Simulation of Richards equation using Newton method and application of the parametrization method.

POLITECNICO DI MILANO
MsC Computational Science and Engineering
Sabrina Bassetto
Matr. 874972
AA 2018-19
Chapter 1

Introduction

The aim of this project is to simulate, using Newton’s method and finite volume scheme, the Richards equation which represent the movement of water in unsaturated soil.

Let us denote by \( \Omega \) a bounded open set of \( \mathbb{R}^d \) \((d \leq 3)\) representing a the porous medium, by \( T > 0 \) a finite time horizon and by \( Q := \Omega \times (0,T) \) the corresponding space-time cylinder. We want to find a saturation profile \( s : Q \to [0,1] \) and a water pressure \( p : Q \to \mathbb{R} \) such that, once setting porosity equal to one:

\[
\partial_t s - \nabla \cdot (\lambda(s)(\nabla p - g)) = 0 \quad (1.1)
\]

where \( \lambda : [0,1] \to \mathbb{R}_+ \) is the mobility function, a nondecreasing \( C^2 \) function that satisfies \( \lambda(s) = 0 \) for \( s \leq 0 \) and \( \lambda(s) > 0 \) for \( s > 0 \), and \( g \in \mathbb{R}^d \) is the gravity vector. Pressure and saturation are supposed to be linked by some monotone relation:

\[
s = S(p) \quad \text{a.e. in } Q \quad (1.2)
\]

where \( S \) is a nondecreasing function, \( S : \mathbb{R} \to [0,1] \). We assume \( S(p) = 1 \) for all \( p \geq 0 \), corresponding to assuming that the porous medium is wet. Moreover \( S \in L^1(\mathbb{R}_-) \) implying that \( \lim_{p \to -\infty} S(p) = 0 \). In Figure 1.1 a plot of the saturation and mobility function.

Consequently, adding Dirichlet boundary condition for the pressure and an initial condition for the saturation, our problem reads:

\[
\begin{aligned}
\partial_t s - \nabla \cdot (\lambda(s)(\nabla p - g)) &= 0 \quad \text{in } Q \\
\bar{s} &= S(p) \\
s|_{t=0} &= s_0 \in L^\infty(\Omega,[0,1]) \\
p|_{\partial \Omega \times (0,T)} &= p_D
\end{aligned}
\quad (1.3)
\]

Sometimes, at least from the mathematical point of view, equation (1.1) is solved by choosing \( p \) or the Kirchhoff transform \( u \), which is also called total pressure, as primary unknown and then deduce \( s = S(p) = \tilde{S}(u) \). We want to solve the following problem:

\[
\begin{aligned}
\partial_t s + \nabla \cdot (\lambda(s)g - \nabla u(s)) &= 0 \quad \text{in } Q \\
\bar{s} &= \bar{S}(u) \\
s|_{t=0} &= s_0 \in L^\infty(\Omega,[0,1]) \\
p|_{\partial \Omega \times (0,T)} &= p_D
\end{aligned}
\quad (1.4)
\]
1.1 Discretization of the domain

Let us denote by $\Omega$ an open set of $\mathbb{R}^d$ ($d \leq 3$) representing the porous medium. We indicate with $T > 0$ a finite time horizon. We can now define $Q := \Omega \times (0, T)$ the corresponding space-time cylinder. In this work we are going to consider cylindrical discretizations of $Q$, this means that we are going to discretize in space and time separately. Let us define an admissible mesh $\Omega$ (which is supposed to be polyhedral for meshing purpose). An admissible mesh $(\mathcal{T}, \mathcal{E})$ of $\Omega$ is given by a set $\mathcal{T}$ of disjointed open bounded convex subsets of $\Omega$ called control volumes, and a family $\mathcal{E}$ of subsets of $\Omega$ called edges contained in the hyperplanes of $\mathbb{R}^d$ with strictly positive measure. Moreover each control volume has center $x_K$, $\forall K \in \mathcal{T}$, and holds $\bigcup_{K \in \mathcal{T}} K = \Omega$. The boundary of a control volume is made of edges: $\forall K \in \mathcal{T}$ there exists a subset $E_K$ of $\mathcal{E}$, where $\bigcup_{E_K \in \mathcal{E}} E_K = \mathcal{E}$, such that $\partial K = \bigcup_{\sigma \in E_K} \sigma$. Let us make the distinction between inner edges and boundary ones. For any $(K, L) \in \mathcal{T}^2$, either the $(d-1)$-dimensional Lebesgue measure of $K \cap L$ is 0, or $K \cap L = \sigma$ for some $\sigma \in \mathcal{E}$. In the last case we can say that $\sigma = K|L$ and we denote by $E_{\text{int}} = \{\sigma \in \mathcal{E}, \exists (K, L) \in \mathcal{T}^2, \sigma = K|L\}$ the set of inner edges and by $E_{\text{ext}} = \{\sigma \in \mathcal{E}, \sigma \subset \partial \Omega\}$ the boundary edges. Finally the family of cell centers $(x_K)_{K \in \mathcal{T}}$ is such that, if $\sigma = K|L$ it is assumed that the straight line $(x_K, x_L)$ is orthogonal to $\sigma$. This is an important property that we will use after. Moreover for all $\sigma \in E_{\text{ext}}$, there exists one unique cell $K$ such that $\sigma \in E_K$. Then we denote $x_\sigma$ the projection of $x_K$ over the hyperplane containing $\sigma$ and $x_\sigma \in \sigma$.

![Figure 1.2: Example of admissible mesh and detail of two cells $K, L$ sharing a face $\sigma \in E_{\text{int}}$](image)

1.2 Space and time discretization

First let us introduce the space of degrees of freedom (including those of the boundary)

$$X_T = \{v = (v_K)_{K \in \mathcal{T}} \} \simeq \mathbb{R}^\#T.$$

Now we will introduce a space-time discretization for our problem.

**Definition 1.** A time discretization of $(0, T)$ is given by an integer value $N$ and a sequence of real values $0 = t_0 < t_1 < ... < t_N = T$. For all $n \in \{1, \ldots, N\}$ the time step is defined by $\Delta t^n = t^n - t^{n-1}$.

**Definition 2.** A space-time discretization $\mathcal{D}$ of $Q$ is a family

$$\mathcal{D} = (\mathcal{T}, \mathcal{E}, (t^n)_{n \in \{0, \ldots, N\}})$$

where $(\mathcal{T}, \mathcal{E})$ is an admissible mesh of $\Omega$, as defined in the previous section, and $(N, (t^n)_{n \in \{0, \ldots, N\}})$ is a discretization of $(0, T)$.

The spaces of degrees of freedom are

$$X_D = \{v = (v^n_K)_{K \in \mathcal{T}, 1 \leq n \leq N} \} \simeq \mathbb{R}^{\#T \times N}.$$

Let $v = (v^n_K)_{K,n} \in X_D$ and we denote $v^n = (v^n_K)_K \in X_T$ for $n \in \{1, \ldots, N\}$. 
1.3 Implicit Finite Volume scheme

The numerical scheme we are going to apply in our work is a finite volume scheme approximating the diffusive fluxes with the so-called two-point flux approximation (TP for short). We can use this approximation if the diffusion tensor is diagonal and if the mesh satisfies the orthogonality condition.

**Definition 3.** The mesh $\mathcal{M}$ satisfies the orthogonality condition if each control volume $K \in \mathcal{M}$ is assumed to contain a point $x_K$ such that, for each pair of neighboring control volumes $K, L \in \mathcal{M}$, the line $(x_K, x_L)$ is orthogonal to the common interface $K \cap L$.

