SPECIALIZATION PROJECT IN MATHEMATICAL SCIENCES

# Title

*Author:*
SINDRE GRØSTAD

*Supervisors:*
*Professor* KNUT-ANDREAS LIE

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING
DEPARTMENT OF MATHEMATICAL SCIENCES

December 12, 2018

# Preface

# Abstract

# Table of Contents

Chapter 1

# Introduction

Noe historie er beskrevet i Baydin et al. (2018).

Chapter 2

# Theory

## 2.1 Automatic differentiation

Automatic differentiation (AD) is a method that makes the computer derive the derivatives with very little effort from the user. If you have not heard of AD before, the first thing that you might think of is algebraic or symbolic differentiation. In this type of differentiation the computer learns the basic rules from calculus like e.g.

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

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 the disadvantage with this approach is that it is computational 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, that may occur i.e. because of machine precision, will be hugely magnified 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 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 the you get round-off errors. Hence the finite differences method is an unstable method 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 does 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 is 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 $f_x$ the numerical value of $f'(x)$. We then define the arithmetic operators for such pairs 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)

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

The only thing that remains to define 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 function are implemented you are able to calculate any scalar function derivative without actually doing any form of differentiation yourselves. Let us look at an step by step example where

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

Then the declaration of the AD-variable gives $x = [2\ ,\ 1]$. All scalars can be looked at 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 $f(x)$

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

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. Similar 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. $\epsilon$ 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 etc? If we look at the Taylor expansion of a function $f(x)$ where x is a dual number, we get

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

This means that to make dual numbers handle elementary functions, the first order Taylor expansion needs to be implemented. This equals the implementation 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:

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

7 is now 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 have $n$ 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 by calculating the derivative of $g_3$ with respect to $x$ in the same matter 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. In the same example with $f(x) = x \cdot \exp(2x)$ and with the same sequence of elementary functions like in (2.6) gives the expressions from the chain rule (2.7)

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

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

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

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! Comparing this to the method of Dual Numbers were 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 a function. The disadvantage of backward AD is that when we implement this, in differ from what we did in the example above when we did it by hand, we still only want to carry along the function- and the derivative values as we did in forward AD. This means that we have 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 reason why we here 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. Knut-Andreas Lie describes in *Lie (2018)* a method where 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 is 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

