UiO : Department of Informatics
University of Oslo

Efficient Calculation of Derivatives using Automatic Differentiation

With applications in optimization and reservoir simulation

Vegard Ove Endresen Kjelseth

Master's Thesis Autumn 2014

Efficient Calculation of Derivatives using Automatic Differentiation

Vegard Ove Endresen Kjelseth

July 31, 2014

Abstract

There is a wide range of computational problems that require the knowledge of derivatives of mathematical functions to be solved. In cases where we know the functions we need the derivatives of, these can simply be calculated by hand. For cases where we need to handle arbitrary mathematical expressions, methods such as finite differences are often used to approximate the derivatives we need.

Both these methods bring with them certain disadvantages that are desirable to avoid. Finite differences can be too expensive to perform when the function we need derivatives of is expensive to evaluate. It also has the downside of introducing errors. Calculating derivatives by hand is the fastest option once the expressions for the derivatives have been calculated, but it is very error-prone and the sheer amount of derivatives necessary to calculate will often make this option too time-consuming to implement.

This thesis explores a different option, Automatic Differentiation. This is a method of computation that calculates the derivatives of mathematical functions to floating-point precision, without introducing any extra complexity to the code that uses it. This thesis will go through the theory and implementation behind this, before proceeding to show how this can be used to help solve computational problems in optimization and reservoir simulation using the Python programming language.

Contents

List of Figures

Preface

This thesis was written at the Department of Informatics at the University of Oslo in cooperation with SINTEF's Center of Applied Mathematics. Stein Krogstad has been my main supervisor during the work on this master's thesis, and I would like to thank him for his guidance and advice throughout this process.

I would also like to thank Halvor Møll Nilsen and Xing Cai for taking on the role as supporting supervisors. Finally, I would like to thank my parents, Bente and Ove, for their advice and moral support.

Chapter 1

Introduction

There are many computational problems that require knowledge of the derivatives of mathematical expressions in order to be solved. Examples are optimization problems where you might need to find the minimum or maximum value of a function, as well as problems where partial differential equations are solved using Newton's method. Two common ways of obtaining the necessary derivatives are calculating the derivatives of the relevant mathematical expressions by hand and approximating the derivatives with numerical differentiation.

Calculating the derivatives by hand obviously does not lend itself well to creating general solvers that can handle arbitrary mathematical expressions. However, it is a viable option for cases where we know all the mathematical expressions we need the derivatives of. Performance-wise this is the best option, but it can be very time consuming and it is easy to introduce bugs by making mistakes in the calculation of the derivatives.

Unlike calculating the derivatives by hand, numerical differentiation allows us to to calculate approximations to the derivative of any mathematical function. This makes it a possible solution for general solvers, but it has the drawback of introducing errors and being too time consuming in cases where we need a large amount of derivatives and the cost of evaluating the mathematical function at hand is high.

Ideally we would like to have a solution that gives us the best of both worlds, in other words simple access to exact derivatives of arbitrary mathematical expressions without having to sacrifice performance. In reality the best we can hope to accomplish is a compromise that supplies us with the exact derivatives without too much of a performance hit. The question that then arises is whether or not such a solution can be made?

The answer is Automatic Differentiation, which is a method of calculating the derivatives of any expression to floating point precision. This can be programmed in a way that makes the derivatives immediately accessible in any program that uses it, thus eliminating the need to write any extra code to calculate the relevant derivatives and making it easier to avoid bugs. In terms of performance it is naturally slower than if we calculated all the derivative expressions beforehand, while compared to using numerical differentiation it will sometimes be faster with the added benefit of no error term.

These factors make the AD framework a good choice for two kinds of applications. For general solvers where we need to find derivatives of arbitrary expressions it can be used instead of numerical differentiation in cases where this is too slow or where we require the exact derivatives. For specialized solvers, where we know the relevant mathematical expressions, the AD framework can be used instead of calculating the derivatives by hand. This allows us to write a complete program faster and if it has to be put to use in a production environment later on, and faster execution is required, it can be rewritten to focus on performance.

The ideas behind Automatic Differentiation have been around for a long time, with the concept being introduced by Wengert as early as 1964 [15]. Automatic Differentiation was further developed the following decades, with Rall publishing a book about it in 1981 [10]. Towards the end of the 1980s and early 1990s a large amount of articles were published on the subject, including Neidinger's article in 1992 [6], which has been used as the starting point for the work on Automatic Differentiation in this thesis. Since then several authors have written articles on ways to improve performance, which includes Griewank in [3] and Neidinger in [7], among others.

The focus of this thesis will be on creating a fully functional Automatic Differentiation framework that can be used to help solve computational tasks requiring derivatives of mathematical expressions. The underlying theory and the implementation of such a framework in Python is the topic of part I of the thesis. The second part of the thesis focuses on applications where the AD framework is used. This includes solving the differential equations related to an oil reservoir, as well as several examples of how it can be used in optimization. The final part of the thesis consists of the conclusion, where the results are discussed, what could have been done differently and how to improve further on what has been made.

Part I

Automatic Differentiation

Chapter 2

Introduction to Automatic Differentiation

Automatic Differentiation (AD) is a method of computation used to calculate the derivatives of any mathematical expression. This can be done for derivatives up to an arbitrary order, involving any number of variables, and the error is bounded by the floating-point errors accumulated during the calculations. To explain how this can be done we will look at how it should work ideally for first order derivatives of expressions involving a single variable, and then outline how that can be achieved.

Consider that we wish to calculate the value of the expression $x \sin(x)$ for $x = 3$, and that we also want to access the first order derivative at this point. The pseudo code below illustrates how this could be done:

```
# Calculate expression
x = 3
expr = x*sin(x)

# Print value and first order derivative
print expr
print expr.der
```

In this example the derivative of the expression is somehow stored in the expression variable in addition to the value. We can achieve this in Python by creating a class that can hold both the value and derivative. Let us call this class AD, in which case the pseudo code looks as follows:

```
# Calculate expression
x = AD(val=3,der=1)
expr = x*sin(x)

# Print value and first order derivative
print expr.val
print expr.der
```

Here we start off by creating an instance of the AD class where the value equals 3 and the first order derivative equals 1, since the derivative of x with respect to x is simply 1. The expression is calculated next and the value and derivative are printed. Somehow the operation of calculating the expression must result in a new AD instance with the correct value and derivative. We can start off by creating our own sin function that takes an AD instance $g(x)$ as an argument and returns a new AD instance with the value $\sin(g(x))$ and the derivative equal to $\cos(g(x))\, g'(x)$.

```
def ADsin(g):
    # Calculate value
    val = sin(g.val)

    # Calculate derivative
    der = cos(g.val)*g.der

    # Create AD instance
    new = AD(val=val,der=der)

    # Return object
    return new
```

Using this function our example code now looks as follows:

```
# Calculate expression
x = AD(3)       # x.val = 3, x.der = 1
expr = x*ADsin(x)

# Print value and first order derivative
print expr.val
print expr.der
```

The only change is that we now use our custom sin function to calculate $\sin(x)$. Since this returns an AD object, we need to have a way of multiplying two AD objects together. Python gives us the option of defining special functions that are called when we multiply a variable with an object of a custom class, a process called overloading. For this example we need to define the *__rmul__* function in the AD class, which could look like this:

```
class AD:
    def __rmul__(self,g):
        # Calculate value
        val = self.val*g.val

        # Calculate derivative
        der = self.val*g.der + self.der*g.val

        # Create AD object
        new = AD(val=val,der=der)

        # Return AD object
        return new
```

In the code above, the product rule was used to calculate the derivative. With this final function defined the example script would now work, with the expression variable now including the correct value and derivative.

We could now proceed to create functions for not just sin, but other mathematical functions as well. Additionally, Python makes it possible to overload operators not just for multiplication like above, but for other common mathematical operations as well. Implementing all this would leave us with the ability to calculate most mathematical expressions and always have the first order derivative readily available. However, we are interested in creating an AD framework that can deal with more than one variable and where we can access derivatives up to an arbitrary order. To do that the mathematical formulas become more complicated, but the general idea of overloading operators and creating functions for a wide range of mathematical functions remain the same. In the next chapter we will derive the necessary mathematical formulas to create a general AD framework, before we proceed to discuss how this is implemented.

Chapter 3

Derivatives of multivariate functions to arbitrary order

3.1 Introduction

In general we are interested in being able to find derivatives up to an arbitrary order involving any number of variables. To get started it is necessary to determine how to store the derivatives. Let n denote the number of variables, max denote the maximum order of the derivatives and $\vec{x}$ denote the values for the variables $x_1, \ldots, x_n$. We can store all the derivatives in a multidimensional array with n dimensions and a length of $max + 1$ along each of the dimensions as presented in [6, p. 3].

If we access the array with the indices $[i_1, i_2, \ldots, i_n]$ the corresponding value is the value of the expression differentiated i_1 times with regard to the first variable, i_2 times with regard to the second, and in general i_j times with regard to the j'th variable. Additionally, all derivatives are evaluated at the point $\vec{x}$. For indices where the sum exceeds the maximum order of the derivatives, the value will not be available.

Assume that $n = 2$ and that we want to find all derivatives up to the order $max = 2$ for the general expression $f(\vec{x})$. We store the derivatives and the value in the matrix F, with dimensionality equal to 2 with a length of 3 along each dimension.

$$F = \begin{pmatrix} f(\vec{x}) & f_{x_2}(\vec{x}) & f_{x_2x_2}(\vec{x}) \\ f_{x_1}(\vec{x}) & f_{x_1x_2}(\vec{x}) & \\ f_{x_1x_1}(\vec{x}) & & \end{pmatrix}$$

Assume that we want to add two expressions $f(\vec{x})$ and $g(\vec{x})$. Since differentiation is distributive we simply need to add the corresponding matrices. [6, p. 3]

$$F+G=\begin{pmatrix} f(\vec{x}) & f_{x_2}(\vec{x}) & f_{x_2x_2}(\vec{x}) \\ f_{x_1}(\vec{x}) & f_{x_1x_2}(\vec{x}) & \\ f_{x_1x_1}(\vec{x}) & & \end{pmatrix}+\begin{pmatrix} g(\vec{x}) & g_{x_2}(\vec{x}) & g_{x_2x_2}(\vec{x}) \\ g_{x_1}(\vec{x}) & g_{x_1x_2}(\vec{x}) & \\ g_{x_1x_1}(\vec{x}) & & \end{pmatrix}$$

The same thing holds for subtraction. If we want to subtract the expression $g(\vec{x})$ from $f(\vec{x})$ we simply subtract the matrix G from F.

$$F-G=\begin{pmatrix} f(\vec{x}) & f_{x_2}(\vec{x}) & f_{x_2x_2}(\vec{x}) \\ f_{x_1}(\vec{x}) & f_{x_1x_2}(\vec{x}) & \\ f_{x_1x_1}(\vec{x}) & & \end{pmatrix}-\begin{pmatrix} g(\vec{x}) & g_{x_2}(\vec{x}) & g_{x_2x_2}(\vec{x}) \\ g_{x_1}(\vec{x}) & g_{x_1x_2}(\vec{x}) & \\ g_{x_1x_1}(\vec{x}) & & \end{pmatrix}$$

A slightly different and indirect way of storing the values of derivatives is to store the corresponding multivariate Taylor coefficients instead of the actual derivatives. The matrix F corresponding to the expression $f(\vec{x})$ will then take the following form.

$$F=\begin{pmatrix} f(\vec{x}) & \frac{1}{1!}f_{x_2}(\vec{x}) & \frac{1}{2!}f_{x_2x_2}(\vec{x}) \\ \frac{1}{1!}f_{x_1}(\vec{x}) & \frac{1}{1!1!}f_{x_1x_2}(\vec{x}) & \\ \frac{1}{2!}f_{x_1x_1}(\vec{x}) & & \end{pmatrix}$$

The formulas for adding and subtracting expressions stays the same when storing the Taylor coefficients instead of the derivatives. In the following sections we will use this as the underlying data representation when deriving formulas for other operators and functions. The reasoning behind this is that it makes many of the necessary calculations a lot faster, which will be evident later on.

3.2 Notation

To denote the derivatives of expressions with an arbitrary number of variables we will use multi-indices. A multi-index $\vec{k}$ is defined as:

$$\vec{k}=(k_1,k_2,\ldots,k_n)\quad k_i\in\mathbb{N}_0\quad \forall\, i\in\{1,2,\ldots,n\}$$

Additionally any multi-indices $\vec{k}$ and $\vec{j}$ satisfy the following relations:

$$\vec{k}\pm\vec{j}=(k_1\pm j_1,k_2\pm j_2,\ldots,k_n\pm j_n)$$

$$\vec{k}\leq\vec{j}\ \Leftrightarrow\ k_i\leq j_i\ \forall\, i\in\{1,2,\ldots,n\}$$

$$|\vec{k}|=k_1+k_2+\cdots+k_n$$

$$\vec{k}!=k_1!k_2!\ldots k_n!$$

$$\binom{\vec{k}}{\vec{j}}=\binom{k_1}{j_1}\binom{k_2}{j_2}\cdots\binom{k_n}{j_n}=\frac{\vec{k}!}{\vec{j}!\left(\vec{k}-\vec{j}\right)!}$$

Additionally we will use the following notation for differential operators:

$$\partial_i^{k_i} = \frac{\partial^{k_i}}{\partial x_i^{k_i}}$$

$$\partial^{\vec{k}} = \partial_1^{k_1} \partial_2^{k_2} \dots \partial_n^{k_n}$$

$$\partial^{\vec{k}} \partial^{\vec{j}} = \partial^{\vec{k}+\vec{j}}$$

The space of all the Taylor coefficient multi-indices, including the 0 vector corresponding to the expression's value, is described as follows:

$$A_d = \left\{ \vec{k} \mid |\vec{k}| \leq max \right\}$$

We will encounter many sums that involve multi-indices in this section. These look as follows:

$$\sum_{\vec{j}=\vec{0}}^{\vec{k}} h\left(\vec{j}, \vec{k}\right) \quad \sum_{\substack{\vec{j}>\vec{0} \\ \vec{j}\leq\vec{k}}} h\left(\vec{j}, \vec{k}\right) \quad \sum_{\vec{j}<\vec{k}} h\left(\vec{j}, \vec{k}\right)$$

What is meant by the left-most sum is that the we are summing the terms $h\left(\vec{j}, \vec{k}\right)$ for all $\vec{j} \in A_d$ that satisfy $\vec{0} \leq \vec{j} \leq \vec{k}$ [6, p. 4]. The middle means that we are performing the sum for all $\vec{j} \in A_d$ that satisfy $\vec{0} \leq \vec{j} \leq \vec{k}$, with the exception of $\vec{j} = \vec{0}$. The right-most means that we are performing the sum for all $\vec{j} \in A_d$ that satisfy $\vec{0} \leq \vec{j} \leq \vec{k}$, with the exception of $\vec{j} = \vec{k}$.

For simplicity we will sometimes drop the arguments to functions and simply use f to describe $f(\vec{x})$. We will use $T_{f,\vec{k}}$ to denote the Taylor coefficient for the expression f corresponding to the derivative $\vec{k}$. Additionally it is assumed that the Taylor expansion is done around the point $\vec{x}$ that we're considering, so all Taylor coefficients are evaluated in this point.

3.3 Derivation of formulas

In this section we will look at the derivation of some of the formulas for the Taylor coefficients resulting from different mathematical functions and operators. We will derive the expressions for multiplication and the natural logarithm, which will outline the general ideas that can be used to prove the formulas for a range of other mathematical expressions as well. These additional formulas will simply be listed in this section, while the full proofs can be found in the appendix. We will also present a corollary that is necessary to derive some of the formulas.

3.3.1 Multiplication

To derive a formula for the derivatives of the product fg we will use the generalized Leibniz's Rule [6, p. 4], which states that if u and v are real-valued functions on an open domain in $\mathcal{R}^n$ the following equation holds:

$$\partial^{\vec{k}}(fg) = \sum_{\vec{j}=\vec{0}}^{\vec{k}} \binom{\vec{k}}{\vec{j}} \partial^{\vec{j}} f \;\; \partial^{\vec{k}-\vec{j}} g \tag{3.1}$$

This gives us the formula for finding the derivatives. Given the expression above the Taylor coefficients,$T_{fg,\vec{k}}$, are given as:

$$\begin{aligned}
T_{fg,\vec{k}} &= \tfrac{1}{\vec{k}!}\partial^{\vec{k}}(fg) \\
&= \tfrac{1}{\vec{k}!}\sum_{\vec{j}=\vec{0}}^{\vec{k}} \tbinom{\vec{k}}{\vec{j}} \partial^{\vec{j}} f \;\; \partial^{\vec{k}-\vec{j}} g \\
&= \tfrac{1}{\vec{k}!}\sum_{\vec{j}=\vec{0}}^{\vec{k}} \tfrac{\vec{k}!}{\vec{j}!(\vec{k}-\vec{j})!} \partial^{\vec{j}} f \;\; \partial^{\vec{k}-\vec{j}} g \\
&= \sum_{\vec{j}=\vec{0}}^{\vec{k}} \tfrac{1}{\vec{j}!(\vec{k}-\vec{j})!} \partial^{\vec{j}} f \;\; \partial^{\vec{k}-\vec{j}} g \\
&= \sum_{\vec{j}=\vec{0}}^{\vec{k}} \left(\tfrac{1}{\vec{j}!}\partial^{\vec{j}} f\right)\left(\tfrac{1}{(\vec{k}-\vec{j})!}\partial^{\vec{k}-\vec{j}} g\right) \\
&= \sum_{\vec{j}=\vec{0}}^{\vec{k}} T_{f,\vec{j}} T_{g,\vec{k}-\vec{j}}
\end{aligned}$$

This gives us the final formula which shows that we can calculate all the desired Taylor coefficients as long as we know the Taylor coefficients of f and g.

$$T_{fg,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}} T_{f,\vec{j}} T_{g,\vec{k}-\vec{j}} \tag{3.2}$$

This formula also illustrates the computational advantage of storing Taylor coefficients instead of the actual derivatives, since it eliminates the need to calculate the binomials present in the generalized Leibniz' Rule.

3.3.2 Supporting Corollary

To derive formulas for different functions and operators it is necessary to introduce a corollary that we will use in later derivations. A similar corollary was derived by Neidinger in [6, p. 5] for derivatives. We will use the same idea as presented by Neidinger, but we will instead derive a corollary that can be used to calculate Taylor coefficients.

Corollary. *Let $\vec{e}$ be a one-order vector, meaning that $|\vec{e}| = 1$, and assume that $\vec{e} \leq \vec{k}$ for $\vec{k} \in A_d$ and that i is the index where $e_i = 1$. Let f,g and h be real valued smooth functions on an open domain in $\mathcal{R}^n$ that satisfy the following relation:*

$$\partial^{\vec{e}} h = f \, \partial^{\vec{e}} g$$

The Taylor coefficients, $T_{h,\vec{k}}$, can then be calculated using the following formula.

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{f,\vec{j}}\, T_{g,\vec{k}-\vec{j}} \tag{3.3}$$

Proof. To prove this suppose that h,f and g are real-valued smooth functions on an open domain in $\mathcal{R}^n$ such that $\partial^{\vec{e}} h = f\ \partial^{\vec{e}} g$ for some $\vec{e}$ that satisfies $|\vec{e}| = 1$ and $\vec{e} \leq \vec{k}$ for $\vec{k} \in A_d$. Let $h' = \partial^{\vec{e}} h$ and $g' = \partial^{\vec{e}} g$. By using the Generalized Leibniz's Rule (3.1) on $h' = fg'$ we get the following expression:

$$\partial^{\vec{k}-\vec{e}} h' = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \binom{\vec{k}-\vec{e}}{\vec{j}} \partial^{\vec{j}} f\ \ \partial^{\vec{k}-\vec{e}-\vec{j}} g' \tag{3.4}$$

We can rewrite two of the terms in the above equation as follows:

$$\begin{aligned} \partial^{\vec{k}-\vec{e}} h' &= \partial^{\vec{k}-\vec{e}} \partial^{\vec{e}} h = \partial^{\vec{k}} h \\ \partial^{\vec{k}-\vec{e}-\vec{j}} g' &= \partial^{\vec{k}-\vec{e}-\vec{j}} \partial^{\vec{e}} g = \partial^{\vec{k}-\vec{j}} g \end{aligned}$$

Substituting this into the (3.4) yields the following expression:

$$\partial^{\vec{k}} h = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \binom{\vec{k}-\vec{e}}{\vec{j}} \partial^{\vec{j}} f\ \ \partial^{\vec{k}-\vec{j}} g = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{\left(\vec{k}-\vec{e}\right)!}{\vec{j}!\left(\vec{k}-\vec{j}-\vec{e}\right)!} \partial^{\vec{j}} f\ \ \partial^{\vec{k}-\vec{j}} g \tag{3.5}$$

Since $\vec{e}$ is a one-order vector, we know that $e_i = 1$ for a single i, while the rest are equal to 0. Let i denote the index where $e_i = 1$. Using this knowledge we can rewrite the factorials as follows:

$$\begin{aligned} \left(\vec{k}-\vec{e}\right)! &= \tfrac{\vec{k}!}{k_i} \\ \left(\vec{k}-\vec{j}-\vec{e}\right)! &= \tfrac{\left(\vec{k}-\vec{j}\right)!}{k_i - j_i} \end{aligned}$$

Note that since $\vec{e} \leq \vec{k}$ we know that $k_i \geq 1$. Additionally, we know that $\vec{j} \leq \vec{k} - \vec{e}$ so $j_i \leq k_i - e_i < k_i$. This shows that we're not at risk of dividing by zero in any of the expressions above. Substituting these expressions into (3.5) yields:

$$\partial^{\vec{k}} h = \vec{k}! \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} \left(\frac{1}{\vec{j}!} \partial^{\vec{j}} f\right) \left(\frac{1}{\left(\vec{k}-\vec{j}\right)!} \partial^{\vec{k}-\vec{j}} g\right) \tag{3.6}$$

Dividing by $\vec{k}!$ in (3.6) yields the final expression:

$$T_{h,\vec{k}} = \frac{\partial^{\vec{k}} h}{\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} \left(\frac{\partial^{\vec{j}} f}{\vec{j}!}\right) \left(\frac{\partial^{\vec{k}-\vec{j}} g}{\left(\vec{k}-\vec{j}\right)!}\right) = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{f,\vec{j}}\, T_{g,\vec{k}-\vec{j}}$$

□

Note that we can not use equation (3.3) to calculate $T_{h,\vec{k}}$ for $\vec{k} = \vec{0}$ since $\vec{k} \geq \vec{e}$ and $\vec{e}$ is a one-order vector. This means that we can not find $T_{h,\vec{0}}$ using this formula. This is not a problem, however, since $T_{h,\vec{0}}$ is just equal to the function value $h\left(\vec{x}\right)$.

3.3.3 Natural Logarithm

Assume that $\vec{h}(\vec{x}) = \ln(g(\vec{x}))$. Then $\partial^{\vec{e}} h = \frac{\partial^{\vec{e}} g}{g}$ for any one-order vector $\vec{e}$, which can be written as $\partial^{\vec{e}} g = g\ \partial^{\vec{e}} h$. Using equation (3.3) yields the following formula for any $\vec{k} \geq \vec{e}$:

$$T_{g,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{g,\vec{j}}\, T_{h,\vec{k}-\vec{j}}$$

The first term in this summation for $\vec{j} = \vec{0}$ is $T_{g,\vec{0}}\ T_{h,\vec{k}}$. Pulling this out of the sum yields:

$$T_{g,\vec{k}} = T_{g,\vec{0}}\ T_{h,\vec{k}} + \sum_{\substack{\vec{j}>\vec{0} \\ \vec{j}\leq\vec{k}-\vec{e}}} \frac{k_i - j_i}{k_i} T_{g,\vec{j}}\, T_{h,\vec{k}-\vec{j}}$$

Rearranging the expression and using the fact that $T_{g,\vec{0}} = g(\vec{x})$ yields the final formula:

$$T_{h,\vec{k}} = \left(T_{g,\vec{k}} - \sum_{\substack{\vec{j}>\vec{0} \\ \vec{j}\leq\vec{k}-\vec{e}}} \frac{k_i - j_i}{k_i} T_{g,\vec{j}}\, T_{h,\vec{k}-\vec{j}} \right) / g(\vec{x}) \tag{3.7}$$

This expression requires us to know the Taylor coefficients of h, which are not yet known prior to the calculation. However, for a given $\vec{k}$ it is only necessary to know the values of the Taylor coefficients of a lower order. This makes it possible to calculate $T_{h,\vec{k}}$ for all $\vec{k} \in A_d$ as long as all Taylor coefficients of a lower order have been calculated beforehand. Note that dividing by 0 might cause a problem with this formula, but this only happens if we were calculating the logarithm of 0 to begin with.

3.3.4 Remaining functions and operators

The proofs for the remaining functions and operators follow the same methods as shown for multiplication and the natural logarithm. These can all be found in the appendix. This section lists the remaining formulas.

Division

For the calculation $h = f/g$ we get the following formula for any $\vec{k} \in A_d$ and one-order vector $\vec{e}$ that satisfies $\vec{e} \leq \vec{k}$:

$$T_{h,\vec{k}} = \frac{T_{f,\vec{k}} - \sum_{\vec{j}<\vec{k}} T_{h,\vec{j}} T_{g,\vec{k}-\vec{j}}}{g(\vec{x})} \tag{3.8}$$

Exponential function

For the calculation $h = e^g$ we get the following formula for any $\vec{k} \in A_d$ and one-order vector $\vec{e}$ that satisfies $\vec{e} \leq \vec{k}$:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{g,\vec{k}-\vec{j}} \tag{3.9}$$

Square Root

For the calculation $\vec{h}\left(\vec{x}\right) = \sqrt{g\left(\vec{x}\right)}$ we get the following formula for any $\vec{k} \in A_d$ and one-order vector $\vec{e}$ that satisfies $\vec{e} \leq \vec{k}$:

$$T_{h,\vec{k}} = \left(T_{g,\vec{k}} - 2 \sum_{\substack{\vec{j}>\vec{0} \\ \vec{j}\leq\vec{k}-\vec{e}}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{h,\vec{k}-\vec{j}} \right) / 2h\left(\vec{x}\right) \tag{3.10}$$

Inverse Trigonometric Functions

For the calculation $\vec{h}\left(\vec{x}\right) = \arctan\left(g\left(\vec{x}\right)\right)$ we need to calculate the expression f given as $f = \frac{1}{1+g^2}$. We get the following formula for any $\vec{k} \in A_d$ and one-order vector $\vec{e}$ that satisfies $\vec{e} \leq \vec{k}$:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{f,\vec{j}}\, T_{g,\vec{k}-\vec{j}} \tag{3.11}$$

For arcsin and arccos the formula is the same as equation (3.11), but where $f = \frac{1}{\sqrt{1-g^2}}$ for arcsin and $f = \frac{-1}{\sqrt{1-g^2}}$ for arccos.

Trigonometric Functions

For the calculations $h\left(\vec{x}\right) = \sin\left(f\left(\vec{x}\right)\right)$ and $g\left(\vec{x}\right) = \cos\left(f\left(\vec{x}\right)\right)$ we get the following formulas for any $\vec{k} \in A_d$ and one-order vector $\vec{e}$ that satisfies $\vec{e} \leq \vec{k}$:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{g,\vec{j}}\, T_{f,\vec{k}-\vec{j}} \tag{3.12}$$

$$T_{g,\vec{k}} = -\sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{f,\vec{k}-\vec{j}} \tag{3.13}$$

This shows that to calculate one we need to know the other as well, so both have to be calculated even if we are only interested in one of them.

