University Of Strasbourg

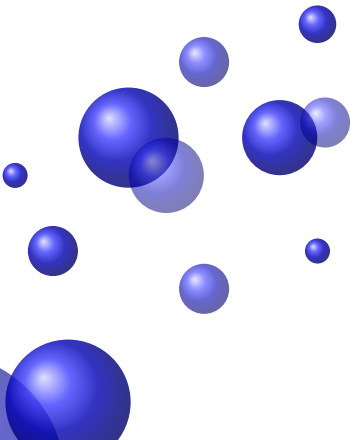
Master CSMI : Scientific computing and mathematics of information

Simulation of flows in heterogeneous porous media with Feel++

Autor :
Obed Saba

Supervisors :
Mr. Joubine Aghili
Mr. Christophe Prud'homme

August 21, 2023



Contents

Acknowledgements

I would like to express my deep gratitude to my internship supervisors, Mr. Joubine AGHILI and Mr. Christophe PRUD'HOMME, for their invaluable support throughout this experience. Their expertise and insightful guidance have been an essential pillar in the successful completion of this internship.

I am grateful for the opportunity they gave me to work on exciting and challenging topics, and for guiding me through the challenges I encountered. Their patience and dedication towards my professional development have made a significant impact on my growth and understanding of the field.

Their tireless dedication to nurturing my understanding, challenging my assumptions, and inspiring my curiosity have truly transformed this internship into a riveting journey of professional discovery. The depth of their knowledge and their willingness to share it has been a beacon guiding me through the complexities of our work. In the tapestry of my career, this internship stands as a vibrant, invaluable thread, enriched by every individual who guided, helped, and encouraged me. I am privileged to have had this opportunity and look forward to carrying these lessons into my future endeavors.

Chapter 1

Context

1.1 Historical context

1.1.1 History of Darcy's law

Darcy's law was first determined experimentally by Darcy, but has since been derived from the Navier–Stokes equations via homogenization methods.[3] It is analogous to Fourier's law in the field of heat conduction, Ohm's law in the field of electrical networks, and Fick's law in diffusion theory.
One application of Darcy's law is in the analysis of water flow through an aquifer; Darcy's law along with the equation of conservation of mass simplifies to the groundwater flow equation, one of the basic relationships of hydrogeology.

Morris Muskat first [4] refined Darcy's equation for a single-phase flow by including viscosity in the single (fluid) phase equation of Darcy. It can be understood that viscous fluids have more difficulty permeating through a porous medium than less viscous fluids. This change made it suitable for researchers in the petroleum industry. Based on experimental results by his colleagues Wyckoff and Botset, Muskat and Meres also generalized Darcy's law to cover a multiphase flow of water, oil and gas in the porous medium of a petroleum reservoir. The generalized multiphase flow equations by Muskat and others provide the analytical foundation for reservoir engineering that exists to this day.

1.2 General context

1.2.1 Definition of Darcy's law

Darcy's law is an equation that describes the flow of a fluid through a porous medium. The law was formulated by Henry Darcy based on results of experiments [2] on the flow of water through beds of sand, forming the basis of hydrogeology, a branch of earth sciences. It is analogous to Ohm's law in electrostatics, linearly relating the volume flow rate of the fluid to the hydraulic head difference (which is often just proportional to the pressure difference) via the hydraulic conductivity.
The problem involves determining the velocity of fluid flow through a porous medium, based on the properties of the fluid, the properties of the porous medium, and the boundary conditions.

Darcy's laws describe the flow of fluid through a porous medium as being proportional to the pressure difference across the medium and inversely proportional to the resistance of the medium to fluid flow.

1.2.2 Definition of the problem

The problem of Darcy's laws is to determine the fluid flow rate Q through a porous medium as a function of the pressure difference ΔP across the medium, the distance L between two points in the medium, and the properties of the medium such as its porosity, permeability, and specific surface area. Darcy's laws allow for the mathematical modeling of fluid flow through porous media, with the Darcy equation given by:

$$Q = -k\frac{A \cdot \Delta P}{L} \tag{1.1}$$

Where Q is the flow rate, k is the permeability of the porous medium, A is the specific surface area of the porous medium, and $\frac{\Delta P}{L}$ is the pressure gradient across the porous medium.

This equation describes the fluid flow rate through the porous medium as a function of the medium's resistance to fluid flow, represented by permeability and specific surface area, as well as the pressure difference across the medium.

1.3 Objectives

The main objectives of this project is the numerical simulation of a two-phase flow model in a *heteregeneous* porous media with `Feel++`. A heterogeneous medium, in this context, refers to a domain characterized by a combination of distinct features. Notably, the media contains fractures which can significantly impact flow patterns. Additionally, there exists a variability in permeability, denoted by K, throughout the domain. This variability can arise from natural variations in grain size, sedimentation processes, or other geological factors. Simulating flows in such a complex environment presents unique challenges, requiring specialized numerical techniques and a deep understanding of both the physical processes and the computational tools.

We have considered intermediate sub-objectives in order to achieve the above objective, splitted as follows:

- understand the fundamental elements of porous media and the important physical quantities involved in two-phase flow modeling
- familiarize with a simpler model, the Richards' equation.
- solve the Richards equation with Feel++
- design 3D meshes
- design test cases in two or three dimensions

1.4 Tools

The work will be organised and coordinated via the following tools:

- **Slack** : for internal communication.
- **Github** : for code sharing, version management and issue tracking.
- **Python** : for the implementation of the numerical model.
- **Gmsh** : for mesh generation
- **CFPDES from Feel++** : for the numerical simulation of PDE models with the finite element method
- **Jupyter notebook** : for internal communications,
- **Paraview** : for visualization.

Chapter 2

Models

2.1 Elements from porous media

Porous media are materials that contain void spaces or pores. Examples of porous media include soils, rocks, and biological tissues. Porous media have important applications in various fields such as hydrology, geology, petroleum engineering, and biomedicine.

In hydrology and soil physics, porous media are used to model the flow of water through soils and rocks. The properties of porous media, such as porosity and permeability, play a crucial role in determining the flow behavior of fluids through these materials.

The following quantities are important in characterizing porous media:

- **Porosity**: It is the ratio of the volume of void spaces to the total volume of the porous medium. It is a measure of the amount of empty space in the material. Porosity is usually expressed as a percentage. It is usually denoted ϕ.
- **Permeability** : It is a measure of the ability of a porous medium to transmit fluids. It is defined as the volume of fluid that flows through a unit area of the porous medium per unit time under a unit hydraulic gradient. Permeability is usually expressed in units of m/s. It is usually denoted K.
- **Saturation** : It is the fraction of the total pore space that is filled with a particular fluid. In the context of groundwater flow, saturation refers to the fraction of the pore space that is filled with water. It is usually denoted $S(x,t)$.

2.2 The Two-Phase Flow Model

To model the flow of fluids through porous media, a two-phase flow model is often used. This model considers the flow of two immiscible fluids, such as water and air, through the void spaces of the porous medium.

The two-phase flow model consists of a set of partial differential equations that describe the conservation of mass and momentum of each fluid phase. These equations are coupled through the capillary pressure-saturation relationship, which relates the pressure difference between the two phases to the degree of saturation of the porous medium.

The two-phase flow model is the set of equations (2.1),(2.2),(2.3) written for a $air - water$ system, denoted with w and o.

- **Mass conservation equation:**

$$\begin{cases} \frac{\partial(\phi\rho_\omega S_\omega)}{\partial t} + div(\rho_\omega \overrightarrow{V}_\omega) = 0 \\ \frac{\partial(\phi\rho_0 S_0)}{\partial t} + div(\rho_0 \overrightarrow{V}_0) = 0 \end{cases} \tag{2.1}$$

- **Volume conservation**

$$\begin{cases} \overrightarrow{V}_\omega = -\frac{k_{r,\omega}(S_\omega)}{\mu_\omega} K(\nabla P_\omega - \rho_\omega \overrightarrow{g}) \\ \overrightarrow{V}_0 = -\frac{k_{r,0}(S_0)}{\mu_0} K(\nabla P_\omega + \nabla P_c(S_\omega) - \rho_0 \overrightarrow{g}) \end{cases} \tag{2.2}$$

Pore volume conservation:

$$S_\omega + S_0 = 1 \tag{2.3}$$

It is commonly used in the study of groundwater flow in unsaturated soils and rocks. In this context, the wetting phase is usually water and the non-wetting phase is air.

With:

- ϕ : the porosity of the porous medium.
- S_i : the saturation of the phase i.
- V : the velocity of the phase i.
- K : the permeability of the porous medium.
- P_i : the pressure of the phase i.
- P_c : the capillary pressure, as a function of the water saturation S_w.
- $k_{r,i}$: the relative permeability of the phase i.
- μ_i : the viscosity of the phase i.
- ρ_i : the density of the phase i.
- $\overrightarrow{g}$: the gravity vector.

2.3 The Richards' model

The Richards model is commonly used to simulate water flow in soils. It describes the flow of water through a porous soil as a function of pressure, permeability, and soil density.

In the context of our study, we have chosen to work with the Richards model instead of the two-phase flow model, as we are focusing on the flow of water in soils. This allows us to simplify the model and focus on the parameters that are most relevant for our study, such as soil permeability and density.

In some cases, the two-phase model is too complicated to deal because for numerical reasons.

The Richards model is a widely used equation for modeling water flow in unsaturated soils and rocks. It is derived from the two-phase flow model by assuming that the non-wetting phase is air and that its density is negligible compared to the density of the wetting phase.

Richards equation is a classic nonlinear parabolic equation to describe flow occurring in the un- saturated zone of an aquifer:

$$\partial_t \theta(\psi) - \nabla.(K(\psi)\nabla\psi + z) = 0 \tag{2.4}$$

- θ : the water content of the porous medium. It is the volume of water contained in the medium per unit of total volume (water + solid material).
- ψ : the water pressure in the porous medium, also called the pressure head. It is defined as the difference in pressure between the water pressure and the atmospheric pressure. It is usually expressed in units of length, such as meters or centimeters.
- K : the hydraulic conductivity of the porous medium. It represents the ability of the medium to allow water to flow.

The Richards equation is difficult to solve analytically. It is often solved numerically using finite difference or finite element methods. The equation is widely used in hydrology and soil-physics to model water flow in unsaturated soils and rocks, and it has also been used to model groundwater flow in sandy beaches.

We will prefer the following formulation taken from the article [Brenner et Cancès] :

Denote by Ω some bounded open subset of $R^d (d \leq 3)$ representing the porous medium (in the sequel, Ω will be supposed to be polyhedral for meshing purpose), by $T > 0$ a finite time horizon, and by $Q := \Omega \times [0, T)$ the corresponding space-time cylinder. We are interested in finding a saturation profile $s : Q \to [0, 1]$ and a water pressure $p : Q \to R$ such that

$$\partial_t s - \nabla.(\lambda(s)(\nabla p - g)) = 0 \tag{2.5}$$

where the the mobility function $\lambda : [0, 1] \to R_+$ is nondecreasing C^2 function that satisfies $\lambda(s \leq 0) = 0$ and $\lambda(s > 0) > 0$, and where $g \in R^d$ is the gravity vector. In order to ease the reading, we have set the porosity equal to 1 in the equation and neglected the residual saturation. The pressure and the water content are supposed to be linked by some monotone relation

$$s = S(p) \text{ a.e. in } Q \tag{2.6}$$

where S is a non-decreasing function from R to $[0, 1]$. In what follows, we assume that $S(p) = 1$ for all $p \geq 0$, that corresponds to assuming that the porous medium is water wet, and that $S \in L^1(R_-)$, implying in particular that $\lim_{p->-\infty} S(p) = 0$.

