MASTER'S THESIS IN MATHEMATICAL SCIENCES

Automatic Differentiation in Julia with Applications to Numerical Solution of PDEs

Author:
SINDRE GRØSTAD

Supervisor:
Professor KNUT-ANDREAS LIE

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEPARTMENT OF MATHEMATICAL SCIENCES

June 4, 2019

Preface

This master's thesis is part of my Master of Science in Industrial Mathematics at the Norwegian University of Science and Technology, Department of Mathematical Sciences. The thesis was carried out in collaboration with SINTEF Digital.

I would like to thank the group from the Computational Geosciences at the department of Mathematics and Cybernetics at SINTEF Digital, consisting of my supervisor Knut-Andreas Lie, Atgeirr Rasmussen, Olav Møyner, and Øystein Klemetsdal for valuable guidance through this thesis.

June 4, 2019
Sindre Grøstad

Abstract

This thesis provides an introduction to Automatic Differentiation (AD) and how it can be utilized to calculate the derivatives of any function with the same precision as an analytic expression, with very little computational effort. It has been used to elegantly solve Partial Differential Equations (PDEs) describing flow in porous media, by using a finite volume method and a discretization of the gradient- and divergence operator. The goal of the thesis has been to investigate whether the new programming language called Julia, can be used both as a language for quickly prototyping new oil recovery simulators, as well as implementing highly efficient industrial simulators.

Three different AD libraries have been implemented in Julia and compared to implementations from the MATLAB Reservoir Simulation Toolbox (MRST) *(MRST Homepage, n.d.)*, created by the Computational Geosciences group at the department of Mathematics and Cybernetics at SINTEF Digital. MRST is a toolbox designed for quick prototyping with high-level- and user-friendly AD tools. The first two implementations in Julia were inspired by the AD tools in MRST, and were called `ForwardAutoDiff(FAD)` and `CustomJacobianAutoDiff(CJAD)`. The third implementation, local AD, was inspired by the implementation of AD in the Open Porous Media (OPM) Flow simulator *(Open Porous Media Hompage, n.d.)*. OPM is a toolbox developed by the same group at SINTEF for creating efficient industrial simulators in C and C++.

To benchmark the AD methods, two simulations were implemented. The first one was a single-phase flow solver, simulating the pressure drop in a reservoir when producing oil. `FAD` was the slowest method being approximately two times slower than both `CJAD` and the implementation in MRST. While `CJAD` and MRST exhibited similar performance, the method of Local AD was approximately six times faster than these two. The second simulation was a two-phase flow solver, simulating how water flows when injected into a reservoir. Analogous to the first simulation, `CJAD` and MRST yielded similar performance, while the method using local AD was approximately five times faster.

The benchmarks show promising results suggesting that Julia may be a language enabling making prototypes of simulators, using a user-friendly AD tool like `CJAD`, as well as creating high performance industrial simulators, using an AD tool like local AD.

Table of Contents

Chapter 1

Introduction

1.1 Automatic Differentiation

Automatic differentiation (AD) is a computational method that automatically calculates exact values for the derivatives of a function. It is, however, not the method of finite differences, nor symbolic differentiation. The method consists of separating an expression into a finite set of elementary operations, $+, -, *$ and $/$, and elementary functions like the exponential and the logarithm. It then applies standard differentiation rules to these operations and functions. However, unlike the way we calculate derivatives by hand, it does not apply differentiation rules to the symbols. Instead, it proceeds with both the function- and the derivative values for all the elementary steps of the evaluation. In this way, it can calculate the next function- and derivative values, based on the current values and the rules for the next elementary operation or function. This yields derivative values that, up to roundoff errors, are as accurate as manually computed derivatives, but without potential human errors and with low computational cost. AD can be split into two different methods: backward AD and forward AD. They both obtain the derivatives, but with different approaches each having different capabilities. The exact difference and capabilities between the two will be discussed closer in section 2.1.

According to *Baydin et al. (2018)*, the first ideas of the AD concept emerged in the 1950s *(Nolan, 1953; Beda et al., 1959)*. More specifically, forward AD was then discovered by *Wengert (1964)*. It is more difficult to date exactly when backward AD was discovered, but the first article containing the essence of backward AD dates back to the 1960s *(Boltyanskii et al., 1960)*. After the initial discovery of forward and backward AD, no significant activities took place for a couple of years, before the method was rediscovered along with the birth of modern computers and computer languages. The first running computer program that used backward AD, and automatically computed the derivatives, arrived in 1980 developed by *Speelpenning (1980)*. Further research on the topic was done by, among others, *Griewank et al. (1989)*. Today, AD is widely used in many applications. One of them is machine learning that specifically uses the backward AD method to minimize functions.

1.2 Objective of the Thesis

The primary focus of this thesis is to use the forward AD method to solve PDEs, using the new programming language Julia; and more specifically, solve PDEs that describe flow in porous media.

The objective is to figure out whether Julia can be used as a language both for rapid prototyping as well as building efficient industrial simulators.

Simulation of flow in porous media can be used in several applications. Examples include modelling the movement of groundwater reserves, subsurface storage of CO_2 to reduce environmental footprint, and reservoir simulations to maximize the amount of hydrocarbons that can be recovered from a reservoir. All these are examples of flow below the surface of the earth, but simulation of flow in porous media can also be used to understand flow inside batteries, fuel cells, textiles, for water purification, and even to describe processes inside our human bodies. However, this thesis will solely look at using AD to simulate flow in oil reservoirs.

1.3 Oil Reservoirs

The textbook, *An Introduction to Reservoir Simulation Using MATLAB/GNU Octave*, by *Lie (2019)*, offers a thorough review of the properties of an oil reservoir and how numerical simulations can be used to understand oil recovery processes. I will not provide a comprehensive review of the subject, as it is outside the scope of this thesis, but I will give a brief introduction to the topic, explaining why it is of interest to use the new programming language Julia to solve these PDEs using AD.

Lie uses an analogy to explain the properties of an oil reservoir. A simplified model of a typical oil reservoir can be compared to a rigid sponge completely soaked in oil. The sponge is compressed under high pressure with solid walls on all sides, preventing any oil from escaping. In reality, oil lies in between tightly compacted sand and mineral particles that constitute a porous medium with the ability to transmit fluids.

If we drill a hole in one of the solid walls surrounding the sponge to create a well, and provided that the pressure is high enough, the oil will flow out of the well while the pressure inside the sponge will decline. In oil recovery, this is called the primary production, during which the reservoir does all the work and the oil floats out on its own. For most oil reservoirs, primary production will extract up to 30% of the oil in the reservoir. To extract more, we have to apply external pressure to the reservoir. One of many methods is to pump salt or fresh water into the reservoir through injectors. This increases the pressure close to the injector and push the oil towards the wells. This is called secondary production. According to (Lie, 2019), the average percentage of oil extracted from oil reservoirs on the Norwegian Continental Shelf is approximately 50%. This means there is still a large quantity of oil we are not able to extract. An increase in the amount we are able to extract will not only be economically beneficial for the owners of the reservoir, but it can also help avoiding exploration for new oil reservoirs in more vulnerable areas. This is why it is so important to be able to simulate the flow inside oil reservoirs. If we can find optimal and better methods to recover more oil from each reservoir, it can make a huge impact both financially and environmentally.

1.4 Creating a Simulator

The process of making new simulators for oil reservoir modelling, is typically started by creating a prototype. The prototype is implemented quickly, usually in a high-level programming language, to illustrate what the finished product will do. This is usually applied to a simplified and conceptual reservoir model, since such prototypes rarely can handle the full extent of a full reservoir simulation. After the prototype phase, performance improvements and further implementations are needed before the actual simulator is ready to perform the simulation on the full reservoir.

The Computational Geosciences group at the department of Mathematics and Cybernetics at SINTEF Digital has created the MATLAB Reservoir Simulation Toolbox (MRST) *(MRST Homepage, n.d.)* for the purpose of prototyping. According to MRST's homepage, "MRST is not primarily a simulator, but is mainly intended as a toolbox for rapid prototyping and demonstration of new simulation methods and modelling concepts." Most of the tools and simulators in MRST are surprisingly efficient and perform well, even for medium-sized models of real reservoirs. However, for more extensive simulations, the Computational Geoscience group recommend using the Open Porous Media (OPM) Flow simulator *(Open Porous Media Hompage, n.d.)*. OPM is a toolbox to build simulations of porous media processes which is mainly written in C and C++.

This is where the problem with the building process of a numerical simulator arises. MATLAB is a high-level language perfect for quickly making prototypes and demonstrations because of its easy-to-use mathematical syntax. It is, however, not ideal for a final simulator, because it is not computationally efficient enough. Instead, we have to implement the final simulator in a lower-level language that is more computationally efficient, such as C or C++. However, C and C++ are not designed for numerical analysis, hence it usually takes longer time and more programming effort to create the simulator. Experience shows that the development time is usually reduced if a prototype is initially created in a high-level scripting language, and then a full-featured simulator is reimplemented in a compiled language. The obvious disadvantage of this approach is that there will be a lot of code from the prototype cannot be used in the final simulator. It is possible to call, for example, C++ code from MATLAB, but it is difficult to make the languages interact properly. This may lead to unstable code and is not an optimal solution. The ideal solution would be a language that supports high-level scripting for rapid prototyping and at the same time offers the full computational efficiency of a compiled language.

This is where the new programming language Julia comes into play. Julia is a language built from scratch, with focus on mathematical programming. It is meant to be a language as familiar as MATLAB in terms of mathematical notations, but as fast as C in terms of computational speed. If the developers have managed to do this, it will be possible, in one single language, to create prototypes and demonstrations and then further develop the existing code into high-performing simulators. This could drastically increase the efficiency of creating a simulator, not only for oil reservoirs, but for many other applications.

1.5 Outline of Thesis

The thesis investigates whether Julia is a language that can be used to implement both prototypes and efficient industrial simulators for flow in oil reservoirs. More specifically, it examines how Julia performs when implementing AD and using it in oil reservoir simulations. The thesis begins by describing the theory behind AD and the difference between backward and forward AD in chapter 2. The chapter continues describing different applications of AD and how we can use forward AD to elegantly solve PDEs using a finite-volume method and a discretization of the differentiation operators. The history of Julia, and the reasons why it is presumed to be as fast as C, but as convenient as MATLAB, will be discussed in chapter 3. Implementation-specific parts, concerning how one can develop high-performing AD tools in Julia, is discussed in chapter 4. It describes two implementations and tests them against a third-party AD implementation in Julia and implementations from MRST. The implemented AD libraries will then be used in chapter 5 to benchmark the performance in a prototype simulator from MRST, simulating primary production. Motivated by the results and the characteristics of the problem in chapter 5, chapter 6 describes another implementation of AD in Julia and benchmarks this new AD tool against the previous tools for simulating secondary production. Lastly, chapter 7 provides a summary of the results and a

performance review of Julia. It also describes conditions and areas not tested in this thesis, and outlines problems and challenges that warrants further inquiry.

Chapter 2

Theory

2.1 Automatic Differentiation

Automatic differentiation (AD) is a method that enables a computer to numerically evaluate the derivative of a function specified by a computer program with very little effort from the user. If you have not heard of AD before, the first thing you might think of is algebraic or symbolic differentiation. In this type of differentiation the computer learns the basic rules from calculus

$$\frac{d}{dx}x^n = n \cdot x^{n-1},$$
$$\frac{d}{dx}\cos(x) = -\sin(x),$$
$$\frac{d}{dx}\exp(x) = \exp(x),$$

and so on, as well as the chain- and product rule

$$\frac{d}{dx}f\big(g(x)\big) = g'(x) \cdot f'\big(g(x)\big)$$
$$\frac{d}{dx}f(x) \cdot g(x) = f'(x) \cdot g(x) + f(x) \cdot g'(x).$$

The computer will then use these rules on symbolic variables to obtain the derivative of any given function. This will give perfectly accurate derivatives, but as the function evaluated becomes more complex, the computed expression for the derivative grows and becomes large. This will make the evaluation of these derivatives very slow. The reason for this effect, that is often referred to as expression swell, is that it is difficult to simplify the exact expressions calculated by the symbolic differentiation. The more complex the function is, the larger will the expression for the exact derivative become, and for computer programs, which often include if-statements and for-loops, it can be very difficult to express the derivative efficiently using symbolic differentiation.

If AD is not symbolic differentiation, you might think that it is finite differences, where you use the definition of the derivative

$$\frac{df}{dx} = \frac{f(x+h) - f(x)}{h}$$

with a small h to obtain the numerical approximation of the derivative of f. This approach is not

optimal because, first of all, if you choose an h too small, you will get problems with rounding errors on your computer. This is because when h is small, you will subtract two very similar numbers, $f(x+h)$ and $f(x)$, and then divide by a small number h. This means that any small rounding errors in the subtraction, which may occur due to machines having a finite precision when storing numbers, will be amplified by the division. Secondly, if you choose h too large, your approximation of the derivative will not be accurate. This is called the truncation error. Hence, with finite differences you have the problem that you need a sufficiently small step size h to reduce the truncation error, but h can not be too small, because then you get round-off errors. Finding the correct value for h can be difficult and wrong choice of h can lead to unstable methods. This is neither what we call AD.

AD can be split into two different methods – forward AD and backward AD. Both methods are similar to symbolic differentiation in the way that we implement the differentiation rules, but they differ by instead of differentiating symbols and then inserting values for the symbols, the methods keep track of the function values and the corresponding values of the derivatives as we go. Both methods do this by separating each expression into a finite set of elementary operations.

2.1.1 Forward Automatic Differentiation

In forward AD, the function and derivative value are stored in a tuple $[\cdot,\cdot]$. This tuple is called an AD-variable. In this way, we can continuously update both the function value and the derivative value for every operation we perform on a given AD-variable.

As an example, consider the scalar function $f = f(x)$ with its derivative f_x, where x is a scalar variable. If we evaluate this function for the AD-variable x, represented as the tuple $[x,\ 1]$, the result will be the AD-variable $[f,\ f_x]$. In the tuple $[x,\ 1]$, x is the numerical value of x and $1 = \frac{dx}{dx}$. Similar for $f(x)$, where f is the numerical value of $f(x)$, and f_x is the numerical value of $f'(x)$. We then define the four elementary arithmetic operators for the AD-variables, such that for functions f and g,

$$\begin{aligned}
&[f,\ f_x] \pm [g,\ g_x] = [f \pm g,\ f_x \pm g_x],\\
&[f,\ f_x] \cdot [g,\ g_x] = [f \cdot g,\ f_x \cdot g + f \cdot g_x],\\
&\frac{[f,\ f_x]}{[g,\ g_x]} = \left[\frac{f}{g},\ \frac{f_x \cdot g - f \cdot g_x}{g^2}\right].
\end{aligned} \tag{2.1}$$

It is also necessary to define the chain rule such that for a function h(x)

$$h\big(f(x)\big) = h\Big([f,\ f_x]\Big) = [h(f),\ f_x \cdot h'(f)].$$

Lastly the rules concerning the elementary functions are defined

$$\begin{aligned}
&\exp\Big([f,\ f_x]\Big) = [\exp(f),\ \exp(f) \cdot f_x],\\
&\log\Big([f,\ f_x]\Big) = \left[\log(f),\ \frac{f_x}{f}\right],\\
&\sin\Big([f,\ f_x]\Big) = [\sin(f),\ \cos(f) \cdot f_x], \text{ etc.}
\end{aligned} \tag{2.2}$$

When these arithmetic operators and the elementary functions are implemented, the program is able to evaluate the derivative of any scalar function without any user input. Let us look at a step by step

example, where

$$f(x) = x \cdot \exp(2x) \qquad \text{for} \quad x = 2. \tag{2.3}$$

The declaration of the AD-variable gives $x = [2,\ 1]$. All scalars can be viewed as AD-variables with derivative equal to 0, such that

$$2x = [2,\ 0] \cdot [2,\ 1] = [2 \cdot 2,\ 0 \cdot 2 + 2 \cdot 1] = [4,\ 2].$$

After this computation, we get from the exponential

$$\exp(2x) = \exp\big([4,\ 2]\big) = [\exp(4),\ \exp(4) \cdot 2],$$

and lastly from the product rule, we get the correct tuple for $f(x)$

$$\begin{aligned} x \cdot \exp(2x) &= [2,\ 1] \cdot [\exp(4),\ 2 \cdot \exp(4)] \\ &= [2 \cdot \exp(4),\ 1 \cdot \exp(4) + 2 \cdot 2 \cdot \exp(4)] \\ [f,\ f_x] &= [2 \cdot \exp(4),\ 5 \cdot \exp(4)]. \end{aligned}$$

This result equals what we obtain from the analytical expression evaluated at $x = 2$

$$\big(f(x),\ f'(x)\big) = \big(x \cdot \exp(2x),\ (1 + 2x)\exp(2x)\big).$$

2.1.2 Dual Numbers

One approach to implementing forward AD is by dual numbers. Similarly to complex numbers, dual numbers are defined as

$$a + b\epsilon. \tag{2.4}$$

Here, a and b are scalars and correspond to the function value and the derivative value, whereas ϵ is like we have for complex numbers $i^2 = -1$, except that the corresponding relation for dual numbers is $\epsilon^2 = 0$. The convenient part of implementing forward AD with dual numbers is that you get the differentiation rules for arithmetic operations for free. Consider the dual numbers x and y on the form of (2.4). We then get the following for addition

$$x + y = (a + b\epsilon) + (c + d\epsilon) = a + c + (b + d)\epsilon,$$

and likewise for multiplication

$$x \cdot y = (a + b\epsilon) \cdot (c + d\epsilon) = ac + (ad + bc)\epsilon + bd\epsilon^2 = ac + (ad + bc)\epsilon,$$

and for division

$$\frac{x}{y} = \frac{a + b\epsilon}{c + d\epsilon} = \frac{a + b\epsilon}{c + d\epsilon} \cdot \frac{c - d\epsilon}{c - d\epsilon} = \frac{ac - (ad - bc)\epsilon - bd\epsilon^2}{c^2 - d\epsilon^2} = \frac{a}{c} + \frac{bc - ad}{c^2}\epsilon.$$

This is very convenient, but how does dual numbers handle elementary functions like sin, exp, log? If we look at the Taylor expansion of a function $f(x)$, where x is a dual number, we get

$$f(x) = f(a + b\epsilon) = f(a) + \frac{f'(a)}{1!}(b\epsilon) + \frac{f''(a)}{2!}(b\epsilon)^2 + \cdots = f(a) + f'(a)b\epsilon.$$

This means that to make dual numbers handle elementary functions, the first-order Taylor expansion needs to be implemented. In practice, this amounts to implementing the elementary differentiation rules described in (2.2).

The drawback of implementing AD with dual numbers becomes clear for functions of multiple variables. Let the function f be defined as $f(x,y,z) = x \cdot y + z$. Say we want to know the function value for $(x,y,z) = (2,3,4)$ together with all the derivatives of f. First we evaluate f with x as the only varying parameter, and the rest as constants:

$$f(x,y,z) = (2+1\epsilon)\cdot(3+0\epsilon) + (1+0\epsilon) = 7+3\epsilon.$$

Here, 7 is the function value of f, while 3 is the derivative value of f with respect to x (f_x). To obtain f_y and f_z, we need two more function evaluations with respectively y and z as the varying parameters. This example illustrates the weakness of forward AD implemented with dual numbers – when the function evaluated has n input variables, we need n function evaluations to determine the gradient of the function.

2.1.3 Backward Automatic Differentiation

The main disadvantage with forward AD is when there are many input variables and you want the derivative with respect to all variables. This is where backward AD is a more efficient way of obtaining the derivatives. To explain backward AD, it is easier to first reconsider the approach for forward AD, and explain the method as an extensive use of the chain rule

$$\frac{\partial f}{\partial t} = \frac{\partial}{\partial t} f\big(u_1(t), u_2(t), \ldots\big) = \sum_i \left(\frac{\partial f}{\partial u_i} \cdot \frac{\partial u_i}{\partial t}\right). \tag{2.5}$$

Take $f(x) = x \cdot \exp(2x)$, like in the forward AD example (2.3). We then rewrite the function evaluation as a sequence of elementary functions

$$x, \qquad g_1 = 2x, \qquad g_2 = \exp(g_1), \qquad g_3 = x \cdot g_2, \tag{2.6}$$

where clearly $f(x) = g_3$. If we want the derivative of f with respect to x, we can obtain expressions for all g's by using the chain rule (2.5)

$$\begin{aligned} \frac{\partial x}{\partial x} &= 1, \qquad \frac{\partial g_1}{\partial x} = 2, \\ \frac{\partial g_2}{\partial x} &= \frac{\partial}{\partial g_1}\exp(g_1) \cdot \frac{\partial g_1}{\partial x} = 2\exp(2x). \end{aligned}$$

Lastly, calculating the derivative of g_3 with respect to x in the same way yields the expression for the derivative of f

$$\frac{\partial f}{\partial x} = \frac{\partial g_3}{\partial x} = \frac{\partial x}{\partial x} \cdot g_2 + x \cdot \frac{\partial g_2}{\partial x} = \exp(2x) + x \cdot 2\exp(2x) = (1+2x)\exp(2x).$$

This show how forward AD can be expressed as using the chain rule on a sequence of elementary functions with respect to the independent variables, in this case x. Backward AD also uses the chain rule, but in the opposite direction; it uses it with respect to dependent variables. The chain rule then has the form

$$\frac{\partial s}{\partial u} = \sum_i \left(\frac{\partial f_i}{\partial u} \cdot \frac{\partial s}{\partial f_i}\right), \tag{2.7}$$

for some s to be chosen.

