SPECIALIZATION PROJECT IN MATHEMATICAL SCIENCES

Automatic Differentiation in Julia

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

February 25, 2019

Table of Contents

Chapter 1

Introduction

Automatic differentiation (AD) is a method that automatically calculates the derivatives of a function with no human calculations. 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 performs standard differentiation rules to these operations and functions. But it does not apply differentiation rules to the symbols like we do when we calculate it by hand; It carries along both the function value and derivative values for all the elementary steps of the evaluation. In this way it can calculate the next function value and derivative value, based on the current values and the rules for the next elementary operation or function. This gives derivative values as accurate as hand derived derivatives, but without possible 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 that have different capabilities. The exact difference and capabilities between the two will be discussed closer in section 2.1.

According to **?** the first ideas of the concept AD dates back to the 1950's *(**??**)*. More specifically, forward AD was discovered by Wengert in 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 1960's *(**?**)*. After the initial discovery of AD further activities hibernated for a couple of years, before it 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, came in 1980 by Speelpenning *(**?**)*. Further research on the topic was done by among others Griewank during the 1980's *(**?**)*. Today AD is widely used in many applications. One of them is machine learning that specifically uses the backward AD to minimize functions. This project's main focus has been using the forward AD method to solve partial differential equations.

The report begins with 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 partial differential equations using a finite volume method and a discretization of the differentiation operators. Chapter 3 starts by elaborating on why we want to implement AD in Julia. Some implementation specific details are then discussed, before the implementation is tested against other AD implementations in Julia and MATLAB. The implemented AD library will then be used in chapter 4 to solve an example that consists of solving partial differential equations that describe the flow inside an oil reservoir using a finite volume method. Lastly, chapter 5 summarizes the results and gives some thoughts on topics that are interesting to look further into.

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

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

etc. and the chain- and product rule

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

The computer will then use these rules on symbolic variables to obtain the derivative of any function given. This will give perfectly accurate derivatives, but it is computationally demanding, and as f(x) becomes more complex the calculations will become slow.

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 truncation error. Hence, in finite differences you have the problem that you need a small step size h to reduce the truncation error, but h can not be too small, because then you get round-off errors. Hence, the finite-difference method is unstable and is not 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, we 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 $[\cdot,\cdot]$. In this way, we can continuously update both the function value and the derivative value for every operation we perform on the function.

As an example, consider the scalar function $f = f(x)$ with its derivative f_x where x is a scalar variable. Then the AD-variable x is the pair $[x,\ 1]$, and for f we have $[f,\ f_x]$. In the pair $[x,\ 1]$, 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 arithmetic operators 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 so that for a function h(x)

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

The only things that remain to be defined are the rules concerning elementary functions like

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

When these arithmetic operators and the elementary functions are implemented, you are able to evaluate the derivative of any scalar function without actually doing any form of differentiation yourself. 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

$$\begin{aligned} 2x &= [2,\ 0] \cdot [2,\ 1] \\ &= [2 \cdot 2,\ 0 \cdot 1 + 2 \cdot 1] \\ &= [4,\ 2]. \end{aligned}$$

After this computation, we get from the exponential

$$\begin{aligned} \exp(2x) &= \exp\big([4,\ 2]\big) \\ &= [\exp(4),\ \exp(4) \cdot 2], \end{aligned}$$

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 is equal

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

for $x = 2$.

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 corresponds to the function value and the derivative value. ϵ is like we have for complex numbers $i^2 = -1$, but the corresponding relation for dual numbers are $\epsilon^2 = 0$. 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 Definition (2.4). Then we get for addition

$$\begin{aligned} x + y &= (a + b\epsilon) + (c + d\epsilon) \\ &= a + c + (b + d)\epsilon, \end{aligned}$$

for multiplication

$$\begin{aligned} x \cdot y &= (a + b\epsilon) \cdot (c + d\epsilon) \\ &= ac + (ad + bc)\epsilon + bd\epsilon^2 \\ &= ac + (ad + bc)\epsilon, \end{aligned}$$

and for division

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

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

$$\begin{aligned}f(x) = f(a+b\epsilon) &= f(a)+\frac{f'(a)}{1!}(b\epsilon)+\frac{f''(a)}{2!}(b\epsilon)^2+\ldots \\ &= f(a)+f'(a)b\epsilon.\end{aligned}$$

This means that to make dual numbers handle elementary functions, the first order Taylor expansion needs to be implemented. In practise, this equals the implementations of elementary differentiation rules described in equations (2.2).

The weakness of implementing AD with dual numbers is clear for functions with multiple variables. Let the function f be defined as $f(x,y,z) = x\cdot y+z$. Let us 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:

$$\begin{aligned}f(x,y,z) &= (2+1\epsilon)\cdot(3+0\epsilon)+(1+0\epsilon) \\ &= 7+3\epsilon.\end{aligned}$$

Here, 7 is the function value of f, while 3 is the derivative value f_x of f with respect to 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 consider the approach for forward AD where the method also can be be explained as an extensive use of the chain rule

$$\frac{\partial f}{\partial t} = \sum_i \left(\frac{\partial f}{\partial u_i}\cdot\frac{\partial u_i}{\partial t}\right). \tag{2.5}$$

Take $f(x) = x\cdot\exp(2x)$, like in the forward AD example (2.3). We then split up the function into 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)

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

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

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

This shows how forward AD actually uses 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 uses the same sequence of elementary functions like in Definition (2.6), the expressions from the chain rule (2.7) becomes

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

By 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 $f(x,y,z) = z(\sin(x^2) + yx)$ 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) becomes

$$\frac{\partial s}{\partial g_5} = \text{Unknown} \qquad \frac{\partial s}{\partial g_2} = \frac{\partial s}{\partial g_4} \qquad \frac{\partial s}{\partial y} = x \cdot \frac{\partial s}{\partial g_2}$$
$$\frac{\partial s}{\partial g_4} = z \cdot \frac{\partial s}{\partial g_5} \qquad \frac{\partial s}{\partial g_1} = \cos(g_1)\frac{\partial s}{\partial g_3} \qquad \frac{\partial s}{\partial z} = g_4 \cdot \frac{\partial s}{\partial g_5}$$
$$\frac{\partial s}{\partial g_3} = \frac{\partial s}{\partial g_4} \qquad \frac{\partial s}{\partial x} = 2x \cdot \frac{\partial s}{\partial g_1} + y \cdot \frac{\partial s}{\partial g_2}$$

substituting s with g_5 yields

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

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, one for each derivative, this is a big improvement. This illustrates the strength of backward AD – no matter how many input parameters a function have, 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 the function- and the 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 a bad implementation of this tree will reduce the advantage of backward AD. Also, if the function is a vector function and not a scalar function, backward AD needs to run m times if $f : \Re^n \to \Re^m$. Hence, if $n \approx m$, forward AD and backward AD will have approximately the same complexity. This is some of the reasons why we will have focus on implementing forward 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 a forward AD, there are more efficient ways of doing this than implementing with dual numbers. Neidinger describes in *Neidinger (2010)* a method in which we do not need n function evaluations for n input

parameters. Say we have a scalar function $f : \Re^n \to \Re$, and we want to obtain the gradient of f. Then, the main idea is when we define our AD-variables, instead of having each AD-variable storing only the derivative with respect to itself, they store their gradient to the corresponding space they are in. This means we have to define what we call our primary variables, which are all the variables in the relevant space. 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 Equations (2.1) and the elementary functions in Equations (2.2) are still valid, but instead of scalar products they 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. In numerical applications, as we are dealing with discretizations, the functions we evaluate are usually vector functions and not scalar functions. Hence $f : \Re^n \to \Re^m$. Then, it is the Jacobian of f we are interested in:

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

The forward AD method described above will be similar for a vector function as it was for a scalar function above, except for two important differences. The first one is that the primal 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 examples.