Typical behaviors of λ and S are depicted in the Figure

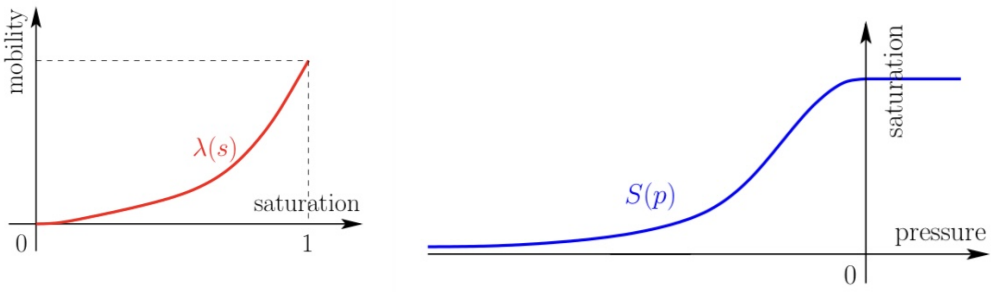


Figure 2.1: Mobility function $\lambda(S)$ and saturation function $S(p)$ from [10]

The mobility function $\lambda : [0,1] -> R_+$ is nondecreasing and satisfies $\lambda(0) = 0$. The saturation function $S : R- > [0,1]$ is nondecreasing, constant equal to 1 on R_+ and increasing on R_-.

2.4 From Two-phase flow model to the Richards model

In this section, we show how to obtain the Richards' model ... as a simplification (under precise assumptions) of the two-phase flow model .... This development will take up the ideas from [10, pp. 3].

We start with mass conservation principle applied on a two-phase flow in porous medium:

$$\begin{cases} \partial_t(\rho_\alpha \phi S_\alpha) + \nabla.(\rho_\alpha q_\alpha) = 0 \\ q_\alpha = -\frac{K_i k_{r,\alpha}(S_\alpha)}{\mu_\alpha}(\nabla(p_\alpha + \rho_\alpha g z)) \\ \alpha \in air, water \end{cases} \tag{2.7}$$

where ψ denotes the porosity, S the saturation, ρ_α the density (kg/m^3), q_α the Darcy velocity (m/s), K_i the tensor of intrinsic permeability (m^2), $k_{r,\alpha}$ the relative permeability (m^2), μ_α the dynamic viscosity $(Pa\dot{s})$, p_α is pressure (Pa) and g is gravitational acceleration (m/s^2). Here, q_α is modelled as an extension of Darcy's law to diphasic system, sometimes called Darcy-Buckingham law. It was initially based on the results of experiments but some theoretical derivations were undertaken via homogenization techniques such as in Whitaker. In any case, underlying assumptions are made about the nature of flows and porous media, in particular, related to Stokes flow. Additionally, two closure conditions go along with equations:

$$\begin{cases} S_{air} + S_{water} = 1 \\ p_{air} - p_{water} = P_c(S_{water}) \end{cases} \tag{2.8}$$

P_c is capillary pressure (Pa), an invertible function known from experiment. Then, the main hypothesis for Richards equation is used to eliminate the equation for air in Equations (2.7). Indeed, air viscosity is considered about 55 times smaller than the water viscosity [**?**] in that case, resulting in the same factor for mobility if relative permeabilities are similar for both fluids. Consequently, pressure gradients balance faster in air than in water phase. Moreover, it is assumed that the air phase is connected continuously at every points with the atmosphere, and patm is known up to a constant so one can write $p_{atm} = 0$ for convenience and thus:

$$P_c = p_{air} - p_{water} = p_{atm} - p_{water} = -p_{water} \tag{2.9}$$

Now, for the sake of simplicity, we omit the subscript water. Some additional assumptions are usually taken into account that is to say the solid skeleton is not deformable, water density is homogeneous and water is incompressible. This leads to write:

$$\phi \partial_t S - \nabla.(\frac{K_i k_r(S)}{\mu}\nabla p + \rho g z) = 0 \tag{2.10}$$

By introducing $\theta = \phi S = K_i k_r(S)\frac{\rho g}{\mu}$ and $\psi = \frac{p}{\rho g}$, we recover the original Richards equation (2.4). It is called the mixed formulation of Richards equation and present better numerical behaviour than water content formulation or head formulation because it is a conservative form,

defined for complete saturation and allowing heterogeneous soils[10]. In regards to our situation, Richards equation is a good choice because extensive experience are available in the community especially for its numerical resolution and its coupling with surface flows. Nevertheless, we have to be aware that any effect involving air phase could not be captured such as trapped air pockets. Moreover, fast dynamics due to incoming waves at the boundary for interface condition may cause some troubles with the validity of assumptions taken for Darcy- Buckingham law, in particular, the one for Stokes flow.

Chapter 3

Mesh generation

3.1 Fractured domain in geology

Numerical simulation in geology is a demanding task, especially when dealing with complex domains such as fractured environments. Accounting for fractures is essential for in-depth studies as they can significantly influence the hydraulic, thermal, and mechanical properties of a rocky medium. Thus, generating an appropriate mesh is crucial for obtaining accurate and reliable simulation results.

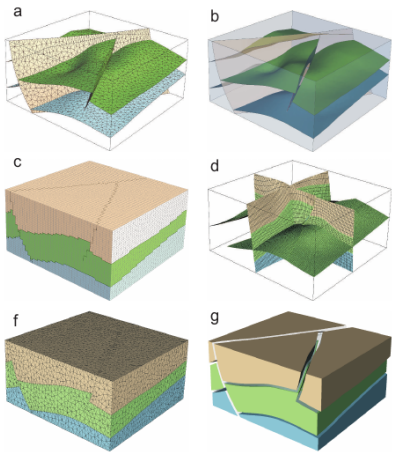


Figure 3.1: Example of a fractured rocky domain.

Figure 3.1 provides an example of what fractures in a rocky environment might look like. While this illustration might not exactly match the mesh we will generate using `Gmsh`, it gives an insight into the complexity and variability of structures we aim to represent. In the subsequent sections, we will detail the methodology embraced to create a mesh true to such an environment.

3.2 Presentation of `GMSH`

`Gmsh` is a free 3D finite element grid generator with a built-in `CAD` engine and post-processor.

Its design goal is to provide a fast, light and user-friendly meshing tool with parametric input and advanced visualization capabilities.

`Gmsh` is a `CAD` software that helps us to create the geometry of the domain. It is a free software that can be downloaded from the official website `http://gmsh.info/`. It is a very powerful tool that allows us to create complex geometries. It is also possible to create the geometry using other `CAD` software and then import it into `Gmsh`. In this work, we will use `Gmsh` to create the geometry of the domain. We will also use it to create the mesh of the domain. The `mesh` is the discretization of the domain into small elements. The mesh is a very important part of the simulation. The accuracy of the simulation depends on the quality of the mesh. The mesh should be fine enough to capture the details of the solution, but not too fine to avoid unnecessary computations. `Gmsh` allows us to create a mesh with different sizes. We can create a mesh with a uniform size, or we can create a mesh with different sizes in different parts of the domain. In this work, we will create a mesh with a uniform size.

3.3 Designing fractured domains

In the scope of this project, our aim is to create a complex three-dimensional model of a cube. This cube isn't a simple solid block, but rather an object containing multiple fractures, akin to the faults that can be found in a piece of rock or an ice block. These fractures are important because they can influence how the cube behaves when forces are applied to it. For instance, water can flow through these fractures or they can open further if enough pressure is applied.

The cube and each fracture are designed in detail, starting from specific points that are connected by lines to form surfaces. These surfaces are then transformed into a three-dimensional model.

Each type of fracture is treated as a distinct element in our model. This means we can study how each of them behaves individually. In addition, each face of the cube is also treated separately, allowing us to examine how different factors, such as pressure or movement, can affect different parts of the cube.

Once we have created our three-dimensional model, we can use it to simulate different conditions and see how our cube reacts. This could help us understand, for instance, how a rock block might react to an earthquake, or how ice might crack under the effects of global warming.

3.3.1 Cube with one fracture in the middle

The first mesh is a cube with a single fracture in the middle. This fracture is a simple line that goes from one side of the cube to the other. The fracture is placed in the middle of the cube, so it divides the cube into two halves.

- The geometric kernel :

```
{
  SetFactory("OpenCASCADE");
}
```

  This line sets the geometric kernel to OpenCASCADE, which is a software development platform that provides services for 3D surface and solid modeling.

- variable $'h'$:

```
      {
        h = 0.1;
      }
```

Here, a variable h is set, which will be used later as the fourth parameter in the Point command. This fourth parameter specifies the target mesh size near the given point.

- `Points` :

```
      {
        // Points for the top part of the cube
        Point(1) = {0.6, 0, 0.7, h};
        Point(2) = {0.6, 0, 0, h};
        Point(3) = {0.6, 1, 0, h};
        Point(4) = {0.6, 1, 0.7, h};
        Point(5) = {0, 0, 1, h};
        Point(6) = {1, 0, 1, h};
        Point(7) = {1, 1, 1, h};
        Point(8) = {0, 1, 1, h};

        // Points for the fracture in the middle
        Point(9) = {0.5, 0, 0.7, h};
        Point(10) = {0.5, 0, 0, h};
        Point(11) = {0.5, 1, 0, h};
        Point(12) = {0.5, 1, 0.7, h};

        // Points for the bottom part of the cube
        Point(13) = {0, 0, 0, h};
        Point(14) = {1, 0, 0, h};
        Point(15) = {1, 1, 0, h};
        Point(16) = {0, 1, 0, h};
      }
```

These lines define the points that will be used to create the cube. Each point is defined by its coordinates and the target mesh size near the point.

- `Lines` :

```
      {

        // Lines for the top part of the cube
        Line(1) = {1, 2};
        Line(2) = {2, 3};
        Line(3) = {3, 4};
        Line(4) = {4, 1};
        Line(5) = {5, 6};
        Line(6) = {6, 7};
        Line(7) = {8, 7};
        Line(8) = {8, 5};
        Line(9) = {5, 13};
        Line(10) = {6, 14};
        Line(11) = {7, 15};
        Line(12) = {8, 16};

        // Lines for the middle part of the cube
        Line(13) = {9, 10};
        Line(14) = {10, 11};
        Line(15) = {11, 12};
        Line(16) = {12, 9};
        Line(17) = {1, 9};
        Line(18) = {2, 10};
```

```
        Line(19) = {3, 11};
        Line(20) = {4, 12};

        // Lines for the bottom part of the cube
        Line(21) = {13, 10};
        Line(22) = {14, 15};
        Line(23) = {16, 11};
        Line(24) = {16, 13};
        Line(25) = {2, 14};
        Line(26) = {3, 15};
    }
```

The 'Line' command connects two previously defined points. The first parameter is the ID of the line, and the second parameter is a list of the IDs of the points that will be connected.

- Line Loop :

```
    {

        // Line Loops for the cube
        Line Loop(27) = {24, 9, -12, -8};//left
        Line Loop(28) = {10, 22, -11, -6};//right
        Line Loop(29) = {8, 5, -7, -6}; //top
        Line Loop(30) = {9, 21, -13, 17, 1, 25, -10, -5};//front
        Line Loop(31) = {12, 23, -15, 20, 3, 26, -11, -7}; //back
        Line Loop(32) = {24, 21, -14, -23};//botto1
        Line Loop(33) = {2, 25, -22, -26};//botto2

        // Line Loops for the middle part of the cube
        Line Loop(34) = {14, 13, -16, -15};//l
        Line Loop(35) = {-17, 13, -1, 18};//f
        Line Loop(36) = {2, 1, -4, -3};//r
        Line Loop(37) = {17, 16, -4, -20};//t
        Line Loop(38) = {18, 14, -2, -19};//bo
        Line Loop(39) = {-20, 15, -3, 19};//ba
    }
```

The 'Line Loop' command creates a loop of lines, which is a necessary step before creating a surface. Command creates a closed loop from a list of lines. The first parameter is the ID of the
loop, and the second parameter is a list of the IDs of the lines that will be used to create the loop. The order of the lines is important, as it determines the orientation of the loop.

