4. The injection rate plot is also checked, and the comparison could be observed in Figure 4.5. Again, there are no differences in injection rate, which means that the simulation results using Flow 2018.10 is correct and there are no significant issues. This means that the error that occurs in plotting FPR is not due to wrong pressure determination or convergence issues in the simulator, but rather because of errors in reporting and/or incorrect formula used to calculate FPR.



(a) Original case.

(b) SinglePhase_2 case.

Figure 4.3: Comparison of pressure distribution plot in each gridblock of the original case tested using Flow 2018.10 and SinglePhase_2 case tested using Flow 2019.10.

Figures 4.6 and 4.7 gives complete comparison of the three single-phase cases on pressure distribution and water injection rate.
Even though the issue of false reporting of FPR has been fixed, Figure 4.6 shows that the numerical issues encountered in Flow 2018.10 still remain until now. As explained before in Section 2.5, the numerical errors in the early timesteps could be mitigated by lowering timestep. However, for the small-scale cases, lowering timestep to a few seconds still produce significant numerical errors. Currently, there is a limit of minimum timestep size in OPM Flow, which is one second. It is hard-coded in the source code and the reason the limit is one second is because OPM Flow uses Coordinated Universal Time (UTC) format for implementing time into the software, and the smallest unit in UTC standards is one second. OPM Flow could change the time format to allow for even smaller timesteps, but most likely the source code would need to be thoroughly modified and revised.
For the pressure distribution plots, it can be seen that the numerical errors in the beginning of the simulation is more amplified for smaller-scale cases. However, after some initial timesteps the pressure plot begins to look smooth and produces a reasonable result. For

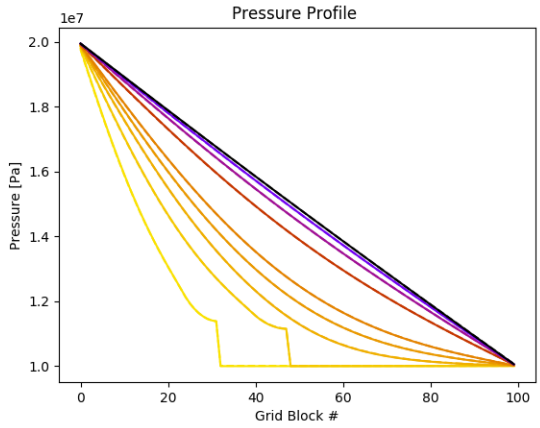
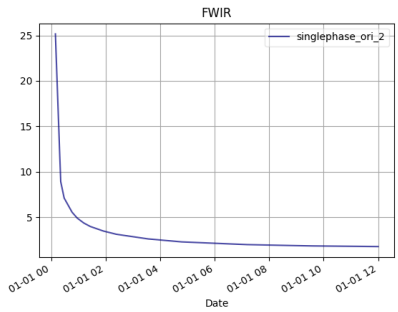
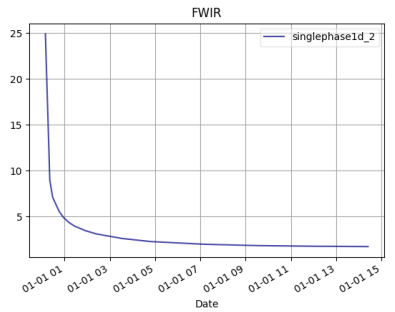


Figure 4.4: Pressure distribution plots of the original case and the new case stacked with each other. The original case is the dashed lines, and the new case is the solid lines. The difference between them is indistinguishable.



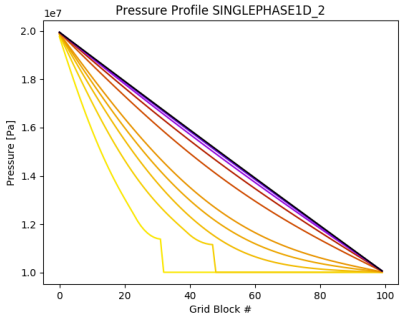
(a) Original case.



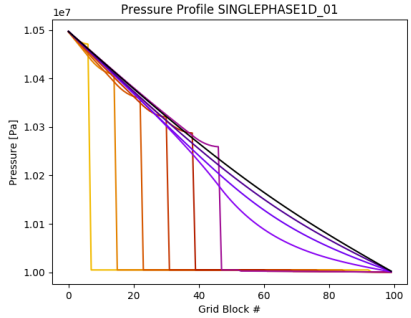
(b) SinglePhase_2 case.

Figure 4.5: Comparison of water injection rate plot in each gridblock of the original case tested using Flow 2018.10 and SinglePhase_2 case tested using Flow 2019.10.

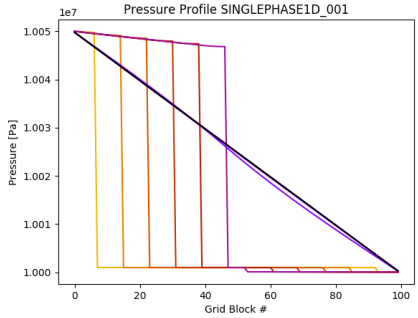
the water injection plots, it is directly connected to the pressure distribution plots. The plot for SinglePhase_2 case is smooth and shows the decreasing injection rate over time. The plot for SinglePhase_01 case also shows gradual decrease, but there was a bump in injection rate in the middle of the simulation. This is caused by the pressure distribution that are being freed from numerical error as seen in Figure 4.6a. The same also occurs for SinglePhase_001 case, where water injection rate is actually small at first, but suddenly jumps to a higher value in the middle of the simulation. From Figure 4.6b, it is seen that during the timesteps in which numerical issues take place, the pressure gradient near the injection well is smaller than when steady-state is acheived. The smaller pressure gradient results in smaller injection rate. Even though there are numerical problems in Flow during



(a) SinglePhase_2 case.



(b) SinglePhase_01 case.



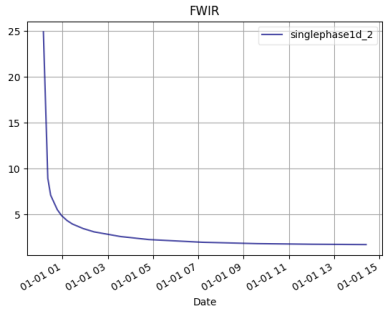
(c) SinglePhase_001 case.

Figure 4.6: Comparison of pressure distribution plot for the three single-phase cases tested using Flow version 2019.10. Early timesteps are colored yellow to red, later timesteps are colored blue to black.

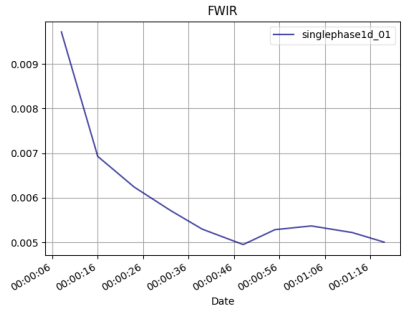
early simulation time, it is only amplified in very small scales and would not be a huge deal in practical applications, where grid size is usually large and simulation time is very long.

Results from single-phase cases show that no convergence issues whatsoever are detected in the newest OPM Flow versions. To further prove this claim, the two-phase cases are tested. For these cases, however, a longer timestep size is needed because injecting water to a water-oil system will drive water out at a smaller rate. The water saturation distribution plots are also created for the two-phase cases, to see how the saturation front moves over time. Figures 4.8, 4.9, and 4.10 respectively display the plots of average field pressure, pressure distribution, and water saturation distribution over time for all three two-phase cases tested.

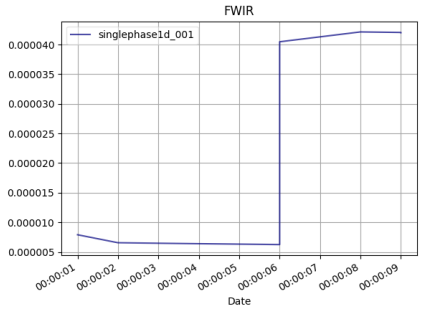
The FPR plots show similar trend for all cases, the average field pressures goes from



(a) SinglePhase_2 case.



(b) SinglePhase_01 case.

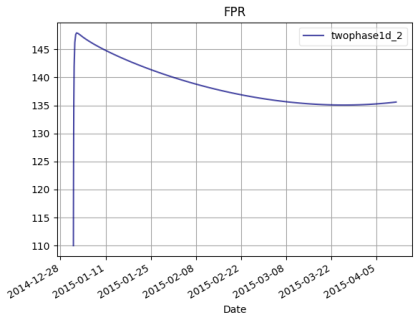


(c) SinglePhase_001 case.

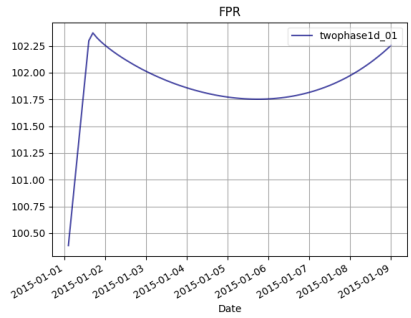
Figure 4.7: Comparison of water injection rate plot for the three single-phase cases tested using Flow version 2019.10.

initial pressure to middle pressure between producer and injector, then it goes down a bit until it reaches a minimum point and the average pressure builds again. This trend can be attributed to the pressure distribution plots. It can be seen that the pressures in each grid-block are not constant like steady-state flow throughout the simulation, due to the different transmissibilities for oil and water that arises from difference in relative permeability curve of k_{rw} and k_{ro}. There are some numerical issues as expected in TwoPhase_01 case and TwoPhase_001 case in the early timesteps, but all three cases present no convergence problems from the pressure plots, improving confidence in the current version of OPM Flow.