If we again choose $f(x) = x \cdot \exp(2x)$ and expand it using the same sequence of elementary functions

as in (2.6), the expressions from the chain rule (2.7) become

$$\begin{aligned}
\frac{\partial s}{\partial g_3} &= \text{unknown} \\
\frac{\partial s}{\partial g_2} &= \frac{\partial g_3}{\partial g_2} \cdot \frac{\partial s}{\partial g_3} = x \cdot \frac{\partial s}{\partial g_3} \\
\frac{\partial s}{\partial g_1} &= \frac{\partial g_3}{\partial g_1} \cdot \frac{\partial s}{\partial g_3} + \frac{\partial g_2}{\partial g_1} \cdot \frac{\partial s}{\partial g_2} = g_2 \cdot \frac{\partial s}{\partial g_2} \\
\frac{\partial s}{\partial x} &= \frac{\partial g_3}{\partial x} \cdot \frac{\partial s}{\partial g_3} + \frac{\partial g_2}{\partial x} \cdot \frac{\partial s}{\partial g_2} + \frac{\partial g_1}{\partial x} \cdot \frac{\partial s}{\partial g_1} = g_2 \cdot \frac{\partial s}{\partial g_3} + 2 \cdot \frac{\partial s}{\partial g_1}.
\end{aligned}$$

Substituting s with g_3 gives

$$\begin{aligned}
\frac{\partial g_3}{\partial g_3} &= 1 \\
\frac{\partial g_3}{\partial g_2} &= x \\
\frac{\partial g_3}{\partial g_1} &= \exp(2x) \cdot x \\
\frac{\partial g_3}{\partial x} &= \exp(2x) \cdot 1 + 2 \cdot \exp(2x) \cdot x = (1 + 2x)\exp(2x).
\end{aligned}$$

Hence, we obtain the correct derivative f_x. By now you might wonder why make this much effort to obtain the derivative of f compared to just using forward AD. The answer to this comes by looking at a more complex function with multiple input parameters. Let $f(x, y, z) = z\big(\sin(x^2) + yx\big)$ and

$$g_1 = x^2, \quad g_2 = x \cdot y, \quad g_3 = \sin(g_1), \quad g_4 = g_2 + g_3, \quad g_5 = z \cdot g_4. \tag{2.8}$$

Now the derivatives from the chain rule in Equation (2.7) become

$$\begin{aligned}
\frac{\partial s}{\partial g_5} &= \text{unknown} & \frac{\partial s}{\partial g_2} &= \frac{\partial s}{\partial g_4} & \frac{\partial s}{\partial y} &= x \cdot \frac{\partial s}{\partial g_2} \\
\frac{\partial s}{\partial g_4} &= z \cdot \frac{\partial s}{\partial g_5} & \frac{\partial s}{\partial g_1} &= \cos(g_1)\frac{\partial s}{\partial g_3} & \frac{\partial s}{\partial z} &= g_4 \cdot \frac{\partial s}{\partial g_5} \\
\frac{\partial s}{\partial g_3} &= \frac{\partial s}{\partial g_4} & \frac{\partial s}{\partial x} &= 2x \cdot \frac{\partial s}{\partial g_1} + y \cdot \frac{\partial s}{\partial g_2}
\end{aligned}$$

substituting s with g_5 yields

$$\begin{aligned}
\frac{\partial g_5}{\partial g_5} &= 1 & \frac{\partial g_5}{\partial g_2} &= z & \frac{\partial g_5}{\partial y} &= xz \\
\frac{\partial g_5}{\partial g_4} &= z & \frac{\partial g_5}{\partial g_1} &= \cos(x^2) \cdot z & \frac{\partial g_5}{\partial z} &= \sin(x^2) + xy \\
\frac{\partial g_5}{\partial g_3} &= z & \frac{\partial g_5}{\partial x} &= 2x \cdot \cos(x^2) \cdot z + yz
\end{aligned}$$

The calculation of the derivatives together with a dependency graph can be seen in Figure 2.1. This shows that we get all the derivatives of $f(x) = g_5$ with a single function evaluation.

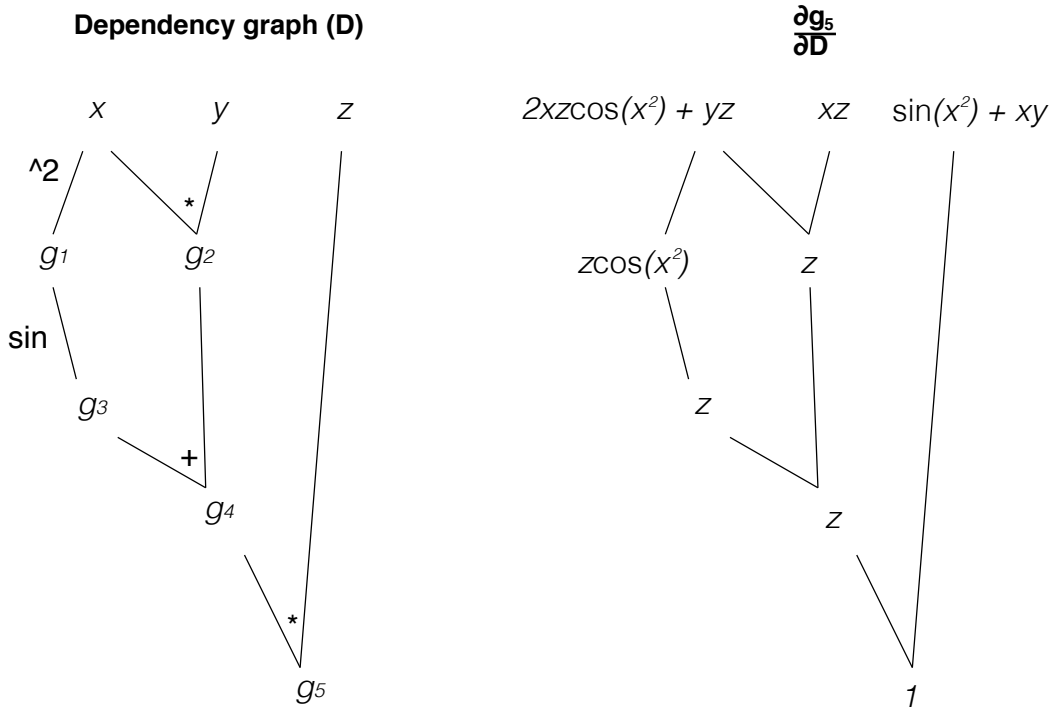


Figure 2.1: Graphs to visualize the process of backward AD. To the left is a dependency graph of the elementary functions in (2.8) and to the right are the derivatives of g_5 with respect to the dependencies given in the dependency graph.

Comparing this to the method of dual numbers from subsection 2.1.2, where we would have to evaluate f three times, this is a big improvement. This illustrates the strength of backward AD – no matter how many input parameters a function has, you only need one function evaluation to get all the derivatives of the function.

The disadvantage of backward AD is that to be able carry along function and derivative values as we did in forward AD, we need to implement the dependency tree shown in Figure 2.1. This makes the implementation of backward AD much harder than for forward AD, and an inefficient implementation of this tree will reduce the advantage of backward AD. Also, if f is a vector-evaluated function $f : \Re^n \to \Re^m$ and not a scalar function, backward AD needs to run m times. Hence, if $n \approx m$, forward AD and backward AD will have approximately the same complexity. This is the main reason why we in the following will focus on implementing forward rather than backward AD.

2.1.4 Forward Automatic Differentiation With Multiple Parameters

When we are dealing with functions with many input parameters and we wish to implement forward AD, there are alternative ways of implementing this, rather than implementing with dual numbers and evaluating the function n times. Neidinger (2010) describes a method that calculates all the derivatives in one function evaluation. To illustrate this method, consider a scalar function $f : \Re^n \to \Re$, that we want to obtain the gradient of. Then, the main idea is that we define what we call our *primary variables*. This is all the variables in the space that we are currently working in. Each primary variable is an AD-variable containing the derivatives of itself with respect to all the other primary variables. Say we have three variables x, y and z, and for any function $f(x, y, z)$ we are interested in

finding the gradient of f, $\nabla f = (f_x, f_y, f_z)^\top$. To achieve this, we define the corresponding AD-variables

$$[x,\ (1,0,0)^\top] \quad , \quad [y,\ (0,1,0)^\top] \quad , \quad [z,\ (0,0,1)^\top].$$

Each primary AD-variable now not only stores its derivative with respect to itself, but also the gradient with respect to all other primary variables. The operators defined in (2.1) and the elementary functions in (2.2) are still valid, except that scalar products are now vector products. As an example, let $f(x,y,z) = xyz$ and $x = 1$, $y = 2$ and $z = 3$, then

$$\begin{aligned} xyz &= [1,\ (1,0,0)^\top] \cdot [2,\ (0,1,0)^\top] \cdot [3,\ (0,0,1)^\top] \\ &= [1 \cdot 2 \cdot 3,\ 2 \cdot 3 \cdot (1,0,0)^\top + 1 \cdot 3 \cdot (0,1,0)^\top + 1 \cdot 2 \cdot (0,0,1)^\top] \\ [f,\ \nabla f] &= [6,\ (6,3,2)^\top]. \end{aligned}$$

This result is equal to the tuple

$$\big(f(x,y,z)\ ,\ \nabla f(x,y,z)\big) = \big(xyz,\ (yz,xz,xy)^\top\big)$$

for the corresponding x, y and z values.

2.1.5 Forward Automatic Differentiation With Vector Functions

For numerical solution of (Partial) Differential Equations, the functions we evaluate as part of the discretization are usually vector functions and not scalar functions. Hence $f : \Re^n \to \Re^m$. Neidinger's method still applies, only that the primary variables are now vectors and instead of containing the gradient of itself with respect to all the other primary variables, we now have the Jacobian. For a function f, the Jacobian with respect to its n primary variables is given as

$$J_f = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{x_n} \end{pmatrix}.$$

The forward AD method described earlier will be similar for a vector function as it was for a scalar function. However, going from scalar functions with multiple parameters to vector functions induce two important differences. The first is that the primary variables need to be initialized with their Jacobians, and not just with a gradient vector. The Jacobian for a primary variable of dimension n is the $n \times n$ identity matrix. The second change is that when evaluating new functions depending on the primary variables, the Jacobians corresponding to the functions will be calculated with matrix multiplication instead of the vector multiplication seen in the previous example. As a simple illustration of the differences, consider the vector function $f = 2 \cdot \mathbf{x} \cdot \mathbf{y}$ for primary variables $\mathbf{x}, \mathbf{y} \in \Re^3$ (we only consider element-wise multiplication). Initialization of the primary variables gives

$$\mathbf{x} = \left[\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \right], \qquad \mathbf{y} = \left[\begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \right]. \tag{2.9}$$

Here, we have decided an order of the variables in the Jacobian, and further on, we need to be consistent with this order. The function value of $\mathbf{x} \cdot \mathbf{y}$ is found by normal element-wise multiplication. The Jacobian of $\mathbf{x} \cdot \mathbf{y}$ is obtained by using the chain rule as defined in (2.1). The difference is now that we have *element-wise* multiplication of a vector and a matrix instead of only scalars or scalars and vectors. Element-wise multiplication corresponds to transforming the vector to an $n \times n$ matrix with

the values on the diagonal. The calculations give

$$\mathbf{x}\cdot\mathbf{y} = \left[\begin{pmatrix}1\\2\\3\end{pmatrix}\cdot\begin{pmatrix}4\\5\\6\end{pmatrix}, \begin{pmatrix}4&0&0\\0&5&0\\0&0&6\end{pmatrix}\cdot\begin{pmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\0&0&1&0&0&0\end{pmatrix}+\begin{pmatrix}1&0&0\\0&2&0\\0&0&3\end{pmatrix}\cdot\begin{pmatrix}0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\end{pmatrix}\right]$$
$$= \left[\begin{pmatrix}4\\10\\18\end{pmatrix}, \begin{pmatrix}4&0&0&1&0&0\\0&5&0&0&2&0\\0&0&6&0&0&3\end{pmatrix}\right].$$

Finally, the expression for the vector function f is found by observing that element-wise multiplication between a scalar and an AD-variable corresponds to multiplying every element in that AD-variable with the scalar. This gives

$$f = \left[\begin{pmatrix}8\\20\\36\end{pmatrix}, \begin{pmatrix}8&0&0&2&0&0\\0&10&0&0&4&0\\0&0&12&0&0&6\end{pmatrix}\right].$$

2.2 Applications of Automatic Differentiation

AD can be used in a wide specter of applications; common for many of them is that we have a vector or scalar function we want to minimize or find the roots of. This section considers some of the applications where AD can be used – from solving simple linear systems to solving the Poisson equation with discrete divergence and gradient operators.

2.2.1 The Newton-Raphson Method

The simplest example for finding roots is for a scalar function f with a scalar input x. Then the Newton–Raphson method

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)},$$

for an initial x_0, will converge to a root of f given that f is sufficiently smooth. With AD, this is quite simple to implement, as you only have to define the function $f(x)$, and then AD finds $f'(x)$ automatically. You can then use the Newton-Raphson method directly. Exactly the same approach can be used to solve nonlinear systems in multiple dimensions. As a simple illustration, let us look at the *linear* system

$$\mathbf{A}\boldsymbol{x} = \mathbf{b}, \tag{2.10}$$

which we can write on residual form such that

$$\boldsymbol{F}(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x} - \boldsymbol{b} = 0.$$

This means that to solve the linear system in Equation (2.10), we need to find the root of $\boldsymbol{F}(\boldsymbol{x})$. This can be done by choosing an initial value $\boldsymbol{x}^0$ and observe that since $\boldsymbol{F}(\boldsymbol{x})$ is linear, the corresponding multivariate Newton-Raphson method will converge in one step. The general form of the multivariate Newton-Raphson method is given by

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n - J_{\boldsymbol{F}}(\boldsymbol{x}^n)^{-1}\boldsymbol{F}(\boldsymbol{x}^n). \tag{2.11}$$

Here, $J_{\boldsymbol{F}}(\boldsymbol{x}^n)^{-1}$ is the inverse of the Jacobian of $\boldsymbol{F}$ at $\boldsymbol{x}^n$.

2.2.2 Solving the Poisson Equation

For a simple linear system like Equation (2.10) it may seem a bit forced and unnecessary to make the effort of using AD to solve for $\boldsymbol{x}$. We could just as well have used some built in linear solver. But for applications to the numerical solution of PDEs, this approach greatly simplifies the process of (linearizing and) assembling the linear systems that appear when you solve the (non)linear system of discretized equations. Indeed, using Equation (2.11) with AD, all you have to do is implement the discretized equations on residual form. As a preface to introducing how we will do this, we consider the Poisson equation

$$-\nabla(\mathbf{K}\nabla u) = q, \tag{2.12}$$

where $\mathbf{K}$ is a spatially variable coefficient, and we want to find u on the domain $\Omega \in \Re^d$. Numerically, this can be done by using a finite volume method. This approach is based on applying conservation laws inside the domain. By dividing the domain into a grid of smaller cells, Ω_i, we can instead of looking at the Poisson equation in differential form, integrate it over each cell such that

$$\int_{\partial\Omega_i} -\mathbf{K}\nabla u \cdot \mathbf{n}\,\mathrm{ds} = \int_{\Omega_i} q\,\mathrm{dA}. \tag{2.13}$$

Here, $\mathbf{n}$ is the unit normal to the cell Ω_i, so Equation (2.13) describes the conservation of mass in the cell Ω_i, where total flux in and out of the boundary of Ω_i is equal to the total accumulation from source and sink terms inside Ω_i. For simplicity, we define $\mathbf{v} = -\mathbf{K}\nabla u$ as the flux. As a simple example to begin with, we will consider Figure 2.2, which shows two cells Ω_i and Ω_k. The average values of u inside the two cells are u_i and u_k, and the interface, or facet, between the cells is defined as $\Gamma_{i,k}$.

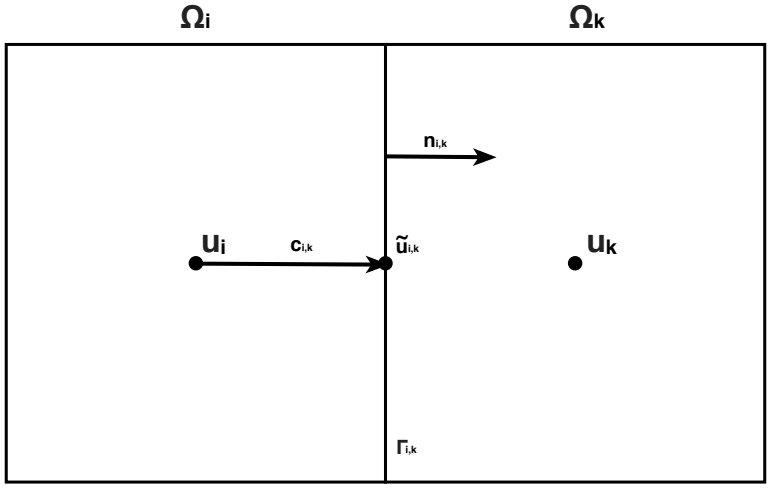


Figure 2.2: Figure of two adjacent cells Ω_i and Ω_k. The average value of the cell is given by u_i. The boundary between the cells is $\Gamma_{i,k}$ with value at the centre equal $\tilde{u}_{i,k}$ and outward normal vector $n_{i,k}$

Now, the flux through the common interface $\Gamma_{i,k}$ can be computed by

$$v_{i,k} = \int_{\Gamma_{i,k}} \mathbf{v} \cdot \mathbf{n}_{i,k}\mathrm{ds}. \tag{2.14}$$

The integral in (2.14) can be approximated by the midpoint rule with $\tilde{\mathbf{v}}_{i,k}$ as the flux on the midpoint

of $\Gamma_{i,k}$. If we let $L_{i,k}$ be the length of $\Gamma_{i,k}$, then

$$v_{i,k} \approx L_{i,k}\tilde{\mathbf{v}}_{i,k} \cdot \mathbf{n}_{i,k} = -L_{i,k}\mathbf{K}\nabla\tilde{u}_{i,k} \cdot \mathbf{n}_{i,k}.$$

Here, $\tilde{u}_{i,k}$ is the value of u at the centre of the facet $\Gamma_{i,k}$. The problem we now face is that in the finite volume method, we only know the average value of u over each cell. If we use a first-order reconstruction, the reconstructed value at the center of cell Ω_i coincides with the average value u_i. Assuming the underlying function is sufficiently smooth, we can then use a finite-difference method to approximate the gradient of u on $\Gamma_{i,k}$, expressed in terms of the value u_i at the cell center and the value $\tilde{u}_{i,k}$ at the midpoint of the facet,

$$v_{i,k} \approx L_{i,k}\mathbf{K}_i \frac{(\tilde{u}_{i,k} - u_i)\mathbf{c}_{i,k}}{|\mathbf{c}_{i,k}|^2} \cdot \mathbf{n}_{i,k}.$$

Here, $\mathbf{c}_{i,k}$ is the vector from u_i to $\tilde{u}_{i,k}$ as seen in Figure 2.2. For brevity, we collect all the known quantities into a scalar quantity, which we call the transmissibility,

$$T_{i,k} = L_{i,k}\mathbf{K}_i \frac{\mathbf{c}_{i,k}}{|\mathbf{c}_{i,k}|^2} \cdot \mathbf{n}_{i,k}. \tag{2.15}$$

Because we know that the amount of flux from cell Ω_i to Ω_k must be the same as from Ω_k to Ω_i, only with opposite sign, we have the relation $v_{i,k} = -v_{k,i}$. In most cases, we will also have continuity across the interface, so that $\tilde{u}_{i,k} = \tilde{u}_{k,i}$. Hence, we have the relation

$$v_{i,k} = T_{i,k}(\tilde{u}_{i,k} - u_i) \qquad -v_{i,k} = T_{k,i}(\tilde{u}_{i,k} - u_k).$$

By subtracting the two equations for $v_{i,k}$ and moving $T_{i,k}$ and $T_{k,i}$ to the other side

$$\begin{aligned}(T_{i,k}^{-1} + T_{k,i}^{-1})v_{i,k} &= (\tilde{u}_{i,k} - u_i) - (\tilde{u}_{i,k} - u_k) \\ v_{i,k} &= (T_{i,k}^{-1} + T_{k,i}^{-1})^{-1}(u_k - u_i) = T_{ik}(u_k - u_i)\end{aligned} \tag{2.16}$$

we manage to eliminate $\tilde{u}_{i,k}$ and get a computable expression for the gradient of u. This is called the two-point flux-approximation (TPFA) (Lie, 2019). Now that we have an approximation of the flux through the interface between Ω_i and Ω_k, we get that Equation (2.14) can be approximated by

$$\sum_k T_{i,k}(u_k - u_i) = q_i, \qquad \forall\Omega_i \in \Omega, \tag{2.17}$$

where q_i is the integrated accumulation over cell Ω_i. Now, we can get a linear system of the form **Au** = **b**, which on residual form becomes **F**(**u**) = **Au** - **b** = 0 where

$$\mathbf{A}_{i,j} = \begin{cases} \sum_k T_{ik} & \text{if } j = i \\ -T_{ij} & \text{if } j \neq i. \end{cases}$$

This means we can solve the Poisson Equation (2.12), using the scheme explained in (2.11) and by having u as an AD-variable. For this simple Poisson equation, we still only end up with a linear system of equations that we may as well solve without AD. The only benefit is that we never need to form the matrix **A** explicitly, which can be bit complicated for more complex grids. For nonlinear PDEs, we would also need to linearize the local discrete equations, and the construction of the matrix **A** generally becomes more tricky. The ease of using AD becomes clearer when we combine it with discrete differentiation operators that we will define in the next subsection.

2.2.3 Discrete Differentiation Operators

To show the real elegance of using AD to solve PDEs, we want to create a framework in which we have defined discrete divergence and gradient operators such that we can write the discrete equations we want to solve on a similar form as in the continuous case. We also want to be able to do this no matter how complex and unstructured our grid is.

Instead of the simple two-cell grid we used in Figure 2.2, we now consider a a general polygonal grid. Figure 2.3 illustrates an example, in which all cells are quadrilaterals. To define the discrete divergence and gradient operators, we need some information about the topology of the grid. The grid can be described in terms of three types of objects: cells, facets and vertices. The cells are each $\Omega_i \subset \Omega$. In our two-dimensional case, the facets are simply the lines that delimit each cell, and the vertices are the endpoints of each facet. In addition, we introduce nodes. In the case demonstrated by Figure 2.2, we had two nodes, u_i and u_k, and for the finite-volume method they are the average value of u on the corresponding cell. Each cell and facet has physical properties like area or length, and centroid or centre. Each facet also has a normal vector.

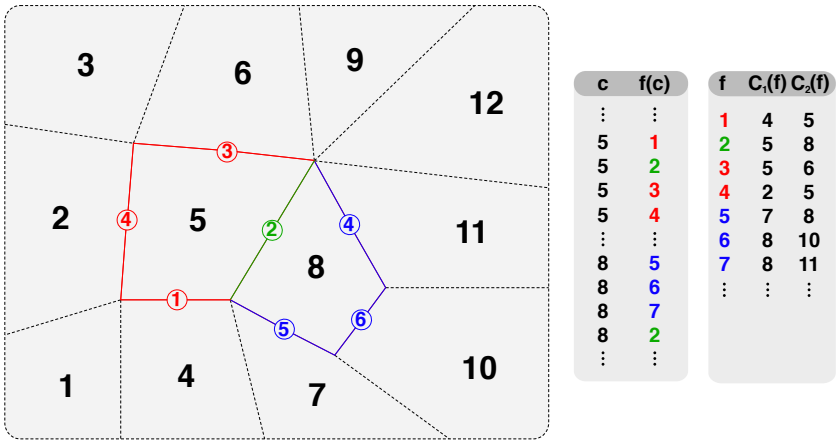


Figure 2.3: Figure of a general polygonal grid with the mapping cell to facets and facet to cells.