- Surface Plane :

```
    {

        // Surfaces for the top part of the cube
        Plane Surface(40) = {27};
        Plane Surface(41) = {28};
        Plane Surface(42) = {29};
        Plane Surface(43) = {30};
        Plane Surface(44) = {31};
        Plane Surface(45) = {32};

        // Surfaces for the middle part of the cube
        Plane Surface(46) = {33};
        Plane Surface(47) = {34};
        Plane Surface(48) = {35};
```

```
    Plane Surface(49) = {36};
    Plane Surface(50) = {37};
    Plane Surface(51) = {38};
    Plane Surface(52) = {39};
  }
```

The `'Plane Surface'` command creates a surface from a list of line loops. The first parameter is the ID of the surface, and the second parameter is a list of the IDs of the line loops that will be used to create the surface. The order of the line loops is important, as it determines the orientation of the surface.

- `Surface Loop` and `Volume` :

```
  {
    // Volume definitions for the top, middle and bottom of the cube

    Surface Loop(53) = {40, 41, 42, 43, 44, 45, 46, 47, 49, 50};
    Volume(54) = {53};


    Surface Loop(55) = {47, 48, 49, 50, 51, 52};
    Volume(56) = {55};
  }
```

The `'Surface Loop'` command creates a loop of surfaces, which is a necessary step before creating a volume. The first parameter is the ID of the loop, and the second parameter is a list of the IDs of the surfaces that will be used to create the loop. The order of the surfaces is important, as it determines the orientation of the loop.
The `'Volume'` creates a 3D volume from a Surface Loop using the ID of the volume and the ID of the Surface Loop as parameters.

- `Physical Surface` :

```
    {
      // Physical groups
      Physical Surface("Top") = {42};
      Physical Surface("Bottom") = {45, 46, 51};
      Physical Surface("Front") = {43, 48};
      Physical Surface("Back") = {44, 52};
      Physical Surface("Left") = {40};
      Physical Surface("Right") = {41};
      Physical Surface("MiddleTop") = {50};
      Physical Surface("MiddleBottom") = {51};
      Physical Surface("MiddleLeft") = {47};
      Physical Surface("MiddleRight") = {49};

    }
```

The `'Physical Surface'` command creates a physical surface from a list of surfaces. The first parameter is the ID of the physical surface, and the second parameter is a list of the IDs of the surfaces that will be used to create the physical surface.

- `Physical Volume` :

```
    {
      // Physical Volumes for the matrice
      Physical Volume ("Omega1") = {54}; // matrice
```

```
        // Physical Volumes for the fracture
        Physical Volume ("Omega2") = {56}; // fracture
    }
```

The `'Physical Volume'` command creates a physical volume from a list of volumes. The first parameter is the ID of the physical volume, and the second parameter is a list of the IDs of the volumes that will be used to create the physical volume.

Here is the final result of the Gmsh script:

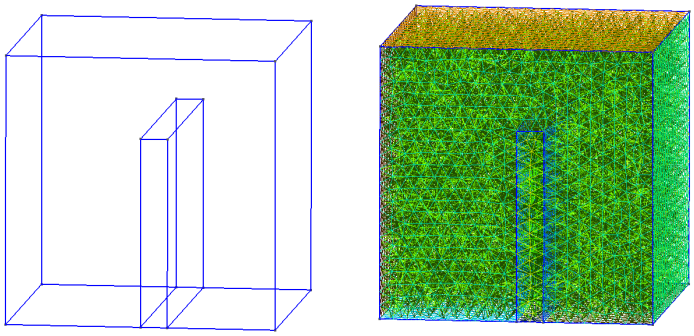


Figure 3.2: *Mesh in 3D with 1 fracture see [11]*

3.3.2 Mesh with 3 obliques fractures

This `Gmsh` script (see `code/geo/rect_multi_petit.geo` in the GH repo [11]) defines a 3D cube with three types of fractures: vertical, horizontal, and oblique. Each fracture is created by defining points, lines, and surfaces within the 3D space. After constructing the cube and fractures, the script defines volumes for each entity. Lastly, it creates `"Physical" surfaces and volumes` to allow for identification of specific areas in subsequent analyses. We see the result of the `Gmsh` script in the figure below:

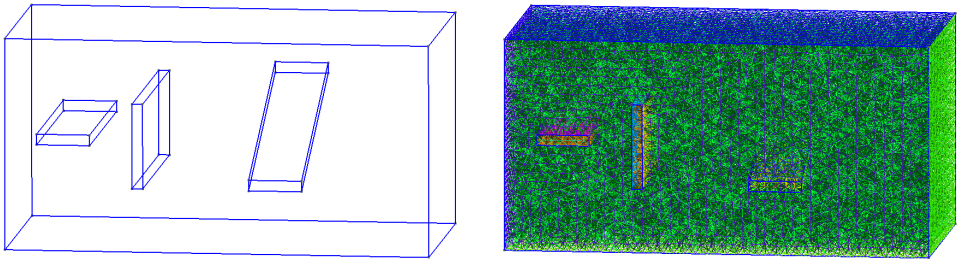


Figure 3.3: *Mesh in 3D with 3 fractures see [11]*

3.3.3 Mesh with a single horizontal fracture

We also did another example of a mesh with fractures. To see more details about this mesh, you can go to the following link: [11].
Here is one of the results of the Gmsh script:

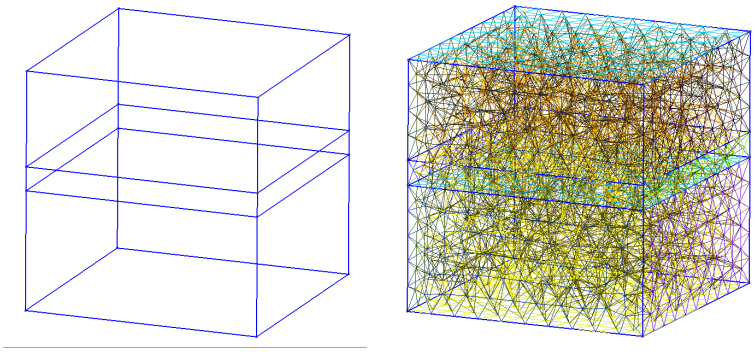


Figure 3.4: *Mesh in 3D with 1 fracture horizontal see [11]*

3.4 Mesh Generation with Python

The `Gmsh` Python library provides a programmatic interface to access all functionalities of `Gmsh`. With this interface, you can automate the creation, manipulation, and visualization of complex meshes.

This specific Python script we are about to look at utilizes `Gmsh's Python API` to generate a meshed 3D terrain. The terrain is defined by a set of points, which are then meshed to create a surface. Additional features of the surface, such as curves and angles, are also defined within the script. The result is a complex 3D mesh that represents the terrain and can be used for further analysis or simulations.

- **Initialization and setting up the model:**

```
{
  import gmsh
  import sys
  import math

  gmsh.initialize()
  gmsh.model.add("terrainfracture")
}
```

This part imports the necessary libraries, initializes the `Gmsh API`, and adds a model named `"terrainfracture"` to the `Gmsh` instance.

- **Preparing the data:**

```
{
  N = 100
  def tag(i, j):
  return (N + 1) * i + j + 1
}
```

Here, a helper function $tag(i, j)$ is defined to return a node tag given two indices i and j. Also, the value of N, the number of input data points for meshing, is prepared.

- **Creation of the mesh node coordinates and tags:**

```
    {
      coords = []
      nodes = []
      tris = []
      lin = [[], [], [], []]
      pnt = [tag(0, 0), tag(N, 0), tag(N, N), tag(0, N)]

      for i in range(N + 1):
      for j in range(N + 1):
      nodes.append(tag(i, j))
      coords.extend([
      float(i) / N,
      float(j) / N, 0.05 * math.sin(10 *
      float(i + j) / N)])
    }
```

Here, the script is creating the coordinates and tags for all the nodes. It's also creating the connectivity of the triangle elements on the terrain surface, the connectivity of the line elements on the 4 boundaries, and the connectivity of the point elements at the 4 corners.

- **Adding the nodes to the mesh :**

```
    {
      gmsh.model.mesh.addNodes(2, 1, nodes, coords)
    }
```

This part adds the nodes to the mesh. The first argument is the dimension of the mesh, the second argument is the tag of the mesh, the third argument is the list of node tags, and the fourth argument is the list of node coordinates.

- **Adding elements to the mesh :**

```
    {
      for i in range(4):
      gmsh.model.mesh.addElementsByType(i + 1,
      15, [], [pnt[i]])
      gmsh.model.mesh.addElementsByType(i + 1,
      1, [], lin[i])
      gmsh.model.mesh.addElementsByType(1, 2, [], tris)
    }
```

Here, the script adds point elements at the 4 points, line elements on the 4 curves, and triangle elements on the surface.

- **Reclassification of nodes and creating geometry :**

```
    {
      gmsh.model.mesh.reclassifyNodes()
      gmsh.model.mesh.createGeometry()
    }
```

This part reclassifies the nodes on the curves and points, and creates geometry for the discrete curves and surfaces.

- **Creating points, lines, curves, surfaces, and a volume to define the discrete mesh size :**

```
{
  p1 = gmsh.model.geo.addPoint(0, 0, -0.5)
  p2 = gmsh.model.geo.addPoint(1, 0, -0.5)
  ...
  v65 = gmsh.model.geo.addVolume([sl1])
  gmsh.model.geo.synchronize()
}
```

  In this part, the script creates points, lines, curves, surfaces, and a volume using the `Gmsh` geometry API.

- **Setting the mesh size and generating the mesh:**

```
{
  gmsh.option.setNumber('Mesh.MeshSizeMin', 0.05)
  gmsh.option.setNumber('Mesh.MeshSizeMax', 0.05)

  gmsh.model.mesh.generate(3)
  gmsh.write('terrainfracture.msh')
}
```

  Here, the script sets the minimum and maximum mesh size, generates the 3D mesh, then writes the mesh to a file named `'terrainfracture.msh'`.

- **Launching the Gmsh GUI to view the mesh:**

```
{
  if '-nopopup' not in sys.argv:
  gmsh.fltk.run()
  gmsh.finalize()
}
```

  Finally, if the `-nopopup` argument is not given when running the script, the Gmsh GUI is launched to visualize the mesh. The script then finishes up by calling `gmsh.finalize()`.

For a more in-depth understanding of the code and the specific details of the meshing utilized in this work, please refer to the Python script titled on [12]. This script holds the instructions for generating a complex terrain with multiple fractures, including a vertical one intersecting two horizontal fractures, all done using the `Gmsh library`.

The 3D mesh generated by the script is shown like so:

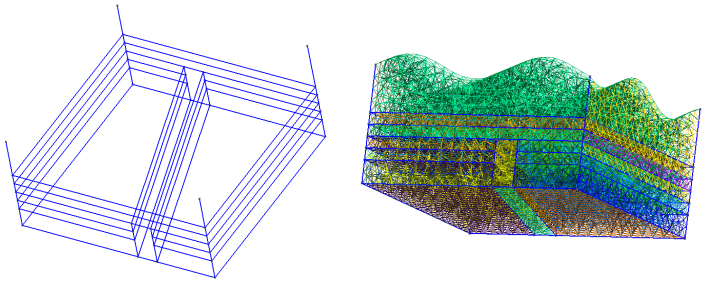


Figure 3.5: *Mesh in 3D with 6 fractures see [11]*

Chapter 4

Tests case with Richards' Model

4.1 Introduction

The Richards equation is a modification of the two phase flow equation, which is the standard model for water flow in a saturated porous medium. The Richards equation extends the two phase flow equation to include the case of an unsaturated porous medium, where air and water can coexist, with water moving under the effect of gravity and capillary forces.

