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

March 7, 2019

Table of Contents

Chapter 1

Theory

1.1 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.

1.1.1 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,

$$\begin{aligned}
&[f,\ f_x] \pm [g,\ g_x] = [f \pm g,\ f_x \pm g_x],\\
&[f,\ f_x] \cdot [g,\ g_x] = [f \cdot g,\ f_x \cdot g + f \cdot g_x],\\
&\frac{[f,\ f_x]}{[g,\ g_x]} = \left[\frac{f}{g},\ \frac{f_x \cdot g - f \cdot g_x}{g^2}\right].
\end{aligned} \tag{1.1}$$

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

$$\begin{aligned}
&\exp\Big([f,\ f_x]\Big) = [\exp(f),\ \exp(f) \cdot f_x],\\
&\log\Big([f,\ f_x]\Big) = \Big[\log(f),\ \frac{f_x}{f}\Big],\\
&\sin\Big([f,\ f_x]\Big) = [\sin(f),\ \sin(f) \cdot f_x], \text{etc.}
\end{aligned} \tag{1.2}$$

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

$$f(x) = x \cdot \exp(2x) \qquad \text{for} \quad x = 2. \tag{1.3}$$

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 $x = 2$.

1.1.2 Dual Numbers

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

$$a + b\epsilon. \tag{1.4}$$

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 (1.4). 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

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

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 (1.2).

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.

1.1.3 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

$$\frac{\partial f}{\partial t} = \sum_i\left(\frac{\partial f}{\partial u_i}\cdot\frac{\partial u_i}{\partial t}\right). \tag{1.5}$$

Take $f(x) = x\cdot\exp(2x)$, like in the forward AD example (1.3). We then split up the function into a sequence of elementary functions

$$x, \qquad g_1 = 2x, \qquad g_2 = \exp(g_1), \qquad g_3 = x\cdot g_2, \tag{1.6}$$

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 (1.5)

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

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

$$\frac{\partial s}{\partial u} = \sum_i \left(\frac{\partial f_i}{\partial u} \cdot \frac{\partial s}{\partial f_i} \right), \tag{1.7}$$

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 (1.6), the expressions from the chain rule (1.7) 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

$$g_1 = x^2, \quad g_2 = x \cdot y, \quad g_3 = \sin(g_1), \quad g_4 = g_2 + g_3, \quad g_5 = z \cdot g_4. \tag{1.8}$$

Now the derivatives from the chain rule in Equation (1.7) becomes

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

substituting s with g_5 yields

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

The calculation of the derivatives together with a dependency graph can be seen in Figure 1.1. This shows that we get all the derivatives of $f(x) = g_5$ with a single function evaluation!

Figure 1.1: Graphs to visualize the process of backward AD. To the left is a dependency graph of the elementary functions in (1.8) 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 1.1.2 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 1.1. 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.

1.1.4 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 alternative ways of implementing this rather than implementing with dual numbers and evaluating the the function n times. Neidinger describes in *Neidinger (2010)* a method where we calculate all the derivatives in one function evaluation. To illustrate the method, consider a scalar function $f : \Re^n \to \Re$, that we want to obtain the gradient of. Then, the main idea is that we define what we call our *primary variables*. This is all the variables in the space that we are currently working in. Each primary variable is an AD-variable containing the derivatives of itself with respect to all the other primary variables. Say we have three variables x, y and z, and for any function $f(x,y,z)$ we are interested in finding the gradient of f, $\nabla f = (f_x, f_y, f_z)^\top$. To achieve this, we define the corresponding AD-variables

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

Each primary AD-variable now not only stores its derivative with respect to itself, but also the gradient with respect to all other primary variables. The operators defined in Equations (1.1) and the elementary functions in Equations (1.2) 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

$$\begin{aligned} xyz &= [1,\ (1,0,0)^\top]\cdot[2,\ (0,1,0)^\top]\cdot[3,\ (0,0,1)^\top] \\ &= [1\cdot2\cdot3,\ 2\cdot3\cdot(1,0,0)^\top + 1\cdot3\cdot(0,1,0)^\top + 1\cdot2\cdot(0,0,1)^\top] \\ [f,\ \nabla f] &= [6,\ (6,3,2)^\top]. \end{aligned}$$

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.