Additional functions and operators

A lot of other functions that have not been mentioned so far can be calculated using the functions we have derived formulas for. Five examples of these follow below.

$$\begin{array}{lll} h = g^f & \Rightarrow & h = e^{f*\ln(g)} \\ h = \sinh(g) & \Rightarrow & h = 0.5*(e^g - e^{-g}) \\ h = \cosh(g) & \Rightarrow & h = 0.5*(e^g + e^{-g}) \\ h = \tanh(g) & \Rightarrow & h = \frac{1-e^{-2g}}{1+e^{-2g}} \\ h = \tan(g) & \Rightarrow & h = \frac{\sin(g)}{\cos(g)} \end{array}$$

3.4 Example usage of formulas

To show how these formulas are used in a programming environment we will take a look at how they can be used to calculate the derivatives of two mathematical expressions. We will consider examples with two variables, the first being x and the second being y, and consider derivatives up to the second order. The variable x will be set to 3 and y will be set to 2. The Taylor coeffecient matrices for x and y, X and Y respectively, will look as follows:

$$X = \begin{pmatrix} 3 & 0 & 0 \\ 1 & 0 & \\ 0 & & \end{pmatrix} \qquad Y = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 0 & \\ 0 & & \end{pmatrix}$$

Now assume that we want to calculate the expression $g(x,y) = xy$. Recall that the formula (3.2) for the Taylor coefficients of two expressions multiplied together was given as follows:

$$T_{fg,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}} T_{f,\vec{j}} T_{g,\vec{k}-\vec{j}} \tag{3.14}$$

Using this we get the following Taylor coefficents for $g(x,y)$:

$$\begin{array}{lll} T_{xy,(0,0)} & = & T_{x,(0,0)}T_{y,(0,0)} \\ & = & 3*2 = 6 \\ T_{xy,(0,1)} & = & T_{x,(0,0)}T_{y,(0,1)} + T_{x,(0,1)}T_{y,(0,0)} \\ & = & 3*1+0*2 = 3 \\ T_{xy,(0,2)} & = & T_{x,(0,0)}T_{y,(0,2)} + T_{x,(0,1)}T_{y,(0,1)} + T_{x,(0,2)}T_{y,(0,0)} \\ & = & 3*0+0*1+0*2 = 0 \\ T_{xy,(1,0)} & = & T_{x,(0,0)}T_{y,(1,0)} + T_{x,(1,0)}T_{y,(0,0)} \\ & = & 3*0+1*2 = 2 \\ T_{xy,(1,1)} & = & T_{x,(0,0)}T_{y,(1,1)} + T_{x,(0,1)}T_{y,(1,0)} + T_{x,(1,0)}T_{y,(0,1)} + T_{x,(1,1)}T_{y,(0,0)} \\ & = & 3*0+0*0+1*1+0*2 = 1 \\ T_{xy,(2,0)} & = & T_{x,(0,0)}T_{y,(2,0)} + T_{x,(1,0)}T_{y,(1,0)} + T_{x,(2,0)}T_{y,(0,0)} \\ & = & 3*0+1*0+0*2 = 0 \end{array}$$

This gives us the following matrix of Taylor coefficients for g:

$$G = \begin{pmatrix} 6 & 3 & 0 \\ 2 & 1 & \\ 0 & & \end{pmatrix}$$

which is the result we expected. Using the same variables as before let us assume that we want to calculate the expression $h(x, y) = e^g = e^{xy}$. Recall that the formula (3.9) for the Taylor coefficients of the exponent of an expression was given as follows for any any $\vec{k} \in A_d$ and one-order vector $\vec{e}$ that satisfies $\vec{e} \leq \vec{k}$:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{g,\vec{k}-\vec{j}} \tag{3.15}$$

Also recall that this formula is not used for calculating $T_{h,(0,0)}$, which is simply the value of the expression e^6. The following notation will be used to denote the one-order vector used in the calculation of a given Taylor coefficient:

$$\left[T_{h,\vec{k}}\right]^{\vec{e}=(e_0,e_1)}$$

Using the formula yields the following values for the different Taylor coefficients:

$$\begin{aligned}
\left[T_{h,(\vec{0,1})}\right]^{\vec{e}=(0,1)} &= \tfrac{1-0}{1} T_{h,(0,0)} T_{g,(0,1)} \\
&= e^6 * 3 = 3e^6 \\
\left[T_{h,(\vec{0,2})}\right]^{\vec{e}=(0,1)} &= \tfrac{2-0}{2} T_{h,(0,0)} T_{g,(0,2)} + \tfrac{2-1}{2} T_{h,(0,1)} T_{g,(0,1)} \\
&= e^6 * 0 + \tfrac{1}{2} * 3e^6 * 3 = \tfrac{9}{2} e^6 \\
\left[T_{h,(\vec{1,0})}\right]^{\vec{e}=(1,0)} &= \tfrac{1-0}{1} T_{h,(0,0)} T_{g,(1,0)} \\
&= e^6 * 2 = 2e^6 \\
\left[T_{h,(\vec{1,1})}\right]^{\vec{e}=(1,0)} &= \tfrac{1-0}{1} T_{h,(0,0)} T_{g,(1,1)} + \tfrac{1-0}{1} T_{h,(0,1)} T_{g,(1,0)} \\
&= e^6 * 1 + 3e^6 * 2 = 7e^6 \\
\left[T_{h,(\vec{2,0})}\right]^{\vec{e}=(1,0)} &= \tfrac{2-0}{2} T_{h,(0,0)} T_{g,(2,0)} + \tfrac{2-1}{2} T_{h,(1,0)} T_{g,(1,0)} \\
&= e^6 * 0 + \tfrac{1}{2} * 2e^6 * 2 = 2e^6
\end{aligned}$$

This gives us the following matrix of Taylor coefficients for h:

$$H = \begin{pmatrix} e^6 & 3e^6 & \frac{9}{2}e^6 \\ 2e^6 & 7e^6 & \\ 2e^6 & & \end{pmatrix}$$

which is the result we expected.

3.5 The special case of first order derivatives

Although the general formulas derived in the previous sections can be used to calculate any order of derivatives, in the case of first order derivatives it is desirable to use a set of simpler formulas. The reasoning behind this is that the simplicity of these formulas allows us to implement them much more efficiently in a programming environment. The formulas for the first order derivatives of the functions and operators described above follow below:

- $\frac{\partial}{\partial x_i}(u+v) = \frac{\partial u}{\partial x_i} + \frac{\partial v}{\partial x_i}$
- $\frac{\partial}{\partial x_i}(u-v) = \frac{\partial u}{\partial x_i} - \frac{\partial v}{\partial x_i}$
- $\frac{\partial}{\partial x_i}(uv) = \frac{\partial u}{\partial x_i}v + u\frac{\partial v}{\partial x_i}$
- $\frac{\partial}{\partial x_i}\left(uv^{-1}\right) = \frac{\partial u}{\partial x_i}v^{-1} - u\frac{\partial v}{\partial x_i}v^{-2}$
- $\frac{\partial}{\partial x_i}(e^u) = e^u\frac{\partial u}{\partial x_i}$
- $\frac{\partial}{\partial x_i}(\log(u)) = \frac{\partial u}{\partial x_i}u^{-1}$
- $\frac{\partial}{\partial x_i}\left(\sqrt{u}\right) = \frac{1}{2}\frac{\partial u}{\partial x_i}u^{-\frac{1}{2}}$
- $\frac{\partial}{\partial x_i}(\cos(u)) = -\frac{\partial u}{\partial x_i}\sin(u)$
- $\frac{\partial}{\partial x_i}(\sin(u)) = \frac{\partial u}{\partial x_i}\cos(u)$
- $\frac{\partial}{\partial x_i}(\arcsin(u)) = \frac{\partial u}{\partial x_i}\left(1-u^2\right)^{-\frac{1}{2}}$
- $\frac{\partial}{\partial x_i}(\arccos(u)) = -\frac{\partial u}{\partial x_i}\left(1-u^2\right)^{-\frac{1}{2}}$
- $\frac{\partial}{\partial x_i}(\arctan(u)) = \frac{\partial u}{\partial x_i}\left(1+u^2\right)^{-1}$

Since first order Taylor coefficients are simply equal to the first order derivatives these formulas are really just simpler formulations of the more general formulas derived in the previous sections. However, working with these formulas instead of the general ones when finding first order derivatives allows us to easily vectorize our calculations in the implementation. Vectorization is the process of working on an entire vector of values at once instead of each of the elements individually, something that results in much faster calculations.

3.6 Alternate approaches

Automatic Differentiation was, as noted in the thesis introduction, first introduced in 1964. For any field of research that has been around for 50 years there is bound to be a lot of different approaches, and Automatic Differentiation is no different. Listing all of these alternative approaches would clearly be next to impossible, so we will instead look briefly at a few more recent ideas that could improve on what has been presented in this chapter. We will first consider the use of univariate Taylor series, before discussing the use of reverse-mode instead of forward-mode.

3.6.1 Univariate Taylor Series

In this chapter we have looked at how to calculate the multivariate Taylor coefficients of different mathematical functions, which is achieved by calculating all the multivariate Taylor coefficients for each mathematical operation along the way. Consider calculating the expression $f\left[g\left(\vec{x}\right)\right]$. We first calculate the Taylor coeffients for $g\left(\vec{x}\right)$ before using these to calculate the Taylor coefficients of the full expression $f\left[g\left(\vec{x}\right)\right]$.

A different approach, which is presented in [3] is to instead calculate a series of univariate Taylor series in different directions for all intermediate calculations. By choosing an appropriate set of directions, these univariate Taylor coefficients can be used to construct the multivariate Taylor coefficents. Going back to the calculation of $f\left[g\left(\vec{x}\right)\right]$, this would be performed by first calculating the univariate Taylor coefficients of $g\left(\vec{x}\right)$ before moving on to using this to calculate the univariate Taylor coefficients of the full expression $f\left[g\left(\vec{x}\right)\right]$. With this accomplished we could use the univariate Taylor coefficients of the full expression to construct the multivariate Taylor coefficients.

Compared to the approach we are using this results in approximately the same complexity for calculating derivatives of order ≤ 5, but with a large reduction in computational effort for larger orders of derivatives [3, p. 5]. One drawback of this method is that although it can be faster, the process of calculating the multivariate Taylor coefficients from the univariate ones results in larger memory usage compared to the method described in this chapter [7, p. 1]. Neidinger improved on the univariate Taylor coefficient method in [7] by eliminating this extra memory usage. We can therefore conclude by stating that although this method is more difficult to implement, it is a definitely a more efficient option for cases where we are dealing with derivatives of a high order.

3.6.2 Reverse-mode

Another alternative that can be implemented regardless of whether we use multivariate or univariate Taylor series, is to use the so called reverse-mode. This is an alternative to forward-mode, which is what is being used for this thesis. The forward-mode has not been described explicity since it is the most natural method to use in a programming environment, but when describing the difference between forward and reverse-mode it is convenient to first look at the forward-mode and explain how the reverse-mode is different. Assume that we set the variables x,y and z to the values a, b and c, respectively. Now assume that we want to calculate the function $h\left(x, y, z\right) = \left(x * z\right) * \sin\left(x * y\right)$, and consider the graph below, which was presented in [8, p. 14]:

$$\begin{tikzcd}
 & & & h = u * w & & & \\
 & & & & & w = \sin(v) \arrow[ull] & \\
 & u = x * z \arrow[uurr] & & & & v = x * y \arrow[u] & \\
z = c \arrow[ur] & & & x = a \arrow[ull] \arrow[urr] & & & y = b \arrow[ul]
\end{tikzcd}$$

When calculating $h(x, y, z)$ using forward-mode we start at the lowest level of the graph, and calculate the value and Taylor coefficients for each level moving upwards, which culminates with the calculation of $h(x, y, z)$. The reverse-mode starts at at the bottom before moving upwards as well, but instead of calculating all the Taylor coefficients it instead calculates the Taylor coefficients with respect to the immediate arguments. This is called the forward pass, and is followed by the reverse pass where all the Taylor coefficients of $h(x, y, z)$ are constructed using the values calculated in the forward pass.

This means that for the evaluation of w, during the forward pass, we calculate the Taylor series coefficients with regard to v, while for the evaluation of h we calculate the Taylor series coefficients with regard to u and w. The result of doing this is that we are calculating fewer coefficients for each node in the graph compared to normal mode. Assuming that x,y and z are only 3 out of a large amount of variables, the forward-mode would result in the calculation of a lot of coefficients that would simply equal 0, while the number of coefficients calculated in the forward pass of the reverse-mode would stay the same. This explains why the reverse-mode is a more efficient option than forward-mode for a large number of variables [8, p. 15].

Chapter 4

Implementation of an AD framework in Python

4.1 Introduction

To implement the ideas presented in the last chapter in a programming environment we need to be able to store the value and Taylor coefficients for an expression in a way that makes it easy to access and simple to perform operations with. The best match for doing this is to use object-oriented programming, and create a class for holding the necessary values and methods. Each expression will then be represented by an instance of this class. The properties of the class are listed below.

```python
class AD:
    """
    Properties :
    val          - Expression value
    T            - Array of taylor coefficients of expression
    num_vars     - Number of variables
    max_o        - Maximum order of derivatives
    ADvector     - True if the AD class holds more than one variable
    N            - Number of variables
    dtype        - Data type to store coefficients with.
                   (Default=complex)
    sparse       - Whether or not to store Taylor coefficients in
                   a sparse array (Default=False)
    dims         - The dimensions of a matrix with a length of
                   'max_o+1' along 'num_vars' dimensions
    sz           - The size of the above mentioned matrix
    counter_map  - Array mapping the index of T to the
                   corresponding derivative
    index_map    - Dictionary mapping a derivative to
                   the corresponding index in T
    """
```

The variables listed under properties represent the data that is stored with each instance of the AD class. The variable 'val' holds the value of the expression, 'T' holds an array with all the Taylor coefficients, 'num_vars' holds the total number of variables that we are working with, while 'max_o' is the maximum order of the derivatives.

The 'ADvector' variable is set to true if the AD instance holds more than one variable, with the number of variables being stored in 'N'. This is explained more in-depth in section 4.3. The variable 'dtype' holds the data type being used to store the Taylor coefficients. By default this is set to complex to cover all use-cases, but if a certain application only requires real numbers this can be set to float to improve performance. The 'dims' and 'sz' variables hold the dimension and size of the matrix of Taylor coefficients as described in the previous chapter. Finally, the index_map and counter_map variables are used to keep track of where the derivatives are stored, something that will be explained in more depth in section 4.2.

While it is important to store all the necessary values in the AD class, something just as important is being able to perform mathematical operations with instances of the AD class. This is done by defining specific methods in the class definition that are called when the corresponding mathematical operation is applied to an instance of the class. As an example assume that v holds an instance of the AD class and we try to calculate $v + 3$. This will result in the __add__ method in the v instance being called to handle this operation. The process of defining the __add__ method, or the corresponding method for other mathematical operations, is called operator overloading. A list of all these methods and what operation they overload follows below.

```
    # Overload negation (-self)
    def __neg__(self):
        pass

    # Overload positive (+self)
    def __pos__(self):
        pass

    # Overload addition (self+v)
    def __add__(self,v):
        pass

    # Overload right-sided addition (v+self)
    def __radd__(self,v):
        pass

    # Overload subtraction (self-v)
    def __sub__(self,v):
        pass

    # Overload right-sided subtraction (v-self).
    def __rsub__(self,v):
        pass

    # Overload multiplication operator (self*v)
    def __mul__(self,v):
        pass

    #  Overload right-handed multiplication operator (v*self)
    def __rmul__(self,v):
        pass

    # Overload division operator (self/v)
    def __div__(self,v):
        pass

    # Overload right-sided division (v/self)
    def __rdiv__(self,v):
        pass

    # Overload power operator (self^v)
    def __pow__(self,v):
        pass

    # Overload right-sided power operator (v^self)
    def __rpow__(self,v):
        pass
```

The AD class includes several other methods, as well as the ones above, the most important of which is the constructor. This is the method that is called when we create a new AD object, and it ensures that all the properties of the class are set correctly. The arguments passed to the constructor are all listed below.

```
def __init__(self,val,var_num,num_vars,max_o,\
             model=None,T=None,dtype=complex,sparse=False):
    """
    Arguments :
    val       - Expression value
    var_num   - Number of variable being initialized
    num_vars  - Maximum number of variables
    max_o     - Maximum order of derivatives
    model     - Another AD object with the same 'num_vars' and
                'max_o'. This is used to avoid creating the mapping
                array and dictionary more than once.
    T         - Array of Taylor coefficients.
                By default this is generated.
    dtype     - Data type used to store Taylor coefficients.
                By default complex to account for all ranges
                of values. For max_o>1 it is necessary to use
                the complex dtype to avoid errors.
    sparse    - If true, stores the Taylor coefficients
                as a sparse matrix.
    """
```

In addition to the variables mentioned already the constructor requires the argument 'var_num'. When creating a variable this represents the number of the variable that we are initializing and is necessary to differentiate between different variables. Calling the constructor will by default simply initialize a variable of the form x, which means that the Taylor coefficients will only be different from 0 for the Taylor coefficients corresponding to $\frac{\partial}{\partial x}$ and the expression's value. The number of the variable can be any value in $\{1,\ldots,num\_vars\}$. If set to 0 only the Taylor coefficient corresponding to the value will be set, meaning that it corresponds to initializing a constant.

The mapping variables counter_map and index_map are the same for AD instances with the same values for max_o and num_vars. Constructing these mapping variables are time consuming tasks, so it is desirable to avoid creating them many times over. Supplying an AD instance as the model argument to the constructor allows the new instance to use the mapping variables of the model instance. This is done for every calculation ensuring the mapping variables are created only once. The reason why these are necessary is explained in section 4.2.

It is also possible to include an optional argument T, which is used when doing calculations where you calculate the new T array. Instead of initializing a new array and then setting all the values, this simply sets the T variable of the object equal to the argument when included. When the T argument is included the value of the 'var_num' argument is ignored.

Finally it should be noted that there is another optional argument, sparse, which when set to true will store the T array as a sparse matrix. This can give significant performance improvements if there is a large amount of variables and the expressions only include a few of them. This is because a large amount of the values in the T matrix will simply be equal to zero, and using a sparse matrix avoids performing operations on these elements. This is only implemented for when the maximum order of the Taylor coefficients equals one.

4.2 A more efficient way of storing Taylor coefficients

Up until now we have assumed that the Taylor coefficients are stored in a multidimensional array with a number of dimensions equal to the number of variables, and a length of $max + 1$ for each dimension. As previously mentioned this has the benefit of allowing us to access the Taylor coefficient $T_{u,\vec{k}}$ with the multi-index $\vec{k}$ in the array. Although this is very convenient it results in a lot of unused elements in the array. The amount of Taylor coefficients, N, equal the amount of derivatives plus one, corresponding to the value of the expression. Let the number of variables be denoted by n, giving the following expression for N.

$$N = 1 + \sum_{k=1}^{max} \left(\begin{array}{c} n+k-1 \\ k \end{array} \right)$$

The amount of values we are storing, however, equal $(max+1)^n$, which is a far larger number. A different approach is to simply store the Taylor coefficients in an array of length N. This introduces the problem of how to access a given Taylor coefficient $T_{u,\vec{k}}$, and how to find out which Taylor coefficient is stored at a given index of the array. To solve this we iterate over all the Taylor coefficients and assign each of them a place in the array. The location of each Taylor coefficient, and what Taylor coefficient is located at a given place in the array are stored in two separate additional arrays.

For the special case where only the first order derivatives are being calculated the Taylor coefficients are simply equal to the first order derivatives and it is possible to store them in a more intuitive way than in the general case. We can just store the derivatives according to the number of the variable we are differentiating. We are in other words storing $\frac{\partial f}{\partial x_1}$ first, $\frac{\partial f}{\partial x_2}$ second and so on. This leaves no ambiguity as to what derivative is located at a given index in the T array, or how to access a given derivative, so additional arrays to keep track of the derivatives are not needed. This saves computational cost both in terms of avoiding having to calculate the arrays and not having to perform the look-up to find out where a derivative is located.

4.3 Vectors of AD variables and expressions

Sometimes it is desirable to group a set of variables together in a single AD instance if the variables are related in some way that means that they will all be used to create similar expressions. This is illustrated in the following pseudocode:

```
# Seperate variables
x1 = AD(1)
x2 = AD(3)
x3 = AD(5)

# Calculate function values
fx1 = f(x1)
fx2 = f(x2)
fx3 = f(x3)

# Alternate approach
# ------------------------------
x = AD([1,2,3])
fx = f(x)
```

Clearly the latter approach is much simpler, since it enables us to perform the necessary mathematical operation by writing far less code. One reason for wanting to do this is simply because we want to calculate the derivatives of an expression for different values of x, as demonstrated in the pseudo-code below:

```
# Initialize variables
y = AD(3)
x = AD([1,2,3])

# Calculate expressions
f = lambda x,y : a*exp(pi*x*y)
f_xy = f(x,y)

# Get individual expressions
f_xy1 = f_xy[0] # Holds f(x,y) for x=1
f_xy2 = f_xy[1] # Holds f(x,y) for x=2
f_xy3 = f_xy[2] # Holds f(x,y) for x=3
```

In this example f_{xy1}, f_{xy2} and f_{xy3} hold the value and derivatives of $f(x,y)$ for the three different x values. It is important to note that in this example the total number of variables is 2, with the expression being evaluated at different x values. This is different from the scenario where the AD object instead of holding different values for the same variable, instead holds several distinct variables. Assume that we want to calculate the following three mathematical expressions, using the 4 variables x_1,x_2,x_3 and y:

$$\begin{gathered} f(x_1,y) \\ f(x_2,y) - g(x_1,y) \\ f(x_3,y) \end{gathered}$$

The pseudo code below shows how this could be accomplished.

```
# Initialize variables
y = AD(3)
x = AD([1,2,3],distinct=True)

# Calculate expressions
f_xy = f(x,y)

# Get individual expressions
expr1 = f_xy[0]
expr2 = f_xy[1] - g(x[0],y)
expr3 = f_xy[2]
```

Where the distinct argument in this case indicates that the x AD instance holds a vector of distinct variables instead of multiple values for the same variable. The functionality above could be used if we are looking at the pressure in an oil reservoir, in which case the pressure in different areas is described by separate variables, but where they are all stored together in the same AD instance. To use this functionality in the actual implementation the val and var_num arguments have to be provided as arrays of a length equal to the number of variables, where val holds the values and var_num holds the numbers indicating the number of each variable.

If we simply want to create several expressions for different values of a variable, we will let each variable have the same value for var_num. This indicates that the AD instance doesn't hold several distinct variables, but instead several different values for one variable. If on the other hand we are dealing with distinct attributes such as pressure at different points, and we need to create expressions where the variables interact with one another, it is necessary to use a seperate var_num value for each variable.

4.4 Initialization and Usage

When initializing variables it is possible to either do this directly by creating instances of the AD class or by using the function 'init_variables'. In general using this function is more convenient than creating the instances directly. It takes the values of the variables being initialized as arguments, in addition to four optional arguments.

- max_o : Maximum order of derivatives, defaults to 1
- dtype : Data type of values, defaults to complex
- sparse : Boolean determining whether to use sparse matrices, defaults to False
- update_num : Boolean affecting the numbering av variables, defaults to False

The optional arguments max_o, dtype and sparse are simply passed along to the AD constructor when initializing the variables. The final optional argument update_num determines whether or not we are using seperate var_num values for any AD vectors we are initializing.

Example usage of this function and calculating an expression follows below:

```
# Initialize variables
x,y,z = init_variables(0,[1,2,3],8,max_o=2)

# Calculate mathematical expression
g=exp(-x*y)*sin(2*pi*z)

# Get single AD objects
expr1 = g[0]    # Holds AD object of expression for x=0,y=1,z=8
expr2 = g[1]    # Holds AD object of expression for x=0,y=2,z=8
expr3 = g[2]    # Holds AD object of expression for x=0,y=3,z=8
```

This starts by initializing the variables where $x = 0$, y is being evaluated for the three values $1, 2, 3$ and $z = 8$. It also sets the maximum order of the Taylor coefficients to 2 meaning that we will be able to extract derivatives up to the second order. Next the code calculates the expression $e^{-xy} \sin(2\pi z)$ and stores it in the g variable. The result is that g holds an AD vector object that includes the value and Taylor coefficients of the expression for the three different values of y. To get a single AD object corresponding to a given set of values, g can be indexed as is shown in the code. The next piece of code shows how specific derivatives or Taylor coefficients can be accessed.

```
# Print derivatives of expression for x=0,y=1,z=8
print expr1.get_derivative([0,2,0]) # Derivative g_yy
print expr1.get_derivatve([1,0,1])  # Derivative g_xz

# Print Taylor coefficients for x=0,y=3,z=8
print expr3.get_tcof([1,0,0]) # Taylor cof. g_x
print expr3.get_tcof([0,0,2]) # Taylor cof. g_zz
```

This shows the usage of the functions get_derivative and get_tcof, which returns a single derivative or Taylor coefficient value respectively. The same can be achieved by working directly with the g object, and the code below will give the same result as above.

```
# Print derivatives of expression for x=0,y=1,z=8
print g.get_derivative([0,2,0],var=0) # Derivative g_yy
print g.get_derivatve([1,0,1],var=0)  # Derivative g_xz

# Print Taylor coefficients for x=0,y=3,z=8
print g.get_tcof([1,0,0],var=2) # Taylor cof. g_x
print g.get_tcof([0,0,2],var=2) # Taylor cof. g_zz
```

Assuming that we need to deal with a large amount of derivatives it can be desirable to have an easy way of retrieving derivatives stored in a certain way. Assuming we have an AD vector object f, we can use the following functions to get the Jacobian and Hessian matrices.

```
# Print Jacobian
print f.get_jacobian()

# Print Hessian
print f.get_hessian()
```

These functions are used in the application part of the thesis, where we need the Jacobian and Hessian matrices to solve certain problems.

4.5 Computing weighted sums of Taylor coefficient products

Many of the mathematical expressions derived for the Taylor coefficients of the different functions and mathematical operators require the calculation of sums of products of Taylor coefficients. It is therefore desirable to create a function, bdot, that manages to calculate these sums in an efficient manner, making it easier to implement the mathematical functions and operators themselves. Neidinger presents such a function in [6, p. 5] for an AD framework that uses derivatives, and briefly notes how this can be done for Taylor coefficients later in the same article [6, p. 14]. In this section we will go step-wise through the process of determining how to create this function. As the first step we can make an initial guess by creating a function that calculates the following sum:

$$\sum_{\vec{j}=\vec{0}}^{\vec{m}} T_{P,\vec{j}} T_{Q,\vec{k}-\vec{j}}$$

Setting $\vec{m} = \vec{k}$, $P = f$ and $Q = g$ would result in the calculation of the Taylor coefficient $T_{fg,\vec{k}}$. This shows that the function could be used to calculate the Taylor coefficients for multiplication, however for other expressions such as exponentiation this would fall short since it requires a weighting factor multiplied with the product of Taylor coefficients. We can expand on our initial guess and add an optional weighting factor, yielding the following expression:

$$\sum_{\vec{j}=\vec{0}}^{\vec{m}} c_{\vec{k},\vec{j},\vec{e}} T_{P,\vec{j}} T_{Q,\vec{k}-\vec{j}} \tag{4.1}$$

The $c_{\vec{k},\vec{j},\vec{e}}$ term above is denoted by the following expression:

$$c_{\vec{k},\vec{j},\vec{e}} = \begin{cases} 1, & \vec{e} = \vec{0} \\ \frac{k_i - j_i}{k_i}, & |e| = 1 \end{cases}$$

In the expression above the vector $\vec{e}$ is a one-order vector and i is the index where $e_i = 1$. By setting $\vec{e} = 0$ this weighting factor is set to 1, meaning we can still use this to calculate the Taylor coefficients for multiplication. If we instead set $\vec{m} = \vec{k} - \vec{e}$, $P = h$ and $Q = u$ for any one-order vector $\vec{e} \leq \vec{k}$ into equation 4.1 we get the following expression:

$$\sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}} T_{u,\vec{k}-\vec{j}}$$

This is the calculation of the Taylor coefficient $T_{h,\vec{k}}$, where $h = e^u$, showing that this can be used to calculate Taylor coefficients of both multiplication and exponentiation. In the following sections we will look at how this can be used to calculate the Taylor coefficients for other mathematical functions and operators as well. In pseudo code an implementation of this, called bdot, looks as follows:

```
def bdot(P,m,Q,k,e):
    # Get object with all multi-indeces less than or equal to m
    iter = get_all_lte(k)

    # Initialize Taylor coefficient
    T_coef = 0

    # Iterate over multi-indices
    for j in iter:
        # Calculate term under sum
        term = calculate_c(k,j,e)*\
               P.get_Tcoef(j)*\
               Q.get_Tcoef(k-j)

        # Add to total
        T_coef += term

    # Return final Taylor coefficient value
    return T_coef
```

The actual implementation of bdot is implemented as a static function of the AD class. It is a bit longer, but the general outline of the above code shows how it works. However, one important difference is that if either P or Q are ADvector objects, the result of any mathematical operation involving them is an ADvector as well. In those cases the return value is not a single Taylor coefficient, but a column vector describing the same Taylor coefficient for different values. In the following sections we will look at the implementation of mathematical operations on the AD object, some of which require use of the bdot function.