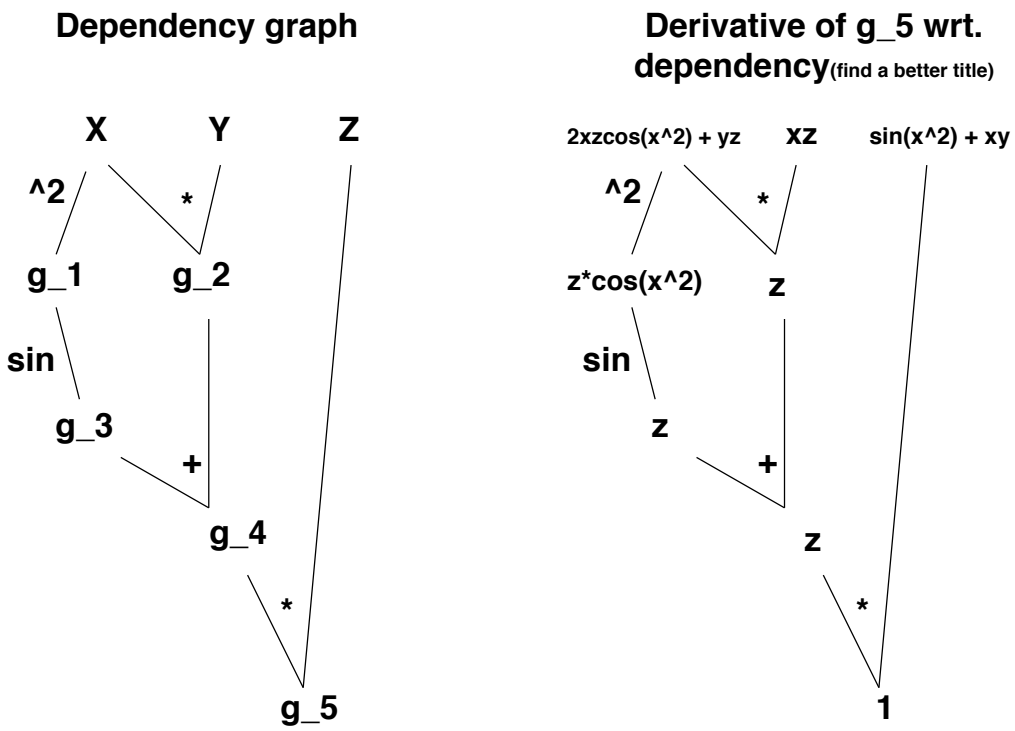



**Figure 2.1:** NOT A FINISHED FIGURE. Just a quick sketch of the idea I have of visualising the process of backward AD as i felt it got a bit messy with all the expressions. Gladly receiving comments on the thought/suggestions to make it more clear

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 store their derivative with respect to themselves, but the gradient. 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, but there will be two differences in particular. The first one is that the primal variables needs to be initialised with their Jacobians and not just a gradient vector. The Jacobian for a primary variable of dimension $n$ will be 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 operations instead of vector operations as seen above.

## 2.2 Applications of Automatic Differentiation

AD can be used in a wide spectre of applications, but one simple application of AD is if you want to find the roots of a function. The simplest example is for a scalar function $f$ with a scalar input $x$ then the iteration scheme

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

for an initial $x_0$, will converge to a root of $f$ given that it satisfies the assumptions made in the derivation of the formulae. This is called the Newton-Raphson method **is it necessary to have reference here?**. With AD this is super easy as you only have to define the function $f(x)$ and then the AD finds $f'(x)$ automatically and you can 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

$$\mathbf{A}\boldsymbol{x} = \mathbf{b}. \tag{2.8}$$

But instead of looking at it like in equation (2.8) 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 (2.8), 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

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n - \left[J_{\boldsymbol{F}}(\boldsymbol{x}^n)\right]^{-1} \boldsymbol{F}(\boldsymbol{x}^n). \tag{2.9}$$

Here $[J_{\boldsymbol{F}}(\boldsymbol{x}^n)]^{-1}$ is the inverse of the Jacobian of $\boldsymbol{F}$ at $\boldsymbol{x}^n$. For a simple linear system like equation (2.8) it may seem a bit forced and not necessary to make the effort to use AD to solve for $\boldsymbol{x}$. But for numerical application we can with this method easily solve non-linear equations by looking at the residual form and use the Newton-Raphson method (2.9) with AD. Consider the Poisson equation

$$-\nabla(\mathbf{K}\nabla u) = q, \tag{2.10}$$

where $\mathbf{K}$ is an diffusion coefficient and we want to find $u$ on $\Omega \in \Re^d$. 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 smaller cells $\Omega_i$ we can, instead of looking at the Poisson equation in differential form integrate it on each cell such that

$$\int_{\partial\Omega_i} -\mathbf{K}\nabla u \cdot \mathbf{n} ds = \int_{\Omega_i} q dA. \tag{2.11}$$

Here $\mathbf{n}$ is the unit normal to the cell $\Omega_i$, so equation (2.11) describes the conservation of mass in the cell $\Omega_i$ where total flux in and out of the boundary of $\Omega_i$ is equal to the total production in $\Omega_i$. 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 who shows two cells $\Omega_i$ and $\Omega_k$. 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 $\Gamma_{i,k}$.

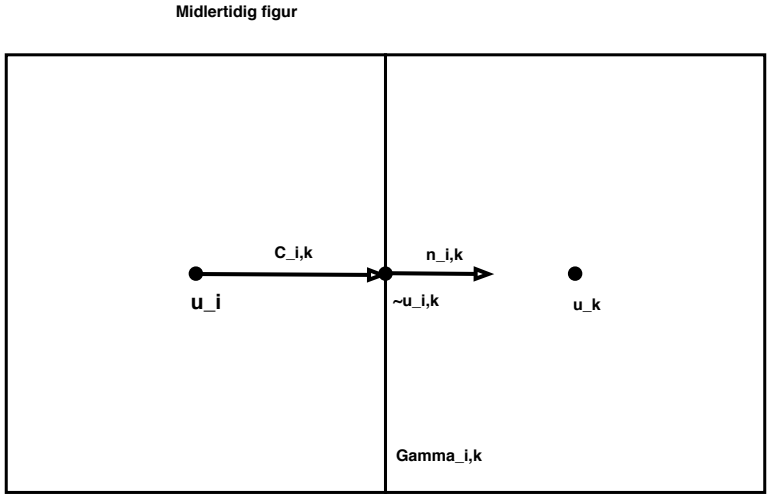



**Figure 2.2**

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

$$v_{i,k} = \int_{\Gamma_{i,k}} \mathbf{v} \cdot \mathbf{n}_{i,k} ds \tag{2.12}$$