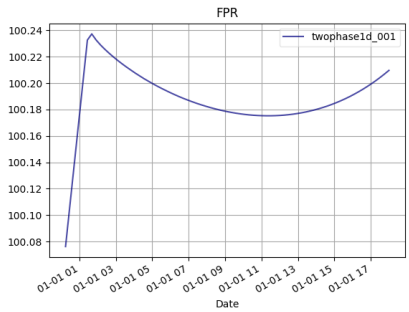
What could be interesting to observe in a two-phase system is the water saturation plot. Figure 4.10 shows that there are no issues in the calculation of water saturation, even at a very small scale model with timestep of one second. The saturation curve is smooth, and the movement of the saturation front is just as expected. The results from the two-phase cases provide further proof that there are no convergence issues in the recent version of OPM Flow that would significantly impact the quality of simulation results.



(a) TwoPhase_2 case.



(b) TwoPhase_01 case.



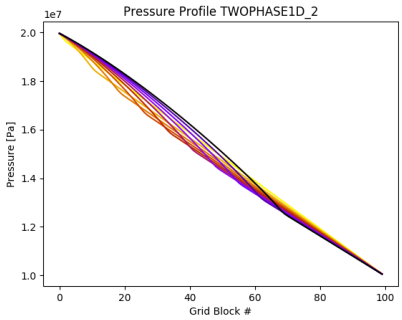
(c) TwoPhase_001 case.

Figure 4.8: Comparison of average field pressure plot for the three two-phase cases tested using Flow version 2019.10.

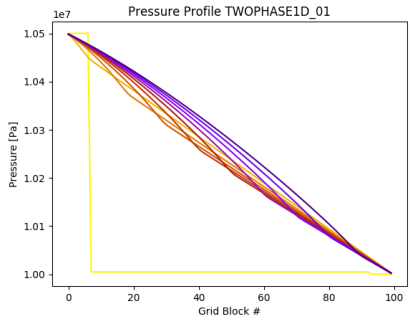
4.2 WTEST Keyword for Testing Wells

The results presented in Section 4.1 seems to imply that OPM Flow version 2019.10 runs perfectly without any issues and errors. This is, however, not quite true. The cases that was run in Section 4.1 had to undergo some adjustments that allow them to display accurate results. This is because initially there were some problems encountered with the production well that continously fails to operate.

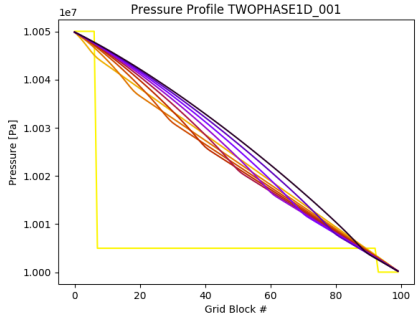
In Tables 3.1 and 3.1 the initial pressure for all cases is defined to be 100 bar, the same value as the producer well BHP. However, when the cases are run it is become apparent that something wrong has occured. The pressure in the reservoir never goes to steady-state but instead climbs up until all the grid block pressures are equal to injection well pressure. It turns out that the producer well is immediately shut at the beginning of the simulation because the grid pressure cannot be equal to minimum BHP constraint. The only solution to mitigate this is to set the initial pressures across the reservoir a little bit above BHP constraint. After setting initial pressure to 100.1 bar, the producer well does not shut down and the simulation runs as expected.



(a) TwoPhase_2 case.



(b) TwoPhase_01 case.



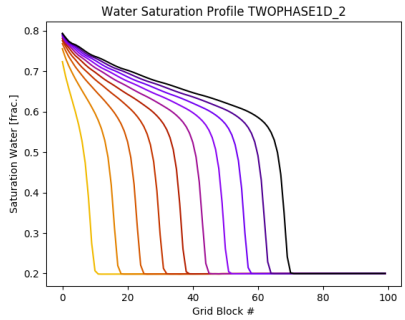
(c) TwoPhase_001 case.

Figure 4.9: Comparison of pressure distribution plot for the three two-phase cases tested using Flow version 2019.10. Early timesteps are colored yellow to red, later timesteps are colored blue to black.

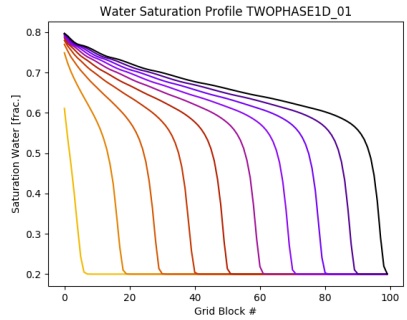
Figure 4.11 shows the comparison between the same SinglePhase_2 case, but one is run with initial reservoir pressure of 100 bar, and another with 100.1 bar. Figure 4.12 shows the comparison of the producer BHP plot for the two scenarios. It can be observed from both figures that the problem could be solved by simply changing the producer BHP a bit above constraint.

In this regard, it could be said that Flow 2019.10 does a worse job than Flow 2018.10 when implementing the WCONPROD keyword used to impose well control to production wells. The original case was run with initial pressure of 100 bar, but the pressure plot shows no shutting of the producer (one can compare Figure 4.11 with Figure 4.3 to prove this).

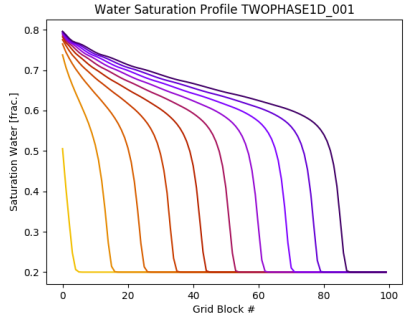
This problem then become a new center of attention in this study other than the convergence issues being tested. After searching for solutions for this, it is found that there exists a keyword in the OPM Flow Manual (Baxendale) that has the function to test wells for physical and/or economical limits called **WTEST**. This keyword should be inputted in the



(a) TwoPhase_2 case.



(b) TwoPhase_01 case.



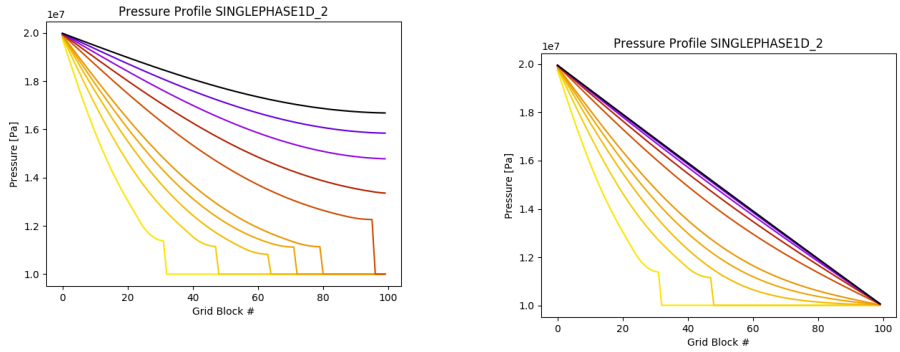
(c) TwoPhase_001 case.

Figure 4.10: Comparison of water saturation distribution plot for the three two-phase cases tested using Flow version 2019.10. Early timesteps are colored yellow to red, later timesteps are colored blue to black.

SCHEDULE section of the .DATA file and contains pointers that specify which well(s) to test, what limits are being tested (physical and/or economical), how often they are tested, and maximum number that a well is allowed to be tested. An example of implementation of WTEST keyword and its location within the RUNSPEC section in the input file can be seen in the following code snippet:

```
    WTEST
-- Item # 1     2   3 4
        'INJW' 0.02 P 0 /
        'PROD' 0.02 P 0 /
/
-- #1: Well to test
-- #2: How often it is tested (in days)
-- #3: What to test: P = physical limits ; E = economical limits
-- #4: Maximum number of testing (0 = infinite tests allowed)
```

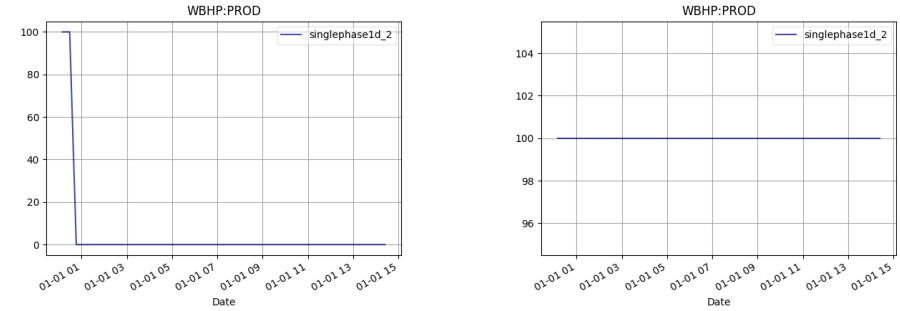
The implementation of the WTEST keyword was successful in eliminating the shutting producer problem. The cases can now be run at initial pressure of 100 bar and yield steady-state pressure results. The plots presented in Section 4.1 was run at the intended



(a) Initial pressure = 100 bar.

(b) Initial pressure = 100.1 bar.

Figure 4.11: Comparison of pressure distribution plot for SinglePhase_2 case tested with different initial pressures. Early timesteps are colored yellow to red, later timesteps are colored blue to black.



(a) Initial pressure = 100 bar.

(b) Initial pressure = 100.1 bar.