Let us consider $u \in X_T$. The finite volume method consist in first writing, using Stokes’ theorem:

$$- \int_K \nabla \cdot (\nabla u) d\mathbf{x} = \int_{\partial K} -\nabla u \cdot \mathbf{n}_K d\sigma = - \sum_{\sigma \in E_K} \int_{\sigma} \nabla u \cdot \mathbf{n}_{K,\sigma} d\sigma$$

Now we specify the approximation $F_{K,\sigma}$ of the flux $-\int_{\sigma} \nabla u \cdot \mathbf{n}_{K,\sigma} d\sigma$. The TP numerical flux $F_{K,\sigma}$ is defined by:

$$F_{K,\sigma} = \begin{cases} m_\sigma \frac{u_{K,\sigma} - u_{L,\sigma}}{d_{\sigma}} & \text{if } \sigma = K | L \in E_{int}, \\ m_\sigma \frac{u_{K,\sigma} - u_{L,\sigma}}{d_{\sigma}} & \text{if } \sigma \in E \cap E_{ext} \end{cases}$$

(1.5)

where $m_\sigma$ is the $(d - 1)$-Lebesgue measure of the edge $\sigma \in \mathcal{E}$, $d_{K,\sigma} = d(x_K, x_\sigma)$ and $d_{\sigma} = d(x_K, x_L)$, finally $u_{K,\sigma}$ correspond to the boundary value of $u$ on $\sigma$ when $\sigma \in E_{ext}$. The numerical flux is then consistent under orthogonality assumption on the mesh in the sense that, for any regular function $\phi \in C^\infty(\Omega)$

$$\left| \int_{\sigma} \nabla \phi \cdot \mathbf{n}_{K,\sigma} + F_{K,\sigma} \right| \leq |\sigma| C_\phi h$$

where $v \in X_T$ is defined by $v_K = \phi(x_K)$, for all $K \in T$, $C_\phi \in \mathbb{R}_+$ depends on $\phi$ and $h = \max\{\text{diam}(K), K \in T\}$. From (1.5), it holds the following local conservation property, when $\sigma = K | L$:

$$F_{K,\sigma} + F_{L,\sigma} = 0$$

From equation (1.5) we define the transmissibilities $(A_{\sigma})_{\sigma \in \mathcal{E}}$ by

$$A_{\sigma} = \begin{cases} m_\sigma & \text{if } \sigma = K | L \in E_{int} \\ m_\sigma & \text{if } \sigma \in E \cap E_{ext} \end{cases}$$

(1.6)

Now we are able to discretize our problem (1.4). The initial data $s_0$ is discretized by $s^0 = (s^0_K)_{K \in T} \in X_{T,\text{int}}$ by setting

$$s^0_K = \frac{1}{m_K} \int_K s_0(x) d\mathbf{x} \quad \forall K \in T$$

We can notice that $0 \leq s^0_K \leq 1$ since $0 \leq s_0 \leq 1$. So we have:

$$\begin{cases} \partial_t s + \nabla \cdot (\lambda(s) \mathbf{g} - \nabla u(s)) = 0 & \text{in } Q \\ s = s_D & \text{on } \partial Q \\ s|_{t=0} = s^0 \end{cases}$$

(1.7)

We integrate the first equation over $K$, discretize the temporal derivative and we apply the Stokes theorem:

$$\int_K \frac{s^n_K - s^{n-1}_K}{\Delta t} d\Omega + \sum_{\sigma \in E_K} \left( \int_{\sigma} \lambda(s^n_{\sigma}) \mathbf{g} \cdot \mathbf{n}_{K,\sigma} d\sigma - \int_{\sigma} \nabla u(s^n_{\sigma}) \cdot \mathbf{n}_{K,\sigma} d\sigma \right)$$

Let us focus on the convective term $\int_{\sigma} \lambda(s^n_{\sigma}) \mathbf{g} \cdot \mathbf{n}_{K,\sigma} d\sigma$. We discretize this term using upwind discretization:

- if $\mathbf{g} \cdot \mathbf{n}_{K,\sigma} \geq 0$ : this means that we see an outward normal. So we stay on cell $K$ (which is cell0 wrt the face $\sigma$),
- if $\mathbf{g} \cdot \mathbf{n}_{K,\sigma} < 0$ : this means that we see an inward normal. So we stay on cell $L$ (which is cell1 wrt the face $\sigma$) or, if $\sigma \in E_{ext}$ we stay on the face itself.
Then the fluxes \( F_{K,\sigma} \) w.r.t. \( m \)
where \( g_{K,\sigma} = g^{+}_{K,\sigma} - g^{-}_{K,\sigma} \) and \( g^{+}_{K,\sigma} = \max(0, g_{K,\sigma}) \), \( g^{-}_{K,\sigma} = -\min(0, g_{K,\sigma}) \). Finally we have:
\[
\frac{s_{K}^{n} - s_{K}^{n-1}}{\Delta t} m_{K} + \sum_{\sigma \in \mathcal{E}_{K}} F^{n}_{K,\sigma} = 0 \quad \forall K \in \mathcal{T}
\]  
(1.8)
where \( m_{K} \) is the \( d \)-dimensional Lebesgue measure of the control volume \( K \in \mathcal{T} \). \( F^{n}_{K,\sigma} \) denotes the outward flux w.r.t. \( K \) across the edge \( \sigma \) at time step \( t^{n} \). Let us denote \( n_{K,\sigma} \) the outward normal to \( \sigma \) w.r.t. \( K \), \( g_{K,\sigma} = g \cdot n_{K,\sigma} \) for all \( \sigma \in \mathcal{E}_{K} \) and all \( K \in \mathcal{T} \). Moreover
\[
s^{n}_{K,\sigma} = \begin{cases} 
 s_{D}^{n} & \text{if } \sigma = K | L \in \mathcal{E}_{int} \\
 s_{D}^{n} & \text{if } \sigma \in \mathcal{E}_{ext}
\end{cases}
\]
Then the fluxes \( F^{n}_{K,\sigma} \) are defined by
\[
F^{n}_{K,\sigma} = m_{\sigma} \left( \lambda(s^{n}_{K})g^{+}_{K,\sigma} - \lambda(s^{n}_{K})g^{-}_{K,\sigma} \right) + A_{\sigma}(u^{n}_{K} - u^{n}_{K,\sigma}).
\]  
(1.9)
Summarizing our system, made by (1.8) and (1.9), reads, \( \forall K \in \mathcal{T} \):
\[
\frac{s^{n}_{K} - s_{K}^{n-1}}{\Delta t} m_{K} + \sum_{\sigma \in \mathcal{E}_{K}} F^{n}_{K,\sigma} = 0 \quad \forall K \in \mathcal{T}
\]  
(1.10)
We can rewrite this formulation in a different way noticing that, since \( g \) is divergence free, we have:
\[
\sum_{\sigma \in \mathcal{E}_{K}} m_{\sigma} g_{K,\sigma} = 0 \quad \forall K \in \mathcal{T}
\]
Hence, if we add \( -\sum_{\sigma \in \mathcal{E}_{K}} m_{\sigma} g_{K,\sigma} \lambda(s^{n}_{K}) \) to (1.9) and we exploit the decomposition \( g_{K,\sigma} = g^{+}_{K,\sigma} - g^{-}_{K,\sigma} \), the convective flux balance can be reformulated as:
\[
\sum_{\sigma \in \mathcal{E}_{K}} F^{n}_{K,\sigma} = \sum_{\sigma \in \mathcal{E}_{K}} m_{\sigma} g_{K,\sigma} \left( \lambda(s^{n}_{K}) - \lambda(s_{K,\sigma}^{n}) \right) + \sum_{\sigma \in \mathcal{E}_{K}} A_{\sigma} \left( u^{n}_{K} - u^{n}_{K,\sigma} \right) \quad \forall K \in \mathcal{T}
\]
1.3.1 The Newton method
The numerical scheme (1.10), for all \( n \in \{1, \ldots, N\} \), lead us to the nonlinear system
\[
\mathcal{F}_{n}(s^{n}) = (f^{n}_{K}(s^{n}))_{K \in \mathcal{T}} = 0
\]  
(1.11)
where
\[
f^{n}_{K}(s^{n}) = m_{K} \frac{s^{n}_{K} - s_{K}^{n-1}}{\Delta t} + \sum_{\sigma \in \mathcal{E}_{K}} \left( m_{\sigma} \lambda(s^{n}_{K})g^{+}_{K,\sigma} - \lambda(s^{n}_{K})g^{-}_{K,\sigma} + A_{\sigma} \left( u^{n}_{K} - u^{n}_{K,\sigma} \right) \right) \quad \forall K \in \mathcal{T}
\]  
(1.12)
Solving (1.18) with the Newton method consists in the construction of a sequence \((s^{n,k})_{k \geq 0}\) defined by:
\[
\begin{cases} 
 s^{n,0} = s^{n-1} \\
 s^{n,k+1} = s^{n,k} - [J_{\mathcal{F}_{n}}(s^{n,k})]^{-1} \mathcal{F}_{n}(s^{n,k})
\end{cases}
\]  
(1.13)
where \( \|J_{\mathcal{F}_{n}} \) is the Jacobian matrix of \( \mathcal{F}_{n} \) at \( s^{n,k} \) and we assume that it is not singular.
The Jacobian matrix is an $\mathcal{T} \times \mathcal{T}$ matrix obtained by computing the partial derivative w.r.t. $s^{n,k}$ of (1.18):

$$
\mathcal{J}_{f_n} = \frac{\partial F_n(s^{n,k})}{\partial s^{n,k}} = \left( \frac{\partial f_n}{\partial s^{n,k}} \right)_{K \in \mathcal{T}, L \in \mathcal{T}}
$$