4.6 Overloaded operators

This section will show and explain the implementation of some of the overloaded operators in the AD class.

4.6.1 Addition

To overload the addition operator we need to define the __add__ method in the AD class. Assume that we have an AD instance u and add another Python object to it by calculating $u + v$. The pseudo code below shows how this is calculated when v is a numeric object:

```
# Calculate value
val = u.val+v

# Calculate Taylor coefficients
T   = u.T

# Set T[0] or T[0,:]
if max_o != 1:
    if u.ADvector:
        T[0,:] = val
    else:
        T[0]   = val

# Return AD object
return AD(val,T)
```

The value will simply be the value of v added to u, while the Taylor coefficients of order higher or equal to 1 remain unchanged. In the scenario where we are considering derivatives of order higher than 1 we need to set the elements in the T array corresponding to the values as well. For an ADvector object this means setting a column of values, while for a single variable AD object it only means setting one. If v is an instance of the AD class as well the code will instead look like the pseudo code below:

```
# Calculate value
val = u.val+v.val

# Calculate Taylor coefficients
T   = u.T+v.T

# Return AD object
return AD(val,T)
```

The value equals the sum of the values of the AD instances, while the Taylor coefficients are simply added together as well. If the maximum order is larger than one the Taylor coefficients corresponding to the value need to be updated as well, but this is done automatically since the Taylor coefficient arrays are added together. To implement subtraction we can follow the exact same outline as above.

4.6.2 Multiplication

To overload the multiplication operator we need to define the __mult__ method in the AD class. Assume that we have an AD instance u and multiply another Python object with it by calculating $u * v$. The pseudo code below shows how this is calculated when v is a numeric object:

```
# Calculate value
val = u.val*v

# Calculate Taylor coefficients
T   = u.T*v

# Return AD object
return AD(val,T)
```

The value is just the value of u multiplied by v, while the Taylor coefficients are all multiplied by v as well. The Taylor coefficients corresponding to the value are set correctly by this operation as well, since all we need to do is multiply by v. Now consider the case when v is an AD object, and the maximum order of derivatives is set to 1. Instead of using the bdot function we will in this case use the following expression:

$$\frac{\partial}{\partial x_i}(uv) = \frac{\partial u}{\partial x_i}v + v\frac{\partial v}{\partial x_i}$$

The pseudo code below illustrates how this is implemented:

```
# Calculate value
val = u.val*v.val

# Calculate derivatives
T   = u.T*v.val + u.val*v.T

# Return AD object
return AD(val,T)
```

When the maximum order is set to one the T array only stores the derivatives, so in this case it is not necessary to set any of the elements in T corresponding to the value of the expression. Finally let us consider the case when u and v are both AD objects and the maximum order is set to more than 1. In this case we need to use the bdot function. Recall that to calculate $T_{uv,\vec{k}}$ we need to set $\vec{m} = \vec{k}$, $P = u$, $Q = v$ and $e = 0$. The pseudo code for this follows below:

```
# Calculate value
val = u.val*v.val

# Create AD object with the value set, but with all Taylor
# coefficents of order larger than 0 set to 0
h   = create_empty_AD(val)

# Get all multi-indices
iter = get_all_indices()

# Calculate terms
```

```python
for k in iter:
    if u.ADvector or v.ADvector:
        # Return value is a column
        h.T[:,k] = bdot(u,k,v,k,0)
    else:
        # Return value is a scalar
        h.T[k] = bdot(u,k,v,k,0)

# Return AD object
return h
```

It is important to note that if u or v is an ADvector, then so is h. In that case the return value is a column vector that should set the column of Taylor coefficients corresponding to $\vec{k}$. This is handled by the if test in the loop over the indices.

4.6.3 Division

To overload the division operator we need to define the __div__ method in the AD class. Assume that we have an AD instance u and divide it with another Python object by calculating u/v. The pseudo code below shows how this is calculated when v is a numeric object:

```python
# Calculate value
val = u.val/v

# Calculate Taylor coefficients
T    = u.T/v

# Return AD object
return AD(val,T)
```

Both the value and Taylor coefficients are divided by v. It is not necessary to make any further changes to the coefficients corresponding to the value in the T array, since they are divided by v as well. Now consider the case when v is an AD object, and the maximum order of derivatives is set to 1. Instead of using the bdot function we will in this case use the following expression:

$$\frac{\partial}{\partial x_i}\left(uv^{-1}\right) = \frac{\partial u}{\partial x_i}v^{-1} - uv^{-2}\frac{\partial v}{\partial x_i}$$

The pseudo code implementing this is shown below.:

```python
# Calculate value
val = u.val/v.val

# Calculate derivatives
T    = u.T/v.val - u.val*v.T/v.val**2

# Return AD object
return AD(val,T)
```

Since only the derivatives are stored in T when the maximum order is set to one, no modification of the elements in the T array is necessary. Finally let us consider the case when u and v are both AD objects and the maximum order is set to more than 1. In this case we need to use the bdot function. Recall that the mathematical formula (3.8) derived for division, where $h = u/v$, was given as follows:

$$T_{h,\vec{k}} = \frac{T_{u,\vec{k}} - \sum_{\vec{j}<\vec{k}} T_{h,\vec{j}} T_{v,\vec{k}-\vec{j}}}{v\left(\vec{x}\right)}$$

We could try to calculate this using the following expression:

$$T_{h,\vec{k}} \equiv \frac{T_{u,\vec{k}} - bdot\left(h,k,v,k,0\right)}{v\left(\vec{x}\right)}$$

The bdot function call would in this case calculate the following expression:

$$bdot\left(h,k,v,k,0\right) = \sum_{\vec{j}=\vec{0}}^{\vec{k}} T_{h,\vec{j}} T_{v,\vec{k}-\vec{j}}$$

This is different from the mathematical expression in that it allows the value $\vec{j} = \vec{k}$, which leads to an extra term including the value $T_{h,\vec{k}}$, which we are calculating. Though the expressions are not equal, it still leads to the correct value. This is because initially all the Taylor coefficients that are yet to be calculated are set to 0, so this term simply vanishes. Using this the pseudo code looks as follows:

```
# Calculate value
val = u.val/v.val

# Create AD object with the value set, but with all Taylor
# coefficents of order larger than 0 set to 0
h   = create_empty_AD(val)

# Get all multi-indices
iter = get_all_indices()

# Calculate terms
for k in iter:
    if u.ADvector or v.ADvector:
        # Return value is a column
        if u.ADvector:
            h.T[:,k] = (u.T[:,k]-bdot(u,k,v,k,0)) / v.val
        else:
            h.T[k] = (u.T[k]-bdot(u,k,v,k,0)) / v.val
    else:
        # Return value is a scalar
        h.T[k] = bdot(u,k,v,k,e)

# Return object
return h
```

This follows the same outline as shown for multiplication, with the only significant difference being the expressions for $T_{h,\vec{k}}$ being different. It is also important to note that for this formula it is necessary to calculate the derivatives in a certain order, since to calculate $T_{h,\vec{k}}$ we need to access all

$\vec{j} \leq \vec{k}$, except $\vec{k}$ itself. In the actual implementation this is done by always calculating all Taylor coefficients for $\vec{j} \leq \vec{k}$, except $\vec{k}$ itself, before calculating $T_{h,\vec{k}}$. In the pseudo code above we can assume that the iter object ensures this order is followed.

4.6.4 Additional overloaded operators

The remaining operators that have not been described follow the same structure as the ones described in detail above. The exception is the power function which uses the fact that $x^y = e^{y*log(x)}$. This is calculated by using the exponential and log function implemented for the AD class, which are covered in the next section. The code for all of these functions can be found in the appendix.

4.7 Mathematical functions defined for the AD class

So far we have gone over the implementation of overloaded functions for basic mathematical operations. To be able to apply mathematical functions to instances of the AD class it is necessary to create Python functions that accept these as arguments and calculate the mathematical expressions derived previously. Pseudo code will be included and explained for the exponential function, while the remainder will simply focus on how the bdot function is used to calculate the mathematical expressions for the Taylor coefficients when the maximum order of derivatives is larger than one. The functions that will be explained are:

- Exponential function
- Natural logarithm
- Square Root
- Inverse trigonometric function
- Trigonometric functions

The functions are included in the separate library mathADI, and they can all be found in the appendix. We will go through the implementation of these functions in the next subsections.

4.7.1 Exponential function

The exponential function for the AD class is defined in the exp function in the mathADi library. Assume that u holds an AD instance, and that we make the function call $exp\,(u)$. If the maximum order is set to 1, we use the following expression to calculate the new AD instance:

$$\frac{\partial}{\partial x_i} e^u = e^u \frac{\partial u}{\partial x_i}$$

The pseudo code for doing this follows below:

```
# Calculate value using regular exp function
val = np.exp(u.val)

# Calculate derivatives
T   = val*u.T

# Return new AD object
return AD(val,T)
```

Since the maximum order is set to 1, the T array only holds the derivatives so we do not need to set any coefficients corresponding to the value. Recall that in section 4.5 we found that if we set $\vec{m} = \vec{k} - \vec{e}$, $P = h$ and $Q = u$ for any one-order vector $\vec{e} \leq \vec{k}$ into equation 4.1 we get the expression for $T_{h,\vec{k}}$, where $h = e^u$. We can in other words calculate $T_{h,\vec{k}}$ using the function call $bdot\,(h, k-e, u, k, e)$. The pseudo code showing this follows below.

```
# Calculate value with regular exp function
val = np.exp(u.val)

# Create AD object with the value set, but with all Taylor
# coefficents of order larger than 0 set to 0
h   = create_empty_AD(val)

# Get all multi-indices
iter = get_all_indices()

# Calculate terms
for k in iter:
    # Get one-order vector <= k
    e = get_one_order_lte(k)

    if u.ADvector:
        # Return value is a column
        h.T[:,k] = bdot(h,k-e,u,k,e)
    else:
        # Return value is a scalar
        h.T[k] = bdot(h,k-e,u,k,e)

# Return object
return h
```

The only major change from the previous examples of using the bdot function, is that we need to find a one-order vector $\vec{e} \leq \vec{k}$ for every Taylor coefficient $T_{h,\vec{k}}$ we calculate. And just like for division, we need to ensure that all Taylor coefficients of a lower order have been calculated before calculating $T_{h,\vec{k}}$.

4.7.2 Natural logarithm

Recall that the formula (3.7) for the Taylor coefficients of $h = \ln(u)$ were given as:

$$T_{h,\vec{k}} = \left(T_{u,\vec{k}} - \sum_{\substack{\vec{j}>\vec{0} \\ \vec{j}\leq\vec{k}-\vec{e}}} \frac{k_i - j_i}{k_i} T_{u,\vec{j}}\, T_{h,\vec{k}-\vec{j}} \right) / u\left(\vec{x}\right)$$

Using the bdot function this can be calculated as:

$$T_{h,\vec{k}} = \left(T_{u,\vec{k}} - bdot\left(u, k-e, h, k, e\right) \right) / u\left(\vec{x}\right)$$

This allows $\vec{j} = \vec{0}$ in the sum, which adds a term including $T_{h,\vec{k}}$. Since this is initially set to 0 in the programming environment it disappears and we are left with the correct result.

4.7.3 Square Root

Recall that the formula (3.10) for the Taylor coefficients of $h = \sqrt{u}$ were given as:

$$T_{h,\vec{k}} = \left(T_{u,\vec{k}} - 2 \sum_{\substack{\vec{j}>\vec{0} \\ \vec{j}\leq\vec{k}-\vec{e}}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{h,\vec{k}-\vec{j}} \right) / 2h\left(\left(\vec{x}\right)\right.$$

Using the bdot function this can be written as:

$$T_{h,\vec{k}} = \left(\frac{1}{2} T_{u,\vec{k}} - bdot\left(h, k-e, h, k, e\right) \right) / h\left(\left(\vec{x}\right)\right.$$

Like with the natural logarithm this allows $\vec{j} = \vec{0}$ in the sum, which adds a term including $T_{h,\vec{k}}$. This disappears and we get the correct result.

4.7.4 Inverse Trigonometric Functions

Recall that for inverse trigonometric functions the formula (3.11) for the Taylor coefficients of $h = inv\left(u\right)$, where inv can be either arccos, arcsin or arctan, was given as:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{v,\vec{j}}\, T_{u,\vec{k}-\vec{j}}$$

This can be calculated using the bdot function as:

$$T_{h,\vec{k}} = bdot\left(v, k-e, u, k, e\right)$$

Where v is given as:

- arccos : $v = -\frac{1}{\sqrt{1-u^2}}$
- arcsin : $v = \frac{1}{\sqrt{1-u^2}}$
- arctan : $v = \frac{1}{1+u^2}$

4.7.5 Trigonometric Functions

Assume that $h = \sin(u)$ and $g = \cos(u)$, and recall that the formulas for sin (3.12) and cos (3.13) were given as follows:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{g,\vec{j}}\, T_{u,\vec{k}-\vec{j}}$$
$$T_{g,\vec{k}} = -\sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{u,\vec{k}-\vec{j}}$$

Using the bdot function we can calculate these Taylor coefficients as:

$$T_{h,\vec{k}} = bdot\,(g, k-e, u, k, e)$$
$$T_{g,\vec{k}} = -bdot\,(h, k-e, u, k, e)$$

Note that these functions must be calculated simultaneously since to calculate the Taylor coefficient corresponding to $\vec{k}$ it is necessary to know the values of all lower Taylor coefficients for both the sin and cos expression.

All other trigonometric functions can be created by using sin and cos, which is how the automatic differentiation framework calculates the Taylor coefficients for these functions. To calculate $tan\,(u)$ the framework simply calculates $h = \sin(u)$ and $g = \cos(u)$ initially, and then calculates $tan\,(u)$ as h/g. The remaining trigonometric functions are calculated using this process as well.

Part II

Applications

Chapter 5

Optimization

5.1 Introduction

Optimization Theory is an area of mathematics that was previously known as Operations Research, which covers many different areas of minimization and optimization [14]. The goal of Optimization Theory is, as the name implies, optimization. What is meant by this is to find the best possible way of doing something under certain conditions. In general it involves expressing a certain quantity as a function that is either maximized or minimized. The function can express things such as profit or product quality in the case of maximization, or things like cost or loss in the case of minimization. It is also common to consider different constraints during the maximization or minimization [13]. For instance if we are trying to maximize the future profits of a company by deciding what areas to invest in, it would be necessary to include the available capital as a constraint on the total amount to be invested.

In this chapter we will not use constraints, but rather focus on simpler examples. The simplest possible example of optimization theory is to find the maxima and minima of a given function. This is closely related to finding zeros of a function, as will be shown in section 5.2. In this section we will go through how zeros, minima and maxima can be found for single variable functions using an approach based on Newton's method. In section 5.3 we will go through how maxima and minima can be found for functions of multiple variables, using two different approaches. Common to all of these examples is the necessity to calculate the derivatives of mathematical expressions, which is what makes the AD framework a useful tool for solving these problems.

5.2 Single variable optimization

For optimization problems involving only a single variable we can generally find the exact solutions, without having to resort to numerical approximations. However, it can still be useful to consider the single variable cases since they show the general idea used in the multivariate cases. In this section we will look at how the AD framework can be used together with Newton's method to approximate roots, maxima and minima of a given function.

5.2.1 Newton's method

Assume that we want to find the zeros of a given expression $g(x)$. Assume that we make an initial guess x_0, and then proceed to Taylor expand g around x_0 to the first order.

$$g(x) \approx g(x_0) + g'(x_0)(x - x_0) = 0$$

If we solve for x, calling it x_1, we get the following result.

$$x_1 = x_0 - \frac{g(x_0)}{g'(x_0)}$$

This process can be repeated, where we Taylor expand around x_1 and solve for x_2 and so on. This is called Newton's method, and the general formula can be written as follows [12]:

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)} \tag{5.1}$$

The assumption here is that by repeatedly solving for x_{n+1}, the value will eventually converge to a root of g. In a programming environment we set a tolerance level where for each iteration we check to see if the value $g(x_{n+1})$ is sufficiently close to zero according to the given tolerance. When this is achieved the value of x_{n+1} is returned. If, on the other hand, the value does not come sufficiently close in a certain amount of iterations an error is thrown instead.

5.2.2 Finding maxima and minima

The maxima and minima of a function are characterized by points where the derivative equals zero. This means that finding the maxima and minima of a function $f(x)$ is equivalent to finding the roots of $g(x) = f'(x)$. Putting this into equation (5.1) for Newton's method yields the following iterative formula:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{5.2}$$

This shows that to find the maxima and minima of a function we can use the same approach as for locating its roots, with the only difference being that we use equation (5.2) instead of equation (5.1) to find x_{n+1}.

5.2.3 Implementation : Newton's method

Let the function $g(x)$ be described as follows:

$$g(x) = xe^x - x^2$$

This function has a single root at $x = 0$. The initial code for finding the root of g follows below:

```
# Define function g
g = lambda x: x*exp(x) - x**2

# Initial guess for root
val = 7
x_r = init_variables(val,max_o=1)

# Current value
current = g(x_r)
```

We start off by initializing the function g, as well as initializing the AD variable x_r, which holds the approximations of the root. Initially this is set to $x_0 = 7$. We then calculate the value of the expression for the current value of x_r. The next piece of code follows below:

```
# Iteration variables
n = 1        # Iteration counter
N = 100      # Maximum number of iterations
tol = 1E-2   # Tolerance
```

Here we initialize variables necessary before starting the main loop where we approximate the root of the function. The variable N sets the maximum number of iterations we do before terminating the program, while the *tol* variable determines how close we need to get to the root. This means that the program terminates if we exceed N iterations, or if the absolute value of the expression $g(x_n)$ falls below the tolerance value. The main loop follows below:

```
# Find root
while n <= N and np.abs(current.val) > tol:
    # Get value and derivative
    val = current.val
    der = current.T[0]

    # Calculate new x_r
    x_r = x_r - (val/der)

    # Calculate new value
    current = g(x_r)

    # Increase iteration counter
    n += 1
```

For each iteration of the loop we store the value and derivative of $g(x_n)$ in the variables *val* and *der*, respectively. We then use equation (5.1) to calculate x_{n+1} before calculating $g(x_{n+1})$. Once the loop finishes we know that it succeeded as long as n is not larger than N.

5.2.4 Implementation : Finding maxima and minima

Let $f(x)$ be described as follows:

$$f(x) = 4x - x^2$$

This has a maximum at $x = 2$. The initial code for finding the maximum of f follows below:

```python
# Define function f
f = lambda x: 4*x-x**2

# Initial guess for root
val = 6.7
x_m = init_variables(val,max_o=2)

# Current value and derivatives
current = f(x_m)
f_x     = current.T[1]
f_xx    = 2*current.T[2]
```

We start off by initializing the function f, as well as initializing the AD variable x_m, which holds the approximations of the maximum. Initially this is set to $x_0 = 6.7$. Note that we need to set the maximum order to 2 in this case, since we require both the first and second order derivative. We then calculate the value of the expression for the current value of x_m and store the first and second order derivatives in the variables f_x and f_{xx}. The next piece of code follows below:

```python
# Iteration variables
n = 1        # Iteration counter
N = 100      # Maximum number of iterations
tol = 1E-2   # Tolerance
```

Just like for finding the roots we initialize a set of necessary variables before starting the main loop. Like before the value of N decides the maximum number of iterations, but the tolerance variable is used slightly differently. Since we are effectively using Newton's method to find the roots of $f'(x)$, we will terminate the main loop once we exceed the maximum number of iterations or when the absolute value of the first order derivative falls below the tolerance value. The main loop follows below.

```python
# Find maximum
while n <= N and np.abs(f_x) > tol:
    # Calculate new x_m
    x_m = x_m - (f_x/f_xx)

    # Calculate new value and derivatives
    current = f(x_m)
    f_x     = current.T[1]
    f_xx    = 2*current.T[2]

    # Increase iteration counter
    n += 1
```

This is very similar to the code for finding the root of a function, but we now use equation (5.2) to find x_{n+1}. Once this is done we calculate the expression $f(x_{n+1})$ and store the first and second order derivatives in f_x and f_{xx}. Just like for finding the roots we know that the method has succeeded as long as n is not larger than N after the loop has finished.

5.2.5 Analysis of results

Recall that the script for finding roots using Newton's method did this for the following function:

$$g(x) = xe^x - x^2$$

Running the script results in a root being found after 11 iterations of the method. Assume that we instead try to find the roots of the function following below:

$$f(x) = 4x - 10$$

In this case the root is found after only a single iteration with no error. Note that in deriving the iteration formula for Newton's method we approximated the function as a first order Taylor polynomial. Since f is a first order function the Taylor approximation is exact. As a result of this the root of f is found exactly in only one step. Now recall that the function we created a script to find the minimum for was given as follows:

$$f(x) = 4x - x^2$$

Running the script to find the minimum results in the exact point being found after only one iteration. This has an explanation similar to that of finding the root of a linear function. When we derived the iteration formula for finding the minimum we simply used Newton's method to find the root of $g(x) = f'(x)$. Since f is a second order polynomial we know that g is a first order polynomial, in which case Newton's method is exact. As a result of this the minimum is found after only a single iteration.

5.3 Multivariate optimization

In this section we will go through how to find minima and maxima for functions of several variables. We will start by going through the method of steepest descent, which uses an intuitive idea to find maxima and minima. We will then proceed to show how maxima and minima can be found using a scheme based on Newton's method for multi-dimensional functions.

5.3.1 Method of Steepest Descent

Assume that we have a function, $f(\vec{x})$, of more than one variable, and that we wish to find its minimum. The method of steepest descent starts with an initial guess $\vec{x}_0$ and and then repeatedly moves in a direction opposite to the local gradient. This is done as many times as is necessary until a minimum is found [11]. We are in other words finding the direction where the function decreases the most and following it. This is an iterative scheme that can be written as follows:

$$x_{i+1} = x_i - \epsilon \nabla f\left(\vec{x}_i\right) \tag{5.3}$$

This equation is used for some small $\epsilon > 0$. The term ϵ must be set to a small value less than 1 to prevent the method from overshooting and moving away from the minimum. It can be set to either a small constant, or it can change in value from each iteration to improve the convergence rate.

A similar method can be used to find the maximum points of a function. Note that the maximum points of $f\left(\vec{x}\right)$ are the minimum points of the function $g\left(\vec{x}\right) = -f\left(\vec{x}\right)$. To find the maximum points of f we can simply find the minima of g. An alternative to this is to follow the direction where the function increases most, instead of where it decreases. This yields the following calculation to find x_{i+1}:

$$x_{i+1} = x_i + \epsilon \nabla f\left(\vec{x}_i\right) \tag{5.4}$$

This can be implemented in a similar way to finding the minimum of a function, but with the terms for calculating the new spatial points replaced with the expression above.

5.3.2 Newton based method for finding extrema

Assume that we have a k-dimensional function of n variables, $\mathbf{F}\left(\vec{x}\right)$, and that we wish to find the points where it equals the k-dimensional zero vector. Like in the one-dimensional case with one variable we can approximate $\mathbf{F}$ with a first order Taylor approximation around a point $\vec{x}_0$.

$$\mathbf{F}\left(\vec{x}\right) \approx \mathbf{F}\left(\vec{x}_0\right) + \mathbf{J}_\mathbf{F}\left(\vec{x}_0\right)\left(\vec{x} - \vec{x}_0\right) = 0 \tag{5.5}$$

The term $\mathbf{J}_\mathbf{F}$ above is the Jacobian matrix of $\mathbf{F}$. Assuming that $\vec{x}_0$ is a guess to the zero-point, $\vec{x}_r$, of $\mathbf{F}$, we can improve on our guess by solving for $\vec{\delta}_0 = \vec{x} - \vec{x}_0$ and setting $\vec{x}_1$ as follows:

$$\vec{x}_1 = \vec{x}_0 + \vec{\delta}_0$$

By doing this iteratively we get increasingly more accurate approximations to $\vec{x}_r$. In general this can be written as follows:

$$\vec{x}_{i+1} = \vec{x}_i + \vec{\delta}_i \tag{5.6}$$

To solve for δ we need to solve equation (5.5). For this equation to have a unique solution the Jacobian matrix must be square. For this to be the case the number of dimensions, k, must equal the number of variables, n. When this is true we can rewrite the equation as follows:

$$\vec{\delta}_i = -\mathbf{J}_\mathbf{F}^{-1}\left(\vec{x}_i\right)\mathbf{F}\left(\vec{x}_i\right) \tag{5.7}$$

Now assume that we want to find the maxima and minima of a function of n variables, $g(\vec{x})$. This is the same as finding the point where all derivatives equal zero. This can be written as follows:

$$\nabla g(\vec{x}) = 0$$

We can now define $\mathbf{F}(\vec{x})$ as follows:

$$\mathbf{F}(\vec{x}) = \nabla g(\vec{x}) = 0$$

$\mathbf{F}$ is now an n-dimensional function of n variables, which means that we can use the multidimensional Newton method to find where it equals zero. Inserting the expression for $\mathbf{F}$ into equation (5.7) we get the following expression for $\vec{\delta_i}$:

$$\begin{aligned} \vec{\delta_i} &= -\mathbf{J_F}^{-1}(\vec{x}_i)\,\mathbf{F}(\vec{x}_i) \\ &= -\mathbf{J_F}^{-1}(\vec{x}_i)\,\nabla g(\vec{x}_i) \\ &= -\mathbf{H_g}^{-1}(\vec{x}_i)\,\nabla g(\vec{x}_i) \end{aligned} \tag{5.8}$$

Where $\mathbf{H_g}$ is the Hessian matrix for the function g, which is defined as follows:

$$\mathbf{H_g} = \begin{bmatrix} \frac{\partial^2 g}{\partial x_1^2} & \frac{\partial^2 g}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 g}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 g}{\partial x_2\,\partial x_1} & \frac{\partial^2 g}{\partial x_2^2} & \cdots & \frac{\partial^2 g}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 g}{\partial x_n\,\partial x_1} & \frac{\partial^2 g}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 g}{\partial x_n^2} \end{bmatrix}.$$

5.3.3 Implementation : Method of Steepest Descent

We will look at how we can create a general solver for finding the minimum of a function of two variables using the method of steepest descent. The code below shows the function definition and the required arguments:

```python
def solver(f,x0,y0,N=1000,tol=1E-2,eps=0.01):
    """
    f   - Function to find minimum of
    x0  - Initial guess for x
    y0  - Initial guess for y
    N   - Maximum number of iterations
    tol - Tolerance
    eps - Gradient multiplier
    """
```

This shows that we need to get a function f passed to the solver as an argument as well as the coordinates for the initial guess (x_0, y_0). The rest of the arguments are optional, with the tolerance in this case denoting how close the norm of the derivative vector must be to zero before deciding that a minimum has been found. The code that precedes the main loop follows below:

```
    # Initial guess for minimum
    x,y = init_variables(x0,y0,max_o=1)

    # Current value
    current = f(x,y)

    # Create list of approximations
    x_a = [x.val]
    y_a = [y.val]

    # Define norm
    norm = lambda grad: np.sqrt(np.sum(np.abs(grad)**2))

    # Initialize iteration counter
    n = 1
```

We start off by creating AD variables set to the arguments x_0, y_0, and follow this up by calculating the value of the function f for these points. We then create lists for holding the approximations to the minimum. At last we define a function for calculating the norm of the derivative vector, and initialize the iteration counter. The main loop follows below:

```
    # Find minimum
    while n <= N and norm(current.T) > tol:
        # Calculate new x,y,z
        x = x - eps*current.T[0]
        y = y - eps*current.T[1]

        # Calculate new value
        current = f(x,y)

        # Add new approximation to list
        x_a.append(x.val)
        y_a.append(y.val)

        # Increase iteration counter
        n += 1

    # Return values
    return x_a,y_a,n
```

This shows that for each iteration we start off by calculating the new approximation to the minimum using equation (5.3). We then calculate the value of $f(x_{n+1}, y_{n+1})$ before adding the new coordinates to the lists of approximations. When the loop finishes we return the lists of approximations as well as the total number of iterations.

