Forward Mode Automatic Differentiation in a Functional Array-Processing Language

Ojasvi Jalal

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2020

Abstract

With many applications in fields like machine learning and computer vision relying on gradient-based optimisation, there is an ever increasing need for tools that offer efficient calculation of derivatives. Automatic Differentiation (AD) is a family of techniques that is capable of computing derivatives with machine precision accuracy. Most popular AD tools are implemented in imperative programming languages as opposed to functional languages. However, $\tilde{F}$, a recently introduced functional array processing language, has shown differentiable programming capabilities that are able to outperform a number of state-of-the-art AD techniques.

This project, inspired by $\tilde{F}$, delivers a system called dd$\tilde{x}$ in a functional framework. dd$\tilde{x}$ is capable of performing both forward mode AD and symbolic differentiation. Unlike $\tilde{F}$, dd$\tilde{x}$ only has a simple interpreter and does not implement reverse mode AD. However, dd$\tilde{x}$'s simple implementation ensures that these features can be introduced with much ease in the near future.

The correctness and efficiency of dd$\tilde{x}$ are demonstrated using several benchmarks. The results obtained from these benchmarks support the argument that AD is more efficient than its counterpart symbolic differentiation.

Acknowledgements

First of all, I would like to thank my mummy, papa, and badi ma, for their constant encouragement throughout my academic life. Without them, I wouldn't be where I am today.

My sincere gratitude goes to my supervisor, Christophe Dubach. First of all, for introducing the course Compiling Techniques to me. The course was taught wonderfully. It ignited my interest in compilers and encouraged me to have my honours project related to the field. And of course, for his valuable guidance and feedback throughout the making of this project and for continually encouraging me to do better!

I shall always remain indebted to Jesse Sigal for his invaluable advice with respect to the soundness of my implementation decisions, his thorough and patient proofreading of my report again and yet again, and for always being there for my endless list of questions.

A special thanks to Cecil Cox without whom I don't see myself understanding and appreciating the magic of functional programming. I learned a lot from our ever so often whiteboard sessions on lambda calculus, typing rules and everything else which greatly influenced many of my design decisions.

Last but not least, my wholehearted thanks to all my friends for their unwavering moral support throughout these four years. Without them, the hard times would have been a lot harder. And of course, I wouldn't have had anyone to annoy all the time.

Table of Contents

Chapter 1

Introduction

A derivative is a description of how a function changes with small changes in one or multiple independent variables. Derivatives are used in every branch of physical sciences, statistics, computer science, medicine, finance and in other fields whenever a problem can be mathematically modeled and an optimal solution sought-after. Conventionally, computational engineering disciplines like machine learning, speech processing, and computer vision have relied on the evaluation of derivatives and many of the traditional learning algorithms have required computation of gradients.

The following methods are used for computation of derivatives in computer programs:

- `Manually` computing derivatives and programming them.
- `Numerical differentiation` by applying finite difference approximations.
- `Symbolic differentiation` by applying expression manipulation in computer algebra systems like Wolfram Mathematica, Maxima, SageMath and Maple.
- `Automatic differentiation,` also known as algorithmic differentiation, which constitutes the subject matter of this report.

Manual differentiation is arduous and error-prone. Numerical differentiation, on the other hand, is straightforward to implement but is highly susceptible to inaccuracy due to truncation and round-off errors (Jerrell, 1997). Besides, it scales rather poorly for gradients, making itself futile for machine learning where gradients with respect to millions of parameters are inevitable. Symbolic differentiation does not suffer from these pitfalls, making it more suitable when compared with the other two methods. However, it often results in complex expressions beset with the problem of “expression swell” [1] (Berz et al., 1996). Moreover, manual and symbolic methods require models to be represented as closed-form expressions [2], thus resulting in severely restricted

[1]Expression Swell is a common phenomenon of exact computations in which the size of numbers and expressions involved in a calculation grows dramatically as the count progresses.

[2]In mathematics, a closed-form expression is a mathematical expression expressed using a finite number of standard operations. It may contain constants, variables, certain “well-known” operations, and functions, but usually, no limit, differentiation or integration.

algorithmic control flow and expressivity (Baydin et al., 2015a).

The increasing significance of computational engineering disciplines like machine learning, speech processing, and computer vision has resulted in a greater need for systems which can overcome the tedious and error-prone methods of computing derivatives. Automatic differentiation (AD) is the powerful fourth technique that facilitates this need. AD accurately evaluates the derivative of numerical functions expressed as computer programs by systematically applying the chain rule. It allows for a precise and accurate evaluation of derivatives with a negligible constant factor of overhead and the best possible asymptotic efficiency (Baydin et al., 2015a). Moreover, it can be conveniently applied to regular code with minimal change, allowing branching, loops, and recursion. All these qualities make AD a useful tool in fields including machine learning, computational fluid dynamics, physical modelling, structural mechanics, optimal control, atmospheric sciences, engineering design optimization and computational finance.

1.1 Motivation

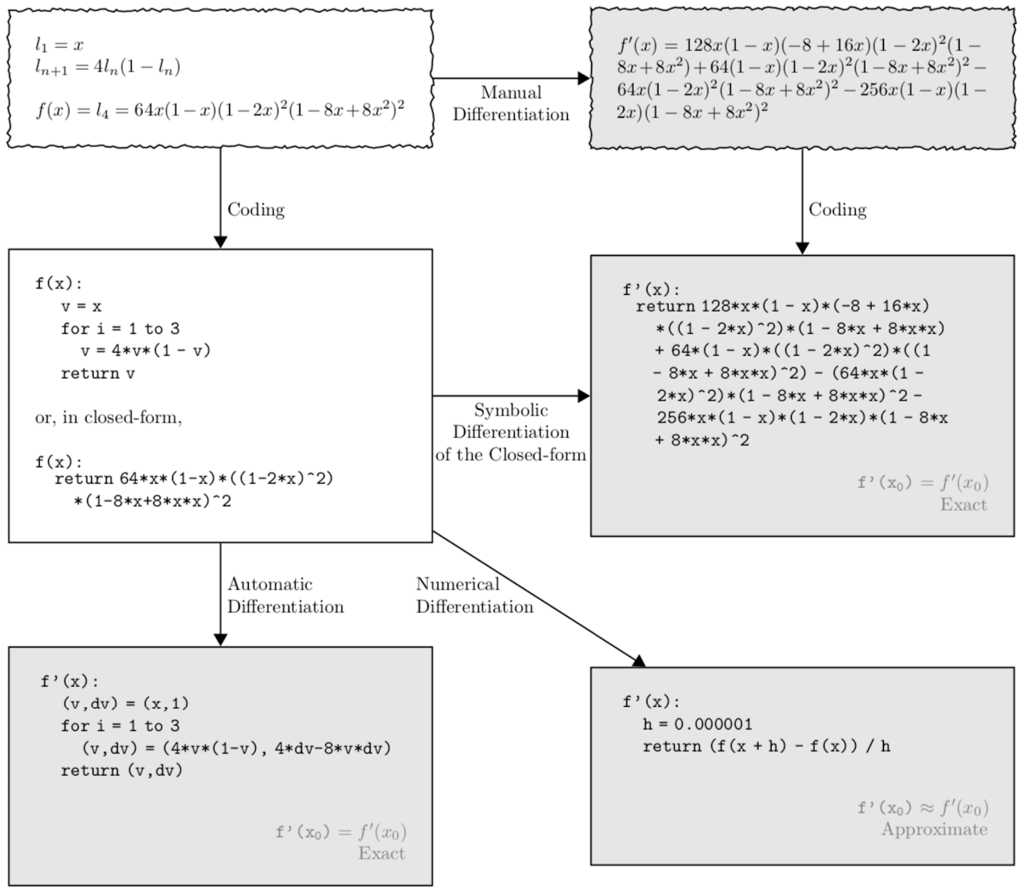
Until recently, general-purpose AD had been missing from the machine learning toolbox, a situation slowly changing with the emergence of several AD packages in the machine learning discipline (Baydin et al., 2015a). However, most of the popular AD packages predominantly implement (E)DSLs [3] in procedural languages like Python and C++. Popular examples include TensorFlow and CNTK. This is despite the fact that functional programming (FP), with its strong correspondence to mathematical specifications, results in less clutter and more elegant solutions. Thus, FP and AD are natural partners. There are arguments related to FP only being able to handle scalar workloads efficiently and being significantly slower than its imperative counterparts when it comes to array processing (Srajer et al., 2018). However, a recently introduced language called $\tilde{F}$ (Shaikhha et al., 2019) implements a functional array-processing AD tool that has proven to be competitive with the best C/C++ tools and Fortran tools like Tapenade.

This project is inspired by $\tilde{F}$ and presents a functional array-processing language called dd$\tilde{x}$ with the capability of performing automatic differentiation (AD) in forward mode.

1.2 Understanding Common AD Misconceptions

AD is often incorrectly assumed to be either a type of numerical or symbolic differentiation. Perhaps this could be because AD does compute numerical values of derivatives (instead of derivative expressions). Moreover, AD computes the numerical value of derivative by applying symbolic rules of differentiation (requiring bookkeeping of derivative values as opposed to the resulting expressions). Therefore, one can argue that AD has a two-sided nature that is partly symbolic and partly

[3] An embedded domain-specific language (eDSL) inherits the generic language construct of its host language while adding domain-specific language elements (data types, methods, macros, etc.). (e.g. jQuery, Embedded SQL, LINQ) (Gill, 2014)


$l_1 = x$
$l_{n+1} = 4l_n(1 - l_n)$

$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$

$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

```
f(x):
  v = x
  for i = 1 to 3
    v = 4*v*(1 - v)
  return v
```

or, in closed-form,

```
f(x):
  return 64*x*(1-x)*((1-2*x)^2)
    *(1-8*x+8*x*x)^2
```

```
f'(x):
  return 128*x*(1 - x)*(-8 + 16*x)
    *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
    + 64*(1 - x)*((1 - 2*x)^2)*((1
    - 8*x + 8*x*x)^2) - (64*x*(1 -
    2*x)^2)*(1 - 8*x + 8*x*x)^2 -
    256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
    + 8*x*x)^2
```

```
f'(x):
  (v,dv) = (x,1)
  for i = 1 to 3
    (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
  return (v,dv)
```

```
f'(x):
  h = 0.000001
  return (f(x + h) - f(x)) / h
```

Figure 1.1: The range of methods for the computation of derivatives in computer programs can be classified into four categories. The example expression (upper left) here is that of a polynomial mapping of degree 2. Symbolic differentiation (centre right) gives accurate results but requires the input to be in closed-form and is beset with the problem of expression swell. Numerical differentiation (lower right) suffers from round-off and truncation errors, leading to inaccurate results. Automatic differentiation (lower left) is as accurate as symbolic differentiation with a small constant factor of overhead, an ideal asymptotic efficiency and support for control flow. It achieves this by interleaving differentiation and simplifying computations by only storing the values of intermediate sub-expressions in memory. This figure is directly taken from Baydin et al. (2015a))

numerical (Griewank, 2003). Nonetheless, AD is neither numerical nor symbolic differentiation. Figure 1 adequately demonstrates how AD is different from, and in many ways superior to, these two conventional techniques of computing derivatives.