Figure 4.12: Comparison of producer BHP plot for SinglePhase_2 case tested with different initial pressures.

producer BHP of 100 bar and does not output inaccurate pressure plots such as Figure 4.11. For future researches using Flow 2019.10, the WTEST keyword is highly important to take into consideration, because setting a well BHP to be equal to grid pressure would produce inaccurate results unless WTEST is inputted to the `.DATA` file.

4.3 Oil Injection Phenomenon in Waterflooding Scenario

As explained in Section 1.1, the case from last semester's Reservoir Simulation course project need to be tested again using the recent OPM Flow version to see if the OPM team has fixed the oil injection issue. In this segment of the study, comparison plots are presented between the old version of Flow (2018,10) and the more recent versions (2019.04 and 2019.10) using the best case obtained from the project course results. It turns out that

results from both 2019.04 and 2019.10 are identical and there are no differences visible between them in the plots. Figure 4.13 shows the field oil injection rate (FOIR) comparison plot.

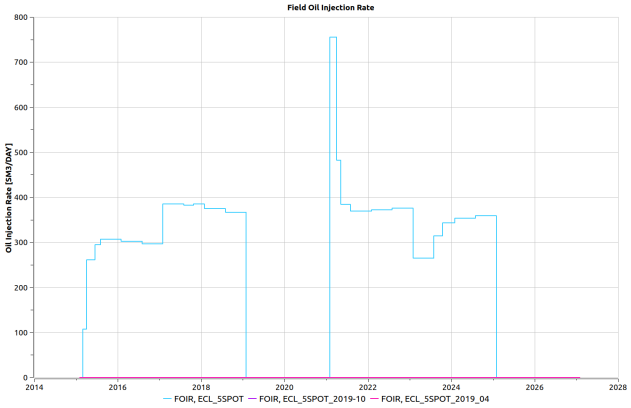
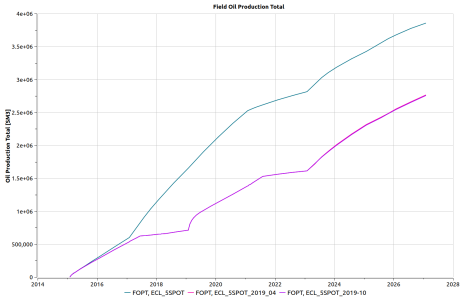


Figure 4.13: Field Oil Injection Rate (FOIR) comparison plot between Flow 2018.10 (green) and newer Flow versions (red).

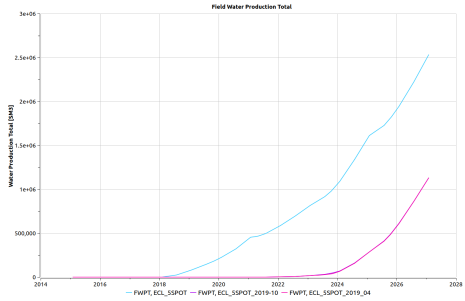
As shown in Figure 4.13, there are no more oil injection phenomenon in the waterflooding case. This means that Flow 2019.04 and 2019.10 have fixed the problem and if the cases are to be used again for next year's course project, a more accurate result can be expected. To see how much this will impact the project result, comparison of total field production (oil and water) and total field water injection are displayed in Figure 4.14.

The main objective of the project course was to find optimum BHP constraints for all wells that would maximize NPV. The NPV itself is defined to be dependent on the three quantities plotted in Figure 4.14. Increasing oil production would increase NPV, while increasing water production and water injection reduce NPV. The updated Flow versions then would need recalculation of NPV to compare how the correct result stack up against the inaccurate version from last semester. Table 4.1 presents the NPV comparison. Before taking discount factor into consideration, it can be seen that the NPV obtained from last semester's results using Flow 2018.10 is highly overestimated by over 338 million dollars. The decrease in water production and injection cannot make up the NPV reduction from the decrease of oil production, since oil price is much higher than water treatment and injection cost per unit volume.

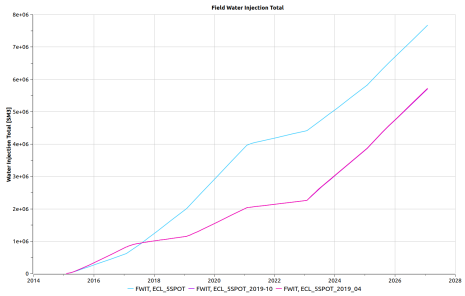
The cause of the oil injection phenomenon was also found during this study. The wells that injected oil were the production wells (P2 and P3), and they did it to prevent shutdown. If there were no oil injection, P2 and P3 would close because the well BHP was below the constraint set. Figure 4.15 presents the comparison plot of P2 and P3 BHP. Notice that in the old Flow version, P2 and P3 never shuts down.



(a) FOPT plot.

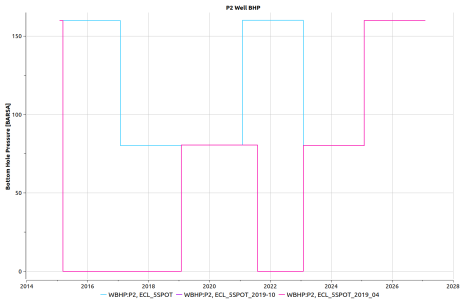


(b) FWPT plot.

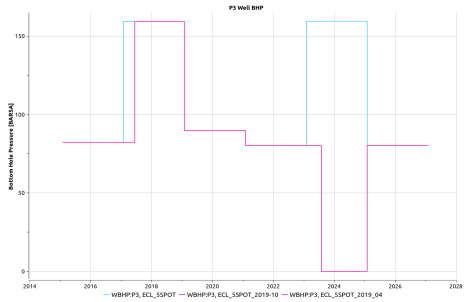


(c) FWIT plot.

Figure 4.14: Field Oil Production Total (FOPT), Field Water Production Total (FWPT), and Field Water Injection Total (FWIT) comparison plot between Flow 2018.10 (green) and newer Flow versions (red).



(a) P2 BHP plot.



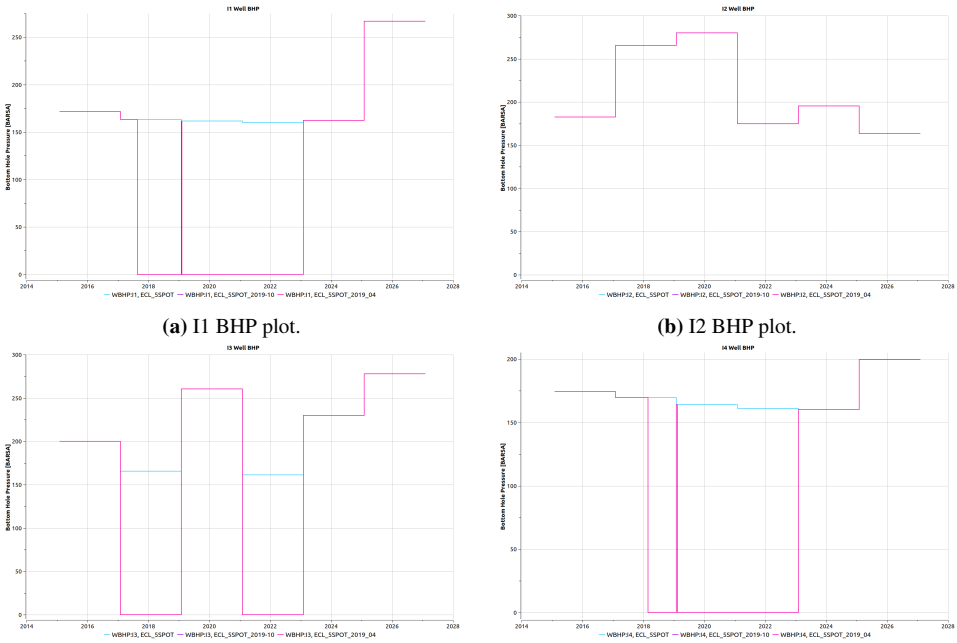
(b) P3 BHP plot.

Figure 4.15: Bottom hole pressure comparison plot of producer wells P2 and P3 between Flow 2018.10 (green) and newer Flow versions (red).

The closing of producer wells would impact the injection wells, because the pressure in reservoir would build up when nothing is produced but water is kept injected. Eventually, the pressure would violate maximum BHP set for injection wells, and some of them would close as well. Figure 4.16 presents the comparison plot of all injection wells. While the BHP plot for I2 is unchanged, in other wells some shutdowns are observed.

Cost Function	Value [$/bbl]	Flow 2018.10 [MMbbl]	Flow 2019.10 [MMbbl]	Difference [MMbbl]	Revenue diff. [MM$]
Oil price	60	24.266	17.335	-6.931	-415.86
Production water	-6	15.932	7.107	-8.825	52.95
Injection water	-2	48.224	35.946	-12.278	24.556
Total NPV difference before discount factor [MM$]					***-338.354***

Table 4.1: NPV comparison table between the results obtained using Flow 2018.10 and Flow 2019.10.



(a) I1 BHP plot.

(b) I2 BHP plot.

(c) I3 BHP plot.

(d) I4 BHP plot.

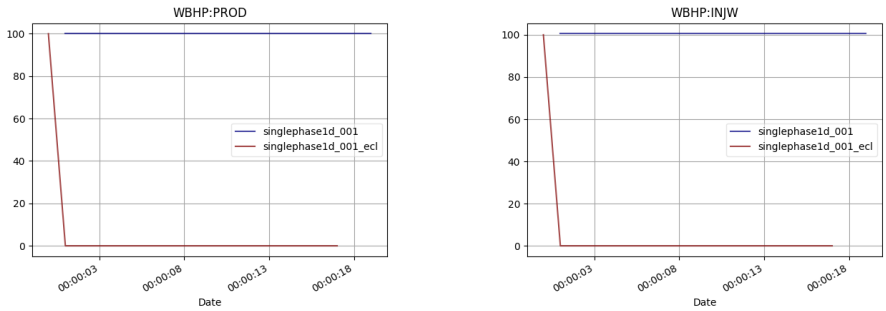
Figure 4.16: Bottom hole pressure comparison plot of injector wells between Flow 2018.10 (green) and newer Flow versions (red).