Starting from equation (1.12) we can write the expression of $(\mathcal{J}_{f_n})_K$:

$$(\mathcal{J}_{f_n})_K = \frac{m_K}{\Delta t} + \sum_{\sigma \in \mathcal{E}_K} \left( A\sigma \left( \frac{\partial u(s_{K})}{\partial s_{K}} - \frac{\partial u(s_{K})}{\partial s_{K}} \right) + m_\sigma \left( g_{K,\sigma}^+ \frac{\partial \lambda(s_{K})}{\partial s_{K}} - g_{K,\sigma}^- \frac{\partial \lambda(s_{K})}{\partial s_{K}} \right) \right)
$$

So:

$$(\mathcal{J}_{f_n})_{K,K} = \frac{m_K}{\Delta t} + \sum_{\sigma \in \mathcal{E}_K} \left( A\sigma \frac{\partial u(s_{K})}{\partial s_{K}} + m_\sigma g_{K,\sigma}^+ \frac{\partial \lambda(s_{K})}{\partial s_{K}} \right)
$$

$$
(\mathcal{J}_{f_n})_{K,L} = - \left( A\sigma \frac{\partial u(s_{K})}{\partial s_{K}} + m_\sigma g_{K,\sigma}^- \frac{\partial \lambda(s_{K})}{\partial s_{K}} \right)
$$

### 1.4 Monotone parametrization

The problem is that this approach may lack of efficiency when one try to solve Richards equation numerically, in particular for dry media (saturation $s$ close to 0) because, in this situation, Newton methods encounter difficulties converging when solving non linear systems obtained by standart implicit numerical methods as two-point flux approximation finite volume method. A classical approach to avoid this problems consist of applying the so-called variable switch which consists of changing the primary variable following the physical configuration.

Here we describe summarily a technique introduces in the article [1] which consists of parameterizing the graph $S$ in order to stabilize Newton algorithm without implementing the switch procedure. The target of this procedure is to keep the derivatives of $u$ and $s$ under control. Indeed a convenient parametrization should satisfy the following:

$$
\exists \alpha_s > 0 \text{ and } \alpha^* \geq \alpha_s \text{ s.t. } \alpha_s \leq \max(s'(\tau), u'(\tau)) \leq \alpha^* \quad \forall \tau \in \mathbb{R}.
$$

Let us introduce two continuously differentiable nondecreasing functions

$$
s : (\tau_*, \infty) \to [0, 1] \text{ and } p : (\tau_*, \infty) \to (-\infty, \infty)
$$

where $\tau_* < 0$ may be equal to $-\infty$, such that $p(0) = 0$ and

$$
s \in S(p) \iff \text{ there exists } \tau \geq \tau_* \text{ s.t. } s = s(\tau) \text{ and } p = p(\tau)
$$

This implies that $s(0) = 1$, $\lim_{\tau \to \tau_*} s(\tau) = 0$ and $\lim_{\tau \to \tau_*, p(\tau) = p_*}. \text{ In the case } \tau_* > -\infty \text{ the functions } s, p \text{ are continuously extended into constants on } (-\infty, \tau_*) \text{. Let us now define the Kirchhoff transform } u : [\tau_*, +\infty) \to \mathbb{R} \text{ defined by}

$$
u(\tau) = \int_0^\tau \lambda(s(a))p'(a)da \quad \forall \tau \geq \tau_*
$$

For technical reasons the function $u$ is artificially extended into a continuous onto function from $\mathbb{R}$ to $\mathbb{R}$ by setting:

$$
u(\tau) = \tau - \tau_* + u_* \quad \forall \tau \geq 0
$$

without influencing the result. We assume that the parametrization is not degenerated, this means that

$$
s'(\tau) + p'(\tau) > 0 \text{ for a.e. } \tau \geq \tau_* \text{ or } s'(\tau) + u'(\tau) > 0 \text{ for a.e. } \tau \in \mathbb{R}
$$

since $\lambda(s(\tau)) > 0$ for all $\tau > \tau_*$. We can deduce from (1.15) and (1.16) that $u' > 0$ a.e. in $\mathbb{R}$ and, because of it, the function $\tilde{S} = s \circ u^{-1}$ is well defined. Is important to highlight that the definition if $\tilde{S}$ is invariant w.r.t. the choice of parametrization of the graph $s$. Notice that a parametrization of the graph $S$ always exists but it is not unique. Let us choose $\tau$ as the primary variable, so (1.1) now reads:

$$
\partial_t S(\tau) - \nabla \cdot (\lambda(s(\tau))(\nabla p(\tau) - g)) = 0 \text{ in } Q
$$

and it turns to

$$
\partial_t S(\tau) + \nabla \cdot (\lambda(s(\tau))g - \nabla u(\tau)) = 0 \text{ in } Q
$$
CHAPTER 1. INTRODUCTION

for \( \tau \geq \tau_* \). Now we impose the boundary condition

\[
\tau|_{\partial \Omega \times (0, T)} = p^{-1}(p_D) =: \tau_D \geq \tau_*
\]

and the initial condition:

\[
\tau^0 = s^{-1}(s^0)
\]

So now the problem reads

\[
\begin{align*}
\partial_s \tau - \nabla \cdot \left( (\lambda(s(\tau)) g - \nabla u(\tau) \right) &= 0 \quad \text{in } Q \\
\tau|_{t=0} &= \tau^0 \\
\tau|_{\partial \Omega \times (0, T)} &= \tau_D
\end{align*}
\]

and its discretization is, \( \forall K \in T \):

\[
\frac{s(\tau^n_K) - s(\tau^{n-1}_K)}{\Delta t^n} m_K + \sum_{\sigma \in \mathcal{E}_K} F^n_{K, \sigma} = 0
\]

\[
\sum_{\sigma \in \mathcal{E}_K} F^n_{K, \sigma} = \sum_{\sigma \in \mathcal{E}_K} \left( m_{\sigma} \lambda(s(\tau^n_K)) g_{K, \sigma}^+ - \lambda(s(\tau^{n-1}_K)) g_{K, \sigma}^+ + \sum_{\sigma \in \mathcal{E}_K} A_{\sigma} (u(\tau^n_K) - u(\tau^{n-1}_K)) \right)
\]

(1.17)

1.4.1 The Newton Method

The numerical scheme (1.17), for all \( n \in \{1, \ldots, N\} \), lead us to the nonlinear system

\[
F_n(\tau^n) = (f^n_K(\tau^n))_{K \in T} = 0 \quad \text{with } F_n \in C^2(\mathbb{R}^T; \mathbb{R}^T)
\]

(1.18)

where, \( \forall K \in T \)

\[
f^n_K(\tau^n) = m_K \frac{s(\tau^n_K) - s(\tau^{n-1}_K)}{\Delta t^n} + \sum_{\sigma \in \mathcal{E}_K} \left( m_{\sigma} \lambda(s(\tau^n_K)) g_{K, \sigma}^+ - \lambda(s(\tau^{n-1}_K)) g_{K, \sigma}^+ + \sum_{\sigma \in \mathcal{E}_K} A_{\sigma} (u(\tau^n_K) - u(\tau^{n-1}_K)) \right)
\]

(1.19)

Solve (1.18) with the Newton method consists in the construction of a sequence \( (\tau^{n,k})_{k \geq 0} \) defined by:

\[
\begin{cases}
\tau^{n,0} = \tau^{n-1} \\
\tau^{n,k+1} = \tau^{n,k} - [J_{F_n}(\tau^{n,k})]^{-1} F_n(\tau^{n,k})
\end{cases}
\]

(1.20)

where \( J_{F_n} \) is the Jacobian matrix of \( F_n \) at \( \tau^n \) and we assume that it is not singular (we will show it after).

