

SPECIALIZATION PROJECT IN MATHEMATICAL SCIENCES

Automatic Differentiation in Julia with Applications to Numerical Solution of Elliptic PDEs

Author:
SINDRE GRØSTAD

Supervisor:
Professor KNUT-ANDREAS LIE



NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEPARTMENT OF MATHEMATICAL SCIENCES

April 28, 2019

Table of Contents

Table of Contents	i
1 Julia	1
1.1 Characteristics of Programming languages	1
1.1.1 Type Checking	1
1.1.2 Compiled and Interpreted Languages	2
1.1.3 Languages for Numerical applications	2
1.2 History of Julia	3
1.3 Metaprogramming in Julia	3
1.3.1 Profiling	4
1.3.2 Debugger	5
1.3.3 Benchmarking	6
2 Theory	7
2.1 Automatic Differentiation	7
2.1.1 Forward Automatic Differentiation	8
2.1.2 Dual Numbers	9
2.1.3 Backward Automatic Differentiation	10
2.1.4 Forward Automatic Differentiation With Multiple Parameters	12
2.1.5 Forward Automatic Differentiation With Vector Functions	13
2.2 Applications of Automatic Differentiation	14
2.2.1 The Newton-Raphson Method	14
2.2.2 Solving the Poisson Equation	15
2.2.3 Discrete Differentiation Operators	17
3 Implementation	19
3.1 Automatic Differentiation in Julia	19
3.2 Implementation of Automatic Differentiation	20
3.2.1 ForwardAutoDiff (FAD)	20
3.2.2 Element-wise and Vector Multiplication	22
3.2.3 Optimizing ForwardAutoDiff	23
3.2.4 Custom Jacobian Automatic Differentiation	24
3.2.5 Efficient Versus Readable and Elegant Code	28
Vectorized and Devectorized Code	29
3.3 Benchmarking Automatic Differentiation	31
4 Flow Solver With Automatic Differentiation	35
4.1 Grid Construction	35

4.2	Setup of Governing Equations	37
4.3	Flow Solver Results	40
5	Local Automatic Differentiation	45
5.1	Implementation	45
5.2	Optimizing Local AD	49
5.2.1	Dynamic VS Static Arrays	49
5.3	Flow Solver with Local AD	50
6	Drafts	53
6.1	Shared Libraries vs Static Libraries	53
6.2	Vectorization vs Non-vectorization	53
	Bibliography	54

Chapter 1

Julia

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

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

1.1 Characteristics of Programming languages

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

1.1.1 Type Checking

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

language each variable needs to have a defined type beforehand and they are checked before the program is executed. We say that type checking happens before run-time. In dynamic languages the types of the variables does not need to be known before run-time and type correctness is checked at run-time, or in other words, continuously as the code is executed. Static languages are faster in the execution since it does not need to check for types as it is already done. You will, however, have to wait for the type checking to finish before the program can be executed. Dynamic languages are slower but the continuous type checking enables language designs that optimizes the coding process such that you can implement your program with less code. The first and most obvious difference from static languages is that you do not have to define what types each variable is. The type checking will find this out on its own at run-time. In addition it opens up for a feature called metaprogramming. Metaprogramming is programs that can take in other code as input and modify it such that implementing your code can be done more efficiently and faster. More specifically how this can be used will be discussed closer in section 1.3.

1.1.2 Compiled and Interpreted Languages

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

1.1.3 Languages for Numerical applications

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

1.2 History of Julia

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

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

Since Julia is an open source and free program to use, one of its strengths is that developers can contribute by creating useful programs that they share with all the other users. One example of this is the Integrated Development Environment (IDE), Juno (n.d.). Juno is a program to help writing Julia code easier and is created by mainly two developers, Sebastian Pfizner and Mike J. Innes. Juno gives the coder an environment where it is easy to run your Julia code, you get auto completion when writing your code, it has built in plotting panes to visualize results, and much more. The program is open source and free to use, and if you look at Juno's Github page (n.d.) there are many other contributors to the IDE than only Pfizner and Innes. These are developers that have either found an error in the existing code or made a feature they wanted for the IDE. This shows the strength of the code being open source – the community can contribute to the code database. This will help finishing improvements and error corrections to the code database quicker. This is why v1.0 of Julia was such a milestone, because as soon as Julia has guaranteed backward compatibility, there is a lot easier for developers to justify spending time to develop programs for Julia. This is because they now know the work they put in, will not be in vain, as the code will work on all future releases.

1.3 Metaprogramming in Julia

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

1.3.1 Profiling

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

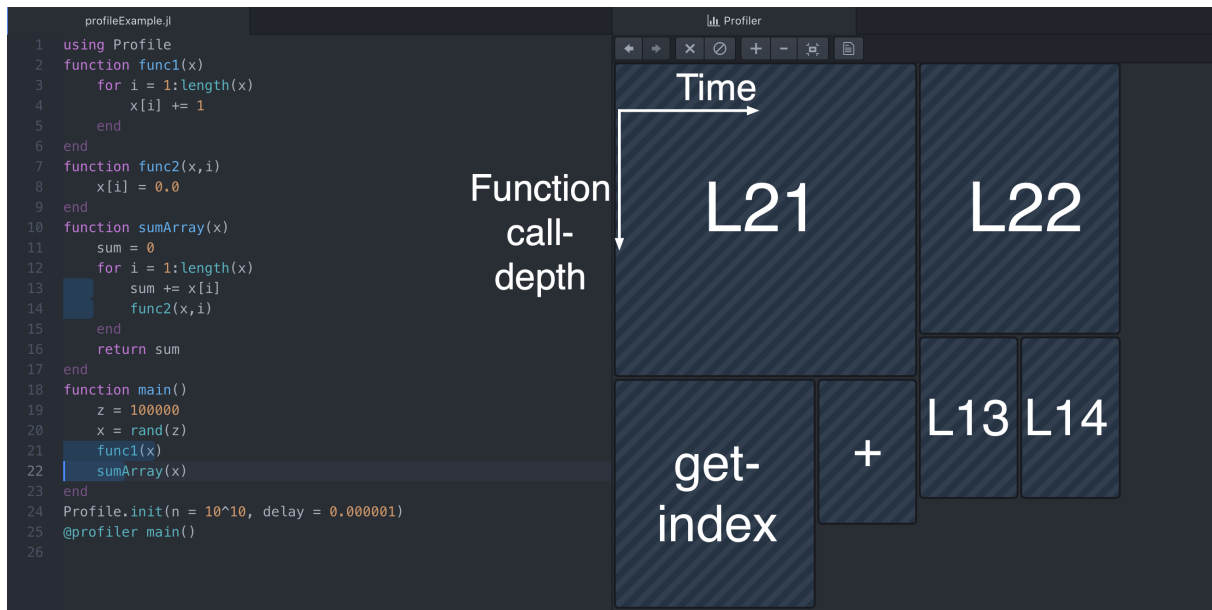


Figure 1.1: Interface in Juno of the `@profiler` macro tool for a simple test case. The left hand side shows the code profiled and the right hand side shows the result visualized. Since the visualization is interactive I have added extra text to show which block represents which line in the code.

Figure 1.1 shows the interface from the Juno IDE using the `@profiler` macro function (*Juno profiler, n.d.*) for a simple test case. The right hand side of Figure 1.1 shows a test case with a `main()` function that creates a random variable `x`. It then calls the `func1(x)` function that adds 1.0 to each element of `x`. Lastly it calls `sumArray(x)` that sum up all the values of `x` and uses `func2()` to set all values of `x` to zero. `@profiler` is a specific Juno macro that uses the built in `@profile` in Julia (*Profile.jl, n.d.*) to profile the `main()` function, but in addition it creates the right hand side of Figure 1.1 which shows the profiling result. Since the Profiler pane is interactive, I have added extra white text for visualization. Each block represents one specific operation inside the code and the wider the block is, the more time is spent in that operation. The vertical axis represents how deep into the stack trace the operation is. For Figure 1.1 line 21 (L21) and 22 (L22) are at top of the stack trace. We can also see that we spend more time in `func1(x)` than `sumArray(x)`. The time spent on each line can also be seen by the blue bars on each line. If we say L21 and L22 is at depth 1, then line 13 and 14 is at depth 2. The `getindex` and `+` blocks are built in functions in Julia that are called from line 4 in `func1(x)`. These functions are deeper than line 13 and 14 at depth 3. Technically it should have been an extra block underneath line 21 that represents line 4 to get the full stack trace. The only reason I have found why this is not part of the visualization is that the `@profiler` tool is still work in progress, and that it does not work perfectly just yet. However, this tool is super efficient to find bottlenecks in your code,

especially when you have a larger program than the simple example in Figure 1.1.