If we let $L_{i,k}$ be the length of $\Gamma_{i,k}$, then the integral in (2.12) 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 $\Omega_i$, $u_i$, 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 . To handle this we first define what we call a transmissibility matrix where

$$T_{i,k}(\tilde{u}_{i,k} - u_i) = L_{i,k}\mathbf{K}_i \frac{(\tilde{u}_{i,k} - u_i)\mathbf{c}_{i,k}}{|\mathbf{c}_{i,k}|^2} \cdot \mathbf{n}_{i,k} \tag{2.13}$$

Now we know that the amount of flux from cell $\Omega_i$ to $\Omega_k$ must be the same as from $\Omega_k$ to $\Omega_i$ only with opposite sign. This gives us the relations $v_{i,k} = -v_{k,i}$ and from figure 2.2 it is clear 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 subtraction the two equations for $v_{k,i}$ and moving $T_{i,k}$ and $T_{k,i}$ to the other side

$$\begin{aligned}(T_{i,k}^{-1} + T_{k,i}^{-1})v_{i,k} &= (\tilde{u}_{i,k} - u_i) - (\tilde{u}_{i,k} - u_k) \\ v_{i,k} &= (T_{i,k}^{-1} + T_{k,i}^{-1})^{-1}(u_k - u_i) \\ v_{i,k} &= T_{i,k}(u_k - u_i)\end{aligned} \tag{2.14}$$

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 boundary between $\Omega_i$ and $\Omega_k$ we get that equation (2.12) can be approximated by

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

Where $q_i$ is the total production in cell $\Omega_i$. Now we can get a linear system of the form **Au** = **b**, which on residual form becomes **F**(**u**) = **Au** - **b** = 0 where

$$\mathbf{A}_{i,j} = \begin{cases} \sum_k T_{i,k} & \text{if } j = 1 \\ -T_{i,j} & \text{if } j \neq i \end{cases}$$

Now we can solve the poisson equation (2.10), using the scheme explained in (2.9) and by having $u$ as an AD-variable. But we still end up with a linear system of equations that we may as well solve without AD.

To show the real elegance of using AD to solve PDE's we want to create a framework where 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 grid we have. Instead of the simple two-cell grid we used in figure 2.2 consider a more complex grid like we have in figure 2.3. The figure shows a part of some larger grid where we will focus on the relation between cell 5 and 8. To define the discrete divergence and gradient operators we need som information from the grid. The grid consist of three types of objects: cells, facets and nodes. The cells are each $\Omega_i \subset \Omega$ and in our two-dimensional case the facets are simply the lines between each cell. The nodes are defined points based on the method we use where

we want to find the value of the function we search. In the case of figure 2.2 we had two nodes, $u_i$ and $u_k$. Each cell and facet has physical properties like area or length and centroid or centre. The facets also has a normal vector. Figure 2.3 shows how we have two different mappings that explains the relation between the cells and the facets. The first mapping $F(c)$ is mapping from cell $c$ to facet $f$. 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.

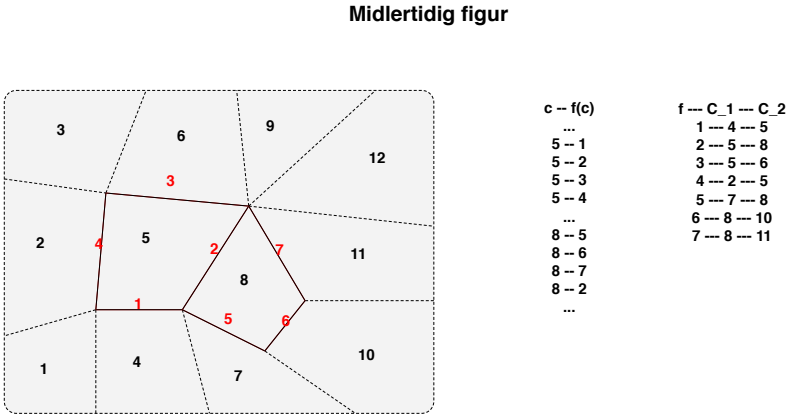



**Figure 2.3**

Our goal now is to, based on the grid properties, create the divergence and gradient operators, corresponding to this grid. Consider the Poisson equation (2.10) for the function $u$. Then the gradient operator for a facet $f$ is defined as