We chose to examine the Richards case in our study for several reasons. First, it is a fundamental case in the study of flows in porous media, providing valuable insights into the basic mechanisms governing flow. Furthermore, despite its apparent simplicity, the Richards equation captures a large number of complex behaviors that occur in real porous media, including the effects of hysteresis and non-linearity.

In this work, we will study a discretized version of the Richards equation. We will present the mathematical model, discuss the choices of functions representing the properties of the porous medium, and finally present the results of our numerical simulations.

4.2 Mathematical model

We consider the equations :

$$\partial_t s - \nabla.(\lambda(s)(\nabla p - g)) = 0. \tag{4.1}$$

$$s = S(p) \tag{4.2}$$

Where s is the saturation, p is the pressure, g is the gravity, λ is the mobility and S is the saturation function.

For our numerical simulations, and from the plots of $\lambda(s)$ and $S(p)$, we have selected functions for λ and S. Here, $S(p)$ is a simplified form of the *Brooks-Corey* function, a common model in studying flow in porous media, let $p_b < 0$ and $\beta > 0$, in *Brooks-Corey* model the saturation and the mobility functions are given by :

$$S(p) = \begin{cases} (\frac{p}{p_b})^{-\beta} & \text{if} \quad p < p_b \\ 1 \quad \text{if} \quad p \geq p_b \end{cases} \tag{4.3}$$

And

$$\lambda(s) = s^m \quad \text{with} \quad m = 1, 2, 4$$

.
However, because of the *Brooks-Corey* function's abrupt changes and lack of smoothness, in this section of our study, we have substituted the traditionally used Brooks-Corey function with two different types of sigmoid functions for the simulation of Richards' problems. Our selection of these functions was guided by their properties and relevance to this particular application.

- Brooks-Corey Function : The Brooks-Corey function has traditionally been used in hydrological modeling to describe the relationship between soil moisture and pressure head.
- Standard Sigmoid Function : The standard sigmoid function, defined as $S(p) = \frac{1}{1+e^{-\alpha x+\beta}}$, is widely utilized in various fields due to its property of having a smooth gradient and producing an output that is pleasingly bounded between 0 *and* 1.
- Hyperbolic Sigmoid Function : The hyperbolic sigmoid function, or $S(p) = tanh(ax + b)$, offers more flexibility to adjust the steepness of the transition from 0 to 1 around the threshold. However, in our case, its performance was not as good as that of the standard sigmoid function.

However, in our case, we found the standard sigmoid function outperformed this model.

$$S(p) = \begin{cases} (\frac{p}{p_b})^{\frac{1}{\eta}} & \text{if } p \leq p_b \\ 1 & \text{if } p > p_b \end{cases} \quad \text{and} \quad S(p) = \frac{1}{1+e^{-\alpha p+\beta}} \quad \text{or} \quad S(p) = tanh(ap + b) \tag{4.4}$$

In these functions, m is a member of the set $1, 2, 4$, and "α, β, a, b" are a positive parameters that needs to be set. The function S is a sigmoid function. It's an increasing function from R to $[0, 1]$, equalling 1 on R_- and approaching 0 as p nears negative infinity. The function "λ" increases from $[0, 1]$ to R_-, equaling 0 at 0. Hence, we've ensured the desired properties, with "λ" being C^2 and S being C^1.

As shown in the figure, the standard sigmoid function $S(p) = \frac{1}{1+e^{-\alpha x+\beta}}$ provides a better fit to the data than the other two functions, and thus has been chosen for our subsequent simulations.

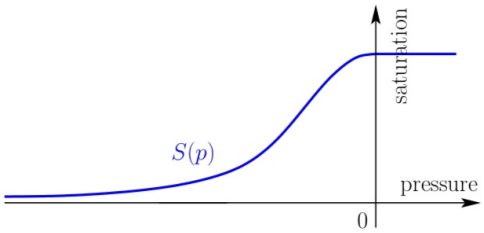


Figure 4.1: $S(p)$ *function taking from* [10]

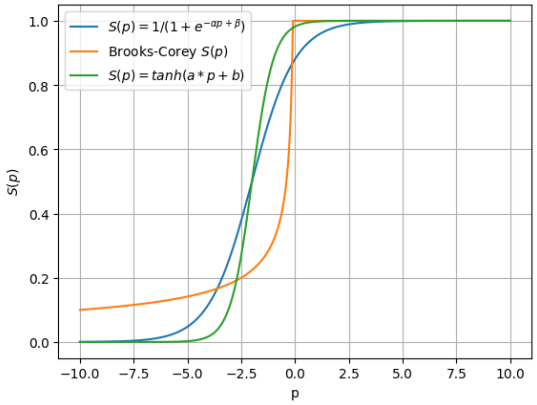


Figure 4.2: *Approximation of Brooks-Corey's function with a standard sigmoid function and a hyperbolic sigmoid function*

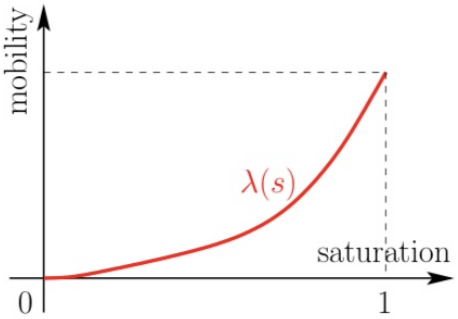


Figure 4.3: $\lambda(s)$*: mobility function taking from* [10]

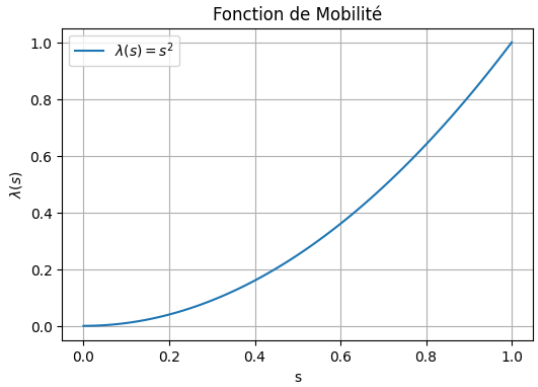


Figure 4.4: *Approximation of* $\lambda(s)$ *function with polynomial functions*

Different scenarios have been modeled in "CFPDES". We selected the case with the mobility function $\lambda(s) = s^2$ and the saturation function $S(p) = \frac{1}{1+e^{-p-2}}$ with $\beta = -2$, $\alpha = 1$ and $m = 2$.

4.3 Checking CFPDES with Richards' Model

CFPDEs (Configurable Finite Element PDE solvers) is a suite of tools for solving partial differential equations (PDEs) based on the finite element method. As the name suggests, it is highly configurable, meaning you can customize it to solve a wide variety of different PDE problems.

The unique thing about CFPDEs is that it allows you to specify the PDE problem to be solved using a configuration file format, typically in JSON . These configuration files define the mathematical models, boundary and initial conditions, parameters, domain, and other necessary information to describe the problem. This makes CFPDEs very flexible and adaptable to a wide range of scientific and engineering problems.

The general form of the equation is as follows:

$$d(u)\partial_t u + \nabla.(-c(u)\nabla u - \alpha(u)u + \gamma(u)) + \beta(u)\nabla u + a(u)u = f(u) \tag{4.5}$$

In this equation:

- u represents the unknown quantity that we are trying to solve for. The equation describes the evolution of u over time, denoted by $\partial_t u$, as well as its spatial variation, represented by the gradient operator ∇.
- The first term $d(u)\partial_t u$ accounts for the diffusion of u over time, where $d(u)$ is the diffusion coefficient that may vary depending on the value of u. This term describes how u spreads or diffuses through the medium.
- The second term $\nabla.(-c(u)\nabla u - \alpha(u)u + \gamma(u))$ represents the convection, absorption, and reaction effects on u.

- Here, $c(u)$ is the convection coefficient, which characterizes the transport of u due to the flow of a surrounding medium.
- $\alpha(u)$ is the absorption coefficient, which determines the amount of u absorbed or attenuated in the medium.
- $\gamma(u)$ represents the reaction coefficient, accounting for any chemical reactions or processes involving u that may occur. The negative sign in front of this term indicates that these effects act against the diffusion of u.
- The third term $\beta(u)\nabla u$ represents the gradient of u weighted by the coefficient $\beta(u)$. This term captures any additional influence on the spatial variation of u due to its own gradient.
- The fourth term $a(u)u$ accounts for the self-interaction of u. Here, $a(u)$ represents the coefficient of self-interaction, which determines how u interacts with itself and influences its own behavior.
- The right-hand side $f(u)$ represents a source term, which represents any external factors or inputs that affect the evolution of u over time. This term could include, for example, external forces, heat sources, or any other factors that contribute to the dynamics of u.

Comparing the model equation with the general form of the equation (4.1)-(4.2), we get the following identifications:

For the equation (4.1) of the model, the suitable choice is:

- $u = s$
- $d(u) = 1$
- $\gamma(u) = -\lambda(s)(\nabla p - \overrightarrow{g})$
- $c(u) = \alpha(u) = \beta(u) = a(u) = f(u) = 0$

In the equation (4.2) of the model, we have:

- $u = p$
- $d(u) = \gamma(u) = c(u) = \alpha(u) = \beta(u) = a(u) = 0$
- $f(u) = s - S(p)$

4.3.1 Tools used to simulate the model

Understanding how to simulate a system using `Gmsh, Feel++, and the JSON, CFG, and GEO files` involves understanding the role each of these components play.

- **Gmsh (GEO files)**: Gmsh is used to create the geometry of the physical problem you are trying to simulate. The GEO file specifically describes the geometry of your model.
- **Feel++ (CFG and JSON files)**: `Feel++` is a `C++` library used for the numerical solution of partial differential equations (PDE).
  - The `CFG` file is a configuration file that specifies the settings `Feel++` should use during its run.
  - The `JSON` file is used to configure the numerical simulation, which includes setting the equations, boundary and initial conditions, and materials used in the simulation.

4.3.2 Simulation Workflow

1. The **GEO file** is used by Gmsh to generate a mesh that represents the domain of the problem. This mesh discretizes the domain, breaking it down into simpler shapes (like triangles or tetrahedrons) over which the PDE can be solved.

2. The **JSON file** is then used to set the parameters, equations, materials, and boundary and initial conditions for the simulation. In the context of your work,this file is used to configure a numerical simulation which solves Richards' equations, a partial differential equation used to model the flow of water in unsaturated soil.