1.3.2 Debugger

Many IDEs for different languages have the ability to debug code. This is a tool to step into your functions and execute them step by step to reveal error and bugs in your code. Recently on March 19th Holy et al. (2019) published a debugger for Julia. This can be used directly in the terminal using the `@enter` macro function or as a built-in debugger in Juno with `Juno.@enter`. Figure 1.2 shows the interface of Juno's debugger for a simple test function. With this you have the opportunity to:

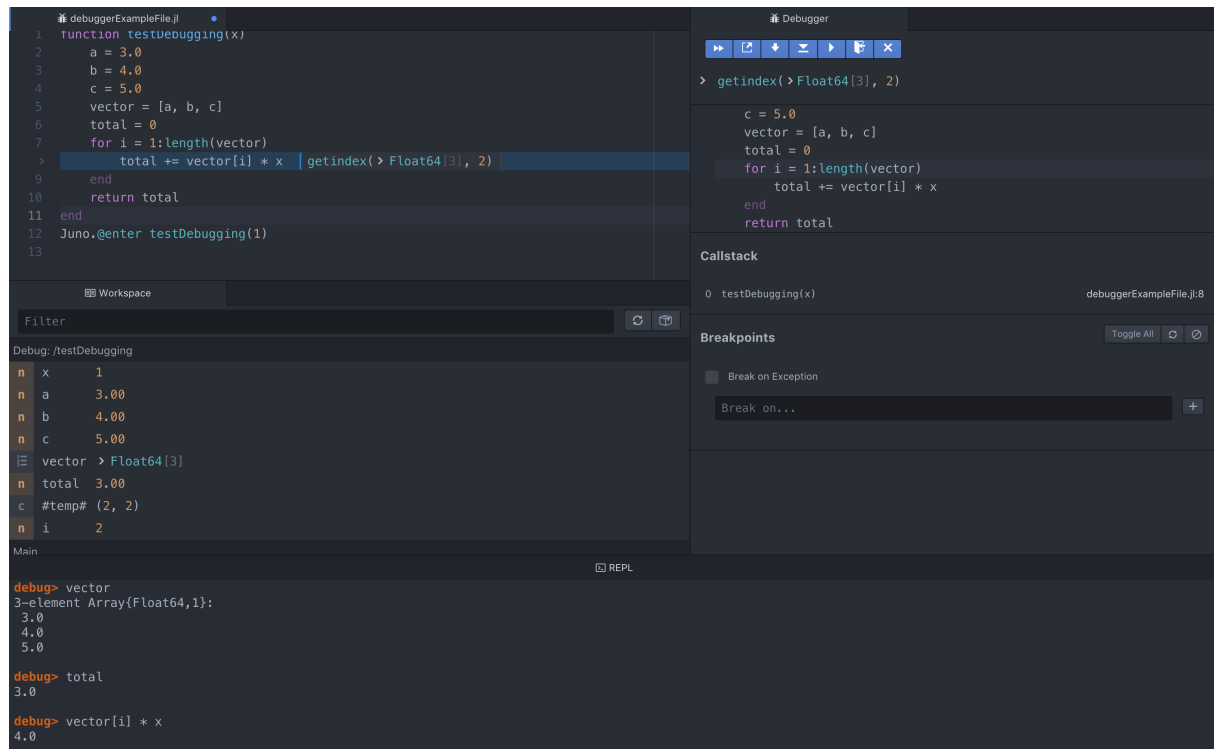


Figure 1.2: Juno's interface in debugging mode.

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

This is a great example of how powerful tools that help developers code are being created less than a year after v1.0 of Julia was released.

1.3.3 Benchmarking

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

An option to reduce possible sources of errors is to benchmark the time multiple times and use the average or mean time. However, then you have to write this extra code every time you want to benchmark a function. *BenchmarkTools (n.d.)* is a library that gives you a macro function called `@btime` that does exactly this. It uses `@time` multiple times to get a result that is less prone to sources of errors. The primary macro function from Benchmarktools is however `@benchmark` which returns minimum, maximum, mean and average time, together with memory allocation and usage and information of number of samples it has taken. All this is easy to use because of Julia's metaprogramming and makes it quick to benchmark all types of simulations and codes safely.

Theory

2.1 Automatic Differentiation

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

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

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

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

The computer will then use these rules on symbolic variables to obtain the derivative of any given function. This will give perfectly accurate derivatives, but 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, with finite differences you have the problem that you need a sufficiently small step size h to reduce the truncation error, but h can not be too small, because then you get round-off errors. Hence, the finite-difference method is not fully robust 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, the methods keep track of the function values and the corresponding values of the derivatives as we go. Both methods do this by separating each expression into a finite set of elementary operations.

2.1.1 Forward Automatic Differentiation

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

As an example, consider the scalar function $f = f(x)$ with its derivative f_x , where x is a scalar variable. If we evaluate this function for the AD-variable x , represented as the pair $[x, 1]$, the result is $[f, f_x]$. 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 four elementary arithmetic operators for our tuples, such that for functions f and g ,

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

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

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

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

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

When these arithmetic operators and the elementary functions are implemented, you are able to evaluate the derivative of any scalar function without actually doing any form of differentiation yourself. Let us look at a step by step example, where

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

The declaration of the AD-variable gives $x = [2, 1]$. All scalars can be viewed as AD variables with

derivative equal to 0, such that

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

After this computation, we get from the exponential

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

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

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

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

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

2.1.2 Dual Numbers

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

$$a + b\epsilon. \quad (2.4)$$

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

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

and likewise for multiplication

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

and for division

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

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

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

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

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

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

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

2.1.3 Backward Automatic Differentiation

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

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

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

$$x, \quad g_1 = 2x, \quad g_2 = \exp(g_1), \quad g_3 = x \cdot g_2, \quad (2.6)$$

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

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

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

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

This 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

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

for some s to be chosen.

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

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

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

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

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

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

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

substituting s with g_5 yields

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

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

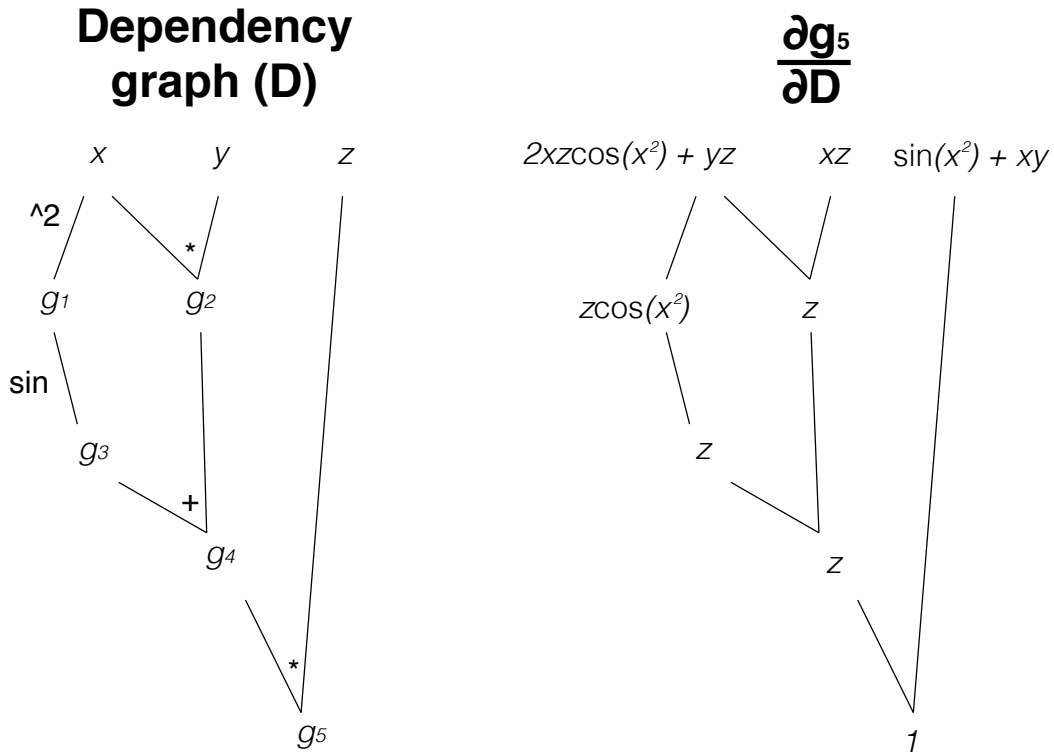


Figure 2.1: Graphs to visualize the process of backward AD. To the left is a dependency graph of the elementary functions in (2.8) and to the right are the derivatives of g_5 with respect to the dependencies given in the dependency graph.

Comparing this to the method of dual numbers from subsection 2.1.2, where we would have to evaluate f three times, once 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 function and derivative values as we did in forward AD, we need to implement the dependency tree shown in Figure 2.1. This makes the implementation of backward AD much harder than for forward AD, and an inefficient implementation of this tree will reduce the advantage of backward AD. Also, if f is a vector-evaluated function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ and not a scalar function, backward AD needs to run m times. Hence, if $n \approx m$, forward AD and backward AD will have approximately the same complexity. This is the main reason why we in the following will focus on implementing forward rather than backward AD.

2.1.4 Forward Automatic Differentiation With Multiple Parameters

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

AD-variables

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

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

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

This result is equal to the tuple

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

for the corresponding x , y and z values.