1.3 Project Goals

This project takes inspiration from an existing functional array-processing language, $\tilde{F}$, for differentiable programming (Shaikhha et al., 2019). Thus, this project aims to implement AD in a functional framework, and evaluate and compare the efficiencies of the implementation of AD and symbolic differentiation.

1.4 Contributions

Taking the project goals into account, the following contributions are made:

1. A system is designed and implemented to perform forward mode AD in a functional setting. This system is called dd$\tilde{x}$.
2. Algorithms dependent on systematic pattern matching are designed for abstract syntax tree (AST) transformations. These are used in implementing the interpreter for dd$\tilde{x}$. The interpreter constitutes of an evaluator, a type-checker, symbolic differentiation functionality, forward mode AD functionality and a pretty-printer.
3. A polymorphic-like type system and its corresponding type-checker are implemented for dd$\tilde{x}$. This feature ensures correctness of programs and prevents invalid operations.
4. Shows how a language can be built in a functional setting to process array workloads.
5. Shows the differentiation programming capabilities offered by dd$\tilde{x}$. Section 4.7 shows the high-level view of how matrix derivatives like scalar derivatives, gradients and Jacobians are performed in dd$\tilde{x}$. Then, section 5.7 explains how this is implemented using an AST-to-AST transformation.
6. Shows how matrix-vector operations can be expressed with the help of map, reduce, and zip patterns.
7. The performance of dd$\tilde{x}$ in terms of correctness and efficiency is evaluated and presented.
8. The array-processing performance of dd$\tilde{x}$ is evaluated on a set of benchmarks. These benchmarks include operations concerning differentiation of vector addition, vector-scalar multiplication, dot product, and the maximum value of a vector.
9. The implementations of symbolic differentiation and automatic differentiation are evaluated and compared. This evaluation demonstrates how AD is in fact more efficient than symbolic differentiation.

1.5 Thesis Organisation

The thesis is organised as follows:

Chapter 2 introduces fundamentals of AD. **Chapter 3** presents a discussion and critical evaluation of related previous work. **Chapter 4** discusses the design details of dd$\tilde{x}$ for AD. **Chapter 5** discusses the implementation details of dd$\tilde{x}$. **Chapter 6**

reports the results of the benchmarks and performance of the different functionalities offered by dd$\tilde{x}$. **Chapter 7** contains a summary of the contributions, a conclusion and an outline for future work.

Chapter 2

Background

In simple words, automatic differentiation (AD) is decomposing a complex function into simpler operations (like +, sin, x^2), coding derivative calculations for them, then representing the complex function using the simpler operations. Therefore, all AD is concerned with is the sequence of operations performed. Its implementation falls into two main categories: forward mode and reverse mode. Forward mode AD evaluates the original program (primal trace) and computes the derivative part (tangent trace) alongside. On the other hand, reverse mode AD uses a forward pass to evaluate the original part of the program, followed by a backward pass for computing the derivative part (adjoint trace). This chapter reviews the fundamentals of the forward and reverse mode automatic differentiation.

The following equation is used in the upcoming sections to illustrate the process of execution in the main modes of AD:

$$f(x_1, x_2, x_3, x_4) = x_1 \sqrt{x_1 x_2 + {x_3}^2 - x_4} \tag{2.1}$$

Figure 2.1 represents the given trace of elementary operations in equation 2.1 in the form of a computational graph [1] (Bauer, 1974).

2.1 Forward Mode

Forward mode AD is conceptually the most straightforward version of automatic differentiation.

Let us consider the evaluation trace of

$$f(x_1, x_2, x_3, x_4) = x_1 \sqrt{x_1 x_2 + {x_3}^2 - x_4}.$$

[1]Computational graphs are useful in visualising dependencies between intermediate variables. It is also important to note that computational graphs do not take into account the complicated logic in the source code such as loops and conditionals. Given a set of input values, whatever computation occurs next gets added to the graph regardless of what “might” have been in the other conditional branch.

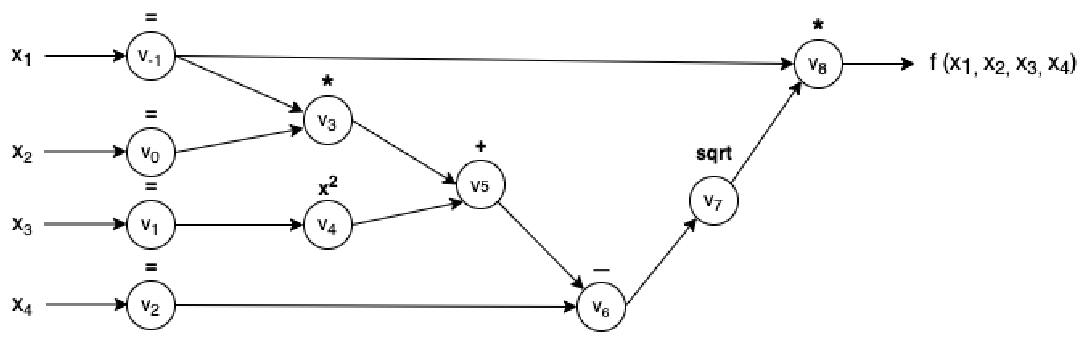


Figure 2.1: Computational graph representing $f(x_1,x_2,x_3,x_4) = x_1\sqrt{x_1x_2 + {x_3}^2 - x_4}$. Table 2.1 has the corresponding primal trace.

Forward Mode AD	
Forward Primal Trace	**Derivative Trace**
$v_{\text{-}1} = x_1$	$\dot{v}_{\text{-}1} = 1$
$v_0 = x_2$	$\dot{v}_0 = 0$
$v_1 = x_3$	$\dot{v}_1 = 0$
$v_2 = x_4$	$\dot{v}_2 = 0$
$v_3 = v_{\text{-}1} \times v_0$	$\dot{v}_3 = \dot{v}_{\text{-}1}v_0 + v_{\text{-}1}\dot{v}_0$
$v_4 = {v_1}^2$	$\dot{v}_4 = 2v_1\dot{v}_1$
$v_5 = v_3 + v_4$	$\dot{v}_5 = \dot{v}_3 + \dot{v}_4$
$v_6 = v_5 - v_2$	$\dot{v}_6 = \dot{v}_5 - \dot{v}_2$
$v_7 = \sqrt{v_6}$	$\dot{v}_7 = \frac{1}{2}{v_6}^{-\frac{1}{2}}\dot{v}_6$
$v_8 = v_{\text{-}1} \times v_7$	$\dot{v}_8 = \dot{v}_{\text{-}1}v_7 + v_{\text{-}1}\dot{v}_7$
$y = v_8$	$\dot{y} = \dot{v}_8$

Table 2.1: Forward mode AD on $f(x_1,x_2,x_3,x_4) = x_1\sqrt{x_1x_2 + {x_3}^2 - x_4}$. $\dot{x}$ is set to 1 for computing $\frac{df}{dx_1}$. The differentiation is accomplished by augmenting the evaluations of the primals on the left by the derivative operations on the right.

The corresponding forward primal trace of this equation is given on the left-hand side in Table 2.1 and in a computational graph form in Figure 2.1. The figure and the table demonstrate how forward primal trace is essentially the composition of the sub-expressions in the equation, starting from input nodes v_{-1},v_0, v_1 and v_2, to the output node v_8 resulting in the value of the function f.

The chain rule is applied to each elementary operation in the forward primal trace to generate the corresponding derivative trace, given on the right hand side in Table 2.1. The derivative trace first associates with each intermediate variable v_i a derivative

$$\dot{v}_i = \frac{d(v_i)}{dx_1}. \tag{2.2}$$

Evaluating these primals v_i alongside with their corresponding tangents $\dot{v}_i$ eventually results in the derivative of the expression, represented by the final tangent

$$\dot{v}_8 = \frac{df}{dx_1}.$$

This method can be generalized to multi-variable functions, deriving which result in Jacobian matrices[2]. Consider the following differentiable function:

$$F : R^n \to R^m \tag{2.3}$$

with n independent variables $(x_1, x_2, ..., x_n)$ and m dependent variables $(y_1, y_2, ..., y_m)$:

$$F := \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_m \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2, ..., x_\text{n}) \\ f_2(x_1, x_2, ..., x_\text{n}) \\ . \\ . \\ . \\ f_m(x_1, x_2, ..., x_\text{n}) \end{bmatrix} \tag{2.4}$$

The Jacobian of F, given by

$$F' := \left(\frac{dy_i}{dx_j}\right)_{m\times n} = \begin{bmatrix} \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \cdots & \frac{dy_1}{dx_n} \\ \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} & \cdots & \frac{dy_2}{dx_n} \\ . & & & \\ . & & & \\ . & & & \\ \frac{dy_m}{dx_1} & \frac{dy_m}{dx_2} & \cdots & \frac{dy_m}{dx_n} \end{bmatrix} \tag{2.5}$$

, is a $m \times n$ matrix consisting of first-order partial derivatives.