$$\texttt{gradient}(u)[f] = u[C_2(f)] - u[C_1(f)]$$

when $u[C_i(f)]$ 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 (2.14). Let $v_{i,k} = v[f]$ when 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 a cell $c$ is defined as

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

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

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

Hence we can now only based on the topology of the grid create discrete divergence and gradient operators such that the Poisson equation (2.10) can be written as

$$\mathbf{F}(\mathbf{u}) = \texttt{divergence}(\mathbf{v}) - \mathbf{q} = 0$$

with $\mathbf{v} = \mathbf{T}\ \texttt{gradient}(\mathbf{u})$ where $\mathbf{T}$ is the transmissibility matrix defined in (2.13). 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 and although the system becomes non-linear it will be easy to solve using AD and Newton-Raphson method. An example of this can be seen in section 3.4.

Chapter 3

# Implementation

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

So in short, it seems to be the perfect language for numerical applications. There are already some packages in Julia for AD. Most of them are backward AD packages designed for machine learning like for example AutoGrad *(Yuret, 2016)* and Zygote *(Innes, 2018)*, but there is one package called ForwardDiff *(ForwardDiff Package, n.d.)* that are using forward AD that are being developed by the Julia community. This package works very well for some applications, but for others it has some limitations that is not ideal i.e.

- The function we want to differentiate only accepts a single argument. This is possible to work around such that 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 fine, it is not optimal and it can cause unreadable code.
- The function we want to differentiate must be on the form of a generic Julia function i.e $g(x) = 3x.*x$. Here $x.*x$ symbolise element wise multiplication. This means that if we have a function like $h(x) = 3x.*x + \text{sum}(x)$ where all elements in $g(x)$ is added with the sum of all elements in $x$, it will not be possible to use ForwardDiff to obtain the Jacobian.
- The Jacobian calculated is a full matrix. In some calculations when the Jacobian is dense anyway, this will not have any major effect, but in many numerical applications the Jacobian will be

sparse. By working with a sparse matrix on a full matrix structure we will loose a lot of potential computation efficiency.

## 3.2 Implementation of Automatic Differentiation

When it comes to implementing Automatic Differentiation(AD) there are two major concerns. First is that it must be easy and intuitive to use, the second is that 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 struct that have two member variables, `val` and `jac`, that stores respectively the value and the corresponding Jacobian. The importance of the way you implement the AD operators and elementary functions can be expressed in a short example: Consider you have two variables $x$ and $y$ and 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 and should be avoided. Instead of making new functions that takes 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 of Multiple Dispatch that satisfies our needs is that at runtime, the compiler understands what types are given as input for either a operator or a function and chooses the correct method based on this. 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 (Base is where the standard functions in Julia lies), and overloads it for AD variables. This means that it is only when there are AD variables on both sides of the operator that this implementation is used. Hence if I declare $x = 1$, $y = 3$ and then $z = x + y$, then Julia understands that it is not the definition above, but the normal addition for integers it should use. But if we declare $x, y = \texttt{initiateADVariables}(1,3)$ such that $x$ and $y$ both are AD variables, then when we write $z = x + y$, Julia's Multiple Dispatch will understand that it is our definition of the "+" operator that is meant to be used. 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

function necessary it gives us the opportunity for the function f above to only write $f = y + \exp(2 * x * y)$ and Julia will understand that it is our implementation of +, * and exp that shall be used, and $f$ will become an AD-variable with the correct value and derivatives.