2.1.5 Forward Automatic Differentiation With Vector Functions

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

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

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

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

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

the values on the diagonal. The calculations give

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

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

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

2.2 Applications of Automatic Differentiation

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

2.2.1 The Newton-Raphson Method

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

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

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

$$\mathbf{Ax} = \mathbf{b}, \tag{2.10}$$

which we can write on residual form such that

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

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

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

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

2.2.2 Solving the Poisson Equation

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

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

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

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

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

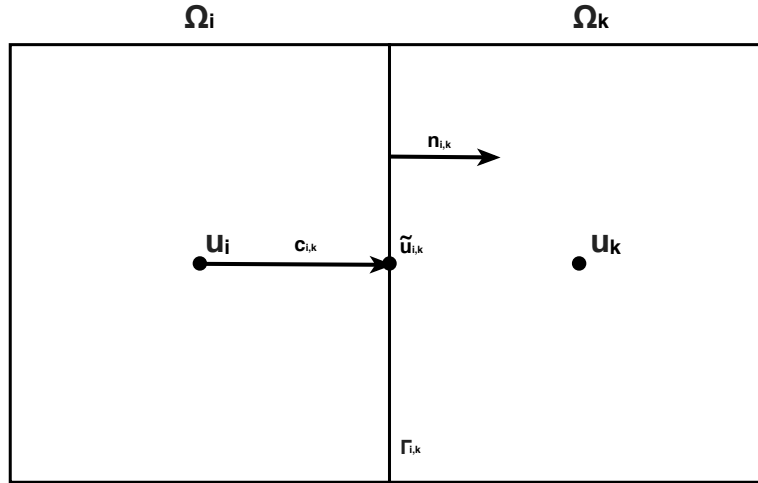


Figure 2.2: Figure of two adjacent cells Ω_i and Ω_k . The average value of the cell is given by u_i . The boundary between the cells is $\Gamma_{i,k}$ with value at the centre equal $\tilde{u}_{i,k}$ and outward normal vector $n_{i,k}$

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

$$v_{i,k} = \int_{\Gamma_{i,k}} \mathbf{v} \cdot \mathbf{n}_{i,k} \, ds. \quad (2.14)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

where q_i is the integrated accumulation over cell Ω_i . Now, we can get a linear system of the form $\mathbf{A}\mathbf{u} = \mathbf{b}$, which on residual form becomes $\mathbf{F}(\mathbf{u}) = \mathbf{A}\mathbf{u} - \mathbf{b} = 0$ 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 (2.12), using the scheme explained in (2.11) and by having u as an AD-variable. For this simple Poisson equation, we still only end up with a linear system of equations that we may as well solve without AD. The only benefit is that we never need to form the matrix \mathbf{A} explicitly, which can be bit complicated for more complex grids. For nonlinear PDEs, we would also need to linearize the local discrete equations, and the construction of the matrix \mathbf{A} generally becomes more tricky. The ease of using AD becomes clearer when we combine it with discrete differentiation operators that we will define in the next subsection.

2.2.3 Discrete Differentiation Operators

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

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

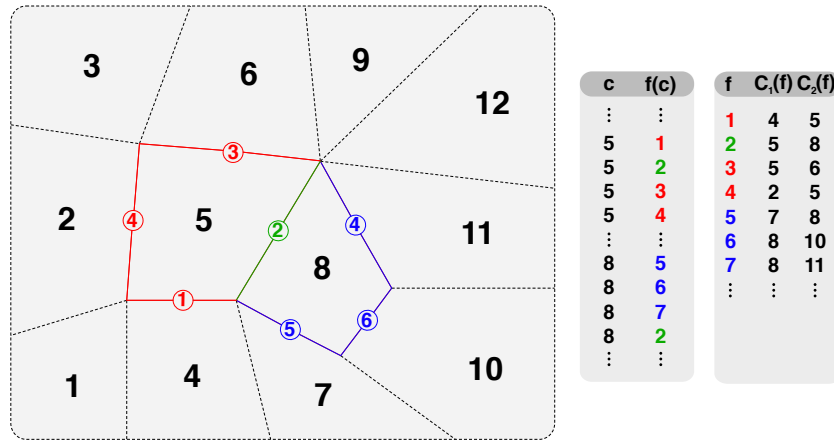


Figure 2.3: Figure of a general polygonal grid with the mapping cell to facets and facet to cells.

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

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

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

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

flux leaving and entering the cell, the discrete divergence operator for cell c is defined as

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

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

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

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

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

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

Here, \mathbf{T} is the transmissibility defined in (2.15). The notation for the discrete equations is clearly similar to the continuous case, and we can actually read the discrete expression and directly understand what equation we are trying to solve. For this simple Poisson equation, we will still have a linear system and we would not necessarily need to use AD to solve it. But for more complex problems, we can derive the discrete divergence and gradient operators in the same approach for any type of grid. Although the system then becomes non-linear, it will be easy to solve using AD and the Newton-Raphson method. An example of this can be seen in chapter 4.

Implementation

intro her må endres da deler av Julia-forklaringen er flyttet utenfor.

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

3.1 Automatic Differentiation in Julia

In short, Julia 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 machine learning, without going too deep into the subject, largely amounts to the minimization of functions with a large number of input parameters, but with only one output parameter. As discussed in subsection 2.1.3, backward AD is much more efficient than forward AD in these types of evaluations. For numerical applications there is usually many input parameters and output parameters. Hence there is no clear advantage of using backward AD compared to forward AD.

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

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

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

- The function to differentiate can only accept a single argument. This is possible to work around; if you have vector function f with input parameters $x, y, z \in \mathbb{R}^n$, you can merge them into one vector of length $3n$ and then obtain the Jacobian. Although this works and you get the correct answer, it is not optimal as you would have to make local workarounds to make the code work, causing unreadable code.
- The function we want to differentiate must be on the form of a generic Julia function, such as: $g(x) = 3x * x$. Here $x * x$ symbolize element-wise multiplication. This means that if we have a function like $h(x) = 3x * x + \text{sum}(x)$, where all elements in $g(x)$ are added with the sum of all elements in x , it will not be possible to use *ForwardDiff* to obtain the Jacobian. This limitation also prevents us from having a function that we evaluate for all points, and then add boundary conditions for some points later. This is a feature that is essential in PDE-based simulators.
- The Jacobian calculated by *ForwardDiff* is a full matrix. In some cases the Jacobian is dense anyway, so this will not have any major adverse effects, but in many numerical applications, and particularly in numerical solution of PDEs, the Jacobian will be sparse. By representing a sparse matrix on a full matrix format, a lot of potential computation efficiency is lost.

3.2 Implementation of Automatic Differentiation

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

```
struct AD
    val
    jac
end
```

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

3.2.1 ForwardAutoDiff (FAD)

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


```

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

```

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

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

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

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

```

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

```

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

```

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

```

Here, the first function will be used if the + operator is used with an FAD variable on the left hand side and a number on the right. The last line is a compact way of writing the opposite function, which

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

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 chapter 4, we want to sum over all the elements in the vector. If we look at how we can overload the `sum` function, one might think that we would try something like

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

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

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

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

3.2.2 Element-wise and Vector Multiplication

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

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

resolved. With this in mind, and that there will only be used element-wise multiplication in this project, I have decided that I herein redefine `*` and implement it as element-wise multiplication. This means that if I have written regular multiplication expressions consisting of at least one AD-variable, element-wise multiplication will be executed.

3.2.3 Optimizing ForwardAutoDiff

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

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

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

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

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

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

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

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

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

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

This way of storing a matrix will decrease both memory usage and computational efficiency dramatically 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, which we with a sparse matrix structure can 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 2.1.5, where we wish to compute $f = 2 \cdot \mathbf{x} \cdot \mathbf{y}$ for the primary variables $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$ defined in (2.9). Initializing the primary variables as FAD-structs gives

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

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

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

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

$$\begin{pmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}. \quad (3.1)$$

We immediately observe that two of the multiplications are unnecessary, since one of the matrices is a 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 (3.1), 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.

3.2.4 Custom Jacobian Automatic Differentiation

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

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

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

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

Now, the multiple dispatch system in Julia comes in handy once again. Since Julia understands at run-time what type of `CustomJac` we have – whether it is a `NullJac`, `IdentityJac`, `DiagJac` or `SparseJac` – we can implement different methods for all possible combinations. In each implementation, we now know what type of matrix we are dealing with, and we can thus optimize the performance. As an example, consider calculating the Jacobian w.r.t primary variable `y` in (3.1) (All the implementations of operators on structs extending the `CustomJac` type are called from outer functions that check the legality of the operations; hence, the following code excerpts contain no safety checks.) First, we have a vector multiplied element-wise by a null matrix. The implementation

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

knows immediately that there is no need to do any calculations, the result will be a null matrix of the same size as before. Without diving too deep into the implementation of `SparseMatrixCSC`, but only by considering what kind of information the `SparseMatrixCSC`-struct contains, this implementation should not make a big difference in the computational efficiency of CJAD compared to FAD. The reason for this is that the implementation of `SparseMatrixCSC` will also quickly realize that its vectors are of length zero and the matrix is a zero matrix. The speed difference of the two implementations will hence be small for this particular operator. However, the second calculation in (3.1) is a vector multiplied element-wise by an identity matrix. The result will be a diagonal matrix with the values of the vector on the diagonal. The `SparseMatrixCSC` implementation only knows that we have a sparse matrix with some values and is unaware that it actually is the identity matrix. This means that it has to do all the calculations without any shortcuts. The implementation for `IdentityJac`, however, automatically knows the result of this calculation and simply makes a diagonal Jacobian with the vector on the left hand side:

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

This is the first calculation we have seen that will make CJAD considerably more computational efficient than FAD. To finish the calculations in (3.1), we finally have to add a null matrix and a diagonal matrix. Since there is really no need of doing this adding, the implementation simply returns the diagonal matrix:

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

For the same reason as explained for the element-wise multiplication between a vector and a `NullJac` matrix, this implementation will only have small advantages compared to the

`SparseMatrixCSC` implementation used in FAD, which will also quickly figure out the needlessness of the calculation.