1.1.5 Forward Automatic Differentiation With Vector Functions

In numerical applications, as we are dealing with discretizations, the functions we evaluate are usually vector functions and not scalar functions. Hence $f : \Re^n \to \Re^m$. Neidingers method still applies, only that the primary variables are now vectors and instead of containing the gradient of itself with respect to all the other primary variables, we now have the Jacobian. For a function f the Jacobian with respect to its n primary variables are given as

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

The forward AD method described above will be similar for a vector function as it was for a scalar function above, except for two important differences. Going from scalar functions with multiple parameters to vector functions induce to 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 example. As a simple illustration of the differences, consider the vector function $f = 2\cdot\mathbf{x}\cdot\mathbf{y}$ for the primary variables $\mathbf{x},\mathbf{y} \in \Re^3$. Again, all multiplications are element-wise. The initialization of the primary variables gives

$$\mathbf{x} = \left[\begin{pmatrix}1\\2\\3\end{pmatrix},\ \begin{pmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\0&0&1&0&0&0\end{pmatrix}\right], \qquad \mathbf{y} = \left[\begin{pmatrix}4\\5\\6\end{pmatrix},\ \begin{pmatrix}0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\end{pmatrix}\right]. \tag{1.9}$$

Here we have decided an order of the variables in the Jacobian, and further on we need to be consistent with this order. The function value of $\mathbf{x}\cdot\mathbf{y}$ is found by normal element-wise multiplication. The Jacobian of $\mathbf{x}\cdot\mathbf{y}$ is obtained by using the chain rule as defined in (1.1). The difference is now that we have element-wise multiplication of a vector and a matrix instead of only scalars or scalars and vectors. Element-wise multiplication of a vector and a matrix corresponds to transforming the vector to an $n \times n$ matrix with the values on the diagonal. The calculations gives

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

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

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

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

1.2 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.

1.2.1 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

$$\mathbf{A}\boldsymbol{x} = \mathbf{b}. \tag{1.10}$$

But instead of looking at it like in equation (1.10), 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 (1.10), we need to find the root of $\boldsymbol{F}(\boldsymbol{x})$. This can be done by choosing an initial value $\boldsymbol{x}^0$ and observe that since $\boldsymbol{F}(\boldsymbol{x})$ is linear, this will converge in one step using the multivariate Newton-Raphson method. The general form of the multivariate

Newton-Raphson method is given by

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^{n} - J_{\boldsymbol{F}}(\boldsymbol{x}^{n})^{-1}\boldsymbol{F}(\boldsymbol{x}^{n}). \tag{1.11}$$

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

1.2.2 Solving the Poisson Equation

For a simple linear system like Equation (1.10) it may seem a bit forced and unnecessary to make the effort of using AD to solve for $\boldsymbol{x}$. We could just as well have used some built in linear solver 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 (1.11)) with AD. As a preface to introducing how we will do this, consider the Poisson equation

$$-\nabla(\mathbf{K}\nabla u) = q, \tag{1.12}$$

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

$$\int_{\partial\Omega_i} -\mathbf{K}\nabla u \cdot \mathbf{n}\,\mathrm{ds} = \int_{\Omega_i} q\,\mathrm{dA}. \tag{1.13}$$

Here, $\mathbf{n}$ is the unit normal to the cell Ω_i, so Equation (1.13) describes the conservation of mass in the cell Ω_i, where total flux in and out of the boundary of Ω_i is equal to the total 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 1.2, 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 $\Gamma_{i,k}$.

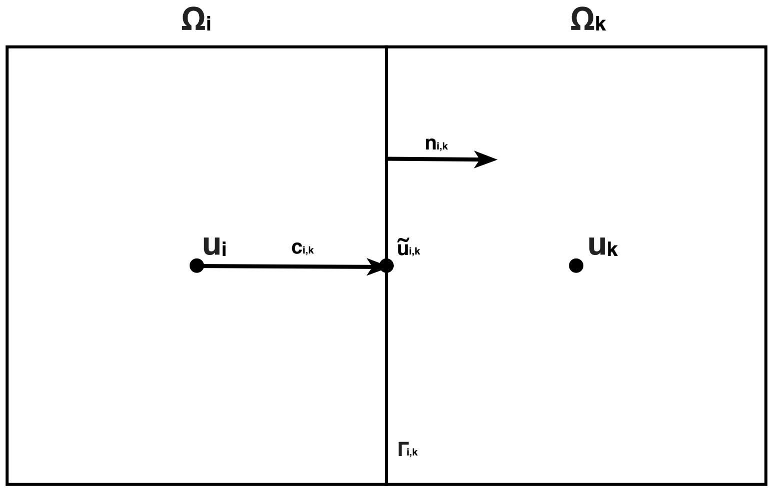


Figure 1.2: Figure of two adjacent cells Ω_i and Ω_k. The average value of the cell is given by u_i. The boundary between the cells is $\Gamma_{i,k}$ with value at the centre equal $\bar{u}_{i,k}$ and outward normal vector $n_{i,k}$