Furthermore, a Jacobian-vector product like

$$F'\boldsymbol{k} := \begin{bmatrix} \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \cdots & \frac{dy_1}{dx_n} \\ \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} & \cdots & \frac{dy_2}{dx_n} \\ . & & & \\ . & & & \\ . & & & \\ \frac{dy_m}{dx_1} & \frac{dy_m}{dx_2} & \cdots & \frac{dy_m}{dx_n} \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \\ . \\ . \\ . \\ k_n \end{bmatrix}, \tag{2.6}$$

can be computed in forward mode AD by simply setting $\dot{x} = \boldsymbol{k}$. Therefore, enabling computation of Jacobian-vector product in a single forward pass and providing a highly efficient and matrix-free way of computing Jacobian-vector products. (Baydin et al., 2015a; Pearlmutter and Siskind, 2008b)

A single forward pass is needed for computing derivatives $\frac{dy_i}{dx_j}$ of functions like $f : R \to R^m$. Whereas, for functions like $f : R^n \to R$, forward mode AD requires a total of n evaluations to compute the gradient

$$\nabla f = \left(\frac{dy}{dx_1}, ..., \frac{dy}{dx_n}\right). \tag{2.7}$$

[2] For a vector-valued function in several variables, its Jacobian Matrix is the matrix of all its first-order partial derivatives. (Haukkanen et al., 2017)

Reverse Mode AD	
Forward Primal Trace	**Derivative Trace**
$v_{\text{-}1} = x_1$	$\dot{v}_{\text{-}1} = \dot{v}_{\text{-}1} + \dot{v}_{\text{-}1}\frac{dv_3}{dv_{-1}}$
$v_0 = x_2$	$\dot{v}_0 = \dot{v}_0 + \dot{v}_0\frac{dv_3}{dv_0}$
$v_1 = x_3$	$\dot{v}_1 = \dot{v}_1 + \dot{v}_4\frac{dv_4}{dv_1}$
$v_2 = x_4$	$\dot{v}_2 = \dot{v}_2 + \dot{v}_6\frac{dv_6}{dv_2}$
$v_3 = v_{\text{-}1} \times v_0$	$\dot{v}_3 = \dot{v}_5\frac{dv_5}{dv_3}$
$v_4 = {v_1}^2$	$\dot{v}_4 = \dot{v}_5\frac{dv_5}{dv_4}$
$v_5 = v_3 + v_4$	$\dot{v}_5 = \dot{v}_6\frac{dv_6}{dv_5}$
$v_6 = v_5 - v_2$	$\dot{v}_6 = \dot{v}_7\frac{dv_7}{dv_6}$
$v_7 = \sqrt{v_6}$	$\dot{v}_7 = \dot{v}_8\frac{dv_8}{dv_7}$
$v_8 = v_{\text{-}1} \times v_7$	$\dot{v}_{\text{-}1} = \dot{v}_8\frac{dv_8}{dv_{-1}}$
$y = v_8$	$\dot{y} = \dot{v}_8 = 1$

Table 2.2: Reverse mode AD on $f(x_1,x_2,x_3,x_4) = x_1\sqrt{x_1x_2+{x_3}^2-x_4}$. It is a two phase process. The first phase (on the left) involves evaluation of the intermediate variables (Forward Primal Trace). The second phase (on the right) evaluates the adjoints in reverse.

In general, forward mode is more efficient than reverse mode for functions $f: R^n \to R^m$ with $m \gg n$ as only n evaluations are necessary, compared to m evaluations for reverse mode. Whereas, for cases where $n \gg m$, reverse mode is preferred.

2.2 Reverse mode

Let us again consider the expression

$$f(x_1,x_2,x_3,x_4) = x_1\sqrt{x_1x_2+{x_3}^2-x_4}$$

and the corresponding nodes in the computational graph in Figure 2.1. During reverse mode AD, the forward pass traverses the nodes from v_{-1} to v_8, computing the resulting value of each node and storing it using a bookkeeping procedure. Following the forward pass, as shown in Table 2.2, the backward pass then traverses the adjoints[3]

$$\dot{v}_i = \frac{dy_j}{dv_i}, \tag{2.8}$$

[3] An adjoint is a derivative of a chosen dependent variable y with respect to a subexpression v.

from $\dot{v}_8$ to $\dot{v}_{-1}$ (from bottom to top), i.e. from the outputs to the inputs. This method corresponds to a generalized backpropagation algorithm as it propagates derivatives backwards from a given output.

In general, for a function $f : R^n \to R^m$, the performance of reverse mode is superior when $n \gg m$, i.e. when there are many inputs and fewer outputs. This advantage, however, comes with the cost of a performance overhead in the form of increased storage requirements. These storage requirements grow linearly (in the worst case) with the number of operations in the evaluated function (Baydin et al., 2015a). Advanced methods like lazy evaluation [4] (Margossian, 2019), checkpointing strategies (Siskind and Pearlmutter, 2018) and data-flow analysis (Dauvergne and Hascoët, 2006) are being actively researched to enable more efficient handling of the storage requirements.

Although this project itself does not implement reverse mode AD, nevertheless, this fact does not rule out the possibility of extending dd$\tilde{x}$ to support this functionality in the near future. Perhaps, the new technique could even combine the two modes to produce a more efficient technique for implementing AD, similar to the one in $\tilde{F}$ (Shaikhha et al., 2019).

2.3 Summary

The chapter discusses the process of carrying out forward mode and reverse mode AD.

Forward mode AD is accomplished by evaluating the intermediate variables (obtained by splitting a complex function into sub-expressions) alongside computing their derivatives. This results in two traces: a forward trace and a derivative trace.

Reverse mode AD is a two-phase process. The first phase is a forward trace, just like in forward mode AD. This phase evaluates the value of a function by populating intermediate variables v_i and storing the dependencies in a computational graph using a bookkeeping method. In the second phase, adjoints ($\dot{v}_i$) are propagated in reverse, giving the derivatives. The use of bookkeeping methods leads to increased storage requirements, making reverse mode vulnerable to performance overhead.

Forward mode is preferred for functions $f : R^n \to R^m$ with $m \gg n$ as only n evaluations are necessary, compared to m evaluations for reverse mode. And for cases where $n \gg m$, reverse mode is more efficient.

Going forward, familiarity with these concepts is assumed in the upcoming chapters.

[4] A lazy evaluation does not evaluate a statement immediately when it appears in the code, but stores it in memory and evaluates it when (and only if) the code explicitly requires it. As a result, we only evaluate expressions required to compute the gradient, or more generally the object of interest.

Chapter 3

Related Work

As mentioned in Chapter 1, there is a large body of work on addressing the problems of differentiation. This chapter reviews literature related to the subject of this report.

3.1 Automatic Differentiation in Functional Frameworks

Karczmarczuk (2001) presents a purely functional implementation for both forward and reverse mode AD (limited to scalar types) in Haskell. The implementation relies on lazy evaluation to compute infinite towers of derivatives of higher-order. *Elliott (2009)* further builds on Karczmarczuk's work by giving a more elegant solution for its forward mode AD using numeric instances for functions. However, in practice, forward mode AD tends to have a higher complexity for machine learning models (Wang et al., 2018). Moreover, both these implementations lack the transformation rules' optimizations like loop fusion (Shaikhha et al., 2019).

Pearlmutter and Siskind (2008b) extends higher-order forward mode AD to the multi-dimensional case. Their work formalizes forward mode and reverse mode AD in a functional framework.

Moreover, there are projects like *DiffSharp* (Baydin et al., 2015b) based on the Pearlmutter and Siskind (2008b) model. DiffSharp is a flexible differentiable functional programming library designed in the F# language with machine learning in mind. It provides techniques like arbitrary nesting of forward/reverse AD operations and AD with linear algebra primitives. However, being a library implementation of AD, it is unable to support the simplification rules like loop-invariant code motion, loop fusion, and partial evaluation (Shaikhha et al., 2019).

The work that this project is inspired by, $\tilde{F}$ (Shaikhha et al., 2019), combines both forward and reverse mode, in many cases outperforming the individual techniques in the process. $\tilde{F}$, augmented with vector AD primitives, is a F# subset designed for efficient compilation of array-processing workloads. Its AD functionality outperforms the existing AD tools on numerous micro-benchmarks and real-world machine learning and computer vision applications. They remark:

"The key idea behind our system is exposing all the constructs used in differentiated programs to the underlying compiler. As a result, the compiler can apply various loop transformations such as loop-invariant code motion and loop fusion." [1] [2]

3.2 AD in Imperative Programming Languages

There exist several packages across different imperative programming languages that implement AD and allow differentiation to occur seamlessly. Some of them are discussed below:

Tapenade, introduced by Hascoet and Pascual (2013), is an AD tool which, given a program computing a function in Fortran or C, creates a new program that computes its tangent or adjoint derivatives. It is itself implemented in Java and can be run locally or as an online service. The *ADIFOR 2.0* system provides automatic differentiation of Fortran 77 programs for first-order derivatives (Bischof et al., 1996). It uses "black box" tools for AD. These tools use the source code and an indication of variable dependencies with respect to differentiation (dependent and independent variables) to generate derivative code without further user intervention. In addition to this, *Adept* (Hogan, 2014) is an operator-overloading implementation of first-order forward mode and reverse mode automatic differentiation for C++ programs. Its use of expression templates and an efficient tape structure make it an efficient and fairly fast tool for the application. It is worth noting that Hascoet and Pascual (2013) argue that source transformation cannot handle typical C++ features and that operator overloading is the appropriate technology in this case. Moreover, AD tools like Tapenade only have limited support for loop fusion, making them susceptible to missing several optimisation opportunities.

For MATLAB, the *ADiMat* (Bischof et al., 2002) tool uses a combination of source transformation and operator overloading to enable the application of the forward mode of AD to a MATLAB program. *ADiGator* (Weinstein and Rao, 2017) performs source transformation via operator overloading using forward mode algorithmic differentiation and produces a file that can be evaluated to obtain the derivative of the original function at a numeric value of the input. A convenient by-product of the file generation is the sparsity pattern[3] of the derivative function. Another line of work, *ForwardDiff*, performs vector forward mode AD (Revels et al., 2016) for Julia programs using dual numbers.

[1]Loop-invariant code is made of statements or expressions which can be moved outside the body of a loop without affecting the semantics of the program. Loop-invariant code motion is a compiler optimization performing this movement automatically. (Luporini, 2015)

[2]Loop fusion (or loop jamming) is a kind of compiler optimization. It consists of a loop transformation which replaces multiple loops with a single one. It is possible when two loops iterate over the same range and do not reference each other's data. Loop fusion may not always improve run-time speed. (Grelck et al., 2005)

[3]Sparsity pattern is a connectivity structure. It is used by linear solvers to recognise structurally non-zero elements of a Jacobian matrix that enter computations.A numerical solver requires this pattern at the initialization stage to allocate objects required for sparse linear algebra algorithms. (Peles and Klus, 2015)

There is particularly a large body of work on AD for Python. Examples include *Tangent* (van Merriënboer et al., 2017) and *TensorFlow* (Abadi et al., 2016) which provide AD in Python through the use of source-to-source transformations. Tangent is mainly aimed at providing intuitive debugging rather than improving performance. Tensorflow uses a distributed execution approach for large-scale machine learning. However, a severe limitation associated with source transformation is that it can only use the information available at compile-time. This prevents it from handling more sophisticated programming statements, such as while loops and other object-oriented features (Hogan, 2014). *AutoGrad* (Maclaurin et al.) performs AD for Python programs that use NumPy library for array manipulation. On the other hand, *Theano* (Bergstra et al., 2010), a CPU and GPU math compiler in Python, uses both symbolic differentiation and AD. Just like most systems discussed above, these systems miss optimisation opportunities like loop fusion (Shaikhha et al., 2019).