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

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

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

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

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

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

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

```
end
```

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

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

I will call this the *dot-multiplication* method. As it is difficult to say without first-hand knowledge of the implementation of `SparseMatrixCSC` which method is best to implement, I will test the two methods against each other. To this end, I have created a random vector of length 40 and a random sparse matrix of size 40×40 having 20 percent nonzero elements. I then perform the element-wise multiplication 10 000 times to separate the efficiency of the two methods. To check which method is actually more efficient, and by how much, we can use the `@btime` macro function from the *BenchmarkTools* (*n.d.*) library. This function gives us the number of allocations, the amount of memory used, and the time spent. Table 3.1 reports the results¹ of this test.

Klarte ikke å
grunnen til d
ved å se i kilo
Kanskje noen
SINTEF vet h

Table 3.1: The number of allocations, amount of memory, and time spent for element-wise multiplication of a vector of length 40 and a sparse matrix of size 40×40 with 20 percent nonzero elements. The vector and matrix are multiplied together 10 000 times.

Method	Number of Allocations	Megabytes	Milliseconds
<i>Matrix-multiplication</i>	1 420 000	194	187
<i>Dot-multiplication</i>	60 000	264	87

The test is unequivocal: the *dot-multiplication* method has far less allocations, is more than twice as fast, but it uses a bit more memory. Based on this it might tempting to say that the *dot-multiplication* method is a better choice. However, in numerical solution of PDEs, the matrices we work with are generally be much larger and very sparse. Table 3.2 reports results for a more representative test example, in which I have used a matrix from chapter 4, which is of size 8000×8000 and only contains 0.08 percent nonzero elements.

Table 3.2: The number of allocations, amount of memory, and time spent for element-wise multiplication of a vector of length 8000 and a matrix of size 8000×8000 with 0.08 percent nonzero elements. The matrix is taken from chapter 4.

Method	Number of Allocations	Megabytes	Milliseconds
<i>Matrix-multiplication</i>	240 370	32	28
<i>Dot-multiplication</i>	180	10 243	2 458

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

¹All 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.

3.2.5 Efficient Versus Readable and Elegant Code

There is usually a fine balance between writing efficient and readable/elegant code. Sometimes you have the opportunity to do both, but often you have to choose which of them you want to give most focus. At the end of subsection 3.2.1, I explained how we can elegantly implement the `sum` function, and all other built-in functions that iterate through the FAD-variables, only by implementing the `iterate` function. This is a very elegant use of Julia's multiple dispatch system, as we get a lot of functionality "for free". The downside, however, is that we loose potential computational efficiency that we could achieve by implementing the `sum` function ourselves. Let us define a random function ψ , where the AD-representation and the `sum` of ψ are given by

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

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

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

Instead of retrieving each row, one by one, like we did using `Base.sum`, we now add up each column in one go. Here, `sum(A.val)` is the built-in `sum` function in `Base` for summation of a vector, and `colSum` is a function implemented in each extended type of `CustomJac`. This function sums up all the columns in the Jacobians and returns a $1 \times n$ matrix. Here, n is a number that depends on the dimension of the relevant primary variable ($n_\psi = 3$). In the three special cases, the function simply returns a $1 \times n$ null matrix for `NullJac`, a $1 \times n$ matrix containing only ones for `IdentityJac`, and a $1 \times n$ matrix, which is the transposed of the diagonal vector, for `DiagJac`. For `SparseJac`, it returns a $1 \times n$ matrix in which all the nonzero values in each column are added together. To check that this new `sum` function actually is more efficient, we can again use the `@btime` macro function from the *BenchmarkTools* (n.d.) library. By creating a random `CJAD`-variable of length 400 with all of the four different Jacobian types (all 400×400 with random values), we can compare how efficient the two methods sum up the `CJAD`-variable. Table 3.3 reports the result and shows that even though only implementing the `iterate` function is an elegant solution, it is much more efficient both in terms of memory consumption and runtime to implement the `sum` function specifically.

Table 3.3: Table with the number of allocations, memory usage, and runtime spent by summing a random CJAD-variable of length 400 with two different summation methods.

Function	Number of Allocations	Kilobytes	Microseconds
sum	40	79	81
Base.sum	28371	19 597	58 051

Vectorized and Devectorized Code

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

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

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

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

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

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

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

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

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

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

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

To separate the three methods, I have calculated $h(x, y)$ 1000 times for random x and y vectors of length 100 000 and evaluated the performance using `@btime`. Table 3.4 reports the results.

Table 3.4: Table with the number of allocations, memory usage, and runtime spent by evaluating $h(x, y) = \exp(x) \cdot \log(y)$ 1 000 times for three different methods; x and y are random vectors of length 100 000.

Method	Number of Allocations	Megabytes	Seconds
devectorized	2000	789	2.104
dotsInside	6000	2 405	2.704
dotsOutside	2000	789	2.121

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

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

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

So what are the consequences for the implementation of CJAD? The result from Table 3.4 implies that as long as we write all our vectorized code like the `dotsOutside` method or in such a manner that it is obvious for the compiler that it is only vectorized operations involved, we will have the same computational efficiency as for `devectorized` code. More specifically for the CJAD implementation, we often have a vector that is added/subtracted/divided/multiplied with another vector, and then the vectorized implementation will have the same computational efficiency as the `devectorized`. Since `devectorized` code is less readable, implementations reported in the following will consist of vectorized code for readability.

3.3 Benchmarking Automatic Differentiation

As mentioned in section 3.1, Julia already has an AD library called `ForwardDiff` (Revels et al., 2016) that uses forward AD. Hence, it would be interesting to see how the different implementations compare as the functions evaluated increase in complexity. In addition to `ForwardDiff`, I have added two AD implementation from the MATLAB Reservoir Simulation Toolbox (MRST) (MRST Homepage, n.d.) to the benchmark, to see how the Julia implementations compare to optimized AD tools in MATLAB. The basic implementation in MRST is similar to `FAD`, where the Jacobians has a sparse matrix structure, and is henceforth referred to as `MRST`. The second MATLAB implementation is similar to CJAD in the sense that it exploits the diagonal structure of some Jacobians. This method is called `MRST_diagonal`. To benchmark the efficiency of the different AD tools, I have evaluated the vector function $f : \mathbb{R}^{n \times 3} \rightarrow \mathbb{R}^n$, where

$$f(x, y, z) = \exp(2xy) - 4xz^2 + 13x - 7, \quad x, y, z \in \mathbb{R}^n. \quad (3.2)$$

Figure 3.1 reports how computational time² for calculating the function value and the Jacobian of the function scales as the length of the vectors n increases for the different methods.

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

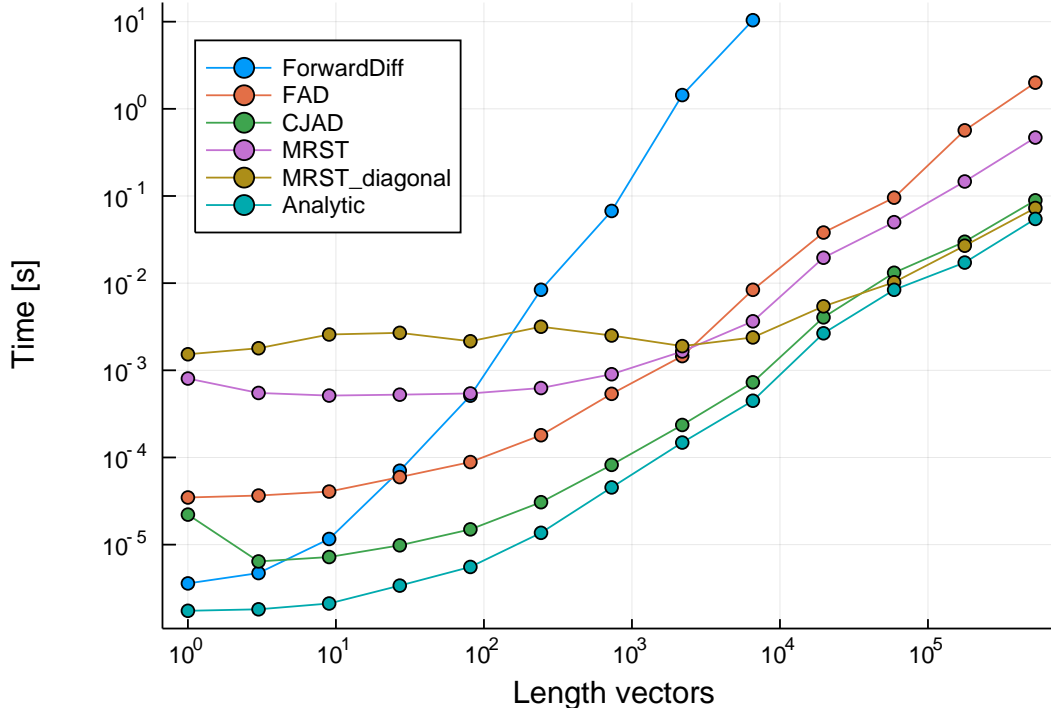


Figure 3.1: Computational time for different methods calculating the value and Jacobian of f in Equation (3.2) as a function of length of the input vectors.

Figure 3.1 reports six different graphs. The first thing you observe is that `ForwardDiff` scales very badly as n becomes large. This is because it creates and works with the full $3n \times 3n$ Jacobian matrix as discussed in section 3.1. I stopped the benchmark of `ForwardDiff` for $n = 3^8$, as the Jacobian at this point already has more than 380 million elements. We can nevertheless observe that for small vectors, the methods implemented in MATLAB, as well as `FAD` and `CJAD`, have more overhead than `ForwardDiff` and the analytic solution. This makes them slower for small n , but as n grows, this overhead becomes quickly negligible. Based on Figure 3.1, from a numerical simulation point of view, `ForwardDiff` is useless for obtaining the Jacobian of functions, as it scales badly for increasing n .