5.3.4 Implementation : Newton's method

We will now look at how we can create a general solver for finding the minimum of a function of two variables using Newton's method. The code below shows the function definition and the required arguments:

```
def solver(f,x0,y0,N=1000,tol=1E-2):
    """
    f   - Function to find minimum of
    x0  - Initial guess for x
    y0  - Initial guess for y
    N   - Maximum number of iterations
    tol - Tolerance
    """
```

This shows that we require the same arguments as for the implementation of the solver for the method of steepest descent, with one exception. This is that we do not include the gradient multiplier ϵ as an optional argument, since this is not required for Newton's method. The code that precedes the main loop follows below:

```
    # Initial guess for minimum
    x,y = init_variables(x0,y0,max_o=2)

    # Current value and derivative vector
    current = f(x,y)
    J       = current.get_jacobian()
    H       = current.get_hessian()

    # Create list of approximations
    x_a = [x.val]
    y_a = [y.val]

    # Define norm
    norm = lambda grad: np.sqrt(np.sum(np.abs(grad)**2))

    # Initialize iteration counter
    n = 1
```

We start off by initializing the variables x and y using the arguments x_0 and y_0. Note that since we require the Hessian matrix we need to set to maximum order to 2. We than calculate the function for the points (x_0, y_0), before storing the first order derivatives and Hessian matrix in the variables J and H. Like for the method of steepest descent we create lists for holding the approximations to the minimum before initializing the iteration counter and defining a function for calculating the norm of the derivative vector. The main loop follows below:

```
    # Find minimum
    while n <= N and norm(J) > tol:
        # Calculate delta
        delta = -solve(H,J)

        # Calculate new x,y
        x = x + delta[0]
        y = y + delta[1]

        # Calculate new value
        current = f(x,y)
        J       = current.get_jacobian()
        H       = current.get_hessian()

        # Add new approximation to list
        x_a.append(x.val)
        y_a.append(y.val)

        # Increase iteration counter
        n += 1

    # Return values
    return x_a,y_a,n
```

For each iteration we start off by calculating δ using equation (5.8). We then calculate (x_{n+1}, y_{n+1}) using equation (5.6), followed by evaluating f at the new points and storing the first order derivatives and the Hessian matrix in J and H. The points (x_{n+1}, y_{n+1}) are then stored in the lists holding the approximations, before the iteration counter is updated. Like with the method of steepest descent the approximations and the total number of iterations are returned once the loop finishes.

5.3.5 Analysis of results

To see how the Method of Steepest Descent and Newton's method compare to each other it is necessary to use the two solvers to find the minimum of the same function. This function is given as follows:

$$f(x, y) = 4x^2y^2 + 8y^2 + 3x^2 + 27$$

This function has a minimum at $(0, 0)$. We set the initial guess at $\left(\frac{3}{2}, 1\right)$ and for the method of steepest descent the gradient multiplier is set to 0.05. We use the defaults for the rest of the arguments. The result of solving this is that they both find the minimum, with the Method of Steepest Descent requiring 15 iterations, while Newton's method only needs 3 iterations. Figure 5.1 on page 51 shows a plot of the vector norm of the distance from the minimum for each iteration for the two methods.

This shows that the distance to the minimum decreases faster for the Method of Steepest Descent initially. However, after a few iterations the method starts to move slower towards the solution. Newton's method, on the other hand, does not slow down as much, which results in fewer iterations. This can be explained by examining a plot of the function f. Figure 5.2 on page 51 shows the plot of f for $x, y \in [-2, 2]$.

This shows that the area around the minimum is very flat, which means that the magnitude of the derivative vectors in this area is small. Recall that the iteration formula for the Method of Steepest Descent was given as follows in equation (5.3):

$$x_{i+1} = x_i - \epsilon \nabla f(\vec{x}_i)$$

This shows that the smaller the magnitude of the derivative vector, $\nabla f(\vec{x}_i)$, the smaller the steps taken for each iteration will be. This explains why the Method of Steepest Descent starts moving slower towards the minimum the closer it gets. Figure 5.3 on page 52 shows a contour plot of the function f, together with the paths followed by the two methods towards the minimum.

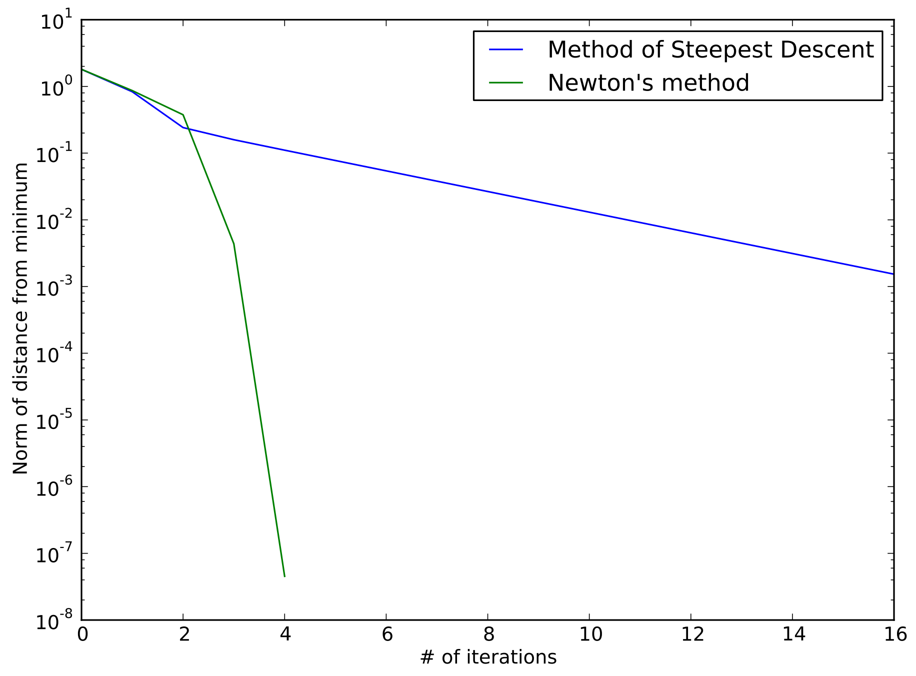

Figure 5.1: Distance to minimum as a function of the number of iterations.

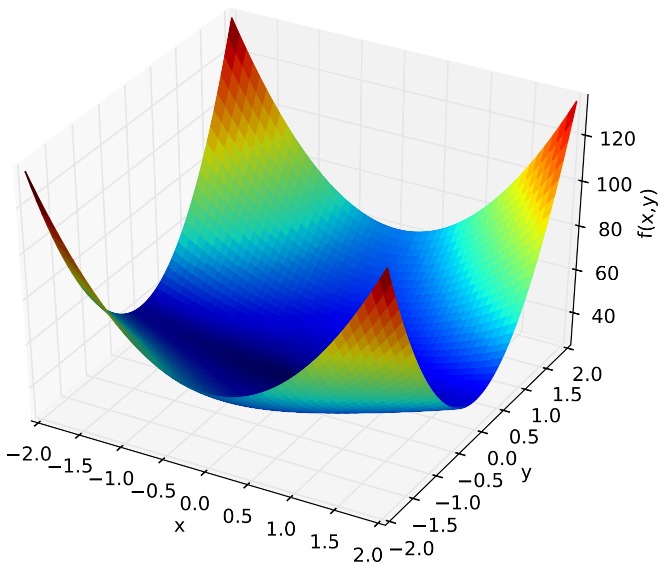

Figure 5.2: Plot of $f(x,y) = 4x^2y^2 + 8y^2 + 3x^2 + 27$

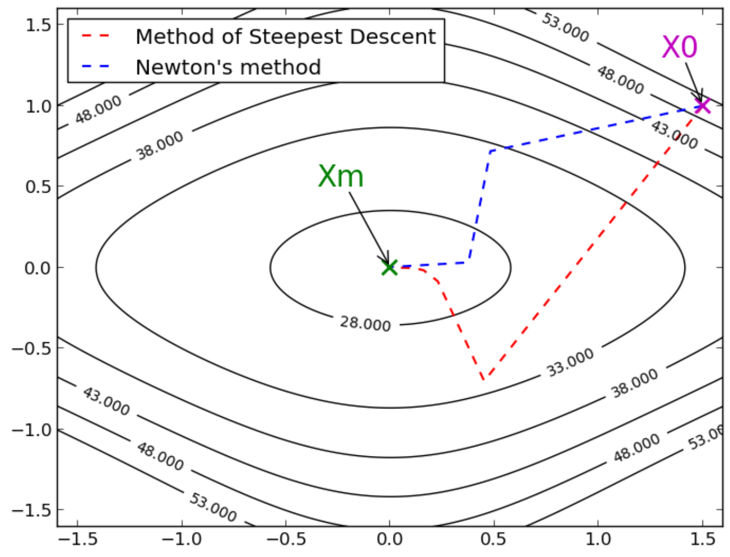

Figure 5.3: Paths toward the minimum point

This shows how Newton's method takes a more direct route towards the minimum compared to the Method of Steepest Descent. It also shows how the latter method takes increasingly smaller steps the closer it gets to the minimum, something that is natural since the gradient of the function decreases the closer we get. One way we could improve on this method would be to use an adaptive value for ϵ, instead of just using a constant. This would allow us to pick the best possible value for each iteration instead of just choosing a constant value up front.

5.4 Final Remarks

The optimization examples that we have gone through in this chapter shows the usefulness of the AD framework to get easy access to the derivatives of different functions. However, although it is easy to appreciate the convenience of using the AD framework, it is also important to consider how much of a difference this makes in relation to alternate ways of obtaining the derivatives. There are primarily two alternate ways we could have done this.

- Using finite differences
- Calculating the derivative expressions by hand

Using finite differences to approximate derivatives is an option that would work not only for the examples above, but for a generalized optimization application as well, since we can use finite differences to approximate the derivatives of arbitrary functions. The big weakness of using finite differences is that errors are introduced into the value of the derivatives, which in optimization applications may cause a method to converge slower or fail altogether. The upside is that finite differences can be faster than using the AD framework, however this depends on the cost of evaluating the function we need the derivatives of.

Calculating the derivative expressions by hand is a tedious exercise, but it does provide exact derivatives just like the AD framework, as well as performing much better. The drawback to this is that it can only be done for applications where we know the functions we will need derivatives for beforehand. For generalized optimization applications it is necessary to access the derivatives of arbritrary functions, thus ruling out this method entirely. Even for applications where we do know the functions we are working with, the AD framework might be a better solution. This could be if there is a large amount of derivative expressions, making deriving them time-consuming and error-prone, and a slight decrease in the speed of the final program is acceptable.

In conclusion the AD framework is a useful tool for optimization applications where derivatives are required, and where we are either dealing with arbitrary functions or the simplicity of using it is preferred to calculating derivatives by hand, despite a slight decrease in speed.

Chapter 6

Reservoir Simulation

6.1 Introduction

Over a period of millions of years, a combination of the accumulation of organic materials in layers of sediment and severe geological activity caused the creation of number of different hydrocarbons beneath the ocean floor. While some hydrocarbons managed to escape, in some places the geological activity trapped the hydrocarbons beneath layers of low-permeable or non-permeable rock. These hydrocarbons can now be found between 1000 and 3000 meters below the sea bed, and make up today's oil and gas reservoirs [2, p. 3-4].

Due to the multiple uses of oil and gas, a lot of time and effort is spent retrieving the hydrocarbons in these reservoirs. This process involves drilling through the rock trapping the hydrocarbons, and extracting them at the surface through wells reaching down into the reservoir. This is obviously a time-consuming and costly exercise, which makes it desirable to know as much as possible about the retrieval process. This is where reservoir simulation, which can be described as the modelling of subsurface flow in oil reservoirs, proves to be an invaluable tool.

Reservoir simulation is used for a wide range of different tasks, such as visualizing flow patterns and providing estimates of the production characteristics. However, the primary objective behind this is to provide the necessary information to help oil companies make decisions that lead to the maximization of the recovery of oil and gas [2, p. 1].

To describe the reservoir mathematically, in a way that allows us to solve it using a computer, we require two types of models. We need a mathematical model that describes the flow of fluids in a porous medium, which is typically expressed as a set of partial differential equations based on the conservation of mass, as well as a set of constitutive relations. We also require a geological model that describes the oil reservoir. This is accomplished by splitting the reservoir into cells, which make up a grid, and storing the relevant physical properties for each cell [5, p. 2].

In this chapter we will create a reservoir simulation solver using the simplest type of mathematical model, which is the Single-Phase Flow model. Geological models describing a reservoir may contain millions of grid cells, which makes it unfeasible to perform simulations with. In practice we use coarser grid-models that are created by upscaling the geophysical parameters of the original geological model [5, p. 3]. We will refer to the upscaled models as synthetic simulation models, which will be passed to the reservoir simulation solver as input, allowing us to use the solver to simulate different grid sizes. Creating a synthetic simulation model of an oil reservoir in a programming environment is a large task in its own right, and as such the solver will use models initially created in MATLAB with the MATLAB Reservoir Simulation Toolbox (MRST), developed by SINTEF Applied Mathematics [5]. This includes objects describing the cells, the necessary spatial information and the relevant physical properties for each cell. An object describing the oil well, created using MRST, is imported as well.

This chapter includes two main parts. First, we discuss the necessary theory behind the reservoir simulation, before moving on to the actual implementation. It should be noted that since we only look at a simple oil reservoir model, there are several issues that are relevant for a real world oil reservoir simulation that we do not account for here. The theory part of this chapter should therefore not be considered as a complete guide to reservoir simulation, but rather as a short introduction that aims to provide the necessary background information and understanding of the problem at hand.

6.2 Background theory on reservoir simulation

6.2.1 Oil Reservoir Characteristics

In this subsection we will look at the physical properties that we need to take into account in a reservoir simulation, and that are relevant for the example at hand.

Porosity and Pore Volume

The rock porosity is the void volume fraction of a piece of rock, usually denoted by ϕ. It is expressed as a number between 0 and 1, and will in most cases depend on pressure. The rock compressibility is defined as follows [2, p. 4]:

$$c_r = \frac{1}{\phi}\frac{d\phi}{dp} \tag{6.1}$$

Solving this equation yields the following expression for ϕ:

$$\phi\left(p\right) = \phi_0 e^{c_r(p-p_r)} \tag{6.2}$$

In the equation above ϕ_0 is the porosity at the reference pressure p_r, and we have assumed that the rock compressibility, c_r, is constant. The rock compressibility can either be constant or a function dependent on pressure, but will be set to a constant in this example. The pore volume is described as the total void volume in a piece of rock, and as such is given as the volume of the rock multiplied with the porosity. The expression for the pore volume follows below:

$$pv = pv_r e^{c_r(p-p_r)} \tag{6.3}$$

The term pv_r is the pore volume at the reference pressure p_r and is given as $pv_r = V\phi_0$.

Permeability

The permeability of a rock describes a rock's ability to transmit a single fluid, and is denoted by **K**. It is related to the porosity of the rock in the sense that the amount of pores present will affect how well it can transmit a fluid, but these quantities are not necessarily proportional since the orientation and interconnection of the pores plays a role as well. **K** is in general given as a tensor, in which case the permeability varies in different directions [2, p. 4-5]. In this example, however, the permeability is assumed to be the same in all directions and unaffected by each other, in which case the permeability is given as follows:

$$\mathbf{K} = k\mathbf{I}$$

The term **I** is the three-dimensional identity matrix, while k is a scalar describing the permeability in all directions.

Oil Density

In a real world reservoir there are several different phases present with different compressibilities. In this example, however, we will consider a reservoir that only includes oil. The oil compressibility is defined similarly to that of rock compressibility, and is defined as follows [2, p. 6]:

$$c_o = \frac{1}{\rho_o}\frac{d\rho_o}{dp} \tag{6.4}$$

Solving this yields an expression for the density of the oil, which is given as follows:

$$\rho_o(p) = \rho_{o,0} e^{c_o(p-p_r)} \tag{6.5}$$

The term $\rho_{o,0}$ is the oil density at the reference pressure p_r, and the oil compressibility, c_o, is assumed to be constant. Just like for rock compressibility the oil compressibility can be either a constant or a function dependent on pressure, but we will use a constant for this example.

6.2.2 Mathematical Model

As previously mentioned, the mathematical model consists of a system of partial differential equations that describe the relationship between the physical parameters of the reservoir. The simplest way to describe the displacements of fluids in a reservoir is by using a single-phase model, which means that we will only consider the flow of a single fluid, which in this example is oil. The basic differential equation we will be using to model the reservoir follows below [2, p. 9]:

$$\frac{\partial\left(\phi\rho\right)}{\partial t}+\nabla\cdot\left(\rho v\right)=q \tag{6.6}$$

This is called the continuity equation, which states that mass is conserved, where ϕ describes the rock porosity, ρ describes the fluid density, v the flow velocity and q models sources and sinks, in other words the inflow and outflow of fluid at the well perforations. This equation holds true for all the individual cells in the grid model. The flow velocity is described by Darcy's law as follows [2, p. 10]:

$$v=-\frac{\mathbf{K}}{\mu}\left(\nabla p+\rho g\nabla z\right) \tag{6.7}$$

Inserting equation 6.7 into equation 6.6 and moving the q term to the other side yields the final equation:

$$\frac{\partial\left(\phi\rho\right)}{\partial t}-\nabla\cdot\left(\rho\frac{\mathbf{K}}{\mu}\left(\nabla p+\rho g\nabla z\right)\right)-q=0 \tag{6.8}$$

Next, we need to specify the boundary conditions to close this model. It is common practice to use no-flow boundary conditions, which means that no liquid can enter or exit the reservoir. This is specified as $v\cdot\vec{n}=0$ on the reservoir boundary $\partial\Omega$, where $\vec{n}$ denotes the normal vector pointing out of the boundary [2, p. 10–11].

Next, we need to specify our constitutive relations. Let q_S denote the flow of liquid out of the oil well given as volume per second (m^3s^{-1}), and let $\hat{q}_i$ denote the flow at well perforation i given as weight per second (kgs^{-1}). Naturally the sum of the flow from individual well perforations should equal q_S, which gives rise to the following equation:

$$q_S-\sum_i\frac{\hat{q}_i}{\rho_S}=0$$

Where ρ_S equals the liquid density of the oil at the surface, something it is necessary to divide by to represent the flow as volume per second instead of weight. The final thing we need to take into account is the bottom hole pressure, p_{bh}. This can be set to any pressure value, which yields the following equation:

$$p_{bh}-p_{bhp}=0$$

Where p_{bhp} denotes the chosen bottom hole pressure value. If we let N denote the number of cells this gives us a total of $N+2$ equations that make up the mathematical model.

6.2.3 Discretizing the Mathematical Model

To be able to solve this set of PDEs it is necessary to find a discrete approximation to the differential operators that are present in the continuity equation (6.8), since these can not be applied in a programming environment. To do this we will use a finite-volume method. The continuity equation holds for all grid cells, Ω_i, so we integrate the equation over each of these grid cells. This yields the following expression:

$$\int\limits_{\Omega_i} \frac{\partial(\phi\rho)}{\partial t} dV + \int\limits_{\Omega_i} \nabla \cdot \left(\rho \frac{\mathbf{K}}{\mu} \left(\nabla p + \rho g \nabla z \right) \right) dV - \int\limits_{\Omega_i} q_i dV = 0 \tag{6.9}$$

We will proceed by discretizing each of the individual integrals in equation 6.9 one at a time.

Discretizing the integral including a time derivative

We will start off by discretizing the following integral:

$$\int\limits_{\Omega_i} \frac{\partial(\phi\rho)}{\partial t} dV$$

We can use a finite difference to approximate the time derivative which yields the following expression:

$$\frac{\partial\left(\phi(t_n)\rho(t_n)\right)}{\partial t} \approx D_t\left(\phi(t_n)\rho(t_n)\right) = \frac{\phi(t_n)\rho(t_n) - \phi(t_{n-1})\rho(t_{n-1})}{\triangle t} \tag{6.10}$$

In the expression above D_t denotes the finite difference derivative operator, while Δt denotes the size of the time step. Since the integral is performed over a single cell we know that ϕ and ρ are constant under the integral, and as such we can pull them out. This leaves us with the final expression:

$$\begin{aligned} \int\limits_{\Omega_i} \tfrac{\partial(\phi(t_n)\rho(t_n))}{\partial t} dV &\approx \int\limits_{\Omega_i} D_t\left(\phi(t_n)\rho(t_n)\right) dV \\ &= D_t\left(\phi(t_n)\rho(t_n)\right) \int\limits_{\Omega_i} dV \\ &= V_i D_t\left(\phi(t_n)\rho(t_n)\right) \end{aligned} \tag{6.11}$$

In the expression above the term V_i denotes the volume of cell i.

Discretizing the sources and sinks integral

The next integral to be discretized is the following:

$$\int\limits_{\Omega_i} q_i dV$$

The sources and sinks term, q_i, is considered a point source defined for certain cells where the well perforations are located. Integrating this yields the following result:

$$\int_{\Omega_i} q_i dV = \hat{q}_i \tag{6.12}$$

Note that the difference between q_i and $\hat{q}_i$ is that while $\hat{q}_i$ is given as weight per second (kgs^{-1}), q_i is given as weight per second per volume ($kgs^{-1}m^{-3}$).

Discretizing the integral involving the divergence operator

Finally, we are discretizing the following integral:

$$\int_{\Omega_i} \nabla \cdot \left(\rho \frac{\mathbf{K}}{\mu} \left(\nabla p + \rho g \nabla z \right) \right) dV = \int_{\Omega_i} \nabla \cdot (\rho v)\, dV$$

This can be rewritten using Green's theorem, which yields the following result:

$$\int_{\Omega_i} \nabla \cdot (\rho v)\, dV = \int_{\partial \Omega_i} (\rho v) \cdot \vec{n} ds$$

$\partial\Omega_i$ is the boundary of Ω_i, and $\vec{n}$ is the outward-pointing unit normal on $\partial\Omega_i$. Let $\Gamma_{i,j}$ denote the interface between cell i and j. We can then rewrite the above expression further to:

$$\int_{\partial\Omega_i} (\rho v) \cdot \vec{n}\, ds = \sum_{j \in J_i} v_{i,j} \tag{6.13}$$

$$v_{i,j} = \int_{\Gamma_{i,j}} (\rho v) \cdot \vec{n}_{ij}\, ds \tag{6.14}$$

In this expression J_i denotes the set of all the neighbors of grid cell i, while $\vec{n}_{ij}$ denotes the unit normal on the interface between cell i and j pointing outwards from cell i. The faces $\Gamma_{i,j}$ will be referred to as half-faces since they are associated with a particular grid cell and normal vector, but where each half face will have a twin half-face $\Gamma_{j,i}$ that has an identical area and opposite normal vector. Using the midpoint method we can approximate v_{ij} as follows [4, p. 123]:

$$v_{i,j} \approx A_{ij} \rho \left(\vec{x}_{i,j} \right) \vec{v} \left(\vec{x}_{i,j} \right) \cdot \vec{n}_{ij}$$

The term $A_{i,j}$ is the area of the face between cell i and j and $\vec{x}_{i,j}$ is the centroid of the face. We will approximate the density at $\vec{x}_{i,j}$ as an average between the two cells:

$$\rho \left(\vec{x}_{i,j} \right) \approx \frac{\rho_i + \rho_j}{2}$$

The Darcy velocity at $\vec{x}_{i,j}$ is approximated as follows:

$$\vec{v} \left(\vec{x}_{i,j} \right) = \left[\frac{\mathbf{K}}{\mu} \left(\nabla p + \rho g \nabla z \right) \right] \left(\vec{x}_{i,j} \right) \approx \frac{\mathbf{K_i}}{\mu} \nabla u \left(\vec{x}_{i,j} \right)$$

In this expression u is given as $u = p + \rho g z$ and we have approximated the permeability as the permeability of cell i. Next we need to approximate the gradient of u at the half-face centroid. We only know the averaged value of u, u_i, inside the cell. Let's assume that this is the value at the centre of the cell, and that the value of u at the half-face centroid is $\pi_{i,j}$. Let $\vec{c}_{i,j}$ denote the vector pointing from the cell centroid to the half-face centroid, and assume that u is a linear function. We can then estimate the gradient as follows [4, p. 123]:

$$\nabla u\left(\vec{x}_{i,j}\right) \approx \frac{\left(u_i - \pi_{i,j}\right)\vec{c}_{i,j}}{|\vec{c}_{i,j}|^2}$$

We can now write $v_{i,j}$ as follows:

$$v_{i,j} \approx A_{ij}\rho_i \frac{\mathbf{K}_i}{\mu}\frac{\left(u_i - \pi_{i,j}\right)\vec{c}_{i,j}}{|\vec{c}_{i,j}|^2}\cdot \vec{n}_{ij} = \frac{\rho_i}{\mu}T_{i,j}\left(u_i - \pi_{i,j}\right)$$

Where $T_{i,j}$ denotes the one-sided transmissibility associated with a single cell. These will be referred to as half-transmissibilities since they are associated with a half-face [4, p. 123]. To eliminate $\pi_{i,j}$ we impose the conditions $v_{i,j} = -v_{j,i}$ and $\pi_{i,j} = \pi_{j,i}$ [4, p. 124]. This gives us the following two equations:

$$T_{i,j}^{-1}v_{i,j} = \frac{\rho\left(\vec{x}_{i,j}\right)}{\mu}\left(u_i - \pi_{ij}\right) \qquad -T_{j,i}^{-1}v_{i,j} = \frac{\rho\left(\vec{x}_{i,j}\right)}{\mu}\left(u_j - \pi_{ij}\right)$$

Combining these equations allows us to eliminate π_{ij} and leaves us with the following two-point flux-approximation scheme:

$$v_{i,j} = \frac{\rho\left(\vec{x}_{i,j}\right)}{\mu}\left(T_{i,j}^{-1} + T_{j,i}^{-1}\right)^{-1}\left(u_i - u_j\right) = \frac{\rho\left(\vec{x}_{i,j}\right)}{\mu}T_{ij}\left(u_i - u_j\right) \tag{6.15}$$

$$T_{ij} = \left(T_{i,j}^{-1} + T_{j,i}^{-1}\right)^{-1}, \quad T_{i,j} = A_{ij}\frac{\mathbf{K_i}}{\mu}\frac{\vec{c}_{i,j}}{|\vec{c}_{i,j}|^2}\cdot \vec{n}_{ij} \tag{6.16}$$

T_{ij} in the above equation is the transmissibility associated with the connection between the two cells [4, p. 124]. The calculations above assume that we are dealing with an interface between two interior cells. For exterior faces we know that $v_{i,j} = 0$ because of the no-flow boundary condition defined as $\vec{v}\cdot\vec{n} = 0$ on the boundary.