4.4 Comparison and Validation of OPM Flow with Other Simulation Tools

4.4.1 Comparison of results using OPM Flow and Schlumberger ECLIPSE

The same sets of cases used in Section 4.1 are tested again using Schlumberger ECLIPSE reservoir simulator for benchmarking purposes. From all six cases (three single-phase cases and three two-phase cases), one case (SinglePhase_001) did not run successfully using ECLIPSE. Even after using WTEST keyword, the production and injection well keep

closing after numerous attempts to revive them. Figure 4.17 shows the producer well BHP and injector well BHP comparison for SinglePhase_001 case using Flow and ECLIPSE. However, the same thing does not happen to TwoPhase_001 case, which uses 0.01 m grid block size as well. This problem is then not caused by the physical size of the model, and looking at the `.DATA` file reveals that the wells shut down due to some convergence error that produces negative IPR and negative injection rate. This might imply that some convergence issues could also occur in ECLIPSE for a small-scale case like this.



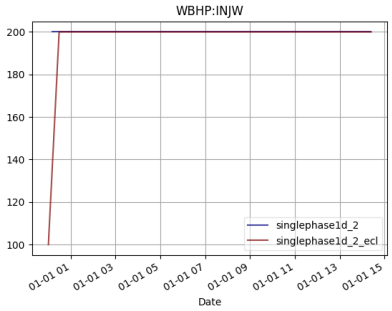
(a) Producer well BHP.

(b) Injector well BHP.

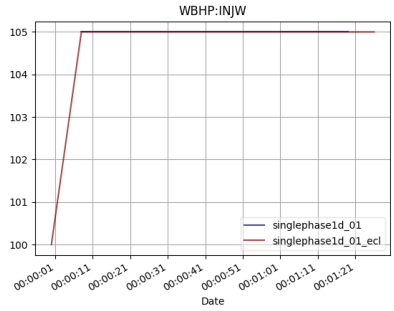
Figure 4.17: Well BHP comparison plots for SinglePhase_001 case, tested using Flow 2019.10 and ECLIPSE.

Moving on to the cases that did run in ECLIPSE, first let us observe the injection well BHP plot for the other two single-phase cases. Figure 4.18 presents the comparison plots. The key takeaway from here is that it seems ECLIPSE did not open the injection well immediately at the first timestep, and uses that timestep to equilibriate the reservoir conditions. In fact, all the cases that run successfully using ECLIPSE did not need to use WTEST keyword to work. Letting the reservoir equilibriate itself first might be what caused this. OPM Flow could implement this in their future updates to prevent issues in Section 4.2 from happening.

Next, let us see how the takeaway from Figure 4.18 would impact other results in the simulation. Figures 4.19 and 4.20 respectively show the comparison plots of water injection rate and field average pressure for SinglePhase_2 case and SinglePhase_01 case. The fact that injection well is not opened at the first timestep shows its effect on Figure 4.19. At the first timestep, injection rate is zero for cases tested using ECLIPSE. Eventually, the injection rate will be identical to cases tested with Flow. What is interesting to discuss is the FPR plot for SinglePhase_001 case in Figure 4.20. The average field pressure value is close to zero at all times. This is interesting because all the other results for SinglePhase_001 case are reasonable and similar to Flow's results. This might be a same problem as encountered in Flow 2018.10, which is error in reporting the FPR value (see discussion of Figure 4.1 and 4.2).

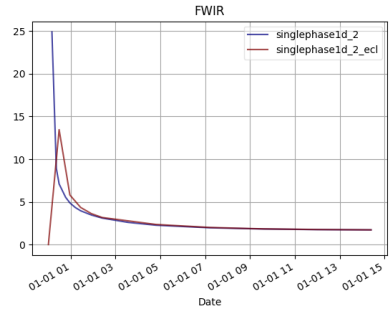


(a) SinglePhase_2 case.

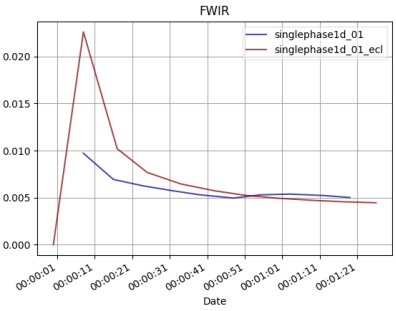


(b) SinglePhase_01 case.

Figure 4.18: Injection well BHP comparison plots for SinglePhase_2 and SinglePhase_01 case, tested using Flow 2019.10 and ECLIPSE.



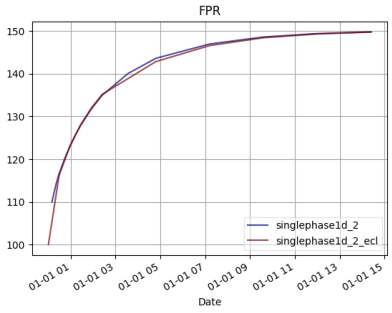
(a) SinglePhase_2 case.



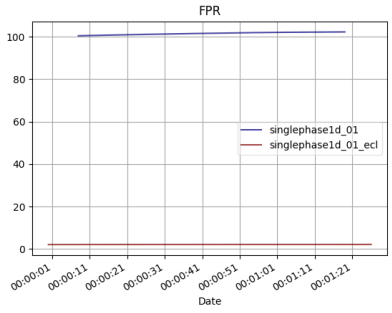
(b) SinglePhase_01 case.

Figure 4.19: Water injection rate comparison plots for SinglePhase_2 and SinglePhase_01 case, tested using Flow 2019.10 and ECLIPSE.

Following the discussion of the single-phase cases, the two-phase cases are next to observe. Figures 4.21 and 4.22 are the comparison plots of water injection rate and field average pressure for the two-phase cases. Notice that TwoPhase_001 case produce a nice and comparable result here, even though its model size is the same as the SinglePhase_001 case. For the water injection rate plots in Figure 4.22, it can be seen that for the small-scale cases both Flow and ECLIPSE starts injecting water a bit later after the start of simulation, and ECLIPSE starts injecting first. For Flow, this is not caused by the equilibriating technique like in ECLIPSE, but because of early timesteps numerical errors. The injection well is actually opened from the beginning and did not shut down, but no water is injected. Revisiting Figure 4.9, it is seen that for the early timesteps, pressure near injection well in TwoPhase_01 and TwoPhase_001 case is the same as injection well BHP (see light yellow line on the plots).

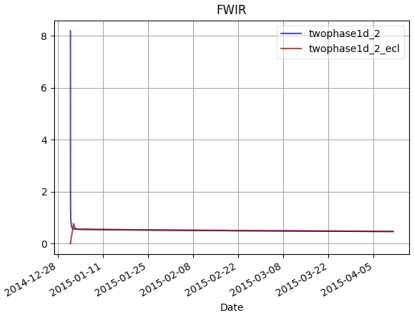


(a) SinglePhase_2 case.

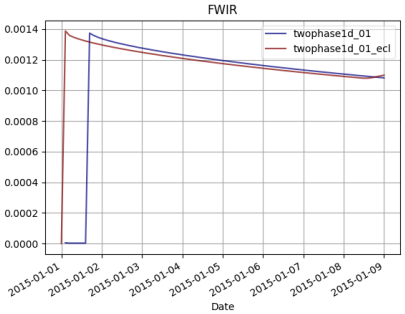


(b) SinglePhase_01 case.

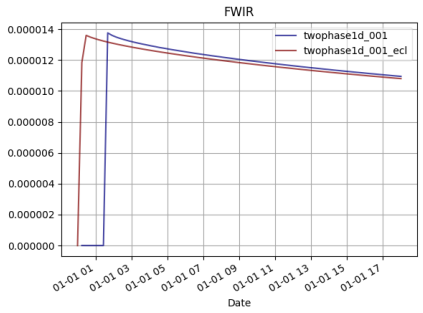
Figure 4.20: Field average pressure comparison plots for SinglePhase_2 and SinglePhase_01 case, tested using Flow 2019.10 and ECLIPSE.



(a) TwoPhase_2 case.



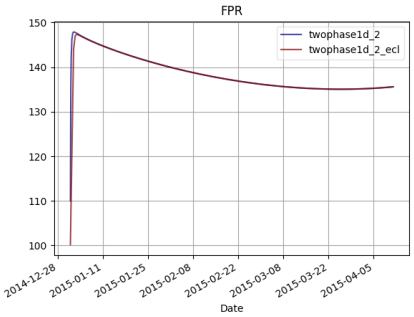
(b) TwoPhase_01 case.



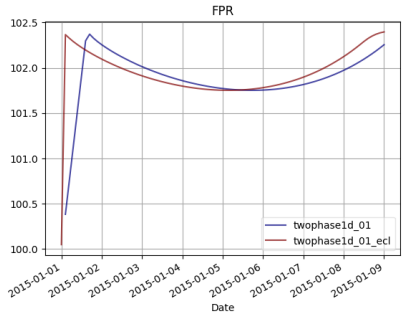
(c) TwoPhase_001 case.

Figure 4.21: Water injection rate comparison plots for two-phase cases, tested using Flow 2019.10 and ECLIPSE.