Apart from the conclusion on `ForwardDiff`, we can observe that it is a change in dominance at $n = 3^7 = 2187$. For shorter vectors, both methods implemented in Julia are more efficient than the two methods in MATLAB. For $n > 2187$ `FAD` is the slowest, followed by `MRST`. Hence, as n becomes larger than 2187, the methods in Julia and MATLAB that exploit the diagonal structure of the Jacobian perform better than the ones that only use a sparse matrix structure. This makes sense, as the evaluation of f will only give a diagonal Jacobian. As n becomes large, `CJAD` and `MRST_diagonal` perform very similarly and their computational costs approach that of the analytic evaluation. What is also interesting to see is that whereas `CJAD` always is faster than `FAD`, `MRST_diagonal` is less efficient than `MRST` for short vectors. Assuming that both the `MRST` implementations are optimized, this indicates that the elegant way of implementing a custom Jacobian in Julia using multiple dispatch adds very little overhead compared to MATLAB.

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

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

the diagonal structure of the Jacobians is gone, and `MRST_diagonal` transitions to use the `MRST` implementation, and `CJAD` has to use the `SparseJac` type, which is similar to the method used in `FAD`. This does not imply that the optimized methods are implemented in vain, since functions evaluated in numerical applications often have parts that will have diagonal Jacobians, and other parts that will not. We can thus expect to gain computational efficiency from parts of the function evaluations using `CJAD` and `MRST_diagonal`. To assess how much we can gain, we have to look at more realistic examples. I will do this in the following chapter 4.

Flow Solver With Automatic Differentiation

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

avsnittet over må endres pga julia-kapittelet.

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

4.1 Grid Construction

The example consist of modelling how the pressure drops within a rectangular reservoir measuring $200 \times 200 \times 50\text{m}^3$ when we have a well that that is producing oil. "Single-phase solver" only means that we do not have different phases of fluid, like liquid and gas, present during the simulation. As the purpose of this example is to compare the AD tools in MATLAB and Julia and not the process of solving the problem, including setting up the grid and other necessary variables, some of the initialization

and plotting have been performed in Julia by calling code from MRST. This has been done by using the package MATLAB.jl (*MATLAB.jl, n.d.*). This package allows calling MATLAB functions from Julia and retrieve the output variables. This is done by the function call

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

where we have three input parameter and two output parameters. We have to specify the number of output parameters after the MATLAB function name. By calling MATLAB from Julia, we can use MRST's `G = computeGeometry(...)` function to set up the grid for the simulation. The grid of the reservoir can be seen in Figure 4.2a. The variable `G` that contains the grid properties is now a structured array (struct), having all the information on cells, facets, vertices and nodes that we need to make the discrete divergence and gradient operators as explained in section 2.2.

Next, we define the properties of the rock. In an oil reservoir, the oil lies inside porous rock and hence the properties of the rock will affect the flow of the oil. The amount of oil we can have inside the rock is measured as pore volume. First, we make a new variable `rock` that contains parameters that describe the rock's ability to store and trasmit fluids for each cell with the function `rock = makeRock(...)`. Then, we say that in our model the rock is compressed constantly as a function of pressure and we obtain the analytic solution for the pore volume as a function of pressure

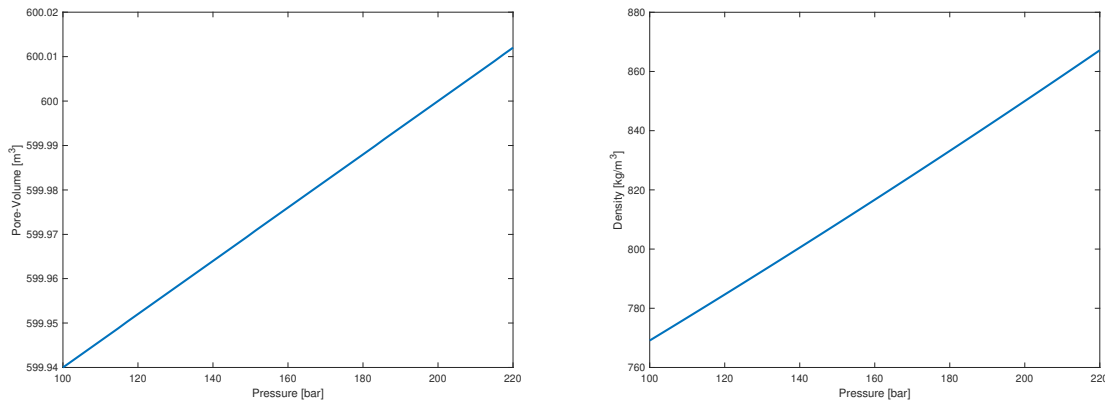
$$pv(p) = pv_r \exp[c_r \cdot (p - p_r)], \quad (4.1)$$

where pv_r is the rock pore volume properties, in a cell, at a reference pressure p_r . c_r is a constant controlling how the rock is compressed. The pore volume of the rock as a function of pressure values are visualized in Figure 4.1a. The graph shows that for a cell with volume 1000m^3 there is approximately room for 600m^3 oil.

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

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

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



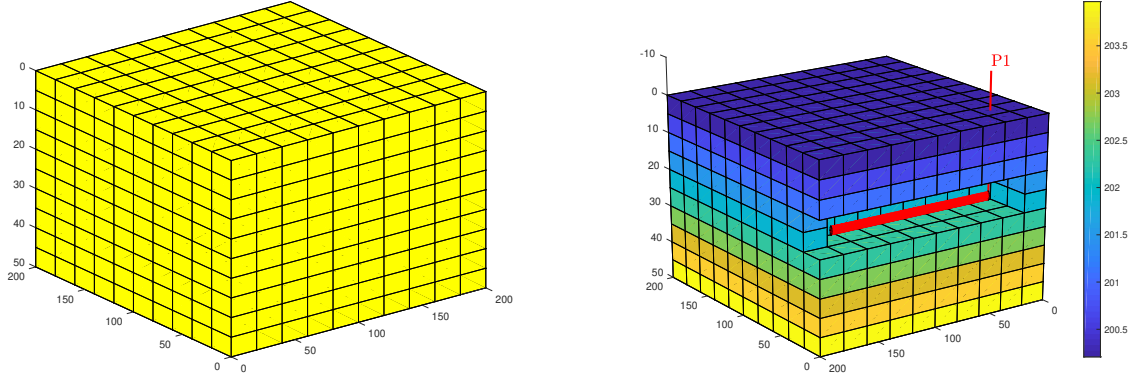
(a) The pore volume of the rock in a cell as a function of pressure. (b) The density of the oil in the reservoir plotted as a function of pressure.

Figure 4.1

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

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

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



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

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

Figure 4.2

4.2 Setup of Governing Equations

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

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

The matrix `C` in the discrete operators are created such that when it is multiplied by the pressure p , the result becomes the pressure difference between two adjacent cells, as defined in Equation (2.18). The matrix `C` is stored as a sparse matrix, and the reason why is clear from Figure 4.3, which shows the sparse structure of `C`. The divergence operator is made using the fact that in the continuous case, the gradient operator is the adjoint of the divergence operator

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

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

Now we have all the ingredients to set up the governing equations for the flow in the reservoir. We use a finite volume method to discretize in space, as explained in section 2.2, and a backward Euler method to discretize in time. In the end, all the equations we want to solve should be on residual form, $\mathbf{F}(\mathbf{x}) = 0$, so that we can use the Newton-Raphson method described in Equation (2.11) to solve the system. As there are multiple equations that will be a part of the residual function $\mathbf{F}(\mathbf{x})$, we define them separately first. One of the advantages of defining the discrete gradient- and divergence operator is that the continuous and discrete forms of the equations look very similar. Hence I will first state the continuous version of the equation and then the discrete, so that it is easy to see how similar they look. I start by defining Darcy's law, which explains how the oil will flow through the porous rock

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

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

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

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

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

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

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

where `pv` is the pore volume of the rock given in Equation (4.1) and $p0$ is the pressure at the previous time step.

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

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

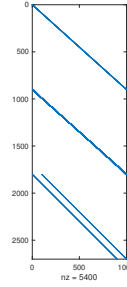


Figure 4.3: Structure of discrete operator C .