3.3 Compiler and Interpreter-based AD Implementations

There are several implementations introducing new languages with integrated AD capabilities through special-purpose compilers and interpreters. This section discusses a few of these implementations.

Stalingrad (Pearlmutter and Siskind, 2008a) is an optimising compiler for a dialect of Scheme. It only operates on λ-calculus IR. The source language Stalingrad compiles is called VLAD. VLAD is purely functional (except for a minimal I/O facility), and provides built-in operators for invoking AD. The compiler provides support for both forward mode and reverse mode of AD. Perturbation confusion [4] (Pearlmutter and Siskind, 2005) is one of the key challenges that is addressed by Stalingrad. However, as with any other work, Stalingrad also comes with its limitations. One of its key limitations is the absence of support for variable-size vectors. The DVL compiler [5] is based on Stalingrad and uses a re-implementation of portions of the VLAD language.

Myia (Breuleux and van Merriënboer, 2017) is a compiler toolchain which uses a dedicated functional representation. Myia performs type inference given the input types and applies a series of optimisations such as inlining, common expression elimination, constant propagation, closure conversion, and algebraic simplifications. It builds a first-order gradient operator for a subset of Python by combining operator overloading and dataflow programming to overcome optimisation issues. However, reverse mode AD in Myia poses a few specific challenges for optimisation. The AD transformation, in this case, produces programs that are substantially larger than the original program, resulting in many unnecessary computations (van Merriënboer et al., 2018).

[4]Forward mode AD attaches a perturbation to each number and propagates through these by overloading the arithmetic operators during the computation. Perturbation confusion is the inability to distinguish between different perturbations introduced by different invocations of the differentiation operator. In particular, it occurs when computing the derivative of the functions for which the derivatives are already computed. (Siskind and Pearlmutter, 2005)

[5]https://github.com/NUIM-BCL/dysvunctional-language

An example of an interpreter-based implementation is AMPL (Fourer et al., 2002), an algebraic modelling language. It enables the representation of objectives and constraints in mathematical notation, from where the system recognises the active variables and then follows the necessary AD computations. The FM/FAD package (Mazourik, 1991), based on the DIFALG language, and COZY (Bischof et al., 1997), the object-oriented Pascal-like language, are other examples that fall under this category.

3.4 Project Contributions

The key contribution of this project is the implementation of an interpreter-based system called dd$\tilde{x}$. This is an attempt to bridge the gap between increasingly popular AD packages built in procedural languages and AD for FP traditionally handling only scalar load efficiently. dd$\tilde{x}$ is built on a framework that brings together forward mode AD with linear algebra primitives and higher-order differentiation capabilities. It is similar in spirit to $\tilde{F}$. However, in addition to implementing AD, dd$\tilde{x}$, also implements symbolic differentiation (SD). This additional feature allows comparing the efficiencies of AD and SD in dd$\tilde{x}$, therefore, providing more evidence to support the claim that AD is indeed more efficient than SD. This contribution is valuable as it is an addition to the data showing AD is often superior to SD in terms of efficiency, even in less empirically explored areas such as AD in functional programming.

3.5 Summary

Earlier, AD for FP was mostly limited to scalar types (Karczmarczuk, 2001). The trend has since changed, and there have been implementations which extend higher-order AD to the multi-dimensional case, examples include Pearlmutter and Siskind (2008b) and DiffSharp. However, they lack support for simplification rules like loop fusion and partial evaluation. A recently introduced $\tilde{F}$ (Shaikhha et al., 2019) presents an AD tool built in functional setting. The tool is competitive with the best Fortran and C/C++ tools on a number of benchmarks.

Several imperative programming languages offer AD packages. Some of the popular examples include Tapenade, ADiMat, TensorFlow and Tangent. Most of these systems rely on source-to-source transformation and suffer from a lack of optimisation opportunities like loop fusion.

Another line of work introduces languages with AD capabilities through special-purpose compilers and interpreters. Stalingrad, an optimising compiler for VLAD, addresses challenges like perturbation confusion; however, it lacks support for variable-size vectors. There are also interpreter-based implementations like AMPL.

This project implements an interpreter-based system called dd$\tilde{x}$. It is capable of performing forward mode AD as well as symbolic differentiation. The evaluation of its differentiation capabilities provides valuable evidence regarding AD being the more efficient differentiation method.

Chapter 4

Design

dd$\tilde{x}$ is inspired by $\tilde{F}$, a higher-order functional language with array support (Shaikhha et al., 2019). dd$\tilde{x}$ is specifically designed keeping functional programming and lambda calculus in mind. Just like $\tilde{F}$, while its restricted nature makes it simpler to introduce automatic differentiation rules, it does not affect its expressivity for matrix-vector operations. Its simple design makes it easy to keep extending the features in the language by simply adding new nodes to its abstract syntax and relevant support for the new nodes to the rest of its components.

4.1 Design Decisions

dd$\tilde{x}$ has a simple abstract syntax tree which is easy to rewrite and further extend. The design relies on lambda abstractions as well as let bindings and conditionals. Since it is designed especially for differentiable programming, its functional nature allows its programs to correspond closely with the mathematical specification they implement. Its higher-order nature means that any program in this language relies on function composition. Function composition is another characteristic which makes it suitable for rewriting and further extension.

4.2 Type System

dd$\tilde{x}$ has a flexible type system because a single expression can have multiple different types. Therefore, the type system behaves similarly to a polymorphic type system without incurring its complexity. The polymorphism-like ability to allow functions to admit many different types is especially useful in higher-order functions, and dd$\tilde{x}$ implements a higher-order functional language. Thus, making this type system a natural design choice for the dd$\tilde{x}$ system.

Simple types include:

```
IntType DoubleType BooleanType
```

Complex types in dd$\tilde{x}$ are parameterised like in Java generics. These types can be represented in terms of type parameters as follows:

```
--FunctionType:  in_type represents the type of the input function
and out_type represents the type of the output.
FunctionType (in_type, out_type)

--VectorType:  elem_type represents the type of vector elements
and length represents the vector size.
VectorType (elem_type, length)

--PairType:  elem_1_type represents the type of the first element
and elem_2_type represents type of the second element in the pair.
 PairType (elem_1_type, elem_2_type)
```

4.3 Type Checking

Following are the typing rules in dd$\tilde{x}$.

$$\frac{\Gamma \vdash e_1 : A \to B \qquad e_2 : A}{\Gamma \vdash funcCall\ e_1 e_2 : B}(FuncCall) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : T}(Variable)$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash let\ x = e_1\ in\ e_2 : B}(Let) \qquad \frac{\Gamma \vdash e_1 : Bool \qquad e_2 : A e_3 : A}{if\ e_1\ then\ e_2\ else\ e_3 : A}(IfElse)$$

These typing rules are enforced by dd$\tilde{x}$'s type checker.

4.4 Abstract Syntax

The abstract syntax specification of dd$\tilde{x}$ is given below in the same standard style as learnt during the UG3 Compiling Techniques coursework: [1]

```
--The program's top AST node (zero or more expressions)
Program ::= Expr*
--Types supported in the language
Type ::= Scalar | VectorType | FuncType | PairType
--Scalar types hold a single data item
Scalar ::= DoubleType | IntType | BooleanType
--VectorType supports both fixed-size and variable-size vectors.
Type represents the element type, Expr represents the number of
elements and can be a constant or a variable.
VectorType ::= Type Expr
```

[1]https://www.inf.ed.ac.uk/teaching/courses/ct/18-19/

```
--Pair types hold the types of the fist and second element
PairType ::= Type Type
--FunctionType holds the types of its argument and its return
expression respectively
FunctionType ::= Type Type


--Kind of expressions supported in the language
Expr ::= DoubleLiteral |IntLiteral | BoolLiteral | Vector | Pair |
Matrix | VarExpr | FunctionCall | Function | VectorAccess


--Literals
--Stores the value of a double
DoubleLiteral ::= double
--Stores the value of an integer
IntLiteral ::= int
--Stores the value of a boolean
BoolLiteral ::= bool


-- Variable (the String is the name of the variable)
VarExpr ::= String
-- A pair, each element in any pair is an Expr
Pair ::= Expr Expr
-- Fixed-size vectors store a one dimensional sequence of expressions
-- Note:  A sequence in Scala is equivalent to a list in Java
Vector ::= Seq[Expr]
-- A variable sized vector, Param stores the vector name, Expr
stores the variable size of the vector
VectorVar ::= Param Expr
-- Sequence of integers, used for operations on variable sized
vectors
Sequence ::= Seq[Int]
-- Matrices are multidimensional sequences of expressions
Matrix ::= Seq[Seq[Expr]]


-- Equivalent to an array access expression :  Expr[int] (e.g.
a[10]), Expr[Expr] (e.g.  a[N])
VectorAccess ::= Vector Expr


-- A function call holds a function and the arguments the fuction
takes
FunctionCall ::= Function Expr
-- Functions are split into two kinds:  built-in and anonymous
Function ::= BuiltInFunctions | AnonymousFunctions
```

```
-- Functions that are built-into ddx̃
BuiltInFunctions ::= Add | Multiply | Power | If_Else | GreaterThan
| Map | Fold | Zip| Drop
-- Anonymous functions
AnonymousFunctions ::= Lambda
-- A lambda expression has a parameter and a body as its inputs.
For example, Lambda(x, x+2)
Lambda ::= Param Expr
-- A let expression is a syntactic sugar to make it more straightforward
to apply lambda abstractions to terms.  They also serve as a
replacement for assignments as asignments are not supprted in ddx̃.
A standard let expression in ddx̃ looks like this:  Let(x, 2, x+2)
which means Let x = 2 in x+2
Let ::= Param Expr Expr
```

4.5 Functions

A ddx̃ program is made of expressions, combined with the help of function application and composition. This section attempts to give a succinct description of the anonymous and the built-in functions in ddx̃.

4.5.1 Anonymous Functions

Anonymous functions are ubiquitous in functional programming languages. An anonymous function (also known as function literals, lambda abstraction, or lambda expression) is a function definition which is not bound to an identifier. Anonymous functions are usually arguments being passed to higher-order functions, for example, map, fold, etc., or used for creating the result of a higher-order function that returns a function.

