Finite volume methods for fluid flow in porous media

Bachelor Thesis

Fabian Mönkeberg

Supervisor: Prof. Dr. Nils Henrik Risebro,
Prof. Dr. Ralf Hiptmair

ETH Zürich,
June 27, 2012

Acknowledgments

This thesis was done to receive the Bachelor of Science in Mathemetics.
I first thank my supervisor Prof. Dr. Nils Henrik Risebro for motivating me and for his good advices. I also thank Prof. Dr. Ralf Hiptmair for his support. Finally, I thank Simon Laumer for all the corrections.

Contents

Abstract

This thesis develops a routine to compute finite volume methods on triangular grids for solving hyperbolic partial differential equations. The implementation in this thesis is based on the Engquist-Osher scheme, but it can be extended to any other finite volume method.
In the second part, we test the implementation against a benchmark dataset for reservoir simulation of the Society of Petroleum Engineers. This dataset is used to define an immiscible and incompressible two-phase flow in a rectangular domain. We had to implement a fast elliptic solver, due the needed split in an elliptic and one hyperbolic equation.
The focus of these thesis is the implementation, not the convergence theory.

Chapter 1

Introduction

The topic of this thesis is first the development of a program of a finite volume method on a triangular mesh. Second focus was its verification based on simulations of flow in porous media. Nowadays, an approved method is the finite volume method on a Cartesian grid. The advantage of a method on a Cartesian grid is its simple handling, due to its nice structure. On the other hand, the advantage of a method on a triangular mesh is a certain flexibility. Being more precise, there exist several simple strategies for local refinements. Examples of such local refinements are shown in figure 1.1 and 1.2. In this figures $*$ denotes the triangle that should be refined. Note that, in order to receiving a correct triangulation, we can not just refine one single triangle. We have to guarantee, that the mesh is complete. Local refinements are used to guarantee convergence, without refining the whole grid.

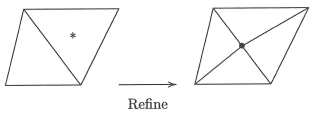


Figure 1.1: local refinement of triangular grid, [6]

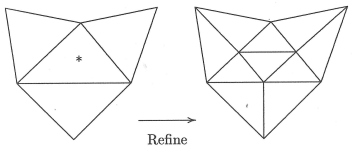


Figure 1.2: local refinement of triangular grid, [6]

An other approach, which can be used on triangular grids, is local grid align-

ment. There you align the triangles in the direction of the flux or parallelize them according to the shock waves. A shock wave is a discontinuous wave. This approach improves the accuracy, too. The disadvantage is exactly that there are a lot of possibilities to build a triangulation, and so we have no fix topology. The aim is to find a method to get access to the informations of this topology (triangular elements, neighbors of triangles, size of triangles,...).

Chapter 2

Modeling fluid flow in porous media

The interest in the behavior of fluid in porous media is not a recency. It started in the early days of the oil production out of reservoirs deep under the surface of the earth. The main purpose in modeling fluid flow in porous media is to simulate the flow of oil and water in the rock, to estimate and optimize the oil and gas production. Another application is the simulation of groundwater contamination. The existence of many depends on the subsistence of groundwater. Problems appear from leaking tanks of chemicals, oil pipelines, or fertilizers. In general, the whole pollution of the environment is problematic for the ground water. The basic process is that we have clean groundwater as one medium and polluted water, or even an toxic fluid as the second medium. To understand the spreading of the second medium and its consequences is the aim.
The challenge is to model an interaction of different fluids in their phases (liquid, gaseous, solid) and the environment. Basically, this means to simulate fluid flow and mass transfer, while some external and internal forces, like gravity, capillary and viscous forces, affect.
The mathematical model of this physical system is set by differential equations and some special boundary conditions. To get these we use the fundamental rules of conservation of mass, momentum and often we apply Darcy's law (p.5) for each phase, instead of Newton's second law. The first challenge is to get an accurate prediction of the flow scenarios, this means to receive the properties of the different media and its interaction. Nowadays, their is an amount of new and reliable techniques for getting these informations. But handling this huge mass of data is the next difficult task. It is just impossible to run simulations with all this information. This is because of the still limited computer resources. So a suitable approximation is the key concept to solve this task. For further informations, this chapter is based on the books [10], [2], [11] and the article [5].

2.1 Rock and fluid properties

The *rock compressibility* c_r is a measure of how much the rock is compressible and is defined by:

$$c_r = \frac{1}{\phi}\frac{d\phi}{dp}, \tag{2.1}$$

where p is the overall reservoir pressure and ϕ, the *porosity*, is the void volume fraction of the medium, $0 \leq \phi < 1$. Often, in simplified models, the rock compressibility is neglected and the porosity ϕ is assumed to depend only at the spatial coordinates. In the other cases, it is common to use a linearization of ϕ:

$$\phi = \phi_0(1 + c_r(p - p_0)). \tag{2.2}$$

Usually, one assumes that the porosity is a piecewise continuous spatial function, since the dimensions of the pores are very small compared to any scale of the simulation.
The next important property of rock is the *(absolute) permeability* K, which is a measure of the ability to transmit a single fluid at certain conditions. Because the permeability does not have to be the same in each direction, K is a tensor, e.g. in shale a fluid flows easily in the direction of the surface, but not through it. In contrast to shale, the water flows easily through sandstone in each direction. As you see in the example of the shale, the permeability is not necessarily proportional to the porosity, but they are often strongly correlated to each other.
The medium is called *isotropic*, if K is independent of the direction. K is modeled in this case as scalar. The opposite is called *anisotropic*. If a rock formation, like sandstone transmit fluids readily, then they are called *permeable*. Otherwise they are called *impermeable*.
Let us assume that the rock pores are always filled with certain fluid or gas, meaning that there exists no vacuum. Then we define the *saturation* s_i of each phase as the volume fraction occupied by it. And we get that:

$$\sum_{\text{all phases}} s_i = 1. \tag{2.3}$$

Each phase contains one or more components, e.g. methane, ethane, propane, etc. are hydrocarbon components. Since the number of them can be quite large and they often have similar properties, it is common to group some components into pseudo-components. We will make no difference between components and pseudo-components. The mass fraction of component i in phase j is denoted by c_{ij}. For N different components in phase j, we get:

$$\sum_{i=1}^{N} c_{ij} = 1. \tag{2.4}$$

The *density* ρ and *viscosity* μ to each phase are functions of the *phase pressure* p_i and the composition of each phase.

$$\rho_i = \rho_i(p_i, c_{1i}, \ldots, c_{Ni}), \quad \mu_i = \mu_i(p_i, c_{1i}, \ldots, c_{Ni}). \tag{2.5}$$

The *compressibility* of the phase is defined as the compressibility for rock:

$$c_i = \frac{1}{\rho_i}\frac{d\rho_i}{dp_i}. \tag{2.6}$$

It is obvious that for some fluids, the compressibility effects are more important than for others. Similarly, some fluids depend more on the density, pressure and component composition than others. For example, the dependencies for the water phase are usually ignored, and the compressibility is more important for the gas phase than for the oil and water phase.
Furthermore, it is obvious that the restriction of the motion of one phase at a certain location depends on the presence of the other phases. This information is stored in the *relative permeability* k_{ri}, which describes how one phase i flows in the presence of the others. This means that it is a function depending on the saturations of the other phases. The relative permeability defines the *(effective) permeability* $K_i = Kk_{ri}$ experienced by phase i. To prevent misunderstanding, it has to be mentioned that the k_{ri} are nonlinear functions of the saturations, so that the sum of all relative permeabilities is not necessarily one. The critical saturation at which a phase starts or stops to move, is called the *residual saturation*.
The discontinuity in fluid pressure occurs across an interface between any two immiscible fluids, e.g. water and oil. This discontinuity is called the *capillary pressure* $p_{cij} = p_i - p_j$.

From now on, we concentrate on the production of oil and its reservoir simulation. The problem setting here is a bounded reservoir space, filled with gas, oil and water. To produce the oil, you will do the following: First, drill some wells, from which the oil will flow out by means of overpressure. This is called the *primary production*. At a certain point of time, the pressure gets to a steady state and the oil stops flowing out. To get the rest of the oil, one starts the *secondary production*: Water or gas is injected into the reservoir to receive again an higher pressure in the reservoir, so the production of the oil continues. Each of these techniques produces about 20 percent of the oil. In order to produce even more oil, one uses the *Enhanced Oil Recovery (EOR, or tertiary recovery)*: There the idea is to change the flow properties of water and oil, to push them out. Therefore one injects some chemicals or even try to heat up the reservoir. But so far these methods are too expensive for large commercial use and are still in test stage.
One important attainment in this topic was *Darcy's law*, which states that the *volumetric flow density* v is proportional to the gradient of the fluid pressure and a pull-down effect due to gravity:

$$v = -\frac{K}{\mu}(\nabla p + \rho g \nabla D), \tag{2.7}$$

where ∇p is the gradient of the pressure, μ is the viscosity of the fluid, D is the spatial coordinate in the upward vertical direction, and K is the absolute permeability of the medium.

2.2 Single-phase flow

2.2.1 Single-phase flow in one dimension

The single-phase flow is the simplest case in reservoir simulation. The result is an equation for the pressure distribution and is used for the early stage in the simulation. To derive a differential equation for flow in one dimension, we assume that the cross-section area A for flow as well as the depth D, are functions of the variable x in our one dimensional space. Additionally we introduce a term for the injection q of fluid, which is equal to the mass rate of injection per unit volume of reservoir. Consider a mass balance in a small box shown in figure 2.1. The length of the box is Δx, the left face has area $A(x)$, the right face has area $A(x + \Delta x)$. So the rate at which fluid mass enters the box at the left face is given by:

$$\rho(x)v_x(x)A(x). \tag{2.8}$$

In the same way we have the rate at which fluid mass leaves at the right face:

$$\rho(x + \Delta x)v_x(x + \Delta x)A(x + \Delta x). \tag{2.9}$$

If we define the average value of A and q between x and $x + \Delta x$ as $\bar{A}$ and $\bar{q}$, we get, that the volume of the box is $\bar{A}\Delta x$, and that the rate at which fluid mass is injected into the box is:

$$\bar{q}\bar{A}\Delta x. \tag{2.10}$$

The mass contained in the box is $\bar{\phi}\bar{\rho}\bar{A}\Delta x$. So we get the rate of accumulation of mass in the box:

$$\frac{\partial(\bar{\phi}\bar{\rho})}{\partial t}\bar{A}\Delta x. \tag{2.11}$$

Because of conservation of mass, we receive:

$$[\text{rate in}] - [\text{rate out}] + [\text{rate injected}] = [\text{rate of accumulation}] \tag{2.12}$$

By using equation (2.8), (2.9), (2.10) and (2.11), we get:

$$\begin{aligned}&\rho(x)v_x(x)A(x) - \rho(x + \Delta x)v_x(x + \Delta x)A(x + \Delta x) + \bar{q}\bar{A}\Delta x \\ &\quad = \frac{\partial(\bar{\phi}\bar{\rho})}{\partial t}\bar{A}\Delta x.\end{aligned} \tag{2.13}$$

Dividing by Δx:

$$-\frac{\rho(x + \Delta x)v_x(x + \Delta x)A(x + \Delta x) - \rho(x)v_x(x)A(x)}{\Delta x} + \bar{q}\bar{A} = \frac{\partial(\bar{\phi}\bar{\rho})}{\partial t}\bar{A} \tag{2.14}$$

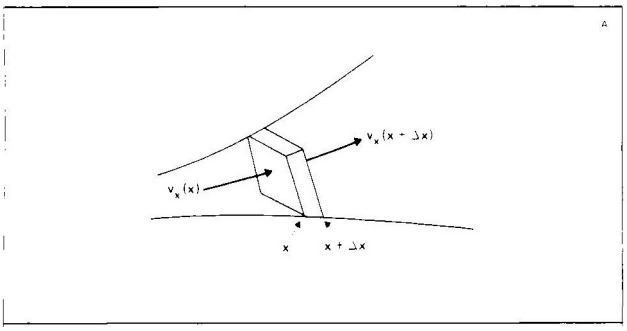


Figure 2.1: Differential elements of volume for a one-dimensional flow, [10]

Taking the limit $\Delta x \longrightarrow 0$, we get the following result:

$$-\frac{\partial(A\rho v_x)}{\partial x} + Aq = A\frac{\partial(\phi\rho)}{\partial t} \tag{2.15}$$

And this is the resulting differential equation.

2.2.2 Single-phase flow in two and three dimensions

To derive the differential equation in two and three dimensions, we use the same techniques as before. In the two dimensional case, we have to introduce the variation of the thickness of the reservoir $H = H(x, y)$.
In 2D:

$$-\frac{\partial(H\rho v_x)}{\partial x} - \frac{\partial(H\rho v_y)}{\partial y} + Hq = H\frac{\partial(\phi\rho)}{\partial t}. \tag{2.16}$$

In 3D:

$$-\frac{\partial(\rho v_x)}{\partial x} - \frac{\partial(\rho v_y)}{\partial y} - \frac{\partial(\rho v_z)}{\partial z} + q = \frac{\partial(\phi\rho)}{\partial t}. \tag{2.17}$$

By using Darcy's law from the beginning, and by defining the function α:
1D: $\alpha(x, y, z) = A(x)$,
2D: $\alpha(x, y, z) = H(x, y)$,
3D: $\alpha(x, y, z) \equiv 1$,
we get the general equation:

$$\nabla \cdot \left[\frac{\alpha\rho K}{\mu}(\nabla p + \rho g \nabla D)\right] + \alpha q = \alpha\frac{\partial(\phi\rho)}{\partial t} \tag{2.18}$$

2.2.3 Boundary conditions

To get a complete solvable system, we have to set some boundary conditions. A frequently used boundary condition is the *no-flow condition*. It means that the

reservoir lies within some closed boundary $\partial\Omega$ and there should hold $v \cdot n = 0$, where n is the normal vector pointing out of the boundary. This results in a closed flow system, where no water can enter or exit the reservoir. The problem with this boundary condition is that it is relatively difficult to obtain numerically. So it is adequate to put our reservoir in a rectangle, and set $K(x, y), \phi(x, y) = 0$ outside of the boundary.
In theory we set a well as a point source or sink, where q is zero everywhere except at this point. Since this is numerically impossible, we let Q be the desired mass rate of injection at a well and let V be the volume of a small box centered at the well. Within this box, we take:

$$q = Q/V$$

and outside we set q zero.

2.2.4 Special cases of single-phase flow

Incompressible single-phase flow

For the incompressible single-phase flow assume that the porosity ϕ of the rock is constant in time and that the fluid is incompressible, which means constant density. Then the time dependent derivative vanishes and we obtain the elliptic equation for the water pressure:

$$\nabla \cdot \left[-\frac{\alpha K}{\mu}(\nabla p - \rho G) \right] = \frac{\alpha q}{\rho}, \tag{2.19}$$

where $G = -g\nabla D$.

