SINGLE-PHASE FLUID FINITE-DIFFERENCE SIMULATOR USING PYTHON

TUGAS AKHIR

Oleh:

BENYAMIN MANULLANG

NIM 12211081

Diajukan sebagai salah satu syarat untuk memperoleh gelar

SARJANA TEKNIK

pada Program Studi Teknik Perminyakan

Fakultas Teknik Pertambangan dan Perminyakan

Institut Teknologi Bandung

**

PETROLEUM ENGINEERING DEPARTMENT

FACULTY OF MINING AND PETROLEUM ENGINEERING

INSTITUT TEKNOLOGI BANDUNG

2016

SINGLE-PHASE FLUID FINITE-DIFFERENCE SIMULATOR USING PYTHON

Benyamin Manullang1, Zuher Syihab, Ph.D.2

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

2Assistant 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)

# 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.

# 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 and 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).

|  |  |
| --- | --- |
|  | (1) |
|  | (2) |

# Finite-difference Calculus

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 as 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 .

|  |  |
| --- | --- |
|  | (3) |
|  | (4) |

App. A presents the derivation of diffusivity equation in cartesian coordinate system.

# 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.

# 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 () or permeability () 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 behavior of porosity stays the same throughout the reservoir.

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).

# 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.

Node

Grid

Fluid

Rock

Reservoir

BoundaryCondition

contains

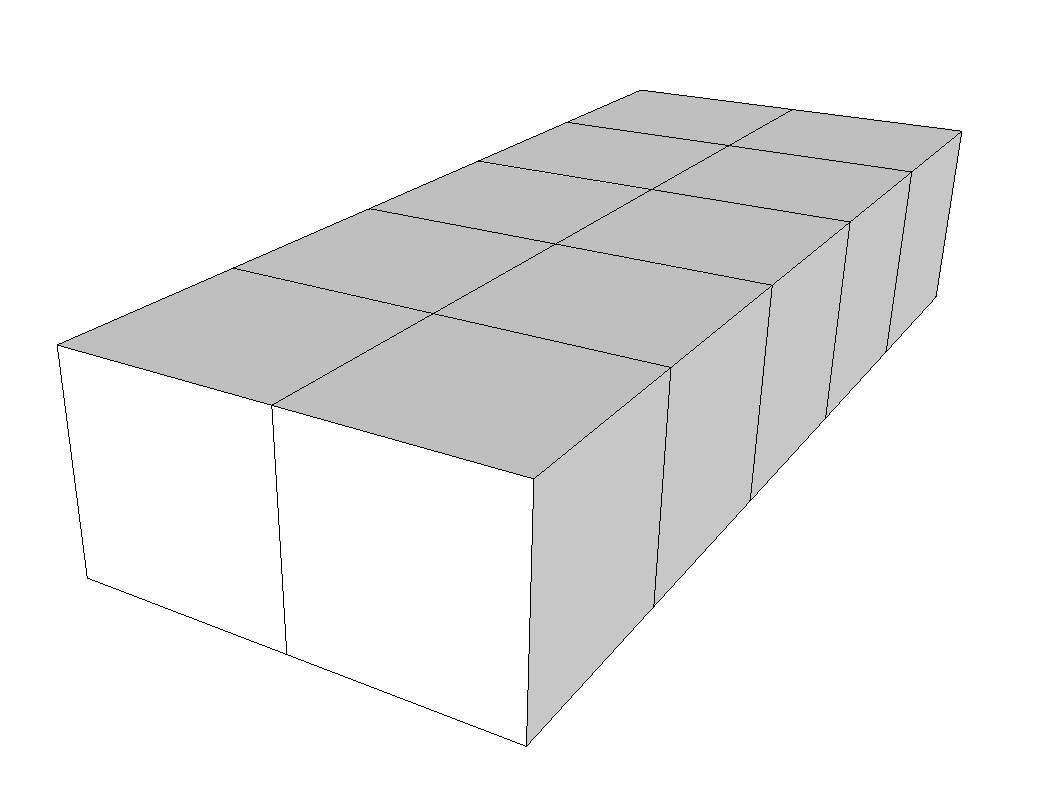
contains

contains

contains

(optional) assigned to

Figure 1 - Schematic of classes



(0, 0, 4)

(0, 0, 3)