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

$$v_{i,k} = \int_{\Gamma_{i,k}} \mathbf{v} \cdot \mathbf{n}_{i,k} \mathrm{d}s. \tag{1.14}$$

If we let $L_{i,k}$ be the length of $\Gamma_{i,k}$, then the integral in (1.14) 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 1.2. For brevity, we first define what we call a transmissibility

$$T_{i,k}(\tilde{u}_{i,k} - u_i) = L_{i,k} \mathbf{K}_i \frac{(\tilde{u}_{i,k} - u_i)\mathbf{c}_{i,k}}{|\mathbf{c}_{i,k}|^2} \cdot \mathbf{n}_{i,k}. \tag{1.15}$$

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

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

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

$$\begin{aligned} (T_{i,k}^{-1} + T_{k,i}^{-1}) v_{i,k} &= (\tilde{u}_{i,k} - u_i) - (\tilde{u}_{i,k} - u_k) \\ v_{i,k} &= (T_{i,k}^{-1} + T_{k,i}^{-1})^{-1}(u_k - u_i) = T_{ik}(u_k - u_i) \end{aligned} \tag{1.16}$$

we manage to eliminate $\tilde{u}_{i,k}$ and get a computable expression for the gradient of u. This is called the two-point flux-approximation (TPFA) *(Lie, 2018)*. Now that we have an approximation of the flux through the interface between Ω_i and Ω_k, we get that Equation (1.14) can be approximated by

$$\sum_k T_{i,k}(u_k - u_i) = q_i, \qquad \forall \Omega_i \in \Omega, \tag{1.17}$$

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

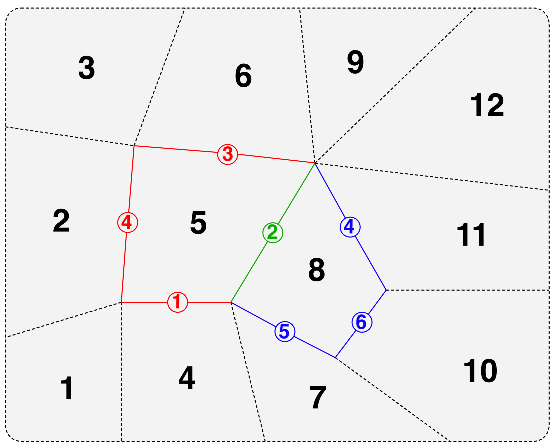
$$\mathbf{A}_{i,j} = \begin{cases} \sum_k T_{ik} & \text{if } j = i \\ -T_{ij} & \text{if } j \neq i. \end{cases}$$

This means we can solve the Poisson equation (1.12), using the scheme explained in (1.11) and by having u as an AD-variable. For this simple Poisson Equation we still only end up with a linear system of equations that we may as well solve without AD. But for more complex PDE's and especially more complex grids, the construction of the matrix $\mathbf{A}$ becomes more tricky and the ease of using AD, together with the discrete differentiation operators, that we will define in subsection 1.2.3, becomes clearer.

1.2.3 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 1.2, we now consider a a general polygonal grid. Figure 1.3 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 1.2 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.



c	f(c)
⋮	⋮
5	1
5	2
5	3
5	4
⋮	⋮
8	5
8	6
8	7
8	2
⋮	⋮

f	C_1(f)	C_2(f)
1	4	5
2	5	8
3	5	6
4	2	5
5	7	8
6	8	10
7	8	11
⋮	⋮	⋮

Figure 1.3: Figure of a general polygonal grid with the mapping cell to facets and facet to cells.