Figure 2.3 shows how we can introduce two different mappings that explain the relation between the cells and the facets. The mappings for cells 5 and 8 are written out. The first relation, $F(c)$, is the mapping from cells c to their delimiting facets f. The second mapping, $C_i(f)$ for $i = 1,2$, is a mapping from a facet f to the two cells C_1 and C_2 that share this facet. All these properties will be used to create the discrete divergence and gradient operators.

We now have all the physical properties of the grid we used to attain the formulae in Equation (2.17). From these, we want to create discrete divergence and gradient operators that correspond to the continuous equivalents for this grid. Consider the Poisson Equation (2.12) for the function u. Then the discrete gradient operator for a facet f is defined as

$$\texttt{dGrad}(u)[f] = u[C_2(f)] - u[C_1(f)], \tag{2.18}$$

where $u[C_i(f)]$ is the value of u at the cell corresponding to $C_i(f)$. For the divergence operator, we remember the expression we found for the flux through a facet in Equation (2.16). Let $v_{i,k} = v[f]$,

where f is the facet between cell i and cell k. Since the divergence in a cell is the same as the sum of flux leaving and entering the cell, the discrete divergence operator for cell c is defined as

$$\texttt{dDiv}(\mathbf{v})[c] = \sum_{f \in f(c)} \text{sgn}(f)\, v[f]$$

where the function $\text{sgn}(f)$ is defined as

$$\text{sgn}(f) = \begin{cases} 1 & \text{if } c \in C_1(f) \\ -1 & \text{if } c \in C_2(f). \end{cases}$$

The discrete operators `dDiv` and `dGrad` can be represented in terms of sparse matrices that are simple to form from the mappings F and C_1, C_2; see Lie (2019) for more details.

The extra `d` in front of the names stands for discrete and is included so that we later avoid name collision with Julia's built in `div` function. Now we can create discrete divergence and gradient operators, only based on the topology of the grid, so that the discrete Poisson equations we want to solve can be written very similar to the continuous case

$$-\nabla(\mathbf{K}\nabla u) - q = 0 \qquad \longleftrightarrow \qquad \mathbf{F}(\mathbf{u}) = \texttt{dDiv}(\mathbf{T}\,\texttt{dGrad}(\mathbf{u})) - \mathbf{q} = 0.$$

Here, $\mathbf{T}$ is the transmissibility defined in (2.15). The notation for the discrete equations is clearly similar to the continuous case, and we can actually read the discrete expression and directly understand what equation we are trying to solve. For this simple Poisson equation, we will still have a linear system and we would not necessarily need to use AD to solve it. But for more complex problems, we can derive the discrete divergence and gradient operators in the same approach for any type of grid. Although the system then becomes non-linear, it will be easy to solve using AD and the Newton-Raphson method. An example of this can be seen in chapter 5.

Chapter 3

Julia

Julia is a new programming language that was created by Jeff Bezanson, Alan Edelman, Stefan Karpinski and Viral B. Shah at Massachusetts Institute of Technology (MIT) (The Julia Lab, n.d.). The language was created in 2009, but was first released publicly in 2012. In 2012 Bezanson et al. (2012) said in a blog post:

> We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled. (Did we mention it should be as fast as C?)

3.1 Characteristics of Programming languages

To understand why the creators of Julia wanted to build this new programming language, we need to have a closer look at what type of programming languages are out there already, and what separates them. All programming languages that are used in numerical applications are written in a high level language. This is a language that is easily readable for a human. The code that is written is called source code. This source code needs to be translated for the machine to understand it. The translated code is called machine code. How this translation happens is a big part of what gives the different languages different capabilities. I will not give a full review of the subject, as it is too comprehensive for this thesis, but try to scratch the surface such that the different capabilities of the different languages becomes clear.

3.1.1 Type Checking

Firstly, before the translation happens, the program needs to be type checked. This consist of verifying that the variables are not set to unsupported types. For example, a simple array consisting of integers cannot suddenly at a point in the program hold a string in one of its elements. There are two ways of type checking, the first is called static and the second is called dynamic. In a static language

each variable needs to have a defined type beforehand and they are checked before the program is executed. We say that type checking happens before run-time. In addition a variable that is declared as one type cannot change type in its scope. The scope of a variable can quickly be explained as the part of the program where the variable is functioning. In dynamic languages the types of the variables do not need to be known before run-time. Type correctness is checked at run-time, or in other words, continuously as the code is executed. A variable can also change type in its scope. Static languages are faster in the execution since it does not need to check for types as it is already done. You will, however, have to wait for the type checking to finish before the program can be executed. Dynamic languages are slower but the continuous type checking enables language designs that optimizes the coding process such that you can implement your program with less code. The first and most obvious difference from static languages is that you do not have to define what types each variable is. The type checking will find this out on its own at run-time. In addition it opens up for a feature called metaprogramming. Metaprogramming is programs that can take in other code as input and modify it such that implementing your code can be done more efficiently and faster. More specifically how this can be used will be discussed closer in section 3.3.

3.1.2 Compiled and Interpreted Languages

The next translation difference between the languages is whether it is compiled or interpreted. A compiled language translates the source code to machine code before run-time while an interpreted language translates the code at run-time. To visualize how run-time compilation works, you can say that for each line of code it is first translated and then executed. Compiled languages are faster in the execution than an interpreted language. The reason for this is that when it translates all the code beforehand it can optimize the code that will be executed. You will, however, have to wait for the compiler to finish translating before the program can be executed. Hence, using an interpreted language can be faster when developing programs, as you do not have to recompile your entire program every time you have made a small change. There is also a third way to translate source code into machine code, called just-in-time compilation (JIT compilation). JIT compiled languages is a combination of compiled and interpreted languages. It compiles blocks of the source code such that it can do optimizations like compiled code, but at the same time it does not need to compile the whole source code before it executes the program. In this way it behaves like an interpreted language, but with the speed advantages of compiled languages. In most cases however, it will be slower than regular compiled languages. Disadvantages for JIT compiled languages is extra memory usage and that writing a JIT compiler is more difficult than other compilers. The latter does not affect the end user of the language.

3.1.3 Languages for Numerical applications

For numerical applications we can separate the most commonly used languages into two groups. The first group is static and compiled languages like C and C++. These are languages that execute very fast but the time it takes to make the programs are longer. The second group are dynamic and interpreted languages like the well known MATLAB and Python (in 2015 MATLAB introduced a new execution engine that uses JIT compilation *(Shure, 2016)*. However, MATLAB is still a dynamic language that function like a fully interpreted language). These languages are easier to use if you want to create numerical simulations, but they execute slower than C and C++. Julia is a dynamic, but (JIT)compiled language. As the creators said in the blog post from 2012: Julia is supposed to have "(...) familiar mathematical notation like Matlab", be "(...) as powerful for linear algebra as Matlab" and at the same time be as fast as C. Summed up their goal has been to make a language that is as easy to use as an interpreted and dynamic language, but with the speed of a compiled and static language. Based on

this description, Julia seems to be the perfect language for numerical simulations.

3.1.4 Source Code Example

So far in this chapter, I have discussed what separates the languages concerning type checking and how the source code is translated to machine code. These differences will in return affect the syntax of the languages and how we can write the source code. Below I have an example illustrating different implementations of an example in Julia, MATLAB and C++. The example consist of writing a program that creates two random vectors of length n, element-wise multiply them together and print the resulting vector in a readable way. This example is first implemented in Julia, where the random vector is created using `rand(n)`, the element-wise multiplication is performed with the `.*` operator and the result is printed with the `println(...)` function:

```
## Code example from Julia
n = 5
v = rand(n)
w = rand(n)
dotProduct = v .* w
println("Dot product of two random vectors:")
println("$dot_product")
```

Very similar to the Julia implementation we have the same program in MATLAB:

```
%% Code example from MATLAB
n = 5;
v = rand(n,1);
w = rand(n,1);
dotProduct = v .* w;
fprintf('Dot product of two random vectors:\n');
disp(dotProduct')
```

The implementations only have a couple of small syntax differences, and they both implement the example efficiently with few lines. C++ on the other side is not created specifically for mathematical programming and without importing any external libraries (except from `iostream`, which is needed to print the result), the implementation of the example becomes much more comprehensive:

```
// Code example from C++
#include <iostream>
int main() {
    int n = 5; int v[n]; int w[n]; int dotProduct[n];
    for (int i = 0; i<n; i++){
        v[i] = rand();
        w[i] = rand();
        dotProduct[i] = v[i] * w[i];
    }
    std::cout<<"Dot product of random vectors:"<<std::endl;
    for (int i = 0; i<n; i++){
        if (i == n-1){
            std::cout<<dotProduct[i]<<std::endl;
        } else{
            std::cout<<dotProduct[i]<<", ";
        }
    }return 0;
}
```

The first obvious difference is that C++ is not a scripting language like MATLAB and Julia. This means that code we want to execute, needs to be inside a function. This could contribute making the implementation of new code more tedious in C++, than in MATLAB and Julia. From the declaration of the variables, it is clear that C++ is a static language, where each variable needs to have a predefined type. Lastly there are no built in functions to create a random vector, element-wise multiply vectors nor printing vectors. These operations needs to be implemented and shows how C++ is a lower-level language than MATLAB and Julia.

You could argue that this comparison is unfair for C++, as you would either import libraries that have functions to create random vectors, do element-wise multiplication and print vectors, or you would overload operators to do this. By either using such libraries or implemented operators, we would be able to implement the example more compact like in Julia and MATLAB. However, the implementations above are not built in, and they need to be implemented. This can either be done by importing an external library that has implemented this already or by overloading the operators yourself. Hence, the example illustrates some of the differences between two languages that are created with mathematical programming in mind and a language that is not.

3.2 History of Julia

The process of creating a new programming language is long. In 2009 the creators began the project of creating Julia and in the blog post *(Bezanson et al., 2012)* from 2012 they said "It's not complete, but it's time for a 1.0 release — the language we've created is called Julia". In a new blog post from August 8. 2018 *(Julia Community, 2018)*, where they released the actual version 1.0 they admitted that they had jumped the gun a little with the mentioning of v1.0 in 2012, as it took more than six years before it actually happened. But after almost ten years of development v1.0 of the Julia language was released. The major consequence of a 1.0 release is that from this version and on, they guarantee backward compatibility. When they did not guarantee backward compatibility, code that worked on version, 0.1, 0.2, etc., would not necessarily run on newer versions of Julia. But from v1.0, all code that run on this version, will also run on future releases. This was a huge milestone for the Julia language.

A consequence of the non backward compatibility is that when you search the internet for help in Julia, currently less than one year after the v1.0 release, you could end up finding solutions to your problem that no longer works. Hence, as of now you need to be careful and check the date of the answers, and keep in mind, if it is from before v1.0, it might no longer work. This can at some times be frustrating, especially when you try to learn the language. However, now that there is backward compatibility for future releases, as time goes by, this problem will disappear as pre-v1.0 answers will eventually drown by post v1.0 answers.

Since Julia is an open source and free program to use, one of its strengths is that developers can contribute by creating useful programs that they share with all the other users. One example of this is the Integrated Development Environment (IDE), Juno (n.d.). Juno is a program to help writing Julia code easier and is created by mainly two developers, Sebastian Pfitzner and Mike J. Innes. Juno gives the coder an environment where it is easy to run your Julia code, you get auto completion when writing your code, it has built in plotting panes to visualize results, and much more. The program is open source and free to use, and if you look at Juno's Github page (n.d.), there are many other contributors to the IDE other than Pfitzner and Innes. These are developers that have either found an error in the existing code or made a feature they wanted for the IDE. This shows the strength of the code being open source – the community can contribute to the code database. This will help finishing improvements and error corrections to the code database quicker. This is why v1.0 of Julia was such a milestone, because as soon as Julia has guaranteed backward compatibility, there is a lot easier for

developers to justify spending time to develop programs for Julia. This is because the developers now know that the work they put in writing Julia code, will not be in vain, as the code will work on all future releases of Julia.

3.3 Metaprogramming in Julia

In the last year there have been published numerous projects that makes coding in Julia easier. A lot of these projects exploit that Julia is a dynamic language and that it has metaprogramming. In the blog post from Bezanson et al. (2012) the creators says that Julia shall contain true macros like the programming language Lisp *(Lisp language, n.d.)*. Macros are functions that are using metaprogramming to modify your existing code to give extra functionality. The macro functions in Julia are easily recognized by an "@" in front of the function name. To use a macro function on any function, you simply call the macro function before the original function call. I will now show four examples of macro functions in Julia that are very useful when creating numerical simulations.

3.3.1 Profiling

Profiling is an effective method to obtain overview of where bottlenecks lie in a code when trying to optimize its performance. The method consists of taking snapshots of the code with small time intervals and for each snapshot we register what function we are at and all the functions that have been called to get to this function. The latter is called the stack trace. By counting how many times a function is in the stack trace, we get an overview of how much time we spend in each function and where they are called from. Since we only register the number of times we have observed that we are in each function, this will not give a perfect picture on how much time we spend on all function. We even risk not registering all functions we use, but since the time interval between the snapshots are small (e.g. every tenth microsecond), a function that is not registered will not be interesting to optimize as it is already very fast.

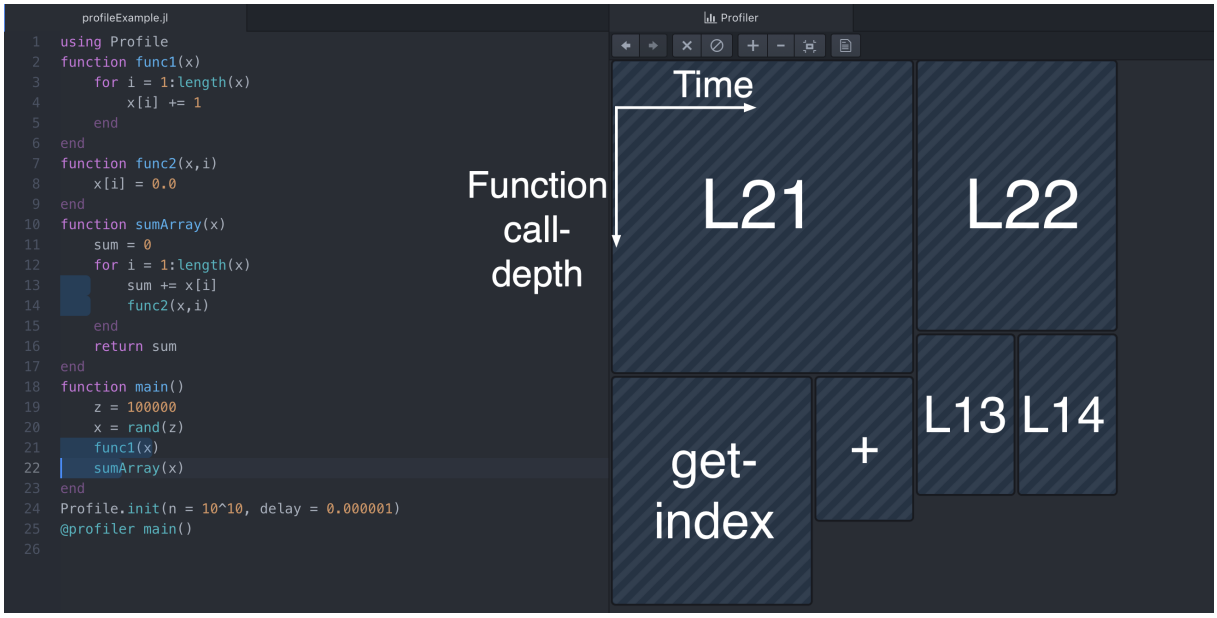


Figure 3.1: Interface in Juno of the `@profiler` macro tool for a simple test case. The left hand side shows the code profiled and the right hand side shows the result visualized. Since the visualization is interactive I have added extra text to show which block represents which line in the code.

Figure 3.1 shows the interface from the Juno IDE using the `@profiler` macro function *(Juno profiler, n.d.)* for a simple test case. The left hand side of Figure 3.1 shows a test case with a `main()` function that creates a random variable `x` of length `z`. It then calls the `func1(x)` function that adds 1.0 to each element of `x`. Lastly it calls `sumArray(x)` that sums up all the values of `x` and uses `func2(x,i)` to set all values of x to zero. `@profiler` is a specific Juno macro that uses the built in `@profile` in Julia *(Profile.jl, n.d.)* to profile the `main()` function, but in addition it creates the right hand side of Figure 3.1 which shows the profiling result. Since the Profiler pane is interactive, I have added extra white text for visualization. Each block represents one specific operation inside the code and the wider the block is, the more time is spent in that operation. The vertical axis represents how deep into the stack trace the operation is. For Figure 3.1 line 21(L21) and 22 (L22) are at top of the stack trace. We can also see that we spend more time in `func1(x)` than `sumArray(x)`. The time spent on each line can also be seen by the blue bars on each line. If we say L21 and L22 is at depth 1, then line 13 and 14 is at depth 2. The `getindex` and + blocks are built in functions in Julia that are called from line 4 in `func1(x)`. These functions are deeper than line 13 and 14 at depth 3. Technically it should have been an extra block underneath line 21 that represents line 4 to get the full stack trace. The only reason I have found why this is not part of the visualization is that the `@profiler` tool is still a work in progress, and that it does not work perfectly just yet. However, this tool is super efficient to find bottlenecks in your code, especially when you have a larger program than the simple example in Figure 3.1.

3.3.2 Debugger

Many IDEs for different languages have the ability to debug code. This is a tool to step into your functions and execute them step by step to reveal error and bugs in your code. Recently, on March 19th, Holy et al. (2019) published a debugger for Julia. This can be used directly in the terminal using the `@enter` macro function or as a built-in debugger in Juno with `Juno.@enter`. Figure 3.2 shows the interface of Juno's debugger for a simple test function. With this you have the opportunity to:

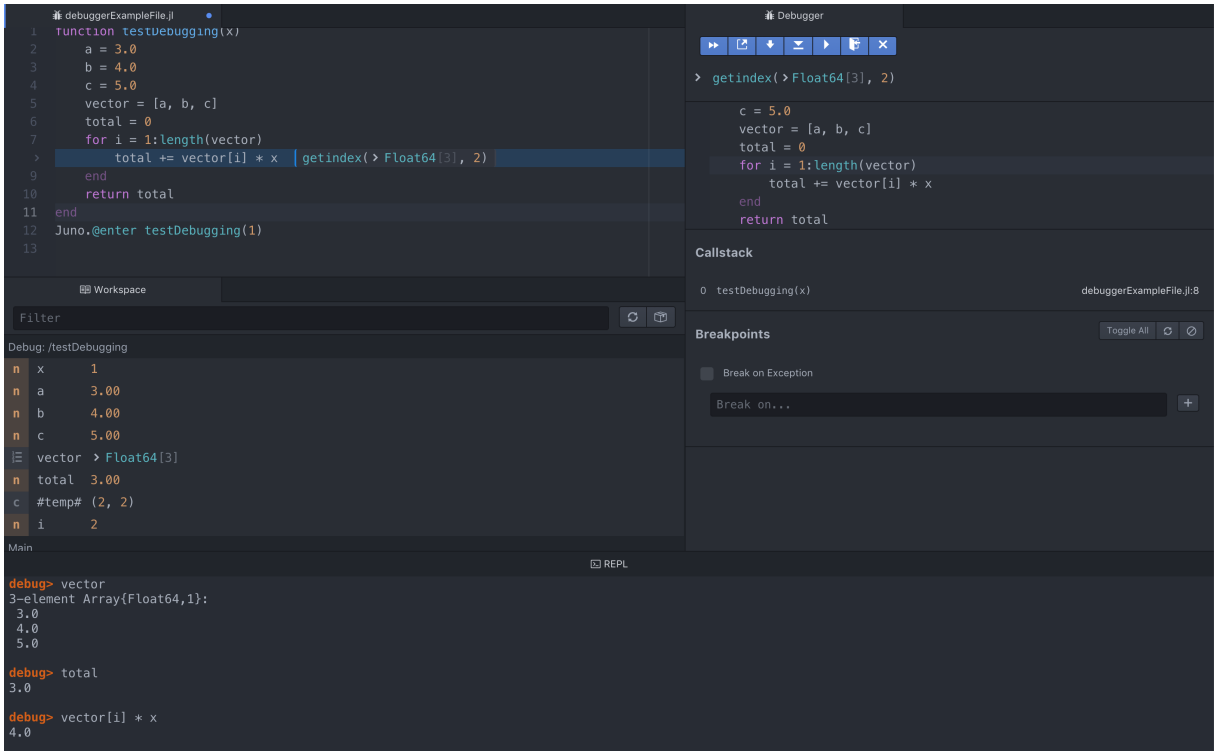


Figure 3.2: Juno's interface in debugging mode.

- Execute the code, step by step.
- See the next function or operation that will be executed.
- See the values of the variables in the workspace.
- Work with the variables from the workspace at their current state.
- See the current call stack (or the stack trace) which shows how you got to this part of the code.
- Set breakpoints such that you can execute your code until the breakpoint.
- Make corrections to the code that will impact the execution of the code immediately.

The opportunity to use a debugger to find bugs and correct your code can be very valuable and in some scenarios save the programmer a lot of time.

3.3.3 Benchmarking

When writing code for numerical simulations, or any other code for that matter, it is interesting to benchmark the simulation. Benchmark will in this thesis refer to testing the time spent to execute the function and the memory usage and compare it to other alternative implementations. Julia has a built in `@time` macro function *(@time docs, n.d.)* that returns the time spent to execute the function together with the number of memory allocations and total memory used. This is a great tool, but especially time can be difficult to measure accurately. This is because there are numerous of other processes happening on your computer that there is impossible to control. Examples of this can be that you might get an email while you execute your code or the computer decides to take backup of some of your files. All these uncontrollable factors can affect the time spent to execute a simulation.