(0, 0, 2)

(0, 0, 1)

(0, 0, 0)

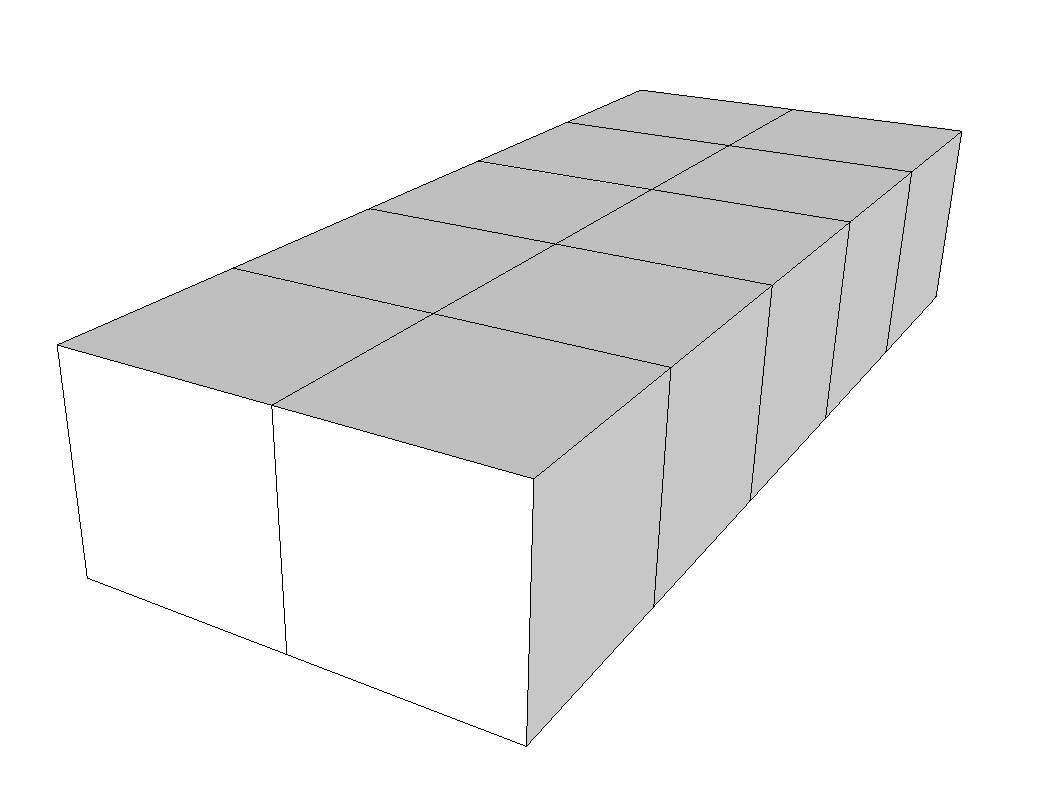
(0, 1, 4)

(0, 1, 3)

(0, 1, 2)

(0, 1, 1)

(0, 1, 0)



4

3

2

1

0

9

8

7

6

5

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 (, , 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 direction, two grid partitions with respect to direction, and five grid partitions with respect to 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 () and viscosity (). Whereas the model of porous medium (rock model) presents as variables porosity () and absolute permeability (). One should notice that density, viscosity, and porosity are functions of pressure. Density and porosity can further be combined to form total compressibility (), which is equal to compressibility of fluid and compressibility of rock, .

|  |  |
| --- | --- |
|  | (5) |

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

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 -factor. Also, for this simulator, we will consider a constant value of viscosity () 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 () at every boundary node when a Grid object is instantiated.

