SINGLE-PHASE FLUID FINITE-DIFFERENCE SIMULATOR USING PYTHON

Bachelor Thesis

By:

Benyamin Manullang

NIM 12211081

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Engineering

at the Department of Petroleum Engineering

Faculty of Mining and Petroleum Engineering

Institut Teknologi Bandung

PETROLEUM ENGINEERING

FACULTY OF MINING AND PETROLEUM ENGINEERING

INSTITUT TEKNOLOGI BANDUNG

2016

VALIDATION SHEET

SINGLE-PHASE FLUID FINITE-DIFFERENCE SIMULATOR USING PYTHON

BACHELOR THESIS

By:

BENYAMIN MANULLANG

NIM: 12211081

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Engineering

at the Department of Petroleum Engineering

Faculty of Mining and Petroleum Engineering

Institut Teknologi Bandung

Approval:

Thesis Advisor

Bandung, 22 June 2016

Zuher Syihab, ST., M.Sc., Ph.D.

197206061999031003

SINGLE-PHASE FLUID FINITE-DIFFERENCE SIMULATOR USING PYTHON

Benyamin Manullang[1], Zuher Syihab, Ph.D.[2]

[1]Student of Petroleum Engineering Department, Institut Teknologi Bandung, Indonesia

[2]Assistant Professor, Petroleum Engineering, Institut Teknologi Bandung, Indonesia

Abstract

The author presents the development of a basic finite-difference reservoir simulator using Python as the programming language. The simulator is similar to that of an old traditional simulator, that is to say that it is used to solve a single-phase fluid flow, in a homogeneous and isotropic medium, and discretized in a Cartesian coordinate system using a finite-difference approach. The author then gives a few examples with different flow direction (1D, 2D, or 3D).

Keywords: finite-difference, reservoir simulation, Python (programming language)

Abstrak

Penulis mempersembahkan hasil pengembangan sebuah simulator *reservoir* metode *finite-difference* yang diprogram menggunakan bahasa pemrograman Python. Cara kera simulator ini mirip dengan simulator konvensional, yaitu digunakan untuk menjelaskan aliran fluida satu fasa, pada sebuah medium homogen dan isotropik, dan didiskritisasi pada sistem koordinat Cartesian dengan pendekatan *finite-difference*. Penulis kemudian memberikan beberapa contoh dengan berbaga macam arah aliran (1D, 2D, atau 3D).

Kata kunci: *finite-difference*, simulasi *reservoir*, Python (bahasa pemrograman)

1. Preface

1.1 Introduction

Often, many undergraduate students majoring in petroleum engineering fail to understand how a reservoir simulator works behind the monitor. Most of these undergraduate students, if not all, must have taken introductory classes on programming, numerical method, and partial differential equations, but still fail to apply them to the field of reservoir simulation. As a result, many of them accept the results at face value without knowing what has happened inside the box. This paper can hopefully give clear explanations on how to utilize those basic knowledges and put them in the form of actual computer program.

Since computers are not able to evaluate the continuous form of a differential equation, we need to approximate the solution to a differential mathematical expression by transforming it into a discrete form. One method that was widely used is known as finite-difference method. Although most commercial reservoir simulators available nowadays no longer use the traditional finite-difference approach, it is still an eye-opening experience to understand how the governing equation is translated into its finite-difference form. In fact, finite-difference is arguably more intuitive than other discretizing approaches (i.e. corner-point, control-volume).

After we derive the finite-difference form of the differential equation of interest, we then proceed with the presentation of how to put it in the form of computer program. Python is chosen as the programming language because it is easy to understand with its clear syntax.

1.2 Research objectives

The objectives covered in this paper are as follows:

a. To produce Python code as an implementation of finite-difference method in solving a single-phase, slightly-compressible, black-oil fluid flow problem
b. To verify the results of simulation examples obtained using this simulator

1.3 Research methodology

The methodology designed in this research is the following:

1. Problem definition
2. Literature review
3. Specify tools
4. Implementation
5. Setting up few examples
6. Verify the results
7. Concluding remark

2. Statements of Theory

2.1 Diffusivity Equation for Fluid Flow in Porous Media

In the field of reservoir engineering, the diffusivity equation governs the fluid flow and is derived using Darcy's law on the basis of conservation of mass. Darcy's law was formulated by a French engineer Henry Darcy in 1856. The theoretical derivation of Darcy's law into diffusivity equation itself dates back to the pioneering work of Wyckoff, Botset, Muskat and Reed in 1934.

Firstly, one needs to choose the coordinate system that will represent the space. The choice of coordinate system is influenced by the predicted nature of the flow. Due to the radial nature of the fluid flow, the diffusivity equation is usually derived using the cylindrical coordinate system with flow in θ and z direction neglected (see Eq. 1, with no sink/source term). However, in this paper we will only consider the Cartesian coordinate system (see Eq. 2).

$$\phi\rho c_T \frac{\partial P}{\partial t} = -\frac{1}{r}\frac{\partial}{\partial r}\left(r\rho \frac{k_r}{\mu}\frac{\partial P}{\partial r}\right) \quad (1)$$

$$\begin{aligned}\phi\rho c_T \frac{\partial P}{\partial t} = -\frac{\partial}{\partial x}\left(\rho \frac{k_x}{\mu}\frac{\partial P}{\partial x}\right) \\ -\frac{\partial}{\partial y}\left(\rho \frac{k_y}{\mu}\frac{\partial P}{\partial y}\right) \\ -\frac{\partial}{\partial z}\left(\rho \frac{k_z}{\mu}\left(\frac{\partial P}{\partial z} - \rho g\right)\right) \\ + q_{sc}(x,y,z,t) \quad (2)\end{aligned}$$

2.2 Finite-difference Method

The concept of finite-difference forms the foundation of solving differential equations numerically. Instead of solving a differential equation that is supposed to be continuous everywhere in its domain, we look at discrete points and form difference equations. Another way to grasp this concept is that we are not evaluating the behavior of $\Delta f(x)$ as Δx gets closer to zero (see Eq. 3) as opposed to the actual definition of derivative (see Eq. 4). This method will introduce some error that depends on the chosen value of h.

$$\frac{df}{dx} \approx \frac{f_{x+h} - f_x}{h} \quad (3)$$

$$\begin{aligned}\frac{df}{dx} &= \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{(x+\Delta x) - (x)} \quad (4) \\ &= \lim_{\Delta x \to 0} \frac{\Delta f(x)}{\Delta x}\end{aligned}$$

App. A presents the derivation of diffusivity equation in Cartesian coordinate system and how it is discretized using finite-difference method.

2.3 Python (programming language) and SciPy stack

Python is a programming language designed by Guido van Rossum in 1991. Python is widely known for its neat design, code readability, and its syntax which allows programmers to express concepts in fewer lines of code. It supports object-oriented programming paradigm which enables programmers to express their thinking as objects that interact with each other. Each object may have their own methods to interact with other objects or to perform specific operations.

This simulator makes use of the **SciPy stack**. SciPy stack is a Python-based ecosystem of open-source software for mathematics, science, and engineering. Two of the core packages which are used in this simulator are **NumPy** and **SciPy library**. NumPy provides numerical array objects, and routines to manipulate them. It has also become fundamental package for scientific computing with Python. SciPy library provides numerical routines and is used hand in hand with NumPy.

2.4 Physical Assumptions on the Model

Frequently, one will be faced with limitations when modelling a physical system. For example, one cannot know the value of porosity (ϕ) or permeability (k) at every coordinate in space since it is impossible to provide core samples of the entire reservoir system. The assumptions made in the development of this simulator are summarized as follows:

Physical assumptions on fluid:

- There is no change in compositions that make up the fluid. This is also known as a black-oil fluid model.
- Fluid has small, constant compressibility. Highly compressible fluid such as gas is not applicable yet.

Physical assumptions on rock:

- The physical behavior of permeability stays the same regardless of the fluid flow (isotropic).

Other assumptions:

- Temperature throughout the reservoir stays constant. Therefore, the only variable that affects mass conservation is pressure.
- Fluid flow in a porous medium being modelled behave under Darcy flow condition (the Reynolds number is below 2000).

3. Implementation

3.1 Designing the Data Structures

The first question we should address is how do we build the computer model? Or specifically, how do we represent the behavior of a **reservoir system** (consisting of **fluid** and **rock**) in a **3D *Cartesian* space** using a computer program? Using this reasoning, the author declare classes as sketched in Fig. 1. These classes form the core data structures that model the reservoir. We store the code for these classes in `core.py` file.

$$\begin{tikzcd}
 & \texttt{Reservoir} \arrow[dl, "\text{contains}"'] \arrow[d, "\text{contains}"] \arrow[dr, "\text{contains}"] & \\
\texttt{Grid} \arrow[d, "\text{contains}"] & \texttt{Fluid} & \texttt{Rock} \\
\texttt{Node} & & \texttt{BoundaryCondition} \arrow[ll, "\text{(optional) assigned to}"']
\end{tikzcd}$$