An option to reduce possible sources of errors is to run the benchmark multiple times and use the average time spent. However, then you have to write this extra code every time you want to benchmark a function. *BenchmarkTools (n.d.)* is a library that gives you a macro function called `@btime` that does exactly this so that you do not have to write repetitive code. It uses `@time` multiple times to get a result that is less prone to sources of errors. The primary macro function from *BenchmarkTools* is however `@benchmark` which returns minimum, maximum, mean and average time, together with memory allocation and usage, and information on how many samples it has taken. All this is easy to use because of Julia's metaprogramming. It makes it quick and safe to benchmark all types of simulations and codes, without repeating your source code.

3.3.4 Parallel Computing

Parallel computing is wide topic that can be performed at many levels. However, the main idea is to split a task into smaller problems and execute them simultaneously. New computers are delivered with multi-core processors, giving them the ability to perform multiple tasks simultaneously. The speedup for programs using parallel computing compared to regular linear computing can be up to the number of cores, and in best case scenarios, the speedup can actually be even more than the number of cores. In these cases the extra speedup is an effect caused by the memory allocation becoming easier when the task executed is split up into smaller problems. However, for the computer to be able to perform parallel computing, the source code needs to be written such that it supports parallelization. This parallelization of the code is important, because even though the possible computational gain from parallelization can be huge, a bad written parallel code can execute slower

than the linear equivalent.

In Julia, parallel computing is implemented using metaprogramming *(Parallel computing Julia docs, n.d.)*. There are different types of parallel programming that are implemented, but I will focus on what is called multi-threading. Multi-threading means that we have different threads that perform different tasks simultaneously. By using the `Threads.@thread` macro in front of a for-loop, Julia will automatically split the different iterations between all the available processing cores. The Julia documentation has a good example on how this works. The example consist of initializing a vector, `a`, containing only zeros and loop through it. The iterations are happening in parallel and each element in `a` becomes the identification number of the core that performed the iteration. The example code can be seen below:

```
a = zeros(8)
Threads.@threads for i = 1:8
    a[i] = Threads.threadid()
end
println(a)
## [1.0, 1.0, 2.0, 2.0, 3.0, 3.0, 4.0, 4.0]
```

In the last line, the output of `println(a)` is commented. We can see that the example has been executed with four available cores, and that each core has performed two iterations. This use of metaprogramming to parallelize code is very elegant and makes parallelization of code very easy. However, as warned in the Julia documentation, the implementation of multi-threading is, at the moment of writing this thesis, experimental. This means that it can work for some examples and for others it will crash and not work. For the simple example displayed above, it will work, but for more extensive examples, the program will crash. The use of multi-threading has been tried applied to the methods described in this thesis, but unfortunately the multi-thread macro functions are too unstable to handle these cases. However, it is an interesting feature to look into when the development has come further and the macro functions are more stable.

Subsection 3.3.1, 3.3.2, 3.3.3 and 3.3.4 are all great examples of how powerful tools, that help developers code, are being created less than a year after v1.0 of Julia was released. This community, together with the ideas of a very fast language, makes Julia a very interesting language to try out and to see what is possible to achieve concerning easiness of code writing and execution performance.

Chapter 4

Implementation

This chapter will give implementation-specific details on how to implement AD in Julia and perform benchmarks comparing AD libraries in both Julia and MATLAB.

4.1 Automatic Differentiation in Julia

As discussed in chapter 3, Julia seems to be the perfect language for numerical applications and it would be interesting to see how it performs compared to MATLAB. For AD there are already some third-party packages in Julia that can be used. Most of them are backward AD-packages designed for machine learning. Two examples are AutoGrad (Yuret, 2016) and Zygote (Innes, 2018). The reason why AD-packages for machine learning are based on backward AD is that machine learning, without going to deep into the subject, largely amounts to the minimization of functions with a large number of input parameters, but with only one output parameter. As discussed in subsection 2.1.3, backward AD is much more efficient than forward AD in these types of evaluations. For numerical applications there is usually many input parameters and output parameters. Hence there is no clear advantage of using backward AD compared to forward AD.

There is one package called *ForwardDiff* (Revels et al., 2016) being developed by the Julia community that uses forward AD. ForwardDiff relies on dual numbers as explained in subsection 2.1.2 extended to multiple dimensions. This is called multidimensional dual numbers and a vector $\mathbf{x}$ of length n is represented as

$$\mathbf{x} = \begin{bmatrix} x_1 + \epsilon_1 \\ \vdots \\ x_i + \epsilon_i \\ \vdots \\ x_n + \epsilon_n \end{bmatrix}.$$

This package works very well for some applications, but especially for numerical solution of PDEs it has some limitations that are not ideal. Examples of this are:

- The function to differentiate can only accept a single argument. This is possible to work around; if you have vector function f with input parameters $x, y, z \in \Re^n$, you can merge them into one vector of length $3n$ and then obtain the Jacobian. Although this works and you get the correct

answer, it is not optimal as you would have to make local workarounds to make the code work, causing unreadable code.

- The function we want to differentiate must be on the form of a generic Julia function, such as: $g(x) = 3x * x$. Here $x * x$ symbolize element-wise multiplication. This means that if we have a function like $h(x) = 3x * x + \text{sum}(x)$, where all elements in $g(x)$ are added with the sum of all elements in x, it will not be possible to use *ForwardDiff* to obtain the Jacobian. This limitation also prevents us from having a function that we evaluate for all points, and then add boundary conditions for some points later. This is a feature that is essential in PDE-based simulators.

- The Jacobian calculated by *ForwardDiff* is a full matrix. In some cases the Jacobian is dense anyway, so this will not have any major adverse effects, but in many numerical applications, and particularly in numerical solution of PDEs, the Jacobian will be sparse. By representing a sparse matrix on a full matrix format, a lot of potential computation efficiency is lost.

4.2 Implementation of Automatic Differentiation

When it comes to efficient implementation of AD, there are two factors to consider. Firstly, it must be easy and intuitive to use. Secondly, the code must be efficient as it will be used in computational demanding calculations. A convenient way to store the AD-variables in Julia is to make a structured array (`struct`) that has two member variables, `val` and `jac`, that store respectively the value and the corresponding Jacobian:

```
struct AD
    val
    jac
end
```

The `val` variable is a vector with elements of type `Float64`. If the AD-variable is only a scalar, the implementation can either stick to `val` being a vector of length one, or it can be a scalar. Always representing `val` as a vector is most consistent and can avoid problems that may occur when we do not know what types the AD struct contains at different times. When it comes to the Jacobian, there are multiple ways of storing the matrix. Depending on the application, how much, and what type of manipulation of the matrix you are going to do, the choice is based on efficiency and convenience. I will describe two different methods on how to store the Jacobian. Both implementations are inspired by two different implementations in MRST (Lie, 2019).

4.2.1 `ForwardAutoDiff` (`FAD`)

In the first implementation, the Jacobian, `jac`, is represented as a list of sub-blocks. Each element in the list is a sparse matrix that represent the Jacobian w.r.t. a single primary variable. This implementation gives the freedom to easily work with the sub-blocks of the Jacobian that correspond to a single primary variable. Before introducing the second implementation of the Jacobian, I will continue to explain the implementation of what I have called `ForwardAutoDiff`(`FAD`), which is defined with the following struct:

```
struct FAD
    val::Vector{Float64}
    jac::Vector{SparseMatrixCSC{Float64,Int}}
```

```
end
```

To obtain a working AD library, we need to implement operators for the `FAD` data structure. The importance of how you implement the AD operators and elementary functions can be expressed in a short example: Assume you have two `FAD` variables x and y and that you want to compute the function $f(x,y) = y + \exp(2xy)$. If the implementation is based on making new functions that take in `FAD`-variables as input parameters, the evaluation of f will look something like this:

$$f = \texttt{FADplus}(y,\texttt{FADexp}(\texttt{FADtimes}(2,\texttt{FADtimes}(x,y)))).$$

This is clearly not a suitable way to implement AD as it quickly becomes difficult to see what type of expression it is. If you did not know what type of function f is, it would take you quite some time to figure it out. And more importantly, the possibility for human error becomes very large when you have to write unreadable code like this. This approach should be avoided.

Neidinger (2010) and Lie (2019) suggest a much more elegant implementation, in which one, instead of making new functions that take in `FAD`-variables as parameters, overloads the standard operators $(+,-,*,/)$ and the elementary functions (exp, sin, log, etc.). This is where the elegance of having a custom `FAD` struct appears. In Julia, we can use *multiple dispatch* to call our implementation of standard operators and elementary functions when they are used on `FAD` structs. A quick explanation of multiple dispatch that satisfies our needs is that the compiler at run-time understands what types are given as input for either an operator or a function and chooses the correct method based on this. To demonstrate, the following function

```
import Base: +
function +(A::FAD, B::FAD)
    return FAD(A.val + B.val, broadcast(+, A.jac, B.jac))
end
```

overloads the + operator. Here, we import the + operator from `Base` (which is where the standard functions in Julia lie) and overload it for `FAD` variables. For brevity, I have removed checks and edge cases and only left the method for `FAD` variables with equal length. The broadcast function adds each Jacobian for each primary variable together. This broadcast function can also be called by adding a dot `(.)` in front of the function or operator. From now on this approach will be used instead of `broadcast(...)`. The implementation of the + operator above is only used when there are `FAD` variables on both sides of the operator. Hence, if $z = x + y$ is computed for $x = 1$ and $y = 3$, Julia understands that it is not the definition above, but the normal addition for integers it should use. But if $x, y =$ `initialize_FAD`$(1,3)$ is declared, so that x and y both are `FAD` variables, then Julia's multiple dispatch will understand that the new definition of the + operator should be used on the expression $z = x + y$. What we need to remember is that if I now write $z = x + 3$, with x as an `FAD` variable, Julia will deploy an error message. This is because we also have to implement

```
import Base: +
function +(A::FAD, B::Number)
    return FAD(A.val .+ B, A.jac)
end
+(A::Number, B::FAD) = B+A
```

Here, the first function will be used if the + operator is used with an `FAD` variable on the left hand side and a number on the right. The last line is a compact way of writing the opposite function, which will be used when the number is the left-hand argument to the + operator. We can also observe

that in the calculation of the new `val` variable, the dot operator for `broadcast(...)` is used. Once all possible options are implemented for the four elementary algebraic operators, as well as for elementary unary functions, we can simply write $f = y + \exp(2 * x * y)$ and Julia will understand that it is our implementation of + and $*$ operators and the exponential function that shall be used. The variable f will now become an `FAD`-struct with the correct value and derivatives.

In some situations, like in chapter 5, we want to take the sum of all the elements in the vector. If we look at how we can overload the `sum` function, one might think that we would try something like

```
import Base: sum
function sum(A::FAD)
    ## Overload sum
end
```

which would indeed work. Nonetheless, a more elegant approach that fully exploit Julia's multiple dispatch, would be to overload the `iterate` function. This function explains how we shall iterate through an AD variable:

```
function iterate(iter::FAD, state = 1)
    if state > length(iter.val)
        return nothing
    end
    return (iter[state], state + 1)
end
```

Now, the built-in `sum` function will work on AD variables since it knows how to iterate through the variables. When it adds up the values, the + operator we defined above is being used, and not only that! All built-in functions that iterate through the input will also work (given that the functions they use on the variable also are overloaded). As an example, if we now overload the division operator as well, the `Base` function `mean` will also work on `FAD` variables with no extra work!

4.2.2 Element-wise and Vector Multiplication

In mathematical programming languages like MATLAB and Julia, there is a difference between the $*$ and $.*$ operators. The first operator, $*$, is regular vector multiplication, meaning if v is a row vector and u is a column vector, both of length n, then $v * u$ is the normal vector product that results in a scalar, whereas $u * v$ gives an $n \times n$ matrix in which each row in v is multiplied by the corresponding row value of u. An attempt to evaluate $u * u$ will end in an error message saying that "The dimensions do not match matrix multiplication".

The $.*$ operator, on the other hand, represents element-wise multiplication. This means that if we have regular column vectors like u and $w = v'$, where w is the transpose of v, the evaluation of $u .* w$ will be element-wise multiplication of u and w, into a new vector of the same dimensions as u and w. Here, one needs to make a choice in the implementation of multiplication and division for AD in Julia, because as of now, there are no good ways of overloading any dot operators for custom types such as `FAD`. *Julia issue:dot operators (2017)* explains the problems of overloading the element-wise $.*$ operator, and states that there is no good way of actually doing this. The issue has still not been resolved. With this in mind, and that there will only be used element-wise multiplication in this project, I have decided that I herein redefine $*$ and implement it as element-wise multiplication. This means that if I have written regular multiplication expressions consisting of at least one `FAD` variable,

element-wise multiplication will be executed.

4.2.3 Optimizing `ForwardAutoDiff`

By looking closer at the implementation of the Jacobian in `FAD`, we can find that in some cases there are better approaches to storing the Jacobian that will gain computational efficiency. Before looking closer at the specific situations, I will explain how the sparse matrix type, `SparseMatrixCSC`, that `FAD` uses is built up and how it works. `SparseMatrixCSC` stands for *Compressed Sparse Column Sparse Matrix Storage* which is a standard format for storing spare matrices *(Saad, 2003)*. According to the Julia documentation *(SparseMatrixCSC docs, n.d.)* the SparseMatrixCSC struct is given as

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column i is in colptr[i]:(colptr[i+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

It represents a matrix with three vectors and two integers. The integers represent the size of the matrix and the three vectors represent all non-zero elements in the matrix. To explain how the vectors work, consider the example of a matrix $\mathbf{A}$ with the corresponding `SparseMatrixCSC` struct variables:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 5 & 0 \\ 0 & 3 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 7 \\ 2 & 4 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{aligned} \texttt{m} &= 4 \\ \texttt{n} &= 5 \\ \texttt{colptr} &= [1,3,5,5,7,8] \\ \texttt{rowval} &= [1,4,2,4,1,2,3] \\ \texttt{nzval} &= [1,2,3,4,5,6,7]. \end{aligned}$$

Here, `nzval` contains all the nonzero elements in $\mathbf{A}$. The order of the numbers is given by column from left to right and then row from top to bottom. The vector `rowval` has the same ordering as `nzval` and gives the row number to the corresponding value in `nzval`. The vector `colptr` contains the information of how many non-zero numbers there are in each column. For column number i, the sequence `colptr`$[i]$:`colptr`$[i+1]-1$ gives the indices in `nzval` and `rowval` that correspond to values in this column. For matrix A and column number 2 we get the indices

$$\texttt{colptr}[2] : (\texttt{colptr}[3] - 1) \Longrightarrow 3:4,$$

which gives row number 2 and 4 and values 3 and 4. For a column with only zero elements, like column 3, the sequence becomes

$$\texttt{colptr}[3] : (\texttt{colptr}[4] - 1) \Longrightarrow 5:4,$$

which indicates that there are no nonzero elements in this column.

This way of storing a matrix will decrease both memory usage and computational efficiency dramatically, compared to storing the full matrix, when working with large and sparse matrices. When performing operations on the matrix, e.g., an element-wise vector-matrix multiplication, the computational gain comes from the opportunity to neglect all zero values. If we store the matrix as a full dense matrix, then we have to compute a lot of multiplications that ends up being zero. With a

sparse matrix structure we can avoid making these computations. The method, however, brings some extra work that consist of doing numerous checks to make sure that we have done the multiplication correctly. Take the example from subsection 2.1.5, where we wish to compute $f = 2 \cdot \mathbf{x} \cdot \mathbf{y}$ for the primary variables $\mathbf{x}, \mathbf{y} \in \Re^3$ defined in (2.9). Initializing the primary variables as `FAD`-structs gives

$$\mathbf{x} = \left[\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right\} \right] \qquad \mathbf{y} = \left[\begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, \left\{ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right\} \right].$$

Here, I have written out the Jacobians as full matrices for better visualization, although in theory they will be stored as `SparseMatrixCSC` structs. In the multiplication of the two `FAD`-variables, the element-wise multiplication of the values is not interesting and we will instead focus on how we obtain the new Jacobian for $\mathbf{x} \cdot \mathbf{y}$. Similarly as in subsection 2.1.5 we find the new Jacobian using the product rule, but now we can separate the operations into two calculations. First, we have the Jacobian for the primary variable $\mathbf{x}$:

$$\begin{pmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \Longrightarrow \begin{pmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{pmatrix}.$$

Then for the primary variable $\mathbf{y}$ we obtain the Jacobian

$$\begin{pmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}. \tag{4.1}$$

We immediately observe that two of the multiplications are unnecessary, since one of the matrices is a null matrix. Remember that the left matrix in all the calculations above actually is a vector that we have transformed into a diagonal matrix to illustrate how the element-wise multiplication happens. This means that when the previous Jacobian is a diagonal matrix, what we actually can do to obtain the new Jacobian, is element-wise multiplication between the vector on the left hand side and the vector that is the diagonal on the Jacobian. And even better, when the previous Jacobian is the identity matrix, like in calculation (4.1), the product is simply a diagonal matrix with the left hand side vector on the diagonal. This is where the idea of optimizing `FAD` comes from. By knowing what type of Jacobian we have at all times, we can sometimes take safe shortcuts in our calculations.

4.2.4 `CustomJacobianAutoDiff`

The optimized `FAD` is called `CustomJacobianAutoDiff` (`CJAD`). *Custom Jacobian* comes from having four different types of Jacobians. The structure of `CJAD` is similar to `FAD`, but instead of storing all the Jacobians as a vector consisting of `SparseMatrixCSC`-types, each element in the vector is now of type `CustomJac`:

```
struct CJAD
    val::Vector{Float64}
    customJacs::Vector{CustomJac}
end
```

`CustomJac` is an abstract type that has four structs that extend it:

- `NullJac` – a struct only containing two numbers that represents the number of rows and columns in a null matrix.
- `IdentityJac` – a struct only containing one number that represents the number of rows and columns in an identity matrix.
- `DiagJac` – a struct containing a vector called `jac` with the diagonal values of a diagonal matrix. The length of `jac` equals the number of rows and columns.
- `SparseJac` – a struct containing a `SparseMatrixCSC` matrix called `jac`.

Now, the multiple dispatch system in Julia comes in handy once again. Since Julia understands at run-time what type of `CustomJac` we have – whether it is a `NullJac`, `IdentityJac`, `DiagJac` or `SparseJac` – we can implement different methods for all possible combinations. In each implementation, we now know what type of matrix we are dealing with, and we can thus optimize the performance. As an example, consider calculating the Jacobian w.r.t the primary variable $\mathbf{y}$ in (4.1) (All the implementations of operators on structs extending the `CustomJac` type are called from outer functions that check the legality of the operations; hence, the following code excerpts contain no safety checks.) First, we have a vector multiplied element-wise by a null matrix. The implementation

```
*(A::Vector{<:Number}, B::NullJac) = B
```

knows immediately that there is no need to do any calculations and the result will be a null matrix of the same size as before. Without diving too deep into the implementation of `SparseMatrixCSC`, but only by considering what kind of information the `SparseMatrixCSC`-struct contains, this implementation should not make a big difference in the computational efficiency of `CJAD` compared to `FAD`. The reason for this is that the implementation of `SparseMatrixCSC` will also quickly realize that its vectors are of length zero and that the matrix is a null matrix. Hence, the speed difference of the two implementations will be small for this particular operator. However, the second calculation in (4.1) is a vector multiplied element-wise by an identity matrix. The result will be a diagonal matrix with the values of the vector on the diagonal. The `SparseMatrixCSC` implementation only knows that we have a sparse matrix with some values and is unaware that it actually is the identity matrix. This means that it has to do all the calculations without any shortcuts. The implementation for `IdentityJac`, however, automatically knows the result of this calculation and simply makes a diagonal Jacobian with the vector on the left hand side:

```
*(A::Vector{<:Number}, B::IdentityJac) = DiagJac(A)
```

This is the first calculation we have seen that will make `CJAD` considerably more computational efficient than `FAD`. To finish the calculations in (4.1), we finally have to add a null matrix and a diagonal matrix. Since there is really no need of doing this adding, the implementation simply returns the diagonal matrix:

```
+(A::NullJac, B::DiagJac) = B
```

For the same reason as explained for the element-wise multiplication between a vector and a `NullJac` matrix, this implementation will only have small advantages compared to the `SparseMatrixCSC` implementation used in `FAD`, which will also quickly figure out the

needlessness of the calculation.

Another type of calculation that CJAD will do more efficient than FAD is when a vector is element-wise multiplied with a diagonal Jacobian. The operator that element-wise multiply a vector with a CustomJac-variable is often used, not only in itself, but also because it appears every time the chain or product rule is used. The product rule is used every time we multiply two CJAD-variables, and the chain rule appears when we evaluate an elementary function like exp or sin or if we evaluate some exponent of a CJAD-variable. If the Jacobian is of type DiagJac, the corresponding element-wise multiplication is simply two diagonal matrices multiplied together. CJAD's implementation knows this and can multiply the two vectors element-wise and obtain the new diagonal of the new Jacobian:

```
*(A::Vector{<:Number}, B::DiagJac) = DiagJac(A .* B.jac)
```

The same operation for FAD will be slower since SparseMatrixCSC does not possess any information that the matrix is diagonal and thus has to perform the operation as a regular multiplication between to sparse matrices. This will include checks to be certain that the multiplication is done correctly, and will hence be slower than multiplying two vectors element-wise. As said, this operation appears often, since it is used in the chain rule and in the product rule. Consider the product rule and the implementation

```
function product_rule(A::CJAD, B::CJAD)
    nJac = length(A.customJacs)
    newJac = Vector{CustomJac}(undef,nJac)
    for i = 1:nJac
        newJac[i] = A.val * B.customJacs[i] + B.val * A.customJacs[i]
    end
    return newJac
end
```

We loop through all the Jacobians and perform the product rule to obtain the new Jacobians. Here, A.val and B.val are the value vectors, whereas B.customJacs[i] and A.customJacs[i] are any of the four extensions of CustomJac. In every iteration in which at least one of the Jacobians is a special case and not the general SparseJac, we will gain computational efficiency like explained for the element-wise multiplication operator between a vector and a NullJac/IdentityJac/DiagJac. If both the Jacobians are different from SparseJac, we also gain computational efficiency from the addition operator. Just like for the element-wise multiplication, element-wise addition is faster for NullJac/IdentityJac/DiagJac than it is for the sparse matrices in SparseMatrixCSC.