2.2 Applications of Automatic Differentiation

AD can be used in a wide spectre 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 linear systems in multiple dimensions. Let us look at the linear system

$$\mathbf{A}\boldsymbol{x} = \mathbf{b}. \tag{2.9}$$

But instead of looking at it like in equation (2.9), we write it 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 (2.9), 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, this will converge in one step using the multivariate Newton-Raphson method. The general form of the multivariate Newton-Raphson method is given by

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n - J_{\boldsymbol{F}}(\boldsymbol{x}^n)^{-1}\boldsymbol{F}(\boldsymbol{x}^n). \tag{2.10}$$

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 (2.9) 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 instead. But for numerical applications, we can with this method easily solve non-linear equations obtained from discretization of partial differential equations (PDE's) by looking at the residual form of the equations and use the Newton-Raphson method (Equation (2.10)) with AD. As a preface to introducing how we will do this, consider the Poisson equation

$$-\nabla(\mathbf{K}\nabla u) = q, \tag{2.11}$$

where $\mathbf{K}$ is an diffusion 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 on each cell such that

$$\int_{\partial\Omega_i} -\mathbf{K}\nabla u \cdot \mathbf{n}\,\mathrm{ds} = \int_{\Omega_i} q\,\mathrm{dA}. \tag{2.12}$$

Here, $\mathbf{n}$ is the unit normal to the cell Ω_i, so Equation (2.12) 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 production in 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. They both have a value u_i and u_k in the centre of the cell, and the boundary, or facet, between the cells is defined as $\Gamma_{i,k}$.

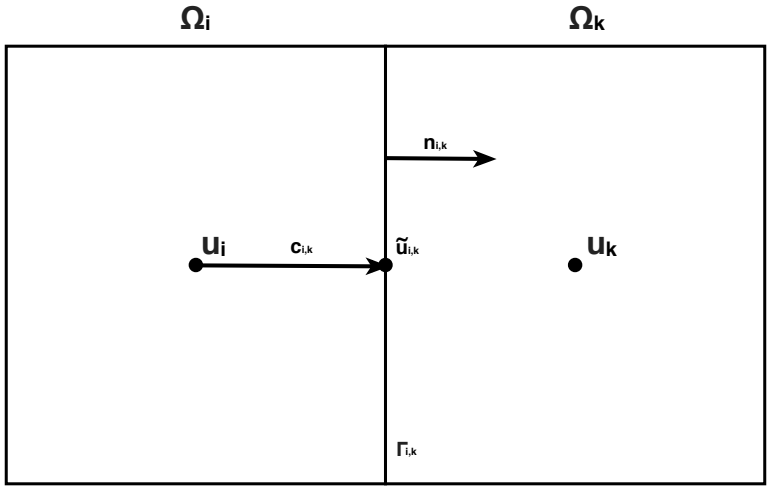


Figure 2.2: Figure of two adjacent cells Ω_i and Ω_k. The average value of the cell is given by u_i. The boundary between the cells is $\Gamma_{i,k}$ with value at the centre equal $\bar{u}_{i,k}$ and outward normal vector $n_{i,k}$

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

$$v_{i,k} = \int_{\Gamma_{i,k}} \mathbf{v} \cdot \mathbf{n}_{i,k} \mathrm{ds}. \tag{2.13}$$

If we let $L_{i,k}$ be the length of $\Gamma_{i,k}$, then the integral in (2.13) can be approximated by the midpoint rule with $\tilde{\mathbf{v}}_{i,k}$ as the flux on the midpoint of $\Gamma_{i,k}$:

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

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 have the value of u at the center of cell Ω_i, u_i, and not on the facet, $\tilde{u}_{i,k}$. This means that finding the gradient of u on $\Gamma_{i,k}$ using the approximation

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

can not be computed directly. 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 first define what we call a transmissibility

$$T_{i,k}(\tilde{u}_{i,k} - u_i) = L_{i,k} \mathbf{K}_i \frac{(\tilde{u}_{i,k} - u_i)\mathbf{c}_{i,k}}{|\mathbf{c}_{i,k}|^2} \cdot \mathbf{n}_{i,k}. \tag{2.14}$$

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

$$\begin{aligned} (T_{i,k}^{-1} + T_{k,i}^{-1})\, v_{i,k} &= (\tilde{u}_{i,k} - u_i) - (\tilde{u}_{i,k} - u_k) \\ v_{i,k} &= (T_{i,k}^{-1} + T_{k,i}^{-1})^{-1}(u_k - u_i) = T_{ik}(u_k - u_i) \end{aligned} \tag{2.15}$$

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, 2018)*. Now that we have an approximation of the flux through the interface between Ω_i and Ω_k, we get that Equation (2.13) can be approximated by

$$\sum_k T_{i,k}(u_k - u_i) = q_i, \qquad \forall \Omega_i \in \Omega, \tag{2.16}$$

where q_i is the total production in 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 (2.11), using the scheme explained in (2.10) 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. But for more complex PDE's and especially more complex grids, the construction of the matrix **A** becomes more tricky and the ease of using AD, together with the discrete differentiation operators, that we will define in subsection 2.2.3, becomes clearer.

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.

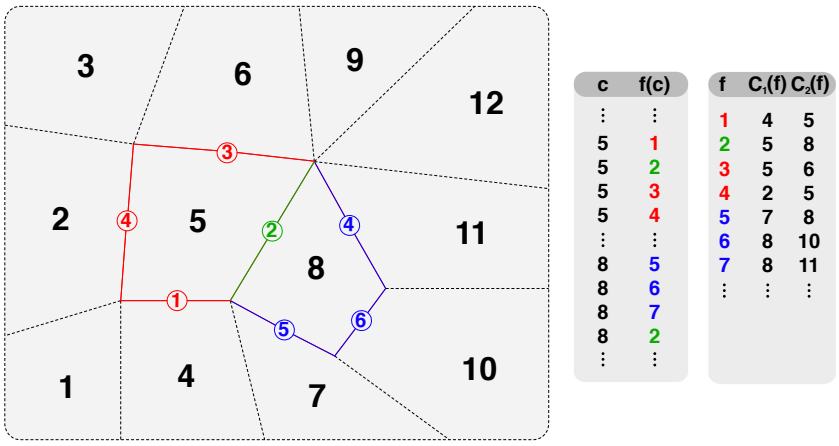


Figure 2.3: Figure of a general polygonal grid with the mapping cell to facets and facet to cells.

Figure 2.3 shows how we can introduce two different mappings that explain the relation between the cells and the facets. The values dependent on 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 (2.16). From these, we want to create discrete divergence and gradient operators that correspond to the continuous equivalents for this grid. Consider the Poisson equation (2.11) for the function u. Then the discrete gradient operator for a facet f is defined as

$$\texttt{dGrad}(u)[f] = u[C_2(f)] - u[C_1(f)], \tag{2.17}$$

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 (2.15). 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

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

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

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

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

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

Here, **T** is the transmissibility defined in (2.14). We can see how similar the notation for the discrete equations is to the continuous equations. 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 4.