Figure 1 - Schematic of classes

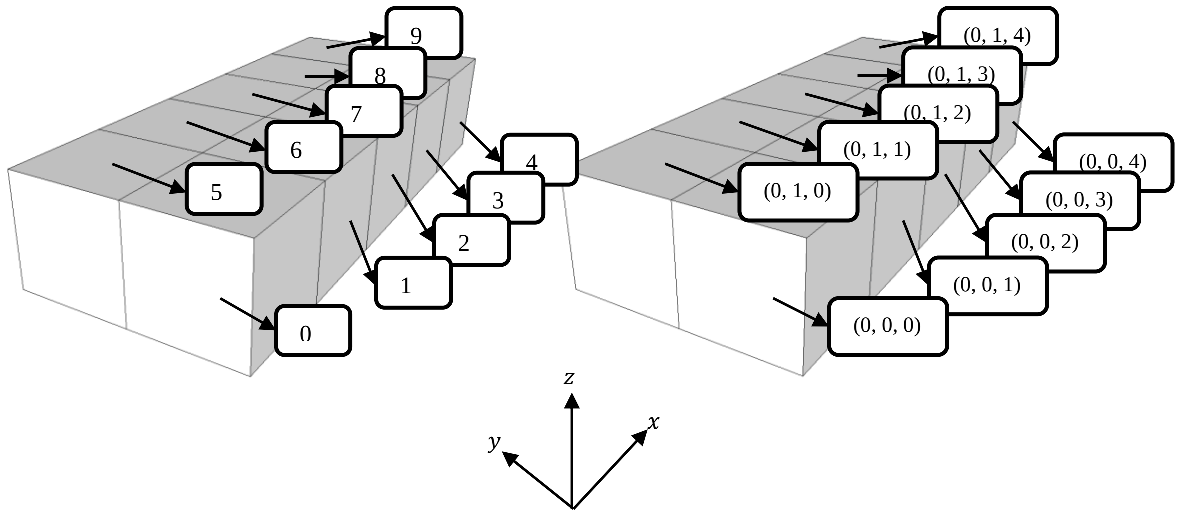


Figure 2 - Flattened index and 3D coordinate index in a 1×2×5 `Grid`

`Node` and `Grid` Classes

`Node` objects will be used to represent a point in space. `Node` will contain information such as the flattened index, the coordinate index (k, j, i notation), whether or not a node is a boundary node, and whether or not a node is a source/sink. Flattened index, in contrast to 3D coordinate index, is a 1D array index. For instance, a `Grid` object with a dimension of (1, 2, 5) will have one grid partition with respect to z direction, two grid partitions with respect to y direction, and five grid partitions with respect to x direction. Each gridblock is bound to a `Node` object. Fig. 2 explains how flattened index and coordinate index are assigned to each gridblock for a (1, 2, 5) `Grid`. This indexing scheme will later be useful when setting up a 2D matrix that consists of linear equations describing pressure relationship among all points at a time level.

`Fluid` and `Rock` Classes

Fluid model in the diffusivity equation presents itself as variables density (ρ) and viscosity (μ). Whereas the model of porous medium (rock model) presents as variables porosity (ϕ) and absolute permeability (k). One should notice that density, viscosity, and porosity are functions of pressure. Density and porosity can further be combined to form total compressibility (c_T), which is equal to compressibility of fluid and compressibility of rock, $c_T = c_l + c_r$.

$$c_l = \frac{1}{\rho}\frac{d\rho}{dP} \qquad c_r = \frac{1}{\phi}\frac{d\phi}{dP} \qquad (5)$$

As given by Eq. 5, for any given pressure value, one can determine the value of density and porosity using the following expression:

$$\rho = \rho_0 e^{c_l(P-P_0)} \qquad \phi = \phi_0 e^{c_r(P-P_0)} \qquad (6)$$

We need to include a reference density or reference porosity with their reference pressure value. For *slightly compressible* fluid model, this equation of state (EOS) is sufficient. However, for *highly compressible* fluid, such as gas, the model must take into account other factors such as z-factor. Also, for this simulator, we will consider a constant value of viscosity (μ) at any given pressure value.

`Reservoir` class

All the information on `Grid`, `Fluid`, and `Rock` objects will then be passed into a `Reservoir` object. In addition, when instantiating a `Reservoir` object, one must also specify the actual dimension of the reservoir (in feet).

`BoundaryCondition` Class

There are two types of boundary condition used in this simulator, Neumann condition (pressure gradient specified) and Dirichlet condition (pressure specified). App. B explains how each condition is implemented on a boundary node. By default, this simulator will specify a no-flow Neumann condition ($\frac{\partial P}{\partial s} = 0$) at every boundary node when a `Grid` object is instantiated.

3.2 Applying the Finite-difference Diffusivity Equation

The finite-difference form of diffusivity equation as derived in App. A is put into Python code in file `differentiator.py`. For each time step, function `oneStepDifferentiator` will go through each coordinate (`Node` object) in `Grid` object and perform what we would like to call as differentiation. The reader may find the code analogous to the derivation in App. A.

Function `oneStepDifferentiator` will then form as many linear difference equations as the number of `Node` objects. These linear equations will then be solved implicitly using `scipy.linalg.solve` function to get pressure distribution for the next time step. Pressure values for each flattened coordinate (as described in Fig. 2) will be generated into a result file.

3.3 Data Visualization

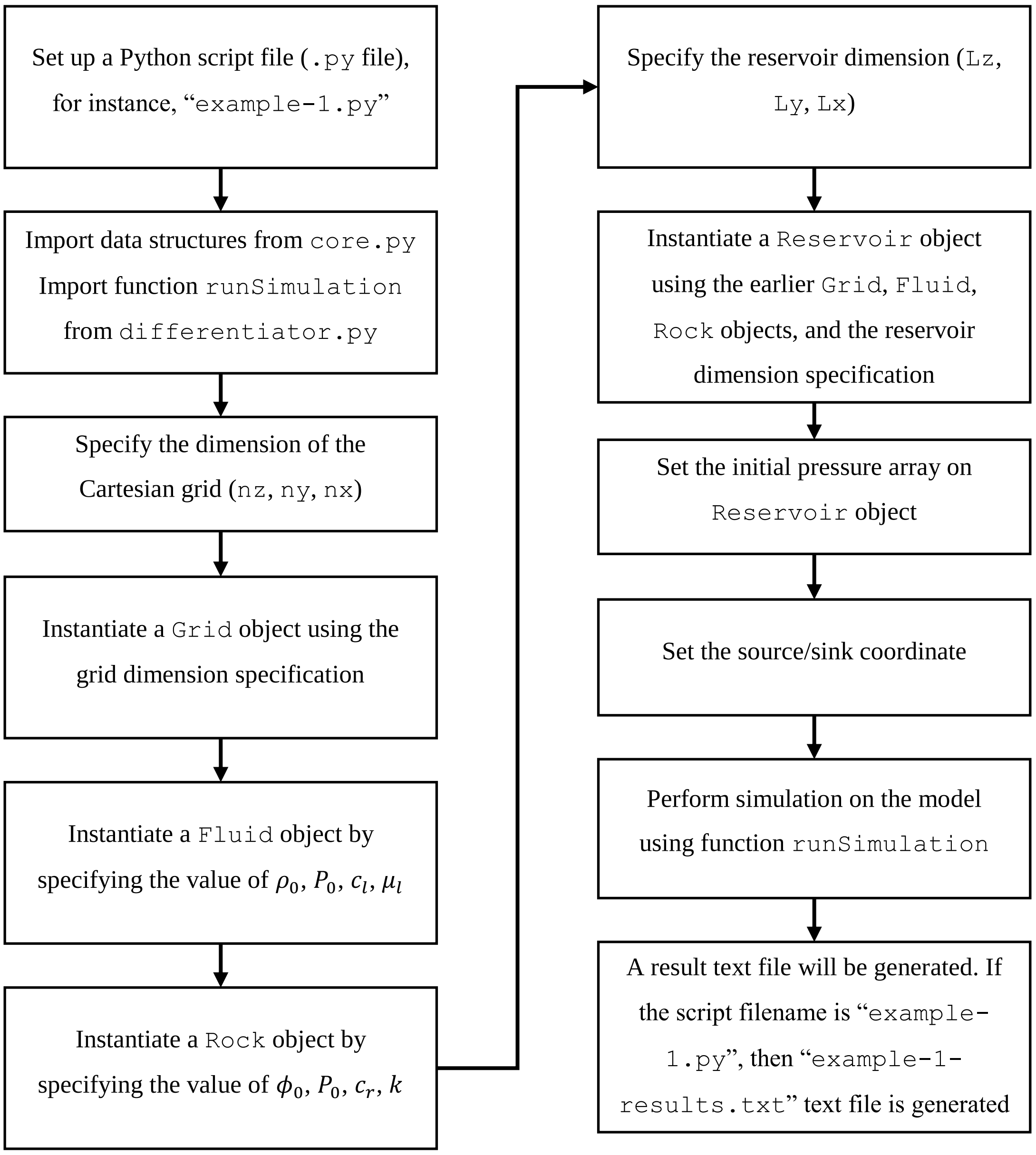
MRST's `plotCellData` function is used to visualize pressure distribution at a time level. This is useful for a 2D or 3D dataset (Lie, K. A., 2014).

3.4 General Workflow on Using the Simulator

App. D provides sample code that is specific to Example 1. Any reader who would like to use this software for other problems may review the code and refer to the workflow sketched in Fig. 3.

4. Simulation Examples

We show how to solve for pressure distribution by presenting four example problems. Each problem shows different flow direction. Using the same physical properties, one can observe how the choice of number of grids affect the calculation of pressure distribution. Example 1 specifically addresses the same problem as given in Ertekin et al. (2001) (Example 5.11).