When both Jacobians are general sparse matrices, CJAD and FAD are equal because CJAD transitions to use the SparseMatrixCSC-library as well with its SparseJac type. Let us now therefore look closer into how we handle the overloading of element-wise multiplication for SparseJac. There are at least two different ways of implementing this operator that will use different parts of the SparseMatrixCSC library. The first possibility is to convert the vector into a sparse diagonal matrix and perform a matrix–matrix multiplication with SparseMatrixCSC matrices:

```
function *(A::SparseJac, B::Vector{<:Number})
    diagIndex = 1:length(B)
    valDiag = sparse(diagIndex, diagIndex, B)
    newJac = valDiag * A.jac
    return SparseJac(newJac)
```

```
end
```

I will call this method the *matrix-multiplication* method. The second method keeps the vector form and uses element-wise multiplication between the vector and the `SparseMatrixCSC` matrix:

```
*(A::SparseJac, B::Vector{<:Number}) = SparseJac(B .* A.jac)
```

I will call this the *dot-multiplication* method. As it is difficult to say without first-hand knowledge of the implementation of `SparseMatrixCSC` which method is best to implement, I will test the two methods against each other. The test consist of element-wise multiplying a random vector of length 40 and a random sparse matrix of size 40×40 having 20 percent nonzero elements. The multiplication will be performed 10,000 times to separate the efficiency of the two methods. To check which method is actually more efficient, and by how much, we can use the *@btime* macro function from the *BenchmarkTools (n.d.)* library. This function gives us the number of allocations, the amount of memory used, and the time spent. Table 4.1 reports the results[1] from this test.

Table 4.1: The number of allocations, amount of memory, and time spent for element-wise multiplication of a vector of length 40 and a sparse matrix of size 40×40 with 20 percent nonzero elements. The vector and matrix are multiplied together 10,000 times.

Method	Number of Allocations	Megabytes	Milliseconds
Matrix-multiplication	1,420,000	194	187
Dot-multiplication	60,000	264	87

The test is unequivocal: the *dot-multiplication* method has far less allocations, is more than twice as fast, but it uses a bit more memory. Based on this it might tempting to say that the *dot-multiplication* method is a better choice. However, in numerical solution of PDEs, the matrices we work with are generally much larger and very sparse. Table 4.2 reports results for a more representative test example, in which I have used a matrix from chapter 5, which is of size $8,000 \times 8,000$ and only contains 0.08 percent nonzero elements.

Table 4.2: The number of allocations, amount of memory, and time spent for element-wise multiplication of a vector of length 8,000 and a matrix of size $8,000 \times 8,000$ with 0.08 percent nonzero elements. The matrix is taken from chapter 5.

Method	Number of Allocations	Megabytes	Milliseconds
Matrix-multiplication	240,370	32	28
Dot-multiplication	180	10,243	2,458

The dot-multiplication method still requires far less allocations, but now consumes a lot more memory than the *matrix-multiplication* method. As a consequence, it is almost 100 times slower than the matrix-multiplication method! This shows that for the purpose of this implementation, the matrix-multiplication method is a much better choice.

[1] All benchmarks in this project are performed on a MacBook Pro (Retina, 13-inch, Late 2013), 2.8 GHz Intel Core i7 processor and 16 GB 1,600 MHz DDR3 memory.

4.2.5 Efficient Versus Readable and Elegant Code

There is usually a fine balance between writing efficient and readable/elegant code. Sometimes you have the opportunity to do both, but often you have to choose which of them you want to give most focus. At the end of subsection 4.2.1, I explained how we can elegantly implement the `sum` function, and all other built-in functions that iterate through the `FAD`-variables, only by implementing the `iterate` function. This is a very elegant use of Julia's multiple dispatch system, as we get a lot of functionality "for free". The downside, however, is that we lose potential computational efficiency that we could achieve by implementing the `sum` function ourselves. Let us define a random function ψ, where the AD-representation and the `sum` of ψ are given by

$$\psi = \left[\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \left\{ \begin{pmatrix} 3 & 0 & 1 \\ 2 & 1 & 0 \\ 0 & 0 & 8 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right\} \right] \qquad \texttt{sum}(\psi) = \left[6, \ \left\{ \begin{pmatrix} 5 & 1 & 9 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \right\} \right].$$

Consider now the previous approach, in which we only implemented the `iterate` function and reused `Base.sum` (i.e., the standard `sum` function in Julia) to sum the values in ψ. For each iterated addition, we access row i in ψ and add it to the total sum. This consist of extracting row i from the value vector and from each Jacobian, and then creating a new AD-variable that only consists of row i. This leads to a lot of extra memory allocation, and in the case for ψ, we have to allocate memory for three new AD-variables. This extra allocation can be avoided by doing the summation of each column at once, instead of operate on a row to row basis. For `CJAD`, this can be done with the following overloaded implementation of `sum`:

```
function sum(A::CJAD)
    val = sum(A.val)
    jac = Vector{CustomJac}(undef, length(A.customJacs))
    for i = 1:length(A.customJacs)
        jac[i] = colSum(A.customJacs[i])
    end
    return CJAD(val, jac)
end
```

Instead of retrieving each row, one by one, like we did using `Base.sum`, we now add up each column in one go. Here, `sum(A.val)` is the built-in sum function in `Base` for summation of a vector, and `colSum` is a function implemented in each extended type of `CustomJac`. This function sums up all the columns in the Jacobians and returns a $1 \times n$ matrix. Here, n is a number that depends on the dimension of the relevant primary variable ($n_\psi = 3$). In the three special cases, the function simply returns a $1 \times n$ null matrix for `NullJac`, a $1 \times n$ matrix containing only ones for `IdentityJac`, and a $1 \times n$ matrix, which is the transposed of the diagonal vector, for `DiagJac`. For `SparseJac`, it returns a $1 \times n$ matrix in which all the nonzero values in each column are added together. To check that this new `sum` function actually is more efficient, we can again use the *@btime* macro function from the *BenchmarkTools (n.d.)* library. By creating a random `CJAD`-variable of length 400 with all of the four different Jacobian types (all 400×400 with random values), we can compare how efficient the two methods sum up the `CJAD`-variable. Table 4.3 reports the result and shows that even though only implementing the `iterate` function is an elegant solution, it is much more efficient both in terms of memory consumption and runtime to implement the `sum` function specifically.

Table 4.3: Table with the number of allocations, memory usage, and runtime spent by summing a random `CJAD`-variable of length 400 with two different summation methods.

Function	Number of Allocations	Kilobytes	Microseconds
`sum`	40	79	81
`Base.sum`	28,371	19,597	58,051

Vectorized and Devectorized Code

The second issue I want to discuss is vectorized and devectorized code. The difference between the two can be quickly explained by an example. Consider two vectors **x** and **y** of the same length. To multiply the elements of the two vectors we can either use the vectorized method, `z = x .* y`, or we can loop through the vectors in the devectorized method:

```
for i = 1:length(x)
    z[i] = x[i] * y[i]
end
```

In a blog post, former Julia developer John Myles White explains how Julia is faster at executing devectorized code compared to vectorized code (White, 2013). This is in sharp contrast to other high-level mathematical programming languages like MATLAB, in which conventional wisdom would tell you that you have to vectorize your code to get the best computational performance. When you vectorize, MATLAB uses optimized code in C to execute the calculations. If you devectorize your code by calling a for-loop, you potentially bring in a lot of extra overhead. Note that, as mentioned in subsection 3.1.3, recent development of MATLAB's just-in-time (JIT) compiler has changed the landscape a bit, and devectorized code can sometimes be significantly faster than vectorized code, in particular if the latter has to allocate a lot of temporary memory.

In Julia, for-loops are generally close to matching C's speed. So you might say that "why not devectorize all the code?" And that could indeed be done, but here we have another example of the conflict between efficient and readable code. From the example with element-wise multiplication of the two vectors, the vectorized form is much more compact and readable, but according to White, the devectorized version is faster. To make a decision whether to implement vectorized or devectorized, we need to know more specifically what the differences are. White's blog post is from 2013, and since then, the vectorization in Julia has improved. Steven G. Johnson has written a newer blog post from 2017 *(Johnson, 2017.01.21)*, in which he closer explains the reason why it is difficult to obtain the same speed for vectorized code as we have for devectorized code in a language like Julia. To explain this, consider two different vector functions f and g that we want to multiply element-wise

$$h(x,y) = f(x) \cdot g(y), \qquad f(x) = \exp(x), \qquad g(y) = \log(y).$$

In Julia, there are now three ways of obtaining $h(x,y)$. First we have the devectorized version

```
function devectorized(x,y)
    h = similar(x)
    for i = 1:length(x)
        h[i] = exp(x[i]) * log(y[i])
    end
    return h
end
```

According to White, this should be the fastest way of obtaining $h(x, y)$. Secondly, we have the method of vectorizing the functions f and g such that we can call the functions with vectors and they will be performed element-wise to the input variables. I have called this method `dotsInside`:

```
f(x) = exp.(x)
g(y) = log.(y)
dotsInside(x,y) = f(x) .* g(y)
```

Lastly, we do not vectorize the functions f and g, but when we call them with the vectors x and y, we use the dot operators (that in reality is the `broadcast(...)` function) on the functions to tell the Julia compiler that the functions should be executed element-wise on the input parameters. I have called this method `dotsOutside`:

```
f(x) = exp(x)
g(y) = log(y)
dotsOutside(x,y) = f.(x) .* g.(y)
```

To separate the three methods, I have calculated $h(x, y)$ 1,000 times for random x and y vectors of length 100,000 and evaluated the performance using *@btime*. Table 4.4 reports the results.

Table 4.4: Table with the number of allocations, memory usage, and runtime spent by evaluating $h(x, y) = \exp(x) \cdot \log(y)$ 1,000 times for three different methods; x and y are random vectors of length 100,000.

Method	Number of Allocations	Megabytes	Seconds
`devectorized`	2,000	789	2.104
`dotsInside`	6,000	2,405	2.704
`dotsOutside`	2,000	789	2.121

The `devectorized` and the `dotsOutside` methods perform identically, whereas `dotsInside` requires three times the number of allocations and memory and is also somewhat slower than the two other methods. This result confirms what Steven G. Johnson explains in *Johnson (2017.01.21)*, where he says that the problem with vectorized operations is that they can generate new temporary arrays for each operation and that every operation is executed in a separate loop. This means that when Julia compiles the program and translates the vectorized code into devectorized code, it will compute the `dotsInside` method similar to the following code:

```
function dotsInside(x,y)
    n = length(x)
    tmp1 = Vector{Float64}(undef,n)
    for i = 1:n
        tmp1[i] = exp(x[i])
    end
    tmp2 = Vector{Float64}(undef,n)
    for j = 1:n
        tmp2[j] = log(y[i])
    end
    tmp3 = Vector{Float64}(undef,n)
    for k = 1:n
        tmp3[k] = tmp1[k] * tmp2[k]
    end
    return tmp3
end
```

The compiler does this because it does not understand that all the operations are element-wise. As a consequence, it has to allocate three new arrays compared to the single new array in the `devectorized` method. However, with the `dotsOutside` method, the compiler understands that all the operations in the expression are vectorized operations and we get what Johnson calls *loop fusion*. This essentially means that Julia devectorize the `dotsOutside` method into the `devectorized` method, and since Julia does not call low-level code like C to perform its vectorized code, the performance of `devectorized` and `dotsOutside` will be similar. Another advantage of using the dot operator like in `dotsOutside` is that we do not have to define if a function is vectorized or not. In other words, f and g can be used as functions for vectors and scalars, we specify what type of use we want when we call the function. This leads to more readable code, as we do not have to check whether a function is written for vectorization or not.

So what are the consequences for the implementation of `CJAD`? The result from Table 4.4 implies that as long as we write all our vectorized code like the `dotsOutside` method or in such a manner that it is obvious for the compiler that it is only vectorized operations involved, we will have the same computational efficiency as for devectorized code. More specifically for the `CJAD` implementation, we often have a vector that is added/subtracted/divided/multiplied with another vector, and then the vectorized implementation will have the same computational efficiency as the devectorized. Since devectorized code is less readable, implementations reported in the following will consist of vectorized code for readability.

4.3 Benchmarking Automatic Differentiation

As mentioned in section 4.1, Julia already has an AD library called `ForwardDiff` (Revels et al., 2016) that uses forward AD. Hence, it would be interesting to see how the different implementations compare as the functions evaluated increase in complexity. In addition to `ForwardDiff`, the Julia implementations are compared to two optimized AD implementations from MRST. The basic implementation in MRST is similar to `FAD`, where the Jacobians has a sparse matrix structure, and is henceforth referred to as `MRST`. The second MATLAB implementation is similar to `CJAD` in the sense that it exploits the diagonal structure of some Jacobians. This method is called `MRST_diagonal`. To test the efficiency of the different AD tools, I have evaluated the vector function $f : \Re^{n\times 3} \to \Re^n$, where

$$f(x, y, z) = \exp(2xy) - 4xz^2 + 13x - 7, \qquad x, y, z \in \Re^n. \tag{4.2}$$

Figure 4.1 reports how the computational time[2] scales for the different methods, calculating the function value, and the Jacobian of the function, as the length of the vectors (n) increases.

[2]The benchmarks in Julia are still performed by using the benchmarking library *BenchmarkTools (n.d.)*. For measuring time in MATLAB I have taken the median of multiple tests using the stopwatch *Tic (n.d.)*.

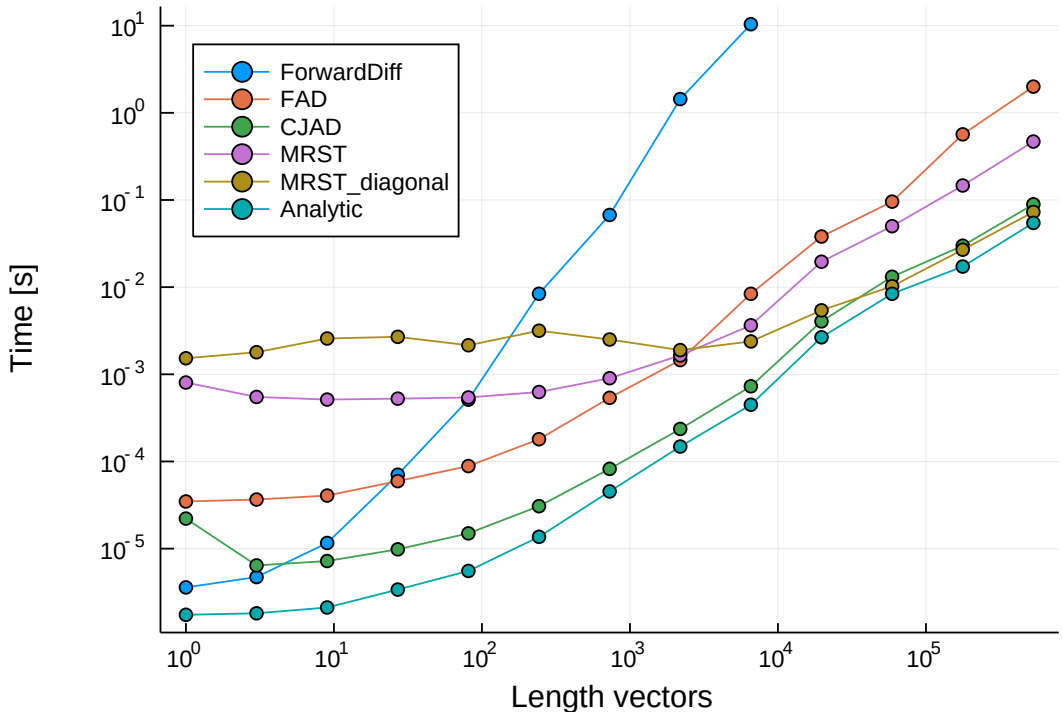


Figure 4.1: Computational time for different methods calculating the value and Jacobian of f in Equation (4.2) as a function of length of the input vectors.

Figure 4.1 reports six different graphs. The first thing you observe is that `ForwardDiff` scales very badly as n becomes large. This is because it creates and works with the full $3n \times 3n$ Jacobian matrix as discussed in section 4.1. I stopped the benchmark of `ForwardDiff` for $n = 3^8$, as the Jacobian at this point already has more than 380 million elements. We can nevertheless observe that for small vectors, the methods implemented in MATLAB, as well as `FAD` and `CJAD`, have more overhead than `ForwardDiff` and the analytic solution. This makes them slower for small n, but as n grows, this overhead becomes quickly negligible. Based on Figure 4.1, from a numerical simulation point of view, `ForwardDiff` is useless for obtaining the Jacobian of functions, as it scales badly for increasing n. Apart from the conclusion on `ForwardDiff`, we can observe that it is a change in dominance at $n = 3^7 = 2,187$. For shorter vectors, both methods implemented in Julia are more efficient than the two methods in MATLAB. For $n > 2,187$ `FAD` is the slowest, followed by `MRST`. Hence, as n becomes larger than 2,187, the methods in Julia and MATLAB that exploit the diagonal structure of the Jacobian perform better than the ones that only use a sparse matrix structure. This makes sense, as the evaluation of f will only give a diagonal Jacobian. As n becomes large, `CJAD` and `MRST_diagonal` perform very similarly and their computational costs approach the analytic evaluation. What is also interesting to see is that whereas `CJAD` always is faster than `FAD`, `MRST_diagonal` is less efficient than `MRST` for short vectors. Assuming that both the MRST implementations are optimized, this indicates that the elegant way of implementing a custom Jacobian in Julia using multiple dispatch adds very little overhead compared to MATLAB.

The test case just discussed is highly idealized: because each element $f[i]$ of f does not depend on $x[k]$, $y[k]$ and $z[k]$ for $k \neq i$, the Jacobians will keep a diagonal structure for f. This means that `CJAD` never uses its `SparseJac` type and `MRST_diagonal` can use its optimized implementation for diagonal Jacobians. If we, for example, want to calculate something like

$$g(x) = \frac{x[2:\text{end}] - x[1:\text{end}-1]}{\texttt{sum}(x)},$$

the diagonal structure of the Jacobians is gone, and `MRST_diagonal` transitions to use the `MRST` implementation, and `CJAD` has to use the `SparseJac` type. This does not imply that the optimized methods are implemented in vain, since functions evaluated in numerical applications often have parts that will have diagonal Jacobians, and other parts that will not. We can thus expect to gain computational efficiency from parts of the function evaluations using `CJAD` and `MRST_diagonal`. To assess how much we can gain, we have to look at more realistic examples. I will do this in the following chapter 5.

Chapter 5

Flow Solver With Automatic Differentiation

To compare the different AD implementations in Julia and MATLAB I have implemented MRST's tutorial, "Single-phase Compressible AD Solver" from `flowSolverTutorialAD.m` *(Single-phase Compressible AD Solver, n.d.)*, in Julia. The example is made as an introduction to how AD can be used in MRST, hence it is a good example to use when the goal is to compare the implementations of AD in MATLAB and Julia.

5.1 Grid Construction

The example consist of modelling how the pressure drops within a rectangular reservoir measuring $200 \times 200 \times 50\text{m}^3$ when we have a well that is producing oil. As mentioned in the introduction, this is called primary production, and because of high pressure in the reservoir, the oil flows out by itself. "Single-phase solver" only means that we do not have different phases of fluids, like water and/or gas, present during the simulation. In this simulation we assume there is only oil present in the reservoir. As the main purpose of this example is to compare the AD tools in MATLAB and Julia, and not the process of solving the problem, including setting up the grid and other necessary variables, some of the initialization and plotting have been performed in Julia by calling code from MRST. This has been done by using the package MATLAB.jl *(MATLAB.jl, n.d.)*. This package allows calling MATLAB functions from Julia and retrieve the output variables. This is done by the function call

```
out1, out2 = mxcall(:matlab_function_name, 2, in1, in2, in3)
```

where we have three input parameter and two output parameters. We have to specify the number of output parameters after the MATLAB function name. By calling MATLAB from Julia, we can use MRST's `G = computeGeometry(...)` function to set up the grid for the simulation. The grid of the reservoir can be seen in Figure 5.2a. The variable `G`, that contains the grid properties, is now a structured array (struct), having all the information on cells, facets, vertices and nodes that we need to make the discrete divergence and gradient operators as explained in section 2.2.

Next, we define the properties of the rock. As explained in section 1.3, in an oil reservoir, the oil lies inside porous rock and the properties of the rock will affect the flow of the oil. The amount of oil we can have inside a cell is measured as pore volume and the ability to transmit fluids is given by the permeability. We start by making a new variable, `rock`, that contains the permeability and the

porosity of the rock. The porosity is a number between zero and one, describing the porosity of the rock at a reference pressure. The amount of fluid inside a cell is given by the pore volume. To find an expression for this, we say that in our model the rock is compressed constantly as a function of pressure. This gives us an analytic solution for the pore volume, given as a function of pressure

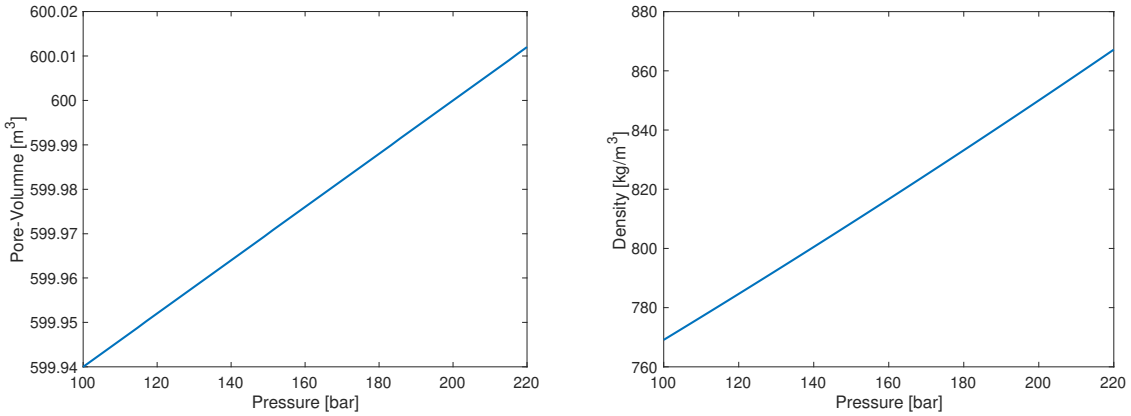
$$pv(p) = \mathrm{pv}_r \exp[c_r \cdot (p - p_r)]. \tag{5.1}$$

In this expression, pv_r is the rock pore volume, in a cell, at a reference pressure p_r. The value of pv_r is given by the porosity of the rock, multiplied with the total volume of the cell. c_r is a constant controlling how the rock is compressed. The pore volume of the rock, as a function of pressure values, are visualized in Figure 5.1a. The graph shows that for a cell with volume $2000\mathrm{m}^3$ there is approximately room for $600\mathrm{m}^3$ oil.