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

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

The pressure drop near the well usually takes place on a much shorter length scale than the size of a grid cell, and is usually modelled through a semi-analytical expression that relates flow rate to the difference between the reservoir and wellbore pressures. Hence the analytic expression for the production in a perforated cell is given by

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

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

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

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

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

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

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

When all the governing equations are defined, we merge them into one large residual vector function $\mathbf{F}(\mathbf{x})$. The first 1000 residual equations are the `presEq` with negative production `q_conn` for the indices `wc`. Equation number 1001 is the `rateEq` and Equation 1002 is the `ctrlEq`. Hence, $\mathbf{x} \in \mathbb{R}^{1002}$, where the first 1000 elements are the average pressure in each cell, element 1001 will be the pressure at the datum point inside the well (`bhp`), and element 1002 is the surface fluid rate `qS`. Now, if we start by defining `p`, `bhp` and `qS` as AD-variables, \mathbf{F} will also be an AD-variable and we will have the Jacobian of the residual vector function \mathbf{F} . This means we can solve the equations using the Newton-Raphson method, defined in Equation (2.11). A pseudo code of how we solve the system can be seen below.

```

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

```

4.3 Flow Solver Results

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

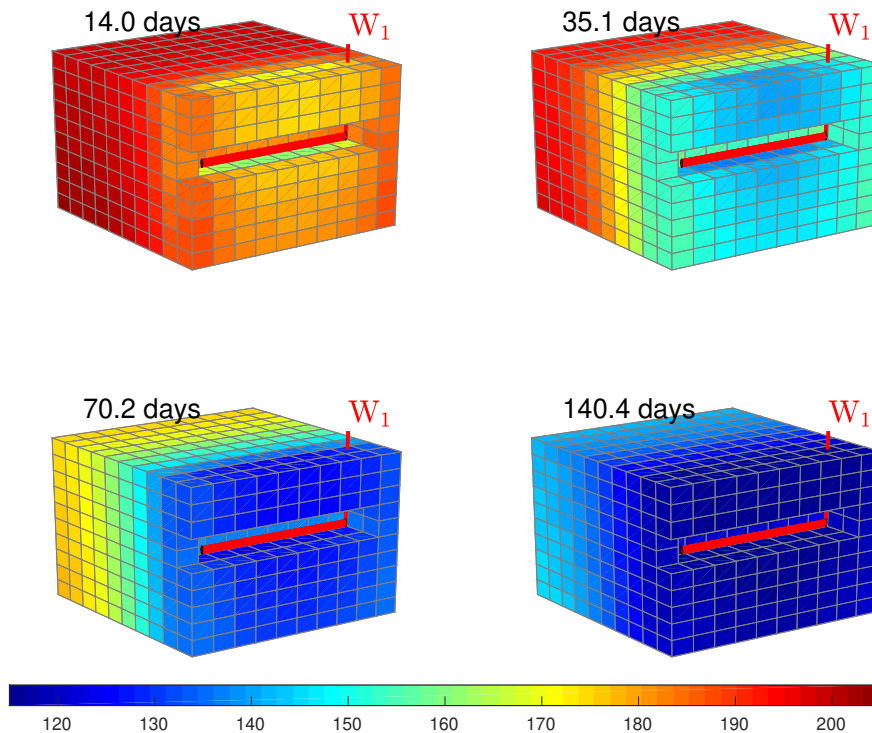


Figure 4.4: The pressure in the reservoir displayed at four different times. Pressure is given in bar.

Note the different intervals for the color bar in Figure 4.4 compared to Figure 4.2b. With the current color bar interval, at initial state, the whole reservoir will be displayed red. Hence, we can see how the reservoir from the beginning being approximately 200 bar everywhere, begins with the largest pressure decay close to the well, but after some time, the oil is pushed towards the well by the pressure differences and the pressure begins to decay also furthest away from the well.

Figure 4.5 shows the development of the production rate and the average pressure inside the reservoir. We can see how the production rate follows the average pressure inside the reservoir, and that after some time it approaches zero. This phase is what is called primary production. In primary production, the pressure in the reservoir is so high that there is no need to pump the oil out, the

pressure difference does all the work. After some time, we can see that the pressure becomes too low and the production decreases. When this happens, we transit to what is called secondary production. To retrieve more oil from the reservoir we need to apply extra pressure inside the reservoir. This can for example consist of injecting water or gas into the reservoir. This is called the secondary production. More details about how this work can be found in *Lie (2019)*, but the main idea here is that in order to keep up the production and fully exploit the resources in the reservoir, we need to apply external pressure. To do this in the best possible way, it is important to be able to simulate how the pressure evolves inside the reservoir so that we can make good decisions on which actions produce the best possible results. This flow solver is a simple example of how we can model such evolution of the pressure in a reservoir elegantly with governing equations on residual form, created by discrete differentiation operators, and AD.

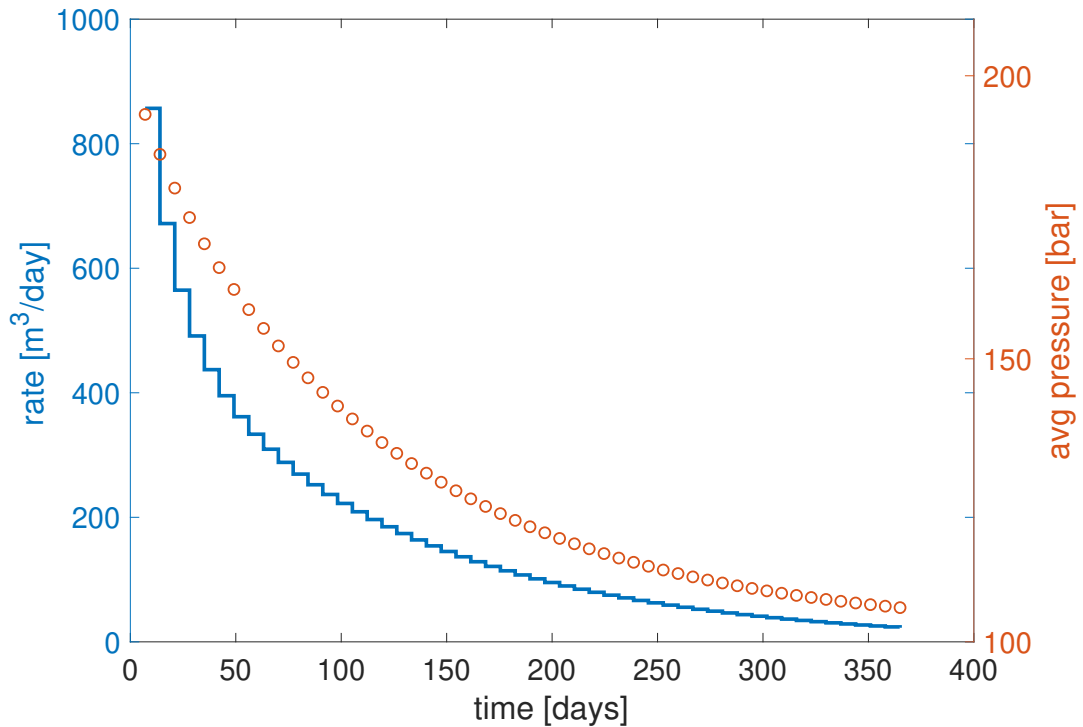


Figure 4.5: The production rate and the average pressure in the reservoir as a function of time.

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

As explained in section 3, the different types of AD mentioned store the Jacobians differently. Both FAD, CJAD and the two AD implementations in MRST, will, at least eventually, store the Jacobians as a list of sparse matrices where each Jacobian element in the list is the Jacobian with respect to one primary variable. In this example, this is a lot more efficient than storing the full 1002×1002 Jacobian as `ForwardDiff` does.

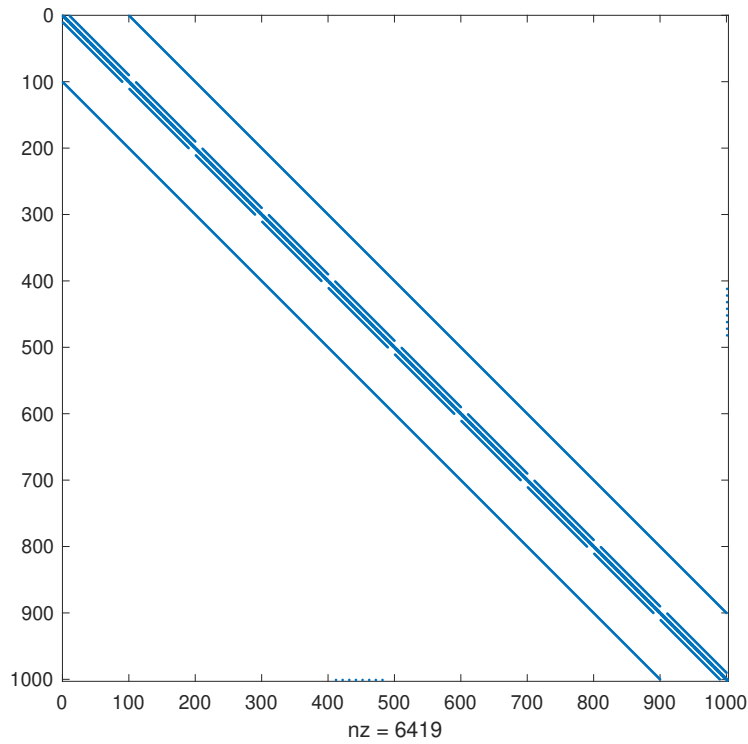


Figure 4.6: Structure of the 1002×1002 Jacobian of the governing equations. There are 6419 non-zero elements.

The difference in implementation of the Jacobian should make a big impact on the computation time for this problem. To benchmark the different methods, we need to do some extra work to make sure that it is actually the AD we benchmark and not other parts of the code. This is especially important for the code running in Julia, since when we call MATLAB from Julia it will be a lot of overhead. This means that the setup of the discrete gradient and divergence operators will take longer when done in MATLAB called from Julia, than when we do it directly in MATLAB. If we want to run the full simulation it is not possible to separate the AD part fully, but if we only benchmark the main-loop containing the Newton-Raphson method, AD will be a dominating part of the computations together with the linear solver `f.jac\f.val`. This means we at least will get an indication of how well the AD tools perform compared to each other. To see how the different methods scale as the discretization becomes finer and the system we solve grow, I have benchmarked three different discretizations. The first is the original setup with 10 cells in spatial direction x , y , and z . This gives a total number of 1000 cells in the reservoir. Then, I have also tested the implementations for 20 and 30 cells in the spatial directions. The time spent solving the system for the different methods can be seen in Table 4.1.