Set up a Python script file (.py file), for instance, "example-1.py"
Import data structures from core.py Import function runSimulation from differentiator.py
Specify the dimension of the Cartesian grid (nz, ny, nx)
Instantiate a Grid object using the grid dimension specification
Instantiate a Fluid object by specifying the value of ρ_0, P_0, c_l, μ_l
Instantiate a Rock object by specifying the value of ϕ_0, P_0, c_r, k
Specify the reservoir dimension (Lz, Ly, Lx)
Instantiate a Reservoir object using the earlier Grid, Fluid, Rock objects, and the reservoir dimension specification
Set the initial pressure array on Reservoir object
Set the source/sink coordinate
Perform simulation on the model using function runSimulation
A result text file will be generated. If the script filename is "example-1.py", then "example-1-results.txt" text file is generated

Figure 3 - General workflow on using the simulator

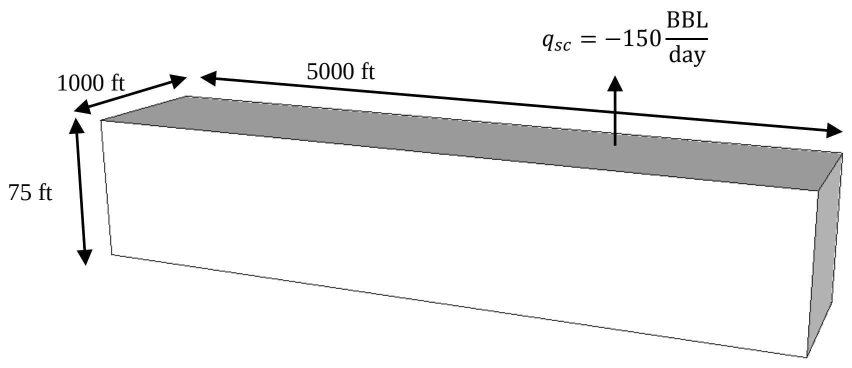
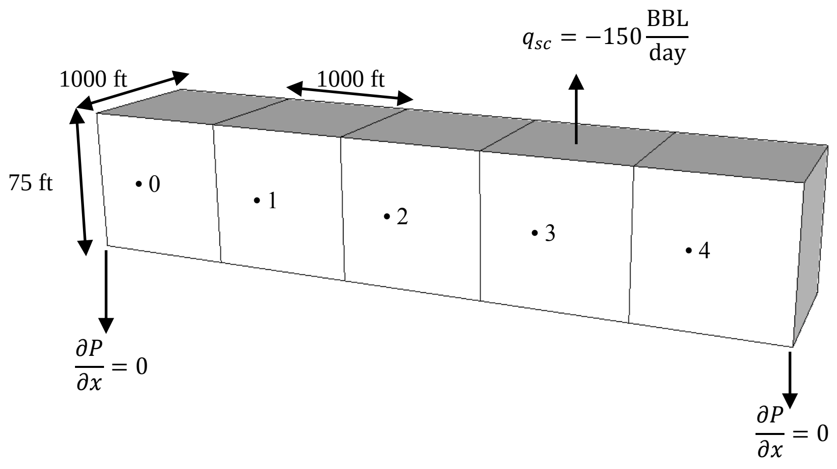


Figure 4 - Sketch of reservoir



$q_{sc} = -150\frac{\text{BBL}}{\text{day}}$
1000 ft
1000 ft
75 ft
• 0
• 1
• 2
• 3
• 4
$\frac{\partial P}{\partial x} = 0$
$\frac{\partial P}{\partial x} = 0$

Figure 5 - Porous medium and grid block system for Example 1

Example 1: 1D-flow ($1 \times 1 \times 5$ Grid dimension)
Suppose we have a porous medium that can be approximated by a cube shape with $75\text{ ft} \times 1000\text{ ft} \times 5000\text{ ft}$ spatial dimension (see Fig. 4). There exists a constant-rate sink term at $x = 3{,}500\text{ ft}$, $y = 500\text{ ft}$. The rock and fluid properties for this problem are: $\rho_l = 62\frac{\text{lb}_\text{m}}{\text{ft}^3}$, $c_l = 3.5 \times 10^{-6}\text{ psi}^{-1}$, $k_x = 15\text{ mD}$, $\phi = 0.18$, and $\mu_l = 10\text{ cP}$. With an initial pressure of $6{,}000\text{ psi}$ and $\Delta t = 15\text{ days}$, determine the pressure distribution during the first year of production.

For this problem, we position the sink term at grid coordinate (0, 0, 3). The resulting pressure distribution by the end of the year is visualized by Fig. 6. It can be seen that the results computed by this simulator agree with the example from Ertekin et al. (2001).

Example 2: 2D-flow ($1 \times 10 \times 20$ Grid dimension)
For this problem, we use the same specifications as Example 1. We approximate the location of the sink term at grid coordinate (0, 4, 13). The 3D visualization of pressure distribution by the end of the year is given by Fig. 7.

Example 3: 3D-flow ($5 \times 40 \times 50$ `Grid` dimension)

We use the same problem specifications as Example 1. We approximate the location of the sink term at grid coordinate (0, 19, 34). The 3D visualization of pressure distribution by the end of the year is given by Fig. 8.

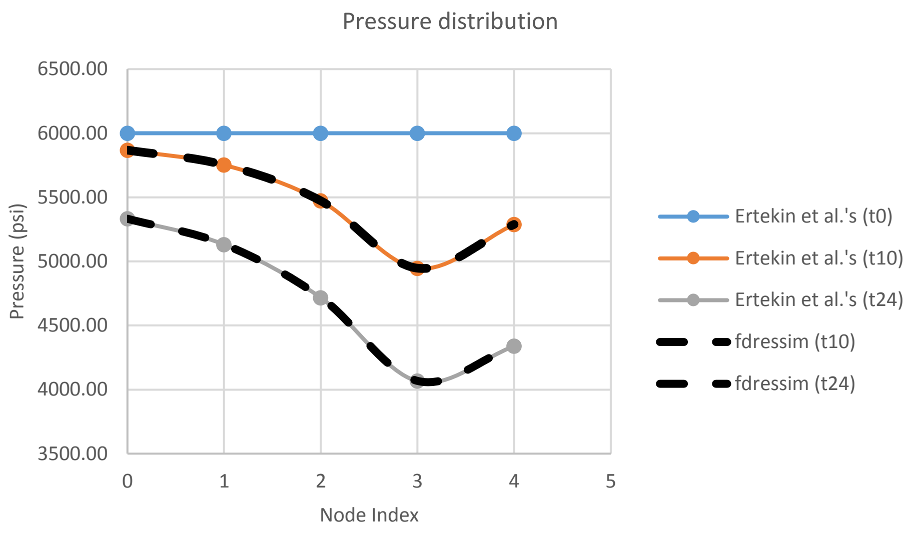


Figure 6 - Pressure distribution by the end of the year for Example 1

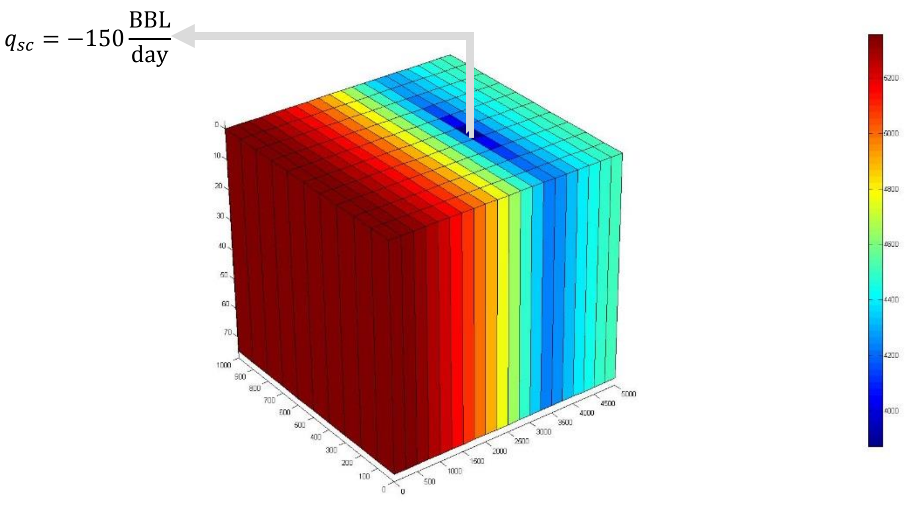


Figure 7 - Pressure distribution by the end of the year for Example 2

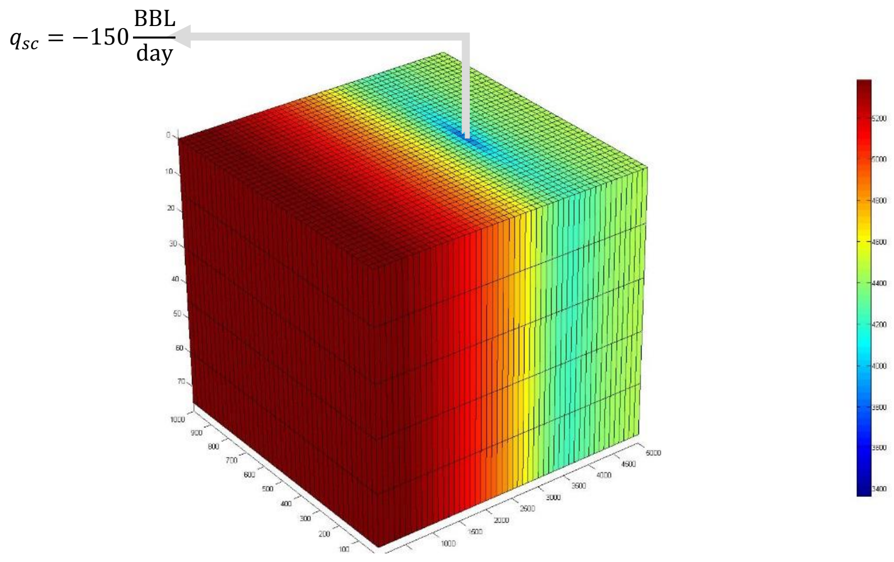


Figure 8 - Pressure distribution by the end of the year for Example 3

Example 4: 2D-flow ($1 \times 171 \times 171$ Grid dimension)

In this example, we consider a cube-shaped physical reservoir with $75 \text{ ft} \times 2{,}500 \text{ ft} \times 2{,}500 \text{ ft}$ spatial dimension. The sink term is positioned right at the center of the reservoir, that is at $x = 1{,}250$ ft and $y = 1{,}250$ ft. Therefore, we set the location of the sink term at grid coordinate (0, 85, 85). It has a constant-rate value of $q_{sc} = -150 \frac{\text{BBL}}{\text{day}}$. We simulate this condition in the span of 3 days with a time step value of $\Delta t = 0.1$ day. Other specifications on fluid and rock properties remain the same as given in Example 1.