Chapter 3

Implementation

In this chapter I will discuss why it is interesting to implement AD in Julia. I will also give some implementation specific details and benchmark the implementation against other AD libraries in Julia and MATLAB.

3.1 Julia

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

> "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?)" *(Bezanson et al., 2012)*.

In short, it seems to be the perfect language for numerical applications and it would be interesting to see how it performs compared to MATLAB. When it comes to AD in Julia, there are already some packages that can be used. Most of them are backward AD-packages designed for machine learning, for example AutoGrad *(Yuret, 2016)* and Zygote *(Innes, 2018)*. The reason why AD-packages for machine learning are based on backward AD is that, without going to deep into the subject, it largely consist of 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. But there is one package called *ForwardDiff (Revels et al., 2016)* that uses forward AD, being developed by the Julia community. ForwardDiff uses dual numbers as explained in subsection 2.1.2 extended to multiple dimensions. This is called multidimensional dual numbers and a vector $\mathbf{x}$ is represented as

This package works very well for some applications, but for others it has some limitations that are not ideal, for example:

- The function we want to differentiate only accepts 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 elementwise 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.

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

3.2 Implementation of Automatic Differentiation

When it comes to efficiently implementing AD, there are two factors to consider. Firstly, it must be easy and intuitive to use. Secondly, it must be efficient code 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 stores respectively the value and the corresponding Jacobian:

```
mutable struct AD
    val
    jac
end
```

The struct is mutable so that we are able to change the values of the struct after it is defined. The `val` variable is a vector unless it is a scalar variable, then it is only of type `Float64`. When it comes to the Jacobian, there are multiple ways of storing the matrix. Depending on the application and how much, and what type of, manipulation of the matrix you are going to do, the choice is based on efficiency and convenience. My implementation is inspired by the implementation in MRST *(Lie, 2018)*, where we represent the Jacobian, `jac`, as a vector of sparse matrices, where each sparse matrix is the Jacobian w.r.t. each primary variable. This implementation gives the freedom to easily work with the Jacobian for just a single primary variable.

Now we need to implement operators for this type of struct. The importance of the way you implement the AD operators and elementary functions can be expressed in a short example: Assume you have two 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 AD-variables as input parameters, it will for the evaluation of f look something like this:

$$f = \texttt{ADplus}(y,\texttt{ADexp}(\texttt{ADtimes}(2,\texttt{ADtimes}(x, y)))).$$

This is clearly not a suitable way to implement AD as it quickly becomes difficult to see what type of expression it is. This approach should be avoided. Lie and Neidinger suggests in *Lie (2018)* and *Neidinger (2010)* a much more elegant implementation, where instead of making new functions that take in AD-variables as parameters, one should overload the standard operators (+,-,*,/) and the

elementary functions (exp, sin, log, etc.). In Julia this can be done by exploiting the fact that Julia has *multiple dispatch*. A quick explanation that satisfies our needs is that at runtime, the compiler understands what types are given as input for either an operator or a function and chooses the correct method based on this. To demonstrate, this is done by implementing a function

```
import Base: +
function +(A::AD, B::AD)
    ## Overload operator
end
```

that overloads the + operator. Here, we import the + operator from Base (which is where the standard functions in Julia lie) and overload it for AD variables. This means that this implementation of the operator is only used when there are AD variables on both sides of the operator. Hence, if $x = 1$, $y = 3$ and then $z = x + y$ is declared, Julia understands that it is not the definition above, but the normal addition for integers it should use. But if $x, y =$ `initialize_AD`$(1,3)$ is declared, so that x and y both are AD variables, then Julia's multiple dispatch will understand that the new definition of the "+"-operator should be used in the expression $z = x + y$. What we need to remember is that if I now write $z = x + 3$, with x as an AD variable, Julia will deploy an error message. This is because we also have to implement

```
import Base: +
function +(A::AD, B::Number)
    ## Overload operator
end
+(A::Number, B::AD) = B+A
```

Here, the first function will be used if the + operator is used with an AD 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 on the right hand side. But as you can see, we do not implement the same thing twice – we use the function we already have made. When we have implemented all the functions necessary, it gives us the opportunity for the function f above to simply write $f = y + \exp(2 * x * y)$ and Julia will understand that it is our implementation of +, * and exp operators that shall be used, and f will become an AD-variable with the correct value and derivatives.

Up until now I have only discussed implementation of AD for scalar variables. But another advantage of Julia's multiple dispatch system is clear if we start looking at vector variables and functions. In some situations, like in chapter 4, we want to sum over 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::AD)
    ## Overload sum
end
```

which would indeed work, but to exploit Julia's multiple dispatch fully, we can instead overload the `iterate` function. This function explains how we shall iterate through an AD variable:

```
function iterate(iter::AD, 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 elementary operation `/`, the Base function `mean` will also work on AD variables.

When one introduces AD for vectors, one need to discuss how to handle multiplication and division. 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, and $u * v$ gives an $n \times n$ matrix where each row is v multiplied by the corresponding row value of u. An attempt to evaluate $u * u$ will end in an error message saying that "the dimensions does not match matrix multiplication". The ".*" operator however, is the elementwise multiplication operator. This means that if we have regular column vectors like u and $w = v'$, the transpose of v, the evaluation of $u .* w$ will be elementwise multiplication of u and w, into a new vector of same dimensions as u and w. Here one need 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 AD. The Julia issue *Julia issue dot operators (n.d.)* from 2017 explains the problems of overloading the elementwise ".*" operator, and 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 elementwise multiplication in this project, I have decided to implement "*" as elementwise multiplication. This means that if I have written regular multiplication expressions consisting of at least one AD-variable, it is elementwise multiplication that is being executed.

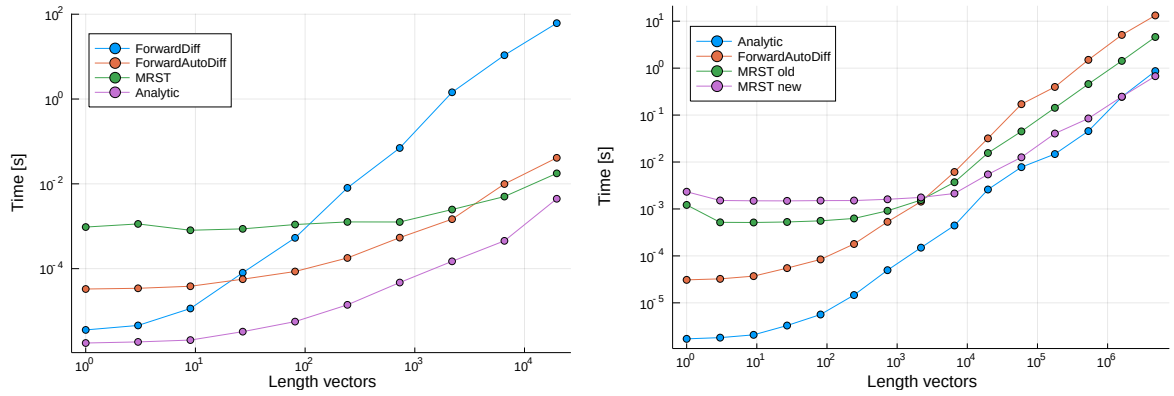
3.3 Benchmarking Automatic Differentiation

As mentioned in section 3.1, 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 two implementations compare when the functions evaluated are getting larger. As another reference, I have added the AD implementation in the MATLAB Reservoir Simulation Toolbox (MRST) *(MRST Homepage, n.d.)* to the benchmark, to see how the Julia implementations compare to an optimized AD tool in MATLAB. To benchmark the efficiency of the different AD tools, I have evaluated the vector function $f : \Re^{n\times 3} \to \Re^n$ where

$$f(x, y, z) = \exp(2xy) - 4xz^2 + 13x - 7, \tag{3.1}$$

and $x, y, z \in \Re^n$. Figure 3.1 shows how computational time for calculating the function value and the Jacobian of the function, for the different methods, scales as the length of the vectors n increases. [1]

[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 1600 MHz DDR3 memory. For benchmarking time in Julia, I have used the benchmarking library *BenchmarkTools (n.d.)*. For benchmarking time in Matlab I have taken the median of mulitple tests using the stopwatch *Tic (n.d.)*.