# 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.

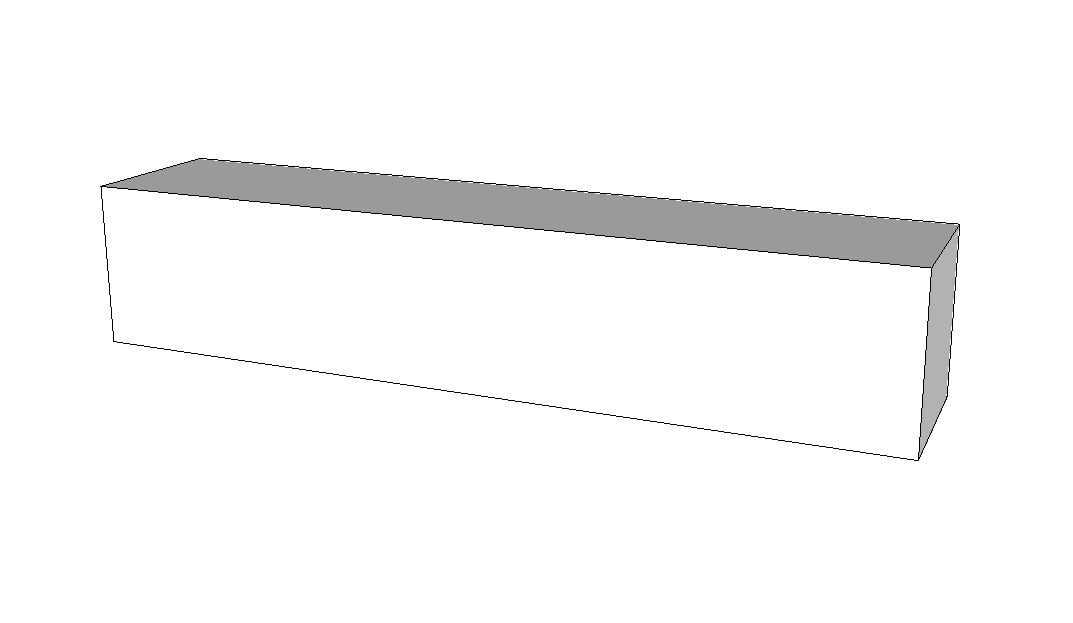
Function oneStepDifferentiator will then form as many linear difference equations as the number of Node objects. These linear equations will then be solved implicitly using scipy.linalg.solve function to get pressure distribution for the next time step. Pressure values for each flattened coordinate (as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

# 3D Data Visualization

MRST’s plotCellData function is used to visualize pressure distribution at a time level.

# 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).

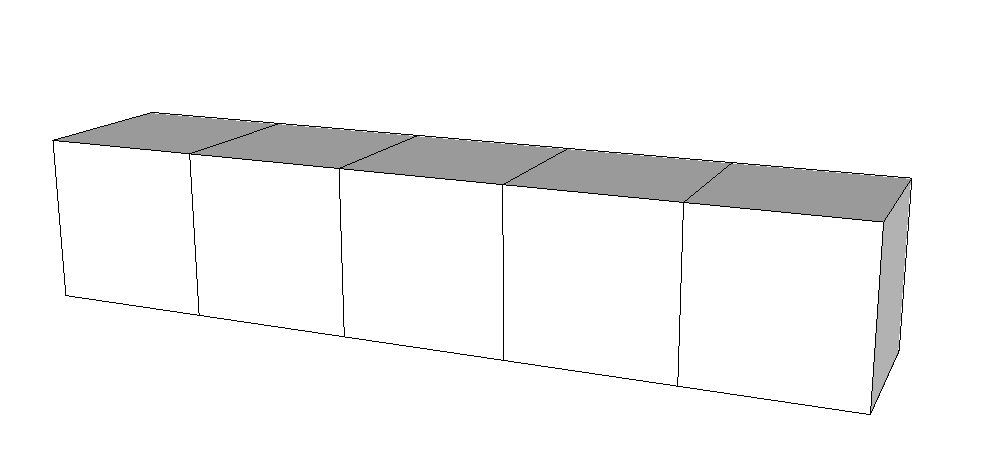


1000 ft

75 ft

5000 ft

Figure 3 - Sketch of reservoir



• 1

• 2

• 3

• 4

• 0

75 ft

1000 ft

1000 ft

Figure 4 - Porous medium and grid block system for Example 1

## ***Example 1: 1D-flow* (Grid *dimension*)**

Suppose we have a porous medium that can be approximated by a cube shape with spatial dimension (see Fig. 3). There exists a constant-rate sink term at , . The rock and fluid properties for this problem are: , , , , and . With an initial pressure of and , 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. 5. It can be seen that the results computed by this simulator agree with the example from Ertekin et al. (2001).