If we consider the pressure distribution for the first three days of production (see Fig. 9), it can be seen that the production well still received the transient effect of fluid flow. Furthermore, one may notice that Fig. 10 gives a 2D visualization of pressure distribution that perfectly depicts the radial nature of the fluid flow since we position the sink term right at the center of this reservoir model. Using analytical *line source solution* for constant-rate production given by Matthews and Russel (Eq. 7), we generate analytical solution for $0 \leq r \leq 1{,}250$, $0 < t \leq 4$. We can see that the solution to $P(x, y, t)$ obtained using this simulator gives satisfying results in comparison to the solution to $P(r, t)$ (see Fig. 11) obtained using analytical solution.

$$P(r,t) = P_i - \frac{70.6 q\mu}{kh}\left[-Ei\left(-\frac{\phi\mu c_t r^2}{0.00105 kt}\right)\right] \qquad (7)$$

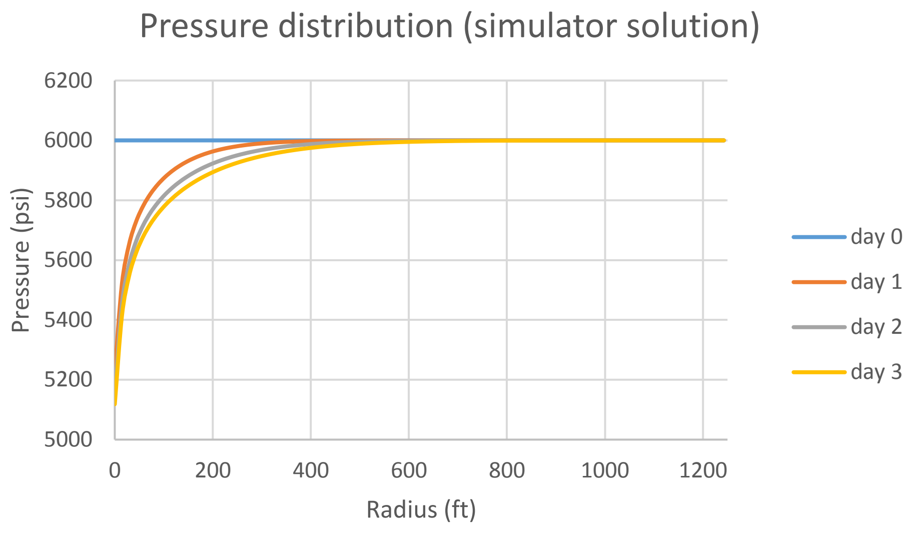


Figure 9 - Pressure distribution for the first four time level for Example 4

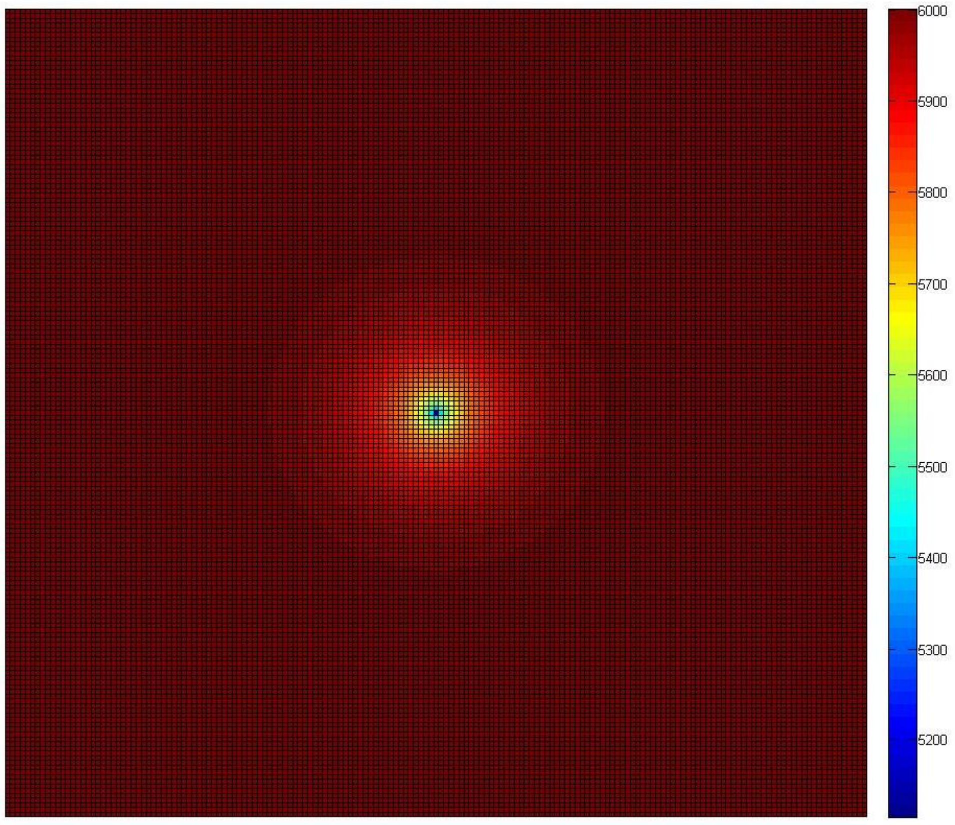


Figure 10 - Pressure distribution at day 3 for Example 4

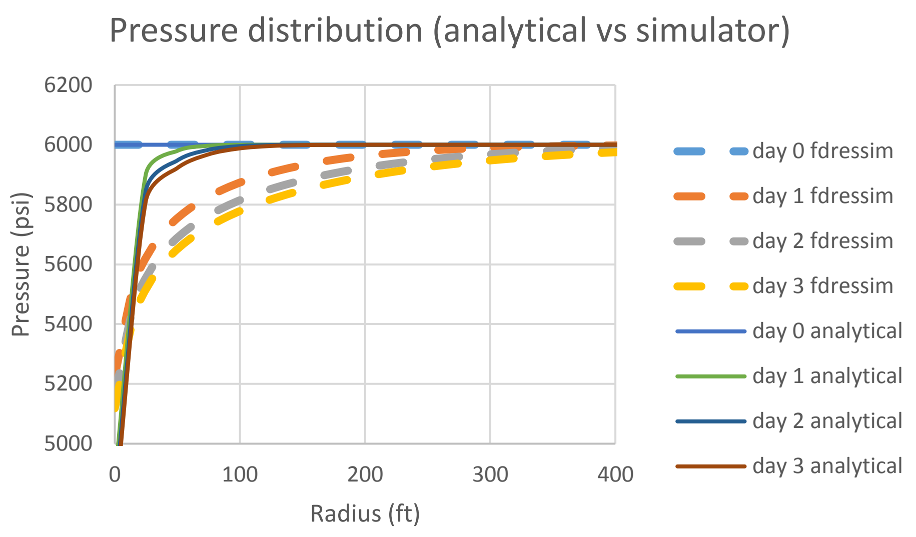


Figure 11 - Pressure distribution of Example 4 (using analytical solution) for the first three days

On the Difference between Numerical Solution and Analytical Solution

Numerical solution obtained using the simulator seems to follow the same trend as analytical *line-source solution* (see Fig. 11). We still need to address what makes the numerical solution does not exactly fit the analytical solution. Two factors to consider are:

a. We are discretizing the diffusivity equation using a Cartesian grid (x, y, z). A better grid type to represent the radial flow better is a cylindrical grid (r, θ, z), since applying finite-difference with respect to both x and y in Cartesian coordinate system is the same as with respect to only r in cylindrical coordinate system.

b. Numerical methods such as finite-difference method can only approximate a differential equation.

5. Concluding Remarks

This paper presented a detailed implementation of building an implicit, single-phase, slightly compressible, black-oil fluid using finite-difference approach in the form of Python code. One can revisit the code for study purpose and may extend or add new features.

6. Recommendations

a. The simulator could include a well model (van Poollen et. al.'s, Peaceman's). So one can specify a pressure-specified or rate-specified well.

b. One can also update the code further to include features like active/inactive gridblocks to approximate various geometries of an actual physical reservoir

c. Further investigation into the accuracy of numerical solution should be further researched.

7. Acknowledgments

The author would like to express gratitude to:

- My family, especially my mother who understands my nerdy need and granted me a rather high-end laptop when I entered college and even further a desktop computer later in my fourth-year.
- My thesis advisor, Zuher Syihab, Ph.D., who is never tired of demonstrating how to code a VBA script in Excel live and spontaneously during his Reservoir Fluid class and keeps telling his students how important it is for a petroleum engineer to be able to code.
- My academic advisor, Asep Kurnia Permadi, Ph.D.
- My friends at UKSU-ITB and HMTM “PATRA” ITB

8. References

C. S. Matthews and D. G. Russell. 1967. ***Pressure Buildup and Flow Tests in Wells, Monograph Vol. 1***. Society of Petroleum Engineers of AIME. Dallas, TX: Millet the Printer.

Ertekin, T., Abou-Kassem, J. H., and King, G. R. 2001. ***Basic Applied Reservoir Simulation***. Richardson, Texas: Society of Petroleum Engineers, Inc. p. 93.

Lie, K.-A. 2014. ***An Introduction to Reservoir Simulation Using MATLAB***. Oslo, Norway: SINTEF ICT, Department of Applied Mathematics. p. 82

Wyckoff, R. D., Botset, H. G, Muskat, M., and Reed, D. W. 1934. ***Measurement of Permeability of Porous Media***. AAPG Bull., 18, No. 2, p. 161.

Muskat, M. 1937. ***The Flow of Homogeneous Fluids through Porous Media***. McGraw-Hill, Inc.