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

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

For 10 cells in each spatial direction it is clear that what I assumed based on the structure of the Jacobian in Figure 4.6 is true; The `ForwardDiff` method takes much longer time than `FAD`, `CJAD` and `MRST`. Since `ForwardDiff` already spends over a minute for $10 \times 10 \times 10$ cells, it is omitted for finer discretizations. `MRST_diagonal` is not presented here because there is a bug that cause the method to fail for this specific example. According to the results in Table 4.1, `MRST` is a bit faster than `FAD`, but `CJAD` and `MRST` are very similar.

Finne ut hva
er slik at det
kjørt?

It is, although, still some uncertainty in these numbers as we do not know if the linear solver in MATLAB and Julia perform equally well. In addition, another factor that can effect the result is that for the current hardware these benchmarks are running on, the computations are so demanding that the hardware becomes warm. This can cause the performance of the hardware to decrease and ruin the test results. To remove the uncertainty of both the linear solver and the hardware heating up, we can perform another benchmark. Instead of solving the linear system `f.jac\f.val` and continuing to the next Newton-Raphson iteration until it converges and then onto a new time step, we only assemble the first residual function multiple times. In this way we only benchmark the AD part of the problem. The test is still relevant as the structure of each function will be equal for all time steps, only that there will be other values. And since the time the computer uses to calculate the residual function value and Jacobian will not depend on what type of numbers we have, but only on the structure of matrices in the Jacobians. Since the structure is the same throughout the whole simulation this benchmark will still be relevant to figure out which AD-implementation is best to solve this specific problem. Table 4.2 shows the three methods assembling the first residual function of the simulation 100 times. It shows the same tendency as Table 4.1, where `CJAD` and `MRST` are

Table 4.2: Table with speed benchmarks of different AD methods assembling the "Single-Phase Compressible AD Solver" residual function 100 times for different discretizations.

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

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

Se om det går an å bruke dette kapittelet som en overgang til lokal AD.

Local Automatic Differentiation

This chapter will consider a different approach on how to use AD to solve PDEs. When solving PDEs using a finite element method each cell will only depend on the neighbouring cells. In the FAD and CJAD implementations, the dependencies are stored in the discrete gradient and divergence operators. The implementation of FAD and CJAD calculates the residual function value and corresponding Jacobian for the whole grid simultaneously by having a vector to store the values and sparse matrices for the Jacobians. Hence when the discrete gradient and divergence operators are used in the calculation of the residual function in chapter 4, the Jacobian are automatically obtained with the structure seen in Figure 4.6. However, this structure is known from the grid properties given in the `G` variable introduced in section 4.1. So instead of calculating the residual function for all cells at once, the new approach takes one cell at the time and sums up the contributions from each neighbour. This is done for each cell until all the residual functions for each cell is calculated. This new approach is called local AD and the method is based on the same idea as how AD is done in OPM (*Open Porous Media, n.d.*). Since OPM is written in C and C++ it is interesting to see if you can write similar type of code in Julia and obtain the same computational efficiency.

Må teste noe mot dette og kommentere dette skal kunne

5.1 Implementation

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

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

Since we now only operate on one cell at the time the implementation of local AD is simpler than for FAD and CJAD. The value of the AD-variable is now only a scalar and the Jacobian matrices is replaced by a vector of derivatives. These are the derivatives with respect to the primary variables in the cell. To easily create a new LAD primary variable with a given number of derivatives and where the derivative with respect to itself is 1.0, I have created a function called `createVariable`:

```
function createVariable(val::Number, numDerivatives::Int, derivativeIndex::Int)
```

```

derivatives = zeros(numDerivatives)
if derivativeIndex > 0 && derivativeIndex <= numDerivatives
    derivatives[derivativeIndex] = 1.0
end
return LAD(val, derivatives)
end

```

For the single phase flow solver in chapter 4, the derivatives vector will be of length one, as each cell only contain one primary variable (pressure). The implementation of LAD is however with derivatives as a vector. This is for the opportunity to implement more complex simulations like a two or three phase simulation where each cell can contain water, oil and/or gas. The implementation of operators for the LAD struct is done similar as explained in section 3 for FAD and CJAD, but since we only have a vector of derivatives instead of a Jacobian matrix, the implementation is easier and follows the lines of the description from subsection 2.1.4.

Where the implementation of the local AD tool is easier than for FAD and CJAD, there is more work when calculating the residual functions. Since we do not use discrete gradient and divergence operators, but traverse through the grid cell by cell, we need somewhere to store the resulting values of the residual function. We also need a method to traverse through all the cells and to calculate the contributions from each neighbouring cell. Where the AD part of the code in FAD and CJAD were fully separated from the simulation, for local AD it is more integrated. This means that when using local AD to create the simulation it becomes a more application specific implementation than the method for FAD and CJAD. To create the flow solver from chapter 4 I have chosen to store the calculated values of the residual functions in another struct called `FlowSystem`:

```

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

```

This struct look very similar to the other AD structs, except from `globalJac` being one single sparse matrix instead of a vector of sparse matrices and each element in `globalJac` being a vector. At this point you might not understand why it is quicker to calculate the residuals cell by cell compared to everything in one go. The key reason for this is that we know that the structure of the global Jacobian will stay the same throughout the whole simulation. This means we can use the grid variable `G`, that contains the information on which cells are neighbours, and build the correct structure of `globalJac` once and for all. When we run the simulation we only change the values inside of `globalJac`, but the structure stays the same. This reuse of `globalJac` will save a lot of memory allocations and hence speed compared to FAD and CJAD which allocates new structs for each calculation. By creating a new constructor for `FlowSystem` that uses the grid variable `G`, the variables `eqVal` and `globalJac` will be allocated with the correct length and the correct structure before the simulation begins. For the grid in chapter 4 we have 1000 cells. This implies 1000 different pressure values and in addition we have the bottom-hole pressure (`bhp`) and the total production (`qS`). Hence `eqVal` will be a vector of length 1002.

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

```

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

```

The input parameter `well` is a struct that contains all necessary information about the well. The first line in `assembleFlowSystem!()` resets `eqVal` and `GlobalJac` such that the structures are unchanged, but all the values are zero. Then the function begin traversing through the grid and for every cell it iterates through all neighbouring cells. Be aware that the looping variable names `fromCell` and `toCell` can be a bit misleading when they represent bottom-hole-pressure and total production, as those primary variables do not belong to any cell. The `eqVal` and `globalJac` variables are updated in line number 6, 8 and 10 inside the inner loop. As a reminder, the residual functions that `FlowSystem` eventually will represent are the functions `presEq`, `rateEq` and `ctrlEq` defined in section 4.2:

```

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

```

In line number 6, `fromCell` and `toCell` are equal, but not the bottom-hole-pressure or total production. Here the first term in the sum in `presEq`, or the backward Euler term, is calculated. This is performed in an outer function called `timeDerivative()` that returns a LAD struct:

```

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

```

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

In line number 10, when `fromCell` and `toCell` are two neighbouring cells, the divergence term in `presEq` is calculated. Like for `timeDerivative()`, we create the primary AD-variables inside the function, but since we calculate the flux from one cell to another we have to be careful with which direction we calculate the flux and which cell we want the derivative with respect to. Since the varying variables is named `fromCell` and `toCell` it is natural that the function `flux()`, seen

below, calculates the flux from `fromCell` to `toCell`.

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

What needs to be chosen is which cell we want the derivative with respect to. The choice only affect which indices of the Jacobian the calculated derivatives should be added or subtracted to. In `flux()` I have decided to calculate the derivative with respect to `fromCell`. The two first lines shows this result where `pFrom` is initialized with a derivative of 1 and `pTo` as a constant. This choice leads to the following code for updating `FlowSystem` at line number 10 in `assembleFlowSystem!()`:

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

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

Line number 8 in `assembleFlowSystem!()` is the last line I have not commented and it is where the well equations, `rateEq` and `ctrlEq`, are handled. This is line is executed if both `fromCell` and `toCell` is either one of the cells containing a well, the bottom-hole pressure or the total production. The information on which cells fulfilling this condition lies in `well`. The calculation is performed by handling every case such that the correct residual function is calculated differentiated with respect to the correct variable and like for the fluxes, the resulting values and derivatives is added to `FlowSystem` at the correct indices.

om det
være litt
væg. Tror det
bli ekstremt
nivå.

Summed up the calculation of the residual functions using local AD is based on two modules. The first is the actual AD tool with the LAD struct, and the second is `FlowSystem` which keeps track of the global system and what calculations should be performed by the AD tool. With this approach we lose some of the advantages using the other AD tools where the discrete residual functions looked very similar to the continuous case, like explained in section 4.2. The main structure of `assembleFlowSystem!()` will however be the same no matter what type of simulation, hence making modifications to the code, or building another simulation, will not demand a complete change of code structure.

n dette ikke
agt litt

5.2 Optimizing Local AD

As explained in the beginning of this chapter, the main idea behind local AD is to reuse the Jacobian to save memory usage and extra memory allocations. The implementation given in section 5.1 has however left out some small, but key parts of the implementation that will massively decrease the memory and speed usage. This is purely implementation specific differences that has been left out to make the introduction to local AD more clear. It will not change the theory behind using local AD to solve PDE's.