## ***Example 2: 2D-flow* (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 is given by Fig. 6.

## ***Example 3: 3D-flow* (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 is given by Fig. 7.

Figure 5 - Pressure distribution by the end of the year for Example 1

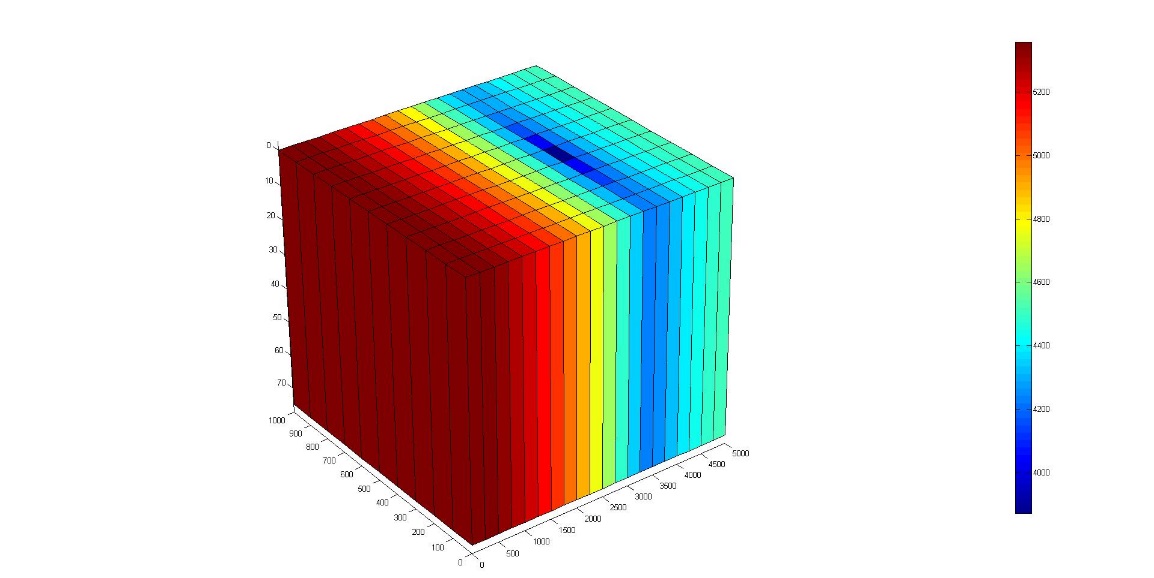


Figure 6 - Pressure distribution by the end of the year for Example 2

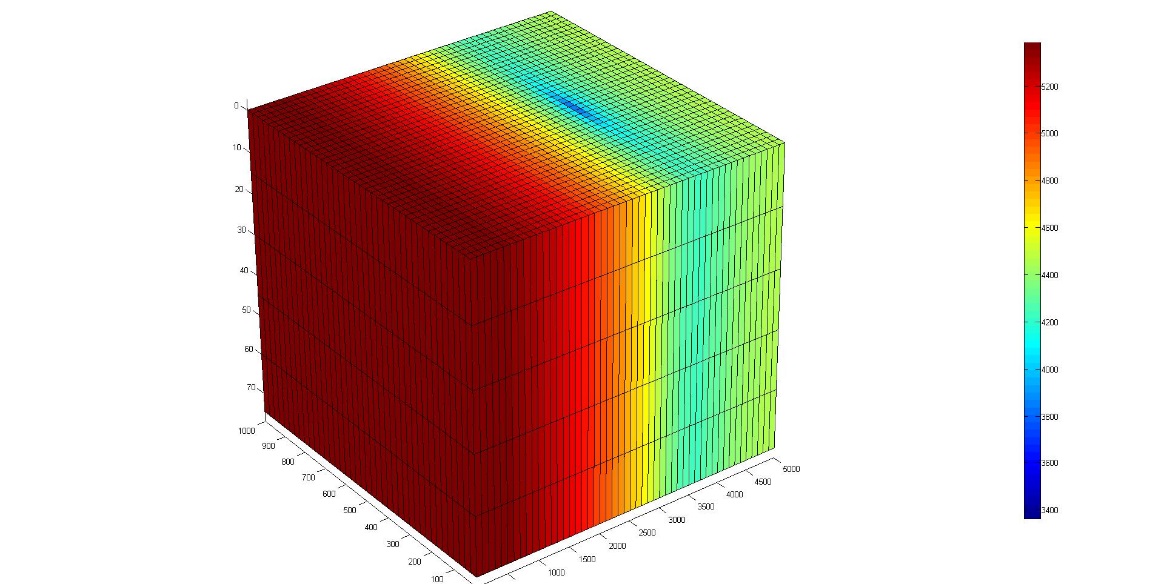


Figure 7 - Pressure distribution by the end of the year for Example 3

## ***Example 4: 2D-flow* (Grid *dimension*)**

In this example, we consider a cube-shaped physical reservoir with spatial dimension. The sink term is positioned at and , so we set the location of the sink term at grid coordinate (0, 25, 25). It has a constant-rate value of . We simulate this condition in the span of with a time step value of . 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. 8), it can be seen that the production well still received the transient effect of fluid flow. Furthermore, one may have noticed that Fig. 9 gives a 2D visualization of pressure distribution that perfectly describes 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. 6), we generate analytical solution for , . We can see that the solution to obtained using this simulator gives satisfying results in comparison to the solution to (see Fig. 10) obtained using analytical solution.

|  |  |
| --- | --- |
|  | (6) |

Figure 8 - Pressure distribution for the first four time level for Example 4

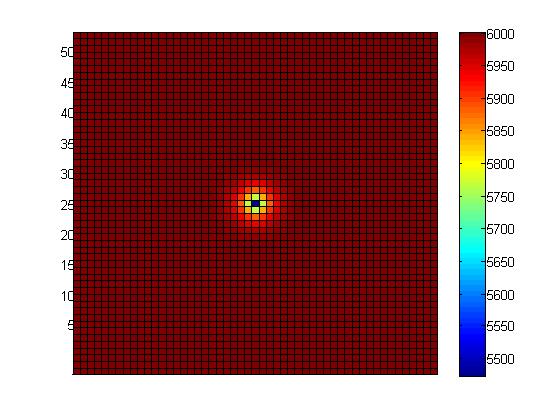


Figure 9 - Pressure distribution at day for Example 4

Figure 10 - Pressure distribution of Example 4 (using analytical solution) for the first three days

# 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.

# Acknowledgments

The author wish to thank Zuher Syihab as the thesis advisor for helpful discussions and recommendations.

# 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.

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

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.

# Appendix A - Finite-difference Derivation of Diffusivity Equation

We begin by observing the conservation of mass equation,

|  |  |
| --- | --- |
|  | **(A1)** |

Evaluating the right-hand side, for simplicity, we consider mass rate () that only flows in direction, with term being ,

Using the derivative definition as follows,

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

Evaluating the left-hand side of the equation, noticing that , with term being ,

Coming back to the earlier mass conservation equation,

Since and are constants,

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

|  |  |
| --- | --- |
|  | **(A2)** |

The moment we evaluate the velocity term (), 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 direction. Recall the equation for Darcy’s law in some direction,

For flow only in direction, the continuity equation becomes,

We begin translating any term (spatial derivative) using finite-difference method,

We can group the term and define them as transmissibility, ,

We then proceed by translating the term (time derivative) using finite-difference. We also assign superscript or to any variable to specify the time level.

|  |  |
| --- | --- |
|  | **(A3)** |

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

|  |  |
| --- | --- |
|  | **(A4)** |

where:

