Undergraduate Project Report
Modeling multi-phase flow through porous media using finite difference approach

Submitted by: Utkarsh (170767)
Supervised by: Prof. Himanshu Sharma

Dept. of Chemical Engineering
The Indian Institute of Technology Kanpur

December 22, 2020

Abstract

The project emphasizes on improving the speed and optimisation of the multi-phase reservoir simulation previously written in MATLAB. Our proposal consists using the Julia language for simulation for better scalability and performance. A support of multi-threading as well as using special data-structures available for Tri-diagonal matrices is also proposed. Further course of action includes adding 2-D and 3-D diamensionality support and using advance non-linear solvers for efficient computations.

Contents

1 Introduction

Reservoir simulation is the means by which we use a numerical model of the geological and petrophysical characteristics of a subsurface reservoir, the (multiphase) fluid system, and the production equipment (wells and surface facilities) to analyze and predict how fluids flow through the reservoir rock to the stock tank or transport pipeline over time.

Predicting how a reservoir will produce over time and respond to different drive and displacement mechanisms therefore has a large degree of uncertainty attached.

2 The Simulation

2.1 Reservoir simulation

Initially, we had a code written for 1-D 2-phase immiscible & slightly compressible flow simulator in MATLAB. A basic reservoir simulation breaks the problem in these ways:

2.1.1 Mathematical Model

First, we need a mathematical flow model that describes how fluids flow in a porous medium. These models are typically given as a set of partial differential equations describing the mass conservation of fluid phases, accompanied by a suitable set of constitutive relations that describe the relationship among different physical quantities.

2.1.2 Geological Model

Second, we need a geological model that describes the given porous rock formation (the reservoir). The geological model is realized as a grid populated with petrophysical properties that are used as input to the flow model, and together they make up the reservoir simulation model.

2.1.3 Pressure and Fluid Dynamics Modelling

Last, but not least, we need a model for the wells and production facilities that provide pressure and fluid communication between the reservoir and the surface.

2.1.4 Our assumption based model

The model assumes a constant compressibility. All of the data are in the SI units. An underlying assumption in the model is that the injector in the first grid block and a producer in the last grid block. It is a simple 1-D simulation which calculates the oil recovery with particulate concentrations.

We start are simulation initially with (15+2) grid blocks, with two grid blocks being the ghost nodes. These ghost nodes are used for central finite difference numerical approximation of the derivatives. We approximate simulate this reservoir for about 144 hrs. It is a space and time dependent model, and we obtain 17 blocks and 1200 blocks of it respectively.
We mainly lay emphasis on the oil recovery, ion concentration and pressure profiles in the grid-blocks.

3 The Algorithm

Our algorithm can be divided mainly into three components:

3.1 Physical constants, Chemical Parameters and Data

A suitable simulation heavily relies on the initial conditions, physical constants and flow diagram.
Here are some of the constants based on our assumption:

1. We estimate the thickness of the grid block which is 0.015 inch (0.381 mm).
2. Permeability, Porosity and the Initial Pressure. Permeability is obtained from the Darcy–Weisbach equation.
3. Total no. of adsorption sites per mole.
4. Injected concentrations of Sodium(Na), Calcium(Ca), Magnesium(Mg), Sulphates(SO4) and Chlorides(Cl).
5. Total pore volumes and time-step.

We also estimate the initial equilibrium surface concentration of these ions by solving the ionic-equilibrium solubility reactions.

3.2 Pressure estimation and concentrations with time

This is the main code block with iterates over time and solves for the pressure profiles in grid-block and ion concentrations. We also calculate the oil recovery at each time-step.

3.2.1 Pressure calculations

The pressure calculation is basic Linear algebra equation (AX = B) solving using LU decomposition. The nature of the A matrix is tri-diagonal and can efficiently compressed in the form of three vectors. MATLAB automatically multi-threads the LU decomposition and inversion. These matrices are obtained by mass-balancing across the grid blocks with discretisation with relative permeability calculations in each phase. The pressure is then connected with capillary pressure.

3.2.2 Geothermal calculations

We now then move towards ion concentration calculations. We calculate the Sodium, Magnesium, Calcium and Sulphate Transport across the grid-blocks. These calculations are done for each grid block and equilibrium concentrations are calculated via *sulfate_ads.m* sub-routine. It encompasses the non-linear ionic-equilibrium balance equations and MATLAB solves it via *fsolve* (Netwon-Raphson iteration method).