(a) **(b)**

Figure 3.1: Computational time for calculating the value and Jacobian of f in Equation (3.1) as a function of length of the input vectors.

In Figure 3.1a we have four different graphs. The analytic graph is simply the evaluation the analytic functions $f(x,y,z)$, f_x, f_y and f_z. *ForwardDiff* is the AD package in Julia, MRST is the AD tool implemented in MATLAB and *ForwardAutoDiff* is the AD tool I have implemented in Julia. 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 Jacobian matrix as discussed in section 3.1. For $f(x,y,z)$ this will be a $3n \times 3n$ matrix which is a matrix with more than 3 billion elements for the largest values of n. We can also observe that for small vectors, MRST and *ForwardAutoDiff* have much more overhead than *ForwardDiff* and the analytic solution. This makes them slower for small n, but as n grows, this overhead becomes more negligible.

The computational costs of both MRST and *ForwardAutoDiff* approach the analytic evaluation as n grows, and it is thus interesting to see how they scale for even larger n. This can be seen in Figure 3.1b. Here, *ForwardDiff* is left out since it becomes too slow, but I have added a new implementation from MRST, that I will call *MRST new*. The MRST implementation in Figure 3.1a is now referred to as *MRST old*. We can observe that the trend seen in Figure 3.1a where *MRST new* is faster than *ForwardAutoDiff* for vectors longer than 10 000 continues for even longer vectors. As we can see from Figure 3.1b *MRST old* is much faster than the two other implementations for long vectors. This is because it is specially optimized for element operations like we have when evaluating the function in Equation (3.1). *MRST new* exploits that all the Jacobians in the calculation of f simply are diagonal matrices with respect to each primary variable. This means that it can store the values of the diagonals as vectors and calculate the new Jacobians with simple vector multiplication. With this approach we skip the overhead accompanying sparse matrix multiplication. This implementation actually becomes just as fast as the analytic evaluation in Julia for vectors of length $\approx 10^7$. As said, this method is especially efficient for functions like in Equation (3.1), but if we for example want to calculate something like

$$g(x) = \frac{x[2:\text{end}] - x[1:\text{end}-1]}{\texttt{sum}(x)}, \tag{3.2}$$

the diagonal structure of the Jacobians are gone, and the *MRST new* implementation can not be used. The *MRST old* implementation with the Jacobians as sparse matrices is then used.

The creators stated in the blog post accompanying the first release of Julia in 2012 *(Bezanson et al., 2012)* that Julia is supposed to be just as fast as C. Hence it would be interesting to see if we can increase, or at least not loose, computational efficiency in the evaluation of the vector function in (3.1) by evaluating it scalar by scalar in a loop instead of by vector multiplications. The difference can be illustrated by the two functions

```
function benchmarkAD(x_vec,y_vec,z_vec)
    ## initialize AD variables x, y, z
    f_ad = exp(2*x*y) - 4*x*z^2 + 13*x - 7
end
function benchmarkADinLoop(x_vec,y_vec,z_vec)
    ## initialize AD variables x, y, z
    f(x,y,z) = exp(2*x*y) - 4*x*z^2 + 13*x - 7
    for i = 1:length(x)
        f_ad[i] = f(x[i],y[i],z[i])
    end
end
```

Implementation specific parts are left out. The result can be seen in Figure 3.2, where the graphs with circles as markers are the same methods as in Figure 3.1a using the function `benchmarkAD`. The graphs with squares are the same methods, only they are tested with the implementation in function `benchmarkADinLoop`.

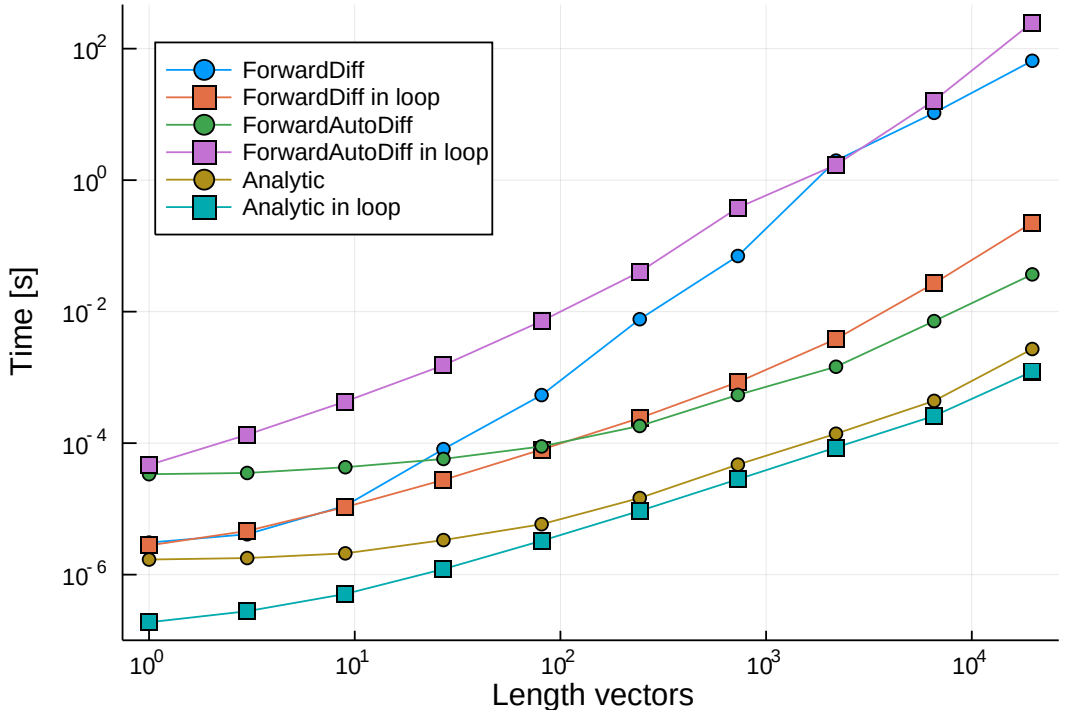


Figure 3.2: Computational time for calculating the value and gradient of f in Equation (3.1) as a function of length of the input vectors.