6.2.4 Production Process and Well Rates

To retrieve oil from the reservoir it is necessary to drill holes down into the reservoir where the oil wells are put. There are many different production processes used to extract the oil to the surface, but in this case we will only consider the bottom hole pressure method. This consists of manipulating the pressure at the top of the well, known as the bottom hole pressure, which results in the oil moving from the reservoir and to the surface through the well. The formula for calculating the flow from a given well perforation i in the reservoir is given as follows [4, p. 172] :

$$\hat{q}_i = \frac{\rho_i}{\mu}WI\left(p_c - p_i\right) \tag{6.17}$$

The term WI is the well injectivity index (as first described by Peaceman in [9]), which is calculated in MATLAB and simply loaded into the Python program. The connection pressure p_c is given as follows in [4, p. 171]:

$$p_c = p_{bh} + g\rho\left(p_{bh}\right)\Delta z$$

In this expression p_{bh} denotes the bottom-hole pressure,Δz is the vertical distance from the bottom-hole to the well perforation and $\rho\left(p_{bh}\right)$ is the density of the oil at the bottom-hole pressure. In this example we will use a similar, but slightly different expression for p_c, where we replace the density at the bottom-hole pressure with the density at the grid cell pressure. The resulting expression follows below:

$$p_c = p_{bh} + g\rho\left(p_i\right)\Delta z = p_{bh} + g\rho_i\Delta z$$

6.2.5 Reservoir Model

Figure 6.1 on page 63, figure 6.2 on page 63 and figure 6.3 on page 64 shows the models of the reservoirs for the $10 \times 10 \times 10$, $20 \times 20 \times 20$ and $30 \times 30 \times 30$ grids respectively. These figures, which were created in MATLAB using MRST, illustrate that we are performing reservoir simulations for box-shaped reservoirs using grids consisting of grid cells that are all of the same size. All the models have the same physical dimensions, as well using the same values for all physical parameters.

Additionally, a single well is used, with the well perforations located at approximately at the same location for all models. The reason why they are not located at the exact same location is due to the fact that the locations of the well perforations are described by the grid cells they are in. Assuming that the well perforations are in the middle of their respective grid cells, this means that changing the grid size will in some cases not make it possible to choose the same locations for the well perforations. This is the case when going from a $10 \times 10 \times 10$ model to a $20 \times 20 \times 20$ model. This is equivalent to splitting each grid cell into two parts along each dimension, placing the well perforations at the boundary between different grid cells. Since this can not be set as a location, the result is that the well perforations are displaced a little compared to the smaller model.

Despite these small differences in the location of the well perforations, these models can be considered as modelling the same reservoir, but with different grid sizes. As such, we expect the results from the reservoir simulations with these models to return similar results.

6.2.6 Newton's method for PDE's

The mathematical models used in reservoir simulation are typically sets of partial differential equations, which we have discussed in this chapter already. This can be formulated in the following manner:

$$\mathbf{F}\left(\vec{x}\right) = 0$$

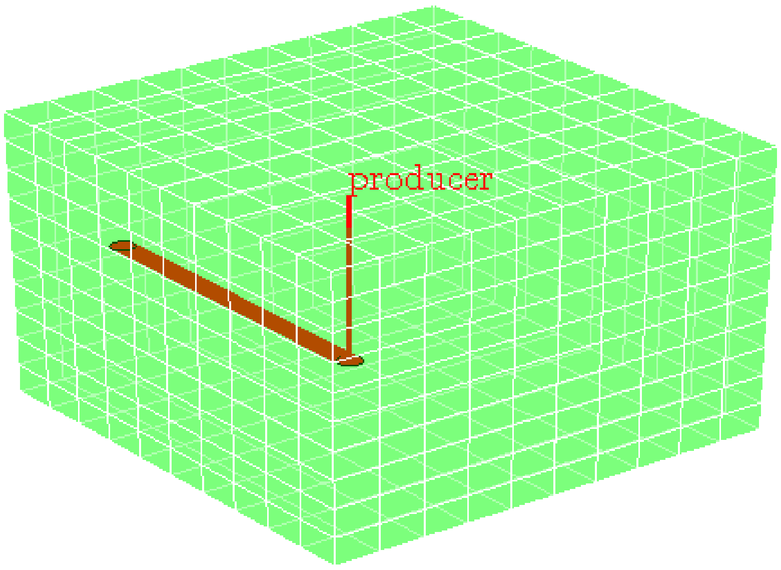

Figure 6.1: 10 × 10 × 10 Reservoir Grid

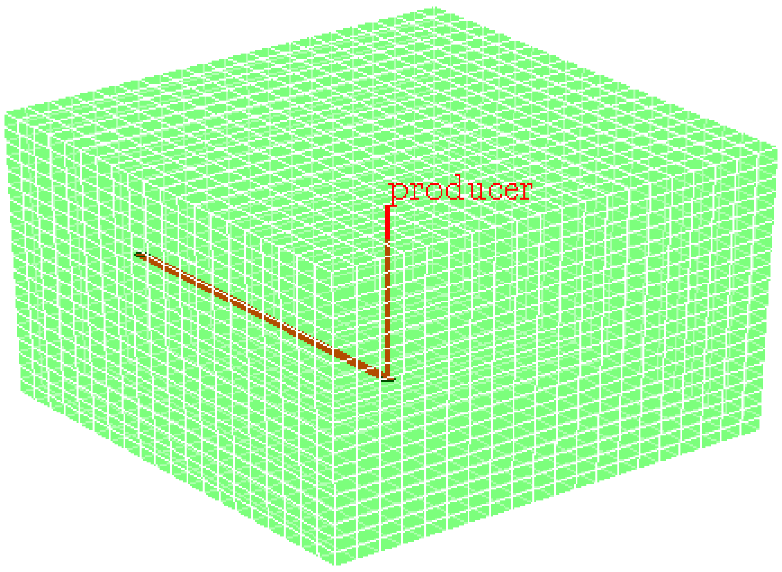

Figure 6.2: 20 × 20 × 20 Reservoir Grid

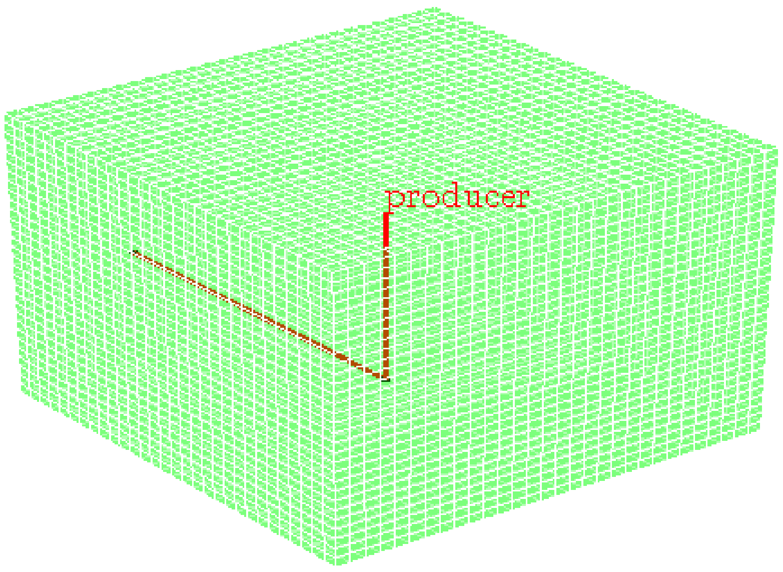

Figure 6.3: $30 \times 30 \times 30$ Reservoir Grid

In this equation each component of $\mathbf{F}$, $\mathbf{F_i}$, represents one of the partial differential equations to be solved. Recall that we in section 5.3.2 derived an iterative method for exactly the same type of equation, which was as follows:

$$\vec{x}_{i+1} = \vec{x}_i + \vec{\delta}_i \qquad \vec{\delta}_i = -\mathbf{J}^{-1}\left(\vec{x}_i\right)\mathbf{F}\left(\vec{x}_i\right)$$

This is Newton's method as noted in section 5.3.2, where the Jacobi matrix $\mathbf{J}$ is given as follows:

$$\mathbf{J_{ij}}\left(\vec{x_0}\right) = \frac{\partial \mathbf{F_i}\left(\vec{x_0}\right)}{\partial x_j}$$

The only difference from the derivation in the optimization section is that we are now considering a set of partial differential equations, instead of a multi-dimensional function. Since we have discretized the partial differential equations, we can simply use this exact formula for solving our system of PDE's as well. It is important to note that to find the inverse of the Jacobian it is necessary for the number of equations to equal the number of variables. We have $N + 2$ equations, where N denotes the number of cells, while we have N pressure variables in addition to q_S and p_{bh}. We have, in other words, the same amount of equations as variables, so we can use Newton's method. The pseudo code illustrating how this is done follows below:

```
# Initialize variables
x0 = val
nit = 1
tol = 1E-5
maxits = 10

```

```
# Begin main loop
while (resNorm > tol) and (nit < maxits):
    F = calculateF(x0)  # Find F(x0)
    J = calculateJ(x0)  # Find J(x0)
    upd = solve(-J,F)   # Find -J(x0)^-1 F(x0)
    x0 = x0 + upd       # Update x0

    # Update resNorm and nit
    resNorm = norm(F)   # Find the vector norm of F
    nit += 1            # Update current iteration

if nit == maxits:
    print "The␣Newton␣method␣did␣not␣converge."
return x0
```

This shows that we set $\vec{x}_0$ to a certain value initially and calculate a new value for $\vec{x}_0$ by finding $\vec{\delta}$, here denoted as 'upd'. For each iteration we find the vector norm of F and update the current iteration 'nit'. The method stops and returns $\vec{x}_0$ when the norm of F falls below a certain threshold, which is usally set to a low number, e.g. 10^{-5}, or when we have performed the maximum number of iterations without finding a satisfactory approximation to $\vec{x}$. For each iteration it is necessary to calculate the vector of values $\mathbf{F}(\vec{x}_0)$ and the Jacobian matrix $\mathbf{J}(\vec{x}_0)$. This shows the usefulness of using the AD framework since it enables us to calculate the vector $\mathbf{F}(\vec{x}_0)$ and automatically be able to access the derivatives that make up the Jacobian matrix.

6.3 Reservoir simulation in Python

In this section we will start off by going through the implementation of the single-phase reservoir simulation solver. We will then look at a comparison between the performance of the solver compared to a similar solver implemented in MATLAB, before moving on to an analysis of the results from running simulations with different grid sizes.

6.3.1 Implementation of Solver

In this section we will go through the file reservoir_sim_1p.py, which includes a solver method that can be used to simulate a given synthetic simulation model using a single-phase mathematical model. The solver is based on a similar implementation in MRST, which can be found in the appendix. The first lines of the file follow below:

```
"""
Description :
Reservoir simulator using a single-phase model.

Parameters :
datafile    - MATLAB data file with data structures for the
              reservoir geometry and well.
profile     - When set to true this enables profiling of the
              main body of the program.
```

```
iter_solve - When set to true the solver uses an iterative
             solver for solving the matrix system J*upd = -F
"""
def solver(datafile, profile=False,iter_solve=False):
    # Initialize profiling
    # ------------------------------------------------------------
    if profile:
        pr = cProfile.Profile()
        pr.enable()

    # Read MATLAB data file
    # ------------------------------------------------------------
    matlab_data = loadmat(datafile,struct_as_record=True)

    # Get geometry grid
    # ------------------------------------------------------------
    G = matlab_data['G']
    G = process_geometry(G)

    # Get data for single horizontal well
    # ------------------------------------------------------------
    W = matlab_data['W']
    W = process_well(W)

    # Get rock data (permeability and porosity)
    # ------------------------------------------------------------
    rock = matlab_data['rock']
    rock = process_rock(rock)
```

This shows that the solver method takes three arguments, datafile, profile and iter_solve. The datafile contains the data structures for the geometric grid, oil well and rock properties which are loaded and processed in the code above. The processing involves making the data easily accessible in Python, since the storage of the data is a bit cumbersome by default. The argument profile can be set to true, in which case the main body of the code is profiled and the results printed, while the iter_solve argument can be set to true to use an iterative solver for the matrix equation in Newton's method. The next few lines of code follow below:

```
    # Rock properties
    # ------------------------------------------------------------
    cr  = 1e-6/barsa              # Set rock compressibility
    pv_r = poreVolume(G,rock)     # Reference pore volume
    p_r  = 200*barsa              # Reference pressure

    # Pressure dependent function for pore volumes
    def pv(p):
        return AD.vec_mult(pv_r,exp(cr*(p-p_r)))
```

This part defines a few physical properties and defines a function for finding the pressure dependent pore volume of the grid cells, as given in equation (6.3). The next section of code follows below:

```
    # Fluid (oil) properties
    # ------------------------------------------------------------
    mu    = 5*centi*poise              # Assume constant viscousity
    c     = 1e-3/barsa                 # Constant oil compressibility
    rho_r  = 850*kilogram/meter**3    # Reference oil density
    rhoS   = 750*kilogram/meter**3    # Surface oil density

    # Pressure dependent function for oil density
    def rho(p):
        return rho_r*exp(c*(p-p_r))
```

This defines another few physical properties and defines a function for the pressure dependent oil density, as given in equation (6.5). The next section of code follows below:

```
    # Initial conditions
    # -----------------------------------------------------------------
    z_0 = 0
    z_max = np.max(G['cells']['centroids'][:,2])
    zz = np.linspace(z_0,z_max,100)
    pp = [p_r]
    func = lambda z,p: g*rho(p)
    odesolver = ode(func)
    odesolver.set_initial_value(p_r,z_0)

    for zval in zz[1:]:
        odesolver.integrate(zval)
        pp.append(odesolver.y)

    f = interp1d(zz,pp,kind='cubic')
    p_init = f(G['cells']['centroids'][:,2])
```

To find the initial pressure for the grid cells we assume that the reservoir is at equilibrium initially, in which case the following differential equation must be satisfied [4, p. 169]:

$$\frac{dp}{dz} = g\rho\left(p\right)$$

This ODE is solved in the piece of code above, using the initial condition $p\left(z_0\right) = p_r$. The point, z_0, is set to 0. Since the reservoir geometry is defined relative to this height, this can be done without loss of generality [4, p. 169]. The next section of code follows below:

```
    # Simulation components
    # -----------------------------------------------------------------
    N       = G['faces']['neighbors']
    intInx  = np.nonzero(np.prod(N,1))[0]
    N       = N[intInx,:]-1
    n       = N.shape[0]

    # Calculate interior transmisibilities
    T = computeTrans(G,rock)

    # Create divergence and gradient matrix
    rows = np.array(range(n),dtype=int)
    i = np.hstack((rows,rows))
    j = np.hstack((N[:,0],N[:,1]))
    ij = np.vstack((i,j))
    data = np.hstack((np.ones(n,dtype=int),-np.ones(n,dtype=int)))
    C = ss.csr_matrix((data,ij),shape=(n,G['cells']['num']))
    C_T = C.transpose()

    # Discrete grad and div operators
    def grad(x):
        return -AD.smat_mult(C,x)

    def div(x):
        return AD.smat_mult(C_T,x)

    # Averaging function
    nc = np.max(np.max(N))+1
    data = 0.5*np.ones(2*n)
   M = ss.csr_matrix((data, ij),shape=(n,nc))

    def avg(x):
        return AD.smat_mult(M,x)
```

The first part finds all the interior faces of the geometrical model. The next part calculates all the transmissibilities for the interior faces as defined in equation (6.16). Because of the boundary condition all the terms $v_{i,j}$, defined in equation (6.15), on the exterior boundary will equal zero, so it is not necessary to calculate the transmissibilities for these surfaces.

Next a matrix C and its transpose are created and used in the functions 'grad' and 'div'. These functions are used to calculate the $v_{i,j}$ terms in the pressure equations. The 'grad' function is used to help create all individual contributions $v_{i,j}$, while the 'div' function is used to sum them together properly. A third matrix is also created that is used to efficiently find the averages of $\frac{\rho}{\mu}$ along all boundaries between grid cells. This is done in the 'avg' function. The next section of code follows below:

```
    # Pressure and well equations
    # ---------------------------------------------------------------------
    # Get z variable
    z = G['cells']['centroids'][:,2]

    # Well rates
    wc = W['cells']-1 # Perforation grid cells
    WI = W['WI']      # Well indices
    dz = W['dZ']      # Perforation depth relative to well ref depth

    # Create function for calculating well rates
    def wellRates(p,bhp):
        p_wc = p[wc]
        rho_pwc = rho(p_wc)
        return AD.vec_mult(WI/mu,rho_pwc)*\
                (bhp-p_wc+AD.vec_mult(g*dz,rho_pwc))

    # Function for the pressure equation
    def pressureEq(p,p0,dt):
        rho_p = rho(p)
        return (1/dt)*(pv(p)*rho_p-pv(p0)*rho(p0)) - \
            div(avg(rho_p/mu)*AD.vec_mult(T, \
            grad(p-AD.vec_mult(g*z,rho_p))))
```

Initially we find the z variable for all the different grid cells. Next we find the grid cells the well is connected to, as well as some properties for these grid cells. The function 'wellRates' is defined and calculates the flow rate $\hat{q}_i$ for each perforation grid cell for a given pressure and bottom hole pressure. This is calculated as given in equation (6.17). Next the function pressureEq is defined. This calculates the discretized single-phase flow equation, but with the exception of the $\hat{q}_i$ term. The next section of code follows below:

```
# Initialize AD variables
# ------------------------------------------------------------------------
p_ad, bhp_ad, qS_ad = init_variables(p_init, p_init[wc[0]], 0, \
        max_o=1,update_num=True,dtype=np.float64,sparse=True)

# Create AD indices
pIx    = range(G['cells']['num'])
bhpIx  = G['cells']['num']
qSIx   = G['cells']['num']+1
```

The AD variables are initialized first, where p_ad is an AD vector that describes the pressure for all grid cells, while bhp_ad and qS_ad are AD variables that describe the bottom hole pressure and the well rate at the surface, respectively. The final three initializations create indices to be used later. The next section of code follows below:

```
# Simulation parameters
# ------------------------------------------------------------------------
numSteps  = 52
totTime   = 365*day
dt        = totTime / numSteps
tol       = 1e-5
maxits    = 10

# Set up structure for holding solution
sol = {}
sol['time'] = []
sol['pressure'] = []
sol['bhp'] = []
sol['qS'] = []

# Add initial values to 'sol'
sol['time'].append(0)
sol['pressure'].append(p_ad.val)
sol['bhp'].append(bhp_ad.val)
sol['qS'].append(qS_ad.val)
```

Some basic simulation parameters are initialized here and the data structure for holding the solution is created. The next section of code follows below.

```
# Simulation
# ------------------------------------------------------------------------
t = 0
step = 0
while t < totTime:
    t += dt
    step += 1
```

```
        resNorm = 1e99
        p0 = p_ad.val
        nit = 0

        # Print progress
        print str(step)+'/'+str(numSteps)

        while (resNorm > tol) and (nit < maxits):
            # Create equations
            wr = wellRates(p_ad,bhp_ad)
            eq_1 = pressureEq(p_ad,p0,dt)
            eq_1[wc] = eq_1[wc] - wr
            eq_2 = qS_ad - AD.ADsum(wr)/rhoS
            eq_3 = bhp_ad - 100*barsa

            # Solve
            eq  = AD.ADappend(eq_1,[eq_2,eq_3])
            J   = eq.T                            # Get Jacobian
            res = eq.val                          # Get residual

            # Newton Update
            if not iter_solve:
                # Use regular solver
                upd = spsolve(-J,res)
            else:
                # Use iterative solver
                upd = ssl.lgmres(-J,res,tol=1e-7)
                upd = upd[0]

            # Update variables
            p_ad.setval(p_ad.val+upd[pIx])
            bhp_ad.setval(bhp_ad.val+upd[bhpIx])
            qS_ad.setval(qS_ad.val+upd[qSIx])

            # Update counter and residual
            resNorm = norm(res)

        if nit == maxits:
            print "The Newton method did not converge."

        # Update solution for current time level
        sol['time'].append(t)
        sol['pressure'].append(p_ad.val)
        sol['bhp'].append(bhp_ad.val)
        sol['qS'].append(qS_ad.val)
```

This shows the main part of the simulation. For each time step the main loop tries to approximate the point where the PDE system $\mathbf{F}(\vec{x}) = 0$ using the Newton iteration method. The Newton iterations stop when the vector norm of $\mathbf{F}(\vec{x})$ falls below the threshold value 'tol' or when it has performed 'maxits' iterations without falling below the threshold value.

```
    # Print profiling results
    if profile:
        pr.disable()
        s = StringIO.StringIO()
        sortby = 'cumulative'
        ps = pstats.Stats(pr,stream=s).sort_stats(sortby)
        ps.print_stats()
        print s.getvalue()

    # Return solution
    return sol
```

These are the final lines of the solver function. If profiling is enabled it is stopped and the results are printed. The last line simply returns the result of the simulation.

6.3.2 Performance Comparison to MATLAB Implementation

To compare how the Python implementation holds up against a similar implementation in MATLAB, that uses an AD framework as well, it is necessary to solve the same problem using both implementations with the same computing resources available. Both implementations were tested for three models of different sizes. The problem was the same, but the reservoir was split into different sized grids with 10,20 and 30 grid cells along each spatial dimension. All simulations were run on the same computer, with the following specifications:

	MATLAB	Python
OS	Windows 7 Home Premium 64-bit	Ubuntu 14.04 64-bit
Processor	Intel Core i7-2670QM	Intel Core i7-2670QM
Memory	8GB	8GB
Hard Drive	Crucial M500 480GB SSD	Seagate Laptop SSHD 1TB

The only difference in hardware is the fact that the simulations were run on different hard drives, but since reading from disk only plays a small part in the Python implementation it is reasonable to assume that this provides a good comparison of the two implementations. The MATLAB implementation was run using MATLAB R2013a, while the Python implementation was run in Python 2.7.3. Additionally, the Python implementation uses the numpy and scipy libraries extensively, with the BLAS and LAPACK implementations being provided by the ATLAS library. The simulations were run 10 times for each grid size. The average run times are presented in the table below:

	N	Total	Solve	Rest
MATLAB	10	4.90	1.36	3.54
	20	26.70	19.14	7.56
	30	153.12	132.42	20.7
Python	10	4.44	0.97	3.47
	20	22.32	17.11	5.21
	30	209.42	197.49	11.93

This shows that the average run times are approximately the same for the two smallest grid sizes, while for largest grid the MATLAB implementation is somewhat faster. The table also shows the two columns 'Solve' and

'Rest', splitting the total time spent into the time used to solve the matrix equation $J * upd = -F$ and the time spent on the rest of the simulation. This shows that everything except solving the matrix equation is performed faster in Python than in MATLAB for all grid sizes. Some of this difference in run times could be explained by the fact that the Python program only reads some of the data structures instead of creating them, but it does show that the Automatic Differentation part of the Python program performs just as well or even better than a similar implementation in MATLAB.

The MATLAB implementation uses the built-in function mldivide to solve the matrix equation $J * upd = -F$, which uses the UMFPACK library. This is used in the Python implementation as well, which means it should be possible to get the run times for solving the matrix equation in Python down to a level similar to that of the MATLAB implementation. The Python implementation also includes the alternative of using an iterative solver, which speeds up the program significantly, that can be used for even larger grid sizes.

6.3.3 Analysis of results

Figure 6.4 on page 73 shows a plot of the calculated volume rate per day for the three different grid sizes, while figure Figure 6.5 on page 73 shows the cumulative extracted volume over the course of a year. The plots show that the the volume rate is about the same for all three models, with the same monotone decreasing behaviour. The fact that they all show about the same volume rate is as expected, since the models are the same, but with different grid sizes.

Increasing the grid size will, in general, increase the accuracy of the calculation. However, the larger the grid gets, the smaller the possible gain will be by increasing it further. This could explain why the graphs for the two larger models are closer together than the $20 \times 20 \times 20$ model is to the $10 \times 10 \times 10$ model, since the increase in accuracy is larger from $10 \times 10 \times 10$ to $20 \times 20 \times 20$ than from $20 \times 20 \times 20$ to $30 \times 30 \times 30$. Another thing that could contribute to the difference in the results is the fact that changing the grid size means changing, if only a little, the location of the well perforations. This was discussed in section 6.2.5 and despite resulting in very similar well models, this could have an effect on the differences between the different grids.

In conclusion, the difference in the results can be attributed to the difference in grid sizes, as well as a slight difference in the location of the well perforations. Additionally, the differences are no larger than what we would expect, which shows that the reservoir simulation solver works for different grid sizes.

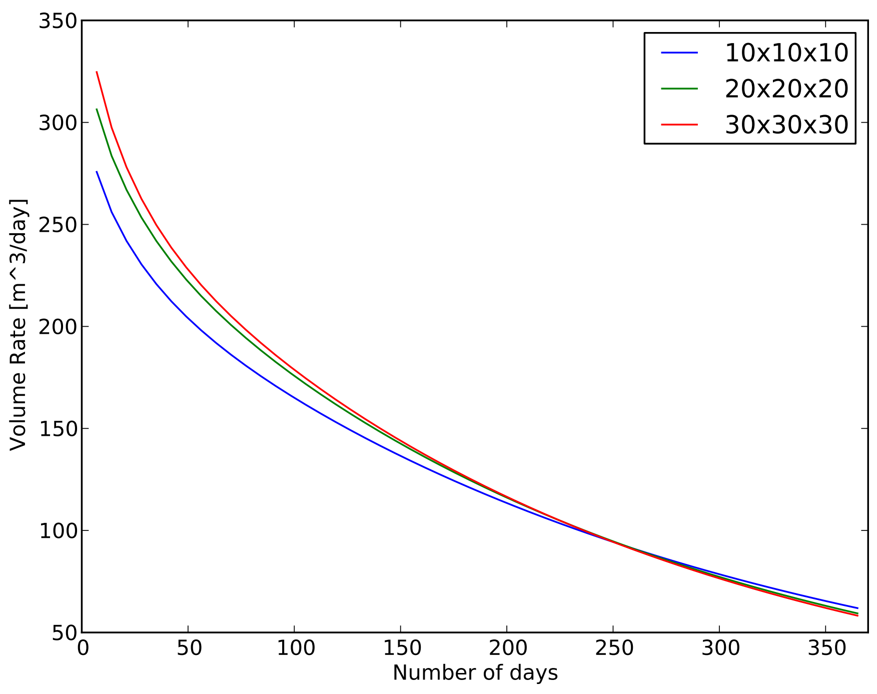

Figure 6.4: Plots surface volume rate per day over the course of a year

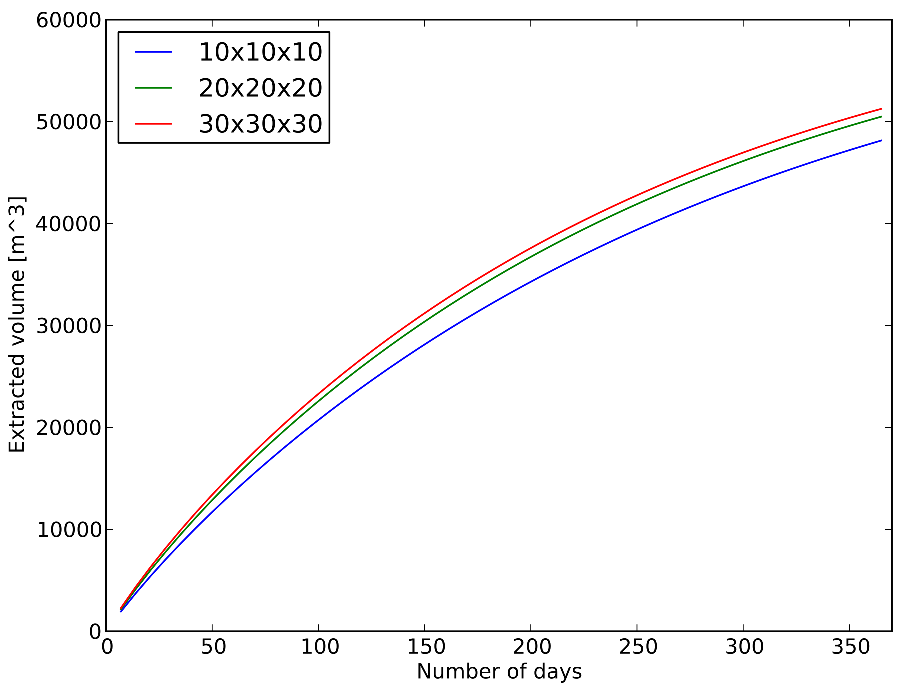

Figure 6.5: Plots the cumulative extracted volume

6.4 Final Remarks

This chapter has shown that the Automatic Differentiation framework is a useful tool for calculating the derivatives necessary to perform a reservoir simulation using Newton's method. The two alternatives to using the AD framework are, just like for the optimization example, to either use finite differences or to calculate all the derivatives by hand. Finding all the derivatives by hand would clearly be a large task and very likely to cause bugs in the program due to miscalculations. Although this might still be a viable option, for even more complex models this would be an immense amount of work. Clearly, this is not a very practical option.