The Jacobian matrix is an \( \mathcal{T} \times \mathcal{T} \) matrix obtained exploiting the partial derivative w.r.t. \( \tau^n \) of (1.18):

\[
\begin{align*}
J_{F_n} &= \frac{\partial F_n(\tau^{n,k})}{\partial \tau^{n,k}} = \left( \frac{\partial f^n_K(\tau^n)}{\partial \tau^{n,k}} \right)_{K \in T, L \in T}
\end{align*}
\]

(1.21)

Starting from equation (1.19) we can write the expression of \( (J_{F_n})_K \):

\[
(J_{F_n})_K = \frac{m_K}{\Delta t^n} \frac{\partial s(\tau^{n,k}_K)}{\partial \tau^{n,k}_K} + \sum_{\sigma \in \mathcal{E}_K} \left( A_{\sigma} \left( \frac{\partial u(\tau^{n,k}_K)}{\partial \tau^{n,k}_K} - \frac{\partial u(\tau^{n,k}_\sigma)}{\partial \tau^{n,k}_\sigma} \right) \right) + \\
\sum_{\sigma \in \mathcal{E}_K} \left( m_{\sigma} \left( g_{K, \sigma}^+ \frac{\partial \lambda(s(\tau^{n,k}_K))}{\partial s(\tau^{n,k}_K)} - \frac{\partial \lambda(s(\tau^{n,k}_K))}{\partial s(\tau^{n,k}_\sigma)} \right) \frac{\partial s(\tau^{n,k}_\sigma)}{\partial \tau^{n,k}_\sigma} \right)
\]

So:

\[
(J_{F_n})_{K,K} = \frac{m_K}{\Delta t^n} \frac{\partial s(\tau^{n,k}_K)}{\partial \tau^{n,k}_K} + \sum_{\sigma \in \mathcal{E}_K} \left( A_{\sigma} \frac{\partial u(\tau^{n,k}_K)}{\partial \tau^{n,k}_K} + m_{\sigma} g_{K, \sigma}^+ \frac{\partial \lambda(s(\tau^{n,k}_K))}{\partial s(\tau^{n,k}_K)} \frac{\partial s(\tau^{n,k}_K)}{\partial \tau^{n,k}_K} \right) + \\
(J_{F_n})_{K,L} = - \left( A_{\sigma} \frac{\partial u(\tau^{n,k}_L)}{\partial \tau^{n,k}_L} + m_{\sigma} g_{K, \sigma}^+ \frac{\partial \lambda(s(\tau^{n,k}_L))}{\partial s(\tau^{n,k}_L)} \frac{\partial s(\tau^{n,k}_L)}{\partial \tau^{n,k}_L} \right)
\]
Chapter 2

The prototype

Our prototype is basically composed by four components: Mesh, Numerics, Physics and Visualisation. Let us analyze each of them.

2.1 Mesh

Our mesh structure is, potentially, adapt to also a 3D mesh. Actually we have only implemented constructors for 1D and 2D mesh but. When we will pass to 3D mesh, we will need just to add a specific constructor for the elements to manage this case. Let us now detail the cases which have been implemented.

2.1.1 2D mesh

For the mesh generation we use a finite element mesh generator. Knowing that we will use a finite volume scheme, we have created this support in order to store all the components of the mesh in an object and also evaluate some quantities of interest such as the volume of the cells, the center of the cells and so on. In Figure 2.1 we can see the structure of our code.

- The class Dof is a very simple class owning an int which is the identifier characterizing an element of the mesh. This class has been created just to be inherited by the other classes. For this reason the attribute m_id is defined as protected. Let us analyze the other classes. The class Node owns the attribute m_point which is a vector of double of size 3 × 1 containing the coordinates of the node. Concerning the methods, this class has a getter method to return each coordinate of the node and operator== in order to compare two node elements.

- The class Cell describes our mesh components: the triangles. It is characterized by its center, its volume and the vector of the ids of the nodes which are the vertices of a triangle. Finally we have a getter to return the center, the volume the number of nodes characterizing the cell and their ids, and an operator== to compare two cells.

- The class Face describes a line that is one edge of a cell. It is characterized by its center, its area, normal, a boolean flag with value true if it is a boundary face or false otherwise and, finally, the pointers to the cells sharing this face. Clearly, if $\sigma \in E_{\text{in}}$ we have that $\sigma = K \setminus L$ so it is shared by two cells called Cell0 and Cell1. Otherwise, if $\sigma \in E_{\text{ext}}$, the face will be owned only by Cell0. In Figure 2.2 we have represented the two cases. The normal of the face is oriented, for $\sigma \in K \setminus L$, from Cell0 (K) to Cell1 (L) and for $\sigma \in E_{\text{ext}}$ from Cell0 to outside. For each of these parameters we have a getter method.
CHAPTER 2. THE PROTOTYPE

Figure 2.1: Structure of Mesh class

Figure 2.2: On the left we see the case $\sigma = K \mid L$, on the right the case $\sigma \in \mathcal{E}_{\text{ext}}$
• The class Mesh contains, stored in vectors, all the elements which form the mesh. The constructor of this class takes the name of the file .msh of our mesh and, reading it, stores all needed informations.

Let us now explain how, starting from a Gmsh file, we create an object Mesh. Thanks to the fact that we know the structure of a .msh file, we read from the file the data we need and manipulate them in order to store the elements we need to generate our mesh. In Figure 2.3 we can see an example of .msh file. We divide the lecture of the file in seven part:

\begin{verbatim}
$MeshFormat
2.2 0 8
$EndMeshFormat
$Nodes
4
1 0 0
2 1 0
3 1.5 0.866025405 0
4 0.5 0.866025405 0
$EndNodes
$Elements
10
1 15 2 0 1 1
2 15 2 0 2 2
3 15 2 0 3 3
4 15 2 0 4 4
5 1 2 0 1 1 2
6 1 2 0 2 2 3
7 1 2 0 3 3 4
8 1 2 0 4 1 4
9 2 2 0 6 2 4 1
10 2 2 0 6 2 3 4
$EndElements
\end{verbatim}

Figure 2.3: Example of .msh file

- Part 0
  It is the MeshFormat part where we find the Gmsh version, the file-type and the data-size. We simply skip it.

- Part 1
  Here we enter in the section Nodes and we simply read and store the number of nodes we are going to read.

- Part 2
  In this part we have the node description: in each line we find the identifier number of the node and its coordinates. For each node, we store the element in a vector of nodes, ordered wrt the identifier number which begin from zero (so actually it will be the identifier read minus one). When the number of read nodes is equal to the nodes number, we pass to the next part.

- Part 3
  Here we enter in the section Elements where we will read data on the cells. Actually here we will find elements of different types: type 1 is a line, type 2 is a triangle, type 4 is a quadrangle, type 15 is a point. Actually we are interested only in type 2: triangles. Here we just read the total number of elements of different types.

- Part 4
  We skip all the elements which are not triangles, when we find the first one we pass to the next part calculating how many type 2 elements we have.

- Part 5
  Now for each line we create a cell element starting from the id-nodes which characterizes it and its identifier number (which starts from zero) and we store it in a vector of cells. What we will also create here is a set whose elements are pairs of identifier numbers characterizing a face. In this way, for each face, we will know which nodes compose it.

- Part 6
  We skip the rest of the file.
After these parts we manage the orientation of the normal, the boundary flags and we assign the pointer to \textit{Cell0} and \textit{Cell1} for each face.

### 2.1.2 1D mesh

For a 1D mesh, the scheme does not change. We have simply create a Face, Cell and Mesh constructor to manage this case:

- \textit{Face} constructor takes one node and the identifier number
- \textit{Cell} constructor takes two nodes and the identifier number
- \textit{Mesh} constructor takes the starting point, the space step $\Delta x$, the number of cells $N$ we want to create and the height of our domain.

![Figure 2.4: Example of 1D mesh with $N = 4$](image)
2.2 Numerics

In this folder we can find the numerical tools we will need, namely: TPFAScheme class, NonLinearSystem class and NewtonSolver class. The skeleton of these classes is reported in Figure 2.5. Let us analyze them deeply.

<table>
<thead>
<tr>
<th>TPFAScheme</th>
<th>NonLinearSystem</th>
<th>CoreyParameterizedModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>- const Mesh&amp; m_mesh</td>
<td># const Mesh&amp; m_mesh</td>
<td>- double m_switchTau</td>
</tr>
<tr>
<td>- vector&lt;double&gt; m_transmissivity</td>
<td># TPFAScheme m_pf</td>
<td>- double m_initTuser</td>
</tr>
<tr>
<td>+ double getTransmissivity(const int) const</td>
<td># SparseDoubleXX m_jacobian</td>
<td>- function&lt;double(const double)&gt; m_waterSaturationFunction</td>
</tr>
<tr>
<td></td>
<td># DoubleX m_residual</td>
<td>- function&lt;double(const double)&gt; m_waterSaturationDerivative</td>
</tr>
<tr>
<td></td>
<td># DoubleX m_actSolution</td>
<td>- function&lt;double(const double)&gt; m_kirchhoff</td>
</tr>
<tr>
<td></td>
<td># DoubleX m_prevSolution</td>
<td>- function&lt;double(const double)&gt; m_kirchhoffDerivative</td>
</tr>
<tr>
<td></td>
<td># const double m_deltat</td>
<td>- function&lt;double(const double)&gt; m_tauFromWaterSaturation</td>
</tr>
<tr>
<td></td>
<td># double m_time</td>
<td>+ double getWaterSaturation(const double) const</td>
</tr>
</tbody>
</table>

NewtonSolver

- shared_ptr<NonLinearSystem> m_ptr
- const int m_iter
- const double m_eps

+ bool solve(int &)

2.2.1 TPFAScheme

The objective of this class is to calculate the transmissivities of the faces of our mesh. We have defined the transmissivity coefficient \( \sigma \) when we have defined the flux discretization using the two-points flux approximation scheme and we can find its definition in Equation (1.6). How does it works? We simply pass a reference to the mesh to the constructor which perform a loop on the inner and outer faces and evaluate the transmissivity coefficient for each face. This coefficient is store in a vector in the position equal to the identifier of its face. Once created the object TPFAScheme we can ask for the transmissivity coefficient of the \( i \)-th face using the getter method giving him the identifier numeral \( i \).