   - `"Name" and "ShortName":` This is the designation and abbreviation of your simulation model

```
{
  "Name": "Richards",
  "ShortName": "Richards"
}
```

   - `"Model":` This is the block where the mathematical equations of your simulation model are defined. In your case, the `"cfpdes"` model is used to describe two types of equations: the saturation equation and the pressure equation. For each of these equations, a specific setup is defined, including the unknown and the coefficients.

```
{
  "Models":
  {
    "cfpdes":{
      "equations":["saturation","pressure"]
    },
    "saturation":{
      "setup":{
        "unknown":{
          "basis":"Pch1",
          "name":"S",
          "symbol":"s"
        },
        "coefficients":{
          "d": "1.0",
          "gamma": "{-saturation_s^m * materials_permkx*
            (pressure_grad_p_0 - gx),
            -saturation_s^m * materials_permky*
            (pressure_grad_p_1 - gy),
            -saturation_s^m* materials_permkz *
            (pressure_grad_p_2 - gz)}:
          saturation_s:pressure_grad_p_0:
          pressure_grad_p_1:pressure_grad_p_2:
          gx:gy:gz:materials_permkx:
          materials_permky:materials_permkz:m"
        }
      }
    },
    "pressure":{
      "setup":{
        "unknown":{
          "basis":"Pch1",
          "name":"P",
          "symbol":"p"
        },
```

```
            "coefficients":{
              "f":"{saturation_s - 1.0/(1 + exp(-pressure_p))}
              :saturation_s:pressure_p"
            }
          }
        }
      }
    }
```

- `"Parameters"`: This block contains the parameters of your simulation model, including values for gravity `(gx, gy, gz)`, an exponent for the saturation equation, the stop time for the simulation, and the initial saturation.

```
    {
      "Parameters": {
        "gx": 0,
        "gy": 0,
        "gz": -1,
        "m" : 2,
        "stop": 0.5,
        "sinit": 0.1
      }
    }
```

- `"Meshes"`: This block provides information related to the mesh that is used in the simulation. In this case, a .geo file is imported and a mesh element size is defined.

```
    {
      "Meshes":
      {
        "cfpdes":
        {
          "Import":
          {
            "filename":"$cfgdir/geo/cube_1_frac_vert.geo",
            "hsize":0.08
          }
        }
      }
    }
```

- `"Materials"`: This block describes the properties of the materials used in the simulation. It identifies two material markers, each with specific permeabilities in the x, y, and z directions.

```
    {
      "Materials":
      {
        "mymat1":
        {
          "markers":"Omega1",
          "permkx":"0",
          "permky":"0",
          "permkz":"1"
        },
        "mymat2":
        {
          "markers":"Omega2",
          "permkx":"0",
          "permky":"0",
          "permkz":"0.2"
```

```
        }
      }
    }
```

- `"BoundaryConditions":` This block specifies the boundary conditions for the pressure and saturation equations. For each condition, a type is defined `(Dirichlet or Neumann)`, as well as corresponding markers (which describe where these conditions apply), and an expression which gives the boundary condition.

```
    {
      "BoundaryConditions":{
        "saturation": {
          "Dirichlet": {

            "mybc": {
              "markers": ["Top"],
              "expr": "stop:stop:stop"
            }
          },
          "Neumann": {
            "mybc": {
              "markers": ["Left", "Right", "Front", "Back"
              ,"MiddleLeft","MiddleRight"
              ,"Bottom"],
              "expr": "0"
            }
          }
        },
        "pressure": {
          "Neumann": {
            "mybc": {
              "markers": ["Left", "Right", "Front", "Back"
              ,"MiddleLeft","MiddleRight",
              "Bottom"],
              "expr": "0"
            }
          },
          "Dirichlet": {
            "mybc": {
              "markers": ["Top"],
              "expr": "-log(1/stop -1):stop:stop:stop"
            }
          }
        }
      }
    }
```

- `"InitialConditions":` This block specifies the initial conditions for pressure and saturation, defining for each condition a set of markers (indicating where these conditions apply) and an expression representing the initial state.

```
    {
      "InitialConditions":
      {
        "saturation":{
          "S":{
            "Expression": {
              "myic1": {
                "markers": "Omega1",
                "expr": "sinit:sinit:sinit:x:y:z"
              },
```

```
            "myic2": {
              "markers": "Omega2",
              "expr": "sinit:sinit:sinit:x:y:z"
            }
          }
        }
      },
      "pressure":{
        "P": {
          "Expression": {
            "myic1": {
              "markers": "Omega1",
              "expr": "-log(1/sinit -1):sinit:sinit"
            },
            "myic2": {
              "markers": "Omega2",
              "expr": "-log(1/sinit -1):sinit:sinit"
            }
          }
        }
      }
    }
```

- `"PostProcess":` This block is used to configure the post-processing of the simulation, specifically by defining which fields should be exported after the simulation ends.

```
    {
      "PostProcess":
      {
        "cfpdes":
        {
          "Exports":
          {
            "fields":["all"]
          }
        }
      }
    }
```

Each block of this `JSON` file corresponds to a particular step or aspect of the simulation configuration. By modifying the values in this file, you can change the behavior of the simulation, such as the materials used, the geometry of the mesh, the boundary and initial conditions, and the parameters of Richards equation. To see more of `JSON` file, run on [16].

3. **The CFG file** is finally read by `Feel++`, providing it with the necessary configuration for the simulation run. This includes the `JSON` file's location and any solver or runtime options. Let's go through each part of the `CFG` file:

   - `directory= richards:` This line specifies the directory where the simulation results should be saved. Here, it's set to a directory named `'richards'`.
   - `case.dimension=3:` This sets the dimension of the case to be solved. In this instance, it's a 3-dimensional problem.
   - `[cfpdes.saturation]:` This section pertains to the settings of the saturation part of the `"cfpdes"` model.

- `time-stepping=Theta:` This specifies the time-stepping method to be used in the saturation model. The Theta method is an implicit time-stepping method that is unconditionally stable and second-order accurate in time.
- `[cfpdes]:` This section pertains to the settings for the `"cfpdes"` model.
- `filename=$ cfgdir/richards_frac_multiple.json:` This line sets the location of the `JSON` file to be used in the simulation. Here, it's set to the path to `'richardsfracmultiple.json'` in the configuration directory.
- `solver=Newton:` This specifies the type of solver to be used for the simulation. Here, it's set to use the `Newton solver`.
- `verbose=1:` This sets the verbosity level of the output. A value of '1' means that more information will be output to the console during the simulation.
- `snes-monitor=1:` This option enables the monitoring of the `SNES` (Scalable Nonlinear Equations Solvers) iterations.
- `ksp-monitor=0:` This option disables the monitoring of the `KSP` (Krylov Subspace) iterations.
- `snes-rtol=1e-4` and `snes-stol=1e-4:` These options set the relative and absolute convergence tolerances for the `SNES` solver.
- `[ts]:` This section contains settings for the time-stepping of the entire simulation.
- `time-initial=0:` This sets the initial time of the simulation.
- `time-step=1e-2:` This sets the size of each time step in the simulation.
- `time-final=5.0:` This sets the final time of the simulation.
- `restart.at-last-save=true:` This option means that if the simulation is restarted, it will start from the last saved time step. We can see more on [15].

4. Once `Feel++` has this information, it can then carry out the simulation, solving the PDEs over the domain specified by the mesh. The results of the simulation can then be post-processed, with certain fields exported for further analysis as specified in the `JSON` file.

By understanding and modifying these three components, we can accurately model and simulate a wide variety of physical systems. Please note that all of these files should be in the correct syntax and accurately represent the problem at hand for the simulation to run properly.

4.4 `Feel++` with python

`Feel++` is a high-level programming framework for the numerical modeling and simulation of complex systems, designed for partial differential equations (PDEs). It provides a powerful and flexible interface for solving these equations using the finite element method. Although `Feel++` is primarily developed in `C++`, a Python interface has been created to make it more accessible, allowing users to leverage the ease of use of Python while benefiting from the intensive computing capabilities of `Feel++`.

The Python interface of `Feel++` is essentially a wrapper around the `C++` core of `Feel++`. It allows writing Python scripts that can interact with `Feel++` and use its features. This makes `Feel++` much more accessible to users who prefer to work in Python or who do not have a lot of

C++ experience. With the Python interface, users can, for example, define geometries, generate meshes, define PDEs, and solve these equations, all while writing Python code.

In the context of our work, using Feel++ with Python is particularly useful for defining and solving complex problems of flow simulation in fractured porous media. With the Python script and configuration files (.json and .cfg), we can define the system geometry, the material properties, the initial and boundary conditions, and the equations governing flow and transport. Once the problem is defined, Feel++ is used to solve the equations and generate the simulation results.

In summary, using Feel++ with Python offers a powerful and flexible way to solve complex numerical simulation problems, while leveraging the ease of use of Python. This simplifies the modeling of complex systems and allows to focus on the physical interpretation of the results, rather than the technical details of numerically solving the equations.

Here is the Python script used to solve the Richards equation with Feel++:

- **Importing Necessary Libraries:** The Python libraries necessary to run the script are imported. feelpp, feelpp.toolboxes.core and feelpp.toolboxes.cfpdes are Feel++ specific libraries that provide the functionalities for solving partial differential equations within the context of this simulation. *xvfbwrapper, pyvista and plotly.subplots* are used for data visualization.

```
{
  import sys
  import feelpp
  import feelpp.toolboxes.core as tb
  from feelpp.toolboxes.cfpdes import *
  from xvfbwrapper import Xvfb
  import pyvista as pv
  import os
  import pandas as pd
  import numpy as np
  import plotly.express as px
  from plotly.subplots import make_subplots
  import itertools
}
```

- **Setting up Feel++ Environment:** The code sets up a new Feel++ environment with options specific to coefficient-form partial differential equations (cfpdes). This environment provides the context for running the simulation.

```
{
  sys.argv = ["feelpp_cfpdes_richards"]
  e = feelpp.Environment(sys.argv,
  opts=tb.toolboxes_options("coefficient-
  form-pdes","cfpdes"),
  config=feelpp.globalRepository("cfpdes
  -richards"))

}
```

- **Definition of getMesh Function:** This function is used to generate a mesh from a .geo file using Gmsh, then load this mesh into Feel++. The mesh is a discrete representation of the problem space on which the equation will be solved. The mesh is returned by this function to be used in the simulation.

```
{
  def getMesh(filename,hsize=0.05,dim=3,verbose=False):
  """create mesh

  Args:
  filename (str): name of the file
  hsize (float): mesh size
  dim (int): dimension of the mesh
  verbose (bool): verbose mode
  """
  for ext in [".msh",".geo"]:
  f=os.path.splitext(filename)[0]+ext
  # print(f)
  if os.path.exists(f):
  os.remove(f)

  if not os.path.exists(f):
  # Generate .msh file from .geo file using Gmsh
  os.system(f"gmsh -{dim} {filename} -o {f}")

  if verbose:
  print(f"generate mesh {filename} with hsize={hsize} and
  dimension={dim}")
  mesh = feelpp.load(feelpp.mesh(dim=dim,realdim=dim),
  filename, hsize)
  return mesh
}
```

- **Definition of richards Function:** This function sets up and runs the simulation for the Richards problem. It starts by initializing the problem with the previously created mesh and loading the model properties from a `json` file. It then solves the problem, either in one go if the problem is stationary, or over time if the problem is non-stationary. The results are then exported for post-processing.

```
{
  # run the richards problem for 3D
  def richards(hsize, json, dim=3, verbose=False):
  if verbose:
  print(f"Solving the richards problem
  for hsize = {hsize}...")

  # Set initial time step
  feelpp.Environment.setConfigFile("../feel/richards.cfg")
  richards = cfpdes(dim=dim, keyword=f"cfpdes-{dim}d")
  richards.setMesh(getMesh(f"square3d.geo",hsize=hsize,dim=dim
  ,verbose=verbose))
  richards.setModelProperties(json)
  richards.init(buildModelAlgebraicFactory=True)
  richards.printAndSaveInfo()
  richards.startTimeStep()
  if richards.isStationary():
  richards.solve()
  richards.exportResults()
  else:
  while not richards.timeStepBase().isFinished():
  if richards.worldComm().isMasterRank():
  print("=========================================\n")
  print("time simulation: ", richards.time(), "s \n")
  print("=========================================\n")
  richards.solve()
```

```
    richards.exportResults()
    richards.updateTimeStep()
    measures = richards.postProcessMeasures().values()
    return measures
}
```

- **Retrieving Measures:** These measures can then be used for analysis and visualization.

```
{
    measures = richards.postProcessMeasures().values()
    return measures
}
```

Overall, this Python script uses the `Feel++` library to define and solve a specific partial differential equation problem (the Richards problem) on a mesh generated from *16.geo* file. The results from the simulation can then be post-processed and visualized.

This section of the script is defining a function to create a `JSON` object that represents the Richards equation problem in 3D.