Jones E, Oliphant E, Peterson P, et al. 2001. SciPy: ***Open Source Scientific Tools for Python***. http://www.scipy.org/

Python Software Foundation. ***Python Documentation***, version 3.4. http://www.python.org/

Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. 2011. ***The NumPy Array: A Structure for Efficient Numerical Computation***. Computing in Science & Engineering magazine (Volume: 13), p. 22-30. http://dx.doi.org/10.1109/MCSE.2011.37

Appendix A - Finite-difference Derivation of Diffusivity Equation

We begin by observing the conservation of mass equation,

$$\frac{d(m_{cv})}{dt} = \dot{m}_{in} - \dot{m}_{out} + \dot{m}_{sc} \tag{A1}$$

Evaluating the right-hand side, for simplicity, we consider mass rate ($\dot{m}_x$) that only flows in x direction, with term $\dot{m}$ being $\rho u A$,

$$\begin{aligned} \dot{m}_{in} &= \rho u_{x-\frac{\Delta x}{2}} A_{x-\frac{\Delta x}{2}} \\ \dot{m}_{out} &= \rho u_{x+\frac{\Delta x}{2}} A_{x+\frac{\Delta x}{2}} \\ \dot{m}_{in} - \dot{m}_{out} + \dot{m}_{sc} &= \rho u_{x-\frac{\Delta x}{2}} A_{x-\frac{\Delta x}{2}} - \rho u_{x+\frac{\Delta x}{2}} A_{x+\frac{\Delta x}{2}} + \dot{m}_{sc} \end{aligned}$$

Using the derivative definition as follows,

$$\begin{aligned} \frac{\partial(\rho u_x A_x)}{\partial x} &= \lim_{\Delta x \to 0} \frac{(\rho u_x A_x)_{x-\frac{\Delta x}{2}} - (\rho u_x A_x)_{x+\frac{\Delta x}{2}}}{\left(x - \frac{\Delta x}{2}\right) - \left(x + \frac{\Delta x}{2}\right)} \\ \lim_{\Delta x \to 0} -\Delta x \frac{\partial(\rho u_x A_x)}{\partial x} &= \lim_{\Delta x \to 0} (\rho u_x A_x)_{x-\frac{\Delta x}{2}} - (\rho u_x A_x)_{x+\frac{\Delta x}{2}} \end{aligned}$$

Thus, with a slight abuse of notation, the right-hand side of the equation becomes,

$$\dot{m}_{in} - \dot{m}_{out} + \dot{m}_{sc} = -\Delta x \frac{\partial(\rho u_x A_x)}{\partial x} + \dot{m}_{sc}$$

Evaluating the left-hand side of the equation, noticing that $m_{cv} = \phi \rho V_b$, with term V_b being $\Delta x \Delta y \Delta z$,

$$\frac{d(m_{cv})}{dt} = \frac{d(\phi \rho V_b)}{dt}$$

Coming back to the earlier mass conservation equation,

$$\begin{aligned} \frac{d(m_{cv})}{dt} &= \dot{m}_{in} - \dot{m}_{out} + \dot{m}_{sc} \\ \frac{d(\phi \rho V_b)}{dt} &= -\Delta x \frac{\partial(\rho u_x A_x)}{\partial x} + \dot{m}_{sc} \end{aligned}$$

Since V_b and A_x are constants,

$$\begin{aligned} V_b \frac{d(\phi\rho)}{dt} &= -A_x \Delta x \frac{\partial(\rho u_x)}{\partial x} + \dot{m}_{sc} \\ \frac{d(\phi\rho)}{dt} &= \frac{-A_x \Delta x}{V_b} \frac{\partial(\rho u_x)}{\partial x} + \frac{\dot{m}_{sc}}{V_b} \\ \frac{d(\phi\rho)}{dP} \frac{\partial P}{\partial t} &= \frac{-A_x \Delta x}{V_b} \frac{\partial(\rho u_x)}{\partial x} + \frac{\dot{m}_{sc}}{V_b} \\ \phi \rho c_T \frac{\partial P}{\partial t} &= \frac{-A_x \Delta x}{V_b} \frac{\partial(\rho u_x)}{\partial x} + \frac{\dot{m}_{sc}}{V_b} \end{aligned}$$

We now have the general continuity equation. One can generalize the equation further by taking into account the flow in y and z direction as follows,

$$\phi\rho c_T \frac{\partial P}{\partial t} = \frac{-A_x\Delta x}{V_b}\frac{\partial(\rho u_x)}{\partial x} + \frac{-A_y\Delta y}{V_b}\frac{\partial(\rho u_y)}{\partial y} + \frac{-A_z\Delta z}{V_b}\frac{\partial(\rho u_z)}{\partial z} + \frac{\dot{m}_{sc}}{V_b} \qquad \textbf{(A2)}$$

The moment we evaluate the velocity term (u_s), using Darcy's law, we will arrive at the diffusivity equation. Again, for clarity we shall derive the equation further by considering only the flow in x direction. Recall the equation for Darcy's law in some s direction,

$$u_s = \frac{-k}{\mu}\left(\frac{\partial P}{\partial s} - \rho g\frac{\partial z}{\partial s}\right)$$

For flow only in x direction, the continuity equation becomes,

$$\phi\rho c_T \frac{\partial P}{\partial t} \quad = \frac{-A_x\Delta x}{V_b}\frac{\partial}{\partial x}\left(\rho\frac{-k}{\mu}\frac{\partial P}{\partial x}\right) + \frac{\dot{m}_{sc}}{V_b}$$

$$\phi\rho c_T V_b \frac{\partial P}{\partial t} = A_x\Delta x\frac{\partial}{\partial x}\left(\rho\frac{k}{\mu}\frac{\partial P}{\partial x}\right) + A_y\Delta y\frac{\partial}{\partial y}\left(\rho\frac{k}{\mu}\frac{\partial P}{\partial y}\right) + A_z\Delta z\frac{\partial}{\partial z}\left(\rho\frac{k}{\mu}\left[\frac{\partial P}{\partial z} - \rho g\right]\right) + \dot{m}_{sc}$$

We begin translating any $\frac{\partial}{\partial x}$ term (spatial derivative) using finite-difference method,

$$\begin{aligned}
\phi\rho c_T \frac{\partial P}{\partial t} &= \frac{A_x\Delta x}{V_b}\frac{\partial}{\partial x}\left(\rho\frac{k}{\mu}\frac{\partial P}{\partial x}\right) + \frac{\dot{m}_{sc}}{V_b} \\
&= \frac{A_x\Delta x}{V_b}\frac{1}{\Delta x}\left[\left(\rho\frac{k}{\mu}\frac{\partial P}{\partial x}\right)_{x+\frac{\Delta x}{2}} - \left(\rho\frac{k}{\mu}\frac{\partial P}{\partial x}\right)_{x-\frac{\Delta x}{2}}\right] + \frac{\dot{m}_{sc}}{V_b} \\
&= \frac{A_x}{V_b}\left[\left(\frac{\rho k}{\mu}\right)_{x+\frac{\Delta x}{2}}\left(\frac{\partial P}{\partial x}\right)_{x+\frac{\Delta x}{2}} - \left(\frac{\rho k}{\mu}\right)_{x-\frac{\Delta x}{2}}\left(\frac{\partial P}{\partial x}\right)_{x-\frac{\Delta x}{2}}\right] + \frac{\dot{m}_{sc}}{V_b} \\
&= \left[\left(\frac{\rho k A_x}{\mu V_b}\right)_{x+\frac{\Delta x}{2}}\frac{1}{\Delta x}\left(P_{x+\Delta x} - P_x\right) - \left(\frac{\rho k A_x}{\mu V_b}\right)_{x-\frac{\Delta x}{2}}\frac{1}{\Delta x}\left(P_x - P_{x-\Delta x}\right)\right] + \frac{\dot{m}_{sc}}{V_b} \\
\phi\rho c_T \frac{\partial P}{\partial t} &= \left[\left(\frac{\rho k A_x}{\mu V_b\Delta x}\right)_{x+\frac{\Delta x}{2}}\left(P_{x+\Delta x} - P_x\right) - \left(\frac{\rho k A_x}{\mu V_b\Delta x}\right)_{x-\frac{\Delta x}{2}}\left(P_x - P_{x-\Delta x}\right)\right] + \frac{\dot{m}_{sc}}{V_b}
\end{aligned}$$

$$\phi\rho c_T V_b \frac{\partial P}{\partial t} = A_x\Delta x\frac{\partial}{\partial x}\left(\rho\frac{k}{\mu}\frac{\partial P}{\partial x}\right) + \dot{m}_{sc}$$

We can group the term $\left(\frac{\rho k A_x}{\mu V_b\Delta x}\right)_x$ and define them as transmissibility, T_x,

$$\begin{aligned}
\phi\rho c_T \frac{\partial P}{\partial t} &= \left[T_{x+\frac{\Delta x}{2}}\left(P_{x+\Delta x} - P_x\right) - T_{x-\frac{\Delta x}{2}}\left(P_x - P_{x-\Delta x}\right)\right] + \frac{\dot{m}_{sc}}{V_b} \\
&= \left[T_{x+\frac{\Delta x}{2}}P_{x+\Delta x} - \left(T_{x+\frac{\Delta x}{2}} + T_{x-\frac{\Delta x}{2}}\right)P_x + T_{x-\frac{\Delta x}{2}}P_{x-\Delta x}\right] + \frac{\dot{m}_{sc}}{V_b}
\end{aligned}$$

We then proceed by translating the $\frac{\partial}{\partial t}$ term (time derivative) using finite-difference. We also assign superscript t or $t + \Delta t$ to any variable to specify the time level.

$$\begin{aligned}
\phi\rho c_T\left(\frac{P_x^{t+\Delta t}-P_x^t}{\Delta t}\right) &= \left[T^t_{x+\frac{\Delta x}{2}}P^{t+\Delta t}_{x+\Delta x}-\left(T^t_{x+\frac{\Delta x}{2}}+T^t_{x-\frac{\Delta x}{2}}\right)P_x^{t+\Delta t}+T^t_{x-\frac{\Delta x}{2}}P^{t+\Delta t}_{x-\Delta x}\right]+\frac{\dot{m}_{sc}}{V_b}\\
\frac{\phi\rho c_T}{\Delta t}\left(P_x^{t+\Delta t}-P_x^t\right) &= \left[T^t_{x+\frac{\Delta x}{2}}P^{t+\Delta t}_{x+\Delta x}-\left(T^t_{x+\frac{\Delta x}{2}}+T^t_{x-\frac{\Delta x}{2}}\right)P_x^{t+\Delta t}+T^t_{x-\frac{\Delta x}{2}}P^{t+\Delta t}_{x-\Delta x}\right]+\frac{\dot{m}_{sc}}{V_b}
\end{aligned} \quad \textbf{(A3)}$$