2.2.2 NonLinearSystem

NonLinearSystem is an abstract class whose children will describe the nonlinear system to solve. Basically this class is made to assemble the Jacobian matrix for the system, evaluate the residual vector, get the previous and actual solution and the evaluated residual. Moreover it saves ad updates the time at which we are during the simulation. As we can clearly imagine, all this tools will be used in the Newton loop because we need the to solve \( (1,20) \). The methods \( \text{assemble()} \) and \( \text{updateResidual()} \) are pure virtual functions because their implementation will change wrt the chosen system to solve. However the structure of these methods will not change: see Algorithm (1) and (2). Functions \( \text{increaseDeltaTime()} \) and \( \text{decreaseDeltaTime()} \) are used at the end of the temporal loop to adapt the \( \Delta t \). If Newton converges, we increase \( \Delta t \), otherwise we decrease it without updating the time of the simulation and we recall NewtonSolver using the previous solution for the loop.
**Data:** assemble()
We initialize the jacobian matrix to zero;
We create a vector of triplets of double that we will use to create our sparse matrix;
Each element of this vector takes the indices of the position of the matrix where we want to store the value, and the value;

```plaintext
for each cell in the vector of cells of the mesh do
  we push back the prescribed value at position (cell-id,cell-id);
end

for each face in the vector of inner faces do
  we push back the prescribed values of diffusion and convection in diagonal and off-diagonal position;
end

for each face in the vector of outer faces do
  we push back the prescribed values of diffusion and convection in diagonal position;
end
```

We create the matrix: 

```
algorithm 1: Algorithm for assemble Jacobian matrix
```

```plaintext
```

**Data:** updateResidual()
We initialize the vector to zero;

for each cell in the vector of cells of the mesh do
  m_residual[id] += prescribed value;
end

for each face in the vector of inner faces do
  we add the prescribed values of diffusion and convection;
  m_residual[idCell0] += diffusionValue;
  m_residual[idCell1] -= diffusionValue;
  m_residual[idCell0] ± convectionValue;
  m_residual[idCell1] ∓ convtionValue;
end