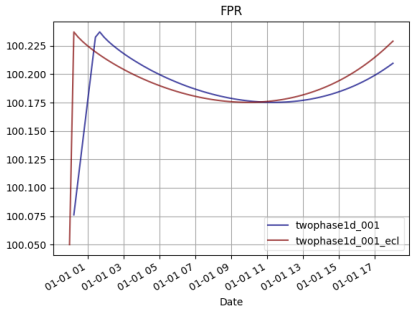
The difference in water injection rate will directly cause difference in FPR plot, as seen



(a) TwoPhase_2 case.



(b) TwoPhase_01 case.



(c) TwoPhase_001 case.

Figure 4.22: Field average pressure comparison plots for two-phase cases, tested using Flow 2019.10 and ECLIPSE.

in Figure 4.22. For the TwoPhase_2 case, the plot is identical. However, for the other cases the plot is a bit different because ECLIPSE starts injecting water first. The FPR plot for Flow could be interpreted as a ”delayed” FPR plot of ECLIPSE. One should realize by now that the ECLIPSE FPR plot for all two-phase cases are comparable to Flow FPR plot, unlike the single-phase cases. Using grid size of 0.1 m and 0.01 poses no problem in calculating and reporting FPR in two-phase cases.

Comparing pressure distribution results from Flow and ECLIPSE reveal something that could give a solid solution to the numerical problems found in Flow. Figure 4.23 presents the pressure distribution comparison plot. The pressure curves in ECLIPSE is smooth from the beginning, even though the timestep size used is exactly the same as Flow. This means that the numerical issues could be handled by implementing the equilibriation approach from ECLIPSE, and there is no need to change the time format in Flow's source code because timestep size does not need to be very small to produce smooth pressure curves.

All in all, ECLIPSE reservoir simulator seems to work best for cases with multiphase flow, which is expected since it is advertised as a multiphase reservoir simulator. For

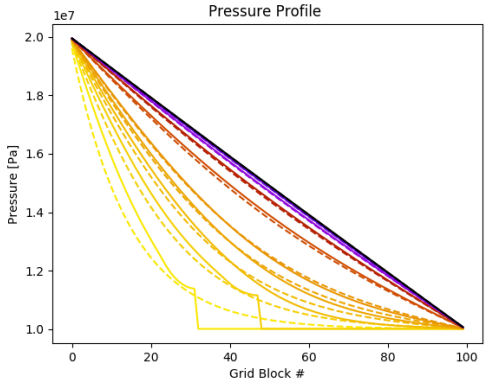


Figure 4.23: Pressure distribution plot comparison between results of Flow 2019.10 (solid lines) and ECLIPSE (dashed lines).

single-phase cases, results from OPM Flow could be more reliable, especially in smaller-scale cases like SinglePhase_01 and SinglePhase_001 cases. Errors in reporting FPR value and some convergence issues that make wells shut down continuously are found in ECLIPSE when working with single-phase cases. However, this might not be a big issue because single-phase flow and small-scale grid sizes are not commonly used in routine reservoir engineering. One thing that OPM Flow could take from this benchmarking is to implement a same equilibriating technique as found in ECLIPSE, in which the first timestep is only used to let the reservoir stay still and no wells are opened. This could solve the well shutdown problems in OPM Flow that for now can only be mitigated by utilizing the WTEST keyword, and also could solve the numerical issues at the early timestep simulations in OPM Flow that for now can only be minimized by reducing timestep size.

4.4.2 Validation of OPM Flow using Python-based fully-implicit simulator

For the purpose of validating the results of OPM Flow, a Python script for simulating reservoir using fully-implicit method is developed. It is hoped that this Python script would be able to function as a tool for anyone developing a full-scale reservoir simulator to test out and benchmark their results. The Python script is available to see in Appendix B.

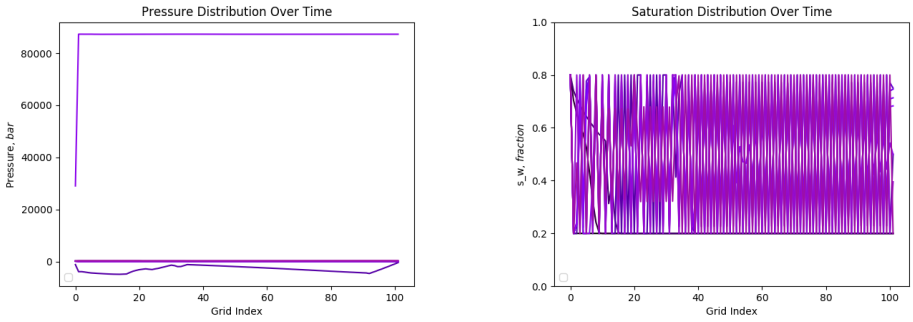
The Python fully-implicit simulator is built to solve the same case as TwoPhase_2 case, using the exact same values of the simulation parameters as stated in the case's `.DATA` file. Some key differences compared to running the case using OPM Flow are:

- For the relative permeability curve, approximation using Corey's relative permeability curve model is used. The Corey's exponent is 2.0 for both oil and water relative permeability curve.
- Implementation of the Newton's iteration to solve the matrix form equation uses

while loop in which the sum of squares of the gridblock saturations are treated as "error" of the iteration. The iteration will stop once this "error" goes below the threshold of 10^{-4}, or if the number of iterations has exceeded 20.

- The various partial derivatives used to build Jacobian Matrix are all calculated analytically and hardcoded into the script. This is different than OPM Flow that uses Automatic Differentiation (AD) for the same process.
- There are no "parallelization" such as in OPM Flow, that allows process to be distributed to different processors on a computer. This will have an impact on runtime.

Timestep of 1.0 day is initially used for running the Python code, to make it consistent with the timestep used in TwoPhase_2 case. This, however, resulted in error as shown in Figure 4.24. From the S_w distribution plot it can be observed that the simulation was working for the first several timesteps, until the solution breaks down and pressure shoots up to very large values. It might be that the relatively simple fully-implicit implementation in the script could not handle large timesteps. To mitigate this, the simulation is run again using timestep of 0.5 days. The results are presented in Figure 4.25. This time, the script produces nice and smooth curve for both pressure and S_w distribution. For future studies, the script could be modified to include variable timestep, such that small timesteps are used first, and gradually become larger as the simulation runs.



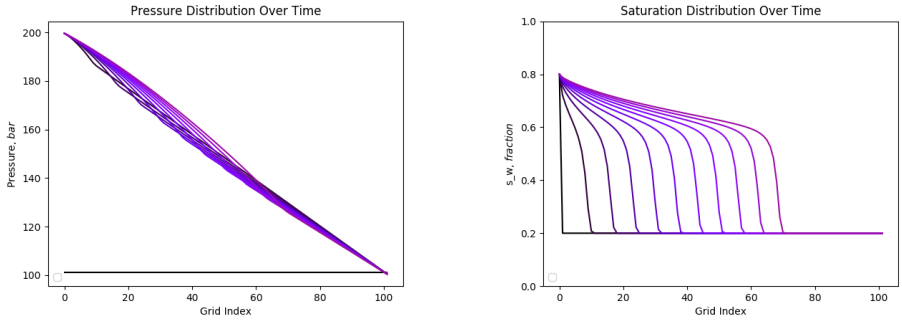
(a) Pressure distribution plot.

(b) Water saturation distribution plot.

Figure 4.24: Pressure and water saturation distribution plot from Python fully-implicit script, using timestep = 1.0 day.

After the successful development of the Python script, the results are compared to those of OPM Flow and ECLIPSE. Figure 4.26 shows the comparison plot for both pressure and S_w distribution throughout the model. Results from Python are drawn by solid lines, results from OPM Flow are the dotted lines, and results from ECLIPSE are the dashed lines. At a glance, the results look identical, which means even though the Python script is much simpler than the source code of Flow and ECLIPSE, the level of accuracy is comparable.

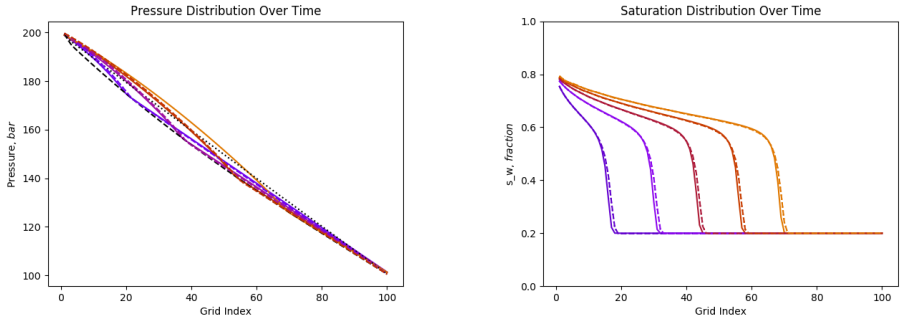
More insight could be gained by zooming on the plots on Figure 4.26. First, let us zoom



(a) Pressure distribution plot.

(b) Water saturation distribution plot.

Figure 4.25: Pressure and water saturation distribution plot from Python fully-implicit script, using timestep = 0.5 days.



(a) Pressure distribution comparison plot.

(b) Water saturation distribution comparison plot.

Figure 4.26: Comparison of pressure and water saturation distribution plot. Results from Python are solid lines, results from Flow are dotted lines, and results from ECLIPSE are dashed lines.

on the S_w distribution plot to the grid blocks near injection well, displayed here as Figure 4.28. The saturation curve for Flow and ECLIPSE could be seen to oscillates around the Python saturation curve, and the effect increases over time. This might mean saturation calculation is more accurate and stable on the Python script, and could come from the fact that Flow and ECLIPSE use tables to interpolate parameters such as relative permeability, while the Python script uses a smooth curve to calculate the parameters. There is a room for improvement here for Flow to implement a better interpolation or trendline modeling for its input tables.