Ideal liquid of constant compressibility

In this model, let us assume for simplicity that $\nabla D = 0$ and $q = 0$. So equation (2.18) becomes:

$$\nabla \cdot \left[\frac{\alpha \rho K}{\mu} \nabla p \right] = \alpha \frac{\partial(\phi\rho)}{\partial t}. \tag{2.20}$$

As mentioned before, the compressibility c of a fluid is defined as:

$$c = \frac{1}{\rho}\frac{d\rho}{dp}. \tag{2.21}$$

For an ideal liquid, this means constant compressibility and constant viscosity, we get by integrating the following equation:

$$\rho = \rho_0 \exp[c(p - p_0)]. \tag{2.22}$$

From equation (2.21) follows:

$$\rho \nabla p = \frac{1}{c}\nabla \rho, \tag{2.23}$$

and so (2.20) becomes by applying (2.23):

$$\nabla \cdot \left[\frac{\alpha K}{\mu c} \nabla \rho \right] = \alpha \frac{\partial(\phi \rho)}{\partial t}. \tag{2.24}$$

If we additionally assume that the porous medium and the reservoir are homogeneous, then α, K, and ϕ are uniform, and we obtain:

$$\nabla^2 \rho = \frac{\phi \mu c}{K} \frac{\partial \rho}{\partial t}. \tag{2.25}$$

2.3 Two-phase flow

2.3.1 General solution

Because general in reservoir simulation, there are involved at least two different phases, the single-phase flow seldom occurs. So we would like to simulate the displacement of oil by water or gas. The difficulty is, that this happens in a simultaneous flow and not with a sharp edge. To make it not too difficult, we start by assuming no mass transfer between the two fluids. In either case, there is one wetting phase, which means that one fluid wets the porous medium more than the other. In a water-oil system, the water is the wetting phase, and in an oil-gas system, the oil would be the wetting phase. We refer to the wetting phase by the subscript w and to the non-wetting phase by the subscript n. As mentioned before, we have:

$$s_n + s_w = 1 \tag{2.26}$$

As an empirical fact, we accept that the capillary pressure is a function of the saturation of the wetting phase.

$$p_{cnw}(s_w) = p_n - p_w \tag{2.27}$$

Darcy's law can be extended to multiphase flow by assuming that the phase pressure forces each fluid to flow. So equation (2.7) can be written as:

$$v_n = -\frac{K k_{rn}}{\mu_n}(\nabla p_n - \rho_n G), \tag{2.28}$$

$$v_w = -\frac{K k_{rw}}{\mu_w}(\nabla p_w - \rho_w G). \tag{2.29}$$

To obtain our differential equations, we apply, in the same way as in the single-phase flow, the conservation of mass to each phase. Except of the rate of accumulation, we get the same equations (2.8), (2.9) and (2.10) for each phase. To receive the rate of accumulation, we have to multiply the volume by the saturation and get:

$$-\nabla \cdot (\alpha \rho_n v_n) + \alpha q_n = \alpha \frac{\partial(\phi \rho_n s_n)}{\partial t}, \tag{2.30}$$

$$-\nabla \cdot (\alpha \rho_w v_w) + \alpha q_w = \alpha \frac{\partial(\phi \rho_w s_w)}{\partial t}. \tag{2.31}$$

After applying Darcy's law, we end up with:

$$\nabla \cdot \left[\frac{\alpha \rho_n K k_{rn}}{\mu_n} (\nabla p_n - \rho_n G) \right] + \alpha q_n = \alpha \frac{\partial(\phi \rho_n s_n)}{\partial t}, \tag{2.32}$$

$$\nabla \cdot \left[\frac{\alpha \rho_w K k_{rw}}{\mu_w} (\nabla p_w - \rho_w G) \right] + \alpha q_w = \alpha \frac{\partial(\phi \rho_w s_w)}{\partial t}. \tag{2.33}$$

In the following we will rewrite this equations in a more practical way consisting of a pressure equation and a saturation equation.

2.3.2 Pressure equation

First we apply the Leibniz rule to the time derivative of equation (2.30) and (2.31):

$$-\nabla \cdot (\alpha \rho_n v_n) + \alpha q_n = \alpha \left[\rho_n s_n \frac{\partial \phi}{\partial t} + \phi s_n \frac{d\rho_n}{dp_n} \frac{\partial p_n}{\partial t} + \phi \rho_n \frac{\partial s_n}{\partial t} \right] \tag{2.34}$$

$$-\nabla \cdot (\alpha \rho_w v_w) + \alpha q_w = \alpha \left[\rho_w s_w \frac{\partial \phi}{\partial t} + \phi s_w \frac{d\rho_w}{dp_w} \frac{\partial p_w}{\partial t} + \phi \rho_w \frac{\partial s_w}{\partial t} \right] \tag{2.35}$$

Now, we divide (2.34) by $\alpha \rho_n$ and (2.35) by $\alpha \rho_w$, add the resulting equations and use (2.26):

$$\begin{aligned} &-\frac{1}{\alpha \rho_n} \nabla \cdot (\alpha \rho_n v_n) - \frac{1}{\alpha \rho_w} \nabla \cdot (\alpha \rho_w v_w) + Q_t \\ &\quad = \frac{\partial \phi}{\partial t} + \phi s_n c_n \frac{\partial p_n}{\partial t} + \phi s_w c_w \frac{\partial p_w}{\partial t}, \end{aligned} \tag{2.36}$$

where

$$Q_t = \frac{q_n}{\rho_n} + \frac{q_w}{\rho w}$$

is the total volumetric injection rate, and c_n, c_w are the phase compressibilities, defined as in (2.6). By using the rock compressibility (2.1), equation (2.23), equation (2.26) and defining the phase mobility λ_i of phase i by:

$$\lambda_i = \frac{k_{ri}}{\mu_i}, \tag{2.37}$$

we get:

$$\begin{aligned} &-\frac{1}{\alpha} \nabla \cdot [\alpha K \lambda_w (\nabla p_w - \rho_w G) + \alpha K \lambda_n (\nabla p_n - \rho_n G)] + c_r \phi \frac{\partial p}{\partial t} \\ &\quad -c_w \left[\nabla p_w \cdot K \lambda_w (\nabla p_w - \rho_w G) - \phi s_w \frac{\partial p_w}{\partial t} \right] \\ &\quad -c_n \left[\nabla p_n \cdot K \lambda_n (\nabla p_n - \rho_n G) - \phi s_n \frac{\partial p_n}{\partial t} \right] = Q_t \end{aligned} \tag{2.38}$$

In the resulting equation we have three different pressures: the total pressure p, the pressure of the wetting phase p_w and the pressure of the non-wetting phase p_n. By assuming that the capillary pressure $p_{cnw} = p_n - p_w$ is known and by setting the non-wetting pressure p_n as the primary variable, we get a parabolic equation that can be solved for the non-wetting-phase pressure p_n.

2.3.3 Pressure equation for incompressible immiscible flow

Since solving the parabolic equation (2.38) is not an easy task and since the temporal derivative terms are quite small, equation (2.38) is nearly elliptic. By assuming that the two phases are incompressible, i.e. $c_r = c_w = c_n = 0$, we get:

$$-\frac{1}{\alpha}\nabla \cdot [\alpha K\lambda_w(\nabla p_w - \rho_w G) + \alpha K\lambda_n(\nabla p_n - \rho_n G)] = Q_t. \tag{2.39}$$

We still have two unknown phase pressures, p_w and p_n, in our equation. As mentioned before, there is a dependency between them, due to the capillary pressure $p_{cnw} = p_n - p_w$, which is assumed to be a function of s_w. The next steps follow an approach which introduces the *global*, the *reduced*, or the *intermediate pressure* $p = p_n - p_c$ (for more details, see [2]). The new variable p_c is called the *saturation-dependent complementary pressure* and is defined by

$$p_c(s_w) = \int_1^{s_w} f_w(\xi)\frac{\partial p_{cnw}}{\partial s_w}(\xi)d\xi, \tag{2.40}$$

where the *fractional-flow function* $f_w = \lambda_w/(\lambda_w + \lambda_n)$ measures the water fraction of the total flow. From (2.40) we get

$$\nabla p_c = f_w \nabla p_{cnw}. \tag{2.41}$$

Next, we express the total velocity $v = v_n + v_w$ as function of the global pressure p:

$$v = -K(\lambda_w + \lambda n)\nabla p - K(\lambda_w\rho_w + \lambda_n\rho_n)G. \tag{2.42}$$

Applying this to equation (2.39), we get

$$\frac{1}{\alpha}\nabla \cdot (\alpha v) = Q_t, \tag{2.43}$$

where $\lambda = \lambda_w + \lambda_n$ is the total mobility. As boundary condition, we use normally the no-flow condition, but in some cases, there can be used some variation.

2.3.4 Saturation equation

In the previous part, we derived the pressure equation, which can be solved for the global pressure p. But to derive a complete model, we should as well derive an equation for each saturation. Since $s_w + s_n = 1$, it suffices to calculate only one saturation. In practice, it is common to calculate s_w. Therefore, we need

to find an expression for the velocity v_w in terms of p and p_c.
From Darcy's law, we get

$$\lambda_n v_w - \lambda_w v_n = K\lambda_n\lambda_w \nabla p_{cnw} - K\lambda_n\lambda_w(\rho_n - \rho_w)G. \tag{2.44}$$

Inserting $v_n = v - v_w$ and dividing by λ, we get

$$v_w = f_w[v + K\lambda_n \nabla p_{cnw} + K\lambda_n(\rho_w - \rho_n)G]. \tag{2.45}$$

Finally, we include $\nabla p_{cnw} = \frac{\partial p_{cnw}}{\partial s_w}\nabla s_w$ in equation (2.45). Applying the result to equation (2.31), and we get:

$$\begin{aligned}\alpha\frac{\partial(\phi\rho_w s_w)}{\partial t} &= \nabla\cdot(\alpha\rho_w h_w \nabla s_w) \\ &\quad -\nabla\cdot(\alpha\rho_w(f_w[v + K\lambda_n(\rho_w - \rho_n)G])) + \alpha q_w,\end{aligned} \tag{2.46}$$

where $h_w = -f_w K\lambda_n \frac{\partial p_{cnw}}{\partial s_w}$.

2.3.5 Saturation equation for incompressible flow

To get a simple equation compared to (2.46), we assume incompressibility of the fluid and constant porosity, which includes that ϕ, ρ_n, and ρ_w are constant. So (2.46) becomes:

$$\alpha\phi\frac{\partial s_w}{\partial t} + \nabla\cdot(\alpha f_w v) - \nabla\cdot(\alpha h_w\nabla s_w) + \nabla\cdot(\alpha K f_w\lambda_n(\rho_w - \rho_n)G) = \alpha\frac{q_w}{\rho_w} \tag{2.47}$$

To complete the model, we have to choose some initial value and again the boundary conditions.We will impose the no-flow condition as boundary condition and take the initial value $s_w(x,0) = s_w^0(x)$.
In general, the saturation equation is a parabolic equation. However, on a reservoir scale, the terms $f_w(s)v$ and $f_w(s)K\lambda_n(\rho_w - \rho_n)G$ usually dominate the term $-\nabla\cdot(\alpha h_w\nabla s_w)$. Hence, it has a strong hyperbolic nature and should be discretized in a different way than the pressure equation.

2.4 Three-phase flow

Let us consider the case where three immiscible fluids are involved in our system. In general these different fluids will be gas, oil and water (g, o, w). We again assume no mass transfer between the three fluids. Deriving the differential equations for three phases works in the same way as for two phases. First we have

$$s_g + s_o + s_w = 1. \tag{2.48}$$

But now we have three capillary pressures, whereof two are independent:

$$p_{cow} = p_o - p_w, \tag{2.49}$$

$$p_{cgo} = p_g - p_o, \tag{2.50}$$

$$p_{cgw} = p_g - p_w = p_{cgo} + p_{cow}. \tag{2.51}$$

In the two-phase model, it is quite easy to obtain some experimental data of the capillary pressures, but in the three-phase case there is only little experimental data. So it is necessary to get some estimates from the two-phase case.
One still important tool is Darcy's law, which is in three phases the same as before. To derive the differential equations, we use again conservation of each phase, and get:

$$-\nabla \cdot (\alpha \rho_g v_g) + \alpha q_g = \alpha \frac{\partial(\phi \rho_g s_g)}{\partial t}, \tag{2.52}$$

$$-\nabla \cdot (\alpha \rho_w v_w) + \alpha q_w = \alpha \frac{\partial(\phi \rho_w s_w)}{\partial t}, \tag{2.53}$$

$$-\nabla \cdot (\alpha \rho_o v_o) + \alpha q_o = \alpha \frac{\partial(\phi \rho_o s_o)}{\partial t}. \tag{2.54}$$

Next, we apply Darcy's law to (2.52), (2.53) and (2.54), and end up with:

$$\nabla \cdot \left[\frac{\alpha \rho_g K k_{rg}}{\mu_g} (\nabla p_g - \rho_g G) \right] + \alpha q_g = \alpha \frac{\partial(\phi \rho_g s_g)}{\partial t}, \tag{2.55}$$

$$\nabla \cdot \left[\frac{\alpha \rho_w K k_{rw}}{\mu_w} (\nabla p_w - \rho_w G) \right] + \alpha q_w = \alpha \frac{\partial(\phi \rho_w s_w)}{\partial t}, \tag{2.56}$$

$$\nabla \cdot \left[\frac{\alpha \rho_o K k_{ro}}{\mu_o} (\nabla p_o - \rho_o G) \right] + \alpha q_o = \alpha \frac{\partial(\phi \rho_o s_o)}{\partial t}. \tag{2.57}$$

Parallel to the pressure equation and saturation equation in the two-phase flow, we can derive an alternative system, which equivalent to the three equations (2.55), (2.56) and (2.57). The new system will consist of one elliptic or near-elliptic pressure equation and two near-hyperbolic saturation equations, where the saturation equation will depend on one saturation and the total velocity $v = v_g + v_o + v_w$.

2.5 Multiphase and multicomponent flows

After looking at the single-phase, two-phase and three-phase flow, we will deal with the general case. Let us now consider N components (or chemical species), whereof each may exist in any or all of the three phases (gas, oil, water). Note that certain components can be in the oil phase and in the gas phase. Additionally, it may be, that some hydrocarbon component can be dissolved in the water phase, too. As defined in the beginning, let c_{ij} be the mass fraction of the ith component in the jth phase.
The main difference to the previous sections is that we can not assume conservation of mass in each phase. This is because it is now possible to transfer various components between the phases. But we still have a certain conservation of

mass, this holds now in each component. As long as the mass flux density of each phase j is $\rho_j v_j$, we have that the mass flux density of the ith component is

$$c_{ig}\rho_g v_g + c_{io}\rho_o v_o + c_{iw}\rho_w v_w. \tag{2.58}$$

And the mass of component i per unit bulk volume of porous medium is

$$\phi(c_{ig}\rho_g s_g + c_{io}\rho_o s_o + c_{iw}\rho_w s_w). \tag{2.59}$$

Due to the conservation of mass for each component, we thus get:

$$\begin{aligned} &-\nabla \cdot [\alpha(c_{ig}\rho_g v_g + c_{io}\rho_o v_o + c_{iw}\rho_w v_w)] + \alpha q_i \\ &= \quad \alpha \frac{\partial}{\partial t}[\phi(c_{ig}\rho_g s_g + c_{io}\rho_o s_o + c_{iw}\rho_w s_w)]. \end{aligned} \tag{2.60}$$

The last missing part is again Darcy's law, which holds in the same way as before:

$$v_i = -K\frac{k_{ri}}{\mu_i}(\nabla p_i - \rho_i G). \tag{2.61}$$

2.5.1 Black-oil model

Solving the general system of the previous section is a really difficult task. Therefore, we look again at a more reduced model. As long as we have a low-volatility oil system, consisting mainly of methane and heavy components, we can use the simplified "Black-oil" model. In this model it is assumed that no mass transfer occurs between the water phase and the other two phases. This means that the hydrocarbon fluid composition remains constant for all times.