for each face in the vector of outer faces do
  we add the prescribed values of diffusion and convection;
  m_residual[idCell0] += diffusionValue;
  m_residual[idCell0] += convectionValue;
end
```

Algorithm 2: Algorithm to update residual vector

**2.2.3 NewtonSolver**

This class contains the solve method which performs Newton’s iterations. When we create an object of this class we simply specify the maximum number of iterations to perform, the tolerance to use and a pointer, precisely a shared pointer, to the NonLinearSystem to solve. Then to perform the Newton method we simply call the function solve(int&T nIter). Let us analyze it: firstly we can notice that the return type is a boolean. It returns true is the method converges, false otherwise. The reference to the integer passed by argument will be updated at the end of the iterations-loop with the number of iterations done. In the test we will use it to control the number of iterations performed by the Newton solver. We now that at the beginning the previous solution and the actual solution are the same and corresponds to the initial one. Once entered in the solve method, we update the residual vector and we evaluate its infinity norm. Then there is the loop to evaluate the new iterative solution. We stay in the loop until the maximum number of iterations is reached or the norm of the residual vector becomes smaller than the tolerance. In the loop we first assemble the jacobian matrix and then we use the biconjugate gradient stabilized method to evaluate the increment solving \( F_n(x, k + \text{incr} = F_n(x) \). Then we update the new solution and the residual evaluating then its norm. When we exit the loop, if the number of iterations we have performed is smaller than the maximum number of iterations expected, it means that the method has converged and we return true. Otherwise we return false.
2.2.4 CancesBrooksCoreyParameterizedModel

We can define this class as a decorateator pattern of the object CancesBrooksCoreyModel (see Physics section): it has the same structure from the point of view of public methods but it menage them in a more complex way. When we introduce the parametrization we have to take into account the switch and we want that our Newton solver is not affected by it. It is here that emerge the utility of the CancesBrooksCoreyParameterizedModel: it uses the laws implemented in the CancesBrooksCoreyModel and menage them wrt the choosen formulation to use. Namely:

- \textbf{formulation} = 0 we solve the problem wrt the Kirchhoff transform $u$,
- \textbf{formulation} = 1 we solve using the automatic switch ($\tau$),
- \textbf{formulation} = 2 we solve the problem wrt the saturation $s$

About C++ tools used, in this class we use the lambda function one:

$$
[captures](parameters) \rightarrow ret\{body\}
$$

where

- \textit{captures} : we pass &, simple by-reference capture of the object. Notice that [&] and [&,this] are equivalent.
- \textit{parameters} : list of the parameters.
- \textit{ret} : is the return type.
- \textit{body} : function body.
2.3 Physics

In this folder we can find classes corresponding to the physical models we are going to use in our simulations. Here we can find the implementation of the physical laws used in the tests. Basically for the moment we are using the Brooks Corey model to describe the relation between the capillary pressure and mobility with the saturation. The skeleton of the two implemented model is reported in Figure 2.6. Each class has for each law a getter method: we pass the value in which evaluate the law and the method returns the result. Let us just focus on the check() method of the class BrooksCoreyModel. The target of this method is to emulate, in the code, this relation:

\[
\text{Denoting } s_{\text{eff}} = \frac{s - s_{rw}}{1 - s_{rn} - s_{rw}} \quad \text{we have that: } s = \begin{cases} 
    s_{rw} & \text{if } s < s_{rw} \\
    s & \text{if } s_{rw} \leq s \leq 1 - s_{rn} \\
    1 - s_{rn} & \text{if } s > -s_{rn} 
\end{cases}
\]

In CancesBrooksCoreyModel it is implemented for \( s \) (we use no more \( s_{\text{eff}} \)) in the getter methods using an if condition:

\[
\text{const double } s( s_{w} > 1. ? 1. : ((s_{w} < 0.) ? 0. : s_{w} )).
\]
2.4 Visualisation

Once we have solved our problem, it would be interesting to plot the obtained solution. The target of this folder is to provide a way to create a file .vtk to open with paraview and see the plot of the solution on the used mesh. For this reason the class in object is named MakeVTK. The skeleton of this class is reported in Figure 2.7. Since we know the structure of a .vtk file, we use the informations we have abouts the mesh, its elements and the the solutions’ values on each cell to write a file .vtk. This is briefly what the methods writeVTK and writeVTK1D do. They takes the name of the file to write, a vector of the solutions we want to plot and their labels, stored in the vector of string passed as last element. Here is an example of .vtk file. In the section POINTS are listed the coordinates of the nodes. Next to the word POINT we find how many number we will read and their type. Then in the section CELLS are listed the identifier numbers of the nodes characterizing each cell. Next to the word CELLS we find the number of the cells of our mesh and how many numbers we are going to read. Each line starts with the number of the identifiers we are going to read and themselves. Then in the section CELL-TYPE we find, for each cell, the number corresponding to the type of the cell. Type 5 is for triangles and type 3 for lines. Finally we have the solution values. SCALARS means that each value is a scalar value, then we find the label of the solution, the type of the value and how many values per line we will have. Then we have the list of the solution value associated to each cell. So the difference between the methods writeVTK and writeVTK1D is that the first, being for 2D mesh, will write 3 identifier numbers for each cell and cell type will be 5. The second method will write 2 identifier numbers for each cell and cell type will be 3. For the plot in time, we simply name the file with a number at the end: “myfile1.vtk”, “myfile2.vtk”, etc. In this way, when we open paraview in the folder in which these files are stores, it will see a file “myfile..<vtk” which is a file containing all the files we have created and if we click on play it will be shown the evolution in time of the solution.
Chapter 3

Tests

3.1 Test for the mesh

3.1.1 Test for 2D mesh

In order to verify the correct assemblage of the Mesh, we pass to the constructor the file reported in Figure 2.3 which encodes the mesh in Figure 3.1. Then we use the `assert` function in order to verify all the elements characterizing the mesh. The `assert` function works in this way: if the expression of this macro is false, a message is written to the standard error and the abort function is called, terminating the program execution.

![Figure 3.1: Mesh used in the test for the 2D mesh](image)

3.1.2 Test for 1D mesh

In order to verify the correct assemblage of the Mesh, we pass to the constructor the needed parameter and we check all the quantities characterizing the elements of the mesh using the function `assert`. 
3.2 Test for the Laplacian problem

The construction of our solver starts taking into account the solution just of the laplacian with the TP scheme. Our test problem for $s = s(x, y)$ in $\Omega = (0, 1) \times (0, 1)$ reads:

$$\begin{align*}
-\Delta s &= 0 \quad \text{in } \Omega \\
\left. s \right|_{\partial \Omega} &= x + y
\end{align*} \quad (3.1)$$

The exact solution of the problem is $s = x + y$. We simulate this problem on the mesh reported in figure 3.2. In the source code for this test (LaplacianTest.cxx) we create an object Mesh and TPFA Scheme in order to evaluate the transmissivity for each face of the mesh. Then we assemble the matrix $A$ which is defined as a Sparse Matrix of doubles of dynamic size. We also assemble the vector $b$ and we solve the problem using the SparseCholesky module of Eigen. After having solved the system we create an object MakeVTK which can write a file .vtk to plot the solution using paraview. Such plot is showed in figure 3.3. What we can observe is that the two points scheme is exact for linear solutions.

![Figure 3.2: Mesh with size $dx = \frac{1}{18}$](image-url)
Figure 3.3: From left to right, on top, plot of the exact solution and of the numerical one. On bottom, plot of $|\text{exact}_{sol} - \text{numerical}_{sol}|$. 
3.3 Test for Newton method on a simplified linear Richards equation

This test has been performed because we want to test our Newton solver for a very simple case. Our problem for \( s = s(x, y, t) \) in \( \Omega = (0, 1) \times (0, 1) \), for \( t \in (0, 0.05) \) reads:

\[
\begin{align*}
\partial_t s - \Delta s &= 0 \quad \text{in } \Omega \\
\left. s \right|_{\partial \Omega} &= s_D
\end{align*}
\]

We check this problem for a stationary solution, namely \( s = x + y \). So, on the boundary \( s|_{\partial \Omega} = x_\sigma + y_\sigma \) where \((x_\sigma, y_\sigma)\) is the center of the face \( \sigma \in \partial \Omega \).

We want to solve it using Newton’s method. Actually we have to solve:

\[ F_n(s^n) = (f_K^n(s^n)) = 0 \quad \forall \ K \in T \]

where

\[ f_K(s^n) = (s^n_K - s^{n-1}_K)m_K + \Delta t \sum_{\sigma \in \mathcal{E}_K} F_{K,\sigma} \]

Let us evaluate the Jacobian matrix \( J = \left( \frac{\partial f_K(s^n)}{\partial s^n_L} \right)_{K, L \in T} \):

\[ J_{KK} = \frac{\partial f_K(s^n)}{\partial s^n_K} = m_K + \Delta t \sum_{\sigma \in \mathcal{E}_K} A_\sigma \]

\[ J_{KL} = \frac{\partial f_K(s^n)}{\partial s^n_L} = -A_\sigma \]