Since we assume that the oil have a constant compressibility, the density of the oil is given as an analytic function of pressure

$$\rho(p) = \rho_r \exp[c \cdot (p - p_r)], \tag{5.2}$$

where $\rho_r = 850\mathrm{kg/m}^3$ is the reference density at the reference pressure p_r and c is a constant controlling how fast the oil is compressed. The density of the oil, as a function of pressure, is plotted for some pressure values in Figure 5.1b.



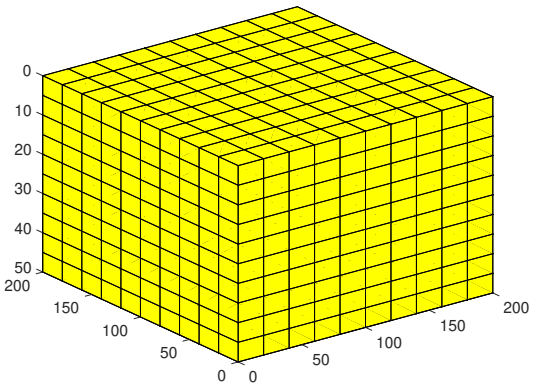
(a) The pore volume in a cell as a function of pressure. **(b)** The density of the oil in the reservoir plotted as a function of pressure.

Figure 5.1

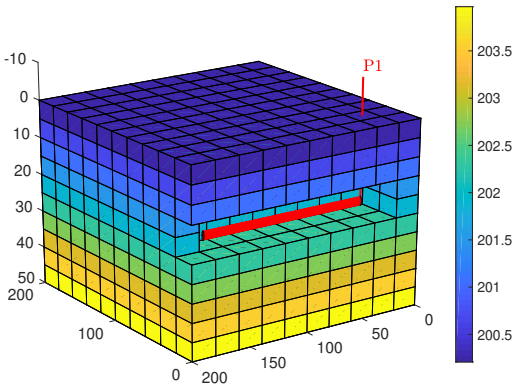
The initial pressure in the reservoir is calculated by solving the nonlinear ODE

$$\frac{dp}{dz} = g \cdot \rho(p), \quad p(z = 0) = p_r = 200\mathrm{bar}$$

for the fluid density given by Equation (5.2) and g as the gravity. The well is then inserted in 8 grid elements using the `W = addWell(...)` function. The grid with initial pressure and well can be seen in Figure 5.2b.



(a) Uniform $10 \times 10 \times 10$ grid of the $200 \times 200 \times 50 \text{m}^3$ big reservoir.



(b) Reservoir grid plotted with initial pressure and well P1. Pressure is given in bar. The well covers 8 grid elements. Some grid elements are removed to give a better visualization of the well.

Figure 5.2

5.2 Setup of Governing Equations

After initializing the grid we want to define the discrete gradient and divergence operators, as well as the transmissibilities as explained in Equation (2.15). This is done by exploiting that we have all the necessary information about the grid properties, such as the cells centroid coordinates, facets area, and so on, stored in the struct `G`. In `rock` we have stored the permeability inside each cell that will affect the flux through each facet. With all this information we can now obtain the transmissibilities `T` and the discrete operators

```
dGrad(x) = C * x
dDiv(x) = -C' * x
```

The matrix C in the discrete operators are created such that when it is multiplied by the pressure p, the result becomes the pressure difference between two adjacent cells, as defined in Equation (2.18). The matrix C is stored as a sparse matrix, and the reason why is clear from Figure 5.3, which shows the sparse structure of the matrix. The divergence operator is made using the fact that in the continuous case, the gradient operator is the adjoint of the divergence operator

$$\int_\Omega p\nabla \cdot \vec{v} d\Omega + \int_\Omega \vec{v}\nabla p d\Omega = 0.$$

This holds for the discrete case as well *(Lie, 2019)*, and hence the adjoint of `C` is the negative transpose of `C`.

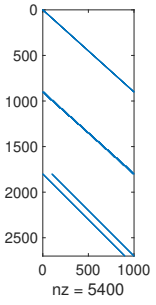


Figure 5.3: Structure of the discrete operator `C`.

Now we have all the ingredients to set up the governing equations for the flow in the reservoir. We use a finite volume method to discretize in space, as explained in section 2.2, and a backward Euler

method to discretize in time. In the end, all the equations we want to solve should be on residual form, $\boldsymbol{F}(\boldsymbol{x}) = 0$, so that we can use the Newton-Raphson method described in Equation (2.11) to solve the system. As there are multiple equations that will be a part of the residual function $\boldsymbol{F}(\boldsymbol{x})$, we define them separately first. One of the advantages of defining the discrete gradient- and divergence operator is that the continuous and discrete forms of the equations look very similar. Hence, I will first state the continuous version of the equation and then the discrete, so that it is easy to see how similar they look. I start by defining Darcy's law, which explains how the oil will flow through the porous rock

$$\mathbf{v} = -\frac{K}{\mu}(\nabla p - \mathbf{g}\rho). \tag{5.3}$$

K is the permeability that we have saved in the `rock` variable and μ is the viscosity of the oil. The corresponding discrete equation that we call `flux` is given by

```
flux(p) = -(T / μ) * (dGrad(p) - g*average(ρ(p))*gradz)
```

Here, T is the transmissibilities that contain K and the other properties of the grid. Since two adjacent cells can have different values of ρ, we use the average for the two cells. `gradz` is the gradient of the cell centroid's z-value. This determines how much the flux depend on g given the orientation of the adjacent cells. When `flux` is defined, we can define the continuity equation in the continuous case

$$\frac{\partial}{\partial t}(\phi\rho) + \nabla \cdot (\rho \mathbf{v}) = q, \tag{5.4}$$

where ϕ is the porosity of the rock. Since we will handle the well later, the source term q representing injection or production fluids, is set to zero for now. In the corresponding discrete case we get the function

```
presEq(p, p0, dt) = (1/dt) * (pv(p)*ρ(p) - pv.(p0).*ρ.(p0)) +
                      dDiv(average(ρ(p)) * flux(p))
```

where `pv` is the pore volume of the rock given in Equation (5.1) and $p0$ is the pressure at the previous time step.

In addition, we need a few equations to represent the flow inside the wellbore. This flow will be the production term q we ignored in the derivation of the `presEq` function. The standard model is to assume that the pressure is in hydrostatic equilibrium inside the wellbore, so that the pressure in a perforated cell (i.e., a cell in which the wellbore is open to the reservoir rock and the fluid can flow in or out of the well) is given as a hydrostatic difference from the pressure at a datum point (the bottom-hole pressure), typically given at the top of the reservoir. That is, the pressure in a perforated cell c is given by

$$p_c = p_{bhp} + g(z_c - z_{bhp})\rho.$$

In the discrete case this is given by the function `p_conn`

```
p_conn(bhp) = bhp + g*dz*ρ(bhp)
```

The pressure drop near the well usually takes place on a much shorter length scale than the size

of a grid cell, and is usually modelled through a semi-analytical expression that relates flow rate to the difference between the reservoir and wellbore pressures. Hence the analytic expression for the production in a perforated cell is given by

$$q = \frac{\rho}{\mu}\mathrm{WI}(p_c - p_r),$$

where WI is properties of the rock and the oil at the applicable cell. Since this equation only apply on a few of the cells in the reservoir, we also need a list of the indices `wc` for the perforated cells. These indices and the WI variable are given by the `W` variable we received from the `addWell` function. The discrete expression for the production in all the perforated cells are given by

```
q_conn(p,bhp) = WI * (ρ(p[wc])/mu) * (p_conn(bhp) - p[wc])
```

The residual expression for the total production `qS` is then given by summing up all the production from each perforated cell, giving the expression `rateEq`

```
rateEq(p,bhp,qS) = qS - sum(q_conn(p,bhp))/rhoS
```

Here `rhoS` is the density of the oil at the surface, to obtain the total volume produced. To control the well, we can either set total inflow or outflow of the well (evaluated at surface pressure) to be constant, or set the datum (bottom-hole) pressure as constant. In either case, we will wish to compute the other (i.e., if the bottom-hole pressure is given, we determine the surface rate, and vice versa). Herein, we assume the bottom-hole pressure to be given as 100 bar and we get

```
ctrlEq(bhp) = bhp - 100*barsa
```

When all the governing equations are defined, we merge them into one large residual vector function $\boldsymbol{F}(\boldsymbol{x})$. The first 1,000 residual equations are the `presEq` with negative production `q_conn` for the indices `wc`. Equation number 1,001 is the `rateEq` and Equation 1,002 is the `ctrlEq`. Hence, $\boldsymbol{x} \in \Re^{1,002}$, where the first 1,000 elements are the average pressure in each cell, element 1,001 will be the pressure at the datum point inside the well (`bhp`), and element 1,002 is the surface fluid rate `qS`. Now, if we start by defining `p`, `bhp` and `qS` as AD-variables, $\boldsymbol{F}$ will also be an AD-variable and we will have the Jacobian of the residual vector function $\boldsymbol{F}$. This means we can solve the equations using the Newton-Raphson method, defined in Equation (2.11). A pseudo code of how we solve the system can be seen below.

```
## Define AD-variables length of simulation, endTime and timestep dt.
while timeNow < endTime
    while ## Newton-Raphson method has not converged.
        f = F(p, bhp, qS)
        updateStep = -(f.jac \ f.val)
        ## Update AD-variables val-value
    end
    timeNow += dt
end
```

5.3 Flow Solver Results

If we simulate how the pressure in the reservoir will decay, we will get the result displayed in Figure 5.4.

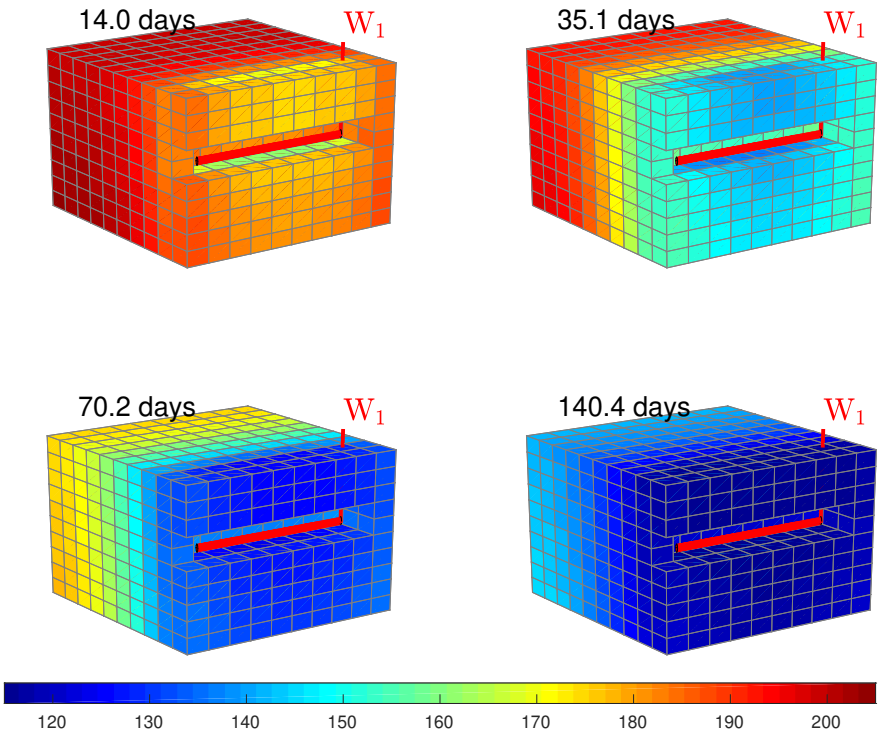


Figure 5.4: The pressure in the reservoir displayed at four different times. Pressure is given in bar.

Note the different intervals for the color bar in Figure 5.4 compared to Figure 5.2b. With the current color bar interval, at initial state, the whole reservoir will be displayed red. Hence, we can see how the reservoir from the beginning being approximately 200 bar everywhere, begins with the largest pressure decay close to the well, but after some time, the oil is pushed towards the well by the pressure differences and the pressure also begins to decay furthest away from the well.

Figure 5.5 shows the development of the production rate and the average pressure inside the reservoir. We can see how the production rate follows the average pressure inside the reservoir, and that after some time, it approaches zero. This is when primary production ends, as explained in section 1.3, and we need to apply external pressure to continue production in secondary production. To do this in the best possible way, it is important to be able to simulate how the pressure evolves inside the reservoir so that we can make good decisions on which actions produce the best possible results and when to take these actions. This flow solver is a simple example of how we can model primary production in a reservoir elegantly with governing equations on residual form, created by discrete differentiation operators and AD. A simulation of secondary production can be seen in section 6.4.

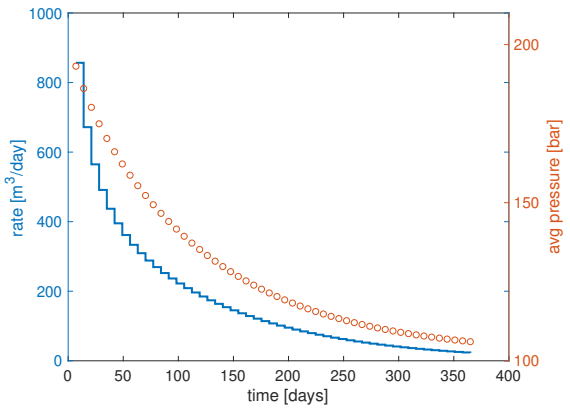


Figure 5.5: The production rate and the average pressure in the reservoir as a function of time.

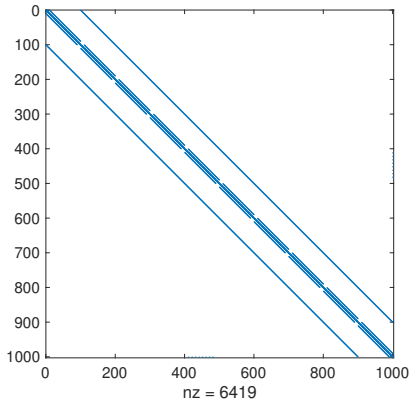


Figure 5.6: Structure of the $1,002 \times 1,002$ Jacobian of the governing equations. There are 6,419 non-zero elements.

However, our main interest is to observe how the different AD tools perform when solving the residual equations with the Newton-Raphson method and AD. Before looking at the results it is interesting to have a look at how the Jacobian of $\boldsymbol{F}$ looks like. Figure 5.6 shows the structure of this Jacobian. The first impression is that the matrix is very sparse. The reason for this is that each cell will only depend on its neighboring cells. A small exception from this are the few cells containing the well. These cells will also depend on the bottom hole pressure. As there are only 6,419 non-zero elements, out of more than 1 million matrix elements, it is clear that storing the full $1,002 \times 1,002$ matrix will be very inefficient.

As explained in chapter 4, the different types of AD mentioned store the Jacobians differently. Both `FAD`, `CJAD` and the two AD implementations in MRST, will, at least eventually, store the Jacobians as a list of sparse matrices where each element in the list is the Jacobian with respect to one primary variable. In this example, this is expected to be a lot more efficient than storing the full $1,002 \times 1,002$ Jacobian as `ForwardDiff` does.

To benchmark the different methods, we need to do some extra work to make sure that it is actually the AD we test and not other parts of the code. This is especially important for the code running in Julia, since when we call MATLAB from Julia, it will be a lot of overhead. This means that the setup of the discrete gradient and divergence operators will take longer when performed in MATLAB called from Julia, than when we do it directly in MATLAB. If we want to run the full simulation, it is not possible to separate the AD part fully, but if we only test the main-loop containing the Newton-Raphson method, AD will be a dominating part of the computations together with the linear solver `f.jac\f.val`. This means that we at least will get an indication of how well the AD tools perform compared to each other. To see how the different methods scale as the discretization becomes finer, and the system we solve grow, the benchmark consist of three different discretizations. The first is the original setup with 10 cells in the spatial directions x, y, and z. This gives a total number of 1,000 cells in the reservoir. Then, I have also benchmarked the implementations for 20 and 30 cells in each spatial direction. The time spent solving the system for the different methods can be seen in Table 5.1.

Table 5.1: Table with speed tests of different AD methods solving the "Single-Phase Compressible AD Solver" for different discretizations.

Number of cells	ForwardDiff	FAD	CJAD	MRST
$10 \times 10 \times 10$	71.5s	2.1s	1.6s	1.8s
$20 \times 20 \times 20$		25.5s	22.5s	22.8s
$30 \times 30 \times 30$		158.5s	133.5s	133.6s

For 10 cells in each spatial direction, it is clear that what was assumed based on the structure of the Jacobian in Figure 5.6 is true; The `ForwardDiff` method takes much longer time than `FAD`, `CJAD` and `MRST`. This also correspond to the result seen in Figure 4.1. Since `ForwardDiff` already spends over a minute for $10 \times 10 \times 10$ cells, it is omitted for finer discretizations. `MRST_diagonal` is not presented here because there is a bug that cause the method to fail for this specific example, and the only fix we have found makes the method very similar to `MRST`. According to the results in Table 5.1, the `CJAD` method is a bit faster than `FAD`, and it perform similar to `MRST` method.

It is, although, still some uncertainty in these numbers as we do not know if the linear solver in MATLAB and Julia perform equally well. In addition, another factor that can affect the result, is that for the current hardware these benchmarks are running on, the computations are so demanding that the hardware becomes warm. This can cause the performance of the hardware to decrease and ruin the benchmark results. To remove the uncertainty of both the linear solver and the hardware heating up, we can perform another benchmark. Instead of solving the linear system `f.jac\f.val` and continuing to the next Newton-Raphson iteration until it converges, and then onto a new time step, we only assemble the first residual function multiple times. In this way we only benchmark the AD part of the problem. The time the computer uses to calculate the residual function value and Jacobian will not depend on what type of numbers we have, only on the structure of the matrices in the Jacobians. Since the structure is the same throughout the whole simulation, this benchmark will still be relevant to figure out which AD-implementation is best to solve this specific problem. Since the benchmark does the same calculation over and over again, we need to be aware that it could have been the case that the compiler figured out that there is no need to perform all calculations, and that it is enough doing it only once. However, this is tested and is not the case. Table 5.2 shows the three methods assembling the first residual function of the simulation 100 times. It shows the

Table 5.2: Table with speed tests of different AD methods assembling the "Single-Phase Compressible AD Solver" residual function 100 times for different discretizations.

Number of cells	FAD	CJAD	MRST
$10 \times 10 \times 10$	0.9s	0.4s	0.6s
$20 \times 20 \times 20$	9.3s	4.0s	3.6s
$30 \times 30 \times 30$	44.2s	17.2s	16.5s

same tendency as Table 5.1, where `CJAD` and `MRST` are still considerable faster than `FAD`. However, in this benchmark, `MRST` is a bit faster than `CJAD` for the highest number of cells. There are however small differences and since there are some uncertainty when benchmarking computational speed on computers, a safe conclusion is that for this "Single-Phase Compressible AD Solver" example, the `MRST` and `CJAD` implementation perform equally good.

Chapter 6

Local Automatic Differentiation

This chapter will consider a different approach on how to use AD to solve PDEs. When solving PDEs using a finite volume method each cell will only depend on the neighboring cells. In the `FAD` and `CJAD` implementations, the dependencies are stored in the discrete gradient and divergence operators. The implementations calculate the residual function value and corresponding Jacobian for the whole grid simultaneously by having a vector to store the values and sparse matrices for the Jacobians. Hence, when the discrete gradient and divergence operators are used in the calculation of the residual function in chapter 5, the Jacobian are obtained with the structure seen in Figure 5.6. Local AD is an alternative approach, that instead of calculating the residual function for all cells at once, using matrix and vector operations, it iterates through the grid and for each cell calculates the local residual. In the end it will perform exactly the same calculations as `FAD` and `CJAD`, but instead of having large vector and matrix calculations, we now have smaller calculations concerning only one cell, and its neighbors, at the time. This new approach is based on the same idea as how AD is implemented in OPM, which, as said in introduction, is written in C and C++. The AD tool in OPM scales better when the simulations become larger and is the reason why it is used when creating efficient industrial simulators. Hence, it is interesting to see if you can write similar type of code in Julia and obtain better computational efficiency than what is achieved in MATLAB with MRST, as well as the methods I have already implemented in Julia. If this is possible, Julia could be a language where both prototyping and industrial simulators can be implemented, making the development process much more efficient.

6.1 Implementation

To get a better understanding of how local AD works, it is best to look at how it is implemented[1]. Like for `FAD` and `CJAD` I have implemented local AD using a struct called `LAD`:

```
struct LAD
    val::Float64
    derivatives::Vector{Float64}
end
```

[1] In the following I will explain a minor simplified implementation. This is to keep the focus on the essential parts of the implementation. The left out parts will be discussed in section 6.2

Since we now only operate on one cell at the time, the implementation of local AD is simpler than for `FAD` and `CJAD`. The value of the AD-variable is now only a scalar and the Jacobian matrices are replaced by a vector of derivatives. These are the derivatives with respect to the primary variables in the cell. For the single-phase flow solver in chapter 5, the derivatives vector will be of length one, as each cell only contain one primary variable (pressure). The implementation of `LAD` is, however, with `derivatives` as a vector. This is for the opportunity to implement more complex simulations like a two- or three-phase simulator, where each cell can contain multiple primary variables for the amount of water, oil and/or gas present in the cell (an example of a two-phase simulator can be seen in section 6.4).

To make sure that each primary variable is initialized correctly, initialization of `LAD` variables is done with the `createVariable` function. This function creates a `LAD` variable with a value and a given number of derivatives. The derivative with respect to itself is set to 1.0, whereas the other derivatives are zero.

```
function createVariable(val::Number, numDerivatives::Int, derivativeIndex::Int)
    derivatives = zeros(numDerivatives)
    if derivativeIndex > 0 && derivativeIndex <= numDerivatives
        derivatives[derivativeIndex] = 1.0
    end
    return LAD(val, derivatives)
end
```

The operators for the `LAD` struct are implemented similarly as explained in chapter 4 for `FAD` and `CJAD`, but since we now, instead of having a Jacobian matrix, only have a vector of derivatives, the implementation is easier and follows the lines of the description from subsection 2.1.4.