5.2.1 Dynamic VS Static Arrays

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

Julia has a package called *StaticArrays.jl* (*n.d.*) that provides static arrays to the the built in arrays without the possibility for changing the size of the array. The package provides two types of vectors, `SVector` and `MVector`. Both are static vectors, but `MVector` is mutable, meaning the values inside the vector can be changed, where the values inside an `SVector` is final when the vector is defined. This makes `SVector` ideal for `LAD.derivatives` since when we perform operations on `LAD` variables, new `LAD` variables are returned. `MVector` is however optimal for the elements in the sparse matrix `FlowSystem.globalJac` as we know how long each element vector will be, but we want to be able to change the values inside each vector.

When the implementation is changed to use static vectors it needs to know how long these vectors are. When building a simulation it is not a problem to say in advance how many derivatives we need and when the compiler knows this before compilation it can optimize the machine code. OPM solves this by creating a template class such that the compiler always knows that for this simulation, the static array will always have this fixed length. This is important because we do not want to create `LAD` structs with different lengths for `LAD.derivatives` and we want all vectors in `FlowSystem.globalJac` to have that same length. If we define this clearly the compiler can optimize the machine code much better than if it does not know if for example some elements in `globalJac` suddenly will be allocated with a longer static mutable array. This will also make the implementation of the operators of `LAD` easier, as we know for certain that all `LAD` variables will have derivatives with the same length. The best implementation I have found for this in Julia is to declare a global constant variable in the local AD module such that the new implementation of `LAD` becomes:

Legg til både
og sjekk form

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

One disadvantage with this implementation is that the Local AD module needs to be modified to work

m det
oen annen
is ikke
kanskje
nteres at det
økkete

with simulations with more primary variables for each cell. The implementation of `FlowSystem` imports the local AD module, hence it can also use the global constant in the new implementation:

```
using StaticArrays, LocalAD
struct FlowSystem
    eqVal::Vector{Float64}
    globalJac::SparseMatrixCSC{MVector{NUM_DERIV,Float64}, Int}
end
```

5.3 Flow Solver with Local AD

Now that `FlowSystem` and LAD has been implemented with static arrays it is interesting to see how this method compares to FAD and CJAD in the simulation from chapter 4. Since OPM is a simulator ment to be much faster than MRST, it is assumed that since Julia is supposed to run as fast as C, the local AD approach will give a computational gain. With the implementation given in this chapter this is however not the case. It performs worse than CJAD and similar to FAD. As explained in subsection 1.3.1, profiling is an efficient way of finding the bottlenecks in a code, and this is a perfect example of when to use it - the code was expected to be faster, but it was not, so it is important to find out if there is a small part of the code that is slow. Figure 5.1a shows a screenshot of a profile result from running the flow solver simulation. The screenshot only contains blocks from the `flux` function. Two blocks are marked with squares and two other blocks with brighter diagonal stripes. The blocks with squares are time spent in `createVariable()` and the brighter diagonal stripes are AD calculations in `flux`.

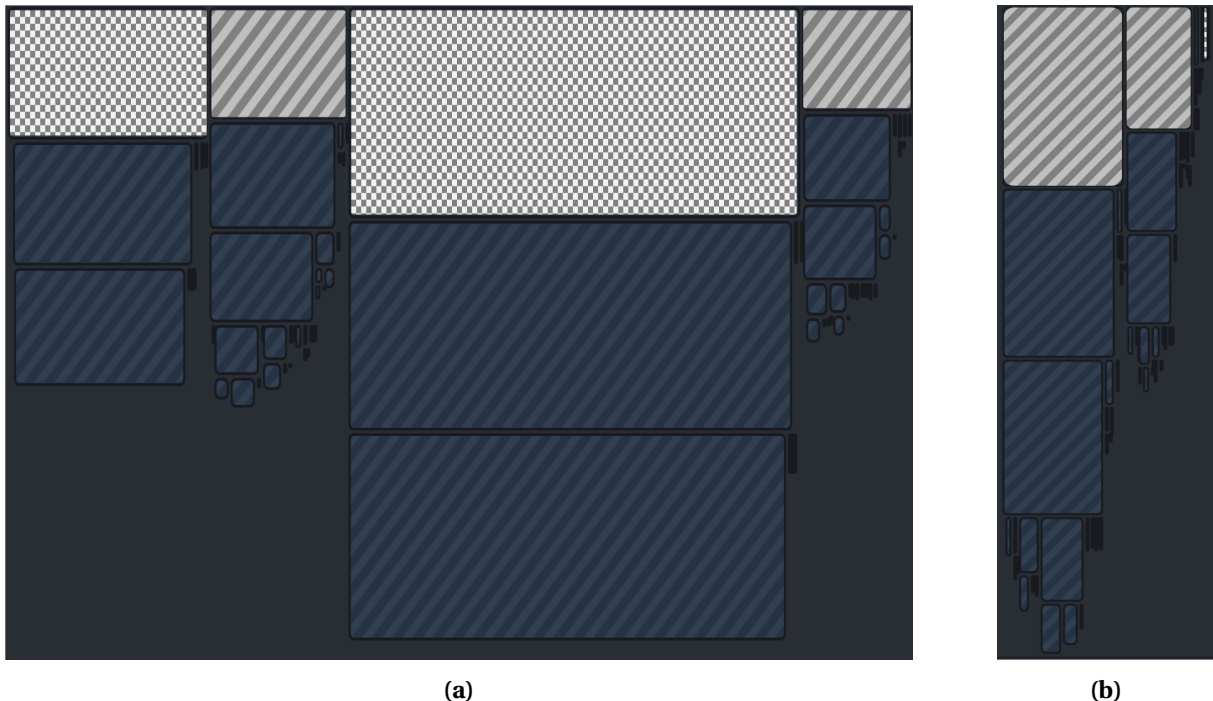


Figure 5.1: The figures shows screenshots of the profile result from the `flux` function in the flow solver simulation from chapter 4. The blocks marked with squares are time spent in `createVariable()` and the blocks with bright diagonal lines are calculations in `flux`. Figure 5.1a uses a slow version of `createVariable()` while Figure 5.1b uses a faster version.

It makes no sense that creating a LAD struct should take more time than the actual AD calculations. This indicates that the `createVariable()` function is poorly implemented. This is also the case as even though we only want to create a static `SVector`, we start by creating a dynamic vector of zeros, then we modify the vector by changing one of the values to one, and finally we convert the dynamic vector into a static `SVector`. The new implementation seen below is a much better implementation where the `SVector` is created immediately as the correct vector without any use of dynamic vectors.

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

The new profiling result with the updated `createVariable()` can be seen in Figure 5.1b

Se på parallellisering for lokal AD?

Drafts

6.1 Shared Libraries vs Static Libraries

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

6.2 Vectorization vs Non-vectorization

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

(*Johnson, 2017.01.21*) Ordinary vectorized code is fast, but not as fast as a hand-written loop (assuming loops are efficiently compiled, as in Julia) because each vectorized operation generates a new temporary array and executes a separate loop, leading to a lot of overhead when multiple vectorized operations are combined.

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>

Holy, T., Carlsson, K., Pfitzner, S., Fischer, K., 2019. A julia interpreter and debugger. Accessed 12.04.2019.

URL <https://julialang.org/blog/2019/03/debuggers>

Innes, M., 2018. Don't unroll adjoint: Differentiating ssa-form programs. arXiv preprint arXiv:1810.07951.

URL <https://github.com/FluxML/Zygote.jl>

Johnson, S. G., 2017.01.21. Accessed 13.03.2019.

URL <https://julialang.org/blog/2017/01/moredots>

Julia Community, 2018. Julia 1.0. Accessed 09.04.2019.

URL <https://julialang.org/blog/2018/08/one-point-zero>

Julia issue:dot operators, 2017. Accessed 02.12.2018.

URL <https://github.com/probcomp/GenExperimental.jl/issues/46>

Juno, n.d. Accessed 11.04.2019.

URL <http://junolab.org>

Juno profiler, n.d. Accessed 12.04.2019.

URL http://docs.junolab.org/latest/man/juno_frontend/#Profiler-1

Juno's Github page, n.d. Accessed 11.04.2019.

URL <https://github.com/JunoLab>

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

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>
- Profile.jl, n.d. Accessed 12.04.2019.
URL <https://docs.julialang.org/en/v1/manual/profile/index.html>
- Revels, J., Lubin, M., Papamarkou, T., 2016. Forward-mode automatic differentiation in julia. arXiv:1607.07892 [cs.MS].
URL <https://arxiv.org/abs/1607.07892>
- Single-phase Compressible AD Solver, n.d. Accessed 22.11.2018.
URL <https://www.sintef.no/contentassets/2551f5f85547478590ceca14bc13ad51/core.html#single-phase-compressible-ad-solver>
- SparseMatrixCSC docs, n.d. Accessed 27.02.2019.
URL <https://docs.julialang.org/en/v1/stdlib/SparseArrays/#SparseArrays.SparseMatrixCSC>
- StaticArrays.jl, n.d. Accessed 26.04.2019.
URL <https://github.com/JuliaArrays/StaticArrays.jl>
- The Julia Lab, n.d. Julia language research and development at mit. Accessed 21.11.2018.
URL <https://julia.mit.edu>
- Tic, n.d. Accessed 21.12.2018.
URL <https://uk.mathworks.com/help/matlab/ref/tic.html>
- @time docs, n.d. Accessed 12.04.2019.
URL <https://docs.julialang.org/en/v1/base/base/#Base.@time>
- 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>