The first observation to make is that the *ForwardAutoDiff* implementation is clearly not optimized for evaluating the vector function scalar by scalar, as it is the slowest method tested so far for all vector lengths. The next interesting observation is that Julia's implementation of AD, *ForwardDiff*, can be made much more efficient in the evaluation of the vector function, by evaluating the function scalar by scalar. Using the method in `benchmarkADinLoop` with *ForwardDiff*, we almost achieve the same test results as the regular *ForwardAutoDiff* for long vectors. Although, it is important to mention that with the approach in `benchmarkADinLoop`, only the gradient of the function is obtained – not the Jacobian. This limits the applicability of the method. In the particular case of the function f in Equation (3.1), the Jacobian will only be a diagonal matrix with the gradient of f on the diagonal, but if we would evaluate a function like in Equation (3.2), this approach would not work. Hence, although we almost manage to obtain the same performance in *ForwardDiff* as we have in *ForwardAutoDiff*, it comes with a cost that some types of functions cannot be evaluated. The implementation necessary

to work around this and obtain the Jacobian with *ForwardDiff* and `benchmarkADinLoop` will slow the computation down. For a vector function with large input vectors, *ForwardAutoDiff* is therefore a better approach.

Other than this, it is interesting to see that the evaluation of the analytical solution in a loop is faster than its vectorized counterpart. Here, Julia shows a real strength compared to MATLAB, where a function evaluation like the vector function f will be much slower in a loop than with vector multiplication.

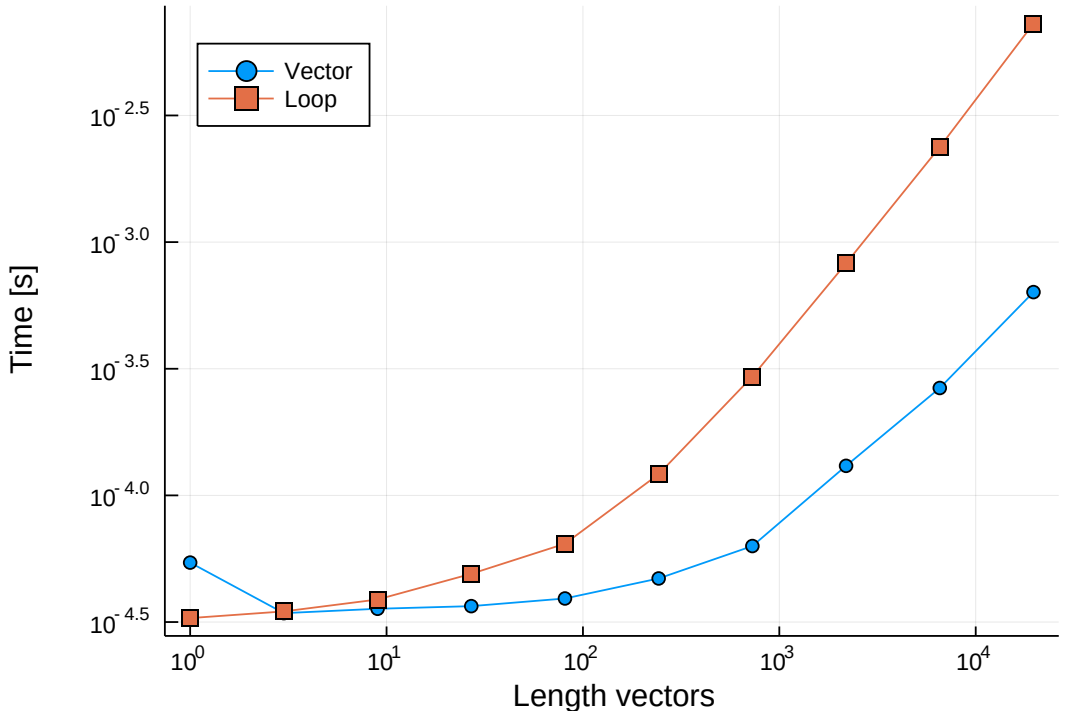


Figure 3.3: Computational time for evaluating the analytic functions f, f_x, f_y and f_z from (3.1) as a function of length of the input vectors in MATLAB.

Figure 3.3 shows how much time MATLAB uses to evaluate the analytic functions f, f_x, f_y and f_z from Equation (3.1) as vector multiplication and in a for-loop. The analytic graphs in Figure 3.2 demonstrates the time Julia uses to evaluate the same functions. Where the vector multiplication and for-loop scale equally good in Julia, and the for-loop actually perform better, MATLAB's for-loops scale much worse than the vector multiplication in contrast. This can be viewed as a first indication that the developers of Julia actually have managed to create a language with similar mathematical syntax as MATLAB and the computational efficiency of C.

Chapter 4

Flow Solver With Automatic Differentiation

To test Julia's AD tools in a real world application I have implemented an example taken from the MATLAB Reservoir Simulation Toolbox (MRST) and implemented it in Julia. MRST is primary developed by the Computational Geosciences group in the department of Mathematics and Cybernetics at SINTEF Digital *(MRST Homepage, n.d.)*. 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." Although most of the tools and simulators in MRST are very efficient and perform well – if you are to simulate more heavy simulation, they recommend to use the Open Porous Media (OPM) Flow simulator *(Open Porous Media, n.d.)*. OPM is a toolbox to build simulations of porous media processes that are mainly written in C++ and C. Here, the differences between the languages become clear. As MATLAB with its easy to use mathematical syntax is a great language to quickly make prototypes and demonstrations of simulations, it is failing somewhat when it comes to computational speed. But where MATLAB fails in computational speed, C++ and C are two very fast languages. The problem with C++ and C is that these languages are not built for numerical analysis, hence it takes longer time to create the simulations. This is where Julia comes in. As the founders of Julia stated: Julia 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. Hence, it is interesting to figure out how Julia can perform compared to MRST.

To compare 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 implementation of AD in MATLAB and Julia.

4.1 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 that is producing oil. "Single-phase solver" only means that we do not have different phases of fluid, like liquid and gas, present during the simulation. As the 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 4.2a. 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. In an oil reservoir, the oil lies inside porous rock and hence the properties of the rock will affect the flow of the oil. The amount of oil we can have inside the rock is measured as pore volume. First, we make a new variable `rock` that contains parameters that describe the rock's ability to store and trasmit fluids for each cell with the function `rock = makeRock(...)`. Then, we say that in our model the rock is compressed constantly as a function of pressure and we obtain the analytic solution for the pore volume as a function of pressure

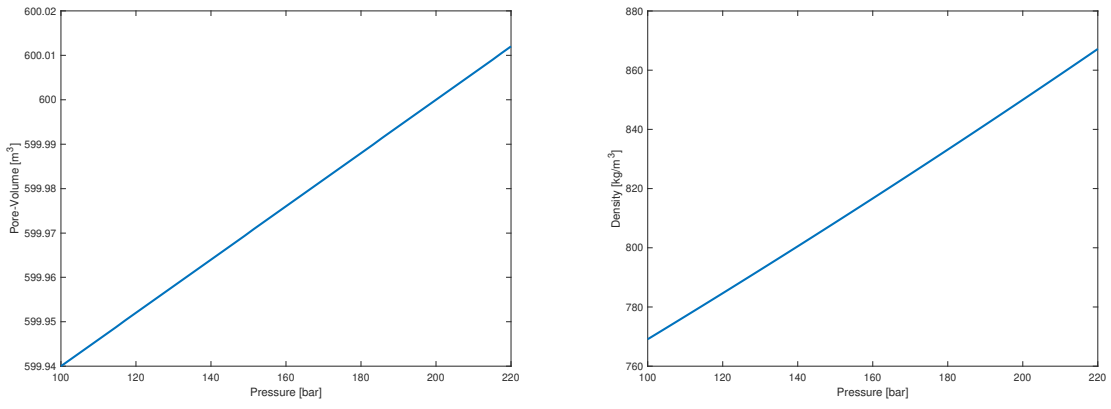
$$pv(p) = \mathrm{pv}_r \exp[c_r \cdot (p - p_r)], \tag{4.1}$$

where pv_r is the rock pore volume properties, in a cell, at a reference pressure p_r. 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 4.1b. The graph shows that for a cell with volume $1000\mathrm{m}^3$ 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

$$\rho(p) = \rho_r \exp[c \cdot (p - p_r)], \tag{4.2}$$

where $\rho_r = 850\mathrm{kg/m}^3$ and c is a constant controlling how fast the oil is compressed. The density of the fluid as a function of pressure is plotted for some pressure values in Figure 4.1a.