Figure 1.3 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 (1.17). From these, we want to create discrete divergence and gradient operators that correspond to the continuous equivalents for this grid. Consider the Poisson equation (1.12) for the function u. Then the discrete gradient operator for a facet f is defined as

$$\texttt{dGrad}(u)[f] = u[C_2(f)] - u[C_1(f)], \tag{1.18}$$

where $u[C_i(f)]$ is the value of u at the cell corresponding to $C_i(f)$. For the divergence operator, we remember the expression we found for the flux through a facet in equation (1.16). Let $v_{i,k} = v[f]$, where f is the facet between cell i and cell k. Since the divergence in a cell is the same as the sum of flux leaving and entering the cell, the discrete divergence operator for cell c is defined as

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

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

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

Here, **T** is the transmissibility defined in (1.15). 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 2

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.

2.1 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 1.1.3, backward AD is much more efficient than forward AD in these types of evaluations. For numerical applications there is usually many input parameters and output parameters. Hence there is no clear advantage of using backward AD compared to forward AD. In Julia 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 1.1.2 extended to multiple dimensions. This is called multidimensional dual numbers and a vector $\mathbf{x}$ of length n is represented as

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

This package works very well for some applications, but for especially numerical applications 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 element-wise multiplication. This means that if we have a function like $h(x) = 3x * x + \text{sum}(x)$, where all elements in $g(x)$ are added with the sum of all elements in x, it will not be possible to use *ForwardDiff* to obtain the Jacobian. This limitation also prevents us from having a function that we evaluate for all points, and then add boundary conditions for some points later. This is a feature that is essential for most numerical analysis.

- 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.

2.2 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:

```
struct AD
    val
    jac
end
```

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

MRST *(Lie, 2018)*. In the first implementation the Jacobian, `jac`, is represented as a vector of sparse matrices. Each element in the vector is a sparse matrix that represent 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. Before introducing the second implementation of the Jacobian, I will continue to explain the implementation of what I will call `ForwardAutoDiff`(`FAD`), which is defined with the following struct:

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

To obtain a working AD library we need to implement operators for the `FAD` 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 `FAD` variables x and y and that you want to compute the function $f(x,y) = y+\exp(2xy)$. If the implementation is based on making new functions that take in `FAD`-variables as input parameters, it will for the evaluation of f look something like this:

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

This is clearly not a suitable way to implement AD as it quickly becomes difficult to see what type of expression it is. If you did not know what type of function f is, it would take you quite some time to figure it out. And more importantly, the possibility for human error becomes very large when you have to write unreadable code like this. This approach should be avoided. Lie and Neidinger suggests in *Lie (2018)* and *Neidinger (2010)* a much more elegant implementation, where instead of making new functions that take in `FAD`-variables as parameters, one should overload the standard operators (+,-,*,/) and the elementary functions (exp, sin, log, etc.). This is where the elegance of having a custom `FAD` struct. In Julia we can use *multiple dispatch* to call our implementation of standard operators and elementary functions when they are used on `FAD` structs. A quick explanation of *multiple dispatch* that satisfies our needs is that 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::FAD, B::FAD)
    return FAD(A.val + B.val, broadcast(+, A.jac, B.jac))
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 `FAD` variables. For compactness purposes I have removed checks and edge cases and only left the method for `FAD` variables with equal length. The broadcast function add each Jacobian for each primary variable together. This implementation of the + operator is only used when there are `FAD` variables on both sides of the operator. Hence, if $x = 1$, $y = 3$ and then $z = x + y$ is computed, Julia understands that it is not the definition above, but the normal addition for integers it should use. But if $x, y =$ `initialize_FAD`(1,3) is declared, so that x and y both are `FAD` variables, then Julia's multiple dispatch will understand that the new definition of the "+"-operator should be used on the expression $z = x + y$. What we need to remember is that if I now write $z = x + 3$, with x as an `FAD` variable, Julia will deploy an error message. This is because we also have to implement

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

Here, the first function will be used if the + operator is used with an FAD variable on the left hand side and a number on the right. The last line is a compact way of writing the opposite function, which will be used when the number is on the right hand side. 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 * operators and the exponential function that shall be used. f will now become an FAD-variable with the correct value and derivatives.