Taking the comparison further, the S_w distribution plot can also be compared with the Buckley-Leverett model that should present a more ideal saturation distribution. Figure 4.27 compares results from Python, Flow, and ECLIPSE to Buckley-Leverett model. The figure shows that all the fully-implicit model smears out the saturation front, instead of a

sharp front in the Buckley-Leverett model. As Berg (2019) discussed in his Lecture Notes, the saturation front could be sharpened by using smaller timesteps, but by doing that would also make the saturation curves oscillates even more.

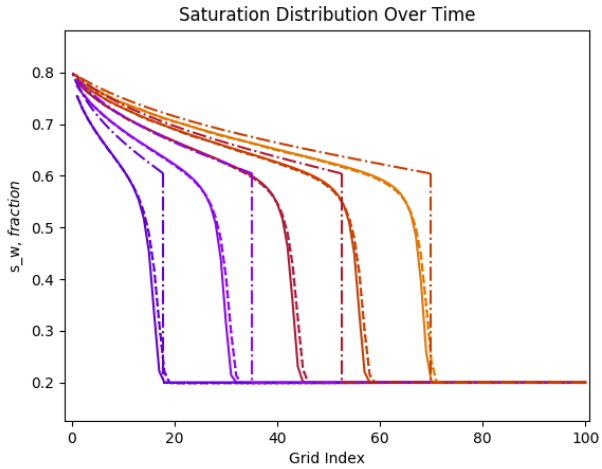


Figure 4.27: Comparison of water saturation distribution plots. Results from Python are solid lines, results from Flow are dotted lines, results from ECLIPSE are dashed lines, and Buckley-Leverett model are the dashed-dotted lines.

Next, let us zoom on the pressure distribution plot, both on the area near injection well and near production well. Figure 4.29 presents the zoomed-in plots. There is a big difference of pressures at early timesteps (colored black) between Flow (dotted lines) and ECLIPSE (dashed lines). This can be attributed to the numerical issues that occur in Flow shortly after simulation starts, but the same issue does not occur in ECLIPSE. At later timesteps, the grid pressures for Flow and ECLIPSE are identical. The pressure result from Python is a bit different compared to other simulators. While pressure in Flow and ECLIPSE goes from 200 bar (injector BHP) to 100 bar (producer BHP), pressure in Python goes from a bit under 200 bar to a bit above 100 bar. This difference comes from how the wells are represented in the model. In Flow and ECLIPSE the wells are located *on* the left and right gridblock, while in Python the wells are the left and right boundary and are not located in any gridblocks. This could be something to improve on the Python script in future studies.

Fully-implicit simulation implementation using Python could be done in a much simpler way than building multiple modules like what OPM Flow did, but there are still some drawbacks that require fixing if the script is to be used by a larger audience:

- A relatively large timesteps would cause huge convergence error, it is safer to use smaller timesteps or varying the timesteps from small at first and gradually become larger.

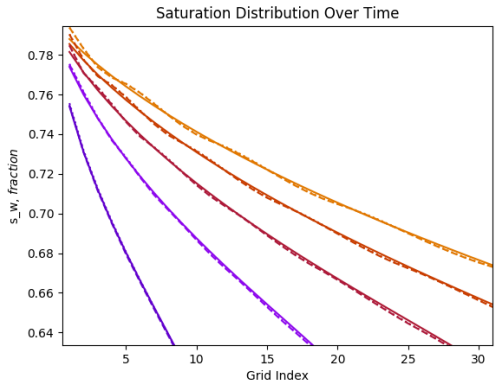
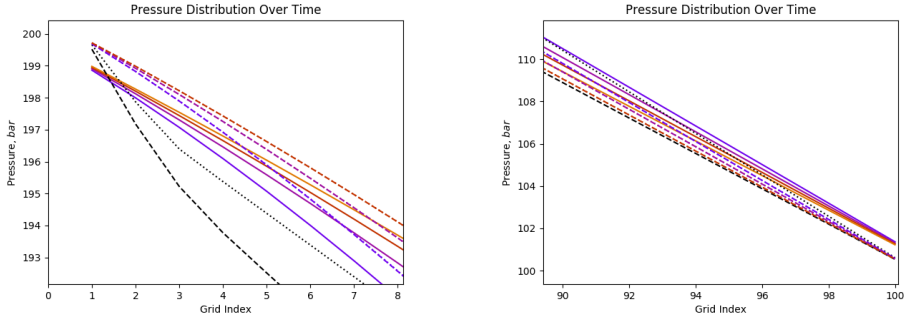


Figure 4.28: Zoomed-in version of water saturation distribution plot in Figure 4.26. Results from Python are solid lines, results from Flow are dotted lines, and results from ECLIPSE are dashed lines.



(a) Pressure distribution near injection well.

(b) Water saturation distribution near production well.

Figure 4.29: Zoomed-in version of pressure distribution plot in Figure 4.26. Results from Python are solid lines, results from Flow are dotted lines, and results from ECLIPSE are dashed lines.

- Simulation runtime is much slower than Flow and ECLIPSE. While the commercial simulators only take a few seconds to finish, the Python script need more than 6 minutes. This could be improved by finding other methods that allow faster convergence in the Newton's iteration, such as Conjugate Gradient (CG) method.
- Well representation could be changed to stay consistent with Flow and ECLIPSE. Instead of depicting a well as a constant-flow or constant-BHP boundary, a well could be represented as an element in a gridblock.

Chapter 5

Conclusions

This study has been done based on the objectives stated in Section 1.2 to obtain results presented and discussed in Chapter 4. From the discussion, conclusions of this study can be taken:

1. The recent versions of OPM Flow investigated in this study (version 2019.04 and 2019.10) have mostly fixed the known problems found previously, while also introducing a new problem.
    - The convergence issue that causes wrong pressure value reported for scaled down models are not found anymore. Furthermore, very small models with centimeter-scale grid size are tested and the results are satisfactory.
    - The numerical errors at the early timesteps of simulation still occur in OPM Flow. For now, it could be mitigated by reducing timestep size, but for small-scale cases it is hard to minimize the errors.
    - One new issue that was noticed in the new version of OPM Flow is the shutting down of wells when its BHP constraint is the same as initial pressure. The well should be able to work unless pressure is below the BHP constraint. The only way to mitigate this now is by using WTEST keyword in Flow input file.
    - The oil injection issue found in Reservoir Simulation course project no longer occurs. Previously, production wells are found to inject oil in order to keep pressure above constraint when reservoir pressure drops.
2. OPM Flow has been tested against Schlumberger ECLIPSE for benchmarking and against a fully-implicit Python script for validation purposes.
    - ECLIPSE is found to work best for multiphase flow scenarios, and has fewer issues compared to Flow such as less numerical issues at early simulation timesteps because ECLIPSE takes the first timestep to equilibriate the model. However, for single-phase cases some issues that are not found in Flow are encountered in ECLIPSE such as wrong reporting of FPR values and wells

shutting down. These issues are found in single-phase cases with small grid sizes.

- A Python script for fully-implicit reservoir simulation have been built for validating results from OPM Flow. The pressure and saturation results are found to be identical. However, the water saturation curve is smoother in Python compared to Flow that suggests there are room for improvements for Flow.

3. From the testing, benchmarking, and validation efforts conducted in this study, several recommendations are suggested for future developments of OPM Flow.

   - Improve well control parameter definition in WCONPROD or WCONINJE that should not shut a well if the BHP constraint is the same as initial reservoir pressure.
   - Implement a similar technique used in ECLIPSE that equilibriate the model during the first timestep. This could fix problems such as numerical issues commonly found in Flow on early timesteps.
   - Check into why saturation curve oscillates after the model is run for some timesteps, as found when validating Flow results with fully-implicit Python script. A suggestion is to improve interpolation or trendline techniques used in getting values in Flow input tables, such as relative permeability table or dead oil properties table.

Bibliography

Baxendale, D., . Open Porous Media Flow Documentation Manual version 2019-04 , 974.

Berg, C.F., 2019. Lecture Notes in Reservoir Simulation , 135.

Craft, B.C., Hawkins, M.F., Terry, R.E., 1991. Applied Petroleum Reservoir Engineering. Prentice Hall. Google-Books-ID: uDFQAQAAIAAJ.

Dake, L.P., 1978. Fundamentals of Reservoir Engineering. Number 8 in Developments in petroleum science, Elsevier Scientific Pub. Co. ; distributors for the U.S. and Canada Elsevier North-Holland, Amsterdam ; New York : New York.

Ertekin, T., Abou-Kassem, J.H., King, G.R., 2001. Basic Applied Reservoir Simulation. Sirsi) i9781555630898.

LeVeque, R.J., 2007. Finite Difference Methods for Ordinary and Partial Differential Equations. Other Titles in Applied Mathematics, Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898717839.

Neidinger, R.D., 2010. Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming. SIAM Review 52, 545–563. URL: http://epubs.siam.org/doi/10.1137/080743627, doi:10.1137/080743627.

Rasmussen, A.F., Sandve, T.H., Bao, K., Lauser, A., Hove, J., Skaflestad, B., Klöfkorn, R., Blatt, M., Rustad, A.B., Sævareid, O., Lie, K.A., Thune, A., 2019. The Open Porous Media Flow Reservoir Simulator. arXiv:1910.06059 [physics] URL: http://arxiv.org/abs/1910.06059. arXiv: 1910.06059.