# 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 , , and 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.

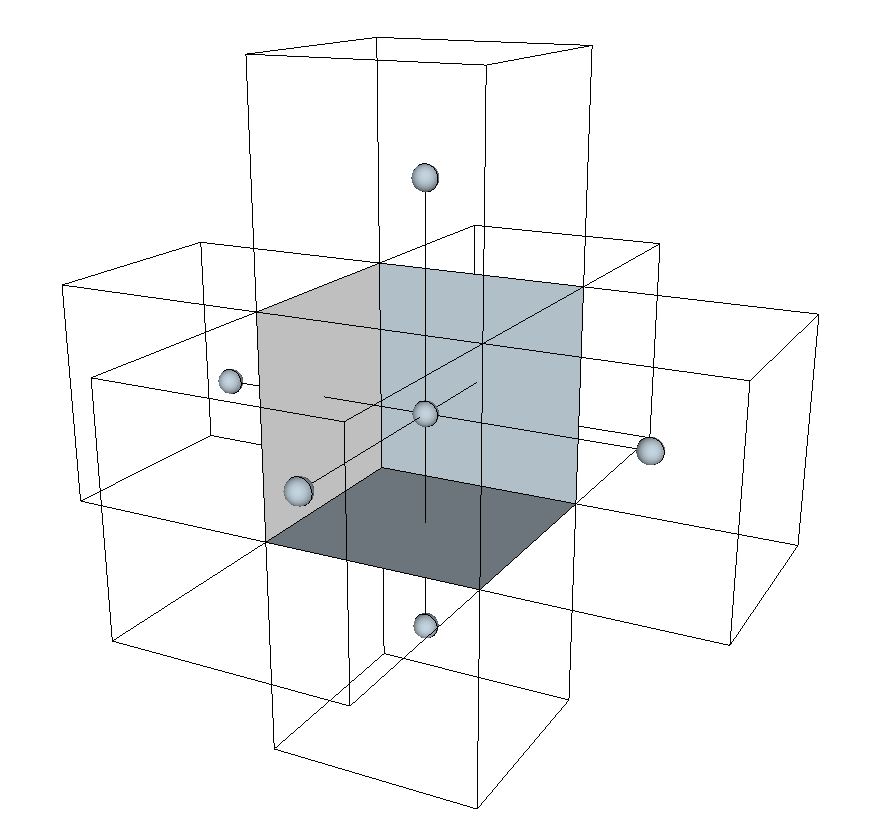
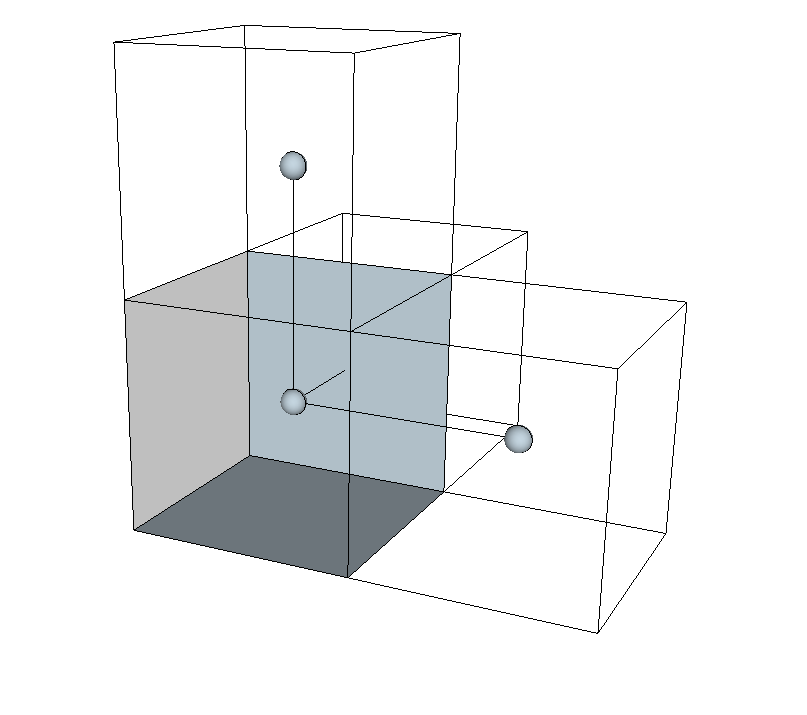


Figure B1 - Sketch of a boundary node (left) and a regular node (right) with their neighboring gridblocks

We restrict further discussion on the flow equation only looking at flow in direction for simplicity. For instance, consider a boundary gridblock at that does not have both gridblocks at and to interact with. This means with respect to , either or 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 , for Dirichlet condition, one can directly specify the boundary pressure,

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

1. Neumann condition (pressure gradient specified)

For this condition, we are given the gradient value, , which is equal to . We can then translate this gradient into the following,

Similarly, if the boundary pressure is , the gradient expression can be approximated as follows,

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

# Appendix C - Source code

There are two main files which contain the source code developed for this simulator: core.py and differentiator.py. 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/res.grid.Vb \

+ phiRhoCompDeltaTMultiplier\*res.fluid.getRho(presBefore[coordIndx])\*res.rock.getPoro(presBefore[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\*res.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(presBefore[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