Using finite differences, on the other hand, is a possible option for calculating the necessary derivatives. However, it does have a few disadvantages compared to using an AD framework, the first of which is that it just approximates the derivatives as opposed to calculating them exactly. In terms of performance it can be very costly as well. If we have a grid with N cells then we have $N + 2$ equations and $N + 2$ variables. This leaves us with $(N + 2)^2$ first order derivatives that need to be calculated for each Newton iteration. Many of these are simply equal to zero, which allows us to reduce the total amount of finite difference calculations, but the total amount will still be $\mathcal{O}\left((N + 2)^2\right)$. Whether or not this is a viable option depends on how costly it is to evaluate the function we are finding the derivatives of. In this example, where evaluating the function involves a lot of matrix mulitplication with large matrices, finite differences is simply not a good option since the function evaluation cost is high.

The conclusion we can draw from this is that the AD framework is the best tool for solving this problem. Additionally, as shown in the performance analysis, Python is a viable alternative to MATLAB for this type of scientific computing since Python outperformed MATLAB on everything except solving the matrix equation $J * upd = -F$. By changing to an even faster solver, Python would equal or even outperform MATLAB's performance, making it a great and free option to using MATLAB.

Part III

Conclusion

Chapter 7

Summary and Analysis of the Thesis

7.1 Summary and evaluation

This thesis consists of two main parts, the first of which is the theory behind and implementation of an Automatic Differentiation framework, while the second is about applications that use the AD framework and show its usefulness. Although the applications take up a large part of the thesis, the underlying focus is to show how the AD framework can be used to help solve different mathematical problems, and not on the problems themselves. The main theme of thesis is therefore on Automatic Differentiation.

When work on the thesis first began, however, the idea was not initially to write a thesis where the main focus was Automatic Differentiation. Instead, the intention was to create a simple AD framework that could provide the necessary derivatives to help solve different reservoir simulation problems using MATLAB. Automatic Differentiation was in other words intended to have a secondary focus, while the main focus was supposed to be on reservoir simulation. Since using an AD framework to solve reservoir simulation problems obviously required an AD framework I started researching Automatic Differentiation. Since I had not delved very deep into reservoir simulation at that point I was not aware that higher order derivatives were not required, and eventually I started working on a general AD framework.

Once I came to the realization that it was not necessary to calculate derivatives higher than the first order for use in reservoir simulation, my thought process was that I might as well complete the general AD framework and use this for solving the reservoir simulation problems. This was followed by the decision to switch from MATLAB to Python, which made it necessary to rewrite the AD framework in Python. At this point a first order derivative AD framework had been implemented as part of MRST, so the reasoning behind the switch was that it would make it possible to compare the performance of reservoir simulations in Python to that of MATLAB.

Once I finished the AD framework in Python and moved on to the reservoir simulation, I discovered that the general formulas for calculating Taylor coefficients of arbitrary order were too slow to calculate the amount of first order derivatives necessary for the reservoir simulation. This was solved by using simpler first-order derivative formulas when only calculating first order derivatives. This allowed me to vectorize the calculations, which combined with the use of sparse matrices to store derivatives resulted in the AD framework being fast enough to efficiently solve the reservoir simulation problem.

The end result is that the AD framework is able to handle arbitrary derivatives as I intended, but it effectively contains two different AD framework implementations. One for general order derivatives and another optimized for the calculation of first order derivatives. This is in itself certainly not a bad thing, since it makes it a very flexible tool for different use-cases. However, the amount of time spent doing this resulted in the main focus of the thesis shifting from reservoir simulation to Automatic Differentiation.

This shift meant that it was not sufficient to include reservoir simulation as the only application using the AD framework. This is due to the fact that the reservoir simulation problem only requires first order derivatives, and as such it was necessary to include an example that uses derivatives of a higher order as well. This resulted in the inclusion of a chapter showing how the AD framework can be used to help solve optimization problems, since this illustrates how the AD framework can be used to supply higher order derivatives.

My own evaluation of the thesis as a whole is that it does a reasonably good job of explaining how to go about implementing an AD framework and of showing the usefulness of it through the applications. I am also pleased that I was able to get the performance of the reservoir simulation in Python to a level similar to that of MATLAB, since this shows that Python is a viable option to MATLAB for computational problems in reservoir simulation. Despite these positives, I still feel that the thesis comes across as a bit disjointed and with a lack of a focussed theme. What I mean by this is that although Automatic Differentiation is the main theme of this thesis in terms of the work put into it, it still comes across as something between a thesis focussed on Automatic Differentiation and one focussed on reservoir simulation. Additionally, there is no clear question that is being asked and answered in the thesis, but rather a set of smaller ones such as whether or not Python is a viable alternative to MATLAB for reservoir simulation.

7.2 What could have been done differently?

There are many things that could have been done differently during the work on this thesis, the most obvious being sticking to the original intention of only creating a simple AD framework and keeping the main focus on how this could be used in reservoir simulation problems. This would have resulted in the main part of the thesis dealing with reservoir simulation, with the AD part being secondary. In this scenario I would have created solvers for more advanced models than the basic single-phase flow model, with the purpose being to provide a more comprehensive comparison of how Python compares to MATLAB for reservoir simulation.

Another possible approach would have been to focus even more on Automatic Differentiation and how to get the best possible performance. This would have resulted in the thesis consisting of even more material on how to calculate derivatives, with the focus on applications being somewhat less. In particular, this would probably have resulted in the implementation of reverse-mode and the use of univariate Taylor series as briefly discussed in section 3.6. It would likely include a discussion of more alternate approaches as well, since taking this approach from the beginning would have given me more time to just focus on Automatic Differentiation.

As for the applications, the optimization chapter would probably have been left largely unchanged, since this provides some examples of how the AD framework can be used to supply higher order derivatives. Reservoir simulation, however, would have played a smaller part than what is the case right now. This would be done out of necessity since there would not have been time to spend as much time on it as I have. Additionally, the part that shows the usefulness of AD in the reservoir simulation program is solving the matrix equation $J * upd = -F$. This is a good example since finite differences are too time consuming for this problem, and it is a lot of work to calculate the derivatives by hand. I would therefore have kept this, but I would have focussed less on the theory behind it and imported more data directly from MATLAB. I could for instance import all the data from MATLAB, and only implement the initialization of the AD variables and the main simulation loop, since this would still show how the AD framework could be used for this problem.

Although my opinion is that both options outlined above would have made for a better thesis overall, I think that the best course of action if I had the chance to do the thesis over again would be to focus less on Automatic Differentiation and more on reservoir simulation. Not only would this have been in line with the original plan for the thesis, but it is also the area of expertise of my advisor, making it a natural choice.

There are also things I would have done differently in terms of how I approached the work on the master's thesis. The first of these is that I would be careful not to narrow my focus too much, too soon. During the early part of the thesis I did exactly this, by focussing too much purely on Automatic Differentiation, when I instead should have tried to get a better overview before delving deeper into any particular subject. Once I realized that what I had spent a fair amount of time working on was not strictly necessary, I was reluctant to simply let it all be for nothing. This eventually resulted in a lot more work than I had intended, which made the shift in focus of the thesis necessary. Additionally, I have focussed too much on the actual programming as opposed to focussing on exploring a problem, with the programming only being a means to that end. This has resulted in a lot of time spent on programming, which might have been better spent focussing on the writing. An example of this is that the AD framework supports a large variety of functions, even though only a few of them are used in this thesis.

7.3 Conclusion

My own evaluation of the thesis is that it lacks a clear and focussed theme, partly because of the shift in focus during the work on the thesis. Despite this, there are many positives that I take away from the work on this thesis. I am pleased with the fact that I managed to implement a complete AD framework that is reasonably efficient, as well as being pleased with the fact that I managed to implement a reservoir simulation example in Python that is able to rival MATLAB in terms of performance. I also feel that I have learned a great many things about working on a large project over a longer period, that I will take with me and benefit from in the future.

Chapter 8

Further Work

Although my evaluation in the previous chapter was that the thesis lacks a bit of focus, it does touch on many different areas that could be interesting to explore further, either on their own or as part of a thesis. This chapter will go through these areas and discuss how they can be worked on in more depth.

8.1 AD framework performance improvements

The AD framework as it stands has seen a series of programming related performance improvements throughout the course of working on the master's thesis. These can be split into two parts, performance improvements on first order derivatives and performance improvements on arbitrary order Taylor coefficients. Of these two the latter has seen the most improvement, since improving this part of the AD framework was necessary for the reservoir simulation program to perform well.

In section 3.6 we went through two possible alternatives that could increase the performance of the AD framework. It was noted that using univariate Taylor coefficients improves the performance for the calculation of large orders of Taylor coefficients, while resulting in similar performance for lower Taylor coefficients. This could therefore be implemented to improve the performance of the AD framework for arbitrary Taylor coefficients.

Another possible improvement is to implement reverse-mode. This is in general more efficient for a large number of variables, and would definitely be an improvement for the calculation of Taylor coefficients of order larger than one for scenarios with a lot of variables. For the calculation of first order derivatives, however, it not quite clear whether or not this would be an improvement. This is because the first order derivative calculations use efficient vectorized calculations based on first order derivative expressions. Additionally, it has the option of using sparse matrices, thus eliminating a lot of calculations with derivatives that are simply equal to zero. This could be researched further and implemented to test whether or not it would result in an improvement in performance.

8.2 Building a reservoir simulation library in Python

The MRST library, implemented by SINTEF, was used to create the grid and well models that was used in the reservoir simulation problem in this thesis. Creating a library similar to MRST in Python would be an enormous task, and certainly too large to do as part of thesis. What was shown in my thesis, however, is that it is perfectly possible to import models created with MRST in MATLAB and use these to perform a reservoir simulation.

This opens the possibility to create a reservoir simulation library in Python that can solve different types of reservoir simulation models with the help of the AD framework, but where certain data structures that are difficult to create from scratch are created in MATLAB first and then imported into Python.

8.3 Using the AD framework for more advanced optimization problems

The optimization examples used to illustrate the usefulness of the AD framework in this thesis were simple and not fully representative of real world optimization problems. They did show how the AD framework simplified the programming of solutions, but this could still be expanded on to show how the AD framework can be used to help solve more advanced optimization problems. This could be done with the focus being on showing how the AD framework can be even more useful than what was shown in this thesis, or the AD framework could simply be used as a tool with the focus being primarily on optimization.

8.4 Building a more complete testing framework

During the process of building the AD framework a testing framework was created using the Python module nose [1], which enabled the running of several tests to check that the core functionality was still working whenever changes were made to the code. This was a great help during the development of the framework, but despite including tests covering a large range of the functionality of the AD framework, there is still plenty that it does not cover. Expanding and improving on the testing framework to make it more complete would therefore be an interesting task on its own, as well as being helpful if any major changes were to be made to the existing AD framework.

Part IV

Appendix

Appendix A

Automatic Differentiation

This appendix includes proofs for Taylor coefficient formulas not shown in the thesis, followed by all code related to the AD framework.

A.1 Proof of Formulas

A.1.1 Division

Assume that we want to know the derivatives resulting from the following operation:

$$h(\vec{x}) = \frac{f(\vec{x})}{g(\vec{x})}$$

This implies that $f = hg$. Using equation (3.2) for multiplication yields the following result:

$$T_{f,\vec{k}} = T_{hg,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}} T_{h,\vec{j}} T_{g,\vec{k}-\vec{j}}$$

The term under the sum for $\vec{j} = \vec{k}$ is $T_{h,\vec{k}} T_{g,\vec{0}} = g(\vec{x})\, T_{h,\vec{k}}$. Pulling this term out of the sum yields the following:

$$T_{f,\vec{k}} = \left(\sum_{\vec{j}<\vec{k}} T_{h,\vec{j}} T_{g,\vec{k}-\vec{j}} \right) + g(\vec{x})\, T_{h,\vec{k}}$$

Rearranging the terms yields an expression for $T_{h,\vec{k}}$:

$$T_{h,\vec{k}} = \frac{T_{f,\vec{k}} - \sum_{\vec{j}<\vec{k}} T_{h,\vec{j}} T_{g,\vec{k}-\vec{j}}}{g(\vec{x})} \tag{A.1}$$

This expression requires us to know the Taylor coefficients of h, which are not yet known prior to the calculation. However, for a given $\vec{k}$ it is only necessary to know the values of the Taylor coefficients in $\left\{ \vec{j} \mid \vec{j} \leq \vec{k} \right\}$, with the exception of $\vec{k}$ itself. This makes it possible to calculate $T_{h,\vec{k}}$ for all $\vec{k}$ as long as all derivatives of a lower order have been calculated beforehand.

A.1.2 Exponential function

Suppose that $\vec{h}(\vec{x}) = e^{g(\vec{x})}$ then $\partial^{\vec{e}} h = h\ \partial^{\vec{e}} g$, for any one-order vector $\vec{e}$. Using equation (3.3) yields the following formula for any $\vec{k} \geq \vec{e}$:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{g,\vec{k}-\vec{j}} \tag{A.2}$$

Additionally this formula uses the values of Taylor coefficients for lower derivatives than $\vec{k}$. So just like with division it is necessary to iterate over the derivatives such that all derivatives less than $\vec{k}$ are calculated beforehand.

A.1.3 Square root

Assume that $\vec{h}(\vec{x}) = \sqrt{g(\vec{x})}$. Then $\partial^{\vec{e}} h = \frac{\partial^{\vec{e}} g}{2h}$ for any one-order vector $\vec{e}$. This can also be rewritten as $\partial^{\vec{e}} g = 2h\ \partial^{\vec{e}} h$. Using equation (3.3) yields the following formula for any $\vec{k} \geq \vec{e}$:

$$T_{g,\vec{k}} = 2 \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{h,\vec{k}-\vec{j}}$$

For $\vec{j} = \vec{0}$ the expression under the sum equals $T_{h,\vec{0}}\, T_{h,\vec{k}} = h(\vec{x})\, T_{h,\vec{k}}$. Pulling this out of the sum yields:

$$T_{g,\vec{k}} = 2h(\vec{x})\, T_{h,\vec{k}} + 2 \sum_{\substack{\vec{j}>\vec{0} \\ \vec{j}\leq\vec{k}-\vec{e}}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{h,\vec{k}-\vec{j}}$$

Rearranging this expression yields:

$$T_{h,\vec{k}} = \left(T_{g,\vec{k}} - 2 \sum_{\substack{\vec{j}>\vec{0} \\ \vec{j}\leq\vec{k}-\vec{e}}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{h,\vec{k}-\vec{j}} \right) / 2h(\vec{x}) \tag{A.3}$$

A.1.4 Inverse trigonometric functions

Assume that $\vec{h}(\vec{x}) = \arctan(g(\vec{x}))$. Then $\partial^{\vec{e}} h = f\ \partial^{\vec{e}} g$ for any one-order vector $\vec{e}$ where $f = \frac{1}{1+g^2}$. Using equation (3.3) yields the following formula for any $\vec{k} \geq \vec{e}$:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{f,\vec{j}}\, T_{g,\vec{k}-\vec{j}} \tag{A.4}$$

In this case f is not known initially, but since we know g and that $f = \frac{1}{1+g^2}$, this can be calculated before calculating the Taylor coefficients of h.

The same formula can be used for arccos and arcsin, but with different expressions for f. For arcsin the expression is $f = \frac{1}{\sqrt{1-g^2}}$, while for arccos it is $f = \frac{-1}{\sqrt{1-g^2}}$. The derivation of formulas for the Taylor coeffecients of the other inverse trigonometric functions follow a similar argument.

A.1.5 Trigonometric functions

Assume that $h(\vec{x}) = \sin(f(\vec{x}))$ and $g(\vec{x}) = \cos(f(\vec{x}))$. Then $\partial^{\vec{e}} h = g\, \partial^{\vec{e}} f$ and $\partial^{\vec{e}} g = -h\, \partial^{\vec{e}} f$ for any one-order vector $\vec{e}$. Using equation (3.3) yields the following two formulas for any $\vec{k} \geq \vec{e}$:

$$T_{h,\vec{k}} = \sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{g,\vec{j}}\, T_{f,\vec{k}-\vec{j}} \tag{A.5}$$

$$T_{g,\vec{k}} = -\sum_{\vec{j}=\vec{0}}^{\vec{k}-\vec{e}} \frac{k_i - j_i}{k_i} T_{h,\vec{j}}\, T_{f,\vec{k}-\vec{j}} \tag{A.6}$$

This shows that to calculate one we need to know the other, but only for derivatives lower than $\vec{k}$. This means that we can find both the sine and cosine of f by calculating both simultaneously.

The rest of the trigonometric functions can all be calculated by finding the sine and cosine first. Assume that $h(\vec{x}) = \tan(f(\vec{x}))$, then we can find $\sin(f(\vec{x}))$ and $\cos(f(\vec{x}))$ and then calculate h as $h(\vec{x}) = \frac{\sin(f(\vec{x}))}{\cos(f(\vec{x}))}$. A similar method can be used for the rest of the trigonometric functions.

A.2 Code

A.2.1 AD class

The code for the file adipy.py with the entire AD class follows below. Note that the function csr_add_sparse_vec, which adds a sparse vector to every row of a sparse matrix, was found on stackoverflow at the following address : http://stackoverflow.com/questions/15239491/adding-a-very-repetitive-matrix-to-a-sparse-one-in-numpy-scipy. This is pointed out in the code as well.

```
import numpy as np
import math
import mathADI
from scipy.misc import factorial
from scipy.sparse import csr_matrix as smat
import scipy.sparse as ss
import sys
import itertools
from numpy import ravel_multi_index as rmi
from numpy import unravel_index as ui