Another advantage of Julia's Multiple dispatch system is clear 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
```

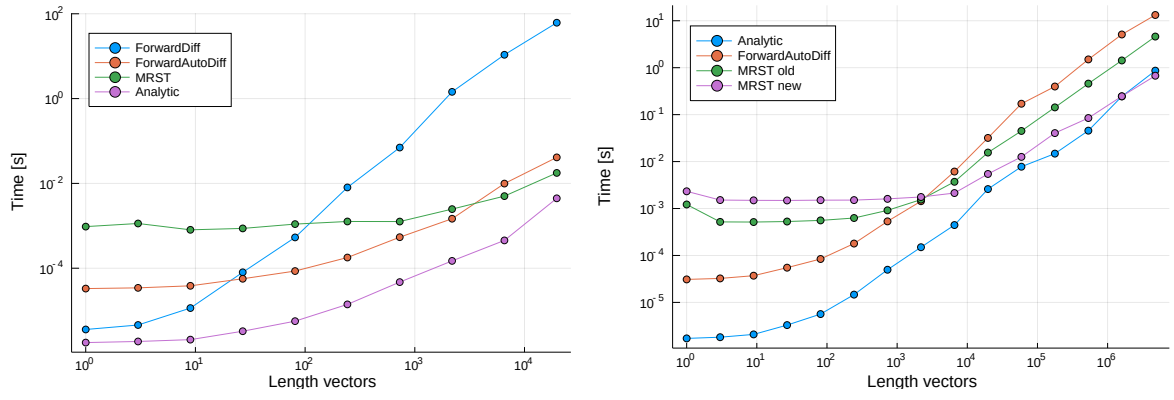
which would indeed work, but to exploit Julia's Multiple Dispatch fully we can instead overload the `iterator` function. This function explains how we shall iterate through an AD variable. Now the built in `sum` function will work on AD variables since it knows how to iterate through the variable and when it adds up the values, the + operator we defined above is being used. And not only that! All built in functions that iterates through the input will also work (given that the functions it uses 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.

## 3.3 Benchmark of Automatic Differentiation

As mentioned in section 3.1, there are already an AD library in Julia called ForwardDiff *(ForwardDiff Package, n.d.)* that uses forward AD. Hence it would be interesting to see how the two implementations compares when the functions evaluated are getting larger. As another reference I have added the AD implementation in MATLAB Reservoir Simulation Toolbox (MRST) *MRST Homepage (n.d.)* to the benchmark, to see how the Julia implementations compare to an optimised 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 time spent calculating the function value and the Jacobian of the function, for the different methods, scales as the length of the vectors $n$ increases. In figure




**(a)** **(b)**

**Figure 3.1:** Time spent calculating the value and Jacobian of $f$ in (3.1) as a function of length of the input vectors.

3.1a we have four different graphs. The analytic graph is simply the evaluation of $f(x, y, z)$, $f_x$, $f_y$

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. 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. We can also observe that for small vectors, MRST and ForwardAutoDiff have much more overhead than ForwardDiff and the analytic solution and hence are slower, but as $n$ grows, this overhead becomes more neglectable. Since both MRST and ForwardAutoDiff approaches the analytic evaluation as $n$ grows, it is 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 which is specially optimised for element operations like we have when evaluating the function in (3.1). 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 specially efficient on functions like in (3.1), but if we for example want to calculate something like

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

the performance is more similar to the other MRST implementation. Other than that we can see that the trend seen in figure 3.1a concerning the speed difference of the older MRST implementation and ForwardAutoDiff continues for longer vectors.