An anonymous function, being syntactically lighter than a named function, is useful if the function is only used once, or a limited number of times. In a number of programming languages, anonymous functions use the keyword lambda. Similarly, ddx̃ support anonymous functions using the "lambda" construct, which is a reference to lambda calculus. It can be represented as follows:

$$\frac{\Gamma \vdash x : A \qquad \Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \rightarrow B}(abs)$$

In addition to the usual lambda-calculus constructs, ddx̃ also supports let bindings to introduce syntactic sugar for the lambda abstraction applied to a term. They can be represented as follows:

`Let x = y in z` $\equiv (\lambda x.z)y$

By chaining such definitions, one can write a lambda calculus "program" as zero or more function definitions, followed by one lambda-term using those functions that constitute the main body of the program. In simpler words, lets are syntactic sugar for binding expressions in a chain of function applications. In this project, they specifically come in handy for implementation of the forward primal trace and derivative trace in forward mode AD. More details are given in the next chapter.

4.5.2 Built-in Functions

The user can add any symbol, given that they specify its type and method of evaluation. Note that the functions in this language are curried [2] by default and that "Vector" below refers to `Vector`, `VectorVar` or `Sequence` in dd$\tilde{x}$. Below is a list of the built-in functions in dd$\tilde{x}$ along with their types:

```
Note:  a and b represent the types in ddx̃
map ::  (a -> b) -> Vector(a) -> Vector(b)
fold ::  (b -> a -> b) -> b -> Vector(a) -> b
zip ::  a -> b -> Pair(a, b)
greaterThan::  a -> b -> BooleanType
access::  Vector(a) -> IntType -> a
drop::  Vector(a) -> IntType -> Vector(a)
if_else::  BooleanType -> a -> a -> a
Note:  the functions below excludes BooleanType and generalises to
IntType, DoubleType and VectorType as shown below:
add ::  a -> a -> a
add ::  Vector(a) -> Vector(a) -> Vector(a)
multiply ::  a -> a -> a
multiply ::  a -> Vector(a) -> Vector(a)
divide ::  a -> a -> a
power ::  a -> a -> a
```

The meaning of most of these built-ins should be self-explanatory to the reader. However, the ones which might not be readily comprehensible are briefly discussed below.

map is a well known pattern in functional programming. It applies a given function f to each element of its input vector and returns a vector of results in the same order.

fold uses a binary operation f on a vector a, the initial element and a list of elements of the same type as the vector to combine all elements of the input vector. To allow for an efficient implementation the function f is required to be associative and commutative.

zip fuses two vectors or a vector and a scalar or scalar and a vector into a vector of pairs.

[2]Currying is a process, wherein a function that takes multiple arguments is transformed into a function that takes a single argument and returns another function which takes further arguments. This technique is advantageous in the implementation of higher-order functions.

greater than takes two expressions and depending on whether both these expressions are values or variables/ complex expressions return an expression which is essentially an if statement of type boolean.

E.g. :

`greaterThan(x, y) returns - if (x > y) return true else return false.`

`greaterThan(1, 2) returns - if (1 > 2) return true else return false and since both the input expressions are constant values the program is evaluated to return false.`

access takes a vector (either varible-size or fixed-length) and an index *i*. It returns the *i*-th element in the vector.

drop takes a vector (either variable-size or fixed-length) and an index *i*. It returns a new vector without the element at the given index.

4.6 Example Programs

We present four example programs in dd$\tilde{x}$. In order to avoid causing a clutter of AST nodes names, we do so by using a pretty-printed, Haskell-like representation instead:

```
dotProduct ::  [DoubleType] -> [DoubleType] -> DoubleType
dotProduct x⃗ y⃗ = fold(+, 0, map(×, zip(x⃗, y⃗))

max ::  Expr -> Expr -> Expr
max x y = ifElse(greaterThan(x, y), x, y)

additionVectors ::  [DoubleType] -> [DoubleType] -> [DoubleType]
additionVectors x⃗ y⃗ = map(+, zip(x⃗, y⃗))

multScalarVector ::  DoubleType -> [DoubleType] -> [DoubleType]
multScalarVector x y⃗ = map(×, zip(x, y⃗))
```

dotProduct takes two equal-length vectors and produces a program that represents the sum of the products of the corresponding entries in the two vectors. First of all, *zip* is applied to the two input vectors to produce a single vector of pairs. Then *map* applies multiplication to the corresponding values in each of the pairs and produces a vector of scalars. Finally, *fold* applies addition to combine the vector elements.

max finds out the maximum value out of the two input values. The result is an *IfElse* program instead of a constant value to take into account when the input values are variables.

additionVectors adds two equal-length sequences and return a new sequence of the same length as the input sequences. First of all, *zip* is applied to the two input vectors to produce a single vector of pairs. Then *map* applies addition to the corresponding values in each of the pairs and produces a vector of scalars.

AD Operation	Type
Differentiation	$(DoubleType \rightarrow DoubleType) \rightarrow DoubleType \rightarrow (DoubleType \rightarrow DoubleType)$
Gradient	$(Vector \rightarrow DoubleType) \rightarrow Vector \rightarrow (Vector \rightarrow Vector)$
Jacobian Matrix	$(Vector \rightarrow Vector) \rightarrow Vector \rightarrow (Vector \rightarrow Matrix)$

Table 4.1: The differentiation functionality provided by the AD module in d$\tilde{x}$

multScalarVector multiplies a sequence by a scalar. This is done by first applying *zip* on the scalar and the vector, producing a vector of pairs, wherein each pair has the scalar as its first element. Then *map* applies multiplication to the corresponding values in each of the pairs and produces a vector of scalars.

4.7 Differentiation

This section shows the differentiation design in d$\tilde{x}$. d$\tilde{x}$ has the capability of using both symbolic as well as automatic differentiation to compute the derivative of an arbitrary function `f`, consisting of scalars or vectors. `f` is differentiated with respect to another expression `x` (referred to as the *independent variable*), which just like `f` can be an arbitrary scalar or vector.

Table 4.1 gives a high-level overview of differentiation operations supported in d$\tilde{x}$. As can be seen from the table, if the independent variable is a scalar, the operation returns a function that produces a scalar. On the other hand, if the independent variable happens to be a vector, the operation will return a function that produces a vector (with the same number of elements as the independent variable), known as the **gradient**. However, if the expression `f` itself is a vector as well as the independent variable `x`, the result is a function that produces a square matrix, called a **Jacobian matrix**. In this case, both `f` and `x` are required to have the same number of elements. So, the number of elements in the matrix is equal to the product of the number of elements in either `f` or `x` with itself.

Chapter 5 further explains the implementation-level details of differentiation in d$\tilde{x}$ along with examples.

4.8 Summary

The design decision to make d$\tilde{x}$ functional was taken in order to have d$\tilde{x}$ allow its programs to correspond closely with the mathematical specification they implement. Its higher-order functional nature makes it expressive enough for operating on vectors. It has a flexible type system, and the typing rules are enforced using a type checker. Any program in d$\tilde{x}$ is a combination of function application and composition, let-constructs, and conditionals. d$\tilde{x}$ supports both built-in functions like map, fold and zip. These functions help enable vector-matrix operations in the language. There are also anonymous functions in the form of lambdas and let bindings are used as syntactic sugar for the lambda expressions. Furthermore, the differentiation capabilities provided by dd$\tilde{x}$ is also shown in this chapter. Overall, dd$\tilde{x}$ has a simple design, which means new functionalities can be added to the language with much ease.

Chapter 5

Implementation

This chapter discusses the implementation level details of dd$\tilde{x}$. The organisation of this chapter closely corresponds to the organisation of Chapter 4. It also includes snippets of the source code as well as pseudo-code to aid the reader's understanding. As the reader progresses through the sections, they shall observe that all the different components in dd$\tilde{x}$ rely on a general mechanism involving pattern matching and recursion. However, the underlying functionality may differ based on the required behaviour.

5.1 Implementation Decisions

dd$\tilde{x}$ is implemented in Scala based on the following rationale:

1. One of the great strengths of Scala is implementing programming languages. Scala makes it possible to design a full-featured interpreted language very quickly. Scala supports programming techniques like pattern matching which have been used extensively in the implementation of dd$\tilde{x}$.
2. Scala combines object-oriented and functional programming in one concise, high-level language. This feature has been particularly beneficial as dd$\tilde{x}$ is implemented in a functional style. The functional aspects of Scala allow the implementation to correspond to the mathematical specification closely. The source code, as a result, has been simpler, shorter and easy to understand.
3. Scala, built on top of the Java Virtual Machine, provides compatibility and interoperability with Java

5.2 Type System

Types in dd$\tilde{x}$ are represented as Algebraic Data Types (ADTs) in Scala. ADTs are a way of structuring data. They are widely used in Scala predominantly because they work well with pattern matching. ADTs also make it easy to make illegal states impossible to represent. Listing 5.1 shows the type system in dd$\tilde{x}$.

```
trait Type extends IR
trait Scalar extends Type
case object DoubleType extends Scalar
case object IntType extends Scalar
case object BooleanType extends Scalar
case class PairType (et1: Type, et2: Type) extends Type
case class VectorType (et: Type, len: Int) extends Type
case class FunctionType (in: Type, out: Type) extends Type
```

Listing 5.1: Overview of Implementation of the type system in dd$\tilde{x}$

5.3 Type Checking

Two possible approaches to implement type checking in dd$\tilde{x}$ were considered: a) have the different nodes in the syntax tree implement their own type methods, b) have a separate generic pass to enforce typing rules (refer to Section 4.3). The latter approach is applied in dd$\tilde{x}$.

The type checker ensures that that type errors are kept to a minimum in any program written in dd$\tilde{x}$. It does so by inferring the type of an expression written in dd$\tilde{x}$ by recursively checking the types of the sub-expressions match. Listing 5.2 gives an example of how this is implemented.

The code snippet shows how an expression like $f = ((a+b)+c)$ is type-checked. The type checker first deduces the expression type using pattern matching. Then it recursively type-checks each argument to ensure that they have matching types before returning the expression type. Other kinds of expressions are type-checked in a similar style.

```
  def typeCheck(e: Expr): Type = {
    e match {
      //the expression is a double literal
      case DoubleLiteral(d) => DoubleLiteral(d).t //returns DoubleType
      //the expression represents the addition operation
      case FunctionCall(FunctionCall(_: AddDouble, arg2), arg1) =>
        //check the types of the two arguments match else throw an error.
        assert(typeCheck(arg1) === typeCheck(arg2))
        typeCheck(arg1)//return the type of the expression.
     }
  }
```

Listing 5.2: An example of implementing type checking in dd$\tilde{x}$.