(a) The pore volume of the rock in a cell as a function of pressure.

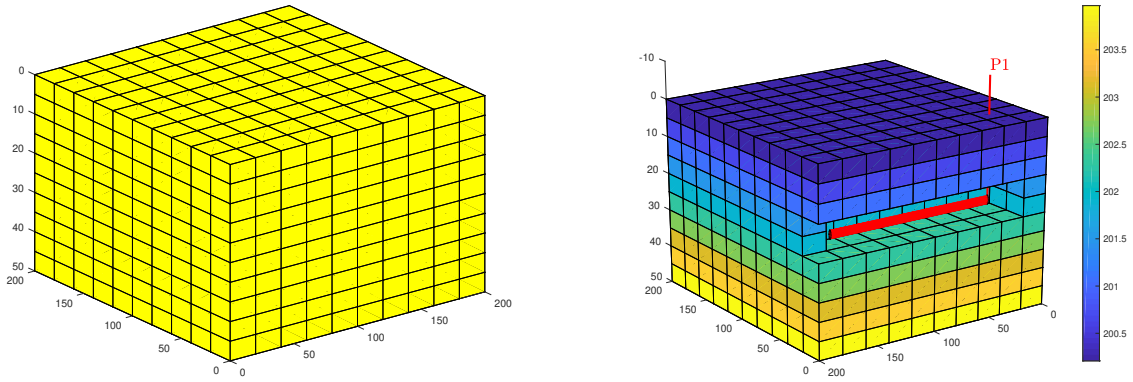
(b) The density of the oil in the reservoir plotted as a function of pressure.

Figure 4.1

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

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

for the fluid density given by Equation (4.2) and g as the gravity. The well is then inserted by removing 8 grid elements using the `W = addWell(...)` function. The grid with initial pressure and well can be seen in Figure 4.2b.



(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 has replaced 8 grid elements. Some grid elements are removed to give a better visualization of the well.

Figure 4.2

4.2 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 (2.14). This is done by exploiting that we have all the necessary information about the grid properties, such as the cells centroid coordinates, facets areas 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 (2.17). The matrix C is stored as a sparse matrix, and the reason why is clear from Figure 4.3, which shows the sparse structure of C. 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, 2018)*, and hence the adjoint of `C` is the negative transpose of `C`.

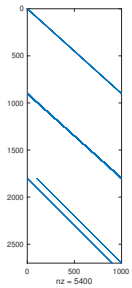


Figure 4.3: Structure of discrete operator `C`.

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 (2.10) 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

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

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 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 define the continuity equation in the continuous case

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

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 (4.1) 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