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 if we can increase computational efficiency of the evaluation of the vector function in (3.1) by evaluating it scalar by scalar in a loop instead of 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`. and the graphs with squares are the same methods only they are tested with the implementation in function `benchmarkADinLoop`. We can start by observing that the ForwardAutoDiff implementation is clearly not optimised for evaluating the vector function scalar by scalar as it is the slowest method we have tested so far for all vector lengths. The next interesting observation is that we can make ForwardDiff's evaluation of the vector function much more efficient. When using the method in `benchmarkADinLoop` we almost achieve the same test results as ForwardAutoDiff. Although, what is important to mention here is that with the approach in `benchmarkADinLoop` we only obtain the gradient of the function and not the Jacobian. In the particular case of the function f in (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 (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. The implementation necessary to obtain the Jacobian with ForwardDiff and `benchmarkADinLoop` will slow the computation down and for a vector function with large input vectors, ForwardAutoDiff is a better approach.

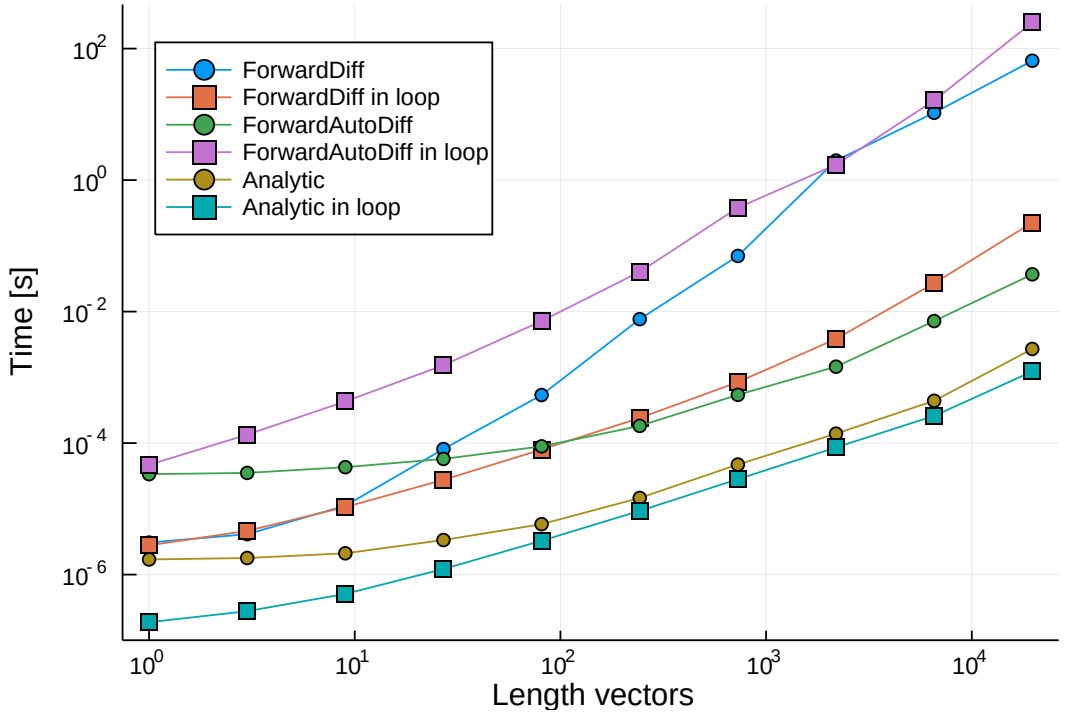



**Figure 3.2:** Time spent calculating the value and gradient of $f$ in (3.1) as a function of length of the input vectors. **TODO: få plottene som tilhører samme metode i samme farge**

Other than this, what is interesting to see is that the evaluation of the analytical solution in a loop is faster than the vectorised. Here Julia shows a real strength compared to MATLAB where a function evaluation like for the vector function f will be much slower in a loop than vector multiplication. **TODO: Legge med testeksempel av matlab-forløkke kode?**

## 3.4 Flow Solver With Automatic Differentiation

To test AD 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". So 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 and 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 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 it is 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 said, Julia is meant to be a language as familiar to mathematical notations as MATLAB, but as fast as C. Hence it is interesting to figure out how Julia can perform compared to MRST.

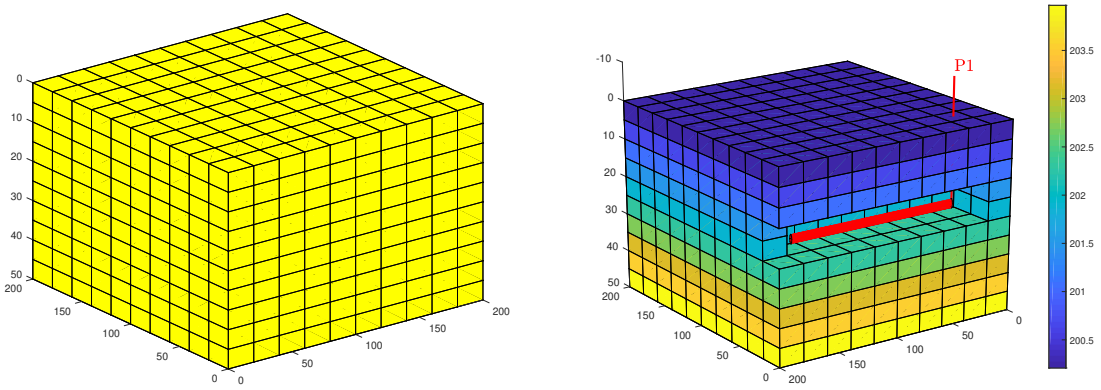