We have arrived at the final finite-difference form of diffusivity equation for flow only in x direction. One should notice that any P_x term encountered so far is actually P at some x, y, z coordinate (i.e. $P_{x,y,z}$). Similarly, the term $P_{x+\Delta x}$ actually denotes $P_{x+\Delta x,y,z}$. The subscript y,z is left out for brevity when considering the differential with respect to x. We can further generalize this form and factor in the flow terms for y and z direction as follows,

$$\begin{aligned}
&\begin{bmatrix}\text{differential}\\ \text{in } t\end{bmatrix}-\begin{bmatrix}\text{differential}\\ \text{in } x \text{ direction}\end{bmatrix}-\begin{bmatrix}\text{differential}\\ \text{in } y \text{ direction}\end{bmatrix}-\begin{bmatrix}\text{differential}\\ \text{in } z \text{ direction}\end{bmatrix}\\
&\qquad=\frac{\phi\rho c_T}{\Delta t}P^t_{x,y,z}+\left(-T_{z+\frac{\Delta z}{2}}+T_{z-\frac{\Delta z}{2}}\right)\rho g\Delta z+\frac{\dot{m}_{sc}}{V_b}
\end{aligned} \quad \textbf{(A4)}$$

where:

$$\begin{aligned}
\begin{matrix}\text{differential}\\ \text{in } x \text{ direction}\end{matrix} &= T^t_{x+\frac{\Delta x}{2}}P^{t+\Delta t}_{x+\Delta x}-\left(T^t_{x+\frac{\Delta x}{2}}+T^t_{x-\frac{\Delta x}{2}}\right)P_x^{t+\Delta t}+T^t_{x-\frac{\Delta x}{2}}P^{t+\Delta t}_{x-\Delta x}\\
\begin{matrix}\text{differential}\\ \text{in } y \text{ direction}\end{matrix} &= T^t_{y+\frac{\Delta y}{2}}P^{t+\Delta t}_{y+\Delta y}-\left(T^t_{y+\frac{\Delta y}{2}}+T^t_{y-\frac{\Delta y}{2}}\right)P_y^{t+\Delta t}+T^t_{y-\frac{\Delta y}{2}}P^{t+\Delta t}_{y-\Delta y}\\
\begin{matrix}\text{differential}\\ \text{in } z \text{ direction}\end{matrix} &= T^t_{z+\frac{\Delta z}{2}}P^{t+\Delta t}_{z+\Delta z}-\left(T^t_{z+\frac{\Delta z}{2}}+T^t_{z-\frac{\Delta z}{2}}\right)P_z^{t+\Delta t}+T^t_{z-\frac{\Delta z}{2}}P^{t+\Delta t}_{z-\Delta z}\\
\begin{matrix}\text{differential}\\ \text{in } t\end{matrix} &= \frac{\phi\rho c_T}{\Delta t}\left(P^{t+\Delta t}_{x,y,z}\right)\\
T_s &= \left(\frac{\rho k A_s}{\mu V_b \Delta s}\right)_s
\end{aligned}$$

Appendix B - Implementing Boundary Condition

The finite-difference derivation as described in **App. A** is based on the central difference scheme. This type of scheme is only applicable to a gridblock which has neighboring gridblocks in its x, y, and z direction. However, this is not the case for a boundary gridblock. This situation is best described using a 3D grid sketch (**Fig. B1**). For a boundary gridblock, a special treatment based on either Dirichlet or Neumann boundary condition needs to be made.

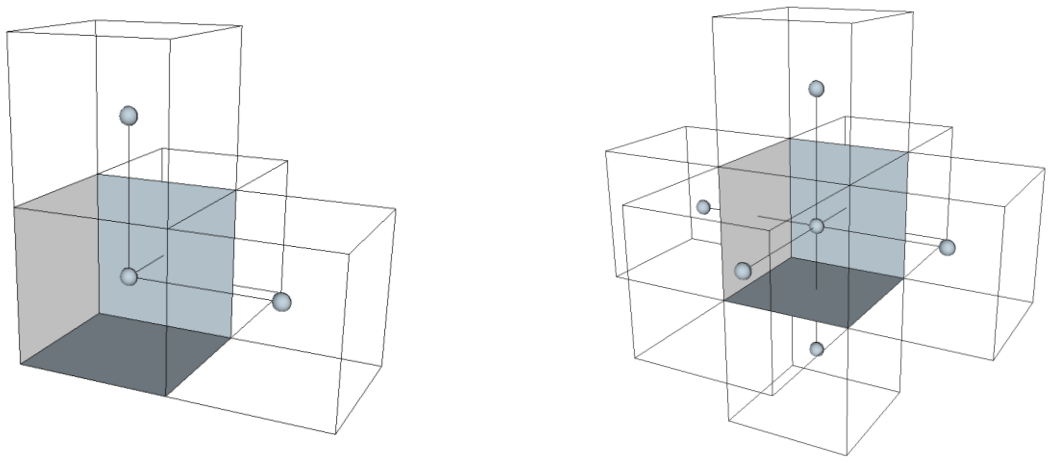


Figure B1 - Sketch of a boundary node (left) and a regular node (right) with their neighboring gridblocks

We restrict further discussion on the flow equation only looking at flow in x direction for simplicity. For instance, consider a boundary gridblock at x that does not have both gridblocks at $x - \frac{\Delta x}{2}$ and $x + \frac{\Delta x}{2}$ to interact with. This means with respect to x, either $P_{x+\Delta x}$ or $P_{x-\Delta x}$ is missing and the diffusivity equation (**Eq. A4**) cannot be completed. Either one of them is the boundary pressure. We need to evaluate how Dirichlet or Neumann condition is imposed on this boundary pressure.

1. Dirichlet condition (pressure specified)

   Suppose the boundary pressure is $P_{x-\Delta x}$, for Dirichlet condition, one can directly specify the boundary pressure,

$$P_{x-\Delta x} = C$$

   The value of C is some constant but can also be a function of time. But we will assume it is constant throughout the time.

2. Neumann condition (pressure gradient specified)

   For this condition, we are given the gradient value, $\frac{\partial P}{\partial x}$, which is equal to C. We can then translate this gradient into the following,

$$\begin{aligned} \frac{\partial P}{\partial x} &= C \\ \frac{\partial P}{\partial x} &= \frac{P_x - P_{x-\Delta x}}{\Delta x} \\ P_{x-\Delta x} &= P_x - C\Delta x \end{aligned}$$

   Similarly, if the boundary pressure is $P_{x+\Delta x}$, the gradient expression can be approximated as follows,

$$\begin{aligned} \frac{\partial P}{\partial x} &= \frac{P_{x+\Delta x} - P_x}{\Delta x} \\ P_{x+\Delta x} &= P_x + C\Delta x \end{aligned}$$

   For a no-flow condition, the gradient is zero ($C = 0$), thus the boundary pressure is directly equal to P_x.

Appendix C - Source code

There are two main files which contain the source code developed for this simulator: `core.py` and `differentiator.py`. They are both hosted online at http://github.com/benjdewantara/fdressim. The data structures are put into `core.py`, while `differentiator.py` describes how finite-difference method is implemented.