3.3 Assimilation of concentrations

We save the effluent concentrations and convert it back to PPM. We also calculate the normal concentrations with oil recovery.

4 Current problems and bottlenecks

4.1 Redundant Memory Allocations

As we discussed earlier, there's a significant space and time complexity. For storing the variables, we need vectors and matrix. As we loop over time, if not taken care, these contribute to allocating memory again in RAM.

Unexpected memory allocation is almost always a sign of some problem with the code, usually a problem with type-stability or creating many small temporary arrays. Consequently, in addition to the allocation itself, it's very likely that the code generated for your function is far from optimal.

4.2 Vectorization of Code

Many CPUs have ”vector” or ”SIMD” instruction sets which apply the same operation simultaneously to two, four, or more pieces of data. Modern x86 chips have the SSE instructions, many PPC chips have the ”Altivec” instructions, and even some ARM chips have a vector instruction set, called NEON.

”Vectorization” (simplified) is the process of rewriting a loop so that instead of processing a single element of an array N times, it processes (say) 4 elements of the array simultaneously N/4 times.
MATLAB supports vectorization and when dealing with huge chunks of data, vectorization becomes essential, often neglected.

4.3 Matrix Calculations

The size of the Matrices in the system scales progressively. Therefore we require efficient linear algebra algorithms to address it so that it does not hinder the performance.

A special data-structures can be used for sparce matrices like tri-diagonal matrices. Sparse arrays are arrays that contain enough zeros that storing them in a special data structure leads to savings in space and execution time, compared to dense arrays.

4.4 Non-linear equation solvers

Since the physical dimension of the system is microscopic, the tolerances must be low. Hence sometimes when finding solution of the Non-linear equations, we may often need tune the accepted tolerances by the solver as solution could be affected significantly. We can also use advanced NL solvers with the benefits of automatic differentiation.

5 Entry of the Julia language

As a part of Google Summer of Code project, I had worked in the Julia language. My work mostly dealt with ODE solvers. Seeing the far-fetched research possibilities in Julia, I suggested we could try to write the Julia and compare its benchmarks.

5.1 About Julia

Julia is a high-level, high-performance, dynamic programming language. While it is a general-purpose language and can be used to write any application, many of its features are well suited for numerical analysis and computational science.

Julia is popular among data scientists and mathematicians. It shares features (such as 1-based array indexing and functional design) with mathematical and data software like Mathematica, and its syntax is closer to the way mathematicians are used to writing formulas. Julia also includes excellent support for parallelism and cloud computing, making it a good choice for big data projects.

5.2 Why Julia?

"As high-level and interactive as Matlab or Python, as general-purpose as Python, as productive for technical work as Matlab or Python+SciPy, but as fast as C."

Writing fast code in Python or Matlab translates to mining the standard library for pre-written functions which are generally implemented in C or Fortran. If the problem doesn't "vectorize" into built-in functions, if you have to write your own inner loops.

Vectorization is not natural for many things. It's eye-opening to find in Julia that you can vectorize any function just by adding a dot to its name. Constructing a matrix through a comprehension makes nested loops (or meshgrid tricks) look like buggy whips in comparison, and avoiding a matrix altogether via a generator for a simple summation feels like getting something for nothing.

Julia allows researchers to write high-level code in an intuitive syntax and produce code with the speed of production programming languages. It saves us from falling back to traditional but fast languages like C and FORTRAN where performance is the key but maintenance of code is tough.

6 Results and Benchmarks

6.1 Converting from MATLAB to Julia

My initial task was to convert the existing MATLAB code to Julia. It was challenging task, converting 500+ lines of code which should yield the same results. So I started converting it and converted the code in Julia using the libraries as when required.