5.4 Abstract Syntax Tree

A dd$\tilde{x}$ program is built up by composing subexpressions into more complex

```
trait Expr extends IR {
   var t: Type
   override def build(newChildren: Seq[IR]): Expr
}
case class FunctionCall(f: Expr, arg: Expr) extends Expr {
   override var t: Type = FunctionType(f.t,arg.t)
   var function: Expr = f
   var argument: Expr = arg
}
case class Variable extends Expr
case class Vector extends Expr
trait Function extends Expr
...
trait AnonymousFunction extends Function
trait BuiltInFunctions extends Function
...
```

Listing 5.3: Abstract Syntax Tree in dd$\vec{x}$.

expressions. Listing 5.3 illustrates how this is done. As can be seen, the *Expr* class is extended by *Function*, *Variable*, *Literals*, *Vector* etc. Then the *Function* class is extended by *Built-In Functions* and *Anonymous Functions* and so on.

5.5 Evaluator

The implementation of dd$\vec{x}$ comes with an evaluator. This evaluator traverses the AST of an algebraic expression to build the corresponding result of the given program. The result represents a simpler, more compact expression written in dd$\vec{x}$. Instead of the different AST nodes implementing an *evaluate* method for program computations, the evaluator implements this feature by using *pattern matching*. It computes the result of a complex expression by decomposing it into simpler sub-expressions and recursively evaluating those. The use of pattern-matching avoids unnecessary blocks of repeated code and makes the evaluator simple and straightforward. Furthermore, this approach makes the evaluation of nested expressions easier.

A snippet of the code implementing the functionality to support evaluation of expressions like in the example below is shown in listing 5.4.
Let us consider two vectors $\vec{x}$ and $\vec{y}$,

$$\vec{x} = [x_0, x_1, x_2]$$

$$\vec{y} = [y_0, y_1, y_2]$$

We want the evaluator to evaluate the addition of these two vectors, i.e.:

$$f = \vec{x} + \vec{y}.$$

The evaluator first deduces the type of the expression and then goes on to infer the type of each argument. In this case, each argument has a vector type, so the evaluator zips

```
//mapping from variables to their values
val paramToArg = HashMap[Expr, Expr]()
//Notice the "hm" parameter is optional. used when variables in an expr
    are mapped to values.
def eval(e: Expr, hm: HashMap[Expr, Expr] = HashMap[]()): Expr {
  e match {
    //the expression is an addition operation
    case FunctionCall(FunctionCall(_: Add, arg1), arg2) =>
      //arg1 = [x1, x2, x3], arg2 = [y1, y2, y3]
      (arg1, arg2) match {
          .
          .
          //Step 1: both arguments are vectors
          case (_: Vector, _: Vector) =>
            //Step 2: the vectors are zipped up producing a single
                vector of pairs
            var array: Expr = eval(Zip(arg1, arg2))
            //Step 3: the elements in each pair are added up using map
                producing the result.
            var x = Param("x")
            var y = Param("y")
            array = eval(Map(Pair(x, y), x + y, array))
            array
          .
          .
          //for nested expressions, evaluate recursively.
          case (_, _) => eval(arg1) + eval(arg2)
      }
  }
}
```

Listing 5.4: Evaluation of addition of two vectors.

up the two input vectors to produce a single vector of pairs. Then it uses the built-in map to add up the elements in each pair producing the resulting vector,

$$f = [x_0 + y_0, x_1 + y_1, x_2 + y_2].$$

Other kinds of expressions are evaluated similarly.

5.6 Implementation of Symbolic Differentiation

dd$\tilde{x}$'s mechanism of using symbolic differentiation to differentiate algebraic expressions is discussed here. The implementation structure closely mirrors that of the evaluator described in section 5.5 and relies on pattern matching and recursion to carry out the differentiation process. The machinery for the differentiation of expressions relies on systematically applying the chain rule of calculus at the elementary operator level. The rule itself can be stated as follows:

$$(f \circ g)'(x) = (f' \circ g)(x) \times g'(x)$$

Consider a vector $\vec{x} = [x_0, x_1]$.

The max of vector $\vec{x}$ in dd$\tilde{x}$ is given by the following statement:

```
If_Else(greaterThan(x0, x1), x0, x1)
```

which is equivalent to the piece of Scala code below.

```
    if(x0 > x1) return x0 else return x1
```

Listing 5.5 shows how the max of a vector is differentiated with respect to (w.r.t.) the vector itself. It captures the general procedure that is employed to carry out a symbolic differentiation operation in dd$\tilde{x}$. First, the types of the inputs are deduced. Then the hashmap is checked to see if the inputs have any values associated with them. This is followed by differentiating the input w.r.t. the independent variable recursively.

```
//expr is what needs differentiated, withRespectTo is the independent
    variable and hm contains variable to value mapping
def differentiate(expr, withRespectTo, hm) = {
 expr match {
      .
      .
      //when expr is an if else statement.
      case If_Else(cond, stmt1, stmt2) =>
        withRespectTo match {
            //withRespectTo is a one-dimensional varible, e.g. x, y.
            case param: Param => ...
            //withRespectTo is a vector, in this case: [x0, x1]
            case vec: Vector =>
                var result: Seq[Expr] = Seq() //will store the result
                //checks if the statements have a value for them in hm
                arg1 = if (hm.contains(stmt1)) hm(stmt1) else stmt1
                arg2 = if (hm.contains(stmt2)) hm(stmt2) else stmt2
                // the if stmt is differentiated w.r.t each vector element
                    and resulting value is added to the "result".
                for(x <- vec.list) {
                  result = result:+ (If_Else(cond, differentiate(arg1, x,
                      hm), differentiate(arg2, x, hm)))
                }
                //the result is written in ddx and returned
                return Vector(result, stmt1.t)
        }
 }
```

Listing 5.5: Differentiating max of a vector w.r.t. the vector: Symbolic Differentiation

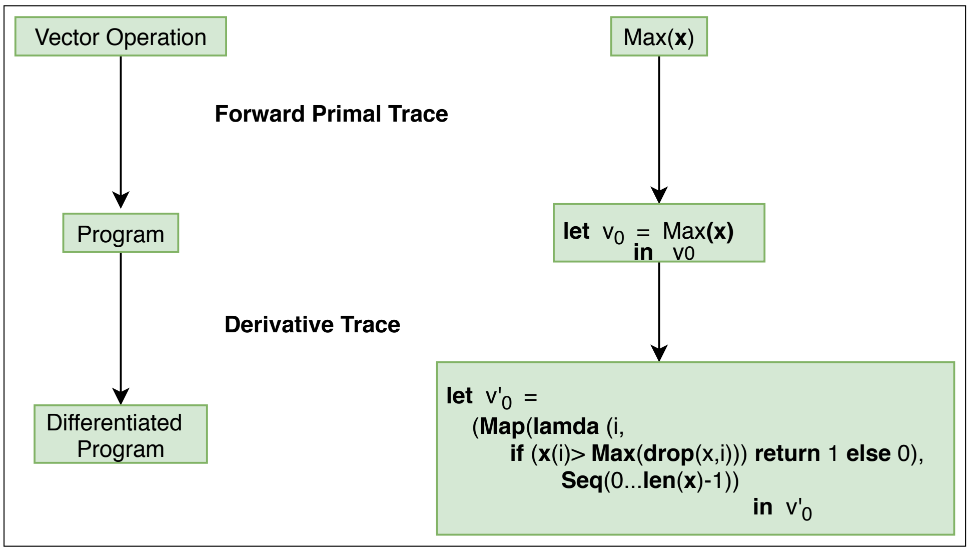


Figure 5.1: The AD pipeline in dd$\tilde{x}$ for AD of $Max(\vec{x})$ w.r.t $\vec{x}$. The L.H.S. gives a high-level view of the process and the R.H.S gives a representation of how the program looks like at each stage. The AST of the input expression is traversed to produce the forward primal trace. This trace is then traversed to produce the resulting derivative trace.

The result once the symbolic differentiation process is over is a vector of `If_Else` statements and looks like this:

```
[if(x0 > x1) return 1 else return 0, if(x0 > x1) return 0 else return 1]
```

Listing 5.6: Result of differentiating max of a vector: symbolic differentiation in dd$\tilde{x}$.

The reader should observe that this is the case for a vector with just two elements in it. The resulting vector will get larger with more complicated and nested `If_Else` statements as the vector size increases. This property, as we shall later see in Chapter 6, proves to be detrimental for the run-time of both the symbolic differentiation of larger vectors as well the result's evaluation given an input vector.

5.7 Forward mode Automatic Differentiation

dd$\tilde{x}$ relies on AST-to-AST transformation for implementing forward mode automatic differentiation. We start with an algebraic expression which is translated into an intermediate representation (IR). There is then a two step process which involves the use of two different passes. The first pass is a forward primal trace pass. It takes the original program AST and introduces temporary v_i variables using the Let construct described in Chapter 4. The resulting program is the *forward primal trace*, like explained in Chapter 2.

The second pass then traverses the newly created forward primal trace and produces a program which consists of let constructs with the new v_i' variables. This new program is known as the *derivative trace*. These v_i' variables correspond to the derivative of each variable in the forward primal trace.

Let us consider a simple example. Max of a vector $\vec{x}$ of size N could be represented as:

$$f = Max(\vec{x})$$

We wish to differentiate f w.r.t. $\vec{x}$. Figure 5.1 fittingly captures the high-level view of how automatic differentiation for the given example is achieved in dd$\tilde{x}$. Furthermore, listings 5.7 and 5.8 provide a more detailed look into the process.

Listing 5.7 shows how the forward primal trace for f is computed in dd$\tilde{x}$. Three global data structures are used: counter, x and hm. The counter stores the number of intermediate variables introduced. We can see that the counter is incremented every time a new primal is added to the trace. The primal trace itself is initially stored in x, a queue of pairs with the first element corresponding to v_is and the second element to the sub-expressions. A hashmap, hm, is also maintained, and it contains sub-expressions to primals mapping. It allows $O(1)$ access to the details of a primal that has already been added to the trace. This is useful for when traversing complex, nested expressions.

Once traversing the expression AST is finished, we have the queue, `x`, fully populated with all the primals. The pairs in this queue are processed in reverse, and a program made of nested let-constructs is computed. This program is the forward primal trace. For the example we have, the forward primal trace will be:

```
Let v0 = Max(x) //x is a vector
    in v0
```

Note that this particular trace has just one primal. This is due to the fact, that unlike symbolic differentiation, the result of AD is not dependent on the vector size.