Example 2.1 (Two-phase flow of water and hydrocarbon)**.** *An example for the Black-oil model in two phases is that we have one water phase w and one hydrocarbon phase, where the hydrocarbon consists of two components, dissolved gas and a residual (or black) oil. So we get the following mass fractions:*

$$\begin{array}{lll} c_{ww} = 1, & c_{ow} = 0, & c_{gw} = 0, \\ c_{wo} = 0, & c_{oo} = \frac{m_o}{m_o+m_g}, & c_{go} = \frac{m_g}{m_o+m_g}, \\ c_{wg} = 0, & c_{og} = 0, & c_{gg} = 0. \end{array}$$

where m_o and m_g are the masses of oil and gas.

Chapter 3

Elliptic partial differential equations

As we have seen in chapter 2, in order to solve a reservoir problem, we encounter an elliptic or near-elliptic partial differential equation. This chapter presents a short introduction to elliptic partial differential equations. Note, that we will just present some basic results and numerical methods. For more details, for example about convergence or existence, we refer to the books [1] and [4], upon which this chapter is based.

3.1 Setting and definitions

Let $u : \mathbb{R}^n \longrightarrow \mathbb{R}, n \in \mathbb{N}$.For simplicity, we introduce the following abbreviations: Let $\alpha \in \mathbb{N}^n, k \in \mathbb{N}$:

$$|\alpha| := \sum_{i=1}^{n} \alpha_i,$$

$$Du := \nabla u,$$

$$D^2 u := \left(\frac{\partial^2 u}{\partial x_i \partial x_j} \right)_{i,j=1}^{n},$$

$$D^\alpha u := \frac{\partial^{|\alpha|} u}{\partial_{x_1}^{\alpha_1} \dots \partial_{x_n}^{\alpha_n}},$$

$$D^k u := \{D^\alpha u; |\alpha| = k\},$$

$$u_{x_i} := \frac{\partial u}{\partial x_i}.$$

Let $U \subset \mathbb{R}^n$ be an open bounded set, $a^{ij} \in C^1(\bar{U})$.
A partial differential equation (pde) is called *elliptic* if it has following form:

$$\begin{cases} Lu = f, & \text{if } x \in U \\ \text{boundary conditions}, & \text{if } x \in \partial U \end{cases}, \tag{3.1}$$

where

$$Lu = -\sum_{i,j=1}^{n} (a^{ij}(x)u_{x_i})_{x_j} + \sum_{i=1}^{n} b^i(x)u_{x_i} + c(x)u, \tag{3.2}$$

and such that there exists $\theta > 0$ such that:

$$\sum_{i,j=1}^{n} \xi_i a^{i,j}(x)\xi_j \geq \theta|\xi|^2, \forall x, \xi \in \mathbb{R}^n. \tag{3.3}$$

One typical example for elliptic pde's is

$$\begin{cases} Lu = f, & \text{if } x \in U \\ u = 0, & \text{if } x \in \partial U \end{cases}, \tag{3.4}$$

where $Lu = \Delta u$.

3.1.1 Weak derivative:

$v = u_{x_i}$ holds weakly in $U \subset \mathbb{R}^n$, where U is open, if

$$\int_U v\varphi dx = (-1)\int_{\partial U} u\varphi_{x_i} dx, \quad \forall \varphi \in C_0^\infty(U). \tag{3.5}$$

Furthermore, we say $v = D^\alpha u$, $\alpha \in \mathbb{R}^n$ holds weakly in $U \subset \mathbb{R}^n$, where U is open, if

$$\int_U v\varphi dx = (-1)^{|\alpha|}\int_{\partial U} uD^\alpha \varphi dx, \quad \forall \varphi \in C_0^\infty(U), \tag{3.6}$$

3.1.2 Sobolev spaces:

Let $k, p \in \mathbb{N}$ and U be a set in $\mathbb{R}^n$. Then we define the Sobolev space as:

$$W^{k,p}(U) := \{u \in L^p(U); \quad D^\alpha u \in L^p(U), \quad |\alpha| \leq k, \quad \alpha \in \mathbb{R}^n\}. \tag{3.7}$$

For $p = 2$ we write:

$$H^k(U) := W^{k,2}(U). \tag{3.8}$$

By defining the corresponding *Sobolev norm*

$$||u||_{W^{k,p}(U)} := \Big(\sum_{|\alpha|\leq k} ||D^\alpha u||^p_{L^p(U)} \Big)^{1/p}, \quad 1 \leq p < \infty, \tag{3.9}$$

$$||u||_{W^{k,\infty}(U)} := \sum_{|\alpha|\leq k} ||D^\alpha u||_{L^\infty(U)}, \tag{3.10}$$

the Sobolev space gets a Banach-space.
The resulting problem can be formulated as follows: Find $u \in H^1_0(U)$ such that (3.1) holds weakly, which means:

$$a(u,v) = \int_U fv dx, \forall v \in H^1_0(U), \tag{3.11}$$

where

$$a(u,v) = \int_U \sum_{i,j=1}^n a^{ij}(x) u_{x_i} v_{x_j} + \sum_{i=1}^n b^i(x) u_{x_i} v + c(x)uv dx. \tag{3.12}$$

To generalize the problem, let H be an Hilbert space, $f \in H^*$, $a : H \times H \longrightarrow \mathbb{R}$ be bilinear, continuous ($|a(u,v)| \leq C||u||||v||, C \in \mathbb{R}$) and coercive ($a(u,u) \geq \beta||u||^2, \beta \in \mathbb{R}$).
Problem: Find $u \in H$ such that:

$$a(u,v) = l(v), \quad \forall v \in H, \tag{3.13}$$

where $l(v) =< f, v >$.

3.2 Discretization of the elliptic problem/Galerkin approximation

Let $V_N \subset V$ be a finite dimensional subset of V, where $\dim V_N = N$. Thus, we get a new discrete problem:
Find $u_N \in V_N$ such that:

$$a(u_N, v) = l(v), \forall v \in V_N. \tag{3.14}$$

For the theory concerning *existence and convergence* of this solutions, we refer to [1].
Assume that U is a polygonal domain, and τ be a triangulation of U, such that two triangles overlap only at a point or along a line. Define N to be the number of nodes of the triangulation τ. Let us now choose a basis of V_N: $\{b_1(x), \ldots, b_N(x)\}$, such that

$$b_i(p_j) = \delta_{ij}, \text{ where } p_j \text{ is the } j\text{th node.} \tag{3.15}$$

Then we can write u_N in terms of basis elements:

$$u_N(x) = \sum_{i=1}^{N} u_i b_i(x) = \underline{u}_N^T \underline{b}(x), \tag{3.16}$$

where $\underline{u}_N := (u_1, \ldots, u_N)^T$ and $\underline{b}(x) := (b_1(x), \ldots, b_N(x))^T$. By applying equation (3.16) to (3.14), we get:

$$\sum_{i=1}^{N} u_i a(b_i, b_j) = l(b_j), \forall j = 1, \ldots, N. \tag{3.17}$$

To rewrite this as a matrix, define:

$$\underline{l} := (l(b_1), \ldots, l(b_N))^T. \tag{3.18}$$

So we get:

$$a(\underline{b}, \underline{b}^T)\underline{u}_N = \underline{l} \Rightarrow A\underline{u}_N = \underline{l}, \tag{3.19}$$

where

$$(A)_{ij} := a(b_i, b_j). \tag{3.20}$$

Element stiffness matrix for triangulation:
Let $\hat{K} = \{\xi; \xi_i \geq 0, \xi_1 + \xi_2 \leq 1\}$ be the reference element and let

$$F_K(\xi) := p_0 + \xi_1(p_1 - p_0) + \xi_2(p_2 - p_0) = p_0 + B_K\xi. \tag{3.21}$$

Define

$$N_0(\xi) = 1 - \xi_1 - \xi_2, \tag{3.22}$$

$$N_1(\xi) = xi_1, \tag{3.23}$$

$$N_2(\xi) = \xi_2, \tag{3.24}$$

as the basis on the reference element. The non-zero basis functions on the element K are:

$$b_{p_j}(x) = N_j(F_K^{-1}(x)), j = 0, 1, 2. \tag{3.25}$$

Then we have:

$$A = \sum_K T_K^T a_K T_K, \tag{3.26}$$

where

$$\begin{aligned} a_K = a(b_{p_i}, b_{p_j}) &= \int_K \nabla b_{p_i} \nabla b_{p_j} \\ &= \int_{\hat{K}} (B_K^T)^{-1} \nabla N_i(\xi) (B_K^T)^{-1} \nabla N_j(\xi) |det(B_K)| d\xi, \end{aligned} \tag{3.27}$$

and

$$(T_K)_{ij} = \begin{cases} 1, & \text{if j is the global number of } p_i \text{ in K} \\ 0, & \text{otherwise} \end{cases}. \tag{3.28}$$

This method is called the *Galerkin method*.

Chapter 4

Hyperbolic partial differential equations

The following chapter is a summary of the theory about hyperbolic equations. For details, we refer to the books of D. Kröner [6] and R. J. LeVeque [8]. The interesting and really difficult fact of hyperbolic partial differential equations is, that the solution does not have to be smooth, even if the initial value is smooth. In one space dimension we look at partial differential equation of the form:

$$u_t + f(u)_x = 0, \tag{4.1}$$

where we define $u : \mathbb{R} \times \mathbb{R}^+ \longrightarrow \mathbb{R}^m$ and the *flux function* $f : \mathbb{R}^m \longrightarrow \mathbb{R}^m$. The equation is called *hyperbolic*, if each eigenvalue of the Jacobian Df(u) is real, for all values.
In general, we look at equations of the form:

$$u_t + f_1(u)_{x_1} + \ldots + f_n(u)_{x_n} = 0, \tag{4.2}$$

where we define $u : \mathbb{R}^n \times \mathbb{R}^+ \longrightarrow \mathbb{R}^m$ and the *flux functions* $f_i : \mathbb{R}^m \longrightarrow \mathbb{R}^m, \forall i = 1, \ldots, n$. Equation (4.2) is called *hyperbolic*, if any real linear combination $\alpha_1 Df_1(u) + \ldots + \alpha_n Df_n(u)$ of the Jacobians has real eigenvalues. There is not much known about the behavior of the solution in higher space dimensions. In the further part, we will concentrate at most on two space dimensions. If you look at u as a density (in one space dimension) in the state variable, and $\int_{x_1}^{x_2} u(x,t)dx$ is the total amount of this variable in the interval $[x_1, x_2]$ at time t. We see that these state variables are conserved in time, this means:

$$\int_{x_1}^{x_2} u(x,t)dx = \int_{x_1}^{x_2} u(x,t_0)dx + \int_{t_0}^{t} f(u(x_1,t))dt - \int_{t_0}^{t} f(u(x_2,t))dxdt. \tag{4.3}$$

If u is a density of a fluid, we can interpret this equation such that the total amount of fluid at time t is equal to the total amount of fluid at time t_0 plus the value, which is flowing in between t_0 and t at x_1 and minus the value, which is

flowing out at x_2. Therefore, equations (4.1) and (4.2) are called *conservation law*.
As a consequence, that we can only prove local existence of the solution of equation (4.2) , we have to generalize the definition of solutions of conservation laws. We multiply equation (4.1) by $\phi \in C_0^1(\mathbb{R} \times \mathbb{R}^+)$ and after integrating over time and space we get:

$$\int_0^\infty \int_{-\infty}^\infty \phi u_t + \phi f(u)_x dx dt = 0. \tag{4.4}$$

Next, we integrate by parts and obtain:

$$\int_0^\infty \int_{-\infty}^\infty \phi_t u + \phi_x f(u) dx dt = -\int_\infty^\infty \phi(x,0)u(x,0)dx. \tag{4.5}$$

The function $u(x,t)$ is called a *weak solution* of the conservation law if (4.5) holds for all functions $\phi \in C_0^1(\mathbb{R} \times \mathbb{R}^+)$. Unfortunately the weak solution does not have to be unique. To restrict this, we look for the physically relevant weak solution, the *entropy solution*. We will do this by making the vanishing viscosity approach: Adding a small viscous term $\epsilon(u^\epsilon)_{xx}$ to equation (4.1), we receive a parabolic equation, which has a unique solution:

$$(u^\epsilon)_t + f(u^\epsilon)_x = \epsilon(u^\epsilon)_{xx}. \tag{4.6}$$

So we define the unique viscosity solution as the limit of $u^\epsilon, \epsilon \longrightarrow 0$. With this limit, you can derive *Kruzkov's entropy condition*:
A weak solution of

$$\begin{cases} u_t + f_1(u)_x + f_2(u)_y = 0, & \text{in } \mathbb{R}^2 \times \mathbb{R}^+ \\ u(\cdot, 0) = u_0, & \text{in}\mathbb{R}^2 \end{cases} \tag{4.7}$$

is called an Kruzkov entropy solution, if we have for all $\phi \in C_0^\infty(\mathbb{R}^2) \times \mathbb{R}^+), \phi \leq 0$ and for all $k \in \mathbb{R}$

$$\begin{aligned} &\int_{\mathbb{R}^2} \int_{\mathbb{R}^+} [\phi_t |u-k| + \phi_x sign(u-k)(f_1(u) - f_1(k)) \\ &\quad + \quad \phi_y sign(u-k)(f_2(u) - f_2(k))] dt dx dy \geq 0. \end{aligned} \tag{4.8}$$

Kruzkov [7] has proved that every entropy solution can be considered as a viscosity limit.

4.1 Riemann problem

One important problem is an arbitrary equation with a piecewise constant initial value, which has one single jump discontinuity. This is known as the *Riemann problem*. Solving this, leads us to three different types of waves: *rarefaction waves*, *shock waves* and *contact discontinuities*.