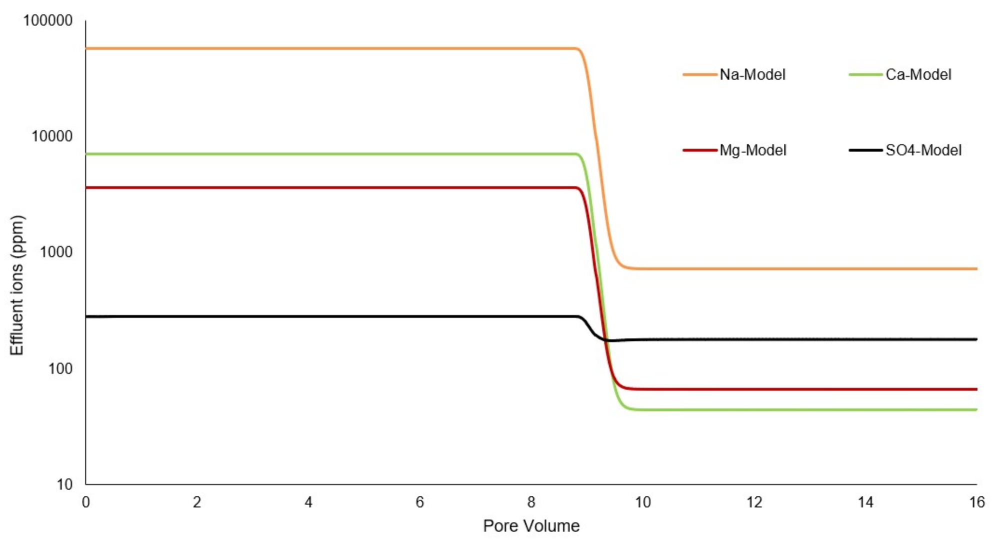
6.2 Optimisations

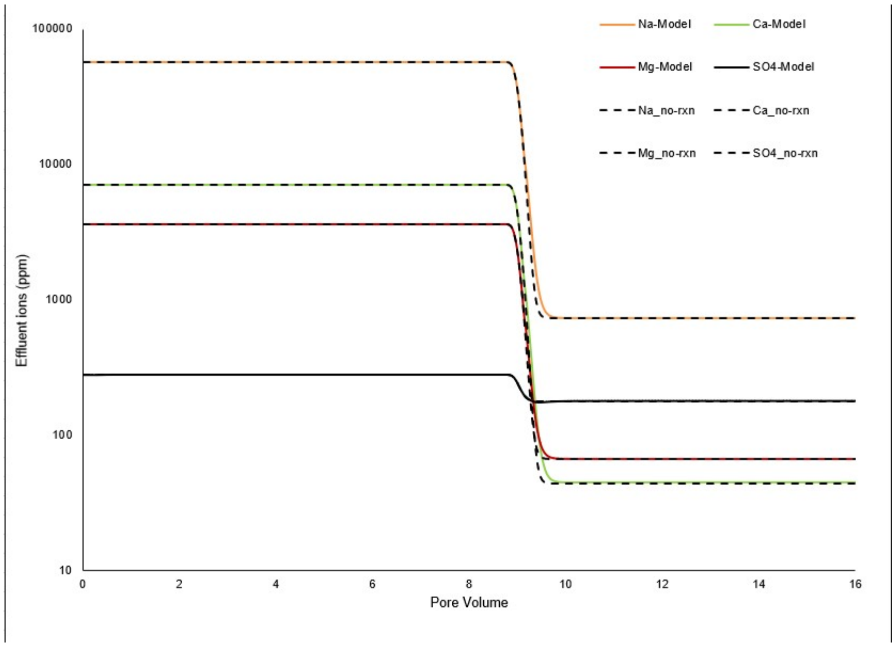
Mostly, the first code code we right is not always optimal. It requires memory as well as some algorithmic optimisations. Since there were many arrays and matrices used in the code, my first task was to fix them my pre-allocating them. It ensures that redundant memory allocations are not present in the code which affects the speed by a significant margin.

There was a significant usage of global variables as well. A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible. Any code that is performance critical or being bench-marked should be inside a function. We find that global names are frequently constants, and declaring them as such greatly improves performance.

6.3 Results

Initially, the unoptimised memory was typically in Megabytes. After optimisation it got reduced to few hundreds of Kilobytes. This turned out to be significant boost in the speed, and code involving Pressure calculations when bench-marked separately resulted in boost of about 10x times when compared with MATLAB. The pressure profiles also obtained where correct too which points to the accuracy of the solution.





7 Future Goals

1. We need to add support for 2-D and 3-D simulation support as well.
2. Implement the multi-threading in non-linear solvers.
3. Work towards more memory optimisations.

Thank you.