6.1.1 Flow Solver for Local AD

Whereas the implementation of the local AD tool is easier than for `FAD` and `CJAD`, there is more work when calculating the residual functions. Since we no longer can use discrete gradient and divergence operators, but instead have to traverse through the grid cell by cell, we need a place to store the resulting residual values and the corresponding Jacobian. We also need a method to traverse through all the cells and calculate the contributions from each neighboring cell. To create the flow solver from chapter 5, I have chosen to store the calculated values of the residual functions in another struct called `FlowSystem`:

```
struct FlowSystem
    eqVal::Vector{Float64}
    globalJac::SparseMatrixCSC{Float64, Int}
end
```

This struct looks very similar to the other AD structs, except from `globalJac` being one single sparse matrix instead of a vector of sparse matrices. At this point you might not understand why it is quicker to calculate the residuals cell by cell compared to everything in one go. The key reason for this lies in the `FlowSystem` struct. Since we know that the structure of the global Jacobian will stay the same, we can reuse this sparse matrix structure throughout the whole simulation. Before the simulation begins, we use the grid variable `G`, which contains the information on which cells are neighbors, to build the correct structure of `globalJac` once and for all. When we later run the simulation, we only change the values inside of `globalJac`, but the structure stays the same. This reuse of `globalJac` will save a lot of memory allocations and hence consume less CPU time compared to `FAD` and `CJAD`,

which both allocate new structs for each calculation. By creating a new constructor for `FlowSystem` that uses the grid variable `G`, the variables `eqVal` and `globalJac` will be allocated with the correct length and structure before the simulation begins. For the grid in chapter 5, we have 1,000 cells. This implies 1,000 different pressure values and in addition we have the bottom-hole pressure (`bhp`) and the total production (`qS`). Hence, `eqVal` will be a vector of length 1,002.

Now that `FlowSystem` stores the values and Jacobian of the residual functions, we need a new function to traverse through all cells and perform the calculations. The function that will execute the calculations is `assembleFlowSystem!()`, where the exclamation mark is a Julia convention for a function that modifies its input parameters. The corresponding code can be seen below. I have removed all declarations of help variables and replaced the code for updating `FlowSystem` with comments to highlight the important parts of the function structure.

```
1  function assembleFlowSystem!(fs::FlowSystem, well::Well)
2      resetFlowSystem!(fs)
3      for fromCell = 1:length(fs.eqVal)
4          for toCell in neighbours
5              if fromCell == toCell && fromCell in gridCell
6                  # Add backward Euler term to FlowSystem
7              elseif fromCell in well && toCell in well
8                  # Add well equations to FlowSystem
9              else
10                 # Add flux to FlowSystem
11             end
12         end
13     end
14 end
```

The input parameter `well` is a struct that contains all necessary information about the well. Line number two in `assembleFlowSystem!()` resets `eqVal` and `GlobalJac` such that the structures are unchanged, but all the values are set to zero. Then, the function starts traversing through the grid and for every cell it adds up the contributions from all neighboring cells. Be aware that the looping variable names `fromCell` and `toCell` can be a bit misleading when they represent bottom-hole-pressure and total production, as those primary variables do not belong to any cell. The intuitiveness of considering the flow from one cell to another is however considered more valuable than the issue that they will also represent bottom-hole-pressure and total production. The `eqVal` and `globalJac` variables are updated in line number six, eight and ten inside the inner loop. As a reminder, the residual functions that `FlowSystem` eventually will represent are the functions `presEq`, `rateEq` and `ctrlEq` defined in section 5.2:

```
presEq(p,p0,dt) = (1/dt) * (pv(p)*ρ(p) - pv.(p0).*ρ.(p0)) +
                  dDiv(average(ρ(p))*flux(p))
rateEq(p,bhp,qS) = qS - sum(q_conn(p,bhp))/rhoS
ctrlEq(bhp) = bhp - 100*barsa
```

In line number six, `fromCell` and `toCell` are equal, but do not reference the bottom-hole-pressure or total production. Here, the first term in `presEq`, or the backward Euler term, is calculated. This is performed in an outer function called `timeDerivative()` that returns a `LAD` struct:

```
function timeDerivative(p::Float64, p0::Float64, dt::Float64)
    pCell = createVariable(p,1,1)
    return (1/dt) * (pv(pCell)*ρ(pCell) - pv(p0)*ρ(p0))
end
```

In FAD and CJAD, we made all the primary AD-variables before the simulation. We then used them as input parameters in the residual functions that returned new AD-variables which represented the values and Jacobians. With local AD, we create new primary AD-variables for the applicable cell inside the function we want to evaluate. For timeDerivative() we create a LAD primary variable that represents the pressure in the cell before we calculate the backward Euler term. When timeDerivative() has returned the new LAD variable, assembleFlowSystem!() adds the calculated value to the correct index in eqVal and the derivative value to the correct diagonal index in globalJac.

In line number ten, when fromCell and toCell are two neighboring cells, the divergence term in presEq is calculated. Like for timeDerivative(), we create the primary AD-variables inside the function, but since we calculate the flux from one cell to another, we have to be careful with which direction we calculate the flux and which cell we want the derivative with respect to. Since the varying variables are named fromCell and toCell, it is natural that the function flux(), seen below, calculates the flux from fromCell to toCell.

```
function flux(fromCell, toCell)
    pFrom = createVariable(pressure[fromCell], 1)
    pTo = createVariable(pressure[toCell], 0)
    ρAvg = avg(ρ(pFrom), ρ(pTo))
    viscousFlux = -T/mu * (grad(pFrom, pTo))
    gravityFlux = T/mu * g * ρAvg * gradz
    return ρAvg * (viscousFlux + gravityFlux)
end
```

What needs to be chosen is which cell we want the derivative with respect to. The choice only affects which indices of the Jacobian the calculated derivatives should be added or subtracted to. In flux(), I have decided to calculate the derivative with respect to fromCell. The two first lines show the consequence of this, where pFrom is initialized with a derivative of 1 and pTo as a constant. This choice leads to the following code for updating FlowSystem at line number ten in assembleFlowSystem!():

```
fluxLAD = flux(fromCell, toCell)
fs.eqVal[fromCell] += fluxLAD.val
fs.globalJac[fromCell, fromCell] += fluxLAD.derivatives[1]
fs.globalJac[toCell, fromCell] -= fluxLAD.derivatives[1]
```

The value of fluxLAD is added to eqVal and the derivative of the flux with respect to fromCell is added to globalJac. In addition we know that the flux from fromCell to toCell is the same as from toCell to fromCell, but with negative sign. This means that the derivative of the flux from toCell with respect to fromCell needs to be subtracted with fluxLad.derivatives. For a facet between two neighbors, we will with assembleFlowSystem!() calculate the value of the flux through the facet twice, only with different signs. You might think that we could save time by subtracting fs.eqVal[toCell] with fluxLAD.val, but since we also will need the opposite derivatives of this particular flux, there will be small, or next to none, computational gain of exploiting this fact. In worst case it might actually become a slower implementation, as we would have had to keep track of which fluxes have been added to which cells. When the fluxes from all the neighbors have been added up for a cell, we have obtained the divergence in that cell.

Line number eight in assembleFlowSystem!() is the last line I have not commented and it is where the well equations, rateEq and ctrlEq, are handled. This line is executed if both fromCell and toCell either refer to one of the cells containing a well, the bottom-hole pressure, or the total

production. The information on which cells fulfilling this condition lies in `well`. The calculation is performed by handling every case such that the correct residual functions, namely the `rateEq` or the `ctrlEq`, are calculated and added to the correct indices in `FlowSystem`.

Summed up, the calculation of the residual functions using local AD is based on two modules. The first is the actual AD tool with the `LAD` struct, and the second is `FlowSystem`, which keeps track of the global system and what calculations should be performed by the AD tool. With this approach, we loose some of the advantages using the other AD tools where the discrete residual functions looked very similar to the continuous case, like explained in section 5.2. The main structure of `assembleFlowSystem!()` will, however, be the same no matter what type of simulation, hence making modifications to the code, or building another simulator, will not demand a complete change of the code structure.

Now that the assemble of the governing equations are implemented, we need a main loop that performs the simulation. Since we have separated the assemble of the equations into a single function, the main loop will be very similar to the one outlined in section 5.2. The only difference is that instead of evaluating a residual function `F`, we assemble the equations using `assembleFlowSystem!()`. A pseudo code of the new main loop can be seen below.

```
## Define FlowSystem(fs), length of simulation and timestep
while timeNow < endTime
    while ## Newton-Raphson method has not converged.
        assembleFlowSystem!(tps, well)
        upd = -(fs.globalJac\fs.eqVal)
        # Update pressure, bottom-hole pressure and total production
    end
    timeNow += dt
end
```

6.2 Optimizing Local AD

As explained in the beginning of this chapter, the main idea behind local AD is to reuse the Jacobian to save memory usage and extra memory allocations. The implementation given in section 6.1 has nevertheless left out some small, but key parts of the implementation that will massively decrease the memory and CPU usage. This is purely implementation specific differences that have been left out to make the introduction to local AD more clear. It will not change the theory behind using local AD to solve PDE's.

6.2.1 Dynamic VS Static Arrays

The difference between a dynamic array and a static array is that a dynamic array can be extended or shortened as much as you like while a static array has a fixed length. Using a static array will give computational gain concerning speed and memory allocations compared to a dynamic array. This is because the compiler knows that a static array has the given fixed length forever and hence it can allocate the exact memory needed in advance and optimize the machine code accordingly. With a dynamic array, the compiler does not know if the array will grow larger, or become smaller, and it is much harder to optimize the machine code.

Julia has a package called *StaticArrays.jl (n.d.)* that provides static arrays with the same abilities as the built-in arrays, but without the possibility for changing the size of the array. The package provides

two types of vectors, `SVector` and `MVector`(for two-dimensional arrays it has the corresponding `SMatrix` and `MMatrix`). Both are static vectors, but `MVector` is mutable, meaning the values inside the vector can be changed, whereas the values inside an `SVector` are final once the vector is defined. This makes `SVector` ideal for `LAD.derivatives`, since when we perform operations on `LAD` variables, new `LAD` variables are returned.[2]

When the implementation is changed to use static vectors, it needs to know how long these vectors are. When building a simulator, it is not a problem to say in advance how many derivatives we need, and once the compiler knows this before compilation, it can optimize the machine code. OPM solves this by creating a template class containing the length of the gradient vector *(Lauser et al., 2018)*. This ensures that the compiler always knows that for this simulator, the static vector will always have the given fixed length. This is important because we do not want to create `LAD` structs with different lengths for `LAD.derivatives`. In addition, if we define this clearly, the compiler knows exactly how much memory is needed when we allocate a new `LAD` variable. By giving the compiler this information before compilation, we help it to optimize the machine code. This will also make the implementation of the operators of `LAD` easier, as we know for certain that all `LAD` variables will have `derivatives` vectors with the same length. The best implementation I have found for this in Julia is to declare a global constant variable in the local AD module such that the new implementation of `LAD` becomes:

```
using StaticArrays
const NUM_DERIV = 1
struct LAD
    val::Float64
    derivatives::SVector{NUM_DERIV, Float64}
end
```

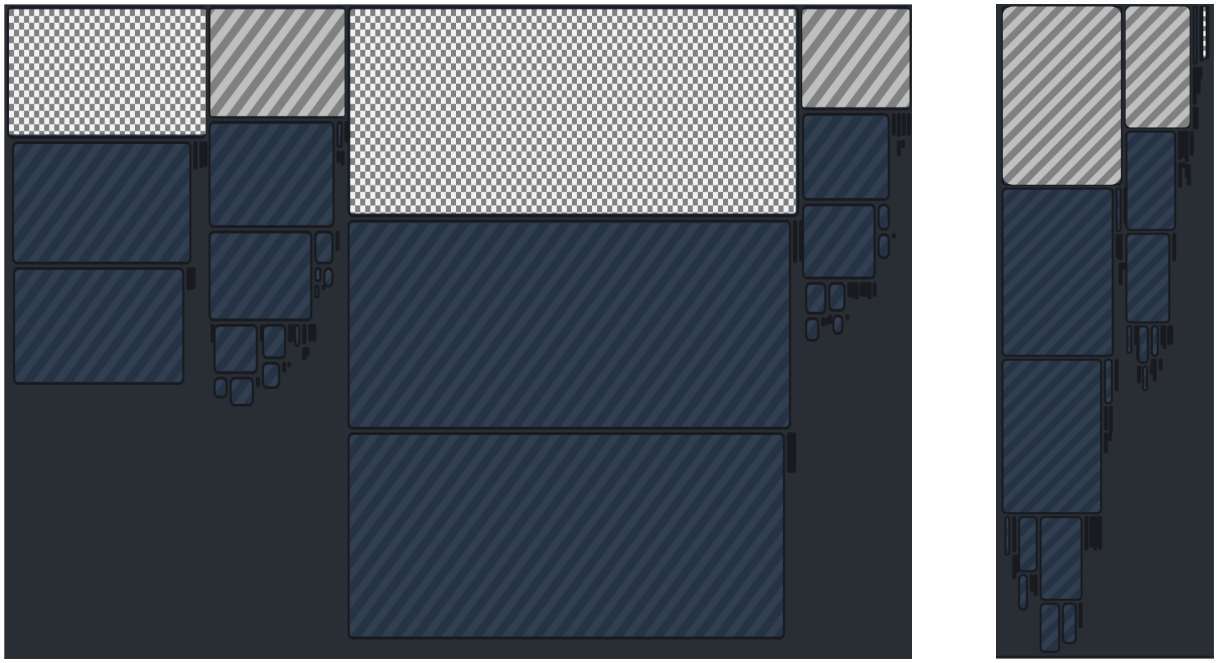
The disadvantage with this implementation is that the Local AD module needs to be modified to work for simulations with more primary variables for each cell. Since the implementations of the operators are unchanged, this modifications will only consist of changing the value of `NUM_DERIV`. However, it would be better if Julia had the opportunity to pass in an argument into the `LAD` module. In this way the module could be compiled with any given value for `NUM_DERIV`. As it is now, it is not possible to have one single `LAD` module and two different simulators with different number of primary variables, at the same time, in the same directory. A workaround could be to have multiple `LAD` modules with the only difference being the value of `NUM_DERIV`. This would be a fully functional workaround, but not a very elegant one.

6.3 Flow Solver Results for Local AD

Now that `LAD` has been implemented with static vectors, it is interesting to see how the local AD method compares to `FAD` and `CJAD` in the simulation from chapter 5. Since the local AD technique from OPM, unlike the vectorized methods from MRST, is constructed with a main focus on computational efficiency, and since it is also assumed that Julia is supposed to run as fast as C, we expect that the local AD approach will give significant computational gain. This is unfortunately not the case with the implementation outlined in this chapter. Indeed, `LAD` performs worse than `CJAD` and similar to `FAD`.

[2]For a multiphase simulation, a possible implementation can be each element in the global Jacobian being an `n×n` matrix, where `n` is the number of primary variables in each cell. For this particular case, an `MMatrix` is optimal to use as we know the size of the matrices, but we want to change the values inside the matrix.

As explained in subsection 3.3.1, profiling is an efficient way of finding the bottlenecks in a code, and this is a perfect example of when to use it - the code was expected to be faster, but it was not, so it is important to figure out if there is a small part of the code that is slow. Figure 6.1a shows a screenshot of a profile result from running the flow solver simulation. The screenshot only contains blocks from the `flux()` function. Two blocks are marked with squares and two other blocks with brighter diagonal stripes. The blocks with squares are time spent in `createVariable()` and the brighter diagonal stripes are AD calculations in `flux()`.



(a) **(b)**

Figure 6.1: The figures shows screenshots of the profile result from `flux()` in the flow solver simulation from chapter 5. The blocks marked with squares are time spent in `createVariable()` and the blocks with bright diagonal lines are calculations in `flux()`. Figure 6.1a uses a slow version of `createVariable()` while Figure 6.1b uses a faster version.

It is surprising that creating a `LAD` struct should take more time than the actual AD calculations. This indicates that the `createVariable()` function is poorly implemented. If we look closer at the implementation, we see that we only want to create a static `Svector`, but we start by creating a dynamic vector of zeros, and then we modify the vector by changing one of the values to one. Finally, we convert the dynamic vector into a static `SVector` in the last line. The new implementation seen below is a Julia specific implementation, where the `SVector` is created immediately with correct values and without any use of dynamic vectors.

```
function createVariable(val::Number, derivIx::Int)
    derivatives = @SVector [if i==derivIx 1 else 0 end for i = 1:NUM_DERIV]
    return LAD(val, derivatives)
end
```

The new profiling result with the updated `createVariable()` can be seen in Figure 6.1b. The time spent in `createVariable()` is almost eliminated and we can see that approximately all the time spent in `flux` is used for AD calculations. This is what is expected as initialization should be far less time consuming than AD calculations.

As discussed in chapter 5, by only looking at the assemble of the residual functions, and not the full simulation, we eliminate the linear solver from the benchmark, and we get a better understanding of how efficient the AD tools are. Table 6.1 is the same table as Table 5.2, assembling the residual functions 100 times, but with the measured time for Local AD added to the end of the table. The table

Table 6.1: Table with speed tests of different AD methods assembling the "Single-Phase Compressible AD Solver" residual function 100 times for different discretizations.

Number of cells	FAD	CJAD	MRST	Local AD
$10 \times 10 \times 10$	0.9s	0.4s	0.6s	0.07s
$20 \times 20 \times 20$	9.3s	4.0s	3.6s	0.6s
$30 \times 30 \times 30$	44.2s	17.2s	16.5s	2.2s

shows that the local AD approach is at least six times faster than the other tested implementations, which is a significant improvement. This gives us an indication that Julia can be a language where we can quickly create new simulators using the high-level and user-friendly `CJAD` or a similar AD tool, and in the same language create more efficient simulators using a method like local AD. To confirm this, however, we need to benchmark the performance for a larger problem than a single-phase pressure solver. In section 6.4, I will test how local AD performs in Julia solving a two-phase problem with a more realistic grid structure. This will give a better indication on how well Julia is suited for creating high performance simulators.

6.4 Two-Phase Incompressible Flow

Previously in this thesis, I have only looked at primary production, where we have single-phase flow, with oil as the only fluid present. As explained in the introduction, at most 30% of the oil in the reservoir will be extracted unless we apply external forces to the reservoir. This normally consists of injecting either water, gas, or both, into the reservoir in a secondary production. There is also a third production phase called enhanced oil recovery or tertiary production, but as the purpose of this thesis is not oil recovery, but to see how the AD tools perform when simulating oil recovery, I will not go into this subject. More detailed information about oil recovery can be found in Lie (2019). I will also describe a simplified model of secondary production, where sufficient information for this example is that we want to simulate how water flows from an injector and into a reservoir full of oil. This will be simulated for a subregion of Model 2 from the 10th SPE Comparative Solution Project *(Christie and Blunt, 2001)*. This model represents a simplified reservoir geometry with somewhat exaggerated geological heterogeneity and was created as a challenging case to benchmark upscaling methods against each other. Like in chapter 5, I have called MATLAB and MRST from Julia to initialize the SPE10 grid and to make plots.

The SPE10 (Model 2) grid is split into 85 layers. Each layer contains 60×220 cells, making the total number of grid cells 1,122,000. To simplify the implementation, I will only look at a single layer, layer 5, making the problem two-dimensional with 13,200 cells. The SPE10 grid differs from the grid in chapter 5 in that the permeability and porosity varies throughout the grid. This will affect the path of the water flow. Figure 6.2 shows the porosity in the grid and Figure 6.3 shows a logarithmic color plot of the permeability. Both figures also display the placement of the water injector in the center of the grid, and the four wells at each corner. Figure 6.2 shows that the rock around wells P3 and P4 are very dense (low porosity) and Figure 6.3 naturally shows that the permeability is also low in this area. This gives us an expectation that the injected water will primarily flow to the left.

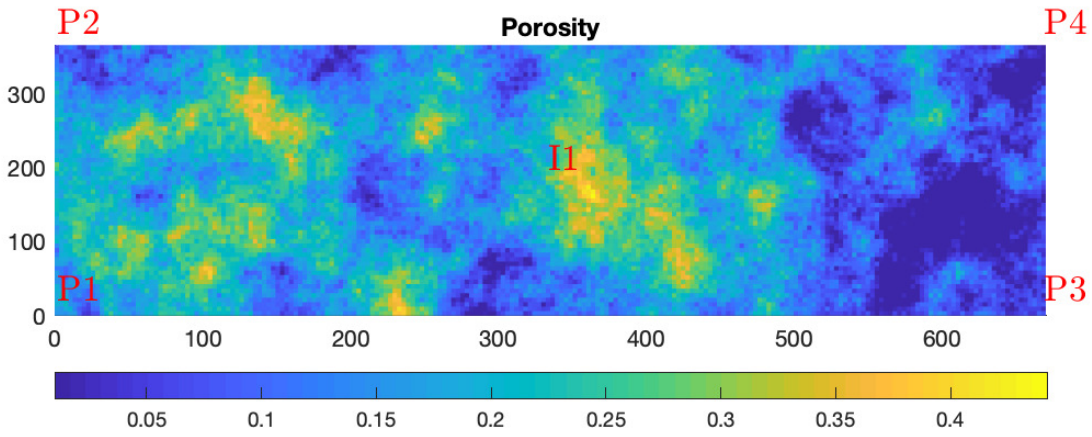


Figure 6.2: Color plot of the porosity in the SPE10 grid.

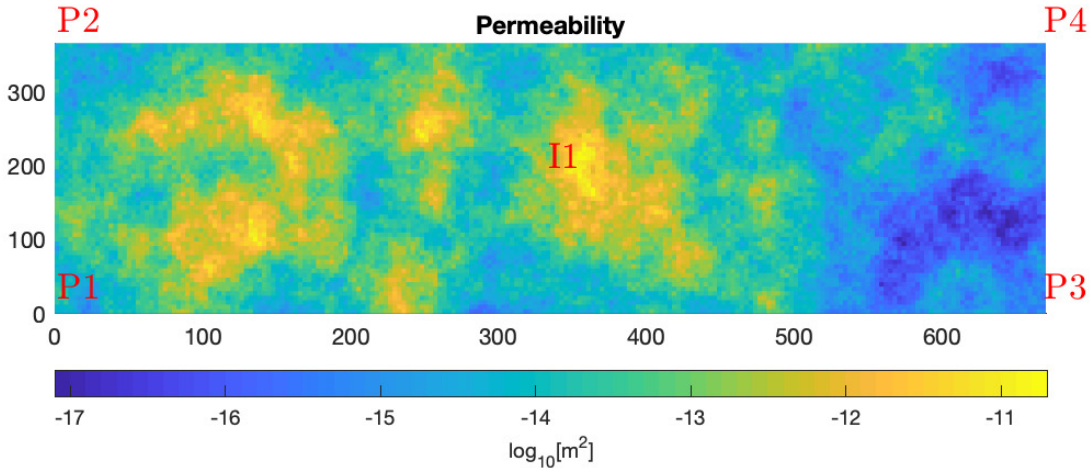


Figure 6.3: Logarithmic color plot of the permeability in the SPE10 grid.