```

```python
class AD:
    """
    Properties :
    val         - Expression value
    T           - Array of taylor coefficients of expression
    num_vars    - Number of variables
    max_o       - Maximum order of derivatives
    ADvector    - True if the AD class holds more than one variable
    N           - Number of variables
    dtype       - Data type to store coefficients with.
                  (Default=complex)
    sparse      - Whether or not to store Taylor coefficients in
                  a sparse array (Default=False)
    dims        - The dimensions of a matrix with a length of
                  'max_o+1' along 'num_vars' dimensions
    sz          - The size of the above mentioned matrix
    counter_map - Array mapping the index of T to the
                  corresponding derivative
    index_map   - Dictionary mapping a derivative to the
                  corresponding index in T
    """

    def __init__(self,val,var_num,num_vars,max_o,\
                 model=None,T=None,dtype=complex,sparse=False):
        """
        Arguments :
        val      - Expression value
        var_num  - Number of variable being initialized
        num_vars - Maximum number of variables
        max_o    - Maximum order of derivatives
        model    - Another AD object with the same 'num_vars'
                   and 'max_o'. This is used to avoid creating
                   the mapping array and dictionary more than once.
        T        - Array of Taylor coefficients. By default
                   this is generated.
        dtype    - Data type used to store Taylor coefficients.
                   By default complex to account for all ranges
                   of values. For max_o>1 it is necessary
                   to use the complex dtype to avoid errors.
        sparse   - If true, stores the Taylor coefficients
                   as a sparse matrix.
        """

        # Initialize variables
        self.ADvector = True
        if val.__class__.__name__ == 'list':
            self.val = np.array(val,dtype=dtype)
        elif val.__class__.__name__ == 'ndarray':
            self.val = val.astype(dtype)
        else:
            self.val = dtype(val)
            self.ADvector = False

        self.num_vars = num_vars
        self.max_o    = max_o
        self.N        = 1 if not self.ADvector else len(val)
        self.dtype    = dtype
        self.sparse   = sparse

        # Initialize T
        if T != None:
            self.T = T
        elif self.ADvector:
            N = self.N
            rowlen = AD.num_derivatives(num_vars,max_o)
            rowlen = rowlen + 1 if (self.max_o != 1) else rowlen
            self.T = ss.lil_matrix((N,rowlen),dtype=dtype)
        else:
            rowlen = AD.num_derivatives(num_vars,max_o)
            rowlen = rowlen + 1 if (self.max_o != 1) else rowlen
            self.T = ss.lil_matrix((1,rowlen),dtype=dtype)

        # Create mapping functions if max_o > 1
        if self.max_o != 1:
            # Dimensions and size of original matrix
            # (only necessary for max_o>1)
            self.dims = tuple((max_o+1)*np.ones(num_vars,dtype=int))
            self.sz   = (max_o+1)**num_vars

            if model != None:
                self.counter_map = model.counter_map
                self.index_map = model.index_map
            else:
                # Maps T index (counter) to derivative index
                csize = self.T.shape[0]*self.T.shape[1]
                self.counter_map = np.zeros(csize,dtype=int)
                # Maps derivative index to T index
                self.index_map = {}
                counter = 0
                for i in xrange(0,self.sz):
                    k = np.sum(ui(i,self.dims))
                    if k <= self.max_o:
                        self.index_map[i] = counter
                        self.counter_map[counter] = i
                        counter += 1

        # Set T[0] value equal to val
        if max_o != 1:
            if self.ADvector and T==None:
                #self.T[:,0] = val
                for i in xrange(len(val)):
                    self.T[i,0] = val[i]
            elif T==None:
                self.T[0,0] = val

        # Set T values
        isVarInit = np.prod(var_num != 0) and (T==None)
        if self.max_o == 1 and isVarInit:
            if self.ADvector:
                #self.T[range(0,len(val)),var_num-1] = 1
                for i in xrange(len(val)):
                    self.T[i,var_num[i]-1] = 1
            else:
                self.T[0,var_num-1] = 1
        elif isVarInit:
            if self.ADvector:
                i = 0
                for num in var_num:
```

```python
                    index = np.zeros(num_vars,dtype=np.int)
                    index[num-1] = 1
                    index = rmi(index,self.dims)
                    index = self.index_map[index]
                    self.T[i,index] = 1
                    i += 1
            else:
                index = np.zeros(num_vars,dtype=np.int)
                index[var_num-1] = 1
                index = rmi(index,self.dims)
                index = self.index_map[index]
                self.T[0,index] = 1

        # Convert to sparse matrix (only supported for max_o=1)
        if self.sparse and self.max_o == 1:
            if T == None:
                self.T = smat(self.T)
        else:
            self.sparse = False
            if self.ADvector and T==None:
                self.T = self.T.toarray()
            elif T==None:
                self.T = self.T.toarray()[0]


    # Getitem - used for getting certain variables if ADvector=True
    def __getitem__(self,index):
        if self.sparse:
            T = self.T[index,:]
        else:
            T = self.T[index,:]
        val = self.val[index]
        return AD(val,0,self.num_vars,self.max_o,model=self,\
                T=T,dtype=self.dtype,sparse=self.sparse)

    # Setitem - used for setting certain variables if ADvector=True
    def __setitem__(self,keys,value):
        self.val[keys] = value.val
        if self.sparse:
            # Necessary for the keys in key to be sorted
            blocks = []
            prev = -1
            counter = 0
            for key in keys:
                if key == (prev+1):
                    blocks.append(value.T[counter,:])
                else:
                    blocks.append(self.T[(prev+1):key,:])
                    blocks.append(value.T[counter,:])
                prev = key
                counter += 1

            # Add final entries
            if prev != (self.N-1):
                blocks.append(self.T[prev+1:,:])

            # Set T to new sparse matrix
            self.T = ss.vstack(blocks,format='csr')

        else:
            self.T[keys,:] = value.T

    # Overload negation (-self)
    def __neg__(self):
        u = self
        T = -u.T
        h = AD(-u.val,0,u.num_vars,u.max_o,model=u,\
                T=T,dtype=u.dtype,sparse=self.sparse)
        return h

    # Overload positive (+self)
    def __pos__(self):
        u = self
        T = u.T.copy() if u.sparse else np.copy(u.T)
        h = AD(u.val,0,u.num_vars,u.max_o,model=u,\
            T=T,dtype=u.dtype,sparse=self.sparse)
        return h

    # Overload addition (self+v)
    def __add__(self,v):
        u = self
        if v.__class__.__name__ != 'AD':
            T = u.T.copy() if u.sparse else np.copy(u.T)
            val = v+u.val
            if self.max_o != 1:
                if self.ADvector:
                    T[:,0] = val
                else:
                    T[0] = val
            h = AD(val,0,u.num_vars,u.max_o,model=u,\
                T=T,dtype=u.dtype,sparse=self.sparse)
        else:
            if self.sparse:
                if u.N == v.N:
                    T = u.T+v.T
                elif u.N == 1:
                    T = AD.csr_add_sparse_vec(v.T,u.T)
                else:
                    T = AD.csr_add_sparse_vec(u.T,v.T)
            else:
                T = u.T+v.T
            h = AD(u.val+v.val,0,u.num_vars,u.max_o,\
                model=u,T=T,dtype=u.dtype,sparse=self.sparse)
        return h

    # Overload right-sided addition
    __radd__ = __add__

    # Overload subtraction (self-v)
    def __sub__(self,v):
        u = self
        if v.__class__.__name__ != 'AD':
            T = smat(u.T,copy=True) if u.sparse else np.copy(u.T)
            val = u.val-v
            if self.max_o != 1:
                if u.ADvector:
                    T[:,0] = val
                else:
                    T[0] = val
            h = AD(val,0,u.num_vars,u.max_o,model=u,\
                    T=T,dtype=u.dtype,sparse=self.sparse)
        else:
            if self.sparse:
                if u.N == v.N:
                    T = u.T-v.T
                elif u.N == 1:
                    T = AD.csr_add_sparse_vec(-v.T,u.T)
                else:
                    T = AD.csr_add_sparse_vec(u.T,-v.T)
            else:
                T = u.T-v.T
            h = AD(u.val-v.val,0,u.num_vars,u.max_o,model=u,\
                    T=T,dtype=u.dtype,sparse=self.sparse)
        return h

    # Overload right-sided subtraction (v-self). This is only
    # called when v is not an instance of AD
    def __rsub__(self,v):
        u = self
        T = -u.T
        val = v-u.val
        if self.max_o != 1:
            if u.ADvector:
                T[:,0] = val
            else:
                T[0] = val
        h = AD(val,0,u.num_vars,u.max_o,model=u,\
                T=T,dtype=u.dtype,sparse=self.sparse)

        return h

    # Overload multiplication operator (self*v)
    def __mul__(self,v):
        u = self
        if v.__class__.__name__ != 'AD':
            T = v*u.T
            h = AD(u.val*v,0,u.num_vars,u.max_o,model=u,\
                    T=T,dtype=u.dtype,sparse=self.sparse)
        else:
            # ------------------------------------------------------------
            if u.max_o == 1:
                if u.sparse and (u.ADvector or v.ADvector):
                    uval = ss.spdiags(u.val,0,u.N,u.N, \
                            format='csr') if u.ADvector else u.val
                    vval = ss.spdiags(v.val,0,v.N,v.N,\
                            format='csr') if v.ADvector else v.val
                    val = u.val*v.val
                    if u.ADvector and v.ADvector:
                        T = uval*v.T + vval*u.T
                    elif u.ADvector:
                        tmp = (smat(u.val).transpose())*v.T
                        T = tmp + vval*u.T
                    else:
                        tmp = (smat(v.val).transpose())*u.T
                        T = uval*v.T + tmp

                    h = AD(val,0,u.num_vars,u.max_o,model=u,\
                        T=T,dtype=u.dtype,sparse=self.sparse)
                elif u.ADvector or v.ADvector:
                    uval = u.val[:,None] if u.ADvector else u.val
                    vval = v.val[:,None] if v.ADvector else v.val
                    T = uval*v.T + u.T*vval
                    val = u.val*v.val
                    h = AD(val,0,u.num_vars,u.max_o,model=u,\
                            T=T,dtype=u.dtype,sparse=self.sparse)
                else:
                    T = u.val*v.T + u.T*v.val
                    val = u.val*v.val
                    h = AD(val,0,u.num_vars,u.max_o,model=u,\
                            T=T,dtype=u.dtype,sparse=self.sparse)
                return h
            #-----------------------------------------------
            h = AD(u.val*v.val,0,u.num_vars,u.max_o,\
                    model=u,dtype=u.dtype,sparse=self.sparse)
            N = u.T.shape[1] if u.ADvector else np.size(u.T)
            for i in xrange(1,N):
                # Get index corresponding to Taylor index i
                index = ui(self.counter_map[i],self.dims)
                k = np.array(index)
                if u.ADvector or v.ADvector:
                    h.T[:,i] = AD.bdot(u,k,v,k,0)
                else:
                    h.T[i] = AD.bdot(u,k,v,k,0)
        return h

    #  Overload right-handed multiplication operator
    __rmul__ = __mul__

    # Overload division operator (self/v)
    def __div__(self,v):
        u = self
        if v.__class__.__name__ != 'AD':
            T = u.T/v
            h = AD(u.val/v,0,u.num_vars,u.max_o,\
            model=u,T=T,dtype=u.dtype,sparse=self.sparse)
        else:
            #-----------------------------------------------
            if u.max_o == 1:
                if u.sparse and (u.ADvector or v.ADvector):
                    val = u.val/v.val
                    if u.ADvector and v.ADvector:
                        M1 = ss.spdiags(1.0/v.val,0,v.N,\
                                    v.N,format='csr')
                        M2 = ss.spdiags(u.val/(v.val**2),0,v.N,\
                                    v.N,format='csr')
                        T = (M1*u.T) - (M2*v.T)
                    elif u.Advector:
                        M = ss.spdiags(u.val/(v.val**2,0),v.N,\
                                        v.N,format='csr')
                        T = u.T/v.val - M*ss.vstack(u.N*[v.T],\
                                        format='csr')
                    else:
                        M1 = ss.spdiags(1.0/v.val,0,v.N,v.N,\
                                format='csr')
                        M2 = ss.spdiags(u.val/(v.val**2),0, \
                                v.N,v.N,format='csr')
                        T = (M1*ss.vstack(v.N*[u.T],format='csr'))\
                             - (M2*v.T)

                    h = AD(val,0,u.num_vars,u.max_o,\
                    model=u,T=T,dtype=u.dtype,sparse=self.sparse)
                elif u.ADvector or v.ADvector:
                    uval = u.val[:,None] if u.ADvector else u.val
                    vval = v.val[:,None] if v.ADvector else v.val
                    T = (u.T/vval)-(v.T*uval/(vval**2))
                    h = AD(u.val/v.val,0,v.num_vars,u.max_o,\
                    model=u,T=T,dtype=u.dtype,sparse=self.sparse)
                else:
                    T = (u.T/v.val)-(v.T*u.val/(v.val**2))
                    h = AD(u.val/v.val,0,v.num_vars,u.max_o,\
                    model=u,T=T,dtype=u.dtype,sparse=self.sparse)

                return h
            # ---------------------------------------------------------
            h = AD(u.val/v.val,0,v.num_vars,u.max_o,\
            model=u,dtype=u.dtype,sparse=self.sparse)
            N = u.T.shape[1] if u.ADvector else np.size(u.T)
            for i in xrange(1,N):
                index = ui(self.counter_map[i],self.dims)
                k = np.array(index)
                if u.ADvector:
                    h.T[:,i] = (u.T[:,i]-AD.bdot(h,k,v,k,0))/v.val
                elif v.ADvector:
                    h.T[:,i] = (u.T[i]-AD.bdot(h,k,v,k,0))/v.val
                else:
                    h.T[i] = (u.T[i]-AD.bdot(h,k,v,k,0))/v.val

        return h

    # Overload right-sided division (Called for v/self) with v!='AD'
    def __rdiv__(self,v):
        u = self

        # ---------------------------------------------------------
        if u.max_o == 1:
            if u.ADvector and u.sparse:
                M = ss.spdiags(-v/u.val**2,0,u.N,u.N,\
                                 format='csr')
                T = M*u.T
            elif u.ADvector:
                T = -v*u.T/(u.val**2)
            else:
                T = -v*u.T/(u.val**2)

            h = AD(v/u.val,0,u.num_vars,u.max_o,model=u,\
                    T=T,dtype=u.dtype,sparse=self.sparse)
            return h
        # ---------------------------------------------------------

        # Create h
        h = AD(v/u.val,0,u.num_vars,u.max_o,\
                model=u,dtype=u.dtype,sparse=self.sparse)

        # Create Taylor coefficient matrix for v
        v = AD(v,0,u.num_vars,u.max_o,\
                model=u,dtype=u.dtype,sparse=self.sparse)

        N = u.T.shape[1] if u.ADvector else np.size(u.T)
        for i in xrange(1,N):
            index = ui(self.counter_map[i],self.dims)
            k = np.array(index)
            if u.ADvector:
                h.T[:,i] = (v.T[i]-AD.bdot(h,k,u,k,0))/u.val
            else:
                h.T[i] = (v.T[i]-AD.bdot(h,k,u,k,0))/u.val

        return h

    # Overload power operator (Called for self^v)
    def __pow__(self,v):
        u = self
        h = mathADI.exp(v*mathADI.log(u))
        return h

    # Overload right-sided power operator (Called for v^self)
    def __rpow__(self,v):
        u = self
        h = mathADI.exp(u*mathADI.log(v))
        return h


    """
    Begin helper methods :
    ----------------------------------------------------------------------
    """
    # Returns the Taylor coefficients as a matrix
    def get_T_matrix(self):
        h = np.zeros(self.dims,dtype=self.dtype)

        if self.max_o == 1:
            for i in xrange(1,self.T.size):
                current = np.zeros(self.num_vars,dtype=int)
                current[i-1] = 1
                h[tuple(current)] = T[i]
                return h

        for i in xrange(1,np.size(self.T)):
            h.ravel()[self.counter_map[i]] = self.T[i]
        h.ravel()[0] = self.val
        return h

    # Returns the derivatives as a matrix
    def get_derivatives(self):
        u = self
        h = u.get_T_matrix()
        if self.max_o == 1:
            return h

        dims = h.shape
        for i in xrange(np.size(h)):
            # Get subscript
            index = ui(i,dims)

            for k in xrange(np.size(index)):
                    h[index] = h[index] * math.factorial(index[k])

            return h

    # Helper method for getting the Jacobian matrix/vector
    def get_jacobian(self):
        if self.max_o == 1:
            return self.T

        # Initialize variables
        num_vars = self.num_vars
        N = self.N
        h = np.zeros(N*num_vars,dtype=complex)
        h.shape = (N,num_vars)

        for i in xrange(N):
            for j in xrange(num_vars):
                # Get derivative vector
                der = np.zeros(num_vars,dtype=int)
                der[j] = 1

                # Set derivative element
                h[i,j] = self.get_tcof(der,var=i)

        # Return Jacobian
        return h[0] if self.N==1 else h

    # Helper method for getting the Hessian matrix
    def get_hessian(self):
        if self.N > 1:
            # Not supported for more than 1 variable
            return None

        # Initialize matrix
        num_vars = self.num_vars
        h = np.zeros(num_vars*num_vars,dtype=complex)
        h.shape = (num_vars,num_vars)

        # Get derivatives
        for i in xrange(num_vars):
            tmp = np.zeros(num_vars,dtype=int)
            tmp[i] += 1
            for j in xrange(num_vars):
                der = tmp.copy()
                der[j] += 1
                h[i,j] = 2*self.get_tcof(der) if i == j \
                              else self.get_tcof(der)

        # Return Hessian
        return h

    # Helper method for getting a single Taylor coefficient
    def get_tcof(self,der, var=0):
        lin = rmi(tuple(der),self.dims)
        index = self.index_map[lin]
        if self.ADvector:
            return self.T[var,index]
        return self.T[index]

    # Helper method for getting a single derivative
    def get_derivative(self,der,var=0):
        der = self.get_single_T(der,var)
        for i in der:
            der *= math.factorial(i)
        return der

    # Method for setting the 'val' attribute of an AD instance
    def setval(self,val):
        self.val = val
        if self.max_o != 1:
            if self.ADvector:
                self.T[:,0] = val
            else:
                self.T[0] = val

    """
    Begin Static helper methods :
    -------------------------------------------------------------
    """

    """
    Originally from :
    http://stackoverflow.com/questions/15239491/
    adding-a-very-repetitive-matrix-to-a-sparse-one-in-numpy-scipy
    -------------------------------------------------------------
    Adds a sparse vector to every row of a sparse matrix
    """
    @staticmethod
    def csr_add_sparse_vec(M, v) :
        rows, cols = M.shape

        new_data = M.data
        new_pointer = M.indptr.copy()
        new_cols = M.indices

        aux_idx = np.arange(rows + 1)

        for value, col in itertools.izip(v.data, v.indices):
            new_data = np.insert(new_data, new_pointer[1:], \
                                 [value] * rows)
            new_cols = np.insert(new_cols, new_pointer[1:], \
                                 [col] * rows)
            new_pointer += aux_idx

        return ss.csr_matrix((new_data, new_cols, new_pointer),
                             shape=M.shape)


    """
    Method for multiplying ADvector with a vector of numbers, where
    vec[i] is multiplied by the variable corresponding to T[i,:]
    """
    @staticmethod
    def vec_mult(vec,u):
        if u.__class__.__name__ == 'ndarray':
            return vec*u
        if u.sparse:
            M = ss.spdiags(vec,0,u.N,u.N,format='csr')
            T = M*u.T
            val = vec*u.val
        else:
            T = u.T*vec[:,None]
            val = vec*u.val
        return AD(val,0,u.num_vars,u.max_o,model=u,\
                T=T,dtype=u.dtype,sparse=u.sparse)

    """
    Method for right-side-multiplying ADvector with a sparse matrix
    of numbers, where the AD variables are distributed along the
    matrix like a vector of single AD instances.
    """
    @staticmethod
    def smat_mult(M,u):
        if u.sparse:
            T = M*u.T
            val = M.dot(u.val)
        else:
            Tsparse = smat(u.T)
            T = np.asarray((M*Tsparse).todense(),dtype=u.dtype)
            val = M.dot(u.val)

        return AD(val,0,u.num_vars,u.max_o,model=u,\
                    T=T,dtype=u.dtype,sparse=u.sparse)

    """
    Method for summing all variables in an ADvector together
    """
    @staticmethod
    def ADsum(u):
        val = np.sum(u.val)
        if u.sparse:
            T = np.sum(u.T.todense(),axis=0)
            T = smat(T)
        else:
            T = np.sum(u.T,axis=0)
        return AD(val,0,u.num_vars,u.max_o,model=u,\
                T=T,dtype=u.dtype,sparse=u.sparse)

    """
    Method for combining AD objects
    """
    @staticmethod
    def ADappend(u,AD_vars):
        # If matrix is sparse
        if u.sparse:
            blocks = [var.T for var in AD_vars]
            blocks.insert(0,u.T)
            T = ss.vstack(blocks,format='csr')
            val = u.val
            for var in AD_vars:
                val = np.hstack((val,var.val))

            return AD(val,0,u.num_vars,u.max_o,model=u,\
                    T=T,dtype=u.dtype,sparse=u.sparse)

        N = u.N+np.sum([entry.N for entry in AD_vars])
        val = np.zeros(N,dtype=u.dtype)
        T    = np.zeros(N*u.T.shape[1],dtype=u.dtype)
        T.shape = (N,u.T.shape[1])

        # Set u values
        val[:u.N] = u.val
        T[:u.N,:] = u.T.todense() if u.sparse else u.T

        # Add entries from AD_vars
        counter = u.N
        for entry in AD_vars:
            if entry.N == 1:
                val[counter] = entry.val
                T[counter,:] = entry.T.todense() \
                            if entry.sparse else entry.T
            else:
                val[counter:counter+entry.N] = entry.val
                T[counter:counter+entry.N,:] = entry.T.todense() \
                            if entry.sparse else entry.T
            counter += entry.N

        # Return new AD object
        T = smat(T) if u.sparse else T
        return AD(val,0,u.num_vars,u.max_o,model=u,\
                T=T,dtype=u.dtype,sparse=u.sparse)

    @staticmethod
    def num_derivatives(n,max_o):
        """
        Parameters :
        n - Number of variables
        max_o - Maximum order of derivatives

        Return :
        d - Total number of derivatives up to order max_o
          - Calculated as sum of 'n+k-1 choose k' with k=1
          - up to max_o.
        """
        k = np.array(range(1,max_o+1))
        d = 0
        for entry in k:
            d += np.prod(range(n,n+entry))/factorial(entry)
        return int(d)


    @staticmethod
    def get_one_order_vec(k):
        """
        Returns a one order vector, sum(e) = 1, that
        is less than or equal to k.
        """
        if np.sum(k)==1:
            return k
        else:
            e = np.zeros(len(k))
            i = 0
            while np.sum(e)==0 and i<len(k):
                e[i] = k[i]>0
                i += 1
        return e

    @staticmethod
    def vec_le(u,v):
        tmp = u <= v
        return np.prod(tmp)

    @staticmethod
    def coef(k,current,e):
        c = 1
        if np.sum(e)==1:
            i = np.nonzero(e)[0][0]
            c = (k[i]-current[i])/float(k[i])

        return c

    @staticmethod
    def bdot(P,m,Q,k,e):
        # Get number of elements and (theoretical)
        # dimensions of Taylor arrays
        N = P.T.shape[1] if P.ADvector else np.size(P.T)
        dims = P.dims

        # Value of calculation
        if P.ADvector:
            h = np.zeros(len(P.val),dtype=P.dtype)
        elif Q.ADvector:
            h = np.zeros(len(Q.val),dtype=P.dtype)
        else:
            h = 0

        # Calculate value
        for j in xrange(0,N):
            # Get current vector
            current = np.array(ui(P.counter_map[j],dims))

            # If current is lte m
            if AD.vec_le(current,m):
                P_index = P.index_map[rmi(current,dims)]
                Q_index = P.index_map[rmi(k-current,dims)]

                if P.ADvector and Q.ADvector:
                    # P and Q are both AD vectors
                    term = P.T[:,P_index]*Q.T[:,Q_index]\
                                    *AD.coef(k,current,e)
                elif P.ADvector:
                    # Only P is an AD vector
                    term = P.T[:,P_index]*Q.T[Q_index]\
                                    *AD.coef(k,current,e)
                elif Q.ADvector:
                    # Only Q is an AD vector
                    term = P.T[P_index]*Q.T[:,Q_index]\
                                *AD.coef(k,current,e)
                else:
                    # Neither P or Q are AD vectors
                    term = P.T[P_index]*Q.T[Q_index]\
                            *AD.coef(k,current,e)

                # Add to sum
                h += term
        return h
```

A.2.2 Custom decorator

When a decorator is applied to a function the result is that whenever the function is called we instead call the decorator. Once there it is possible to call the original function, though it's not necessary, and any other calculations or function calls can be made instead. The decorator 'adec' is applied to all the mathematical functions in the mathADI library and includes a series of tests that are common to all the functions. The Python code follows below.

```
import numpy as np
import adipy

"""
Decorator used for the additional functions in adipy.
"""
class adec:
    def __init__(self,f):
        # Save function
        self.f = f

        # Save pointer to fallback function
        if f.__name__ == 'exp':
            self.fallback = np.exp
        elif f.__name__ == 'log':
            self.fallback = np.log
        elif f.__name__ == 'sqrt':
            self.fallback = np.sqrt
        elif f.__name__ == 'arctan':
            self.fallback = np.arctan
        elif f.__name__ == 'arcsin':
            self.fallback = np.arcsin
        elif f.__name__ == 'arccos':
            self.fallback = np.arccos
        elif f.__name__ == 'cos':
            self.fallback = np.cos
        elif f.__name__ == 'sin':
            self.fallback = np.sin
        elif f.__name__ == 'tan':
            self.fallback = np.tan
        elif f.__name__ == 'cot':
            #self.fallback = np.cot
            self.fallback = lambda u: 1.0/np.tan(u)
        elif f.__name__ == 'sec':
            #self.fallback = np.sec
            self.fallback = lambda u: 1.0/np.cos(u)
        elif f.__name__ == 'csc':
            #self.fallback = np.csc
            self.fallback = lambda u: 1.0/np.sin(u)
        elif f.__name__ == 'sinh':
            self.fallback = np.sinh
        elif f.__name__ == 'cosh':
            self.fallback = np.cosh
```

```python

    def __call__(self,u):
        if u.__class__.__name__ == 'AD':
            return self.f(u)
        else:
            # u is not an AD instance - call fallback function
            return self.fallback(u)
```

When any of the functions are called we enter the __call__ method of the decorator. This shows that if the input is an AD instance, the original function is called right away. If it's not it is assumed that it's a number and the corresponding numpy function is called instead. This makes it possible to simply import the mathADI functions and use them for calculations on both the AD instances and regular numbers.

Using a decorator on the functions instead of writing out the same tests for all of them makes them shorter and more concise. Additionally this means that we can assume that all input being sent to functions are instances of the AD class.

A.2.3 Math Library

The code for the file mathADI.py with the implemenations of all functions for the AD class follows below.

```python
import numpy as np
import adipy
from adec import adec
import scipy.sparse as ss
from scipy.sparse import csr_matrix as smat

"""
Mathematical functions for operating on AD instances
--------------------------------------------------------------------
"""
@adec
def exp(u):
    # ----------------------------------------------------------
    if u.max_o == 1:
        val = np.exp(u.val)
        if u.ADvector and u.sparse:
            M = ss.spdiags(val,0,u.N,u.N,format='csr')
            T = M*u.T
        elif u.ADvector:
            T = val[:,None]*u.T
        else:
            T = val*u.T
        h = adipy.AD(val,0,u.num_vars,u.max_o,\
                T=T,model=u,dtype=u.dtype,sparse=u.sparse)

        return h
    # ----------------------------------------------------------

    h = adipy.AD(np.exp(u.val),0,u.num_vars,u.max_o,\
            model=u,dtype=u.dtype,sparse=u.sparse)
    N = u.T.shape[1] if u.ADvector else np.size(u.T)
    for i in xrange(1,N):
        index = np.unravel_index(u.counter_map[i],u.dims)
        k = np.array(index)
        e = adipy.AD.get_one_order_vec(k)
        if u.ADvector:
            h.T[:,i] = adipy.AD.bdot(h,k-e,u,k,e)
        else:
            h.T[i] = adipy.AD.bdot(h,k-e,u,k,e)

    return h

@adec
def log(u):
    # ----------------------------------------------------------------
    if u.max_o == 1:
        if u.ADvector and u.sparse:
            M = ss.spdiags(1.0/u.val,u.N,u.N,format='csr')
            T = M*u.T
        elif u.ADvector:
            T = u.T/u.val[:,None]
        else:
            T = u.T/u.val

        h = adipy.AD(np.log(u.val),0,u.num_vars,u.max_o,\
                T=T,model=u,dtype=u.dtype,sparse=u.sparse)
        return h
    # ----------------------------------------------------------------

    h = adipy.AD(np.log(u.val),0,u.num_vars,u.max_o,\
            model=u,dtype=u.dtype,sparse=u.sparse)
    N = u.T.shape[1] if u.ADvector else np.size(u.T)
    for i in xrange(1,N):
        index = np.unravel_index(u.counter_map[i],u.dims)
        k = np.array(index)
        e = adipy.AD.get_one_order_vec(k)
        if u.ADvector:
            h.T[:,i] = (u.T[:,i]-adipy.AD.bdot(u,k-e,h,k,e))/u.val
        else:
            h.T[i] = (u.T[i]-adipy.AD.bdot(u,k-e,h,k,e))/u.val

    return h

@adec
def sqrt(u):
    # ----------------------------------------------------------------
    if u.max_o == 1:
        val = np.sqrt(u.val)
        if u.ADvector and u.sparse:
            M = ss.spdiags(1.0/(2*val),u.N,u.N,format='csr')
            T = M*u.T
        elif u.ADvector:
            h.T = u.T/(2*val[:,None])
        else:
            T = u.T/(2*val)

        h = adipy.AD(val,0,u.num_vars,u.max_o,\
                T=T,model=u,dtype=u.dtype,sparse=u.sparse)
        return h
    # ---------------------------------------------------------------
    h = adipy.AD(np.sqrt(u.val),0,u.num_vars,u.max_o,\
                model=u,dtype=u.dtype,sparse=u.sparse)
    N = u.T.shape[1] if u.ADvector else np.size(u.T)
    for i in xrange(1,N):
        index = np.unravel_index(u.counter_map[i],u.dims)
        k = np.array(index)
        e = adipy.AD.get_one_order_vec(k)
        if u.ADvector:
            h.T[:,i] = (0.5*u.T[:,i]-adipy.AD.bdot(h,k-e,h,k,e))/h.val
        else:
            h.T[i] = (0.5*u.T[i]-adipy.AD.bdot(h,k-e,h,k,e))/h.val

    return h

@adec
def arctan(u):
    # ---------------------------------------------------------------
    if u.max_o == 1:
        if u.ADvector and u.sparse:
            M = ss.spdiags(1.0/(1.0 + u.val**2),\
                            u.N,u.N,format='csr')
            T = M*u.T
        elif u.ADvector:
            T = u.T/(1+u.val[:,None]**2)
        else:
            T = u.T/(1+u.val**2)

        h = adipy.AD(np.arctan(u.val),0,u.num_vars,u.max_o,\
                T=T,model=u,dtype=u.dtype,sparse=u.sparse)
        return h
    # ---------------------------------------------------------------
    v = 1.0/(1.0+u*u) # Helper variable
    h = adipy.AD(np.arctan(u.val),0,u.num_vars,u.max_o,\
                model=u,dtype=u.dtype,sparse=u.sparse)
    N = u.T.shape[1] if u.ADvector else np.size(u.T)
    for i in xrange(1,N):
        index = np.unravel_index(u.counter_map[i],u.dims)
        k = np.array(index)
        e = adipy.AD.get_one_order_vec(k)
        if u.ADvector:
            h.T[:,i] = adipy.AD.bdot(v,k-e,u,k,e)
        else:
            h.T[i] = adipy.AD.bdot(v,k-e,u,k,e)

    return h

@adec
def arcsin(u):
    # ---------------------------------------------------------------
    if u.max_o == 1:
        if u.ADvector and u.sparse:
            M = ss.spdiags(1.0/np.sqrt(1.0-u.val**2),\
                            u.N,u.N,format='csr')
            T = M*u.T
        elif u.ADvector:
            T = u.T/np.sqrt(1-u.val[:,None]**2)
        else:
            T = u.T/np.sqrt(1-u.val**2)

            h = adipy.AD(np.arcsin(u.val),0,u.num_vars,u.max_o,\
                    T=T,model=u,dtype=u.dtype,sparse=u.sparse)
            return h
        # ------------------------------------------------------------
        v = 1.0/sqrt(1.0-u*u) # Helper variable
        h = adipy.AD(np.arcsin(u.val),0,u.num_vars,u.max_o,\
                model=u,dtype=u.dtype,sparse=u.sparse)
        N = u.T.shape[1] if u.ADvector else np.size(u.T)
        for i in xrange(1,N):
            index = np.unravel_index(u.counter_map[i],u.dims)
            k = np.array(index)
            e = adipy.AD.get_one_order_vec(k)
            if u.ADvector:
                h.T[:,i] = adipy.AD.bdot(v,k-e,u,k,e)
            else:
                h.T[i] = adipy.AD.bdot(v,k-e,u,k,e)

        return h

@adec
def arccos(u):
        # ------------------------------------------------------------
        if u.max_o == 1:
            if u.ADvector and u.sparse:
                M = ss.spdiags(-1.0/np.sqrt(1-u.val**2),\
                                u.N,u.N,format='csr')
                T = M*u.T
            elif u.ADvector:
                T = -u.T/np.sqrt(1-u.val[:,None]**2)
            else:
                T = -u.T/np.sqrt(1-u.val**2)

            h = adipy.AD(np.arccos(u.val),0,u.num_vars,u.max_o,\
                T=T,model=u,dtype=u.dtype,sparse=u.sparse)
            return h
        # ------------------------------------------------------------
        v = -1.0/adipy.AD.sqrt(1.0-u*u) # Helper variable
        h = adipy.AD(np.arccos(u.val),0,u.num_vars,u.max_o,\
                model=u,dtype=u.dtype,sparse=u.sparse)
        N = u.T.shape[1] if u.ADvector else np.size(u.T)
        for i in xrange(1,N):
            index = np.unravel_index(u.counter_map[i],u.dims)
            k = np.array(index)
            e = adipy.AD.get_one_order_vec(k)
            if u.ADvector:
                h.T[:,i] = adipy.AD.bdot(v,k-e,u,k,e)
            else:
                h.T[i] = adipy.AD.bdot(v,k-e,u,k,e)
        return h

# Private method used for calculating sin and cos
def __sincos(u):
        # ------------------------------------------------------------
        if u.max_o == 1:
            if u.ADvector and u.sparse:
                Msin = ss.spdiags(np.cos(u.val),u.N,u.N,format='csr')
                Mcos = ss.spdiags(-np.sin(u.val),u.N,u.N,format='csr')
                sinuT = Msin*u.T
                cosuT = Mcos*u.T
            elif u.ADvector:
                sinuT = u.T*np.cos(u.val[:,None])
                cosuT = -u.T*np.sin(u.val[:,None])
            else:
                sinuT = u.T*np.cos(u.val)
                cosuT = -u.T*np.sin(u.val)

            sinu = adipy.AD(np.sin(u.val),0,u.num_vars,u.max_o,\
                    T=sinuT,model=u,dtype=u.dtype,sparse=u.sparse)
            cosu = adipy.AD(np.cos(u.val),0,u.num_vars,u.max_o,\
                    T=cosuT,model=u,dtype=u.dtype,sparse=u.sparse)
            return sinu,cosu
        # ---------------------------------------------------------
        sinu = adipy.AD(np.sin(u.val),0,u.num_vars,u.max_o,\
                model=u,dtype=u.dtype,sparse=u.sparse)
        cosu = adipy.AD(np.cos(u.val),0,u.num_vars,u.max_o,\
                model=u,dtype=u.dtype,sparse=u.sparse)
        N = u.T.shape[1] if u.ADvector else np.size(u.T)
        for i in xrange(1,N):
            index = np.unravel_index(u.counter_map[i],u.dims)
            k = np.array(index)
            e = adipy.AD.get_one_order_vec(k)
            if u.ADvector:
                sinu.T[:,i] = adipy.AD.bdot(cosu,k-e,u,k,e)
                cosu.T[:,i] = -adipy.AD.bdot(sinu,k-e,u,k,e)
            else:
                sinu.T[i] = adipy.AD.bdot(cosu,k-e,u,k,e)
                cosu.T[i] = -adipy.AD.bdot(sinu,k-e,u,k,e)

        return sinu,cosu

@adec
def sin(u):
    h = __sincos(u)[0]
    return h

@adec
def cos(u):
    h = __sincos(u)[1]
    return h

@adec
def tan(u):
    sinu,cosu = __sincos(u)
    h = sinu / cosu
    return h

@adec
def cot(u):
    sinu,cosu = __sincos(u)
    h = cosu / sinu
    return h

@adec
def sec(u):
    cosu=__sincos(u)[1]
    h = 1.0 / cosu
    return h
```

```

@adec
def csc(u):
    sinu=__sincos(u)[0]
    h = 1.0 / sinu
    return h

@adec
def sinh(u):
    h = 0.5*(exp(u)-exp(-u))
    return h

@adec
def cosh(u):
    h = 0.5(exp(u)+exp(-u))
    return h
```

A.2.4 initADI

The file initADI with the function init_variables follows below.

```
from adipy import AD
import numpy as np

"""
Method for initializing variables
 - *args holds the values of the variables
 - *kwargs holds the keyword arguments:
        max_o, update_num, dtype, sparse
"""
def init_variables(*args,**kwargs):
    # Get keyword arguments
    max_o = 1
    update_num = False
    dtype = complex
    sparse = False
    for k,v in kwargs.iteritems():
        if k=='max_o':
            max_o = v
        if k=='update_num':
            update_num = v
        if k=='dtype':
            dtype = v
        if k=='sparse':
            sparse = v

    # Initialize variables
    adVars   = []
    var_num  = 1
    num_vars = len(args) if not update_num \
               else np.sum([np.size(arg) for arg in args])

    # Create AD instances
    for (counter,val) in enumerate(args):
        isNumber = isinstance(val,(int,long,float,complex))
        model = None if counter == 0 else adVars[-1]
        # Set the var_nums variable and update var counter
```

```python
        if update_num and (not isNumber):
            var_nums = np.array(range(var_num,var_num+len(val)),\
                                dtype=int)
            var_num = var_nums[-1]+1
        elif not isNumber:
            var_nums = var_num*np.ones(len(val),dtype=int)
            var_num += 1
        else:
            var_nums = var_num
            var_num += 1

        # Create the AD object
        var = AD(val,var_nums,num_vars,max_o,model=model,\
                    dtype=dtype,sparse=sparse)

        # Add latest variable
        adVars.append(var)


    if len(adVars) == 1:
        return adVars[0]
    return adVars
```

Appendix B

Testing Framework

During the development of the AD framework a series of tests were created to ensure that the framework worked as expected. This was particularly useful when making changes to the code to improve efficiency, since the tests could be run quickly to ensure no functionality was broken. Each test is just a simple function that calculates a set of derivatives and proceeds to check that these are equal to the exact ones. This appendix will list all the files that inlude these tests.