Listing 5.8 illustrates how the forward primal trace obtained in Listing 5.7 is used to compute the derivative trace which is used to differentiate the initial expression. This pass only uses a single data structure, `paramToArg`. `paramToArg` is a hashmap which is populated with v_is to sub-expressions mappings whilst the forward primal trace is traversed. Once the hashmap is fully populated, the derivative trace is constructed in a similar manner to that in Listing 5.7. However, this time, each v_i instead corresponds to the derivative of the sub expression w.r.t. the independent variable. This results in a program representing the derivative trace using nested Let-constructs.

The derivative trace of the example will look like this:

```
Let v0' = Map(lambda (i,
              if (x[i] > max(Drop(x,i))) return 1 else 0),
              Seq[0...N-1]) //N is the length of the input vector
                      in v0'
```

It is worth noting here that, unlike in the case of symbolic differentiation (Listing 5.6), the resulting program here, i.e. the derivative trace for any other vector, regardless of its size, will be the same. The only thing that will vary would be `N` and the body of `Map` depending on the kind of vector operation. For example, simply differentiating a

```
 //keeps the count of the number of intermediate variables used.
 var counter = 0
 //A queue of pairs, elem_1 in each pair corresponds to vi and elem_2
 //corresponds to the sub-expression
 val x = Queue[(Expr, Expr)]()
 //hm is used to store the primals and allows differentiation of
    expressions
 //consisting of nested operations
 var hm = mutable.HashMap[Expr, Expr]()

 def autoDifferentiate(e: Expr, withRespectTo: Expr): Expr = {
    //step 1: forwardPrimal Trace: see listing 5.7 for more details
    forwardPrimalTrace (e)
    // for x * y, reverseQueue = [(v_2, v_0 * v_1),(v_1, b),(v_0, a)]
    var reverseQueue = x.reverse
    var z = e
    //using lets to represent the forward primal trace
    //in this case, z will be Let(v0, x, Let(v1, y, Let(v2, v0*v1, e)))
    reverseQueue.foreach (x => z = (Let(x._1, x._2, z) ))
    //derivative trace: see listing 5.8 for further details
    z = DerivativeTrace.derivativeTrace (z, withRespectTo, reverseQueue)
    //clear the data structures, i.e., x, counter and hm.
    return z
 }

 def forwardPrimalTrace(e: Expr): Unit = {
   e match {
       .
       .
     //for when we wish to differentiate max of a vector
      case Max(x: Vector) =>
        // yields x =[(v_0, Max(x))]
        x.addOne(Param("v_".concat(counter.toString)), MaxVar(x))
        hm.put(MaxVar(x), Param("v_".concat(counter.toString)))
        counter = counter + 1 //counter = 1
        .
        .
   }
}
```

Listing 5.7: Differentiating max of a vector w.r.t. the vector: forward mode AD. Computing the forward primal trace. Another example of computing the forward primal trace of a scalar operation is given in Appendix A

```
//stores the mapping from intermediate variables to their values in the
//forward primal trace for easy access
val paramToArg = mutable.HashMap[Expr, Expr]()
//e is the forward primal trace, withRespectTo the independent variable,
    queue has the primals
def derivativeTrace(e: Expr, withRespectTo: Expr, queue): Expr = {
    e match {
     //the forward primal trace is traversed and paramToArg populated
      case FunctionCall(Lambda(param, body), arg) =>
        paramToArg.put(param, arg.asInstanceOf[Expr])
        derivativeTrace(body, withRespectTo, queue)

     //the derivative trace is built
      case _ =>
        //z_prime = v_2
        var z_prime: Expr = queue.apply(0)._1
        //z_prime = Let(v_0', Map(lambda (i, if else.., Seq(0..N-1))),
            v_0'))
        //Note N is the size of the the length of withRespectTo, which in
            this case is vector x, so N is the length of x.
        queue.foreach(x => z_prime = Let(x._1,
            differentiate(paramToArg(x._1), withRespectTo, paramToArg),
            z_prime))
        paramToArg.clear() //clear the data structures
        return z_prime //return the derivative trace
    }
}
```

Listing 5.8: The derivative trace of $f = Max(\vec{x})$ w.r.t $\vec{x}$ in dd$\tilde{x}$

vector $\vec{y}$ w.r.t. itself, will result in the derivative trace in Listing 5.9 which computes the Jacobian matrix:

```
Let v0' = Map(lambda (i,
            Map(lambda (j, if(i==j) return 1 else 0,
               Seq[0...N-1])),
                   Seq[0...N-1]) //N is the size of the vector y.
                        in v0'
```

Listing 5.9: Result of differentiating a vector $\vec{y}$ w.r.t. itself: AD in dd$\tilde{x}$.

This property of AD, as we shall observe later in Chapter 6, is what makes it more efficient for array processing as opposed to symbolic differentiation.

```
def printStr(e: Expr, hm: mutable.HashMap[Expr, Expr]): Expr = {
     e match {
      case FunctionCall(FunctionCall(_: PowerDouble, arg1), arg2) =>
        (arg1, arg2) match {
          //for expressions like x^2, 2^x, x^x...
          case (arg1: Param, _) => Param("(" + arg1 + " ^ " + arg2 + ")")
          case (_, arg2: Param) => Param("(" + arg1 + " ^ " + arg2 + ")")
          //for expressions like, 2^2, i.e, constants.
          case (DoubleLiteral(arg1), DoubleLiteral(arg2)) =>
            val exponent = java.lang.Double.valueOf(arg2)
            val base = java.lang.Double.valueOf(arg1)
            return DoubleLiteral(scala.math.pow(base, exponent))
         //for expressions with nested operations
          case (_, _) => printStr((printStr(arg1) ^ printStr(arg2)))
        }
    }
}
```

Listing 5.10: Printing exponents in dd$\tilde{x}$

5.8 Printer

The previous sections show how dd$\tilde{x}$ achieves evaluation of expressions, symbolic differentiation and forward mode AD using AST-to-AST transformations. However, the programs produced need to be presented to the user in a way that is easy to understand. Hence, the interpreter has a *printer* which represents the expressions in dd$\tilde{x}$ in a user-friendly, pretty-printed format. This feature enhances usability and aids greatly in debugging. The printer, just like the evaluator and the differentiation modules, uses pattern matching to operate.

An example of how this is done can be seen in the short excerpt in Listing 5.10. The code shows how exponents are printed in dd$\tilde{x}$. Just like every other component of dd$\tilde{x}$, the type of the expression is first inferred. If either of the arguments is of the type parameter, the pretty-printed version of the expression is created right away. For the case when the expression represents a constant, it is simplified further and then returned. For any other cases, each sub expression is recursively traversed until finally resulting in the pretty-printed version of the original expression.

5.9 Summary

Scala has excellent support for the functional programming paradigm. Scala's functional aspects made it relatively simpler and straight-forward to implement the design of dd$\tilde{x}$, an interpretable and type-checkable language. Algebraic Data Types are used to represent the types, function composition, application and abstraction are used for modelling algebraic expression in dd$\tilde{x}$. There is an *evaluator* to evaluate these expressions, which relies on pattern matching and recursion. Similar to the evaluator, there is the *symbolic differentiation* pass which is similar in structure to the Evaluator, and it uses the calculus chain rule to differentiate programs. The *forward mode AD* mechanism is divided into two main procedures captured in two separate passes. These passes transform the original program into first, a forward-primal trace, then, a derivative trace, sequentially. Finally, the *printer* provides a user-friendly way to show the end-result to the user.

Chapter 6

Evaluation

In this chapter, several benchmarks are used to evaluate the performance of dd$\tilde{x}$ in practice. This evaluation explores the following areas:

1. **Performance of the evaluator in dd$\tilde{x}$.**
2. **Run time of differentiating a program P to produce a program dP in dd$\tilde{x}$ using symbolic differentiation vs using forward mode AD.**
3. **Run time of dP obtained in** 2. **using symbolic differentiation vs using forward mode AD.**

The correctness of the evaluator is verified by running a series of tests in addition to checking it against the benchmarks listed in section 6.2. The same benchmarks are used to support the argument made in the previous chapters with regards to automatic differentiation being the more efficient method to differentiate. The results obtained are presented, and the factors influencing the results investigated and explained.

6.1 Experimental Setup

The experiments are performed using a MacBook Pro equipped with an Intel Core i5 processor running at 2.3GHz, 8GB of LPDDR3 RAM at 2133 Mhz. The operating system is OS X 10.14.6. IntelliJ IDEA 2018.2.4 is used to run Scala 2.13.1. The JDK version is 1.8.0_181. For each experiment execution time in milliseconds (ms) is measured 20 times for each input size, and the median is taken. The Scala API is used to measure the execution time.

6.2 Benchmarks

The benchmarks are inspired by Shaikhha et al. (2019). They consist of the following vector operations:

1. Differentiation of addition of two vectors with respect to the first vector is a Jacobian matrix which happens to be an identity matrix.

For example,

Input Vectors : $\vec{x} = [x_0, x_1, x_2] \quad , \quad \vec{y} = [y_0, y_1, y_2]$

Add the vectors : $\vec{f} = \vec{x} + \vec{y} = [(x_0 + y_0), (x_1 + y_1), (x_2 + y_2)]$

Jacobian Matrix :
$$\frac{d\vec{f}}{d\vec{x}} = \begin{bmatrix} \frac{df_0}{dx_0} & \frac{df_0}{dx_1} & \frac{df_0}{dx_2} \\ \frac{df_1}{dx_0} & \frac{df_1}{dx_1} & \frac{df_2}{dx_2} \\ \frac{df_2}{dx_0} & \frac{df_2}{dx_1} & \frac{df_2}{dx_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2. Differentiation of the multiplication of a vector with a scalar value with respect to the scalar value results in a vector (i.e. the gradient).
   For example,

   Input Vector : $\vec{x} = [x_0, x_1, x_2]$

   Scalar-Vector product : $\vec{f} = v \times \vec{x} = [(v \times x_0), (v \times x_1), (v \times x_2)]$

   Gradient :
$$\frac{d\vec{f}}{dv} = \begin{bmatrix} \frac{df_0}{dv} \\ \frac{df_1}{dv} \\ \frac{df_2}{dv} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

3. Differentiation of dot product of two vectors with respect to the first vector results in a Jacobian matrix with a single row.

   For example,

   Input Vectors : $\vec{x} = [x_0, x_1, x_2] \quad , \quad \vec{y} = [y_0, y_1, y_2]$

   Dot Product : $f = \vec{x} \cdot \vec{y} = ((x_0 * y_0) + (x_1 * y_1) + (x_2 * y_2))$

   Jacobian Matrix : $\frac{df}{d\vec{x}} = [\ \frac{df}{dx_0}, \frac{df}{dx_1}, \frac{df}{dx_2}] = [\ y_0,\ y_1,\ y_2]$

4. Differentiation of the maximum value of a vector with respect to the input vector. The output expression is a Jacobian matrix with a single row.