- **Defining a Lambda Function:** A lambda function is a small anonymous function that is defined with a single expression. Here, richards_json is a function that takes *order, dim, name1, and name2* as arguments and returns a `JSON` object

```
{
    # Create a json file for the richards problem for 3D
    richards_json = lambda order,dim=3,name1="S",name2="P": {
      "Name": "Richards",
      "ShortName": "richards",
      "Models":
      {
        f"cfpdes-{dim}d":{
          "equations":["saturation","pressure"]
        },
        "saturation":{
          "setup":{
            "unknown":{
              "basis":f"Pch{order}",
              "name":f"{name1}",
              "symbol":"s"
            },
            "coefficients":{
              "d": "1.0",
              "gamma": "{-saturation_s^m *
                materials_permkx *
                (pressure_grad_p_0 - gx),
                -saturation_s^m *
                materials_permky *
                (pressure_grad_p_1 - gy),
                -saturation_s^m *
                materials_permkz *
                (pressure_grad_p_2 - gz)}:
              saturation_s:pressure_grad_p_0
              :pressure_grad_p_1:
              pressure_grad_p_2:gx:gy:gz:
              materials_permkx:
              materials_permky:
              aterials_permkz:m"
            }
          }
```

```
        },
        "pressure":{
          "setup":{
            "unknown":{
              "basis":f"Pch{order}",
              "name":f"{name2}",
              "symbol":"p"
            },
            "coefficients":{
              "f":"{saturation_s - materials_sigmoid}:
              saturation_s:materials_sigmoid"
            }
          }
        }
      }
    }
```

- `"PostProcess" Section:` This section defines how post-processing is conducted after a simulation or computation.
  - `"Exports" Section:` This section defines the fields that should be exported after the simulation ends. Here, we export all fields.
    - `"fields":["all"]:` Indicates that all the fields from the model or simulation will be exported.
    - `"expr":` This key defines expressions to be computed and added to the exported results.
    - `"S_exact": "materials_sigmoid:materials_sigmoid:x:y:z":` An expression associated with a material sigmoidal function, dependent on coordinates.
    - `"grad_S_exact":` This denotes the gradient of the exact function S. The expression is a function dependent on parameters $\alpha 1, \beta 1, pressure_p$, and coordinates x, y, and z.

```
      {
        "Exports":
        {
          "fields":["all"],
          "expr":{
            "S_exact": "materials_sigmoid:materials_sigmoid:
            x:y:z",
            "grad_S_exact": "{alpha1*exp(-alpha1*pressure_p
              - beta1)/(1+exp(-alpha1 *
              pressure_p - beta1))^2,alpha1
              * exp(-alpha1*pressure_p -beta1)
              /(1+exp(-alpha1*pressure_p -
              beta1))^2, alpha1*exp(-alpha1 *
              pressure_p - beta1)/(1 +
              exp(-alpha1 * pressure_p -
              beta1))^2}:alpha1:beta1:
            pressure_p:x:y:z",
          }
        }
      }
```

  - `"Measures" Section:`This section is devised to define measures or calculations performed on the results. - `"Norm":` A measure of a vector's magnitude, typically associated with norm.

- `"saturation"`: The variable upon which the norms will be calculated.

- `"type":["L2-error", "H1-error"]`: Specifies the types of errors to compute. The L2-error is a standard measure of error, whereas the H1-error also incorporates derivatives.

- `"field":f"saturation.name1"`: Defines the field upon which the error will be assessed, with name1 as a variable placeholder.

- `"solution" and "grad_solution"`: These keys define the exact solutions and their gradients respectively, in relation to the material sigmoidal function.

- `"markers":"Omega"`: Indicates the region or subset of the domain where the measures are carried out.

- `"quad": 5`: Refers to the quadrature technique used to estimate the value of integrals. The number 5 signifies the order or precision level of this quadrature.

```
      {
        "Measures":
        {
          "Norm" :
          {
            "saturation" :
            {
              "type":["L2-error", "H1-error"],
              "field":f"saturation.{name1}",
              "solution": "materials_sigmoid:materials_sigmoid:
              x:y:z",
              "grad_solution":"{alpha1*exp(-alpha1 * pressure_p
                - beta1)/(1+exp(-alpha1 *
                pressure_p - beta1))^2, alpha1
                *exp(-alpha1*pressure_p - beta1)
                /(1+exp(-alpha1*pressure_p -
                beta1))^2, alpha1*exp(-alpha1*
                pressure_p - beta1)/(1 +
                exp(-alpha1 * pressure_p -
                beta1))^2}:alpha1:beta1
              :pressure_p:x:y:z",
              "markers":"Omega",
              "quad": 5
            }
          }
        }
      }
```

This part of the code is about visualizing the results of the simulation using `pyvista`, a Python library for 3D plotting and mesh analysis. Let's break it down:

- **Setting up the Graphic Environment:** Before embarking on the actual analysis, it is essential to prime the working environment for proper visualization. In contexts where servers lack a graphical interface, `Xvfb` acts as a virtual display, allowing us to carry out graphical operations unhindered. Concurrently, the `PyVista` library is configured to operate within the Jupyter environment via the *'panel'* backend.

```
    {
      vdisplay = Xvfb()
      vdisplay.start()
      pv.set_jupyter_backend('panel')
    }
```

- **Setting Jupyter Backend:** `pv.set_jupyter_backend('panel')` sets the jupyter notebook backend to use `Panel` for rendering. `PyVista` supports a variety of backends for rendering in a Jupyter environment, and panel is one of them.

- **Function 'pv_get_mesh':** `pv_get_mesh(mesh_path)` function loads the mesh from the provided file path using `pv.get_reader(mesh_path).read()`, which creates a `PyVista` reader object for the file, and then reads it.

```
{
  # load the mesh
  def pv_get_mesh(mesh_path):
  reader = pv.get_reader(mesh_path)
  mesh = reader.read()
  return mesh
}
```

- **Visualization and Saving the Mesh:** Function `'pv_plot_save'` , It allows for the visualization of a scalar field on the mesh and saves this visualization as an image.

```
{
  # save the mesh
  def pv_plot_save(mesh, field, filename, clim=None,
    cmap='viridis', cpos='xy', show_scalar_bar=True,
      show_edges=True):
  p = pv.Plotter(off_screen=True)
  p.add_mesh(mesh, scalars=field, clim=clim, cmap=cmap,
  show_scalar_bar=show_scalar_bar,
  show_edges=show_edges)
  p.screenshot(filename)
}
```

- **Plotting the Mesh:** `pv_plot(mesh, field)` is Similar to `pv_plot_save`, but this directly displays the mesh within the Jupyter environment. Function plots the provided mesh with the given scalar field using the plot function of the mesh object

```
{
  # plot the mesh
  def pv_plot(mesh, field, clim=None, cmap='viridis',
    cpos='xy', show_scalar_bar=True, show_edges=True):
  mesh.plot(scalars=field, clim=clim, cmap=cmap, cpos=cpos,
  show_scalar_bar=show_scalar_bar,
  show_edges=show_edges)
}
```

- **Plotting the Results:** `myplots(dim=3, field="cfpdes.richards.s", factor=1, cmap='viridis')` function plots the results of the simulation. It first loads the mesh, then plots the mesh using the *pv_plot* function. After that, it creates a PyVista Plotter object, and adds the original mesh and its contours to the plotter. If the simulation is 2D, it warps the mesh by the scalar field and plots the warped mesh. If the simulation is 3D, it slices the mesh orthogonally at specific coordinates and plots the slices.

```
{
  # plot the results
  def myplots(dim=3, field="cfpdes.richards.s", factor=1,
  cmap='viridis'):
  mesh = pv_get_mesh(f"cfpdes-{dim}d.exports/Export.case")
  pv_plot(mesh, field)
  pl = pv.Plotter()
```

```
    contours = mesh[0].contour()
    pl.add_mesh(mesh[0], opacity=0.85)
    pl.add_mesh(contours, color="white", line_width=5,
    render_lines_as_tubes=True)
    pl.show()
    if dim == 2:
    warped = mesh[0].warp_by_scalar(field, factor=factor)
    warped.plot(cmap=cmap, show_scalar_bar=False,
    show_edges=True)
    else:
    slices = mesh.slice_orthogonal(x=0.2,y=0.4,z=.6)
    slices.plot()
}
```

- **Convergence Analysis:** Function `'runRichardsPk'`, It performs a convergence test for a given model, computing errors and rates of convergence for various norms.

```
{
    # run the convergence test
    def runRichardsPk(df,model,verbose=False):
    """generate the Pk case

    Args:
    order (int, optional): order of the basis. Defaults to 1.
    """
    meas=dict()
    dim,order,json=model
    for h in df['h']:
    m=richards(hsize=h,json=json,dim=dim,verbose=verbose)
    for norm in ['L2','H1']:
    meas.setdefault(f'P{order}-Norm_saturation_{norm}-
    error', [])
    meas[f'P{order}-Norm_saturation_{norm}-error'].append(
    m.pop(f'Norm_saturation_{norm}-error'))
    df=df.assign(**meas)
    for norm in ['L2','H1']:
    df[f'P{order}-saturation_{norm}-convergence-rate']=
    np.log2(df[f'P{order}-Norm_saturation_
    {norm}-error'].shift() / df[f'P{order}-Norm_
    saturation_{norm}-error']) / np.log2(df['h'].
    shift() / df['h'])
    return df
}
```

- **Overall Convergence Analysis:** Function `'runConvergenceAnalysis'`, It orchestrates the entire convergence analysis process across various configurations.

```
{
    # run the convergence test
    def runConvergenceAnalysis(json,dim=2,hs=[0.1,0.08,0.05],
    orders=[1],verbose=False):
    df=pd.DataFrame({'h':hs})
    for order in orders:
    df=runRichardsPk(df=df,model=[dim,order,json(dim=dim,
    order=order)],verbose=verbose)
    print(df.to_markdown())
    return df
}
```

- **Visualization of the Convergence :** Function `'plot_convergence'`, This final function visualizes the convergence rates, providing a clear graphical interpretation of the

results.

```
    {
      #  plot the convergence
      def plot_convergence(df,dim,orders= [1]):
      fig=px.line(df, x="h", y=[f'P{order}-Norm_saturation_{norm}-
      error' for order,norm in list(itertools.product
      (orders,['L2','H1']))])
      fig.update_xaxes(title_text="h",type="log")
      fig.update_yaxes(title_text="Error",type="log")
      for order,norm in list(itertools.product(orders,['L2','H1'])):
      fig.update_traces(name=f'P{order} - {norm} error -
      {df[f"P{order}-saturation_{norm}-convergence-rate"].
        iloc[-1]:.2f}', selector=dict(name=f'P{order}-Norm_
      saturation_{norm}-error'))
      fig.update_layout(
      title=f"Convergence rate for the {dim}D Richards problem",
      autosize=False,
      width=900,
      height=900,
      )
      return fig
    }
```

This way, you can visualize the results of your simulation in various ways, allowing you to better understand and analyze the output of your simulation. You can see more on [17].

4.5 Numerical results

4.5.1 Reproducting a 2D test case from the literature

We have chosen to work with the following parameters and conditions for the numerical simulations in 2D:

- **Parameters:**
  - $g_x = 0$
  - $g_y = -1$
  - $m = 2$ in $\lambda(s) = s^m$
  - $S_{top} = 0.5$: saturation at the top of the domain
  - $S_{init} = 0.1$: initial saturation
- **Initial conditions:**
  - $S = S_{init}$ in the domain Ω
  - $p = -log(\frac{1}{S_{init}} - 1)$ in the domain Ω
- **Boundary conditions:**
  - Drichlet condition on the top of the domain:
    $S = S_{top}$ and $p = -log(\frac{1}{S_{top}} - 1)$
  - Neumann condition on the **left, right and bottom** of the domain:
    $S = 0$ and $p = 0$

The simulation was carried out using the standard sigmoid function, which is formulated as follows: $S(p) = \frac{1}{1+e^{-\alpha p+\beta}}$. This function was chosen due to its desirable properties such as producing an output that is pleasingly bounded between 0 and 1, and its smooth gradient, which is advantageous for simulations.