The file test_log.py follows below.

```
import sys
sys.path.append('..') # Add above folder to path to import functions

from adipy import AD
from mathADI import log,exp
from numpy import pi
import nose.tools as nt
import numpy as np

def test_log_1():
    x = AD(3,1,3,3)
    y = AD(2,2,3,3)
    z = AD(1,3,3,3)
    g = log(13*y)*exp(log(2*x*z))
    actSolution = g.get_derivatives()

    # Calculate real solution
    expSolution = np.zeros((4,4,4))

    # expSolution(:,:,0)
    tmp1 = np.array(
            [log(13*2)*2*3, (2*3)/2.0, -(2*3)/(2.0**2), (4*3)/(2.0**3), \
             log(13*2)*2,   1,         -0.5,            0, \
             0,             0,         0,               0, \
             0,             0,         0,               0] \
            )
    tmp1.shape = (4,4)
    expSolution[:,:,0] = tmp1

    # expSolution(:,:,1)
    tmp2 = np.array(
                [log(13*2)*2*3, (2*3)/2.0,  -(2*3)/(2.0**2), 0, \
                 log(13*2)*2,   1,          0,               0, \
                 0,             0,          0,               0, \
                 0,             0,          0,               0] \
                )
    tmp2.shape = (4,4)
    expSolution[:,:,1] = tmp2

    # Check actual solution
    diff = np.max(np.abs(expSolution-actSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)
```

The file test_polynomials.py follows below.

```
import sys
sys.path.append('..') # Add above folder to path to import functions

from adipy import AD
from numpy import pi
import nose.tools as nt
import numpy as np


def test_polynomial_1():
    x = AD(3,1,2,2)
    y = AD(pi,2,2,2)
    z = (x**2)*(y**3)-x*y**2-x*y+y
    actSolution = z.get_derivatives()

    # Calculate real solution
    expSolution = np.array(\
                    [(3**2)*(pi**3)-3*pi**2-3*pi+pi, \
                     3*(3**2)*pi**2-2*3*pi-3+1, \
                     6*(3**2)*pi-2*3, \
                     2*3*(pi**3)-pi**2-pi,\
                     3*2*3*pi**2-2*pi-1,\
                     0, \
                     2*(pi**3), \
                     0, \
                     0])
    expSolution.shape = (3,3)

    # Check actual solution
    diff = np.max(np.abs(actSolution-expSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)

def test_polynomial_2():
    x = AD(3,1,2,4)
    y = AD(7,2,2,4)
    z = (x**2)*(y**3)
    actSolution = z.get_derivatives()

    # Calculate real solution
    expSolution = np.array( \
                  [(3**2)*(7**3), \
                   3*(3**2)*(7**2), \
                   6*(3**2)*7, 6*(3**2),\
                   0, \
                   2*3*(7**3), \
                   6*3*(7**2), \
                   12*3*7, \
                   12*3, \
                   0, \
                   2*(7**3),\
                   6*(7**2),\
                   12*7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] \
                  )
    expSolution.shape = (5,5)

    # Check actual solution
    diff = np.max(np.abs(actSolution-expSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)

def test_polynomial_3():
    x = AD(5,1,3,4)
    y = AD(3,2,3,4)
    z = AD(7,3,3,4)
    g = ((x+3)**2)*((z+2)**3)*(-1+y**2)/(((z+2)**3)*(y-1))

    # Get actual solution
    actSolution = g.get_derivatives()

    # Calculate real solution
    expSolution = np.zeros([5,5,5])
    tmp = np.array( \
          [(8**2)*4, 8**2, 0, 0, 0, \
           2*8*4,    2*8,  0, 0, 0, \
           2*4,      2,    0, 0, 0, \
           0,        0,    0, 0, 0, \
           0,        0,    0, 0, 0]
          )
    tmp.shape = (5,5)
    expSolution[:,:,0] = tmp

    # Check actual solution
    diff = np.max(np.max(actSolution-expSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)
```

The file test_trig.py follows below.

```
import sys
sys.path.append('..') # Add above folder to path to import functions
from adipy import AD
from mathADI import cos,sin
from numpy import pi
import nose.tools as nt
import numpy as np

def test_trig_1():
    # Find actual solution
    x = AD(pi,1,3,3)
    y = AD(pi/2,2,3,3)
    z = AD(pi/4,3,3,3)
    g = cos(2*x)*sin(y)*cos(z)
    actSolution = g.get_derivatives()

    # Calculate real solution
    expSolution = np.zeros((4,4,4))

    # expSolution[:,:,0]
    tmp1 = np.array(
            [cos(pi/4),     0, -cos(pi/4), 0, \
             0,             0, 0,          0, \
             -4*cos(pi/4),  0, 0,          0, \
             0,             0, 0,          0] \
            )
    tmp1.shape = (4,4)
    expSolution[:,:,0] = tmp1

    # expSolution[:,:,1]
    tmp2 = np.array(
            [-sin(pi/4),    0, cos(pi/4),  0, \
             0,             0, 0,          0, \
             4*sin(pi/4),   0, 0,          0, \
             0,             0, 0,          0] \
            )
    tmp2.shape = (4,4)
    expSolution[:,:,1] = tmp2

    # expSolution[:,:,2]
    expSolution[0,0,2] = -cos(pi/4)

    # expSolution[:,:,3]
    expSolution[0,0,3] = sin(pi/4)

    # Check actual solution
    diff = np.max(np.abs(expSolution-actSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)
```

The file test_vec_func.py follows below.

```
import sys
sys.path.append('..') # Add above folder to path to import functions

from adipy import AD
from mathADI import cos,sin,exp,log,sqrt
from initADI import init_variables
from numpy import pi
import nose.tools as nt
import numpy as np

# Test vector functionality
def test_vec_func_1():
    x = init_variables([pi,pi/2,-pi],max_o=2)
    z1 = cos(x)
    z2 = sin(x)
    z3 = exp(x)

    # Find actual and expected solution
    actSolution = np.array( \
                    [[z.get_derivatives() for z in z1], \
                     [z.get_derivatives() for z in z2], \
                     [z.get_derivatives() for z in z3]])
    expSolution = np.array([ \
                      [[cos(pi), 0, -cos(pi)], \
                       [0, -sin(pi/2), 0],\
                         [cos(-pi), 0, -cos(-pi)]], \
                        [[0,cos(pi),0], \
                         [sin(pi/2),0,-sin(pi/2)], \
                         [0,cos(-pi),0]], \
                        [[exp(pi),exp(pi),exp(pi)], \
                         [exp(pi/2),exp(pi/2),exp(pi/2)], \
                         [exp(-pi),exp(-pi),exp(-pi)]] \
                      ])

    # Test actual solution with expected solution
    diff = np.max(np.abs(expSolution-actSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)

# Test vector functionality
def test_vec_func_2():
    x,y = init_variables([pi,pi/2],3,max_o=2)
    z1 = cos(y)+3*(x**2)*y

    # Find  actual and expected solution
    actSolution = np.array([z.get_derivatives() for z in z1])
    expSolution = np.array([\
    [[cos(3)+3*(pi**2)*3,-sin(3)+3*(pi**2),-cos(3)],  \
     [6*pi*3, 6*pi, 0],  \
     [6*3, 0, 0]], \
    [[cos(3)+3*((0.5*pi)**2)*3,-sin(3)+3*((0.5*pi)**2),-cos(3)], \
     [6*0.5*pi*3, 3*pi, 0], \
     [6*3, 0, 0]]])

    # Test actual solution with expected solution
    diff = np.max(np.abs(expSolution-actSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)

# Test vector functionality
def test_vec_func_3():
    x,y = init_variables([pi,pi/2],[-1,2],max_o=2)
    z1 = cos(x)*sin(y)+(x**2)*(y**2)

    # Find actual and expted solution
    actSolution = np.array([z.get_derivatives() for z in z1])
    expSolution = np.array([\
    [[cos(pi)*sin(-1)+(pi**2)*((-1)**2),\
      cos(pi)*cos(-1)+(pi**2)*2*(-1),\
      cos(pi)*-sin(-1)+(pi**2)*2],  \
     [-sin(pi)*sin(-1)+2*pi*((-1)**2),\
      -sin(pi)*cos(-1)+4*pi*(-1),0], \
     [-cos(pi)*sin(-1)+2*((-1)**2),0,0]], \
    [[cos(pi/2)*sin(2)+((pi/2)**2)*(2**2),\
      cos((pi/2))*cos(2)+((pi/2)**2)*2*2,\
      cos((pi/2))*-sin(2)+((pi/2)**2)*2],  \
     [-sin((pi/2))*sin(2)+2*(pi/2)*(2**2),\
      -sin((pi/2))*cos(2)+4*(pi/2)*2,0],  \
     [-cos((pi/2))*sin(2)+2*(2**2),0,0]]])

    # Test actual solution with expected solution
    diff = np.max(np.abs(expSolution-actSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)

# Test vector functionality
def test_vec_func_4():
    x,y = init_variables([2,4],[3,1],max_o=2)
    z1 = 3-1/(3*x*y)

    # Find actual and expted solution
    actSolution = np.array([z.get_derivatives() for z in z1])
    expSolution = np.array([\
        [[3 -(1.0/(3*2*3)),1.0/(3*2*3**2),-2.0/(3*2*3**3)],  \
         [1.0/(3*3*2**2),-1.0/(3*(2**2)*(3**2)),0],  \
         [-2.0/(3*3*2**3),0,0]], \
        [[3 -(1.0/(3*4*1)),1.0/(3*4*1**2),-2.0/(3*4*1**3)],  \
         [1.0/(3*1*4**2),-1.0/(3*(4**2)*(1**2)),0],  \
         [-2.0/(3*1*4**3),0,0]]])

    # Test actual solution with expected solution
    diff = np.max(np.abs(expSolution-actSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)

# Test vector functionality
def test_vec_func_5():
    x,y = init_variables([2,4],[3,1],max_o=2)
    z1 = sqrt(y**2/(x*y**2))

    # Find actual and expted solution
    actSolution = np.array([z.get_derivatives() for z in z1])
    expSolution = np.array([\
                    [[2.0**(-0.5),0,0],  \
                     [-0.5*2.0**(-1.5),0,0],  \
                     [0.75*2.0**(-2.5),0,0]], \
                    [[4.0**(-0.5),0,0],  \
                     [-0.5*4.0**(-1.5),0,0],  \
                     [0.75*4.0**(-2.5),0,0]]])

    # Test actual solution with expected solution
    diff = np.max(np.abs(expSolution-actSolution))
    nt.assert_almost_equal(diff,0,delta=1E-10)
```

Appendix C

Reservoir Simulation Code

The complete program including the reservoir simulation solver was included in the main part of the thesis, and is for that reason not included here as well. This section includes relevant files that were not included in the main part of the thesis.

C.1 MATLAB Implementation

The MATLAB implementation is a part of MRST and was used to to compare the performance between the two implementations. Some minor changes were made to this script to perform a reservoir simulation for larger grid models. Additionally, it is important to note that the plotting parts of the script were removed for all performance testing. The initial script that was used follows below:

```
%% Single phase flow simulation using AD
% This example goes through the steps of setting up a single-phase
% simulation with a single horizontal well using the automatic
% differentiation framework.

mrstModule add ad-fi

% Setup 10x10x10  grid of 200x200x50 m model.
nx = 10;    ny=10;      nz = 10;
Dx = 200;   Dy = 200;   Dz = 50;
G = cartGrid([nx, ny, nz], [Dx, Dy, Dz]);
G = computeGeometry(G);

% Assume homogeneous/isotropic rock.
permX = 30*milli*darcy;
poro  = 0.3;
rock.perm = ones(G.cells.num, 1)*permX;
rock.poro = ones(G.cells.num, 1)*poro;
% set rock compressibility:
cr = 1e-6/barsa;

%% Rock properties
% In the case of non-zero rock-compressiblity, the input rock porosity is
% taken as reference at a given reference pressure p_r. The grid pore volumes
% (pv) becomes a function of pressure given by the differential equation
%                    cr = (d pv/d p)/pv
% which results in
%                    pv(p) = pv_r e^( cr(p-p_r) )
% where pv_r is the reference pore volume (rock.poro x volume) and p_r is
% the reference pressure. We assume the reference pressure is 200:
pv_r = poreVolume(G, rock);
p_r  = 200*barsa;
% Finally, the pressure-dependent function for pore-volumes becomes:
pv   = @(p)pv_r.*exp( cr*(p-p_r) );

%% Fluid (oil) properties
% Assume constant viscosity:
mu   = 5*centi*poise;
% and that the oil compressibility can be approximated as constant in the
% reservoir:
c    = 1e-3/barsa;
% With constant compressibility, density becomes a function of pressure
% given the differential equation
%                    c = (d rho/d p)/rho
% which results in
%                    rho(p) = rho_r e^( c(p-p_r) )
% where rho_r is the reference density at the reference pressure p_r. We
% that rho_r = 800 at p_r = 200:
p_r   = 200*barsa;
rho_r = 850*kilogram/meter^3;
% finally define the pressure dependent function for rho:
rho   = @(p)rho_r*exp( c*(p-p_r) );

% to compute the surface volume rates, we need also need the surface
% density which we assume is 750:
rhoS = 750*kilogram/meter^3;

%% Single horizontal well:
W = [];
nperf = 8;
I = ones(nperf,1)*2;
J = (1:nperf)' + 1;
K = ones(nperf,1)*5;
% Convert IJK-indices to linear index (as used in G)
cellInx = sub2ind(G.cartDims, I, J, K);

W = addWell(W, G, rock, cellInx, 'Name', 'producer');

% plottings
figure(1);clf
plotGrid(G, 'FaceColor', 'g', 'FaceAlpha', .3, 'EdgeColor', 'w');
plotWell(G, W);
axis off, set(gcf, 'Color', 'w'), camproj perspective, view(3)

%% Initial conditions
% We assume that the reservoir is initially at equilibrium. This means that
% the following condition must be satisfied:
%           dp/dz = g rho,
% where g is the gavitational accelleration. This relation can be solved
% analytically for p, but alternatively one can solve the above ODE with
% 'initial condtion' p(z_0) = p_r:
gravity on;
g = norm(gravity);

```

```
z_0 = 0; z_max = 20*meter;
[zz,pp] = ode23(@(z,p)g*rho(p),[z_0 z_max],p_r);
% we then interpolate onto the grid using cell centers:
p_init = interp1(zz, pp, G.cells.centroids(:,3), 'spline');

%% Setting up components needed for the simulation
% Since we have no-flow boundary in this example, we restrict to interior
% faces
N  = double(G.faces.neighbors);
intInx = (prod(N,2)~=0);
N  = N(intInx, :);
% We will be using the 2-point flux approximation. First the one-sided
% transmissibilities are computed, then the harmonic average is taken to
% obtain the 2-sided transmissibilites.
hT = computeTrans(G, rock);
cf = G.cells.faces(:,1);
nf = G.faces.num;
T  = 1 ./ accumarray(cf, 1./hT, [nf, 1]);
T  = T(intInx);
% In setting up the equations, we need descrete forms of the div and grad
% operators, and we represent these as multiplication by sparse matrices.
% In particular, we construct the 'gradient matrix' C as folows:
n = size(N,1);
C = sparse( [(1:n)'; (1:n)'], N, ones(n,1)*[1 -1], n, G.cells.num);
% The descrete grad and div operators are now given by
grad = @(x)-C*x;
div  = @(x)C'*x;
% In addition we will need to take the average of neighboring cells, and
% define the following function
avg  = @(x)0.5*(x(N(:,1))+x(N(:,2)));

%% Pressure and well equations:
% The pressure equation (without well contributions) is given by:
%%
% $\frac{d}{dt}(\phi\rho)+\nabla\cdot(\rho v)=0, \quad v = -\frac{K}{\mu}\nabla(p-g\rho z)$
%In discretized form, this leads to
z = G.cells.centroids(:,3); % z-ccordinate of grid cells
pressureEq = @(p, p0, dt) (1/dt).*(pv(p).*rho(p)-pv(p0).*rho(p0))...
    - div( avg(rho(p)./mu).*T.*grad(p-g*rho(p).*z) );
% Wellrates are given as Peaceman well-index times pressure drop
wc = W(1).cells; % perforation grid cells
WI = W(1).WI;    % well indices
dz = W(1).dZ;    % perforation depth relative to well reference depth
wellRates = @(p, bhp) WI.*(rho(p(wc))./mu).*(bhp -p(wc) + g*dz.*rho(p(wc)));
%% define ADI variables
% We let our primary variables be grid-cell pressures, well bhp and surface
% rate:
[p_ad, bhp_ad, qS_ad] = initVariablesADI(p_init, p_init(wc(1)), 0);
% for convenience, make indices to variables when stacked:
pIx    = 1:G.cells.num; bhpIx = G.cells.num +1; qSIx   = G.cells.num +2;

%% Set up simulation parameters
numSteps = 52;
totTime  = 365*day;
dt       = totTime/numSteps;
tol      = 1e-5;
maxits   = 10;
% save output in array 'sol'
sol = repmat(struct('time', [], 'pressure',[], 'bhp', [], 'qS', []), [numSteps+1, 1]);
```

```
sol(1).time     = 0;
sol(1).pressure = double(p_ad);
sol(1).bhp      = double(bhp_ad);
sol(1).qS       = double(qS_ad);
%setup plot
figure(2);clf; set(gcf, 'Color', 'w')
subplot(2,1,1); plotCellData(G, p_init/barsa);
title('pressure␣[bar]','EdgeColor', 'w');
colorbar, view(3), camproj perspective
subplot(2,1,2);
axis([0 convertTo(totTime,day) 0 300]);
title('Surface␣volume␣rate␣[m^3/day]'); hold on

%% Main simulation
t=0; step=0;
while t< totTime
    t=t+dt;
    step=step+1;
    fprintf('\nTime␣step␣%i␣␣time␣\t␣%f␣days\n', step, t/day);
    % newton loop
    resNorm = 1e99;
    p0  = double(p_ad); % previous step pressure
    nit = 0;
    while (resNorm > tol) && (nit < maxits)
        % create equatoins:
        eqs{1} = pressureEq(p_ad, p0, dt);
        % add well contributions in perforated cells:
        eqs{1}(wc) = eqs{1}(wc) - wellRates(p_ad, bhp_ad);
        % sum of wellrates should equal total rate:
        eqs{2} = qS_ad - sum(wellRates(p_ad, bhp_ad))/rhoS;
        % final equation is prescribed bhp
        eqs{3} = bhp_ad - 100*barsa;

        % concatenate equations and solve:
        eq  = cat(eqs{:});
        J   = eq.jac{1};  % Jacobian
        res = eq.val;     % residual
        upd = -J\res;     % Newton update
        %update variables
        p_ad.val   = p_ad.val   + upd(pIx);
        bhp_ad.val = bhp_ad.val + upd(bhpIx);
        qS_ad.val  = qS_ad.val  + upd(qSIx);

        resNorm = norm(res);
        nit     = nit +1;
        fprintf('Iteration␣%i␣\t␣%e\n',nit, resNorm);
    end

    if(nit > maxits)
        error('Newton␣solves␣did␣not␣converge')
    else
        sol(step+1).time     = t;
        sol(step+1).pressure = double(p_ad);
        sol(step+1).bhp      = double(bhp_ad);
        sol(step+1).qS       = double(qS_ad);
        % plot evolution
        figure(2);
        subplot(2,1,1); cla; caxis([120 205])
        plotCellData(G, convertTo(sol(step+1).pressure, barsa), 'EdgeColor', 'w');
        subplot(2,1,2);
        plot(convertTo(sol(step+1).time, day), convertTo(-sol(step+1).qS, meter^3/day),'*');
        drawnow
    end
end
```

C.2 Reservoir Simulation Script

The file reservoir_sim.py simply runs a given grid model and checks the relative errors compared to the same problem solved in MATLAB. This was used to test that everything was correctly implemented. The file follows below:

```
from reservoir_sim_1p import solver
from scipy.io import loadmat
import numpy as np

# Filename with data
filename = 'data10.mat'

# Solve system
numSteps = 52
sol = solver(filename,profile=True)

# Get exact solution
matlab_data = loadmat(filename,struct_as_record=True)
msol = matlab_data['sol']

# Compare to exact solution
# ----------------------------------
# List for holding errors
p_err_list = []
bhp_err_list = []
qS_err_list = []

# Calculate errors
for i in xrange(numSteps):
    # Get exact values
    p_exact = np.array([entry[0] for entry in msol[i]['pressure'][0]])
    bhp_exact = msol[i]['bhp'][0][0][0]
    qS_exact = msol[i]['qS'][0][0][0]

    # Calculate errors
    p_err     = (sol['pressure'][i]-p_exact)/p_exact
    p_err_max = np.max(np.abs(p_err))
    bhp_err   = (sol['bhp'][i]-bhp_exact)/bhp_exact
    qS_err    = (sol['qS'][i]-qS_exact)/qS_exact

    # Append errors
    p_err_list.append(p_err_max)
    bhp_err_list.append(bhp_err)
    qS_err_list.append(qS_err)

# Print errors
print "------------------------------"
print "Pressure␣Errors␣:␣"
print p_err_list
print "-----------------------------------------"
print "bhp Errors : "
print bhp_err_list
print "-----------------------------------------"
print "qS Errors : "
print qS_err_list
```

C.3 Reservoir Constants

The file reservoir_constants.py includes a series of constants used in the reservoir simulation. The file follows below:

```
# Declare constants
milli = 0.001
centi = 0.01
darcy = 9.8692E-13
barsa = 1.0E5
poise = 0.1
kilogram = 1.0
meter = 1.0
g = 9.8066
day = 86400.0
```

C.4 Reservoir Functions

The file reservoir_functions includes a series of functions used in the reservoir simulation. The file follows below:

```
import numpy as np
import sys

# Create function for matrix multiplication
def ADI_mat_mult(M,v):
    N = M.shape[0]
    A = [0]*N

    for i,j,val in zip(M.row,M.col,M.data):
        A[i] += v[j]*val
    return np.array(A)

# Create function for getting vector of values from AD vector
def get_vector_values(ad_vec):
    vals = []
    for var in ad_vec:
        vals.append(var.val)
    return np.array(vals)

# Function for calculating the norm
def norm(a):
    return np.sqrt(np.sum(a*a))

# Function for calculating the poreVolume
def poreVolume(G, rock):
    return rock['poro']*G['cells']['volumes']

# Function for calculating interior transmissibilities
def computeTrans(G,rock):
    # Get interior faces
    N       = G['faces']['neighbors']
    intInx  = np.nonzero(np.prod(N,1))[0]
    N       = N[intInx,:]-1

    # Get cell centroids
    Cc1 = G['cells']['centroids'][N[:,0],:]
    Cc2 = G['cells']['centroids'][N[:,1],:]

    # Get face centroids
    Fc = G['faces']['centroids'][intInx,:]

    # Calculate vectors pointing from cell to face centroids
    c1 = Fc-Cc1
    c2 = Fc-Cc2
    cnorm1 = np.sqrt(np.sum(c1*c1,axis=1))
    cnorm1.shape = (N.shape[0])
    cnorm2 = np.sqrt(np.sum(c2*c2,axis=1))
    cnorm2.shape = (N.shape[0])

    # Get normal vectors
    n = np.array(G['faces']['normals'][intInx,:],dtype=float)
    nnorm = np.sqrt(np.sum(n*n,axis=1))
    nnorm.shape = (N.shape[0],1)
    n = n / nnorm

    # Get cell permeabilities
    perm1 = rock['perm'][N[:,0]]
    perm2 = rock['perm'][N[:,1]]

    # Get face areas
    A = G['faces']['areas'][intInx]

    # Calculate half-face transmissibilities
    cn1 = np.sum(c1*n,axis=1)
    cn1.shape = (N.shape[0])
    cn2 = np.sum(-c2*n,axis=1)
    cn2.shape = (N.shape[0])
    Th1 = A*perm1*cn1/(cnorm1**2)
    Th2 = A*perm2*cn2/(cnorm2**2)

    # Take harmonic average to get 2-sided transmissibilities
    T = 1.0/((1.0/Th1)+(1.0/Th2))
    return T
```

Bibliography

[1] Nose : Python testing framework. `http://nose.readthedocs.org`. Accessed: 29-07-2014.

[2] Jørg E. Aarnes, Tore Gimse, and Knut-Andreas Lie. An introduction to the numerics of flow in porous media using matlab. In Geir Hasle, Knut-Andreas Lie, and Ewald Quak, editors, *Geometric Modelling, Numerical Simulation, and Optimization*, pages 265–306. Springer Berlin Heidelberg, 2007.

[3] Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate taylor series. *Math. Comput.*, 69(231):1117–1130, July 2000.

[4] Knut-Andreas Lie. *An Introduction to Reservoir Simulation Using MATLAB*. SINTEF ICT, 2014.

[5] Knut–Andreas Lie, Stein Krogstad, Ingeborg Skjelkvåle Ligaarden, Jostein Roald Natvig, Halvor Møll Nilsen, and Bård Skaflestad. Open-source matlab implementation of consistent discretisations on complex grids. *Computational Geosciences*, 16(2):297–322, 2012.

[6] Richard D. Neidinger. An efficient method for the numerical evaluation of partial derivatives of arbitrary order. *ACM Transactions on Mathematical Software*, 18(2):159–173, June 1992.

[7] Richard D. Neidinger. Directions for Computing Truncated Multivariate Taylor Series. *Mathematics of Computation*, 74(249):321–340, May 2004.

[8] Richard D. Neidinger. Introduction to automatic differentiation and matlab object-oriented programming. *SIAM Review*, 52(3):545–563, August 2010.

[9] D.W. Peaceman. Interpretation of well-block pressures in numerical reservoir simulation. *Soc. Petrol. Eng. J.*, 18(3):183–194, 1978.

[10] Louis B. Rall. *Automatic Differentiation : Techniques and Applications*. Springer Berlin Heidelberg, 1981.

[11] Eric W. Weisstein. Method of steepest descent. `http://mathworld.wolfram.com/MethodofSteepestDescent.html`. Accessed: 03-05-2014.

[12] Eric W. Weisstein. Newton's method. `http://mathworld.wolfram.com/NewtonsMethod.html`. Accessed: 23-05-2013.

[13] Eric W. Weisstein. Operations research. `http://mathworld.wolfram.com/OperationsResearch.html`. Accessed: 29-04-2014.

[14] Eric W. Weisstein. Optimization theory. `http://mathworld.wolfram.com/OptimizationTheory.html`. Accessed: 29-04-2014.

[15] R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8):463–464, August 1964.