6.4.1 Derivation of the Governing Equations for Two-Phase Flow

The governing equations for a two-phase flow are similar to the ones expressed in chapter 5, with some modifications. I will also make some simplifications to the model that I will note in the following derivation. Darcy's law, expressing the volumetric flow rate of a single-phase fluid, is given in Equation (5.3), but since our grid only has one layer, we neglect the gravity, and Darcy's law becomes

$$\mathbf{v} = -\frac{K}{\mu}\nabla p.$$

However, now we need to separate between the flow of water, flow of oil and their overall flow. Before injection of water, each cell is only filled with oil. This implies that the amount of oil in a cell equals the pore volume in that cell. Since our grid only contain one layer, we can find the pore volume by simply multiplying the total volume of the cell with the porosity, a number between zero and one. The volumetric fractions of water and oil, called the saturation, are defined similarly as the porosity as pointwise quantities, subject to the condition

$$\sum_{\alpha} S_{\alpha} = 1,$$

stating that the fluids fully occupy the available pore volume. In this two-phase situation, subscript α refers to the phase and can be either o for oil or w for water. As an example, in a cell without water, the oil saturation will be $S_o = 1$ and water saturation $S_w = 0$. The effective permeability a fluid phase experiences when flowing through a medium partially occupied by another mobile fluid phase can be significantly lower than if the phase flows alone. The reduction is generally not linear in S and hence we cannot find the phase flux by simply multiplying Darcy's law with the saturation. To get an expression for the phase flux, we introduce the property relative permeability. Relative permeability is often represented as a tabulated quantity, but herein I have chosen to use one of the simplest commonly used analytic models, given by

$$k_{r\alpha} = S_\alpha^2,$$

for both water and oil. The flux for phase α is then given by the modified Darcy's law

$$\mathbf{v}_\alpha = -\frac{K k_{r\alpha}}{\mu_\alpha} \nabla p_\alpha = -K\lambda_\alpha \nabla p_\alpha.$$

Here, p_α is the pressure for the given phase and λ_α is called the mobility of phase α. The difference between oil and water pressure is given by the capillary pressure, $p_c = p_o - p_w$. For an oil–water system, the capillary pressure would cause water to rise in the reservoir. However, in this model we neglect capillary pressure so that the oil- and water pressure are considered to be equal. Now that we have an expression for the phase flux, we obtain the conservation equation for each phase:

$$\frac{\partial}{\partial t}(\phi \rho_\alpha S_\alpha) + \nabla \cdot (\rho_\alpha \mathbf{v}_\alpha) = \rho_\alpha q_\alpha. \tag{6.1}$$

If you remove α from Equation (6.1), it becomes the same equation as Equation (5.4), except that in Equation (5.4), ρ is inside of the source term q. In our two-phase example we assume both phases to be incompressible such that the governing equations are given by

$$\frac{\partial}{\partial t}(\phi S_\alpha) + \nabla \cdot \mathbf{v}_\alpha = q_\alpha, \quad \alpha \in o, w. \tag{6.2}$$

Since $S_o = 1 - S_w$, the equations from Equation (6.2) contain only two unknown variables, the saturation, for either water or oil, and the pressure. This means that we have enough equations to solve for the unknown variables and we could solve them simultaneously. However, the equations are strongly coupled, and for this example we will rather reformulate the two equations as an elliptic equation describing the pressure and a hyperbolic equation describing the water saturation, since these reformulated equations are less coupled. This gives us the opportunity to solve the equations separately using an operator splitting method I will explain later. We start by obtaining an equation for the pressure. By adding the two equations from Equation (6.2) we obtain

$$\phi \frac{\partial}{\partial t}(S_o + S_w) + \nabla \cdot (\mathbf{v}_o + \mathbf{v}_w) = q_o + q_w.$$

Since $S_o + S_w = 1$, we can remove the time dependent term, and by defining the total flux as

$$\mathbf{v} = \mathbf{v}_o + \mathbf{v}_w = -K\lambda_o \nabla p - K\lambda_w \nabla p = -K\lambda \nabla p,$$

and the total source term as $q = q_o + q_w$, we obtain the pressure equation

$$-\nabla \cdot (K\lambda \nabla p) = q. \tag{6.3}$$

Next, we derive the equation for the water saturation. Equation (6.2) with $\alpha = w$ gives one formulation, but we want an expression that depends on the total flux, and not the water flux. The reason for this will be explained later. To find a relation between the water flux and the total flux,

observe that the difference between the water flux, multiplied with the mobility of oil, and the oil flux, multiplied with the mobility of water, is equal to zero.

$$\begin{aligned}\lambda_o \mathbf{v}_w - \lambda_w \mathbf{v}_o = \lambda \mathbf{v}_w - \lambda_w \mathbf{v} &= (\lambda_o + \lambda_w)\mathbf{v}_w - \lambda_w(\mathbf{v}_o + \mathbf{v}_w) \\ &= \left(\lambda_o \lambda_w + \lambda_w^2 - (\lambda_w^2 + \lambda_w \lambda_o)\right)\left(-K\nabla p\right) = 0.\end{aligned}$$

This gives us the relation

$$\mathbf{v}_w = \frac{\lambda_w}{\lambda_o + \lambda_w}\mathbf{v} = f_w \mathbf{v},$$

where f_w is called the fractional water flow. Inserting the expression for water flux into the conservation law for the water phase, given in Equation (6.2), we obtain the transport equation

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \left(f_w \mathbf{v}\right) = q_w. \tag{6.4}$$

Even though we now have separated the governing equations into a pressure equation and a transport equation, they are still coupled. Equation (6.3) is depending on the water saturation in the total mobility λ and Equation (6.4) is depending on the pressure in Darcy's law for the flux, $\mathbf{v}$. However, this coupling is weaker than in Equation (6.2), and this weaker coupling opens up the opportunity to use an operator splitting method. This consist of first solving the pressure equation, holding the saturation constant. Then, when we have found the correct pressure, we hold the pressure and the total flux constant for a (short) time step and solve the transport equation. Since the total flux is held constant as well when solving the transport equation, the total flux calculated in the pressure equation can be used to solve the transport equation. This is why we wanted to find an expression for the water flux, given by the total flux. When the water saturation is updated, we continue with the next time step by solving the pressure equation for constant saturation, and so on.

6.4.2 Implementation of a Two-Phase Solver

In the implementation of the two-phase simulator for the SPE10 grid, a lot will be very similar as for the previous single-phase implementation from subsection 6.1.1. The `FlowSystem` struct is replaced with a `TwoPhaseSystem` struct that hold the residuals and Jacobians for both equations. In addition to this, for implementation convenience purposes, it also holds the current pressure and water saturation.

```
struct TwoPhaseSystem
    pressure::Vector{Float64}
    waterSaturation::Vector{Float64}

    pressureEq::Vector{Float64}
    pressureEqJac::SparseMatrixCSC{Float64,Int}

    transportEq::Vector{Float64}
    transportEqJac::SparseMatrixCSC{Float64,Int}
end
```

Naturally, instead of having a single function to assemble equations, the simulator now has two. These are called `assemblePressureEquations!()` and `assembleTransportEquations!()`, and are implemented with the same structure as `assembleFlowSystem!()`, only that the equations calculated are different. The main loop that controls the simulation is also extended. As explained above, the pressure equation and the transport equation are solved interchangeably and the implementation of this can be seen below.

```
## Define TwoPhaseSystem(tps), length of simulation, endTime and timestep dt.
while timeNow < endTime
    while ## Newton-Raphson method has not converged.
        assemblePressureEquations!(tps, well)
        upd = -(tps.pressureEqJac\tps.pressureEq)
        tps.pressure += upd
    end
    prevWaterSaturation = copy(tps.waterSaturation)
    while ## Newton-Raphson method has not converged.
        assembleTransportEquations!(tps, prevWaterSaturation, dt, well)
        upd = -(tps.transportEqJac\tps.transportEq)
        tps.waterSaturation += upd
    end
    timeNow += dt
end
```

6.4.3 Two-Phase Solver Results

Even though we are most interested to see how the AD tools perform, we also have to see what we are simulating to make sure that it is correct implemented. Figure 6.4 shows the development of the water saturation in the SPE10 grid, for a simulation over three years. The total amount of water we have injected after three years is 0.135 of the total pore volume in the reservoir. As we can see from the figure, the assumption we had that the water will flow to the left was correct. We can also see that it follows paths in the reservoir that are optimal for the water to flow. One example that is easy to spot, is that the water, at least in the beginning, avoids the area straight left of the injector. If we go back to Figure 6.2 and Figure 6.3, we can see that in this area, both the porosity and the permeability are low.

To benchmark the AD performance in the two-phase simulation, the AD calculations are separated from the linear solver, similar to the previous tests. The simulation in Figure 6.4 used approximately 150 time steps, and for each time step, the Newton–Raphson method used two iterations to solve the pressure equations and approximately six iterations to solve the transport equations. The benchmark is essentially the same simulation, but without the linear solver and the update of the pressure and water saturation. This will give us a good estimation on how much time the simulator spend on AD during the simulation. As mentioned in section 5.3, since the benchmark does not update the pressure and water saturation, we need to double check that the compiler does not understand that it can skip iterations, since it calculates the same thing over and over again. It was not the case in section 5.3, and it is not the case here. The benchmark is performed for the local AD-, `CJAD`- and `MRST` method. The results of the benchmark can be seen in Table 6.2. The table shows that `CJAD` and `MRST` perform similar, with approximately 30 seconds in total. The local AD outperforms both of the two other methods, and for this specific test, it is approximately five times as fast. This result

Table 6.2: Table with speed tests of different AD methods assembling the two-phase incompressible flow residual equations. The number of equations assembled corresponds to what was performed in the simulation for Figure 6.4.

Local AD	CJAD	MRST
5.89s	31.97s	30.15s

gives us an even stronger indication that Julia is a language that can be used both to create quick simulators using a high-level and user-friendly AD method, like what is implemented in MRST, as well as to create more performance strong simulators, using low-level implementations like in OPM.

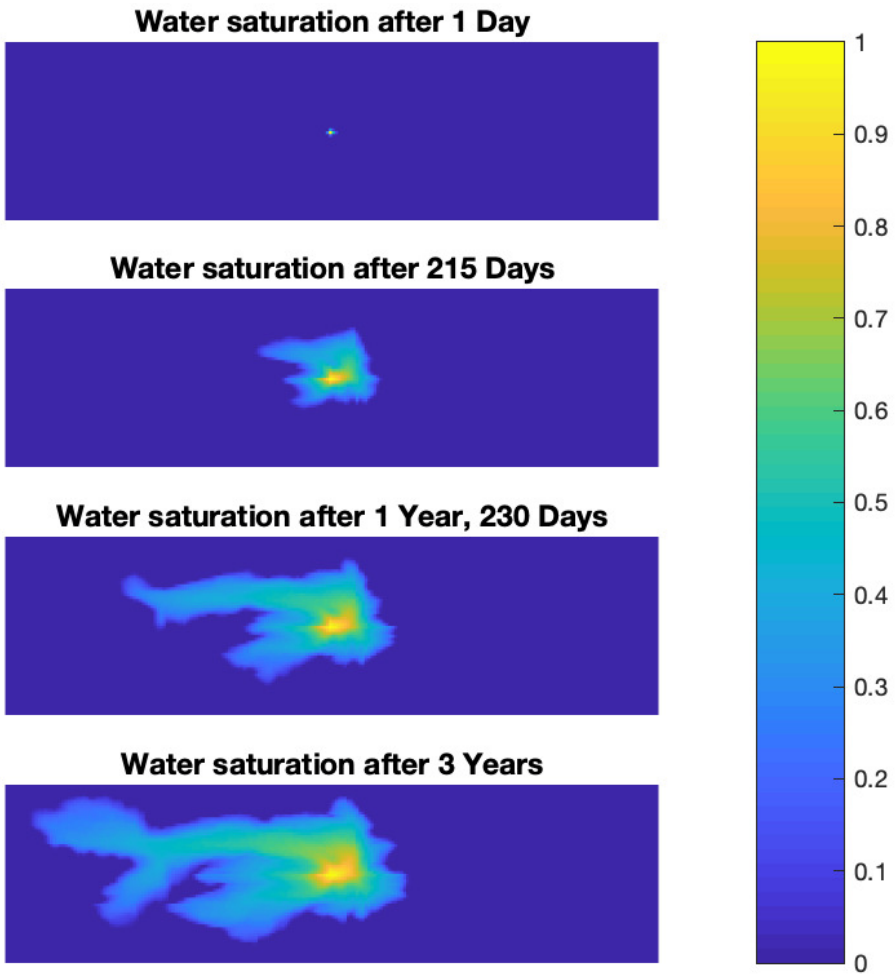


Figure 6.4: Color plot of the water saturation development in the SPE10 grid over a period of three years. Total injected water after three years is 13.5% of the total pore volume in the reservoir.

Chapter 7

Conclusion and Further Work

This thesis has investigated the possibility of using the new programming language, Julia, as a language both for implementing prototypes of new oil reservoir simulators, as well as building efficient industrial simulators.

The process of creating an industrial simulator usually consists of first creating a prototype and subsequently developing a more comprehensive industrial simulator based on the prototype model. SINTEF's approach is to implement the prototype and the finished simulator in two different languages: a high-level scripting language for the prototype, and a lower-level compiled language for the industrial simulator. The possibility of performing the entire implementation in Julia may contribute to significantly increase the efficiency of developing new computational tools.

To investigate this, the thesis has explored solving Partial Differential Equations (PDEs) that describe flow in porous media, using AD and a finite-volume method. The PDEs are solved by discretizing the gradient- and divergence operator so that the discrete equations can be implemented in almost the same form as the continuous formulas. By setting up the equations as a vector function in residual form, the system can be solved using AD to obtain the Jacobian of the system and the Newton–Raphson method to find the roots of the function. The focus of the thesis has been to examine the performance of the AD tools. Three new AD implementations in Julia have been presented and compared to a third-party implementation in Julia and AD tools from MRST.

The AD libraries were benchmarked against each other in a prototype of a single-phase flow solver, simulating primary production from a reservoir with a single well producing oil. The third-party implementation of AD in Julia, with its dense matrix structure, proved ineffectual for calculating the sparse Jacobians corresponding to the residual functions. For this simulation, it was more than 30 times slower than the other AD tools and was eliminated for further consideration. The first two implementations in Julia are called `ForwardAutoDiff(FAD)` and `CustomJacobianAutoDiff(CJAD)`. These are high-level AD tools that are easy to use and that are based on the implementation in MRST. `CJAD` is an extension of `FAD`, where `FAD` only uses sparse matrix structure for its Jacobians, `CJAD` makes optimized calculations when the Jacobian is a diagonal-, identity-, or null matrix. For the single-phase simulation, `CJAD` and MRST performed similarly, while `FAD` was approximately twice as slow.

The last AD tool in Julia is called local AD and was implemented motivated by the properties of the single-phase simulation as well as the way AD is implemented in OPM. Hence, local AD is a lower-level implementation than `FAD` and `CJAD`, and since OPM is the tool SINTEF recommends for creating efficient industrial simulators, it was expected that this would offer an improved

performance. For the single-phase solver, local AD was indeed approximately six times faster than `CJAD` and MRST. To further test the capabilities of local AD in Julia, a second simulator was implemented. This was a two-phase solver, simulating the flow of water in a single layer of the SPE10 model 2 reservoir, when injecting water into the center of the reservoir. For this simulation, `CJAD` and MRST continue to exhibit similar performance, while the local AD method was approximately five times faster.

The implementation and benchmarks of `CJAD` indicate that Julia is well suited for making quick prototypes of simulators, and the benchmarks of local AD provide good indications that it may also be possible to create efficient industrial simulators for oil recovery. However, the local AD implementation has only been tested on a grid with less than 15,000 cells and not on a fully realistic simulation model. As a point of reference, the full SPE10 model 2 grid has 1,122,000 cells. A natural next step in testing Julia is to make an implementation in MATLAB, similar to local AD, and compare the two languages. Since local AD depends on fast execution of for-loops, it is not expected that MATLAB will manage to execute local AD efficiently, but it is a good test to see if Julia actually is a step forward compared to MATLAB.

Macro functions to parallelize for-loops are currently under development in Julia. It would be easy to insert these into the functions that assemble the equations in local AD. This has already been attempted, but the macro functions are too unstable at the moment. When these functions become more stable, it will be interesting to see if, and possibly by how much, this can improve the current implementation. A final investigation will necessarily be to confirm whether Julia can match the computational efficiency of OPM, or other industrial simulators, in a full-scale realistic simulation.

Bibliography

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., Siskind, J. M., 2018. Automatic differentiation in machine learning: a survey. Accessed 21.11.2018.
URL http://jmlr.org/papers/volume18/17-468/17-468.pdf

Beda, L., Korolev, L., Sukkikh, N., Frolova, T., 1959. Programs for automatic differentiation for the machine besm. Inst. Precise Mechanics and Computation Techniques, Academy of Science, Moscow.

BenchmarkTools, n.d. Accessed 21.12.2018.
URL https://github.com/JuliaCI/BenchmarkTools.jl

Bezanson, J., Karpinski, S., Shah, V., Edelman, A., 2012. Why we created julia. Accessed 21.11.2018.
URL https://julialang.org/blog/2012/02/why-we-created-julia

Boltyanskii, V. G., Gamkrelidze, R. V., Pontryagin, L. S., 1960. The theory of optimal processes. i. the maximum principle. Tech. rep., TRW SPACE TECHNOLOGY LABS LOS ANGELES CALIF.

Christie, M., Blunt, M., 2001. Tenth spe comparative solution project: A comparison of upscaling techniques. SPE Reservoir Evaluation & Engineering 4 (04), 308–317.

Griewank, A., et al., 1989. On automatic differentiation. Mathematical Programming: recent developments and applications 6 (6), 83–107.

Holy, T., Carlsson, K., Pfitzner, S., Fischer, K., 2019. A julia interpreter and debugger. Accessed 12.04.2019.
URL https://julialang.org/blog/2019/03/debuggers

Innes, M., 2018. Don't unroll adjoint: Differentiating ssa-form programs. arXiv preprint arXiv:1810.07951.
URL https://github.com/FluxML/Zygote.jl

Johnson, S. G., 2017.01.21. Accessed 13.03.2019.
URL https://julialang.org/blog/2017/01/moredots

Julia Community, 2018. Julia 1.0. Accessed 09.04.2019.
URL https://julialang.org/blog/2018/08/one-point-zero

Julia issue:dot operators, 2017. Accessed 02.12.2018.
URL https://github.com/probcomp/GenExperimental.jl/issues/46

Juno, n.d. Accessed 11.04.2019.

URL http://junolab.org

Juno profiler, n.d. Accessed 12.04.2019.
URL http://docs.junolab.org/latest/man/juno_frontend/#Profiler-1

Juno's Github page, n.d. Accessed 11.04.2019.
URL https://github.com/JunoLab

Lauser, A., Rasmussen, A., Sandve, T., Nilsen, H., 2018. Local forward-mode automatic differentiation for high performance parallel pilot-level reservoir simulation. In: ECMOR XVI-16th European Conference on the Mathematics of Oil Recovery.

Lie, K.-A., 2019. An Introduction to Reservoir Simulation Using MATLAB/GNU Octave: User guide for the MATLAB Reservoir Simulation Toolbox (MRST). Cambridge University Press.

Lisp language, n.d. Accessed 13.05.2019.
URL https://lisp-lang.org

MATLAB.jl, n.d. Accessed 22.11.2018.
URL https://github.com/JuliaInterop/MATLAB.jl

MRST Homepage, n.d. Matlab reservoir simulation toolbox.
URL https://www.sintef.no/projectweb/mrst

Neidinger, R., 2010. Introduction to automatic differentiation and matlab object-oriented programming. SIAM Review 52 (3), 545–563.
URL https://doi.org/10.1137/080743627

Nolan, J. F., 1953. Analytical differentiation on a digital computer. Ph.D. thesis, Massachusetts Institute of Technology.

Open Porous Media Hompage, n.d. Accessed 22.11.2018.
URL https://opm-project.org

Parallel computing Julia docs, n.d. Accessed 29.05.2019.
URL https://docs.julialang.org/en/v1/manual/parallel-computing/index.html

Profile.jl, n.d. Accessed 12.04.2019.
URL https://docs.julialang.org/en/v1/manual/profile/index.html

Revels, J., Lubin, M., Papamarkou, T., 2016. Forward-mode automatic differentiation in julia. arXiv:1607.07892 [cs.MS].
URL https://arxiv.org/abs/1607.07892

Saad, Y., 2003. Iterative Methods for Sparse Linear Systems. Ch. 3. Sparse Matrices, pp. 73–101.
URL https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.ch3

Shure, L., 2016. Run code faster with the new matlab execution engineAccessed 13.05.2019.
URL https://blogs.mathworks.com/loren/2016/02/12/run-code-faster-with-the-new-matlab-execution-engine/

Single-phase Compressible AD Solver, n.d. Accessed 22.11.2018.
URL https://www.sintef.no/contentassets/2551f5f85547478590ceca14bc13ad51/core.html#single-phase-compressible-ad-solver

SparseMatrixCSC docs, n.d. Accessed 27.02.2019.
URL https://docs.julialang.org/en/v1/stdlib/SparseArrays/#SparseArrays.SparseMatrixCSC

Speelpenning, B., 1980. Compiling fast partial derivatives of functions given by algorithms. Tech. rep., Illinois Univ., Urbana (USA). Dept. of Computer Science.

StaticArrays.jl, n.d. Accessed 26.04.2019.
URL https://github.com/JuliaArrays/StaticArrays.jl

The Julia Lab, n.d. Julia language research and development at mit. Accessed 21.11.2018.
URL https://julia.mit.edu

Tic, n.d. Accessed 21.12.2018.
URL https://uk.mathworks.com/help/matlab/ref/tic.html

@time docs, n.d. Accessed 12.04.2019.
URL https://docs.julialang.org/en/v1/base/base/#Base.@time

Wengert, R. E., 1964. A simple automatic derivative evaluation program. Communications of the ACM 7 (8), 463–464.

White, J. M., 2013. The relationship between vectorized and devectorized code. Accessed 17.01.2019.
URL https://www.johnmyleswhite.com/notebook/2013/12/22/the-relationship-between-vectorized-and-devectorized-code/

Yuret, D., 2016. Knet: beginning deep learning with 100 lines of julia. In: Machine Learning Systems Workshop at NIPS 2016.
URL https://github.com/denizyuret/AutoGrad.jl