While an exact analytical solution to the Richards equation for our specific scenarios was elusive, we resorted to qualitative verification methods to ascertain the accuracy of our numerical simulations. By visually inspecting the results and comparing them against expected patterns and behaviors , we noted a strong alignment, suggesting that our numerical implementation is capturing the essence of the underlying physics. However, for a more rigorous validation, future work should consider methods such as manufactured solutions, convergence studies, or comparisons against well-established benchmark problems in the literature.

- We can see that the results are similar. The difference is that the results of the book are more accurate than ours. This is due to the fact that we have used a mesh with a small number of cells, different parameters and conditions, and a different function for the saturation.

$$K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad and \quad \alpha = 1, \beta = -1 \tag{4.6}$$

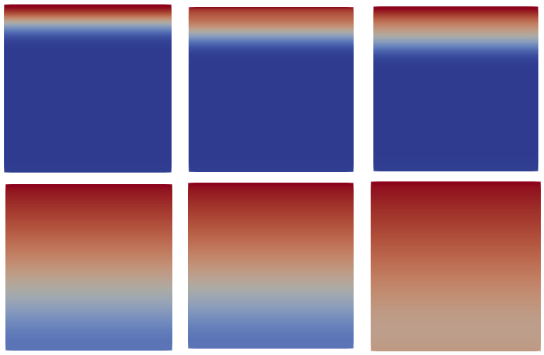


Figure 4.5: Results of the simulation in 2D at $t = 0.1s$, $t = 0.5s$, $t = 0.7s$, $t = 0.81s$, $t = 0.95s$ and $t = 1.23s$

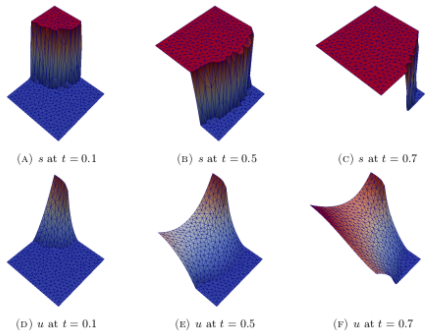


Figure 4.6: Results taken from the book of *KONSTANTIN BRENNER AND CLÉMENT CANCÈS*

As you can see, the saturation is increasing over time, and the water is moving from the top of the domain to the bottom in the direction of the gravity.

- For the same parameters, we have also tried to simulate the results in 2D but with a different geometry (a square domain with a hole at the top). Here are the results:

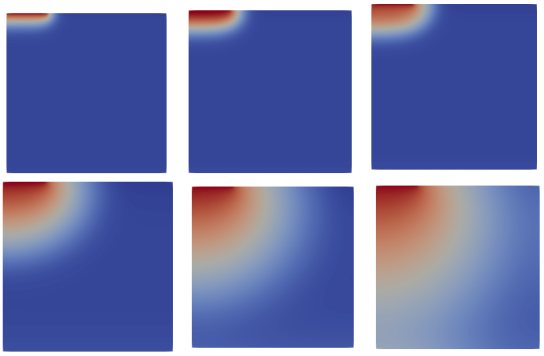


Figure 4.7: Results of the simulation in 2D at $t = 0.1s$, $t = 0.5s$, $t = 0.7s$, $t = 2.9s$, $t = 7.8s$ and $t = 14.1s$

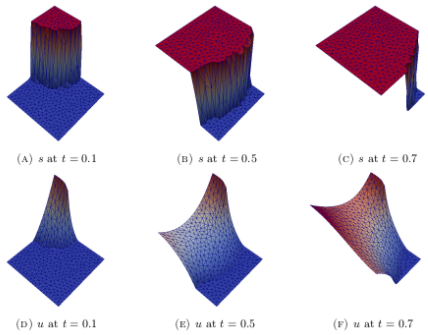


Figure 4.8: Results taken from the book of *KONSTANTIN BRENNER AND CLÉMENT CANCÈS*

We can see that the saturation is increasing over time, and the water is moving from the Opening of the domain to the bottom in the direction of the gravity. The results are similar to the previous ones.

- We have also tried to simulate the results, with the same geometry but different permeability. Here are the results:

$$K = \begin{bmatrix} 1 + 10(y < 0.7) & 0 \\ 0 & 1 + 10(y < 0.7) \end{bmatrix} \quad and \quad \alpha = 1, \beta = -1 \tag{4.7}$$

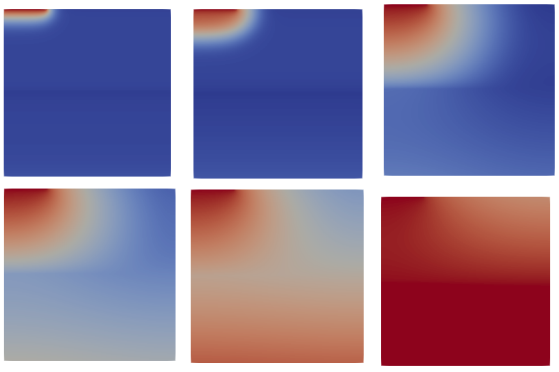


Figure 4.9: Results of the simulation in 3D at $t = 0.1s$, $t = 0.5s$, $t = 0.7s$, $t = 3.2s$, $t = 8.1s$ and $t = 16.5s$

Here we can see that the saturation is increasing over time, and the water is moving from the top opening to the bottom but change the direction by taking the y-axis after $y < 0.7$.

4.5.2 3D test case

Now, we will show the results in 3D. The results of the simulation, visualized over time, are shown below with these parameters :

- **Parameters:**
  - $g_x = 0$
  - $g_y = 0$
  - $g_z = -1$
  - $m = 2$ in $\lambda(s) = s^m$
  - $S_{top} = 0.5$: saturation at the top of the domain
  - $S_{init} = 0.1$: initial saturation
- **Initial conditions:**
  - $S = S_{init}$ in the domain Ω
  - $p = -log(\frac{1}{S_{init}} - 1)$ in the domain Ω
- **Boundary conditions:**
  - Drichlet condition on the top of the domain:
    $S = S_{top}$ and $p = -log(\frac{1}{S_{top}} - 1)$
  - Neumann condition on the **left, right, front, back and bottom** of the domain:
    $S = 0$ and $p = 0$

1. **Test on cube:** In our simulation, we utilized a simple cubic geometry to model the flow of fluid through a porous medium. This is a common choice for the initial step of modeling flow in subsurface environments due to its simplicity and the ease with which the results can be interpreted.

   The simulation was conducted using the standard sigmoid function as our saturation-pressure relationship. The results of this simulation, visualized over time, are shown below:

$$K = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad and \quad \alpha = 1, \beta = -1 \tag{4.8}$$

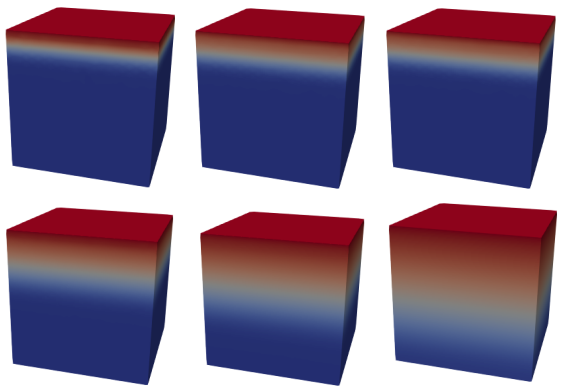


Figure 4.10: Results of the simulation in 3D at $t = 0.1s$, $t = 0.5s$, $t = 0.7s$, $t = 3.2s$, $t = 8.1s$ and $t = 16.5s$

As we can see, the fluid flows from the top to the bottom of the cube. This is expected as it reflects the influence of gravity on fluid movement in porous media. At the beginning of the simulation, different colors indicate varying levels of saturation throughout the medium.

Over time, as the simulation proceeds, the entire cube becomes red, indicating high saturation. This shows the progression of fluid flow and saturation within the cubic geometry over time. It illustrates the ability of our model to capture the evolution of saturation levels in a porous medium under the influence of gravity.

This simulation validates the effectiveness of using the standard sigmoid function for simulating Richards' problems in simple cubic geometries. Future work will involve exploring more complex geometries and different types of fluid to further validate and extend this approach.

2. **Test on cube with one fracture:** To broaden our investigation, we conducted another simulation on a cubic geometry that includes a vertical fracture. This fracture does not touch the top of the cube, which makes the scenario more realistic for many underground fluid flow situations.

   We used the standard sigmoid function for the saturation-pressure relationship in three different scenarios, altering alpha and beta parameters and playing with permeability values.

   - **Same Sigmoid Function for the Matrix and the Fracture:** In this case, as soon as the fluid flow reaches the top of the fracture, the presence of the fracture allows the fluid to move more quickly, leading to a rapid saturation of the fracture. Once the fracture is saturated, the fluid continues to flow through the rest of the cube.

$$K = \begin{cases} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} & \text{for the matrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \text{for the fracture} \end{cases} \quad \text{and} \quad \alpha = 1, \beta = -1 \tag{4.9}$$

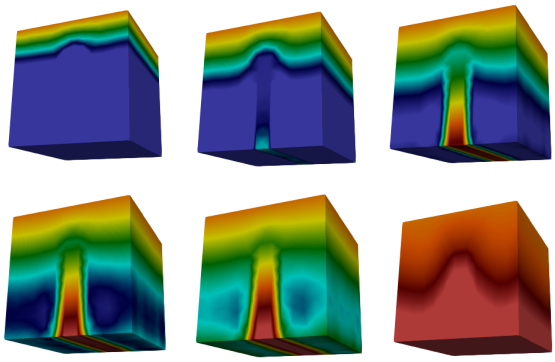


Figure 4.11: Results of the simulation in 3D at $t = 8.2s$, $t = 17.9s$, $t = 25.6s$, $t = 27.8s$, $t = 29.7s$ and $t = 44.9s$

- **Lower Permeability in the Fracture:** In this scenario, we reversed the situation and gave the fracture a lower permeability than the matrix. The result is that as soon as the fluid reaches the top of the fracture, its flow becomes very slow. Therefore, the matrix fills before the fracture gets saturated.

$$K = \begin{cases} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \text{for the matrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} & \text{for the fracture} \end{cases} \quad \text{and} \quad \alpha = 1, \beta = -1 \tag{4.10}$$

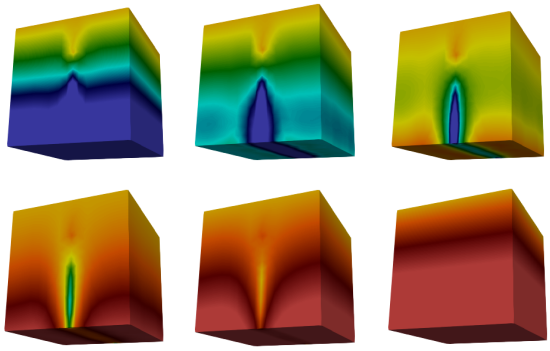


Figure 4.12: Results of the simulation in 3D at $t = 0.36s$, $t = 0.52s$, $t = 0.70s$, $t = 0.94s$, $t = 1.12s$ and $t = 4.99s$

- **Different Sigmoid Function for the Fracture and the Matrix:** Here, we used different values for alpha and beta for the fracture and the matrix. The results of this simulation showed faster saturation of the fracture compared to the matrix, indicating

preferential flow within the fracture. We consider Ω_1 as the matrix and $\Omega_2, \Omega_3, \Omega_4$ as fractures :

$$\begin{cases} K_{\Omega_1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} \quad \text{and} \quad K_{\Omega_2} = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ K_{\Omega_3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad K_{\Omega_4} = \begin{bmatrix} cos(1.11) & 0 & 0 \\ 0 & cos(0.46) & 0 \\ 0 & 0 & cos(0.57) \end{bmatrix} \end{cases} \tag{4.11}$$

$$\begin{cases} \alpha = 1, \beta = -5 & \text{for the matrix} \\ \alpha = 2, \beta = 10 & \text{for the fractures} \end{cases} \tag{4.12}$$

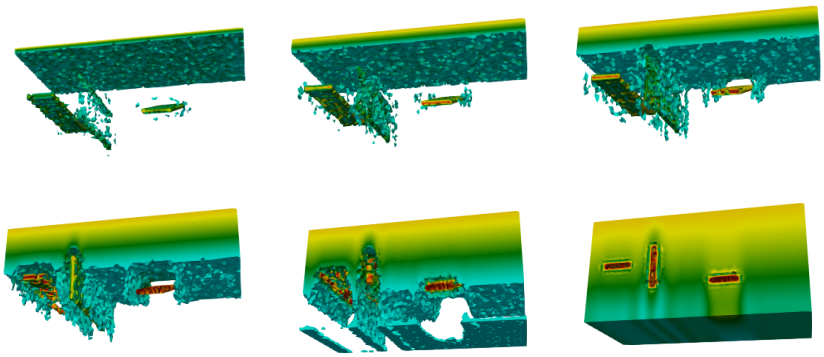


Figure 4.13: Results of the simulation in 3D at $t = 0.1s$, $t = 0.9s$, $t = 3.4s$, $t = 9.0s$, $t = 20.9s$ and $t = 49.9s$

These results demonstrate that the use of the standard sigmoid function is capable of accurately representing fluid flows in porous media with fractures, and that modifying the alpha and beta parameters as well as permeability values can assist in modeling various flow scenarios. This paves the way for future research on fluid flows in more complex geometries, including those with multiple fractures and other types of discontinuities.