Up until now I have only discussed implementation of FAD 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 **??**, 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::FAD)
    ## Overload sum
end
```

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

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

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

2.2.1 Element-wise and Vector Multiplication

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 in v is multiplied by the corresponding row value of u. An attempt to evaluate $u*u$ will end in an error message saying that "the dimensions does not match matrix multiplication". The ".*" operator however, is the element-wise 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 element-wise 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 element-wise ".*" 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 element-wise multiplication in this project, I have decided to implement "*" as element-wise multiplication. This means that if I have written regular multiplication expressions consisting of at least one AD-variable, it is element-wise multiplication that is being executed.

2.2.2 Optimizing `ForwardAutoDiff`

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

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

It represents a matrix with three vectors and two integers. The integers represents the size of the matrix and the three vectors represents all non-zero elements in the matrix. Two explain how the vectors work, consider the example of a matrix **A** with the corresponding `SparseMatrixCSC` struct variables:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 5 & 0 \\ 0 & 3 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 7 \\ 2 & 4 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{aligned} \texttt{m} &= 4 \\ \texttt{n} &= 5 \\ \texttt{colptr} &= [1,3,5,5,7,8] \\ \texttt{rowval} &= [1,4,2,4,1,2,3] \\ \texttt{nzval} &= [1,2,3,4,5,6,7]. \end{aligned}$$

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

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

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

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

which indicates that there are no non-zero elements in this column.

This way of storing a matrix will decrease both memory usage and computational efficiency dramatically when working with large and sparse matrices compared to storing the full matrix. When performing operations on the matrix, e.g. an element-wise vector matrix multiplication, the computational gain comes from the opportunity to neglect all zero values. If we store the matrix as a full dense matrix then we have to compute a lot of multiplications that ends up being zero that we with a sparse matrix structure could avoid computing. The method however brings some extra work that consist of doing numerous checks to make sure that we have done the multiplication correctly. Take the example from subsection 1.1.5 where $f = 2 \cdot \mathbf{x} \cdot \mathbf{y}$ for the primary variables $\mathbf{x}, \mathbf{y} \in \Re^3$ as in Definition (1.9). Initializing the primary variables as `FAD`-structs gives

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