4.2 Numerical methods for hyperbolic equations in 1-D

4.2.1 Finite difference schemes

Linear equations

First we will look at numerical methods for linear equations of the form

$$\begin{cases} u_t + Au_x = 0, & x \in \mathbb{R}, t \geq 0, A \in \mathbb{R}^{m\times m}, \\ u(x,0) = u_0(x). \end{cases} \tag{4.9}$$

We have to define a discrete map on the x-t plane by the points (x_j, t_n)

$$x_j = jh, j \in \mathbb{Z},$$

$$t_n = nk, n \in \mathbb{N},$$

$$x_{j+1/2} = x_j + h/2 = (j + 1/2)h,$$

where $h = \Delta x$, $k = \Delta t$. For simplicity, we assume a uniform mesh, with constant h and k.
To get a numerical method, we approximate the value of u on the mesh. Hence, it is common to use the cell average of the cell $[x_{j-1/2}, x_{j+1/2}]$ at time t_n for the value of $u_j^n \approx u(x_j, t_n)$. To receive a method, we choose some finite difference, e.g. backward Euler, and rewrite the differential equation by using the finite difference instead of the real derivative.

Example 4.1 (Backward Euler)**.**

$$u_t(x,t) \approx \frac{u(x,t+k) - u(x,t)}{k},$$

$$u_x(x,t) \approx \frac{u(x+h) - u(x-h)}{2h},$$

so instead of equation (4.9) *we get*

$$\frac{u_j^{n+1} - u_j^n}{k} + A\frac{u_{j+1}^n - u_{j-1}^n}{2h} = 0 \tag{4.10}$$

After solving equation (4.10) *for* u_j^{n+1}*, we obtain the Backward Euler formula:*

$$u_j^{n+1} = u_j^n - \frac{k}{2h}A(U_{j+1}^n - u_{j-1}^n). \tag{4.11}$$

But we will find out that in practice this method is useless, because of stability problems.

An other approach to derive a method is by looking at the Taylor expansion, where we replace the exact derivatives again by some approximations. For example, the Lax-Wendroff and Beam-Warming method are based on this approach.

Example 4.2 (Lax-Wendroff scheme)**.**

$$u_j^{n+1} = u_j^n - \frac{k}{2h}A(u_{j+1}^n - u_{j-1}^n) + \frac{k^2}{2h^2}A^2(u_{j+1}^n - 2u_j^n + u_{j-1}^n). \tag{4.12}$$

Example 4.3 (Beam-Warming scheme)**.**

$$u_j^{n+1} = u_j^n - \frac{k}{2h}A(3u_j^n - 4u_{j-1}^n + u_{j-2}^n) + \frac{k^2}{2h^2}A^2(u_j^n - 2u_{j-1}^n + u_{j-2}^n). \tag{4.13}$$

Intuitionally it would be better if the numerical scheme takes the information from the same direction as the direction in which the original solution flows. This means that it follows the direction of the characteristics. This sort of schemes are called *upwind methods*. The theoretical background can be looked up in the book of R. J. LeVeque [8]. The scheme is build up such that, we do some sort of decomposition into characteristics. We can decouple the system by making a change of the basis and set $v(x,t) = R^{-1}u(x,t)$, where R is the matrix of eigenvectors of A. So we get the equation:

$$v_t + \Lambda v_x = 0, \tag{4.14}$$

where Λ is the diagonal matrix of the eigenvalues. Let us define

$$\lambda_p^+ = max(\lambda_p, 0), \qquad \Lambda^+ = diag(\lambda_1^+, \ldots, \lambda_m^+), \tag{4.15}$$

$$\lambda_p^- = min(\lambda_p, 0), \qquad \Lambda^- = diag(\lambda_1^-, \ldots, \lambda_m^-), \tag{4.16}$$

So the upwind method for (4.14) is

$$v_j^{n+1} = v_j^n - \frac{k}{h}\Lambda^+(v_j^n - v_{j-1}^n) - \frac{k}{h}\Lambda^-(v_{j+1}^n - v_j^n). \tag{4.17}$$

By multiplying R, the scheme can be transformed back to the original u.

$$u_j^{n+1} = u_j^n - \frac{k}{h}A^+(u_j^n - u_{j-1}^n) - \frac{k}{h}A^-(u_{j+1}^n - u_j^n), \tag{4.18}$$

where $A^+ = R\Lambda^+R^{-1}$, $A^- = R\Lambda^-R^{-1}$.

Nonlinear equations

Let us now take a look at numerical methods to nonlinear problems

$$\begin{cases} u_t + f(u)_x = 0, \quad x \in \mathbb{R}, t \geq 0, \\ u(x,0) = u_0(x). \end{cases} \tag{4.19}$$

Here we might have the problem that our method converges to a function that is not a weak solution of our equation, or that is the wrong weak solution (not Entropy solution). It turns out to be quite simple to guarantee that we converge at least not to a non-solution. The requirement is that the method has to be in *conservation form*, which means that:

$$u_j^{n+1} = u_j^n - \frac{k}{h}[F(u_{j-p}^n, u_{j-p+1}^n, \ldots, u_{j+q}^n) - F(u_{j-p-1}^n, u_{j-p}^n, \ldots, u_{j+q-1}^n)], \tag{4.20}$$

for some *numerical flux function* F of $p+q+1$ arguments. If we take the simplest case, $p = 0$ and $q = 1$, we can see that this is really natural, by using the cell average and integration by parts.

Let us assume a nonlinear system $u_t + f(u)_x = 0$, for which holds $Df(u_j^n)$ has only nonnegative eigenvalues for all u_j^n. Then, we can define a generalized upwind method as

$$F(u,v) = f(u). \tag{4.21}$$

It would be exactly the opposite, if $Df(u_j^n)$ has only nonpositive eigenvalues for all u_j^n.
But still we have not received a method converging to an entropy satisfying solution. So we take look at *Godunov's method*, invented in 1959 by Godunov. The idea is to define a function $\tilde{u}^n(x,t^n)$ with the value u_j^n on grid cell $(x_{j-1/2}, x_{j+1/2})$. We will use $\tilde{u}^n(x,t^n)$ as initial data of the hyperbolic equation. Because these are multiple Riemann problems, which we can solve exactly to obtain $\tilde{u}^n(x,t), t \in [t_n, t_{n+1}]$. After calculating the solution between t_n and t_{n+1}, we define the approximate solution by averaging the exact solution over $[x_{j-1/2}, x_{j+1/2}]$. Now, repeat this progress.
Using equation (4.19) at the average we get

$$\begin{aligned} u_j^{n+1} &= \frac{1}{h}\int_{x_{j-1/2}}^{x_{j+1/2}} \tilde{u}^n(x,t_{n+1})dx = \frac{1}{h}\int_{x_{j-1/2}}^{x_{j+1/2}} \tilde{u}^n(x,t_n)dx \\ &+ \frac{1}{k}\int_{t_n}^{t_{n+1}} f(\tilde{u}^n(x_{j-1/2},t))dt - \frac{1}{k}\int_{t_n}^{t_{n+1}} f(\tilde{u}^n(x_{j+1/2},t))dt \end{aligned} \tag{4.22}$$

So we get:

$$u_j^{n+1} = u_j^n - \frac{k}{h}[F(u_j^n, u_{j+1}^n) - F(u_{j-1}^n, u_j^n)], \tag{4.23}$$

where the numerical flux function F is

$$F(u_j^n, u_{j+1}^n) = \frac{1}{k}\int_{t_n}^{t_{n+1}} f(\tilde{u}^n(x_{j+1/2}, t))dt. \tag{4.24}$$

Theoretical the application of the Godunov method is no problem, because it is possible to solve the Riemann problem at each cell. But in practice, it would be too expensive. And notice that the most information of the exact solution gets lost by averaging over the cells. So there are some different ways to solve it approximately. One possibility is *Roe's approximate Riemann solver*. It is based on solving a constant coefficient linear system of conservation laws instead of the original nonlinear system. The exact idea can be read in [8].

4.3 Numerical methods for hyperbolic equations in 2-D

Up to now, we have only considered the one dimensional problem, but today the relevant cases are in general in two or three dimensions. Let us take a look especially at two dimensional scalar problems. So the conservation law has the form

$$\begin{cases} u_t + f_1(u)_x + f_2(u)_y = 0, & \text{in}\mathbb{R}^2 \times \mathbb{R}^+, \\ u(x, y, 0) = u_0(x, y), & \text{in}\mathbb{R}^2, \end{cases} \tag{4.25}$$

where $u : \mathbb{R}^2 \times \mathbb{R}^+ \longrightarrow \mathbb{R}$, $f_1, f_2 : \mathbb{R} \longrightarrow \mathbb{R}$ and $u_0 \in L^\infty(\mathbb{R}^2)$.

4.3.1 Splitting schemes

The dimensional splitting is an intuitive method, where we just use one of the previous 1D schemes in each direction. We use the solution of the first direction as the initial value of the second one. Crandall and Majda proved in [3], that for scalar conservation laws the following holds:
Let $\mathcal{H}_k^x$, $\mathcal{H}_k^y$ be a one dimensional monotone method with step size k. Then the methods

$$\tilde{u}^n = (\mathcal{H}_k^y \mathcal{H}_k^x)^n u_0, \tag{4.26}$$

$$\hat{u}^n = (\mathcal{H}_{k/2}^x \mathcal{H}_k^y \mathcal{H}_{k/2}^x)^n u_0 \tag{4.27}$$

will converge to the solution of the 2D problem.

4.3.2 Finite volume methods

In this section, we will take a look at a new discretization method. We will consider an unstructured grid in two dimensions. Let us introduce first some

definitions and notations.
Let a *k-polygon* be a closed polygon with k vertices. Then we define the set

$$T := \{T_i; T_i \text{ is a k-polygon for } i \in I \subset \mathbb{N}\}, \tag{4.28}$$

which is called an *unstructured grid* of $U \subset \mathbb{R}^2$ if the following properties are satisfied:

(i) $U = \cup_{i\in I} T_i$.

(ii) For two different T_i and T_j we have $T_i \cap T_j = \emptyset$ or $T_i \cap T_j =$ a common vertex of T_i and T_j or $T_i \cap T_j =$ a common edge of T_i and T_j.

Let $(T_j)_{j\in I}$ denote an unstructured grid of $\mathbb{R}^2$. Then we have:

- T_j: the jth cell of the grid;
- $|T_j|$: area of Tj;
- $T_{jl}, l = 1, \ldots, k$: neighboring cells of T_j;
- u_j^n: approximation of the exact solution u on T_j at time $n\Delta t$;
- u_{jl}^n: approximation of the exact solution u on T_{jl} at time $n\Delta t$;
- S_{jl}: lth edge of T_j;
- ν_{jl}: outer normal to S_{jl} of length $|S_{jl}|$;
- $n_{jl} = \nu_{jl}/|\nu_{jl}|$: outer unit normal to S_{jl};
- z_{jl}: midpoint of the lth edge of the cell j;
- w_j: centre of gravity of T_j;
- w_{jl}: centre of gravity of T_{jl};
- $\alpha(j,l)$: global number of the lth neighboring cell of T_j such that $T_{jl} = T_{\alpha(j,l)}$;

To get a basic idea of the finite volume method, assume that $\varphi \in C^\infty(\mathbb{R}^2, \mathbb{R}^2)$. And let $(T_j)_{j\in I}$ be a triangulation and n the outer unit normal, then we have

$$\begin{aligned}
div\varphi &= \frac{1}{|T_j|}\int_{T_j} div\varphi + \mathcal{O}(h) \\
&= \frac{1}{|T_j|}\int_{\partial T_j} n\varphi + \mathcal{O}(h) \\
&= \frac{1}{|T_j|}\sum_{l=1}^{3}\int_{S_{jl}} n_{jl}\varphi + \mathcal{O}(h) \\
&= \frac{1}{|T_j|}\sum_{l=1}^{3} n_{jl}\varphi(z_{jl})|S_{jl}| + \mathcal{O}(h) \\
&= \frac{1}{|T_j|}\sum_{l=1}^{3} \nu_{jl}\varphi(z_{jl}) + \mathcal{O}(h).
\end{aligned} \tag{4.29}$$

Because the approximation of the exact solution is a piecewise constant function, which is constant on each element, the term $\varphi(z_{jl})$ is not defined and we replace it by some weighted mean value $g_{jl}(\varphi(w_j), \varphi(w_{jl}))$.

Finite volume scheme: For given initial values $u_0 \in L^\infty(\mathbb{R}^2)$ let u_j^n be defined by the following numerical scheme:

$$u_j^0 := \frac{1}{|T_j|} \int_{T_j} u_0, \tag{4.30}$$

$$u_j^{n+1} := u_j^n - \frac{\Delta t}{|T_j|} \sum_{l=1}^{k} g_{jl}(u_j^n, u_{jl}^n), \tag{4.31}$$

where for $g_{jl}, l = 1, \ldots, k$, we assume that for any $R > 0$ and for all $u, v, u', v' \in B_R(0)$ we have

$$|g_{jl}(u, v) - g_{jl}(u', v')| \leq c(R)h(|u - u'| + |v - v'|), \tag{4.32}$$

$$g_{j,\alpha(j,l)}(u, v) = -g_{\alpha(j,l),j}(v, u), \tag{4.33}$$

$$g_{j,\alpha(j,l)}(u, u) = \nu_{j,\alpha(j,l)} f(u), \tag{4.34}$$

where

$$f(u) := \begin{pmatrix} f_1(u) \\ f_2(u) \end{pmatrix}. \tag{4.35}$$

The condition (4.32) is called the local Lipschitz condition, (4.33) the conservation property and (4.34) the consistency property.

Example 4.4 (Engquist-Osher scheme)**.** *Consider a scalar conservation law in 2D. Define*

$$c_{jl} := n_{jl} f(u), \tag{4.36}$$

and

$$c_{jl}^+(u) := c_{jl}(0) + \int_0^u max\{c_{jl}'(s), 0\} ds, \tag{4.37}$$

$$c_{jl}^-(u) := \int_0^u min\{c_{jl}'(s), 0\} ds. \tag{4.38}$$

By setting

$$g_{jl}(u, v) := |S_{jl}|[c_{jl}^+(u) + c_{jl}^-(v)], \tag{4.39}$$

we get the Engquist-Osher scheme in two dimensions.
For convergence, there is a necessary condition, called CFL-condition. For the Engquist-Osher scheme the CFL-condition is:

$$\sup_{j \in I} \frac{\Delta t}{|T_j|} \sum_{l=1}^{k} max\{\nu_{jl} f'(u_j), 0\} \leq 1. \tag{4.40}$$

Chapter 5

Finite volume method in 2-D on a triangular grid

In this chapter, we will focus at the construction of the finite volume method on a triangular grid, instead of a method on a Cartesian grid. The interest in doing this, is that there are simple methods to improve the grid during the running simulation. A benefit on a triangular mesh is the possibility of local refinements. One method defines a new point in a triangle and connecting each vertex with this point. Other options are the examples shown in figure 1.1 and 1.2. A further benefit is the existence of methods to align the mesh in direction of the flux, resulting in a much more accurate solution. This methods are described by D.Kröner in [6]. In section 5.1 we describe the construction of the triangulation and its properties. In section 5.2 we build the finite volume method on this triangular mesh and look at one example. For solving reservoir simulations, we build a fast Galerkin solver in section 5.3. And finally in section 5.4 we present a reservoir simulation, combining an hyperbolic and an elliptic equation.