For each time \( t \in (0.0, 0.05) \), with \( \Delta t = 5 \cdot 10^{-5} \) eps = 10^{-16} \( n_{max} = 20 \), we will perform Newton while-cycle:

Initialization: \( s^{n,0} = s^{n-1} \);

\[ \text{norm} = ||F_n(s^n)||_\infty; \]

\[ \textbf{while} \ (\text{number of iteration} < n_{max} \textbf{ and norm} > \text{eps}) \textbf{ do} \]

\[ s^{n,k+1} = s^{n,k} - J^{-1}(s^{n,k}) F_n(s^{n,k}); \]

\[ \text{evaluate } F_n(s^{n,k+1}); \]

\[ \text{update norm} = ||F_n(s^n)||_\infty; \]

\[ \textbf{end} \]

We simulate this problem on the mesh reported in figure 3.4. Being the solution stationary, the result will be the same for each time. In the source code for this test (LinearRichardTest.cxx) we create an object Mesh, a pointer to an object LinearRichards (son of the abstract class NonLinearSystem), in order to create an object of type NewtonSolver. After it we can define the initial solution which correspond, in this case, to the exact solution evaluated at the center of each cell. Then in a for loop we call the fuction solve of NewtonSolver. In this method the Jacobian matrix and the residual vector are assembled and we perform Newton’s iterations. At the end of the for loop we create an object MakeVTK to create the .vtk file to open with paraview. After having solved the system we create an object MakeVTK to call the function to write the file .vtk to plot the solution using paraview, showed in figure 3.5.

For the point of view of Newton loop, we expect that the method converges in just one iteration because we are simulating a linear case. This expectation is satisfied and we check it with an assert call in the test.
3.4 Test for Newton method on a linear Richards equation

Our problem for \( s = s(x, y, t) \) in \( \Omega = (0, 1) \times (0, 1) \), for \( t \in (0, 0.05) \) reads:

\[
\begin{aligned}
\partial_t s + \nabla \cdot (s e_y - \nabla s) &= 0 \quad \text{in } \Omega \\
|s|_{\partial \Omega} &= s_D
\end{aligned}
\]

We check this problem for a stationary solution, namely \( s(x, y, t) = e^{-\alpha + \frac{1}{2} \sqrt{\pi} \cos(\pi y) + \frac{1}{2} \sin(\pi y)} + \pi e^y - \frac{1}{2} \). So, on the boundary \( s|_{\partial \Omega} = s(x_\sigma, y_\sigma, t) \) where \( (x_\sigma, y_\sigma) \) is the center of the face \( \sigma \in \partial \Omega \). We want to solve it using Newton’s method. Actually we have to solve:

\[
F_n(s^n) = (f_K(s^n)) = 0 \quad \forall K \in T
\]

where

\[
f_K(s^n) = (s^n_K - s^{n-1}_K)m_K + \Delta t \sum_{\sigma \in E_K} F_{K,\sigma}
\]

Let us evaluate the Jacobian matrix \( J = \left( \frac{\partial f_K(s^n)}{\partial s^n_L} \right)_{K,L \in T} \):

\[
J_{KK} = \frac{\partial f_K(s^n)}{\partial s^n_K} = m_K + \Delta t \sum_{\sigma \in E_K} (m_\sigma(e_y^{K,\sigma})^- + A_\sigma)
\]

\[
J_{KL} = \frac{\partial f_K(s^n)}{\partial s^n_L} = -(m_\sigma(e_y^{K,\sigma})^- + A_\sigma)
\]

For each time \( t \in (0.0, 0.05) \), with \( \Delta t = 5 \cdot 10^{-5} \) \( \text{eps} = 10^{-16} \) \( n_{\text{max}} = 20 \), we will perform Newton while-cycle:

Initialisation: \( s^{n,0} = s^{n-1} \); \( \text{norm} = ||F_n(s^n)||_\infty \);

\[\text{while (number of iteration < n_{\text{max}} and norm > eps) do}\]

\[\quad \text{evaluate } F_n(s^{n,k+1}); \quad \text{update } \text{norm} = ||F_n(s^n)||_\infty;\]

\[\text{end}\]

We simulate this problem on the mesh reported in figure 3.6. In the source code for this test (Linear-RichardTest.cxx) we create an object Mesh, a pointer to an object LinearRichards (son of the abstract class NonLinearSystem), in order to create an object of type NewtonSolver. After it we can define the initial solution which correspond, in this case, to the exact solution evaluated at the center of each cell at time \( t = 0 \), showed in Figure 3.7. Then in a for loop we call the function solve of NewtonSolver. In this method the Jacobian matrix and the residual vector are assembled and we perform Newton’s iterations. At the end of the for loop we create an object MakeVTK to create the .vtk file to open with paraview. After having solved the system we create an object MakeVTK to call the function to write the file .vtk for plot the solution using paraview, showed in figure 3.8.
Figure 3.5: From left to right, on top plot of the exact solution and the numerical one. On bottom plot of $|exact_{sol} - numerical_{sol}|$. 

CHAPTER 3. TESTS

21
CHAPTER 3. TESTS

Figure 3.6: Mesh with size $dx = \frac{1}{18}$

Figure 3.7: Initial solution
Figure 3.8: On the left column the exact solution, on the right one the numerical solution, from top to bottom, at $t = \Delta t$, $t = 500\Delta t$, $t = T_{fin}$

Figure 3.9: Plot of $|\text{exact}_{sol} - \text{numerical}_{sol}|$ at final time
In Figure 3.9 we can notice that we have obtained a quite good approximated solution. Let us perform a convergence analysis. We will simulate the problem of four mesh having size $\in (0.2, 0.1, 0.005, 0.0025)$. We obtain the following results:

- size = 0.2 \quad l2error = 0.0193822
- size = 0.1 \quad l2error = 0.00786937
- size = 0.05 \quad l2error = 0.00307009
- size = 0.025 \quad l2error = 0.00152773

In Figure 3.10 we can see the plot showing the rate of convergence, obtained using the formula $\alpha = \frac{||s_h - s_{exact}||_{L^2}}{||s_{h,2} - s_{exact}||_{L^2}}$, that is $\sim 1$. So we linearly converge in space. Why do we not converge quadratically? For this scheme a quadratic convergence in $L^2$ norm is expected. We know that $||e_h||_{L^2} = \sqrt{\int_K |s_{exact} - s_h|dK}$, but we evaluate this using a quadrature rule: $||e_h||_{L^2} = \sqrt{\sum_K |K||s_{exact} - s_h|}$. Here the exact solution is thought to be evaluated in the gravity center of the cells. In our code, we evaluate it in the center of the circumscribed circle. So we are introducing an approximation in addition to the one introduced by the quadrature rule. So we think that this is the reason why we lose the quadratic rate. In Figure 3.11 we can see the difference between the exact solution and the numerical one at the final time, for the various meshes. We can notice that in the last images almost all the cells are at zero, but there are still a few of them characterized by an error of the order of $10^{-2} \sim 10^{-3}$. If we carefully analyze them we can notice that they are not far from being right-angled triangles and here is the problem: the two point scheme is valid if it is applied to a mesh which satisfies the orthogonality condition. In this case these cells do not respect completely this property and for this reason we notice this “irregularities” on the solution. Moreover it influences the convergence because we do not respect the condition $d_{K,\sigma} \geq \psi d_\sigma$, with $\psi > 0$, $d_{K,\sigma} = d(x_K, x_\sigma)$, $d_\sigma = d(x_K, x_L)$ (see [2]). Unfortunately we can not control this fact because we do not have the possibility to impose some condition when creating the mesh with Gmsh. A way to avoid this problematic should be use a mesh made by squares rather than triangles.
### 3.5 Test for Newton method on Richards equation

Our problem for $s = s(x, y, t)$ in $\Omega = (0, 1) \times (0, 1)$, for $t \in (0.0, 0.05)$ reads:

\[
\begin{align*}
\partial_t s + \nabla \cdot (\lambda(s) g - \nabla u(s)) &= 0 \quad \text{in } \Omega \\
\nabla u(s) \cdot n &= 0 \quad \text{on } \partial \Omega \\
\lambda(s) g \cdot n &= \lambda(s_D) g \cdot n \quad \text{if } g \cdot n < 0 \quad \text{on } \partial \Omega
\end{align*}
\]

where

- $u(s)$ is the Kirchhoff transform which is defined by the relation: $u(s) = \int_0^s \lambda(\tau) \nabla p(\tau) d\tau$
- $p(s)$ is the capillary pressure of water
- $s$ is the water saturation
- $\lambda(s)$ the mobility function

In this simulation we will use the *Brooks Corey* model for mobility and capillary pressure which reads:

\[
\begin{align*}
\lambda(s) &= \frac{s_{eff}^{2+3\lambda}}{\lambda_{eff}} \\
p(s) &= p_e s_{eff}^{-\frac{1}{4}} \\
\text{with } s_{eff} &= \frac{s - s_{rw}}{1 - s_{rn} - s_{rw}}
\end{align*}
\]

where

- $s_{eff}$ is the effective saturation,
- $s_{rw}$ is the residual saturation of wetting phase,
- $s_{rn}$ is the residual saturation of nonwetting phase,
• \( p_{e} \) is the entry pressure, also called bubbling pressure.

**Remark** The capillary pressure is defined as \( p = p_{nw} - p_{w} \), where \( p_{nw} \) is the non-wetting pressure (air) and \( p_{w} \) is the wetting pressure (water). In our case \( p_{nw} \sim 0 \) so \( p < 0 \).

Moreover we have that:

\[
s = \begin{cases} 
  s_{rw} & \text{if } s < s_{rw} \\
  s & \text{if } s_{rw} \leq s \leq 1 - s_{rn} \\
  1 - s_{rn} & \text{if } s > 1 - s_{rn}
\end{cases}
\]

in order to have \( s_{\text{eff}} \in [0, 1] \). In this case, performing calculations, the Kirchhoff transform reads:

\[
u(s) = -\frac{p_{e}}{1 + 3\lambda} s_{\text{eff}}^{\frac{1+3\lambda}{\lambda}}
\]

In the simulation we take \( s_{rw} = s_{rn} = 0.1 \), \( \lambda = 2 \) and \( p_{e} = -0.5 \). The plots of mobility function, Kirchhoff transform and their derivatives are reported in Figure 3.12.

Finally we impose the following initial condition for \( s \): let \( C \) the circle having center in \((0.5, 0.5)\) and radius \( r = 1 \)

\[
s^{0} = \begin{cases} 
  0.8 & \text{in } C \\
  0.5 & \text{in } \Omega \setminus C
\end{cases}
\]

Now the discretize problem is the one we have reported in Equation (1.10) except for the fact that the diffusive term now is null on the boundary faces \( \sigma \in \mathcal{E}_{\text{ext}} \):

\[
\sum_{\sigma \in \mathcal{E}_{K}} \int_{\sigma} \nabla u(s_{K}^{n}) \cdot n_{K,\sigma} d\sigma = \sum_{\sigma \in \mathcal{E}_{K} \cap \mathcal{E}_{\text{int}}} \int_{\sigma} \nabla u(s_{K}^{n}) \cdot n_{K,\sigma} d\sigma + \sum_{\sigma \in \mathcal{E}_{K} \cap \mathcal{E}_{\text{ext}}} \int_{\sigma} \nabla u(s_{K}^{n}) \cdot n_{K,\sigma} d\sigma