3. **Simulation Results with a Terrain-Like Geometry and Multiple Fractures :** As an extension to our study, we carried out a simulation on a terrain-like geometry, further increasing the complexity of the model. This geometry incorporates four horizontal fractures along with an obliquely oriented vertical fracture. The vertical fracture begins at the midpoint of the third horizontal fracture, thus intersecting the third and fourth horizontal fractures.

   The objective was to observe the interplay between these fractures in terms of fluid flow behavior when subjected to the forces of gravity. This would also allow us to evaluate the influence of fracture intersections, which are often key points of interest in reservoir

studies. Here is the figure depicting the fluid flow pattern in this scenario (frac 1, frac 2, frac 3, frac 4):

$$K = \begin{cases} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.9 \end{bmatrix} \text{ for the top matrix} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} \text{ for frac 1} \\ \begin{bmatrix} 0.9 & 0 & 0 \\ 0 & 0.9 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} \text{ for frac 2} \end{cases} \quad \text{and} \quad K = \begin{cases} \begin{bmatrix} 0.85 & 0 & 0 \\ 0 & 0.85 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} \text{ for frac 3} \\ \begin{bmatrix} 0.9 & 0 & 0 \\ 0 & 0.9 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ for frac 4} \end{cases}$$

$$K = \begin{cases} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} \text{ for the matrix} \\ \begin{bmatrix} cos(-1.262) & 0 & 0 \\ 0 & cos(-0.291) & 0 \\ 0 & 0 & cos(-0.983) \end{bmatrix} \text{ for the oblique fracture} \end{cases}$$

$$\begin{cases} \alpha = 1, \beta = -5 & \text{for the matrix and frac 1, 2, 3, 4} \\ \alpha = 2, \beta = 10 & \text{for the oblique fracture} \end{cases}$$

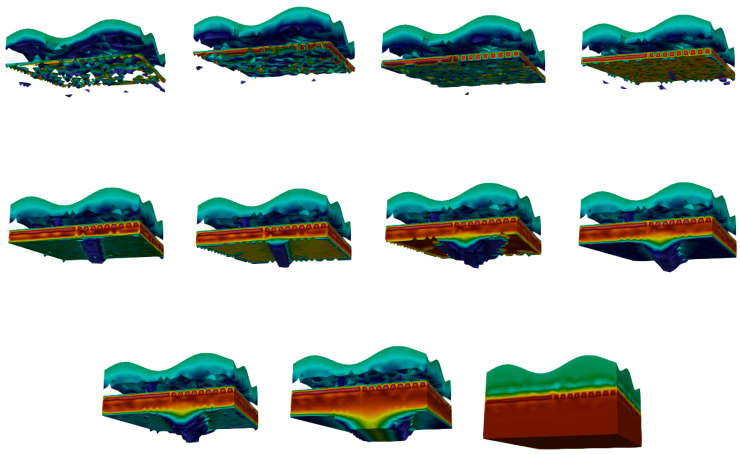


Figure 4.14: Results of the simulation in 3D at $t = 0.1s$, $t = 0.2s$, $t = 0.3s$, $t = 0.4s$, $t = 0.8s$, $t = 0.9s$, $t = 1.1s$, $t = 1.2s$, $t = 1.3s$, $t = 1.9s$ and $t = 4.0s$

The results, as seen above, are illuminating. When the fluid was introduced, it preferentially filled the horizontal fractures from the third to the topmost, driven by the gravitational forces and the elevated permeability of the fractures. Upon reaching the intersection

with the vertical fracture, a part of the fluid started descending down, filling the lower horizontal fractures.

This simulation emphasizes the complexity of fluid movement in fractured media, particularly when the fractures are interconnected and positioned in different orientations. The impact of the fractures' relative permeability is once again underlined, dictating the sequence and speed of the fluid's progression. It also highlights the significance of fracture intersections as zones of fluid redirection and potential points of high fluid transfer.

For more detailed information about this simulation and others conducted in this study, please click the following link: [14].

Chapter 5

Adapting time step in case of non-convergence when solving Richards equations with `Feel++`

In the scope of our stage project, we focused on solving 3D Richards' problems using the `Feel++` library. More specifically, we aimed at implementing an adaptive approach to handle the time step size when solving the nonlinear equations, which can be crucial for the convergence of simulations.

The `Feel++` library is a `C++` library dedicated to solving physics and mathematics problems using finite element methods. We used the Python interface of `Feel++`, which allows directly calling `C++` functions from Python. We employed the Cfpdes class of `Feel++` for solving the $3D$ Richards problem.

5.1 Problem with Adaptation

The primary challenge we encountered was that the `solve()` method of the `Cfpdes` class behaved in a blocking manner: once it was started, it had to finish before any other Python code could be executed. Moreover, if the `solve()` method did not manage to converge, it halted the program entirely instead of allowing the execution of other Python codes.

5.2 Proposed Solution

To overcome this issue, we wrote a `while` loop in our Python code to manage the time step. When `solve()` is called, we tried to detect whether it has converged or not, and if the solve time is acceptable. If it does not converge or the solve time is too long, we halved the time step and retried `solve()`.

5.3 Issue with Proposed Solution

However, this approach failed as the `solve()` method halted the program when it failed to converge.

We also tried to catch exceptions raised by the `solve()` method using a `try/except` block in Python, but this also did not work. We learned that exceptions thrown by the underlying `C++` code are not automatically converted to Python exceptions. To address this issue, it would likely require modifying the `C++` code to produce suitable Python exceptions.

5.4 Corresponding Python Code

Here is an excerpt of the Python code corresponding to the solution we attempted to implement, to see the full code, please click : [13].

```
{

  if richards.isStationary():
  try:
  richards.solve()
  except Exception as e:
  print(f"Error encountered during solve: {e}")
  return None
  richards.exportResults()
  else:
  while not richards.timeStepBase().isFinished():
  if richards.worldComm().isMasterRank():
  print("========================================\n")
  print("time simulation: ", richards.time(), "s \n")
  print("========================================\n")

  dt = richards.timeStepBase().timeStep()
  while dt > 1e-10:  # replace our own lower limit for time step
  start_time = time.time()
  try:
  converged = richards.solve()
  solve_time = time.time() - start_time
  if solve_time < 1000.0:  # our own threshold for solve
  break  # exit the inner while loop
  else:
  print(f"Solve() took too long ({solve_time}
  seconds), reducing time step")
  dt *= 0.5  # reduce time step by a factor of two
  richards.setTimeStep(dt)  # set new time step
  except Exception as e:
  print(f"Error encountered during solve: {e}")
  traceback.print_exc() # print traceback to see the error
  if not converged:
  print(f"Warning: solve() did not converge :
  {converged}")
  dt *= 0.5
  richards.setTimeStep(dt)

  if dt <= 1e-10:  # replace our own lower limit for time step
  print("Warning: Time step became too small even though
  solve() takes too long. Exiting simulation.")
  return None

  richards.exportResults()
  richards.updateTimeStep()
}
```

In conclusion, managing the time step in case of non-convergence when solving Richards equations with `Feel++` proved to be a complex challenge. Despite our efforts to find a Python

solution, the limitations imposed by the `Feel++` interface made the task difficult. A modification to the `Feel++ C++` library itself seems to be necessary to solve this problem, which is unachievable within our current project scope.

Chapter 6

Conclusion

Throughout this exploration, we journeyed from understanding the historical and general contexts of Darcy's law to adapting it in contemporary models. We began with a deep dive into its historical foundations, highlighting its significance and the issues related to it. A subsequent section was dedicated to acquainting ourselves with various models associated with porous media, focusing on the Two-Phase Flow and Richards' models. The importance of geometry construction was addressed next, utilizing Gmsh as our primary tool and advancing to more intricate mesh generation with Python.

Our meticulous testing phase with the Richards' Model unveiled the intricacies of the mathematical model and the simulation workflows. Emphasizing the practical applications of our endeavors, we faced challenges, especially regarding time step adaptation during non-convergence scenarios. This segment not only illustrated the hurdles we encountered but also underscored the innovative solutions we devised, and the subsequent challenges they posed.

In essence, this report not only acts as an exhaustive guide to understanding and applying Darcy's law and its allied models but also underlines the iterative nature of scientific exploration. This involves recognizing challenges, crafting solutions, and persistently extending the frontiers of our knowledge. As we conclude this chapter, it becomes clear that the domain of porous media and its associated flows present a vast landscape of opportunities, beckoning further exploration and innovation.

Chapter 7

Bibliography

Bibliography

[1] A. Szymkiewicz. Modeling of Water Flow in insaturated Porous Media. 2013.

[2] https://en.wikipedia.org/wiki/Darcy%27s_law

[3] Darcy, H. "Les fontaines publiques de la ville de Dijon". Paris: Dalmont & 1856.

[4] Whitaker, S. "Flow in porous media I: A theoretical derivation of Darcy's law". Transport in Porous Media. 1: 3–25. doi:10.1007/BF01036523. S2CID 121904058 & 1986.

[5] Read "Memorial Tributes: Volume 14" at NAP.edu. 2011. doi:10.17226/12884. ISBN 978-0-309-15218-1.

[6] Guy Chavent, Jerome Jaffre. Mathematical models and finite element for reservoir simulation. Single, multiphase and multicomponent flows through.

[7] Walker, S. "Flow in Porous Media." John Wiley & Sons, 1986.

[8] Roland Masson - "Numerical simulation of two phase porous media flow models with application to oil recovery."

[9] Szymkiewicz, A. ": Modelling Water Flow in Unsaturated Porous Media". Springer Berlin Hei-delberg: & 2013.

[10] Discontinuous Galerkin Method for steady-state Richards Equation

[11] https://github.com/master-csmi/2023-stage-obed-darcy/blob/main/code/geo

[12] https://github.com/master-csmi/2023-stage-obed-darcy/blob/main/code/geo/terrainfracture.py

[13] https://github.com/master-csmi/2023-stage-obed-darcy/tree/main/code/infiltration_python/richards_adapt.py

[14] https://github.com/master-csmi/2023-stage-obed-darcy/tree/main/pdf/image

[15] https://github.com/master-csmi/2023-stage-obed-darcy/blob/main/code/infiltration_3D/richards.cfg

[16] https://github.com/master-csmi/2023-stage-obed-darcy/tree/main/code/infiltration_3D

[17] https://github.com/master-csmi/2023-stage-obed-darcy/tree/main/code