5.1 Mesh construction

To use a mesh for calculating a finite volume method, we first have to define its properties, and how to access them. Let us assume, that we construct the mesh on a rectangular domain. We construct this mesh as a class with its members:

- coordinates: $M \times 2$ -matrix, which defines the mesh points by its coordinates;
- elements: $N \times 3$ -matrix, which defines the three vertices of each triangular element;
- neighbors: $N \times 3$ -matrix, which stores the neighbors of each element;
- size: $N \times 1$ -matrix, which defines the size of each element;

- incenters: $N \times 2$ -matrix, which stores the coordinates of the incenter of each element;
- norm: $N \times 2 \times 3$ -matrix, which stores the three normal vectors of each element (stored with the indices as the neighbor to which it is pointing);
- edge size: $N \times 3$ -matrix, which saves the size of edges in each element (stored with the indices as the neighbor to which it is the boundary);

We decided to use MATLAB for implementation in this thesis. The advantage of MATLAB is its support of initial setups. This means, you can start implementing the main part of the program really fast.
By using MATLAB one way to set the different members of the triangulation is by using the MATLAB function set *DelaunayTri*. Most of the above mentioned members can be calculated with this set of functions.

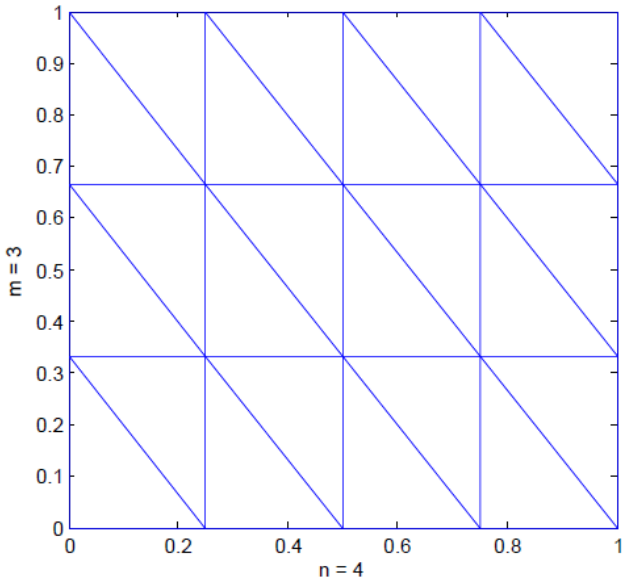


Figure 5.1: Computed grid by Listing 5-1

The problematic part are the boundaries. The members of the elements at the boundary are dependent on the boundary condition. As mentioned before, usually the boundary conditions are the periodic and the no-flux boundary condition. To identify the neighbors of the boundary element for the periodic boundary condition, do following:

- get the elements at the boundary by using the MATLAB functions *freeBoundary* and *edgeAttachments*,

- look at each element for "free edges", the edges with no neighbor,
- set as the neighbor of the "free edges" the one on the other side (for periodicity),
- set the normal vectors of this edges as $(1,0)$, $(-1,0)$, $(0,1)$ or $(0,-1)$.

In the end you just have to calculate the rest of the normal vectors by computing the orthogonal vector of each edge. Listing 5-1, Listing 5-2 illustrate this. The parameters n and m in Listing 5-1 are defined as in Figure 5.1, where $n = 4$, $m = 3$.

It has to be highlighted that it is really important that the neighbors, normal vectors and the edge sizes of the particular elements are indicated in the same manner. Because otherwise it can not be used to vectorize the method.

```
function Mesh = Built_TriangulationP(n,m)
% Construction of a periodic mesh on [0,1]x[0,1]
% Mesh.Coordinates = coordinates of vertices
% Mesh.Elements = M-by-3 matrix where M is the number of elements of
% the triangulation
% Mesh.Neighbors = Neighbors of each triangle M-by-3 matrix
% Mesh.size = volume of each triangle
% Mesh.incenters = incenters of each element
% Mesh.norm = M-by-2-by-3 matrix where Mesh.norm(i,:,j) is the
% jth normal
% of the ith element
% Mesh.edgesize = M-by-3, size of each edge


% Defining the vertices of the mesh and applying the Delaunay ↩
    Triangulation
A = zeros((n+1)*(m+1),2);
for i=1:n+1
    for j=1:m+1
        A(i+(n+1)*(j-1),:)=[(i-1)/n;(j-1)/m];
    end
end
x = A(:,1);
y = A(:,2);
dt = DelaunayTri(x,y);

ntri = size(dt.Triangulation,1);
Mesh.Coordinates = dt.X;
Mesh.Elements = dt.Triangulation;
Mesh.Neighbors = zeros(ntri,3);
Mesh.size = 1/2*abs((Mesh.Coordinates(Mesh.Elements(:,2),1)-Mesh.↩
    Coordinates(Mesh.Elements(:,1),1)).*(Mesh.Coordinates(Mesh.↩
    Elements(:,3),2)-Mesh.Coordinates(Mesh.Elements(:,1),2))-(Mesh.↩
    Coordinates(Mesh.Elements(:,3),1)-Mesh.Coordinates(Mesh.Elements↩
    (:,1),1)).*(Mesh.Coordinates(Mesh.Elements(:,2),2)-Mesh.↩
    Coordinates(Mesh.Elements(:,1),2)));
Mesh.incenters = incenters(dt);
Mesh.norm = zeros(ntri,2,3);
Mesh.edgesize = zeros(ntri,3);

% Setting the neighbors of each element by the MATLAB funct neighbors
for i=1:ntri
    neig = neighbors(dt,i);
    Mesh.Neighbors(i,:) = neig;
end

% Saving the boundary edges to calculate later the boundary values
```

```
fe = freeBoundary(dt)';
n0 = size(fe,2);
xe = x(fe);
ye = y(fe);

% fit the neighbors st we have the specific boundary conditions
Mesh=PeriodicBddCond(Mesh,dt,fe,xe,ye,n0);

%calculating the normal vectors of each edge
for i=1:ntri
    n_i = Mesh.Neighbors(i,:);
    for j=1:3
        if (Mesh.norm(i,:,j)==[0,0])
            v = Mesh.Elements(i,:);
            v1 = Mesh.Elements(n_i(j),:);
            h=1;
            for s=1:3
                for t=1:3
                    if (v(s)==v1(t)&&h==1)
                        a=v(s);
                        h=2;
                    elseif(v(s)==v1(t)&&v(s)~=a&&h==2)
                        b=v(s);
                    end
                end
            end
            n = (dt.X(a,:)-dt.X(b,:))/norm(dt.X(a,:)-dt.X(b,:))↩
                *[0,-1;1,0];
            if (norm(-Mesh.incenters(i,:)+Mesh.incenters(n_i(j),:)+n)<=↩
                norm(-Mesh.incenters(i,:)+Mesh.incenters(n_i(j),:)-n))
                n = -n;
            end
            Mesh.norm(i,:,j)=n;
            Mesh.edgesize(i,j)= norm(dt.X(a,:)-dt.X(b,:));
        end
    end
end
end
```

Listing 5-1: Triangulation of the mesh on periodic conditions

```
function [Mesh]=PeriodicBddCond(Mesh,dt,fe,xe,ye,n0)
% Calculate the periodic boundary condition on [0,1]x[0,1].
for i = 1:n0
    a = edgeAttachments(dt,fe(:,i)');   % a is the triangle attached ↩
        at the edge fe(:,i)
    a = a{:};
    x_i = xe(:,i);                      % x coordinates of the ↩
        vertices from the edge
    y_i = ye(:,i);                      % y coordinates of the ↩
        vertices from the edge
    if(y_i==[0;0])
        p_1 = [x_i(1),1];
        p_2 = [x_i(2),1];
        p1 = 0;
        p2 = 0;
        for i = 1:size(dt.X,1)
            if(p_1==dt.X(i,:))
                p1=i;
            end
            if(p_2==dt.X(i,:))
                p2=i;
            end
        end

        b = edgeAttachments(dt,p1,p2);
```

```
        b = b{:};
        h = 0;
        for j=1:3
            if (isnan(Mesh.Neighbors(a,j))==1&&h==0)
                Mesh.Neighbors(a,j)=b;
                Mesh.norm(a,:,j) = [0,-1];
                Mesh.edgesize(a,j) = sqrt((x_i(1)-x_i(2))^2+(y_i(1)-↩
                    y_i(2))^2);
                h=1;
            end
        end
    end
    if(y_i==[1;1])
        p_1 = [x_i(1),0];
        p_2 = [x_i(2),0];
        p1 = 0;
        p2 = 0;
        for i = 1:size(dt.X,1)
            if(p_1==dt.X(i,:))
                p1=i;
            end
            if(p_2==dt.X(i,:))
                p2=i;
            end
        end
        b = edgeAttachments(dt,p1,p2);
        b = b{:};
        h = 0;
        for j=1:3
            if (isnan(Mesh.Neighbors(a,j))==1&&h==0)
                Mesh.Neighbors(a,j)=b;
                Mesh.norm(a,:,j) = [0,1];
                Mesh.edgesize(a,j) = sqrt((x_i(1)-x_i(2))^2+(y_i(1)-↩
                    y_i(2))^2);
                h=1;
            end
        end
    end
    if(x_i==[0;0])
        p_1 = [1,y_i(1)];
        p_2 = [1,y_i(2)];
        p1 = 0;
        p2 = 0;
        for i = 1:size(dt.X,1)
            if(p_1==dt.X(i,:))
                p1=i;
            end
            if(p_2==dt.X(i,:))
                p2=i;
            end
        end
        b = edgeAttachments(dt,p1,p2);
        b = b{:};
        h = 0;
        for j=1:3
            if (isnan(Mesh.Neighbors(a,j))==1&&h==0)
                Mesh.Neighbors(a,j)=b;
                Mesh.norm(a,:,j) = [-1,0];
                Mesh.edgesize(a,j) = sqrt((x_i(1)-x_i(2))^2+(y_i(1)-↩
                    y_i(2))^2);
                h=1;
            end
        end
    end
    if(x_i==[1;1])
        p_1 = [0,y_i(1)];
        p_2 = [0,y_i(2)];
        p1 = 0;
```

```
        p2 = 0;
        for i = 1:size(dt.X,1)
            if(p_1==dt.X(i,:))
                p1=i;
            end
            if(p_2==dt.X(i,:))
                p2=i;
            end
        end
        b = edgeAttachments(dt,p1,p2);
        b = b{:};
        h = 0;
        for j=1:3
            if (isnan(Mesh.Neighbors(a,j))==1&&h==0)
                Mesh.Neighbors(a,j)=b;
                Mesh.norm(a,:,j) = [1,0];
                Mesh.edgesize(a,j) = sqrt((x_i(1)-x_i(2))^2+(y_i(1)-↩
                    y_i(2))^2);
                h=1;
            end
        end
    end
end
```

Listing 5-2: Construction of periodic boundary condition

5.2 Construction of the finite volume method

The finite volume method used in this section is the Engquist-Osher scheme, defined in example 4.4. Every other two dimensional, scalar finite volume method can be used instead. The aim of this thesis was to design a vectorized routine on a triangular mesh. Vectorization is requested to get better performance. To reach this goal, we use the mesh introduced in section 5.1.
Assume, we are at the nth time step at time t_n, this means that we have already calculated the vector

$$U(:,n) = (U(1,n),\dots,U(N,n))^T. \tag{5.1}$$

This is our initial value for calculating the value of the next time step. The idea is to calculate the vector $(g_{1l}(u_1^n,u_{1l}^n),\dots,g_{Nl}(u_N^n,u_{Nl}^n))^T$ for the lth neighbor. Next, we compute $U(:,n+1)$ by using equation (4.31) element wise. Because of the construction of the grid, that the lth neighbor, the lth edge and the lth normal vector are stored with the same indices, we can calculate $(g_{1l}(u_1^n,u_{1l}^n),\dots,g_{Nl}(u_N^n,u_{Nl}^n))^T$ without a for-loop statement, see Listing 5-3.

Example 5.1 (Burgers' equation)**.** *We consider the nonlinear scalar equation*

$$u_t + f_1(u)_x + f_2(u)_y = 0, \tag{5.2}$$

where $f_1(u) = 1/2u^2$, $f_2(u) = 0$ *and the domain* $U = [0,1]\times[0,1]\times[0,T]$ *with periodic boundary conditions. In Listing 5-3* n *defines the number of intersections of* $[0,1]$ *and the parameters* $mu\_1 = 1$ *and* $mu\_2 = 0$*. The derived routine is the following:*

```
function [U,Mesh] = BurgerEq(n,T,u0,mu_1,mu_2)

f = @(u) [mu_1/2*u.^2, mu_2/2*u.^2];
df = @(u) [mu_1*u,mu_2*u];

Mesh = Built_TriangulationP(n,n);
N = size(Mesh.Elements,1);

MU = ones(size(Mesh.Elements,1),2);
MU(:,1) = MU(:,1)*mu_1;
MU(:,2) = MU(:,2)*mu_2;

U = u0(Mesh.incenters);
%% Time stepping
i=1;
t = 0;
while (t(i)<T)
    % CFL-condition
    ...

    dt = ...
    t(i+1) = t(i)+dt;
    % FVM
    G_1 = abs(Mesh.edgesize(:,1)).*(U(:,i)/2.*max(sum(Mesh.norm↩
        (:,:,1).*df(U(:,i)),2),zeros(N,1))+U(Mesh.Neighbors(:,1),i↩
        )/2.*min(sum(Mesh.norm(:,:,1).*df(U(Mesh.Neighbors(:,1),i)↩
        ),2),zeros(N,1)));
    G_2 = abs(Mesh.edgesize(:,2)).*(U(:,i)/2.*max(sum(Mesh.norm↩
        (:,:,2).*df(U(:,i)),2),zeros(N,1))+U(Mesh.Neighbors(:,2),i↩
        )/2.*min(sum(Mesh.norm(:,:,2).*df(U(Mesh.Neighbors(:,2),i)↩
        ),2),zeros(N,1)));
    G_3 = abs(Mesh.edgesize(:,3)).*(U(:,i)/2.*max(sum(Mesh.norm↩
        (:,:,3).*df(U(:,i)),2),zeros(N,1))+U(Mesh.Neighbors(:,3),i↩
        )/2.*min(sum(Mesh.norm(:,:,3).*df(U(Mesh.Neighbors(:,3),i)↩
        ),2),zeros(N,1)));
    U(:,i+1) = U(:,i) - dt./(Mesh.size).*(G_1+G_2+G_3);
    i = i+1;
end
```