`core.py` ***code***

```
import numpy as np

class Node(object):
    def __init__(self, nodeIndx, dims):
        '''
        '''
        self.nodeIndx = int(nodeIndx)
        self.coordIndx = np.unravel_index(self.nodeIndx, dims)
        self.qsrc = 0
        self.initBoundaryNode(dims)

    def initBoundaryNode(self, dims):
        self.boundaryWRTx = np.array([False, {'before':None, 'after':None}])
        self.boundaryWRTy = np.array([False, {'before':None, 'after':None}])
        self.boundaryWRTz = np.array([False, {'before':None, 'after':None}])

        if(self.coordIndx[2] == 0):
            self.boundaryWRTx[0] = True
            self.boundaryWRTx[1]['before'] = BoundaryCondition()
        if(self.coordIndx[1] == 0):
            self.boundaryWRTy[0] = True
            self.boundaryWRTy[1]['before'] = BoundaryCondition()
        if(self.coordIndx[0] == 0):
            self.boundaryWRTz[0] = True
            self.boundaryWRTz[1]['before'] = BoundaryCondition()

        if(self.coordIndx[2] == dims[2]-1):
            self.boundaryWRTx[0] = True
            self.boundaryWRTx[1]['after'] = BoundaryCondition()
        if(self.coordIndx[1] == dims[1]-1):
            self.boundaryWRTy[0] = True
            self.boundaryWRTy[1]['after'] = BoundaryCondition()
        if(self.coordIndx[0] == dims[0]-1):
            self.boundaryWRTz[0] = True
            self.boundaryWRTz[1]['after'] = BoundaryCondition()

    def setSrc(self, flowrate):
        #flowrate in bbl/day, it needs to be converted to ft^3/s
        #multiply with 6.498360546816345e-05
        self.qsrc = 6.498360546816345e-05 * flowrate
        pass
```

```
class Grid(object):
    def __init__(self, dims):
        '''
        self.deltaX, self.deltaY, self.deltaZ are all in ft^3
        '''
        nz, ny, nx = dims
        self.dims = dims
        self.numOfNodes = nz*ny*nx
        self.nodes = self.initNodes()

    def initNodes(self):
        nodes = np.zeros([self.numOfNodes], dtype='O')
        for i in range(self.numOfNodes):
            nodes[i] = Node(i, self.dims)
        return nodes

    def initGridblockGeometry(self, resDim):
        self.deltaX = resDim[2]/self.dims[2]
        self.deltaY = resDim[1]/self.dims[1]
        self.deltaZ = resDim[0]/self.dims[0]
        self.Vb = self.deltaX * self.deltaY * self.deltaZ


class Rock(object):
    def __init__(self, refPoro, refPres, compress, perm):
        '''
        A Rock object will represent the reservoir's rock.
        The properties of a rock we are interested in are:
        porosity (self.poro), and
        absolute permeability (self.perm, in mD).
        Furthermore, the behavior of porosity with respect to pressure is
        explained using getPoro() function on the basis of
        its compressibility (self.compress, in psi^-1)


        '''
        self.refPoro = refPoro
        self.refPres = refPres
        self.compress = compress
        self.perm = perm
        pass

    def getPoro(self, pres):
        return self.refPoro*np.exp(self.compress*(pres - self.refPres))


class Fluid(object):
    def __init__(self, refRho, refPres, compress, mu):
        '''
        A Fluid object will represent a fluid residing in a reservoir.
        The properties of a fluid we are interested in are:
        density (self.rho, in lbm/ft^3), and
        viscosity (self.mu, in cP).
        The behavior of density with respect to pressure is explained using
        compressibility (compressibility() function, in psi^-1)

        '''
        self.refRho = refRho
        self.refPres = refPres
        self.compress = compress
        self.mu = mu

    def getRho(self, pres):
        return self.refRho*np.exp(self.compress*(pres - self.refPres))




class Reservoir(object):
    def __init__(self, grid, fluid, rock, resDim):
        '''
        '''
        self.grid = grid
        self.fluid = fluid
        self.rock = rock
        self.resDim = resDim
        self.arrayOfIndices = np.arange(self.grid.numOfNodes).reshape(self.grid.dims)
        self.grid.initGridblockGeometry(self.resDim)

    def setInitPressure(self, initPressure):
        self.initPressure = np.full(self.grid.dims, initPressure, dtype='float64')




    def addBoundaryCondition(self, bc, **kwargs):
        if not all([key in ["x", "y", "z"] and kwargs[key] in ['after', 'before'] for key
in kwargs]):
            raise ValueError("Key argument must be one of 'x', 'y', or 'z' and the
argument must be either 'after' or 'before'")

        nz, ny, nx = self.grid.dims

        for key in kwargs:
            if(key=='x'):
                if(kwargs[key] == 'before'):
                    for nodes in self.grid.nodes[self.arrayOfIndices[:, :, 0]]:
                        for node in nodes:
                            node.boundaryWRTx[1]['before'] = bc
                elif(kwargs[key] == 'after'):
                    for nodes in self.grid.nodes[self.arrayOfIndices[:, :, nx-1]]:
```

```
                    for node in nodes:
                        node.boundaryWRTx[1]['after'] = bc
            elif(key == 'y'):
                if(kwargs[key] == 'before'):
                    for nodes in self.grid.nodes[self.arrayOfIndices[:, 0, :]]:
                        for node in nodes:
                            node.boundaryWRTy[1]['before'] = bc
                elif(kwargs[key] == 'after'):
                    for nodes in self.grid.nodes[self.arrayOfIndices[:, ny-1, :]]:
                        for node in nodes:
                            node.boundaryWRTy[1]['after'] = bc
            elif(key == 'z'):
                if(kwargs[key] == 'before'):
                    for nodes in self.grid.nodes[self.arrayOfIndices[0, :, :]]:
                        for node in nodes:
                            node.boundaryWRTz[1]['before'] = bc
                elif(kwargs[key] == 'after'):
                    for nodes in self.grid.nodes[self.arrayOfIndices[nz-1, :, :]]:
                        for node in nodes:
                            node.boundaryWRTz[1]['after'] = bc


class BoundaryCondition(object):
    def __init__(self, bcType="n", value=0):
        '''
        '''
        if(bcType not in ['n', 'd']):
            raise ValueError("Wrong boundary condition specification!")
        self.bcType = bcType
        self.value = value
        pass
```