Appendix

A Scripts to Install OPM

A.1 Install OPM from binary packages

```
#!/bin/bash

# script to install OPM from binary package
sudo apt-get update
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:opm/ppa
sudo apt-get update

# install Flow and dependencies
sudo apt-get install mpi-default-bin
sudo apt-get install libopm-simulators-bin
```

A.2 Install OPM by building from source

```
#!/bin/bash

# create directory for opm
cd OPM

# install prerequisites
sudo apt-get update -y
sudo apt-get install -y software-properties-common
sudo add-apt-repository -y ppa:opm/ppa
sudo apt-get install -y build-essential gfortran pkg-config cmake
sudo apt-get install -y doxygen ghostscript
sudo apt-get-install -y texlive-latex-recommended libpgf-dev gnuplot
sudo apt-get install -y git-core libdune-geometry-dev
sudo apt-get install -y mpi-default-dev
sudo apt-get install -y libblas-dev libboost-all-dev
sudo apt-get install -y libsuperlu-dev libsuitesparse-dev
sudo apt-get-install -y libtrilinos-zoltan-dev libdune-common-dev
sudo apt-get install -y libdune-istl-dev libdune-grid-dev

# clone directories from repository
git clone https://github.com/OPM/opm-common.git
git clone https://github.com/OPM/opm-material.git
git clone https://github.com/OPM/opm-grid.git
git clone https://github.com/OPM/opm-models.git
git clone https://github.com/OPM/opm-simulators.git
git clone https://github.com/OPM/opm-upscaling.git
```

```
27
28 # build and make
29 for repo in opm-common opm-material opm-grid opm-models
30       opm-simulators opm-upscaling
31 do
32     mkdir $repo/build
33     cd $repo/build
34     cmake ..
35     make
36     cd ../..
37 done
```

B Python script for a Fully-Implicit Reservoir Simulator

```
#!/bin/env python
# This script is intended for fully-implicit simulation of 1-dimensional oil-water system #
import math
import time
import numpy as np
from matplotlib import pyplot as plt

############################## FUNCTIONS DEFINITION ###################################

def relPerm(Sw):
    "Function to calculate water [0] and oil [1] relative permeability based on Sw of a gridblock."
    normSw = (Sw-Swirr)/(1.0-Sorw-Swirr)
    krWater = krwo*normSw**coreyNw
    krOil = (1-normSw)**coreyNo
    return krWater,krOil

def derivRelPerm(Sw):
    "Function to calculate gradient of water [0] and oil [1] relative permeability based on Sw of a gridblock."
    normSw = (Sw-Swirr)/(1.0-Sorw-Swirr)
    dkrWater = (krwo/(1-Sorw-Swirr))*coreyNw*(normSw**(coreyNw-1))
    dkrOil = (coreyNo/(1-Sorw-Swirr))*((1-normSw)**(coreyNw-1))
    return dkrWater,dkrOil

def determineConnection(NGrid):
    "Function to determine connecting gridblocks of a gridblock."
    mConnectionL = np.zeros((NGrid),dtype=int)
    mConnectionR = np.zeros((NGrid),dtype=int)
    for n in range(NGrid):
        mConnectionL[n] = n-1
        mConnectionR[n] = n+1
    mConnectionL[0] = -1
    mConnectionR[NGrid-1] = -1
    mConnections = np.array([mConnectionL,mConnectionR])
    return mConnections

def darcyFlow(Perm,Visc,Pin,Pout):
    "Function to calculate flow rate according to Darcy's Equation."
    return (Perm/Visc)*AGrid*(Pin-Pout)

def calculateTransmissibility(Sw,iGrid):
    "Function to calculate water [0] and oil [1] transmissibilities to all connecting gridblocks."
    fGeo = cPerm * (AGrid/xGrid)
    fViscOil = 1/(cMuo*refBo)
    fViscWat = 1/(cMuw*refBw)
    fRelPermWat = np.zeros((2))
    fRelPermOil = np.zeros((2))
    if iGrid==0:
        fRelPermWat[1] = relPerm(Sw[iGrid])[0]
        fRelPermOil[1] = relPerm(Sw[iGrid])[1]
    elif iGrid==NGrid-1:
```

```
        fRelPermWat[0] = relPerm(Sw[iGrid-1])[0]
        fRelPermOil[0] = relPerm(Sw[iGrid-1])[1]
    else:
        fRelPermWat[0] = relPerm(Sw[iGrid-1])[0]
        fRelPermWat[1] = relPerm(Sw[iGrid])[0]
        fRelPermOil[0] = relPerm(Sw[iGrid-1])[1]
        fRelPermOil[1] = relPerm(Sw[iGrid])[1]
    TransOil = np.array([fGeo*fViscOil*fRelPermOil[0],fGeo*fViscOil*
    fRelPermOil[1]])
    TransWater = np.array([fGeo*fViscWat*fRelPermWat[0],fGeo*fViscWat*
    fRelPermWat[1]])
    return TransWater,TransOil

def fillFlowVectorElement(Sw,Po,phase,iGrid):
    "Function to fill an element of the Flow Vector."
    fVectorElement = 0.
    for m in range(len(mConnections)):
        if mConnections[m][iGrid] < 0:
            fVectorElement += 0.
        else:
            dPressure = Po[mConnections[m][iGrid]]-Po[iGrid]
            Trans = calculateTransmissibility(Sw,iGrid)
            fVectorElement += Trans[phase][m] * dPressure
    return fVectorElement

def fillFlowMatrixDiagElement(Sw,Po,phase,iGrid):
    "Function to fill a diagonal element of the Flow Matrix."
    fMatrixElement = 0.
    for m in range(len(mConnections)):
        im = mConnections[m][iGrid]
        if im < 0 or im > iGrid:
            fMatrixElement += 0.
        else:
            Swm = Sw[im]
            Pom = Po[im]
            dPressure = Pom-Po[iGrid]
            if relPerm(Swm)[phase] == 0:
                dTrans = 0.
            else:
                dTrans = calculateTransmissibility(Sw,iGrid)[phase][m]*
    derivRelPerm(Swm)[phase]/relPerm(Swm)[phase]
            fMatrixElement += dTrans * dPressure
    return fMatrixElement

def fillFlowMatrixNondiagElement(Sw,Po,phase,iGrid,mGrid):
    "Function to fill a non-diagonal element of the Flow Matrix."
    dPressure = Po[mGrid] - Po[iGrid]
    Swm = Sw[mGrid]
    if mGrid > iGrid or relPerm(Swm)[phase] == 0:
        fMatrixElement = 0.
    else:
        mIdx = (mGrid+1-iGrid)/2
        dTrans = calculateTransmissibility(Sw,iGrid)[phase][mIdx]*
    derivRelPerm(Swm)[phase]/relPerm(Swm)[phase]
        fMatrixElement = dTrans * dPressure
    return fMatrixElement