To compare different AD implementations in Julia and MATLAB I have implemented MRST's tutorial called Single-phase Compressible AD Solver from `flowSolverTutorialAD.m` *(Single-phase*

*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 to compare the implementation of AD in MATLAB and Julia.

The example consist of modelling the pressure within a rectangular reservoir measuring $200 \times 200 \times 50\text{m}^3$. That it is a single-phase solver only means that we do not have different phases of the 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. The grid of the reservoir can be seen in figure 3.3a. The initial pressure is calculated by solving

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

for the fluid density given by $\rho(p) = \rho_r \cdot \exp\left(c \cdot (p - p_r\right)$ for some reference constant $\rho_r$ and $c$. The well is then inserted by removing 8 grid elements. The grid with initial pressure and well can be seen in figure 3.3b. After initializing the grid we define the governing equations for the flow in the reservoir.




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

**Figure 3.3**

We use a finite volume method to discretize in space and a backward Euler method to discretize in time. To solve the equations we use the Newton-Raphson method described in equation (2.9) with the residual form $\boldsymbol{F}(\boldsymbol{x}) = 0$. If we take a look at figure 3.4, we can see the structure of the Jacobian of $\boldsymbol{F}$. 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 (can look like 5 from perspective) with nonzero elements and the rest of the elements are 0. As we also can see from the figure, there are only 6419 non-zero elements out of more than 1 million. It is clear that storing the full $1002 \times 1002$ matrix will be very inefficient. In MRST 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 much more efficient than storing the full $1002 \times 1002$ Jacobian as ForwardDiff does.

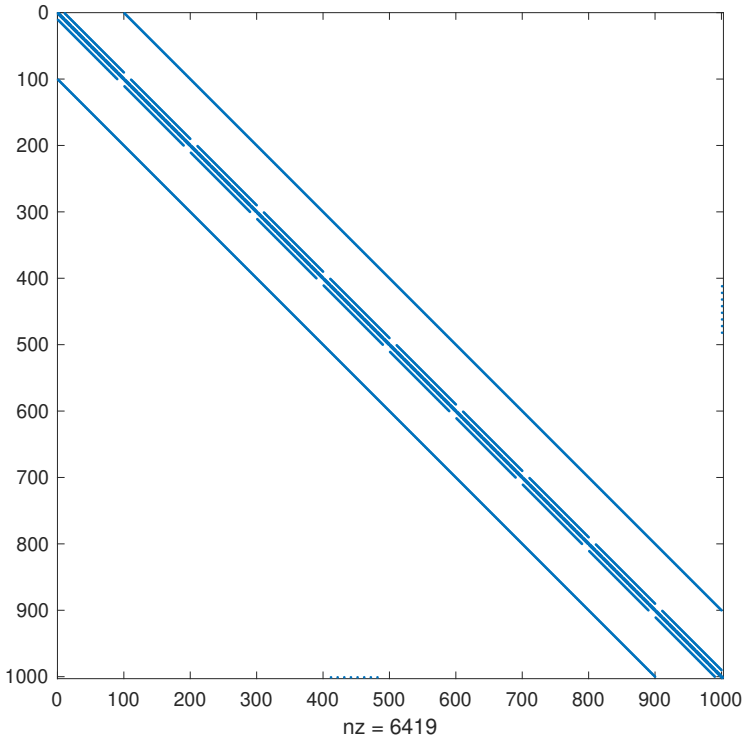



**Figure 3.4:** Structure of the 1002 × 1002 Jacobian. There are 6419 non-zero elements.

Chapter 4

# Result and discussion

Chapter 5

# Conclusion and Future Work

# 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

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

ForwardDiff Package, n.d. Accessed 21.11.2018.
URL http://www.juliadiff.org/ForwardDiff.jl/stable/

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

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

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

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

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