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

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 the 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 pressure is given, we determine the surface rate, and vice versa). Herein, we assume 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 1000 residual equations are the `presEq` with negative production `q_conn` for the indices `wc`. Equation number 1001 is the `rateEq` and Equation 1002 is the `ctrlEq`. Hence, $\boldsymbol{x} \in \Re^{1002}$, where the first 1000 elements are the average pressure in each cell, element 1001 will be the pressure at the datum point inside the well (`bhp`), and element 1002 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 (2.10). 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-values
    end
    timeNow += dt
end
```

4.3 Flow Solver Results

If we simulate how the pressure in the reservoir will decay during one year, we will get the result displayed in Figure 4.4.

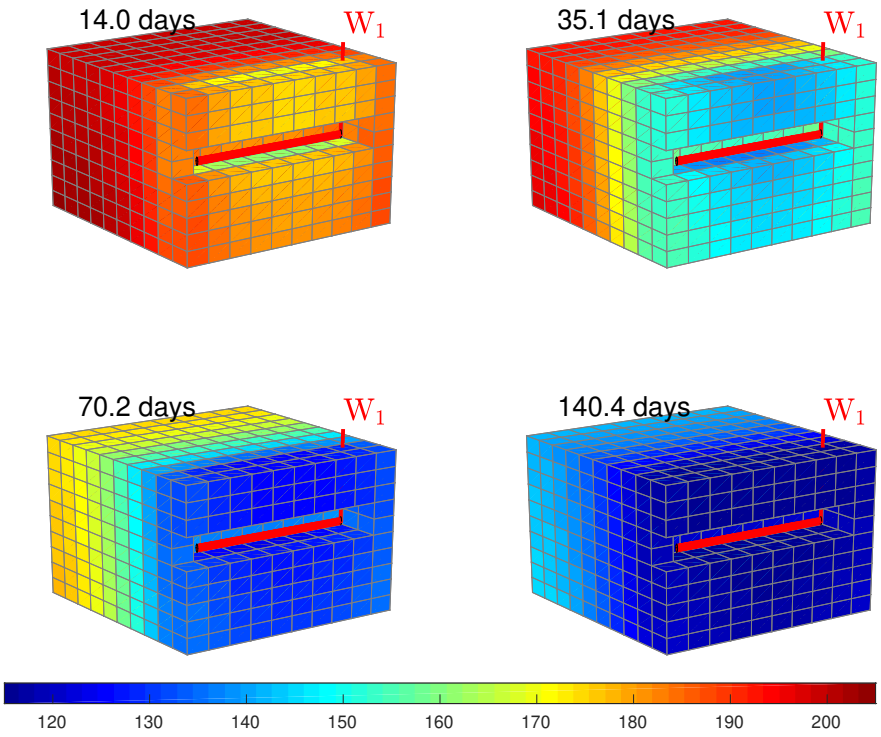


Figure 4.4: The pressure in the reservoir displayed at four different times. Pressure is given in bar.

Note the different intervals for the color bar in Figure 4.4 compared to Figure 4.2b. 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 begins to decay also furthest away from the well.

Figure 4.5 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 phase is what is called primary production. In primary production, the pressure in the reservoir is so high that there is no need to pump the oil out, the

pressure difference does all the work. After some time, we can see that the pressure becomes too low and the production decreases. When this happens, we transit to what is called secondary production. To retrieve more oil from the reservoir we need to apply extra pressure inside the reservoir. This can for example consist of injecting water or gas into the reservoir. This is called the secondary production. More details about how this work can be found in *Lie (2018)*, but the main idea here is that in order to keep up the production and fully exploit the resources in the reservoir, we need to apply external pressure. 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. This flow solver is a simple example of how we can model such evolution of the pressure in a reservoir elegantly with governing equations on residual form, created by discrete differentiation operators, and AD.

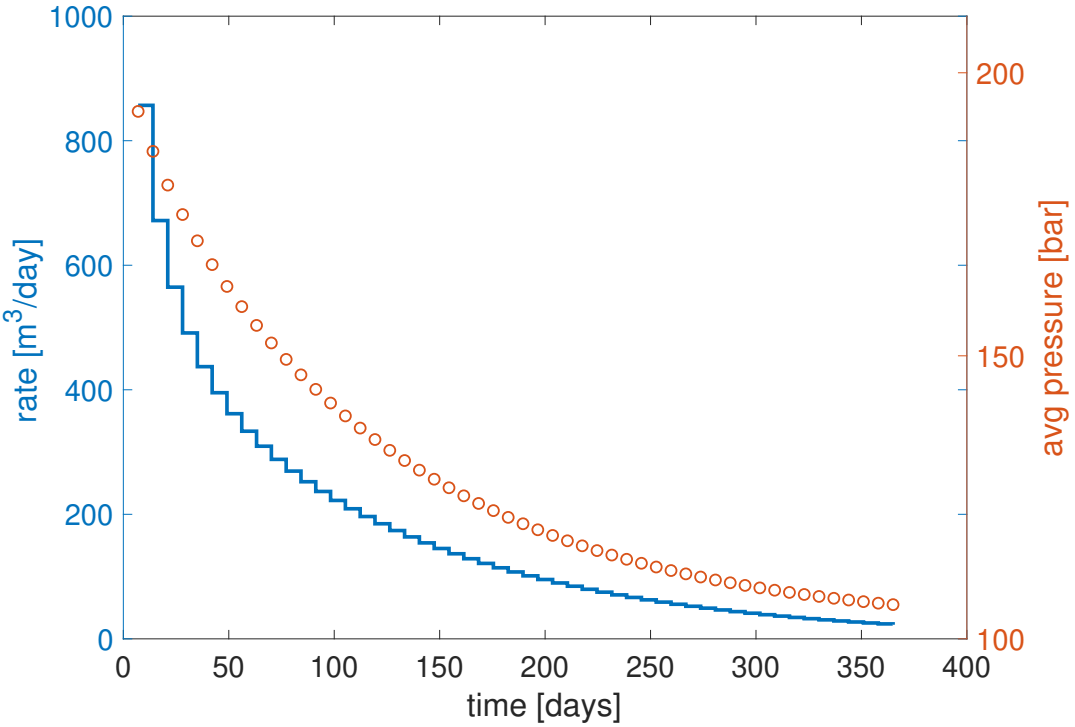


Figure 4.5: The production rate and the average pressure in the reservoir as a function of time.

When it comes to solving the residual equations with the Newton-Raphson method, it is interesting to have a look at how the Jacobian of $\boldsymbol{F}$ looks like. Figure 4.6 shows the structure of the Jacobian. The first impression is that the matrix is very sparse. Except from a few nonzero points in row 1001 and column 1001 because of the well, the Jacobian consist of 7 diagonals with nonzero elements and the rest of the elements being 0. As there are only 6419 non-zero elements out of more than 1 million matrix elements, it is clear that storing the full 1002×1002 matrix will be very inefficient.

As explained in section 2.2, the different types of AD mentioned store the Jacobians differently. In *ForwardAutoDiff* and MRST's implementations of AD, the Jacobians are stored as a list of sparse matrices where each Jacobian element in the list is the Jacobian with respect to one primary variable. In this example, this is a lot more efficient than storing the full 1002×1002 Jacobian as *ForwardDiff* does.

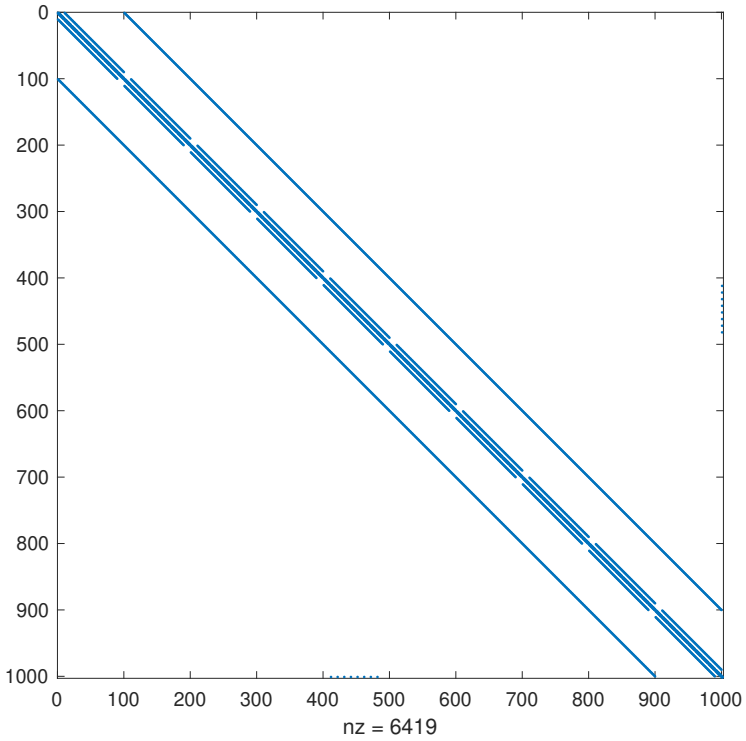


Figure 4.6: Structure of the 1002×1002 Jacobian of the governing equations. There are 6419 non-zero elements.

The difference in implementation of the Jacobian should make a big impact on the computation time for this problem. To benchmark the different methods, we need to do some extra work to make sure that it is actually the AD we benchmark 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 done 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 benchmark 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 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, I have benchmarked three different discretizations. The first is the original setup with 10 cells in spatial direction x, y, and z. This gives a total number of 1000 cells in the reservoir. Then, I have also tested the implementations for 20 and 30 cells in the spatial directions. The time spent solving the system for the different methods can be seen in Table 4.1.

Table 4.1: Table with speed benchmarks of different AD methods solving the "Single-Phase Compressible AD Solver" for different discretizations.

Number of cells	ForwardDiff	ForwardAutoDiff	MRST	$\frac{\text{ForwardAutoDiff}}{\text{MRST}}$
$10 \times 10 \times 10$	71.5s	2.3s	1.9s	1.21
$20 \times 20 \times 20$		31.9s	23.4s	1.36
$30 \times 30 \times 30$		188.8s	147.8s	1.28

For 10 cells in each spatial direction it is clear that what I assumed based on the structure of the Jacobian in Figure 4.6 is true; The *ForwardDiff* method takes much longer time than *ForwardAutoDiff* and MRST. Since *ForwardDiff* already spends over a minute for $10 \times 10 \times 10$ cells, it is omitted for finer discretizations. According to the results in Table 4.1, MRST is a bit faster than *ForwardAutoDiff*. We can also see from the column farthest to the right that the ratio between time spent using MRST and *ForwardAutoDiff* stays approximately constant as the complexity of the problem grows. This indicates that the two different solvers scales equally. It is, although, not possible to say if it is the AD or the linear solver part that makes the solver in MATLAB faster than the one in Julia. To get a better indication of what part of the code makes the difference, I have tested the speed of the linear solvers solving the system $\mathbf{Ax} = \mathbf{b}$ in MATLAB and Julia. The built in linear solvers use different methods based on the structure of the matrix. Hence, to make the test as relevant as possible, I have used the Jacobian from the first step of the flow solver as the matrix $\mathbf{A}$. The $\mathbf{b}$ vector is a randomized vector of values between 0 and 1. The results for the three different discretizations with 10, 20 and 30 cells in each spatial direction can be seen in Figure 4.7.