`differentiator.py` *code*

```
import numpy as np
from scipy import linalg


import sys, os
resultsFilename = os.path.basename(sys.argv[0])[:-3] + "-results.txt"


TABchar = "  "

def runSimulation(res, dt, nTime):
    '''
    '''

    resultsFile = open(resultsFilename, 'w')

    presBefore = None

    for i in range(nTime):
        print(TABchar*0 + "Evaluating t=%i" %(i))
        if(i == 0):
            presBefore = res.initPressure
            printArrayToFile(resultsFile, presBefore.reshape(res.grid.numOfNodes))

        print(TABchar*1 + 'presBefore = %s' %(presBefore))

        sle, known = oneStepDifferentiator(res, presBefore, dt)



        presBefore = linalg.solve(sle, known).reshape(res.grid.dims)
        printArrayToFile(resultsFile, presBefore.reshape(res.grid.numOfNodes))


    resultsFile.close()


def printArrayToFile(f, nparray):
    for elm in nparray:
        print("%.4f" %(elm), end=' ', file=f)
    print("\n", end='', file=f)


def oneStepDifferentiator(res, presBefore, dt):
    '''
    res: Reservoir object
    presBefore: array of values of pressure (in psi)
    dt: the value of time interval (whatever unit variable dt is in, it must be
    converted to second)
    '''
    # this function should be designed such that it returns
    # a system of linear equations
    # with its known solution for each linear equation

    # sle stands for system of linear equations :P
    sle = np.zeros([res.grid.numOfNodes, res.grid.numOfNodes], dtype='float64')
    known = np.zeros(res.grid.numOfNodes, dtype='float64')


    for indx in range(res.grid.numOfNodes):
        known[indx] += knownRHS(indx, res, presBefore, dt)

        linEqWRTt = differentialInTime(indx, res, presBefore, dt)
        linEqWRTx, knownWRTx = differentialInX(indx, res, presBefore)
        linEqWRTy, knownWRTy = differentialInY(indx, res, presBefore)
        linEqWRTz, knownWRTz = differentialInZ(indx, res, presBefore)


        known[indx] += knownWRTx + knownWRTy + knownWRTz
        sle[indx] += linEqWRTt
        sle[indx] -= linEqWRTx
        sle[indx] -= linEqWRTy
        sle[indx] -= linEqWRTz

    return sle, known


#this constant is used to modify units of res.fluid.getRho() * 32.17 * res.grid.deltaZ
#so it's in psi
rhoGDeltaZDimMultiplier = 0.3048/(144*9.80665)
phiRhoCompDeltaTMultiplier = 0.3048/(144*9.80665*86400)
srcTermMultiplier = 0.3048/(144*9.80665)

def knownRHS(nodeIndx, res, presBefore, dt):
    coordIndx = res.grid.nodes[nodeIndx].coordIndx
    coordIndxBefore = coordIndx[0]-1, coordIndx[1], coordIndx[2]
    coordIndxAfter = coordIndx[0]+1, coordIndx[1], coordIndx[2]


    #do not forget to modify the dimension!!!
    knownTerm =
srcTermMultiplier*res.fluid.getRho(presBefore[coordIndx])*res.grid.nodes[nodeIndx].qsrc/re
s.grid.Vb \
               +
phiRhoCompDeltaTMultiplier*res.fluid.getRho(presBefore[coordIndx])*res.rock.getPoro(presBe
fore[coordIndx])*totalCompressibility(res, presBefore[coordIndx])/dt *
presBefore[coordIndx] \
               + (transmissibility(coordIndx, coordIndxAfter, res, presBefore)-
transmissibility(coordIndx, coordIndxBefore, res,
presBefore))*rhoGDeltaZDimMultiplier*res.fluid.getRho(presBefore[coordIndx])*32.1740485*re
s.grid.deltaZ
```

```
    return knownTerm



def differentialInTime(nodeIndx, res, presBefore, dt):
    # In a linear equation with some unknowns variables,
    # the left-hand side is usually arranged to contain the unknown terms
    # while the right-hand side contains the constant terms.
    # Hence, the name linEq and knownRHS
    coordIndx = res.grid.nodes[nodeIndx].coordIndx
    linEq = np.zeros(res.grid.numOfNodes, dtype='float64')

    linEq[nodeIndx] +=
phiRhoCompDeltaTMultiplier*res.fluid.getRho(presBefore[coordIndx])*res.rock.getPoro(presBe
fore[coordIndx])*totalCompressibility(res, presBefore[coordIndx])/dt

    return linEq



def differentialInX(nodeIndx, res, presBefore):
    # determine the coordIndx that interacts with nodeIndx
    coordIndx = res.grid.nodes[nodeIndx].coordIndx

    # coordIndxAfter and coordIndxBefore correspond to the coordinate that interact
    # with coordIndx
    coordIndxAfter = coordIndx[0], coordIndx[1], coordIndx[2]+1
    coordIndxBefore = coordIndx[0], coordIndx[1], coordIndx[2]-1

    linEq = np.zeros(res.grid.numOfNodes, dtype='float64')
    known = 0.0
    deltaLen = res.grid.deltaX

    boundaryPresCri = res.grid.nodes[nodeIndx].boundaryWRTx[1]

    # check if there's a 'before' boundary condition w.r.t. coordIndx
    bc = boundaryPresCri['before']
    if(bc != None):
        if(bc.bcType == 'd'):
            known -= transmissibility(coordIndx, coordIndxBefore, res,
presBefore)*bc.value
        elif(bc.bcType == 'n'):
            linEq[nodeIndx] += transmissibility(coordIndx, coordIndxBefore, res,
presBefore)
            known -= -1*bc.value*deltaLen
    else:
        nodeIndxBefore = np.ravel_multi_index(coordIndxBefore, res.grid.dims)
        linEq[nodeIndxBefore] += transmissibility(coordIndx, coordIndxBefore, res,
presBefore)



    # check if there's an 'after' boundary condition w.r.t. coordIndx
    bc = boundaryPresCri['after']
    if(bc != None):
        if(bc.bcType == 'd'):
            known -= transmissibility(coordIndx, coordIndxAfter, res, presBefore)*bc.value
        elif(bc.bcType == 'n'):
            linEq[nodeIndx] += transmissibility(coordIndx, coordIndxAfter, res,
presBefore)
            known -= 1*bc.value*deltaLen
    else:
        nodeIndxAfter = np.ravel_multi_index(coordIndxAfter, res.grid.dims)
        linEq[nodeIndxAfter] += transmissibility(coordIndx, coordIndxAfter, res,
presBefore)


    # finally, with confidence, we perform calculation for nodeIndx (aka coordIndx)
    linEq[nodeIndx] += -1*(transmissibility(coordIndx, coordIndxBefore, res, presBefore) +
transmissibility(coordIndx, coordIndxAfter, res, presBefore))


    return linEq, known




def differentialInY(nodeIndx, res, presBefore):
    # determine the coordIndx that interacts with nodeIndx
    coordIndx = res.grid.nodes[nodeIndx].coordIndx

    # coordIndxAfter and coordIndxBefore correspond to the coordinate that interact
    # with coordIndx
    coordIndxAfter = coordIndx[0], coordIndx[1]+1, coordIndx[2]
    coordIndxBefore = coordIndx[0], coordIndx[1]-1, coordIndx[2]

    linEq = np.zeros(res.grid.numOfNodes, dtype='float64')
    known = 0.0

    deltaLen = res.grid.deltaY

    boundaryPresCri = res.grid.nodes[nodeIndx].boundaryWRTy[1]

    # check if there's a 'before' boundary condition w.r.t. coordIndx
    bc = boundaryPresCri['before']
    if(bc != None):
        if(bc.bcType == 'd'):
            known -= transmissibility(coordIndx, coordIndxBefore, res,
presBefore)*bc.value
        elif(bc.bcType == 'n'):
            linEq[nodeIndx] += transmissibility(coordIndx, coordIndxBefore, res,
presBefore)
            known -= -1*bc.value*deltaLen
    else:
        nodeIndxBefore = np.ravel_multi_index(coordIndxBefore, res.grid.dims)
        linEq[nodeIndxBefore] += transmissibility(coordIndx, coordIndxBefore, res,
presBefore)

    # check if there's an 'after' boundary condition w.r.t. coordIndx
    bc = boundaryPresCri['after']
    if(bc != None):
        if(bc.bcType == 'd'):
            known -= transmissibility(coordIndx, coordIndxAfter, res, presBefore)*bc.value
        elif(bc.bcType == 'n'):
            linEq[nodeIndx] += transmissibility(coordIndx, coordIndxAfter, res,
presBefore)
            known -= 1*bc.value*deltaLen
    else:
        nodeIndxAfter = np.ravel_multi_index(coordIndxAfter, res.grid.dims)
        linEq[nodeIndxAfter] += transmissibility(coordIndx, coordIndxAfter, res,
presBefore)


    # finally, with confidence, we perform calculation for nodeIndx (aka coordIndx)
    linEq[nodeIndx] += -1*(transmissibility(coordIndx, coordIndxBefore, res, presBefore) +
transmissibility(coordIndx, coordIndxAfter, res, presBefore))

    return linEq, known


def differentialInZ(nodeIndx, res, presBefore):
    # determine the coordIndx that interacts with nodeIndx
    coordIndx = res.grid.nodes[nodeIndx].coordIndx

    # coordIndxAfter and coordIndxBefore correspond to the coordinate that interact
    # with coordIndx
    coordIndxAfter = coordIndx[0]+1, coordIndx[1], coordIndx[2]
    coordIndxBefore = coordIndx[0]-1, coordIndx[1], coordIndx[2]

    linEq = np.zeros(res.grid.numOfNodes, dtype='float64')
    known = 0.0

    deltaLen = res.grid.deltaZ

    boundaryPresCri = res.grid.nodes[nodeIndx].boundaryWRTz[1]

    # check if there's a 'before' boundary condition w.r.t. coordIndx
    bc = boundaryPresCri['before']
    if(bc != None):
        if(bc.bcType == 'd'):
            known -= transmissibility(coordIndx, coordIndxBefore, res,
presBefore)*bc.value
        elif(bc.bcType == 'n'):
            linEq[nodeIndx] += transmissibility(coordIndx, coordIndxBefore, res,
presBefore)
            known -= -1*bc.value*deltaLen
    else:
        nodeIndxBefore = np.ravel_multi_index(coordIndxBefore, res.grid.dims)
        linEq[nodeIndxBefore] += transmissibility(coordIndx, coordIndxBefore, res,
presBefore)

    # check if there's an 'after' boundary condition w.r.t. coordIndx
    bc = boundaryPresCri['after']
    if(bc != None):
        if(bc.bcType == 'd'):
            known -= transmissibility(coordIndx, coordIndxAfter, res, presBefore)*bc.value
        elif(bc.bcType == 'n'):
            linEq[nodeIndx] += transmissibility(coordIndx, coordIndxAfter, res,
presBefore)
            known -= 1*bc.value*deltaLen
    else:
        nodeIndxAfter = np.ravel_multi_index(coordIndxAfter, res.grid.dims)
        linEq[nodeIndxAfter] += transmissibility(coordIndx, coordIndxAfter, res,
presBefore)


    # finally, with confidence, we perform calculation for nodeIndx (aka coordIndx)
    linEq[nodeIndx] += -1*(transmissibility(coordIndx, coordIndxBefore, res, presBefore) +
transmissibility(coordIndx, coordIndxAfter, res, presBefore))

    return linEq, known


def totalCompressibility(res, pres):
    return (res.fluid.compress + res.rock.compress)


#this constant is used to modify units of transmissibility so it's in s/ft^2
transmissibilityDimMultiplier = 0.3048**-3 * 0.453592 * 1e-4 / 101325
#transmissibilityDimMultiplier = 1

def transmissibility(coordIndx, wrtCoord, res, presBefore):
    # this function getBoundaryPres() is only used inside transmissibility() function
    def getBoundaryPres(pres, bc, direction, deltaLen):
        if(bc.bcType == 'n'):
            if(direction=='before'):
                return pres - (bc.value*deltaLen)
            elif(direction=='after'):
                return pres + (bc.value*deltaLen)
        elif(bc.bcType == 'd'):
            return bc.value


    # wr is used to determine w.r.t. which 3D direction this transmissibility term is
calculated
    # forward is used to determine if it's the forward transmissibility (x+) or not (x-)
    wr = None
    forward = True
    for i in range(len(coordIndx)):
        differ = coordIndx[i]-wrtCoord[i]
        if(differ != 0):
            wr = i
            if(differ > 0):
                forward = False
            break



    # if wr = 0, we're looking at the transmissibility with respect to z
    # if wr = 1, it is with respect to y
    # if wr = 2, it is with respect to x

    # we need to complete variables rho, perm, area, mu, Vb, and deltaLen
    # then return the transmissibility

    deltaLen = None
    area = None
    boundaryPresCri = None

    if(wr == 0):
        deltaLen = res.grid.deltaZ
        area = res.grid.deltaX * res.grid.deltaY
        boundaryPresCri = res.grid.nodes[np.ravel_multi_index(coordIndx,
res.grid.dims)].boundaryWRTz[1]
    elif(wr == 1):
        deltaLen = res.grid.deltaY
        area = res.grid.deltaX * res.grid.deltaZ
        boundaryPresCri = res.grid.nodes[np.ravel_multi_index(coordIndx,
res.grid.dims)].boundaryWRTy[1]
    elif(wr == 2):
        deltaLen = res.grid.deltaX
        area = res.grid.deltaY * res.grid.deltaZ
        boundaryPresCri = res.grid.nodes[np.ravel_multi_index(coordIndx,
res.grid.dims)].boundaryWRTx[1]

    Vb = res.grid.Vb
    perm = res.rock.perm

    wrtPres = None

    if(forward) and (boundaryPresCri['after'] != None):
        wrtPres = getBoundaryPres(presBefore[coordIndx], boundaryPresCri['after'],
'after', deltaLen)
    elif(not forward) and (boundaryPresCri['before'] != None):
        wrtPres = getBoundaryPres(presBefore[coordIndx], boundaryPresCri['before'],
'before', deltaLen)
    else:
        wrtPres = presBefore[wrtCoord]
```

```
presAvg = (presBefore[coordIndx] + wrtPres)/2


rho = res.fluid.getRho(presAvg)
mu = res.fluid.mu

transmiss = transmissibilityDimMultiplier*rho*perm*area/(mu*Vb*deltaLen)
return transmiss
```

Appendix D - Python script for Example 1

```
from core import *
from differentiator import runSimulation

# Firstly, we can specify the dimension of the Cartesian grid
nz, ny, nx = 1, 1, 5
dims = (nz, ny, nx)
g = Grid(dims)

# Build the fluid and rock model
# rho is in lbm/ft^3
# refPres is in psi
# compress is in psi^-1
# mu is in cP
# perm is in D (Darcy)

f = Fluid(refRho=62.428, refPres=14.7, compress=3.5*1e-6, mu=10)
r = Rock(refPoro=0.18, refPres=14.7, compress=0, perm=0.015)

# We contain all information in a Reservoir object
# But we need to specify the whole reservoir dimension first
Lz, Ly, Lx = 75, 1000, 5000
resDimension = (Lz, Ly, Lx)

res = Reservoir(grid=g, fluid=f, rock=r, resDim=resDimension)

# By default, the moment we declare a Node object, a no-flow Neumann
# condition has already been imposed if the Node is a boundary Node.

# Set the initial pressure array
res.setInitPressure(6000)

# Set a source/sink in coordinate (0, 0, 3)
res.grid.nodes[np.ravel_multi_index((0, 0, 3), res.grid.dims)].setSrc(-150)

# Finally, run the simulation!
# dt is delta time (in day)
runSimulation(res, dt=15, nTime=24)

# A result text file will be generated
```