```

```
104 def fillUnknownVector(Sw,Po):
105     "Function to build the Unknown Vector consisting of Sw and Po."
106     xVector = np.zeros((NGrid,1,2))
107     xVector[:,0,0] = Sw
108     xVector[:,0,1] = Po
109     return xVector
110 
111 def fillResidualVectors(Sw,Po):
112     "Function to build the Residual Vectors consisting of Flow Vector [0],
         Accumulation Vector [1], and Sink/Source Vector [2]."
113     qVector = np.zeros((NGrid,1,2))
114     cVector = np.zeros((NGrid,1,2))
115     fVector = np.zeros((NGrid,1,2))
116     for n in range(NGrid):
117         fVector[n][0][0] = fillFlowVectorElement(Sw,Po,0,n)
118         fVector[n][0][1] = fillFlowVectorElement(Sw,Po,1,n)
119         if n==0 or n==NGrid-1: # for well gridblocks, assume
    transmissibility 1000 times that of reservoir gridblocks
120             fVector[n,:,:] *= 1e3
121     cVector[:,0,0] = (AGrid*xGrid/dTime)*(cPoro*Sw/refBw)
122     cVector[:,0,1] = (AGrid*xGrid/dTime)*(cPoro*(1-Sw)/refBo)
123     cVector[0,:,:] *= (0.5/cPoro) # for well gridblocks, assume porosity
    of 0.5
124     cVector[-1,:,:] *= (0.5/cPoro)
125     qVector[0][0][0] = darcyFlow(cPerm*1e3*relPerm(Sw[0])[0],cMuw,BHPInj,
    Po[0])
126     # for well gridblocks, assume permeability 1000 times that of
    reservoir gridblocks
127     qVector[NGrid-1][0][0] = -1.0*darcyFlow(cPerm*1e3*relPerm(Sw[NGrid-1])
    [0],cMuw,Po[NGrid-1],BHPProd)
128     qVector[NGrid-1][0][1] = -1.0*darcyFlow(cPerm*1e3*relPerm(Sw[NGrid-1])
    [1],cMuo,Po[NGrid-1],BHPProd)
129     return fVector,cVector,qVector
130 
131 def fillJacobianMatrix(Sw,Po):
132     "Function to build the Jacobian Matrix consisting of Flow Matrix [0],
    Accumulation Matrix [1], and Sink/Source Matrix [2]."
133     qMatrix = np.zeros((NGrid,NGrid,2,2))
134     cMatrix = np.zeros((NGrid,NGrid,2,2))
135     fMatrix = np.zeros((NGrid,NGrid,2,2))
136     for n in range(NGrid):
137         cMatrix[n][n][0][0] = (AGrid*xGrid/dTime)*(cPoro/refBw)
138         cMatrix[n][n][0][1] = (AGrid*xGrid/dTime)*Sw[n]*(cPoro*compw/refBw
    )
139         cMatrix[n][n][1][0] = (-AGrid*xGrid/dTime)*(cPoro/refBo)
140         cMatrix[n][n][1][1] = (AGrid*xGrid/dTime)*(1-Sw[n])*(cPoro*compo/
    refBo)
141         cMatrix[0,0,:,0] *= (0.5/cPoro) # for well gridblocks, assume
    porosity of 0.5
142         cMatrix[-1,-1,:,0] *= (0.5/cPoro)
143         if n==NGrid-1:
144             if Sw[n] >= Swc and Sw[n] <= 1-Sorw: # for well gridblocks,
    assume permeability 1000 times that of reservoir gridblocks
145                 qMatrix[0][0][0][0] = (cPerm*1e3*AGrid/cMuw)*(BHPInj-Po
    [0])*(derivRelPerm(Sw[0])[0])
146                 qMatrix[0][0][0][1] = -(cPerm*1e3*relPerm(Sw[0])[0]*AGrid/
    cMuw)
```

```
                qMatrix[n][n][0][0] = -(cPerm*1e3*AGrid/cMuw)*(Po[n]-
    BHPProd)*(derivRelPerm(Sw[n])[0])
                qMatrix[n][n][1][0] = (AGrid*cPerm*1e3/cMuo)*(Po[n]-
    BHPProd)*(derivRelPerm(Sw[n])[1])
                qMatrix[n][n][0][1] = -(cPerm*1e3*relPerm(Sw[n])[0]*AGrid/
    cMuw)
                qMatrix[n][n][1][1] = -(cPerm*1e3*relPerm(Sw[n])[1]*AGrid/
    cMuo)
    for n in range(NGrid):
        for m in range(NGrid):
            if n==m:
                fMatrix[n][m][0][0] = fillFlowMatrixDiagElement(Sw,Po,0,n)
                fMatrix[n][m][0][1] = (-1)*np.sum(
    calculateTransmissibility(Sw,n)[0])
                fMatrix[n][m][1][0] = fillFlowMatrixDiagElement(Sw,Po,1,n)
                fMatrix[n][m][1][1] = (-1)*np.sum(
    calculateTransmissibility(Sw,n)[1])
                if n==0 or n==NGrid-1: # for well gridblocks, assume
    transmissibility 1000 times that of reservoir gridblocks
                    fMatrix[n,m,:,:] *= 1e3
            else:
                if m in mConnections[:,n]:
                    mIdx = (m+1-n)/2
                    fMatrix[n][m][0][0] = fillFlowMatrixNondiagElement(Sw,
    Po,0,n,m)
                    fMatrix[n][m][0][1] = calculateTransmissibility(Sw,n)
    [0][mIdx]
                    fMatrix[n][m][1][0] = fillFlowMatrixNondiagElement(Sw,
    Po,1,n,m)
                    fMatrix[n][m][1][1] = calculateTransmissibility(Sw,n)
    [1][mIdx]
                    if n==0 or n==NGrid-1: # for well gridblocks, assume
    transmissibility 1000 times that of reservoir gridblocks
                        fMatrix[n,m,:,:] *= 1e3
    return fMatrix,cMatrix,qMatrix

def NextIterDeltaUnknown(RV,JM,cVC):
    "Function to calculate delta unknown vector for one Newton linear
    iteration."
    blockJacobianMatrix = JM[0] - JM[1] + JM[2]
    blockResidualVector = (RV[1]-cVC) - RV[0] - RV[2]
    jacobianMatrix = np.zeros((NGrid*2,NGrid*2))
    residualVector = np.zeros((NGrid*2))
    for i in range(NGrid):
        residualVector[i*2] = blockResidualVector[i][0][0]
        residualVector[i*2+1] = blockResidualVector[i][0][1]
        for j in range(NGrid):
            jacobianMatrix[i*2][j*2] = blockJacobianMatrix[i][j][0][0]
            jacobianMatrix[i*2][j*2+1] = blockJacobianMatrix[i][j][0][1]
            jacobianMatrix[i*2+1][j*2] = blockJacobianMatrix[i][j][1][0]
            jacobianMatrix[i*2+1][j*2+1] = blockJacobianMatrix[i][j][1][1]
    jacobiInv = np.linalg.inv(jacobianMatrix)
    deltaUnknownNew = np.matmul(jacobiInv,residualVector)
    blockDeltaUnknown = np.zeros((NGrid,1,2))
    for i in range(NGrid):
        blockDeltaUnknown[i][0][0] = deltaUnknownNew[i*2]
        blockDeltaUnknown[i][0][1] = deltaUnknownNew[i*2+1]
```

```
    return blockDeltaUnknown

def NextTimestepUnknown(Sw,Po):
    "Function to calculate the objective unknowns (Sw and Po) for the next timestep."
    SwInit = Sw
    PoInit = Po
    cVectorCurrent = fillResidualVectors(SwInit,PoInit)[1]
    currentUnknown = fillUnknownVector(SwInit,PoInit)
    Err = 1.0
    idx = 0
    while Err >= 1e-4 and idx < 20:
        Err = 0
        RVElements = fillResidualVectors(Sw,Po)
        JMElements = fillJacobianMatrix(Sw,Po)
        deltaUnknown = NextIterDeltaUnknown(RVElements,JMElements,cVectorCurrent)
        newUnknown = currentUnknown + deltaUnknown
        Sw = newUnknown[:,0,0]
        Po = newUnknown[:,0,1]
        for n in range(NGrid):
            if newUnknown[n][0][0] < Swc:
                newUnknown[n][0][0] = Swc
            if newUnknown[n][0][0] > 1-Sorw:
                newUnknown[n][0][0] = 1-Sorw
            Err += math.sqrt((currentUnknown[n][0][0]-newUnknown[n][0][0])**2)
        currentUnknown = newUnknown
        idx += 1
    print 'No. of linear iteration:',idx
    return newUnknown

################################### MAIN CODE ########################################

# Rel. Perm. Curve
Swirr = 0.20
Sorw = 0.20
krwo = 0.40
coreyNo = 2.0
coreyNw = 2.0

# Reservoir Initialization
NGrid = 102
xGrid = 2.0 # m (1 m = 3.281 ft)
AGrid = xGrid**2 # cube grid blocks
cPerm = 1e-13 # m2 (1e-15 m2 = 1 mD)
cMuo = 3e-3 # Pa.s (1e-3 Pa.s = 1.0 cp)
cMuw = 1e-3 # Pa.s (1e-3 Pa.s = 1.0 cp)
cPoro = 0.20
Swc = 0.20
Poi = 1.01e7 # Pa (1e5 Pa = 14.50 psi)
refBw = 1.01 # valid for pressure 50-250 bar
refBo = 1.02 # valid for pressure 55-550 bar
compw = 1e-9 # 1/Pa (1e-5/Pa = 1/bar)
compo = 2e-10 # 1/Pa (1e-5/Pa = 1/bar)

```

```
Sw = np.full((NGrid),Swc)
Po = np.full((NGrid),Poi)
Sw[0] = 1-Sorw

mConnections = determineConnection(NGrid)

# Well Initialization
BHPInj = 2e7 # Pa (1e5 Pa = 14.50 psi)
BHPProd = 1e7 # Pa (1e5 Pa = 14.50 psi)

# Simulation Specifications
TimeStep = 0.5 # days
dTime = TimeStep*86400 # convert to seconds

################################### TESTING AREA
    ########################################

# Print and Plot Sw and Po over time
NSteps = 200 # Number of timesteps for the simulation
SaturationWater = np.zeros((NSteps+1,NGrid))
PressureGrid = np.zeros((NSteps+1,NGrid))
idxGrid = np.arange(NGrid)

TStep0 = fillUnknownVector(Sw,Po)
SaturationWater[0] = Sw
PressureGrid[0] = Po*1e-5

totaltic = time.time()
for i in range(NSteps):
    print 'Timestep',i+1
    tic = time.time()
    newUnknown = NextTimestepUnknown(Sw,Po)
    toc = time.time()
    print 'Time elapsed:',toc-tic,'s \n'
    Sw[:] = newUnknown[:,0,0]
    Po[:] = newUnknown[:,0,1]
    SaturationWater[i+1] = Sw
    PressureGrid[i+1] = Po*1e-5
totaltoc = time.time()
print 'Total time of simulation',totaltoc-totaltic,'s \n'

plt.figure()
cmap = plt.get_cmap('gnuplot')
plt.title('Saturation Distribution Over Time')
for i in range(NSteps):
    if i==0 or (i+1)%(NSteps/10) == 0:
        plt.plot(idxGrid,SaturationWater[i],color=cmap(100*i/NSteps))
plt.xlabel('Grid Index') ; plt.ylabel('s_w, $fraction$')
plt.ylim([0,1])
plt.legend(loc='lower left')

plt.figure()
plt.title('Pressure Distribution Over Time')
for i in range(NSteps):
    if i==0 or (i+1)%(NSteps/10) == 0:
        plt.plot(idxGrid,PressureGrid[i],color=cmap(100*i/NSteps))
plt.xlabel('Grid Index') ; plt.ylabel('Pressure, $bar$')
```

```
plt.legend(loc='lower left')
plt.show()

# Plot kr curve
SwVals = np.linspace(Swirr,1-Sorw,num=50)
SwPlot = SwVals
kroPlot = relPerm(SwVals)[1]
krwPlot = relPerm(SwVals)[0]

plt.figure()
plt.title('Oil-Water Relative Permeability Plot')
plt.plot(SwPlot,kroPlot,color='r',label=r'kro')
plt.plot(SwPlot,krwPlot,color='b',label=r'krw')
plt.xlabel('Saturation, $s_w$') ; plt.ylabel('Rel. perm., $fraction$')
plt.xlim([0,1]); plt.ylim([0,1])
plt.legend(loc='lower right')
# plt.show()
```

NTNU