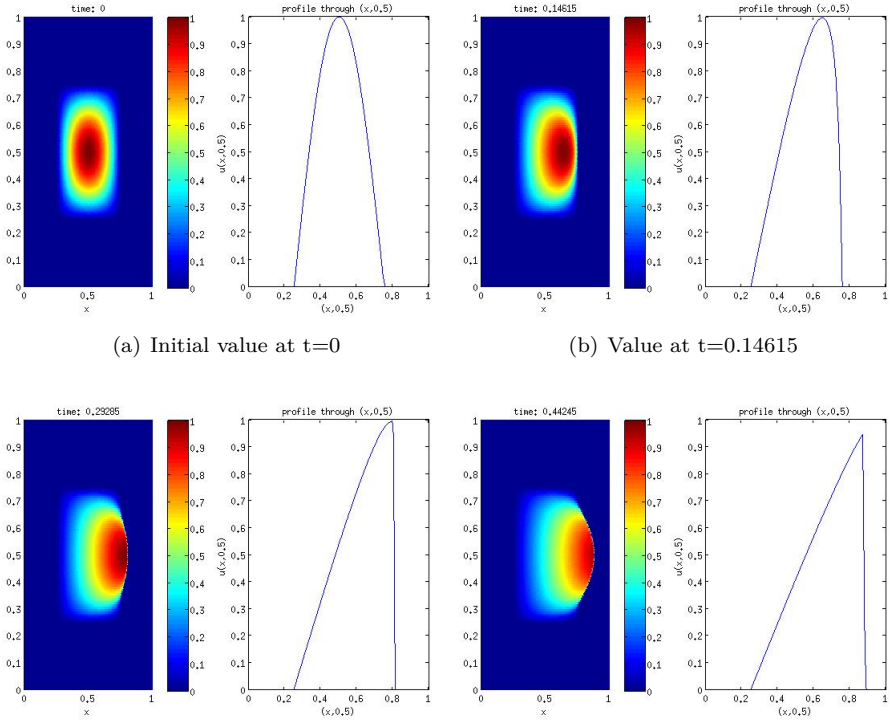
Listing 5-3: Burgers' equation by Engquist-Osher scheme

As initial condition, we take a sine curve in the middle of the domain. Figure 5.2 shows the resulting movements of this bubble and the profile through the line $y = 0.5$.
To derive the missing CFL-conditions we use equation (4.40)*, and so we derive Listing 5-4.*

```
D_1 = max([Mesh.edgesize(:,1).*sum(Mesh.norm(:,:,1).*df(U(:,i)),2),↩
    zeros(size(Mesh.edgesize(:,1)))],[],2);
D_2 = max([Mesh.edgesize(:,2).*sum(Mesh.norm(:,:,2).*df(U(:,i)),2),↩
    zeros(size(Mesh.edgesize(:,2)))],[],2);
D_3 = max([Mesh.edgesize(:,3).*sum(Mesh.norm(:,:,3).*df(U(:,i)),2),↩
    zeros(size(Mesh.edgesize(:,3)))],[],2);
D = max(1./(Mesh.size).*(D_1+D_2+D_3));
if (D>0.1)
    dt = 0.7/D;
else
    D_1 = max([Mesh.edgesize(:,1).*sum(Mesh.norm(:,:,1).*MU,2),zeros(↩
        size(Mesh.edgesize(:,1)))],[],2);
    D_2 = max([Mesh.edgesize(:,2).*sum(Mesh.norm(:,:,2).*MU,2),zeros(↩
        size(Mesh.edgesize(:,2)))],[],2);
    D_3 = max([Mesh.edgesize(:,3).*sum(Mesh.norm(:,:,3).*MU,2),zeros(↩
        size(Mesh.edgesize(:,3)))],[],2);
```

```
    D = max(1./(Mesh.size).*(D_1+D_2+D_3));
    dt = 0.7/D;
end
if(t(i)+dt>T)
    dt = T-t(i);
end
```

Listing 5-4: CFL-condition of the Engquist-Osher scheme



(a) Initial value at t=0

(b) Value at t=0.14615

(c) Value at t=0.29285

(d) Value at t=0.44245

Figure 5.2: Burger equation and its profile.

5.3 Construction of a Galerkin approximation routine

Looking ahead to section 5.4, in this section we construct a fast scheme to solve elliptic equations. Of course, we take the Galerkin method introduced in section 3.2. The challenge is to improve the performance by avoiding loops, but using vector and matrix operations.

We implement two different functions, which calculate the element stiffness matrix (3.20) and the load vector (3.18).

5.3.1 Element stiffness matrix

To get the element stiffness matrix, we use equation (3.27). First we calculate the inverse of B_K and its determinant, where invBK1_ $(i,j) = B_K^{-1}(1,j)$ and invBK2_$(i,j) = B_K^{-1}(2,j)$ for the ith element.
Then, to get the resulting matrix A, we compute the local matrix a_K and finally assemble the global one (Listing 5-5). One important fact is, that the function F is piecewise constant. Therefore it is much easier to integrate it.

```
function [A] = Assembly_mat(Mesh,F)

% this are vectors which safe the coordinates for the i-th vertex
% from every element
Vert1 = Mesh.Coordinates(Mesh.Elements(:,1),:);
Vert2 = Mesh.Coordinates(Mesh.Elements(:,2),:);
Vert3 = Mesh.Coordinates(Mesh.Elements(:,3),:);

% Element mapping BK = [a,b;c,d]
a = Vert2(:,1)-Vert1(:,1);
c = Vert2(:,2)-Vert1(:,2);
b = Vert3(:,1)-Vert1(:,1);
d = Vert3(:,2)-Vert1(:,2);

invBK1_ = [a~=0,a~=0].*[1./(a+(a==0))+c.*b./((d.*(a.^2)-c.*b.*a)+((d↩
    .*(a.^2)-c.*b.*a)==0)),-b./((d.*a-c.*b)+((d.*a-c.*b)==0))];
invBK1_ = invBK1_+[a==0,a==0].*[-d./((b.*c)+((b.*c)==0)),1./(c+(c==0))↩
    ];
invBK2_ = [a~=0,a~=0].*[-c./((a.*d-c.*b)+((a.*d-c.*b)==0)),a./((d.*a-c↩
    .*b)+((d.*a-c.*b)==0))];
invBK2_ = invBK2_+[a==0,a==0].*[1./(b+(b==0)),zeros(size(b))];

detBK = abs(a.*d-b.*c);

% calculating the local matrix
i11 = invBK1_(:,1);
i12 = invBK1_(:,2);
i21 = invBK2_(:,1);
i22 = invBK2_(:,2);

A11 = 1/2*detBK.*(F(:,1).*i11.^2+F(:,2).*i12.^2+2*F(:,1).*i11.*i21+2*F↩
    (:,2).*i12.*i22+F(:,1).*i21.^2+F(:,2).*i22.^2);
A12 = 1/2*detBK.*(-F(:,1).*i11.^2-F(:,2).*i12.^2-F(:,1).*i11.*i21-F↩
    (:,2).*i12.*i22);
A13 = 1/2*detBK.*(-F(:,1).*i11.*i21-F(:,2).*i12.*i22-F(:,1).*i21.^2-F↩
    (:,2).*i22.^2);
A21 = 1/2*detBK.*(-F(:,1).*i11.^2-F(:,2).*i12.^2-F(:,1).*i11.*i21-F↩
    (:,2).*i12.*i22);
A22 = 1/2*detBK.*(F(:,1).*i11.^2+F(:,2).*i12.^2);
A23 = 1/2*detBK.*(F(:,1).*i11.*i21+F(:,2).*i12.*i22);
A31 = 1/2*detBK.*(-F(:,1).*i11.*i21-F(:,2).*i12.*i22-F(:,1).*i21.^2-F↩
    (:,2).*i22.^2);
A32 = 1/2*detBK.*(F(:,1).*i11.*i21+F(:,2).*i12.*i22);
A33 = 1/2*detBK.*(F(:,1).*i21.^2+F(:,2).*i22.^2);

% assemble the matrices
I = zeros(9*size(Mesh.Elements,1),1);
J = zeros(9*size(Mesh.Elements,1),1);
A = zeros(9*size(Mesh.Elements,1),1);
```

```
I = [Mesh.Elements(:,1);Mesh.Elements(:,1);Mesh.Elements(:,1);Mesh.↩
    Elements(:,2);Mesh.Elements(:,2);Mesh.Elements(:,2);Mesh.Elements↩
    (:,3);Mesh.Elements(:,3);Mesh.Elements(:,3)];
J = [Mesh.Elements(:,1);Mesh.Elements(:,2);Mesh.Elements(:,3);Mesh.↩
    Elements(:,1);Mesh.Elements(:,2);Mesh.Elements(:,3);Mesh.Elements↩
    (:,1);Mesh.Elements(:,2);Mesh.Elements(:,3)];
A = [A11;A12;A13;A21;A22;A23;A31;A32;A33];
A = sparse(I,J,A);
```

Listing 5-5: Computing the element stiffness matrix

5.3.2 Load vector

Calculate the load vector of a piecewise constant function is not a too difficult task. Again, we first calculate the local Load vector and sum it up afterwards. But now, we do not need the B_K matrix, we just need the determinant of it. So we get the function Assembly_Load_PC (Listing 5-6).

```
function [L] = Assembly_Load_PC(Mesh,F)
% Built the load vector of the piecewise constant function f, where f
% is a vector, which stores for each element i the value f(i).

% this are vectors which safe the coordinates for the i-th vertex
% from every element
Vert1 = Mesh.Coordinates(Mesh.Elements(:,1),:);
Vert2 = Mesh.Coordinates(Mesh.Elements(:,2),:);
Vert3 = Mesh.Coordinates(Mesh.Elements(:,3),:);

% calculating the integral of the local load vector
detBK = abs( (Vert2(:,1)-Vert1(:,1)).*(Vert3(:,2)-Vert1(:,2))-(Vert3↩
    (:,1)-Vert1(:,1)).*(Vert2(:,2)-Vert1(:,2)));
Lloc1 = detBK.*F*1/6;
Lloc2 = detBK.*F*1/6;
Lloc3 = detBK.*F*1/6;

% assemble the load vectors
L = zeros(size(Mesh.Coordinates,1),1);
for i = 1:size(Mesh.Elements,1)
    L(Mesh.Elements(i,1)) = L(Mesh.Elements(i,1))+Lloc1(i);
    L(Mesh.Elements(i,2)) = L(Mesh.Elements(i,2))+Lloc2(i);
    L(Mesh.Elements(i,3)) = L(Mesh.Elements(i,3))+Lloc3(i);
end
```

Listing 5-6: Computing the load vector

5.4 Simulation of flow in porous media

Succeeding in implementing the finite volume routine, we like to test it on reservoir models. In this section we present some complete simulators for reservoir simulation. We look at the following problems of different difficulty:

- incompressible single-phase flow,
- incompressible two-phase flow with some assumptions,

- incompressible two-phase flow with data from 10th SPE Comparative Solution Project [9].

The following examples are based on the examples from J. E. Aarnes, T. Gimse and K. Lie in [5], who have used different solvers. In this chapter, we work with no-flow boundary condition, this means that we need another triangulation. Hence, we use Listing 5-1 and use Listing 5-7 for the boundary condition.

```
function [Mesh]=NFBddCond(Mesh,dt,fe,xe,ye,n0,a_x,a_y)
% Calculate boundary condition for no-flow condition on
% [0,a_x]x[0,a_y]
for i = 1:n0
    a = edgeAttachments(dt,fe(:,i)');
    a = a{:};
    x_i = xe(:,i);
    y_i = ye(:,i);
    check = 0;
    for j=1:3
        if (isnan(Mesh.Neighbors(a,j))==1&&check==0)
            Mesh.Neighbors(a,j)=a;
            if(y_i==[0;0])
                Mesh.norm(a,:,j) = [0,-1];
            elseif(y_i==[a_y;a_y])
                Mesh.norm(a,:,j) = [0,1];
            elseif(x_i==[0;0])
                Mesh.norm(a,:,j) = [-1,0];
            elseif(x_i==[a_x;a_x])
                Mesh.norm(a,:,j) = [1,0];
            end
            Mesh.edgesize(a,j) = sqrt((x_i(1)-x_i(2))^2+(y_i(1)-y_i(2)↩
                )^2);
            check = 1;
        end
    end
end
```

Listing 5-7: Construction of no-flow boundary condition

Example 5.2 (Incompressible single-phase flow)**.** *As in chapter 2 introduced, for the incompressible single-phase flow, we get the elliptic equation* (2.24)*. To create an example as simple as possible, assume homogeneous and isotropic permeability* $K \equiv 1$ *for all* $x \in \mathbb{R}$*. In this model, we set an injection well at the origin and production wells at the points* $(\pm 1, \pm 1)$ *and assume no-flow conditions at the boundary of* $[-1,1] \times [-1,1]$*. In the model of the single-phase flow, we have an initially filled domain with fluid. We inject additional fluid at the injection well. Due to this injection the fluid is pushed out at the production wells. The flow in this five-spot domain is symmetric about both the coordinate axes. Therefore, we can reduce this model to a model on the domain* $[0,1]^2$ *with only one injection well and one production well at* $(0,0)$ *and* $(1,1)$*. This is called a* ***quarter-five spot*** *problem.*
The pressure p *is calculated by first evaluating the element stiffness matrix* A *using Listening* **??** *and 5-5, then the load vector* L *by Listening 5-6 and in the end solving the linear equation:*

$$Ap = L. \tag{5.3}$$

This routine gives us the pressure distribution for the incompressible single-phase flow as plotted in Figure 5.3.

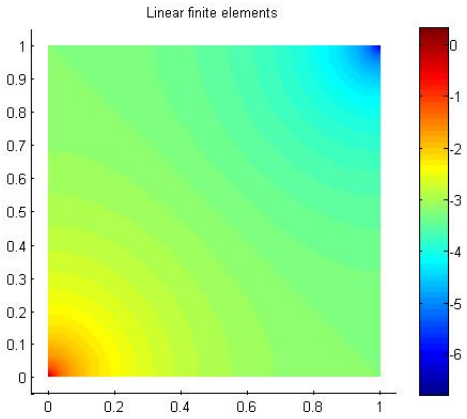


Figure 5.3: Pressure distribution for the incompressible single-phase flow

```
Mesh = Built_Triangulation_NF(120,120,1,1);

spot = sum(Mesh.size)/Mesh.size(1,1);
q = @(x,varargin) spotfunct(x,Mesh,spot, varargin);

%permeability
K = ones(size(Mesh.Elements,1),2);

N = size(Mesh.Elements,1);
Q = q(Mesh.incenters)/spot;

% Assemble stiffness matrix and load vector
A = Assembly_mat(Mesh,K);
A(1,:) = 0;
A(1,1) = 1;
L = Assembly_Load_PC(Mesh,Q*spot);
L(1)=0;

% Solve linear system
P = A\L;
```

Listing 5-8: Solver of the incompressible single-phase flow.

```
function y = spotfunct(x,Mesh,spot,varargin)
% This the function which returns at the lower-left triangle the
% value and +spot and at the upper-right triangle the value -spot.
   [v_max,l_max]=max(Mesh.incenters(:,1).^2+Mesh.incenters(:,2).^2)↩
       ; % searches the highest right triangle
   [v_min,l_min]=min(Mesh.incenters(:,1).^2+Mesh.incenters(:,2).^2)↩
       ; % searches the lowest left triangle
   a_max = Mesh.Coordinates(Mesh.Elements(l_max,:)',:);
   a_min = Mesh.Coordinates(Mesh.Elements(l_min,:)',:);
   y = zeros(size(x,1),1);
for i =1:size(x,1)
```

```
    if (dist_tri(x(i,:),a_max(1,:),a_max(2,:),a_max(3,:))<=0)
        y(i,1) = -1*spot;
    elseif(dist_tri(x(i,:),a_min(1,:),a_min(2,:),a_min(3,:))<=0)
        y(i,1) = 1*spot;
    else
        y(i,1) = 0;
    end
end

function dist = dist_tri(x,a,b,c)
%calculates the distance of x and the triangle defined
% by the pts a,b,c
    dist = -min(min(((x(:,1)-a(1))*(a(2)-b(2))-(x(:,2)-a(2))*(a(1)-↩
        b(1)))/norm(a-b), ...
         ((x(:,1)-b(1))*(b(2)-c(2))-(x(:,2)-b(2))*(b(1)-c(1)))/norm↩
             (b-c)), ...
         ((x(:,1)-c(1))*(c(2)-a(2))-(x(:,2)-c(2))*(c(1)-a(1)))/norm↩
             (c-a));
  return
```

Listing 5-9: Injection and production term.