\]

The Newton system we are going to solve is reported in Equation (1.20). In NonLinearRichardsTest.cxx we simply create all the needed object, we initialize \( s \) with the initial condition and in the temporal loop we call the `solve` method of the class `NewtonSolver` to perform Newton’s iterations. For the moment we use a fixed \( \Delta t \) and, if Newton converges, we pass to the next time, otherwise we quit the for loop. The results are reported in Figure 3.13.

It is interesting to see the effect of the gravity comparing this simulation with the one done taking \( g = 0 \) reported in Figure 3.14. We can clearly notice that, with non-null gravity, the pattern of the solution is the same of the simulation without gravity but simply moved towards the bottom.

For the convergence of the Newton method, we use the following formula to estimate the rate of convergence (see [3]):

\[
\alpha \approx \frac{\log(||s^{n,k+1} - s^{n,k}||_{L^2})}{\log(||s^{n,k} - s^{n,k-1}||_{L^2})} = \frac{\log(||s^{n,k} - s^{n,k-1}||_{L^2})}{\log(||s^{n,k-1} - s^{n,k-2}||_{L^2})}
\]

and we obtain a rate of convergence \( \sim 2 \). In Figure 3.15 we can see the behavior of the infinity norm of the residual during Newton’s loop. At the beginning of the simulation it performs three iterations but then they becomes only two. This means that the initial condition we give is quite far from a good “initial guess”.

The simulation of this test case takes 21.42 sec (in release mode) but, if we adapt the \( \Delta t \) using methods `increaseDeltaTime()` and `decreaseDeltaTime()`, the simulation time decrease to 0.85 sec. However Newtons performs, in this case, one more iteration.
Figure 3.12: Plot of the laws of Brooks Corey model used in our test.
Figure 3.13: Simulation of Richards equation, with gravity, at $t \in [0, 92 \cdot \Delta t, 366 \cdot \Delta t, 574 \cdot \Delta t, T_{fin}]$
Figure 3.14: Simulation of Richards equation, without gravity, at $t \in [0, 92 \cdot \Delta t, 366 \cdot \Delta t, 574 \cdot \Delta t, T_{fin}]$
Figure 3.15: Behavior of infinity-norm of the residual vector at the beginning, top, and after some time-steps, bottom.
3.6 Test for Newton method on Richards equation with parametrization

This test is based on first test case of the article [1]. Our problem for $s = s(x, y, t)$ in $\Omega = (0, 1) \times (0, 1)$, for $t \in (0, 0.7)$ reads:

$$
\begin{align*}
\frac{\partial s}{\partial t} + \nabla \cdot (\lambda(s)\mathbf{g} - \nabla u(s)) &= 0 & \text{in } \Omega \\
p &= p_D & \text{on } \Gamma_D \\
(\nabla u(s) + \lambda(s)\mathbf{g}) \cdot \mathbf{n} &= 0 & \text{on } \partial \Omega \setminus \Gamma_D \\
s|_{t=0} &= s_0 & \text{in } \Omega
\end{align*}
$$

Moreover we have:

- mobility function $\lambda(s) = s^3$,
- saturation function $S(u) = \min \left(1, \left(\frac{u - u_*}{u_h}\right)^{\frac{1}{3}} \right)$ for $u \geq u_*$.

Let us introduce $\hat{u}$, $\hat{s}$:

$$
\begin{align*}
\hat{u}(\hat{\tau}) &= \begin{cases}
\hat{\tau} + \tau_s & \text{if } \hat{\tau} \geq \tau_s, \\
u_b \hat{\tau}^\beta + u_* & \text{if } \hat{\tau} \in [0, \tau_s]
\end{cases} \\
\hat{s}(\hat{\tau}) &= \begin{cases}
\hat{S}(\hat{\tau} + \tau_s) & \text{if } \hat{\tau} \geq \tau_s, \\
\hat{\tau} & \text{if } \hat{\tau} \in [0, \tau_s]
\end{cases}
\end{align*}
$$

where $\hat{\tau} = \tau - \tau_s$ with $\tau_s = \min \left(\frac{1}{u_h}, 1\right)$, $s$ stands for switch, and $\tau_* = u_b \tau_s^\beta + u_* - \tau_s$. We define the $\tau$-formulation by $u(\tau) = \hat{u}(\tau - \tau_s)$ and $s(\tau) = \hat{s}(\tau - \tau_s)$.

So, while $\hat{\tau} \in [0, \tau_s]$ we are in variable saturation, when $\hat{\tau} \geq \tau_s$ we pass to variable Kirchhoff transform. This parametrization satisfy the target of keep bounded the derivatives of the saturation and Kirchhoff transform. For this test we take:

$s_0(x) = 10^{-6}$, $p_D = 1$, $\Gamma_D = \{(x_1, x_2) | x_1 \in (0, 0.3), x_2 = 1\}$, $\mathbf{g} = -\nabla x_2$, $u_b = 10^{-1}$, $u_* = -1$, $\beta = 4$.

The formulation of the discretized problem we will solve using Newton’s method is the one written in Section 1.4.1. Let us analyze some elements of the code regarding the assemblage of the Jacobian matrix and the residual vector. The principal difference is the assemblage of the diffusive term for the boundary faces of $\Gamma_D$ because we perform an upwind discretization. We start from

$$
\partial_t s(\tau) - \nabla \cdot (\lambda(s(\tau)) (\nabla p(\tau) - \mathbf{g})) = 0
$$

Let us focus on the diffusive term. We integrate over $K$ and apply Stokes theorem obtaining:

$$
- \int_K \nabla \cdot (\lambda(s(\tau_K)) \nabla p(\tau_K)) dK = - \int_{\partial K} \lambda(s(\tau_{\partial K})) \nabla p(\tau_K) \cdot \mathbf{n}_K d\sigma = - \sum_{\sigma \in \mathcal{E}_K} \int_{\sigma} \lambda(s(\tau_\sigma)) \nabla p(\tau_K) \cdot \mathbf{n}_{K,\sigma} d\sigma
$$

Let us consider all $\sigma \in \Gamma_D$

$$
- \sum_{\sigma \in \Gamma_D} \int_{\sigma} \lambda(s(\tau_\sigma)) \nabla p(\tau_K) \cdot \mathbf{n}_{K,\sigma} d\sigma
$$

In the assemblage matrix:

- if $A_\sigma(p(\tau_K) - p(\tau_D)) \geq 0$: diffusion $= \lambda(s(\tau_K)) A_\sigma p'(\tau_K) + \lambda'(s(\tau_K)) (A_\sigma(p(\tau_K) - p(\tau_D)))$,
- if $A_\sigma(p(\tau_K) - p(\tau_D)) < 0$: diffusion $= \lambda(s(\tau_K)) A_\sigma p'(\tau_K)$.

In the evaluation of the residual vector:

- if $A_\sigma(p(\tau_K) - p(\tau_D)) \geq 0$: diffusion $= \lambda(s(\tau_K)) (A_\sigma(p(\tau_K) - p(\tau_D)))$,
- if $A_\sigma(p(\tau_K) - p(\tau_D)) < 0$: diffusion $= \lambda(s(\tau_K)) (A_\sigma(p(\tau_K) - p(\tau_D)))$. 

Figure 3.16: Simulation in variable saturation using upwind discretization at time $t \in [0, \Delta t, 30 \cdot \Delta t, 60 \cdot \Delta t, 128 \cdot \Delta t, T_{\text{fin}}]$
Figure 3.17: Simulation in variable saturation not using upwind discretization at time $t \in [0, \Delta t, 30 \cdot \Delta t, 60 \cdot \Delta t, 128 \cdot \Delta t, T_{\text{fin}}]$
We have performed this discretization in order to use directly the given boundary condition that is for the pressure variable \( p_D = p(\tau_D) \). If we do not resort to upwind discretization, it is necessary to obtain the boundary condition for \( \tau \) inverting the law \( \tilde{S}(u) \) and exploiting the relation between \( u \) and \( p \). Finally this formulation results to be a little bit more diffusive. Here we show the results obtained solving the equation in saturation variable, using upwind discretization (Figure 3.16) and not (Figure 3.17) in order to show the resulting effect.

Now let us solve the problem using the automatic switch. Results are showed in Figure 3.18 for the saturation variable and in Figure 3.19 for the Kirchhoff transform variable.

Figure 3.18: Simulation with automatic switch. Variable saturation at time \( t \in [0, \Delta t, 30 \cdot \Delta t, 60 \cdot \Delta t, 128 \cdot \Delta t, T_{fin}] \)

We can observe, comparing Figure 3.16 and 3.18 that the obtained solution is exactly the same.

In this test we use the \( \Delta t \)-adaptation and it takes 4.9 sec, starting from \( \Delta t = 5 \cdot 10^{-4} \). We have estimate the rate of convergence, keeping \( \Delta t \) fixed, using formula

\[
\alpha \approx \frac{\log(||s_{n,k}^{k+1} - s_{n,k}^{k}||_{L^2}/||s_{n,k}^{k} - s_{n,k}^{k-1}||_{L^2})}{\log(||s_{n,k}^{k} - s_{n,k}^{k-1}||_{L^2}/||s_{n,k}^{k-1} - s_{n,k}^{k-2}||_{L^2})}
\]

obtaining \( \alpha \sim 2 \). For each time, Newton’s method performs 4 iterations. In Figure 3.20 we can see the behavior of the infinity norm of the residual during Newton’s loop.
Figure 3.19: Simulation with automatic switch. Variable Kirchhoff transform at time $t \in [0, \Delta t, 30 \cdot \Delta t, 60 \cdot \Delta t, 128 \cdot \Delta t, T_f n]$
Figure 3.20: Behavior of infinity-norm of the residual vector
Chapter 4

Compilation

In the README file we have explain how to compile the code. For the generation of a makefile, we use CMake, which is a cross-platform for managing the build process of a software using a compiler-independent method. In the Darcy directory we have the principal CMakeList where we set if to work in release or debug mode, we specify the code-language used (C++11), we include the directories and import the used external libraries (Eigen in our case). Moreover we add the source directory for compilation and we add libraries to link with the executable. For our test, they can be automatically run thanks to the ctest. The test to perform are specified in the CMakeList of the directory Darcy. Here an example of command to add a test in the CMakeList:

```cpp
enable_testing()
#
# Unit tests
#
add_test(NAME MeshUnitTest COMMAND ${CMAKE_BINARY_DIR}/bin/Darcy MeshUnitTest
          WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}/test/MeshUnitTest)
add_test(NAME LaplacianUnitTest COMMAND ${CMAKE_BINARY_DIR}/bin/Darcy
         LaplacianUnitTest WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}/test/LaplacianUnitTest)
```

Finally, we have cite the library Eigen. Our points, vectors and matrices types are taken from this library and, in ./src/Types.h, are defined the typedef we use for these types.

- `typedef Eigen::Matrix< double, 3, 1 > Double3` is a vector of double of fixed size \(3 \times 1\). We use it to represent a point coordinates;
- `typedef Eigen::SparseMatrix< double > SparseDoubleXX` is a sparse matrix of dynamic size;
- `typedef Eigen::VectorXd DoubleX` is a vector of double of dynamic size. It is the type of vectors like the residual vector in Newton Solver or the problem’s solution;
- `typedef Eigen::Triplet< double > DoubleTriplet` is a triplet of double. It is necessary for the assemblage of a sparse matrix.
Bibliography