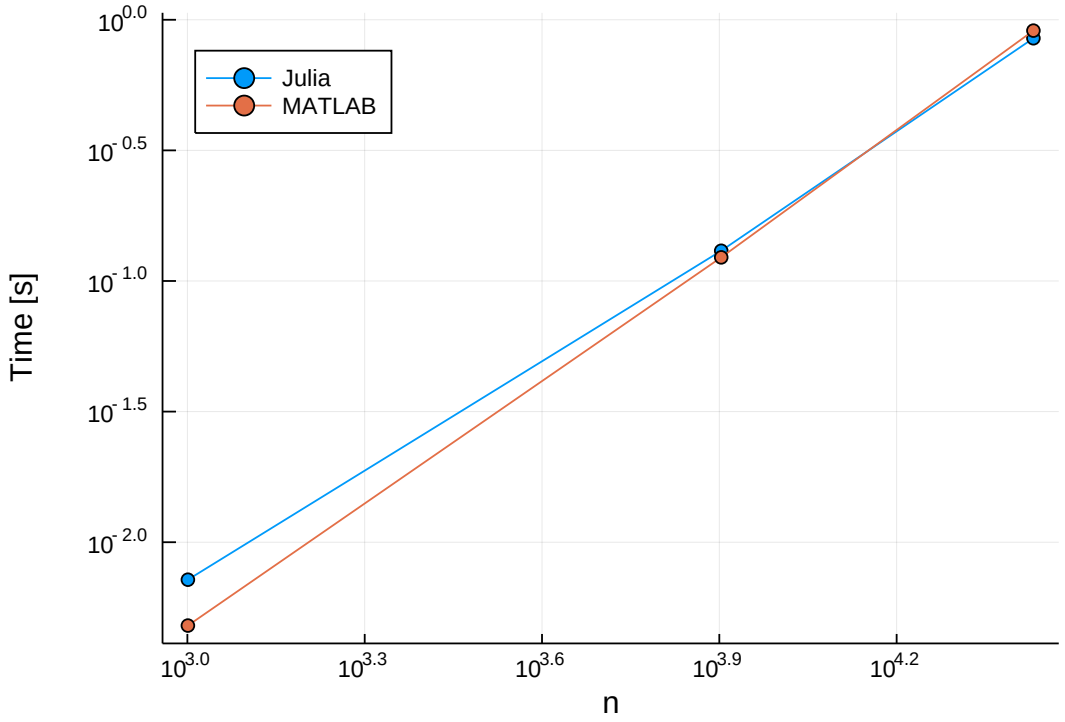


Figure 4.7: Computational time for solving the system $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x} \in \Re^n$ and $\mathbf{A} = \mathbf{J}$, the Jacobian from the first step in the flow solver.

We can see that the linear solvers perform very similar for all three cases, with MATLAB having small victories in computation time for 10 and 20 cells and Julia for 30. With this in mind, we can say that the linear solvers in MATLAB and Julia perform so similar that it is a valid assumption to assume that they perform equally well. Hence the speed differences in Table 4.1 is most likely because of speed differences in the AD-tools. This means that for the equations in the "Single-phase Compressible AD Solver" example, the *ForwardAutoDiff* implementation is approximately 30% slower than the AD tool in MRST.

Chapter 5

Conclusion and Future Work

This project report consisted of discussing the difference in method and advantages between forward and backward Automatic Differentiation (AD), the applications of AD, and implementation of forward AD in Julia. The implementation has been benchmarked against other AD libraries. Finally, an example where AD is used in a real world example for solving partial differential equations (PDEs) that describe the flow inside an oil reservoir has been provided.

My implementation of forward AD in Julia (*ForwardAutoDiff*) performs well in the benchmark for evaluating the value and the Jacobian of a standard vector function with three vector inputs. For vectors of lengths less than 1000, but more than 50, Figure 3.1b and Figure 3.1a show that it is the fastest implementation tested. It is interesting to see in the benchmarking (Figure 3.2) that there is an opportunity to gain computational efficiency by using for-loops in Julia. Currently, *ForwardAutoDiff* is implemented similarly as the one in MATLAB, using vectorization. MATLAB has the limitation of having a lot of overhead if you use for-loops – since Julia does not have this limitation, it would be interesting to look further into different implementations of AD in Julia to see if it is possible to make a more efficient implementation by a more extensive use of for-loops. Figure 3.1a shows that the built-in AD implementation in Julia (*ForwardDiff*) has very little overhead for small vectors, but scales badly as the size of the vectors increase. Even though it scales badly for long vectors, considering the low overhead for small problems, an interesting approach would be to use this library as a building brick of a new implementation of *ForwardAutoDiff*. Since Julia has proven itself to have little overhead when using for-loops, this may lead to a better implementation of *ForwardAutoDiff* in Julia.

For PDEs it has been demonstrated that by introducing discrete divergence and gradient operators we can solve the PDEs describing the flow of oil in a reservoir elegantly by using a finite volume method and AD. Using these operators we can write the discrete equations on a very similar form as the continuous. By setting up the equations as a function on residual form, we can solve it by using AD and the Newton-Raphson method to find the roots of the function. For the real world example in chapter 4, the AD implementation in MATLAB solves the problem quicker than the implementation in Julia for all discretizations tested. The discretization with fewest cells gives a system of 1002 equations. So even though *ForwardAutoDiff* was the fastest at evaluating a vector function (Figure 3.1a) for this length, it is still slower at solving the example. This indicates that when the function evaluated becomes more complex, and the Jacobian less sparse and diagonal, the AD tool in MATLAB handles it better than *ForwardAutoDiff*. Hence it would be interesting to see if, with new implementations, either with less vectorization or with using *ForwardDiff* as a building brick, the computational efficency of the flow solver example or other real world examples can be improved by using Julia.

Chapter 6

Drafts

6.1 Shared Libraries vs Static Libraries

The difference between shared libraries and static libraries is how they handle their dependencies. When static libraries are compiled all the code that is needed for the functions in the library is compiled into the library. This means that all the code that is needed for the library to function properly is "copied" to where it is needed. Shared libraries on the other hand has a reference to its dependencies. This will ad an extra cost to the execution since the code need to look up where the code it is supposed to run lies. The advantage of this compared to Static libraries is that the size of the library becomes smaller as we avoid replicates of code.

6.2 Profiling

Profiling is an effective method to obtain overview of where the bottlenecks lie in a code when trying to optimize its performance. The method consist of taking snapshots of the code with small time intervals and for each snapshot we register what function we are at and if relevant the functions that have called this function. In this way we obtain a register of how many times we have observed that we have been in every function. This will not give a perfect overview of how much time we spend in each function, and we even risk not registering all functions we use. But as 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 already is very fast.

6.3 Vectorization vs Non-vectorization

(White, 2013) explains how Julia is faster at executing devectorized code compared to vectorized code.

Bibliography

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

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

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

Lie, K.-A., 2018. An introduction to reservoir simulation using matlab: User guide for the matlab reservoir simulation toolbox (mrst). Accessed 25.10.2018.
URL https://folk.ntnu.no/andreas/mrst/mrst-cam.pdf

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

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

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

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

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

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