Since the incompressible single-phase flow is a trivial case, we take now a look at a more complex example, the immiscible and incompressible two-phase flow.

Example 5.3 (Immiscible and incompressible two-phase flow)**.** *The immiscible and incompressible two-phase flow was introduced in section 2.3. In addition let us consider the setting of example 2.1, the quarter-five spot as before, no-flow boundary condition and that it takes one time unit to inject one pore-volume of water. In this model, the reservoir is initially filled with oil and we simulate a water injection at* $(0,0)$*. When solving the problem, we solve the almost-elliptic pressure equation* (2.43) *and the almost-hyperbolic saturation equation* (2.47)*. This equations are nonlinearly coupled primarily through the saturation dependent mobility* λ_i *and the pressure dependent velocity* v_i*. For simplicity, we disregard the gravity and capillary forces. So we get the following equations:*

$$-\nabla \cdot K\lambda(s)\nabla p = Q_t, \tag{5.4}$$

$$\phi\frac{\partial s}{\partial t} + \nabla \cdot (f(s)v) = \frac{q_w}{\rho_w}. \tag{5.5}$$

The strategy is to solve the two equations sequentially by the two routines derived in the previous sections 5.2 and 5.3. As long as we only inject water, and produce whatever reaches our production well, we get:

$$\frac{q_w}{\rho_w} = max\{Q_t, 0\} + f(s)min\{Q_t, 0\}. \tag{5.6}$$

For the last missing saturation dependent quantities, we use:

$$\lambda_w(s) = \frac{(s^*)^2}{\mu_w}, \quad \lambda_o(s) = \frac{(1-s^*)^2}{\mu_o}, \quad s^* = \frac{s-s_{wc}}{1-s_{or}-s_{wc}},$$

where s_{or} *is the lowest oil saturation, that can be achieved by displacing oil by water, and* s_{wc} *is the connate water saturation. To keep the problem still simple, we assume furthermore unit porosity, unit viscosity and set* $s_{or} = s_{wc} = 0$*.*
The solver of Example 5.2 can be used for the elliptic part by updating it. Instead of K *we now use* $K\lambda$*. The procedure of our solver is:*

- *solve the elliptic pde;*
- *calculate the velocity* $v = K\lambda(s)\nabla p$*, where* ∇p *is the discrete gradient, Listing 5-10;*
- *solve the hyperbolic problem between* 0 *and* Δt*, Listing 5-11;*
- *solve the elliptic pde with the new saturation;*
- *calculate the velocity* v*, Listing 5-10;*
- *solve the hyperbolic problem between* Δt *and* $2\Delta t$*, Listing 5-11;*
- ...
- *solve the elliptic pde with the saturation at final time* T*.*

Listing 5-12 contains the code to simulate all of this. For calculating the time steps, we use the CFL-condition calculated by equation (4.40)*. We choose some definitions a little bit more general, such that we can upgrade the code easily.*

```
function V = VeloTri(P,Mesh,Kf,lambda,s)
% Calculate v velocity on Mesh grid
a_1 = (P(Mesh.Elements(:,1))-P(Mesh.Elements(:,2))).*(Mesh.↩
    Coordinates(Mesh.Elements(:,3),2)-Mesh.Coordinates(Mesh.↩
    Elements(:,2),2))-(P(Mesh.Elements(:,3))-P(Mesh.Elements(:,2))↩
    ).*(Mesh.Coordinates(Mesh.Elements(:,1),2)-Mesh.Coordinates(↩
    Mesh.Elements(:,2),2));
a_2 = (Mesh.Coordinates(Mesh.Elements(:,1),1)-Mesh.Coordinates(↩
    Mesh.Elements(:,2),1)).*(Mesh.Coordinates(Mesh.Elements(:,3)↩
    ,2)-Mesh.Coordinates(Mesh.Elements(:,2),2))-(Mesh.Coordinates(↩
    Mesh.Elements(:,3),1)-Mesh.Coordinates(Mesh.Elements(:,2),1))↩
    .*(Mesh.Coordinates(Mesh.Elements(:,1),2)-Mesh.Coordinates(↩
    Mesh.Elements(:,2),2));
b_1 = (P(Mesh.Elements(:,1))-P(Mesh.Elements(:,2))).*(Mesh.↩
    Coordinates(Mesh.Elements(:,3),1)-Mesh.Coordinates(Mesh.↩
    Elements(:,2),1))-(P(Mesh.Elements(:,3))-P(Mesh.Elements(:,2))↩
    ).*(Mesh.Coordinates(Mesh.Elements(:,1),1)-Mesh.Coordinates(↩
    Mesh.Elements(:,2),1));
b_2 = -a_2;

V = -Kf.*[lambda(s),lambda(s)].*[a_1./a_2,b_1./b_2];
end
```

Listing 5-10: Computation of velocity using the pressure, $v = K\lambda(s)\nabla p$.

```
function [s,t]=EOS(s0,t0,delta_t,V,f,df,Q,phi,Mesh,s_wc,s_or)
% solves the hyperbolic equation by using the engquist osher ↩
    scheme between
% t0 and t0+delta_t with initial value s0
N = size(Mesh.Elements,1);
t=0;
st=s0;
while (t<delta_t)
    qw_over_rhow = max([Q,zeros(size(Q))],[],2)+f(st).*min([Q,↩
        zeros(size(Q))],[],2);
    % CFL-condition
    D_1 = max([Mesh.edgesize(:,1).*sum(Mesh.norm(:,:,1).*df(st,V↩
        (:,1),V(:,2)),2),zeros(size(Mesh.edgesize(:,1)))],[],2);
```

```
    D_2 = max([Mesh.edgesize(:,2).*sum(Mesh.norm(:,:,2).*df(st,V↩
        (:,1),V(:,2)),2),zeros(size(Mesh.edgesize(:,2)))],[],2);
    D_3 = max([Mesh.edgesize(:,3).*sum(Mesh.norm(:,:,3).*df(st,V↩
        (:,1),V(:,2)),2),zeros(size(Mesh.edgesize(:,3)))],[],2);
    D = max(1./(Mesh.size.*phi).*(D_1+D_2+D_3+1));
    if (D>0.1)
        dt = 0.7/D;
    else
        D_1 = max([Mesh.edgesize(:,1).*sum(Mesh.norm(:,:,1).*df(2,↩
            V(:,1),V(:,2)),2),zeros(size(Mesh.edgesize(:,1)))↩
            ],[],2);
        D_2 = max([Mesh.edgesize(:,2).*sum(Mesh.norm(:,:,2).*df(2,↩
            V(:,1),V(:,2)),2),zeros(size(Mesh.edgesize(:,2)))↩
            ],[],2);
        D_3 = max([Mesh.edgesize(:,3).*sum(Mesh.norm(:,:,3).*df(2,↩
            V(:,1),V(:,2)),2),zeros(size(Mesh.edgesize(:,3)))↩
            ],[],2);
        D = max(1./(Mesh.size.*phi).*(D_1+D_2+D_3+1));
        dt = 0.7/D;
    end
    if(t+dt>delta_t)
        dt=delta_t-t;
    end
    t = t+dt;
    % calculation of the Flux
    G_1 = abs(Mesh.edgesize(:,1)).*(f(st).*max(sum(Mesh.norm↩
        (:,:,1).*V,2),zeros(N,1))+f(st(Mesh.Neighbors(:,1))).*min(↩
        sum(Mesh.norm(:,:,1).*V,2),zeros(N,1)));
    G_2 = abs(Mesh.edgesize(:,2)).*(f(st).*max(sum(Mesh.norm↩
        (:,:,2).*V,2),zeros(N,1))+f(st(Mesh.Neighbors(:,2))).*min(↩
        sum(Mesh.norm(:,:,2).*V,2),zeros(N,1)));
    G_3 = abs(Mesh.edgesize(:,3)).*(f(st).*max(sum(Mesh.norm↩
        (:,:,3).*V,2),zeros(N,1))+f(st(Mesh.Neighbors(:,3))).*min(↩
        sum(Mesh.norm(:,:,3).*V,2),zeros(N,1)));
    st = st - dt./(Mesh.size.*phi).*(G_1+G_2+G_3)+dt.*qw_over_rhow↩
        ./(Mesh.size.*phi);
    st = max(-(1-s_wc),min(1-s_wc,st));
end
s=st;
t=t0+delta_t;
end
```

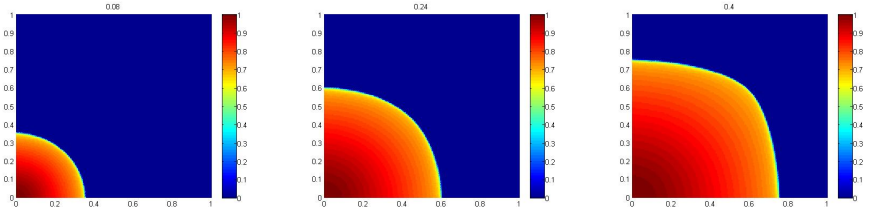
Listing 5-11: Solver of the hyperbolic problem between $t0$ and $t0+delta\_t$ using the Engquist-Osher scheme.

```
%Main code

Mesh = Built_Triangulation_NF(120,120,1,1);

T = 1; %End time
k = 1/25; % time step after which the elliptic equation is solved
nt = T/k; % number of times the elliptic equation is solved.
s_wc = 0;
s_or = 0;
mu_w = 1;
mu_o = 1;
n=1;
lambda_o = @(s) (1-(s-s_wc)./(1-s_or-s_wc)).^2/mu_o;
lambda_w = @(s) ((s-s_wc)./(1-s_or-s_wc)).^2/mu_w;
lambda = @(s) lambda_o(s)+lambda_w(s);
f = @(s)lambda_w(s)./lambda(s);
df = @(s,u,v) [2*mu_w*mu_o*(s+s_or-1).*(s-s_wc)*(s_or+s_wc-1)./(↩
    mu_w*(s+s_or-1).^2+mu_o*(s-s_wc).^2).^2.*u,2*mu_w*mu_o*(s+s_or↩
    -1).*(s-s_wc)*(s_or+s_wc-1)./(mu_w*(s+s_or-1).^2+mu_o*(s-s_wc)↩
    .^2).^2.*v];
```

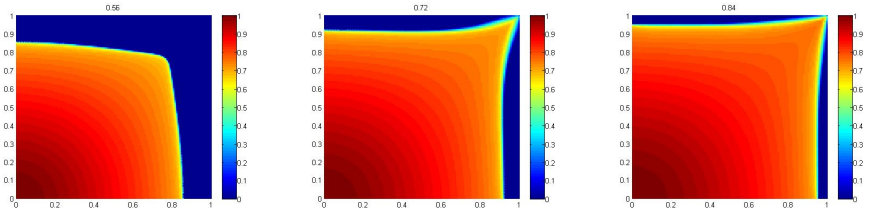
```
I = [0,-1;1,0];
s = s_wc*ones(size(Mesh.Elements,1),1); %initial saturation

spot = sum(Mesh.size)/Mesh.size(1,1);
q = @(x,varargin) spotfunct(x,Mesh,spot, varargin);
% Permeability
Kf = ones(size(Mesh.Elements,1),2);
phi = ones(size(Mesh.Elements,1),1);
N = size(Mesh.Elements,1);
Q = q(Mesh.incenters)/spot;

%% Assemble stiffness matrix and load vector at time t0 = 0
K = [lambda(s(:,1)),lambda(s(:,1))].*Kf;
A = Assembly_mat(Mesh,K);
A(1,:) = 0;
A(1,1) = 1;
L = Assembly_Load_PC(Mesh,Q*spot);
L(1)=0;

% Solve linear system
P = zeros(size(Mesh.Coordinates,1),nt+1);
P(:,1) = A\L;

% Calculate v velocity on Grid
V=VeloTri(P(:,1),Mesh,Kf,lambda,s(:,1));

%% Time stepping
t=0;
for i = 1:nt
    i
    % Solve hyperbolic part between t and t+k
    [s(:,i+1),t] = EOS(s(:,i),t,k,V,f,df,Q*spot,phi,Mesh,s_wc,s_or↩
        );

    % Assemble stiffness matrix and load vector at time i*k
    K = [lambda(s(:,i+1)),lambda(s(:,i+1))].*Kf;
    A = Assembly_mat(Mesh,K);
    A(1,:) = 0;
    A(1,1) = 1;
    L = Assembly_Load_PC(Mesh,Q*spot);
    L(1) = 1;

    % Solve linear system
    P(:,i+1) = A\L;

% Calculate v velocity on Mesh grid
    V=VeloTri(P(:,i+1),Mesh,Kf,lambda,s(:,i+1));

end
```

Listing 5-12: Simulator for the immiscible and incompressible two-phase flow.

The saturation distribution computed by Listing 5-12 is shown in figure 5.4. The appropriate functionality of the implementation is established by the following two results. The first is, that the basic structure looks correct. And the second is, that the theoretical value of the break through, the time at which the water first reaches the production well, coincides with the theoretical value, what is approximately 0.7 pore-volumes of water [5].
If you look at the exact shape of the water wave, you will recognize first a shock wave and after this a monotonically increasing saturation.
Comparing the L^1-error over time and space of the solution computed by the program of this thesis and the solution of J. E. Aarnes, T. Gimse and K. A. Lie in [5] we get a sublinear convergence plotted in figure 5.5.



(a) Value at t=0.08 (b) Value at t=0.24 (c) Value at t=0.40



(d) Value at t=0.56 (e) Value at t=0.72 (f) Value at t=0.84

Figure 5.4: Saturation profiles for the homogeneous quarter-five spot.