Here I have written out the Jacobians as full matrices for better visualization although in theory they will be stored as `SparseMatrixCSC` structs. In the multiplication of the two `FAD`-variables, the element-wise multiplication of the values are not interesting and we will focus on how we obtain the new Jacobian for $\mathbf{x} \cdot \mathbf{y}$. Similarly as in subsection 1.1.5 we find the new Jacobian using the product rule, but now we can separate the operations into two calculations. First we have the Jacobian for the primary variable $\mathbf{x}$:

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

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

$$\begin{pmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}. \tag{2.1}$$

What we immediately observe is that two of the multiplications are unnecessary since one of the matrices are the null matrix. Remember then that the left matrix in all the calculations above actually is a vector that we have transformed into a diagonal matrix to illustrate how the element-wise multiplication happens. This means that when the previous Jacobian is a diagonal matrix, what we actually can do to obtain the new Jacobian, is element-wise multiplication between the vector on the left hand side and the vector that is the diagonal on the Jacobian. And even better, when the previous Jacobian is the identity matrix, like in calculation (2.1), the product is simply a diagonal matrix with the left hand side vector on the diagonal. This is where the idea of optimizing `FAD` comes from. By knowing what type of Jacobian we have at all times we can sometimes take safe shortcuts in our calculations.

2.2.3 Custom Jacobian Automatic Differentiation

Custom Jacobian Automatic Differentiation(CJAD) is what I have called the optimized FAD. *Custom Jacobian* comes from having four different types of Jacobians. The structure of CJAD is similar to FAD, except that the Jacobians are in a vector of type CustomJac:

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

CustomJac is an abstract type that has four structs that extends it:

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

Now it is time for Julia's multiple dispatch to shine. Since Julia understands at runtime what type of CustomJac we have, whether it is a NullJac, IdentityJac, DiagJac or SparseJac, we can implement different methods for all possible combinations. Since we in each implementation know what type of matrix we are dealing with, we can optimize the performance. Take the calculation of the Jacobian w.r.t the primary variable **y** in (2.1). First we have a vector multiplied element-wise by a null matrix. The implementation

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

efficiently knows immidietly that there is no need to do any calculations, the result will be a null matrix of the same size as before. The second calculation is a vector multiplied element-wise by an identity matrix. The result will be a diagonal matrix with the vector on the diagonal. The implementation

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

automatically knows the result of this and makes a diagonal Jacobian with the vector on the left hand side without doing any calculations. Finally we have to add a null matrix and a diagonal matrix, and since there is really no need of doing the adding of zero number, the implementation simply returns the diagonal matrix:

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

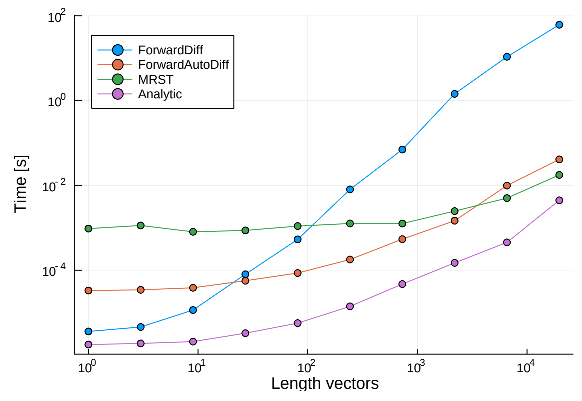
All the implementations of operators on structs extending the CustomJac type are called from outer functions that check the legality of the operations. Hence there are no checks in the presented implementations.

2.3 Benchmarking Automatic Differentiation

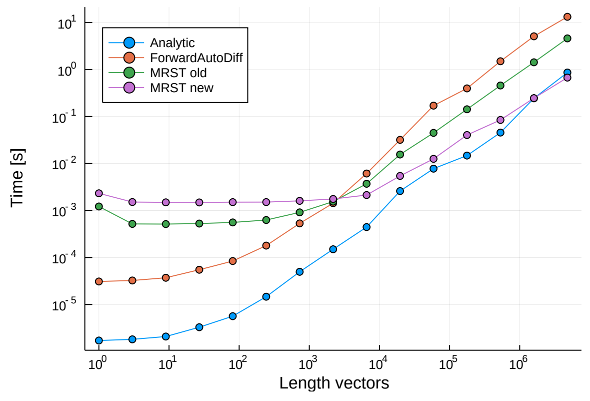
As mentioned in section 2.1, Julia already has an AD library called *ForwardDiff (Revels et al., 2016)* that uses forward AD. Hence, it would be interesting to see how the different implementations compares when the functions evaluated are getting larger. As another reference, I have added the AD implementation 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

$$f(x, y, z) = \exp(2xy) - 4xz^2 + 13x - 7, \tag{2.2}$$

and $x, y, z \in \Re^n$. Figure 2.1 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]



(a)



(b)

Figure 2.1: Computational time for calculating the value and Jacobian of f in Equation (2.2) as a function of length of the input vectors.

In Figure 2.1a 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 2.1. 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 2.1b. Here, *ForwardDiff* is left out since it becomes too slow, but I have added a new implementation from

[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.)*.

MRST, that I will call *MRST new*. The MRST implementation in Figure 2.1a is now referred to as *MRST old*. We can observe that the trend seen in Figure 2.1a where *MRST new* is faster than *ForwardAutoDiff* for vectors longer than 10 000 continues for even longer vectors. As we can see from Figure 2.1b *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 (2.2). *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 (2.2), but if we for example want to calculate something like

$$g(x) = \frac{x[2:\text{end}] - x[1:\text{end}-1]}{\texttt{sum}(x)}, \tag{2.3}$$

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 (2.2) 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 2.2, where the graphs with circles as markers are the same methods as in Figure 2.1a using the function `benchmarkAD`. The graphs with squares are the same methods, only they are tested with the implementation in function `benchmarkADinLoop`.

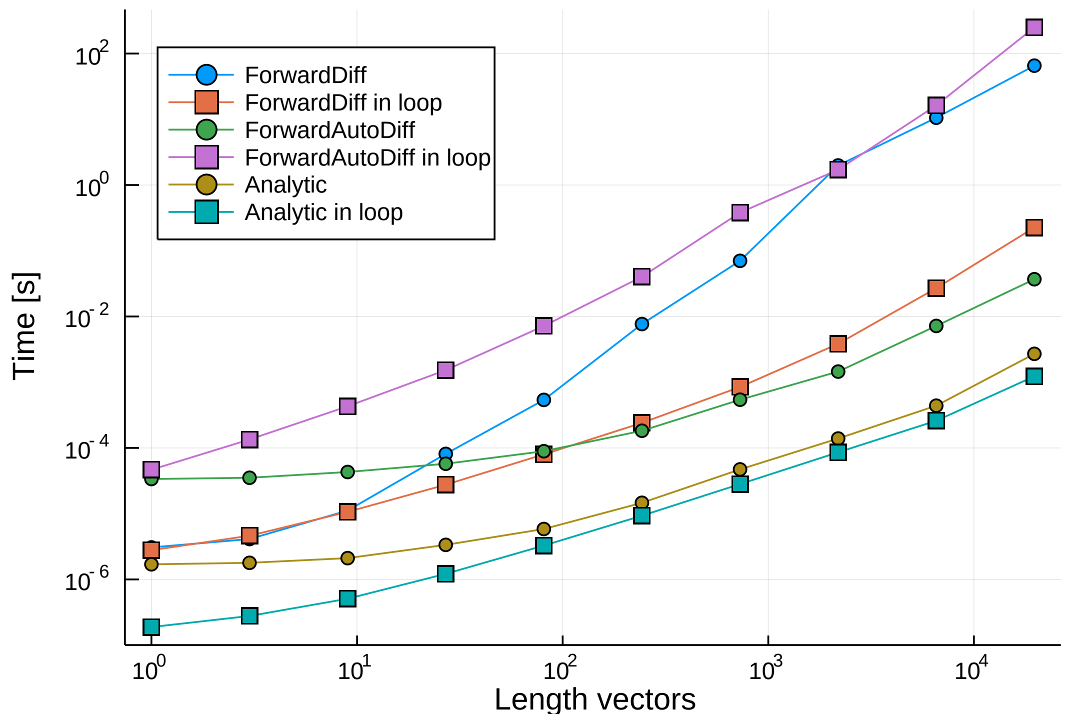


Figure 2.2: Computational time for calculating the value and gradient of f in Equation (2.2) as a function of length of the input vectors.

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 (2.2), 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 (2.3), 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.

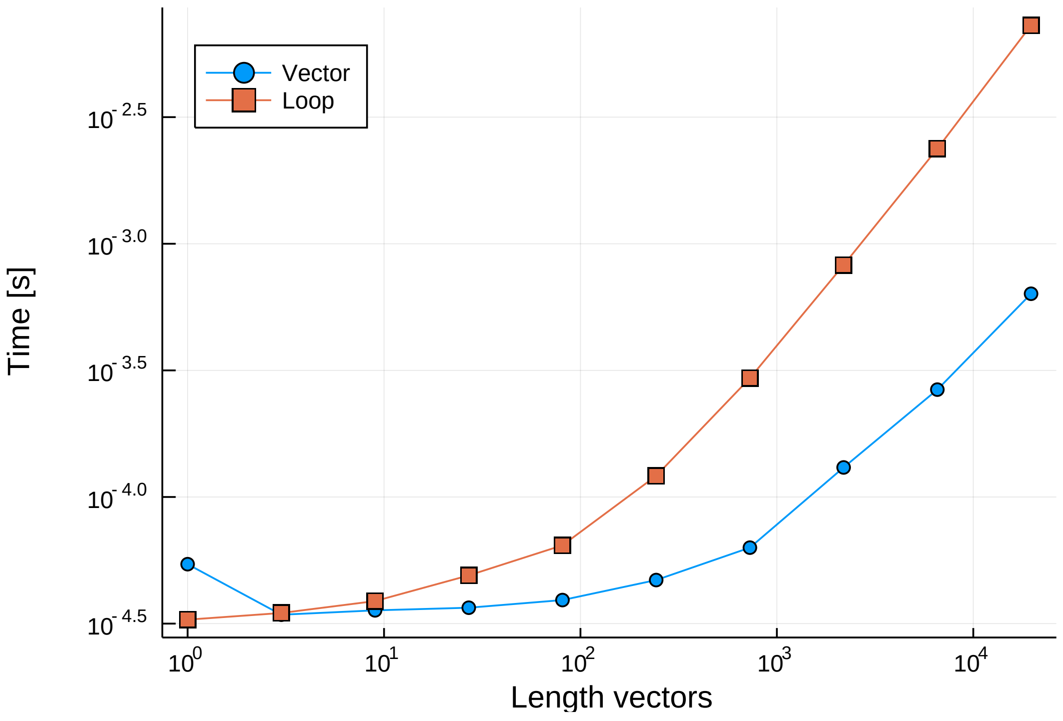


Figure 2.3: Computational time for evaluating the analytic functions f, f_x, f_y and f_z from (2.2) as a function of length of the input vectors in MATLAB.

Figure 2.3 shows how much time MATLAB uses to evaluate the analytic functions f, f_x, f_y and f_z from Equation (2.2) as vector multiplication and in a for-loop. The analytic graphs in Figure 2.2 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 3

Drafts

3.1 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.

3.2 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 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.

3.3 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