Example 5.4 (Simulation of the SPE-model by using finite volume method)**.** *In the example 5.3, we still looked at a simplified two-phase model. Let us take a look at a much more realistic setting of porosity and permeability, by looking at the model 2 from the 10th SPE Comparative Solution Project [9]. The model dimensions are* $1200 \times 2200 \times 170$ *(ft) and the reservoir is described by a heterogeneous distribution over a regular Cartesian grid with* $60 \times 220 \times 85$ *grid-blocks. Because we use a triangular mesh, we have to transform this to a triangular grid, Listing 5-13. We use only the top layer, in which the permeability is smooth. We set again the injection well at the lower-left corner and the production well at the upper-right one. For brevity, gravity and capillary forces are neglect. And we consider a incompressible oil-water system, for which* $s_{wc} = s_{or} = 0.2$, $\mu_w = 0.3$ *cp and* $\mu_o = 3.0$ *cp. The reservoir is initially filled with oil, this means that the initial water saturation is equal the connate water saturation. Since the porosity is not uniform anymore, it may be zero. To avoid division by zero, we set it to a minimal nonzero value.*

Figure 5.6 shows the saturation distribution over the first 2000 days computed by Listing 5-14. Figure 5.7 shows the distribution computed by J. E. Aarnes, T. Gimse and K. A. Lie in [5]. By comparing the different results, it is obvious that in our solution the water is much faster than in the solution of [5]. The difference between the simulated flows occurs probably from a different setting of parameters. By comparing the different production rates (figure 5.8), you see that the presented algorithm in this thesis yields the correct behavior.
Comparing the derived code of this thesis and the code given in [9], you notice

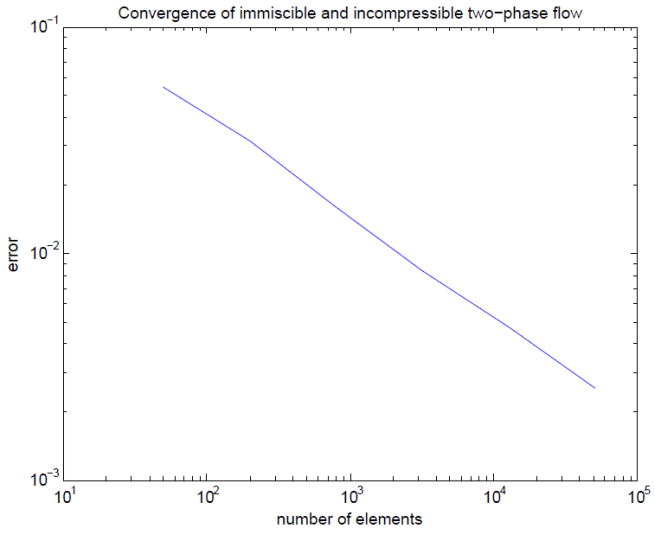


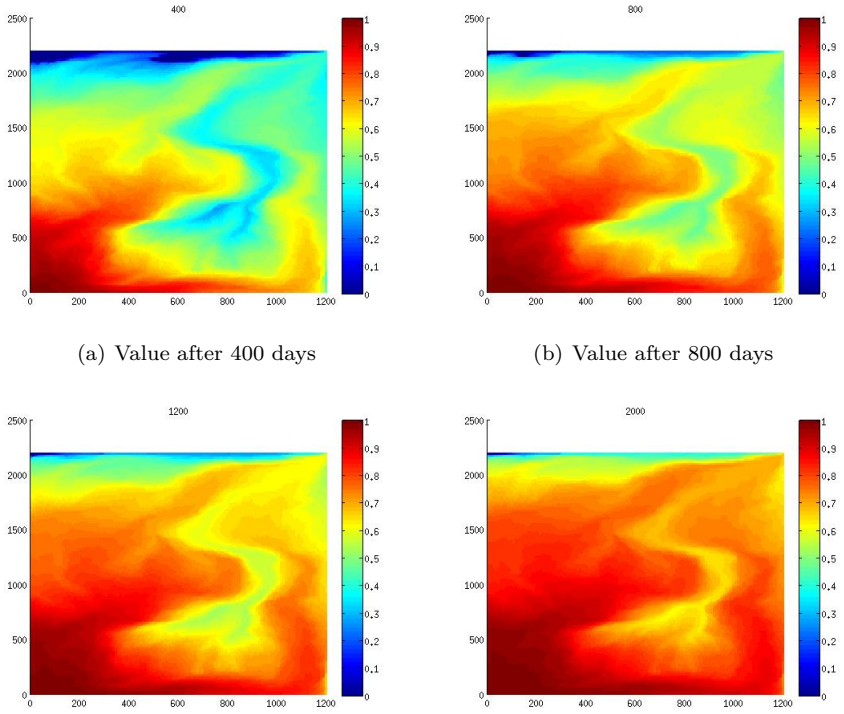
Figure 5.5: Rate of convergence for the immiscible and incompressible two-phase flow

that the original code is much faster than this one. The reason is, that they use an implicit solver, in comparison to our explicit solver. In a implicit solver the discretization is a large nonlinear system of equations, which are usually solved with a Newton or a Newton-Raphson iterative method. The problem with explicit solvers is that for convergence, we need to fulfill a CFL-condition, which in the heterogeneous case gives us really tiny time steps. So in this case an implicit solver would be faster.

```
function [phi,Kf] = load_Phi_Perm(Mesh,spe_perm,spe_phi)
% Transformation of the permeability perm and porosity phi to the
% triangular mesh
load spe_perm.dat; load spe_phi.dat;
perm2 = reshape(spe_perm',size(spe_perm,1)*2,3);
perm = zeros(60,220,2);
for i = 1:60, for j = 0:219, l = 0; perm(i,j+1,1) = perm2(i+j*60+l↩
    *220*60,1);end,end
for i = 1:60, for j = 0:219, l = 0; perm(i,j+1,2) = perm2(i+j*60+l↩
    *220*60,2);end,end

spe_phi2 = reshape(spe_phi',size(spe_phi,1)*size(spe_phi,2),1);
phi = zeros(60,220,85);
for i = 1:60, for j = 0:219, for l = 0, phi(i,j+1,1 ) = spe_phi2(i↩
    +j*60+l*220*60);end,end,end
Kg=perm; % Permeability in layer 1
Por=phi(:,:,1); % Preprocessed porosity in layer 1

Kf = zeros(size(Mesh.Elements,1),2);
phi = zeros(size(Mesh.Elements,1),1);
for i = 1:size(Mesh.Elements,1)
    phi(i) = Por(floor(Mesh.incenters(i,1)/20)+1,floor(Mesh.↩
        incenters(i,2)/10)+1);
```

```
    Kf(i,:) = Kg(floor(Mesh.incenters(i,1)/20)+1,floor(Mesh.↩
        incenters(i,2)/10)+1,:);
end
phi=max(phi,1e-3);
```

Listing 5-13: Transformation of the data of the SPE-10 model to the triangular grid.



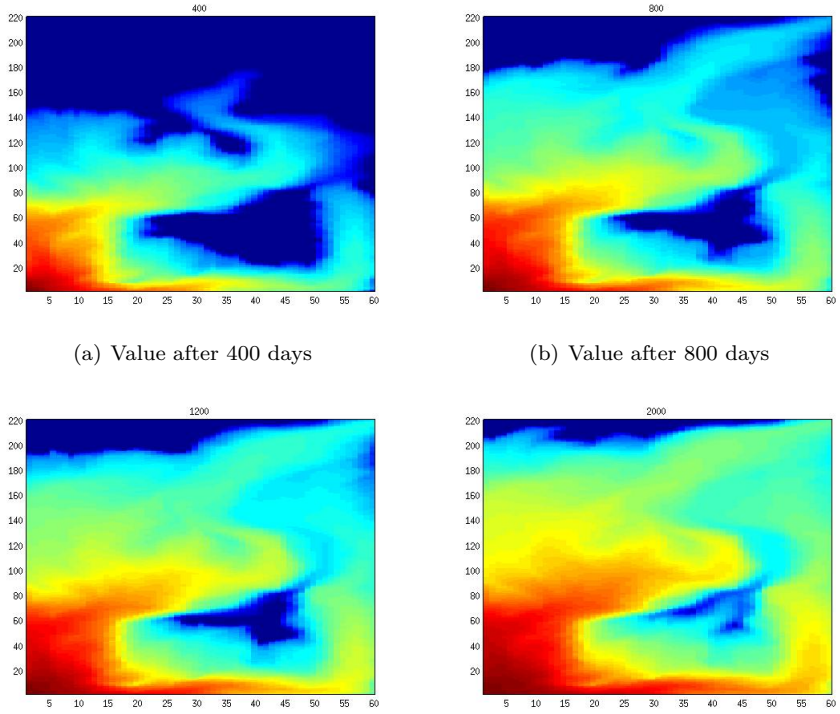
(a) Value after 400 days

(b) Value after 800 days

(c) Value after 1200 days

(d) Value after 2000 days

Figure 5.6: Saturation profiles in the top layer of the SPE-10 model, computed by Listing 5-14.



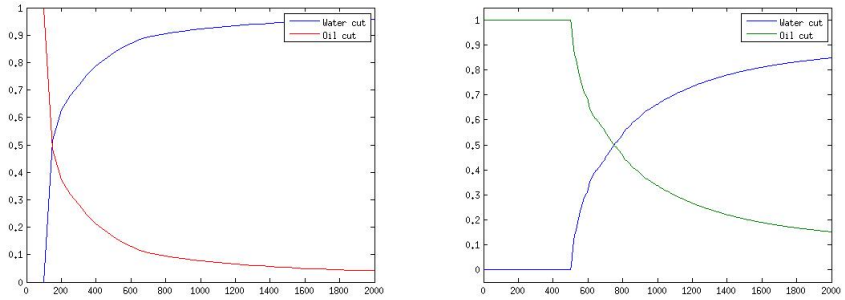
(a) Value after 400 days

(b) Value after 800 days

(c) Value after 1200 days

(d) Value after 2000 days

Figure 5.7: Saturation profiles in the top layer of the SPE-10 model, computed by J. E. Aarnes, T. Gimse and K. A. Lie in [5].



(a) Solution from Listing 5-14

(b) Solution from [5]

Figure 5.8: Production profiles.

```
%Main code

% Construct mesh
Mesh = Built_Triangulation_NF(120,120,1200,2200);
spot =795/85*(120*120*2)/(60*220);
q = @(x,varargin) spotfunct3(x,Mesh,spot,varargin);
T = 1000;
k = 50;
nt = T/k;
s_wc = 0.2;
s_or = 0.2;
mu_w = 0.3;%3e-4;
mu_o = 3;%3e-3;
lambda_o = @(s) (1-(s-s_wc)./(1-s_or-s_wc)).^2/mu_o;
lambda_w = @(s) ((s-s_wc)./(1-s_or-s_wc)).^2/mu_w;
lambda = @(s) lambda_o(s)+lambda_w(s);
f = @(s)lambda_w(s)./lambda(s);
df = @(s,u,v) [2*mu_w*mu_o*(s+s_or-1).*(s-s_wc)*(s_or+s_wc-1)./(mu_w*(↩
    s+s_or-1).^2+mu_o*(s-s_wc).^2).^2.*u,2*mu_w*mu_o*(s+s_or-1).*(s-↩
    s_wc)*(s_or+s_wc-1)./(mu_w*(s+s_or-1).^2+mu_o*(s-s_wc).^2).^2.*v];
I = [0,-1;1,0];
s = s_wc*ones(size(Mesh.Elements,1),1);
%n=1;
% Permeability and porosity
[phi, Kf] = load_Phi_Perm(Mesh,spe_perm,spe_phi);

N = size(Mesh.Elements,1);
Q = q(Mesh.incenters)/spot;
%% Assemble stiffness matrix and load vector at time t0 = 0
K = [lambda(s(:,1)),lambda(s(:,1))].*Kf;
A = Assembly_mat(Mesh,K);
A(1,:) = 0;
A(1,1) = 1;
L = Assembly_Load_PC(Mesh,Q*spot);
L(1)=0;

% Solve linear system
P = zeros(size(Mesh.Coordinates,1),nt+1);
P(:,1) = A\L;

% Calculate v velocity on Mesh grid
V=VeloTri(P(:,1),Mesh,Kf,lambda,s(:,1));

%% Time stepping
t=0;
for i = 1:nt
    i
    % Solve hyperbolic part between t and t+k
    [s(:,i+1),t] = EOS(s(:,i),t,k,V,f,df,Q*spot,phi,Mesh,s_wc,s_or);

    % Assemble stiffness matrix and load vector at time i*k
    K = [lambda(s(:,i+1)),lambda(s(:,i+1))].*Kf;
    A = Assembly_mat(Mesh,K);
    A(1,:) = 0;
    A(1,1) = 1;
    L = Assembly_Load_PC(Mesh,Q*spot);
    L(1) = 1;

    % Solve linear system
    P(:,i+1) = A\L;

    % Calculate v velocity on Mesh grid
    V=VeloTri(P(:,i+1),Mesh,Kf,lambda,s(:,i+1));

end
```

Listing 5-14: Simulator for the immiscible and incompressible two-phase flow for the top layer of the SPE-10 model.

Chapter 6

Conclusion and further work

The aim of this thesis was to design a routine to solve conservation laws by finite volume methods on triangular grids. This goal was achieved in chapter 5. The tests of the implementation were successful. The behavior of the Burgers equation, incompressible single-phase flow and incompressible two-phase flow are correct and the break-through of the water in example 5.3 fits to theory. The disadvantage of this implementation is the run-time. It is really large, compared to the run-time of an implicit method. Possibly there is some potential of improvement. The other time-consuming part is the building of the triangulation.
Unfortunately, we could not implement a finite volume method for systems of conservation laws. But this should work in the same way by using the constructed triangular mesh of section 5.1.
A remaining interesting question is the convergence behavior of the method on a triangular grid compared to a method on a Cartesian grid. As well, it would be interesting to extend the code by some automatically local refinement or even local grid alignment.
Hopefully this thesis will motivate and make curious to develop methods on triangular grids for more dimensional and complex applications.

Bibliography

[1] D. Braess. *Finite Elements*. Cambridge University Press, 2007.

[2] G. Chavent and J. Jaffre. *Mathematical models and finite elements for reservoir simulation*. North Holland, 1982.

[3] M. G. Crandall and A. Majda. The method of fractional steps for conservation laws. *Numerische Mathematik*, 34:285–314, 1980.

[4] L. C. Evans. *Partial differential equations*. AMS, 2010.

[5] T. Gimse J. E. Aarnes and K. A. Lie. An introduction to the numerics of flow in porous media using matlab. 2007.

[6] D. Kröner. *Numerical Schemes for Conservation Laws*. Wiley Teubner, 1997.

[7] S. N. Kruzkov. First order quasilinear equations in several independent variables. *Math. USSR Sbornik*, 10:217–243, 1970.

[8] R. J. LeVeque. *Numerical Methods for Conservation Laws*. Birkhäuser, 1992.

[9] M. Blunt M. Christie. Spe comparative solution project, 2001. http://www.spe.org/web/csp/ (11.06.2012).

[10] D. W. Peaceman. *Fundamentals of numerical reservoir simulation*. Elsevier scientific publishing company, 1977.

[11] G. Huan Z. Chen and Y. Ma. *Computational methods for multiphase flows in porous media*. Society for Industrial and Applied Mathematics (SIAM), 2006.