   For example,

   Input Vector : $\vec{x} = [x_0, x_1]$

Expression Kind	Examples
Addition	1. $5+2$ 2. $x+y+z$ 3. $[x_1,y_1,z_1]+[x_2,y_2,z_2]$
Multiplication	1. $5*2$ 2. $x*y*z$ 3. $2*[x_2,y_2,z_2]$
Exponents	1. 5^2 2. x^2 3. $[x_2,y_2,z_2]^2$
Complex Expressions	1. $(5+4)*2$ 2. $x*(y^2+z^2)$ 3. $5*[x_1+x_2,y_1*y_2,z_2^2]$
GreaterThan	1. greaterThan$(2,3)$ 2. greaterThan(x,y)
Map	1. map $(+1)$ $[1,2,3]$ 2. map $(*z)$ $[x,y,z]$
Fold	1. fold $(+)$ $[1,2,3]$ 2. fold $(*)$ $[x,y,z]$
Zip	1. zip $[2,4,5]$ x 2. zip y $[1,2,9]$ 3. zip $[x,y,z]$ $[a,b,c]$

Table 6.1: The different kinds of expressions used to test the correctness of dd$\tilde{x}$'s Evaluator.

Maximum value in the vector : $f = max(\tilde{x}) = \; if(x_0 > x_1) \; then \; x_0 \; else \; x_1$

Jacobian Matrix :

$$\frac{df}{d\tilde{x}} = [\ (if(x_0 > x_1)\ then\ \frac{dx_0}{dx_0}\ else\ \frac{dx_1}{dx_0})\ ,(if(x_0 > x_1)\ then\ \frac{dx_0}{dx_1}\ else\ \frac{dx_1}{dx_1})]$$

$$\frac{df}{d\tilde{x}} = [\ (if(x_0 > x_1)\ then\ 1\ else\ 0)\ ,\ (if(x_0 > x_1)\ then\ 0\ else\ 1)]$$

6.3 Performance of the Evaluator

This section evaluates the quality of the dd$\tilde{x}$'s evaluator in terms of correctness and efficiency.

6.3.1 Correctness

A series of tests written in Scala are performed to evaluate the correctness of the evaluator. These tests mainly focus on the correctness of the built-in functions,

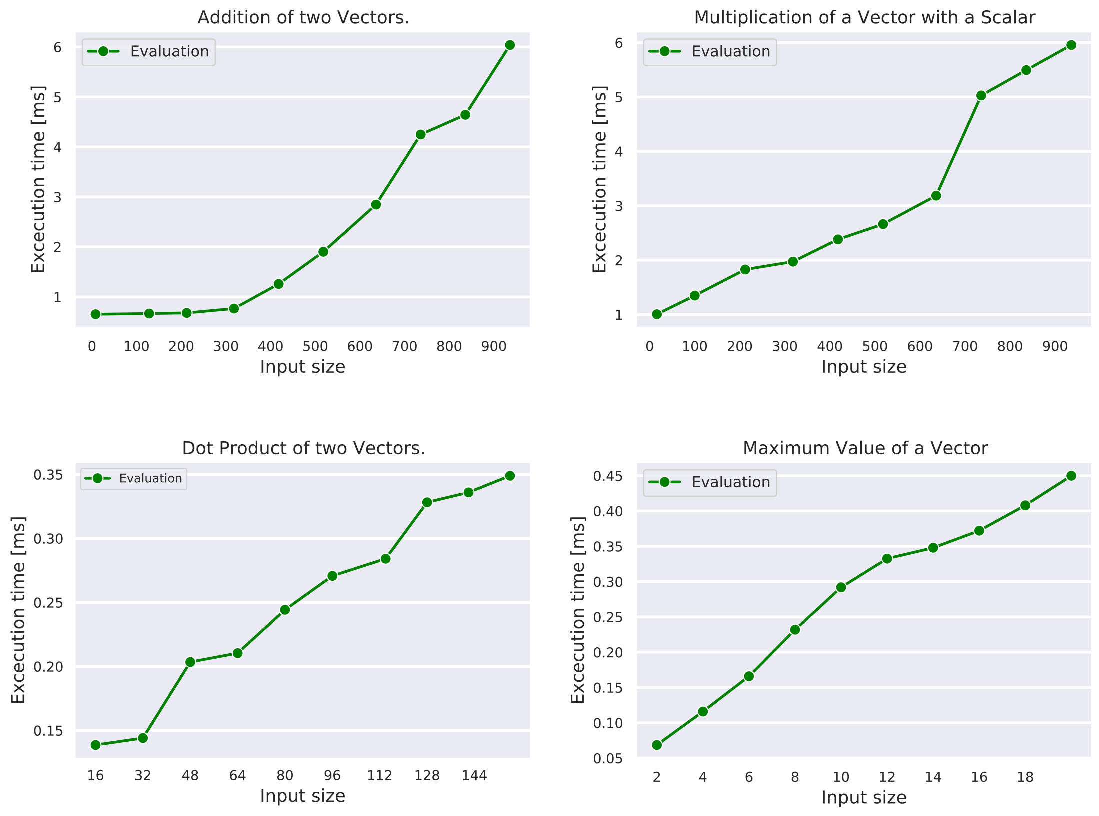


Figure 6.1: Time taken to evaluate the four different kind of vector operations as specified in the graph titles. The input size of the vectors vary as follows: ADDITION OF TWO VECTORS: The vector lengths vary between 8 and 936. MULTIPLICATION OF A VECTOR WITH A SCALAR: The vector lengths vary between 16 and 936. DOT PRODUCT OF TWO VECTORS: The vector lengths vary between 16 and 156. MAXIMUM VALUE OF A VECTOR: The vector lengths vary between 2 and 20.

anonymous functions and operations on vectors. The results obtained upon running these tests are verified against the results obtained by evaluating same operations on Scala, WolframAlpha and Wolfram Mathematica. Table 6.1 gives examples of the different kinds of expressions used to test the correctness. In addition to these tests, the microbenchmarks were also used to evaluate the correctness. The tests have a success rate of 100%.

6.3.2 Efficiency

The focus of evaluating the efficiency of the Evaluator has not been to measure how fast it can evaluate. Neither has the purpose of the Evaluator been to outperform or perform as good as existing evaluators in similar languages.

The benchmarks are primarily used to perform this evaluation. The goal of this evaluation has been to confirm that the Evaluator behaves as expected, i.e. the execution time of evaluation increases with an increase in the size of the vectors. The plots shown in Figure 6.1 confirm this.

6.4 Performance of Computing Differentiated Programs

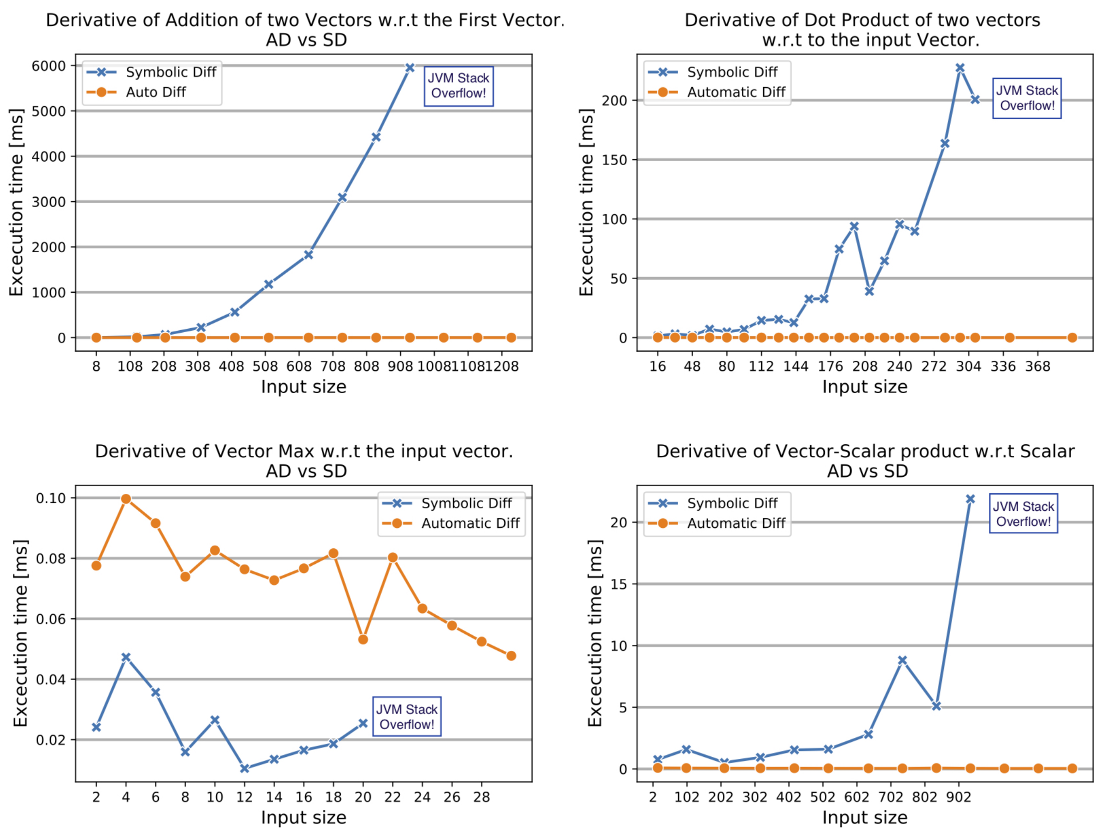


Figure 6.2: The plots show the time it takes to differentiate programs using symbolic differentiation (SD) and forward mode AD (AD), as a function of the size of the input vectors. Exact measurements in table-form can be found in Appendix B

In this section, we evaluate the efficiency of dd$\tilde{x}$'s implementations of symbolic differentiation and that of forward mode AD with the help of the benchmarks listed in section 6.2. We do so by evaluating the execution time of differentiating a program P to give a new program dP using the respective methods. The numerical results of these are plotted in Figure 6.2.

The experiments could be only run on vectors up to a certain size. This limitation was due to the fact that symbolic differentiation can only operate on fixed-size arrays. Thus, going beyond a certain size in each benchmark leads to `"JVM stack overflow"`[1] exception.

The interesting observation from the experiments is that the run-time of AD is as good or much better than the run-time of SD. In fact, AD almost always has a constant run-time (with some standard deviation). Hence, it is also able to process much larger

[1] A StackOverflowError is an exception thrown by a thread, when it's stack has no more space to add a new frame to make the next method call.

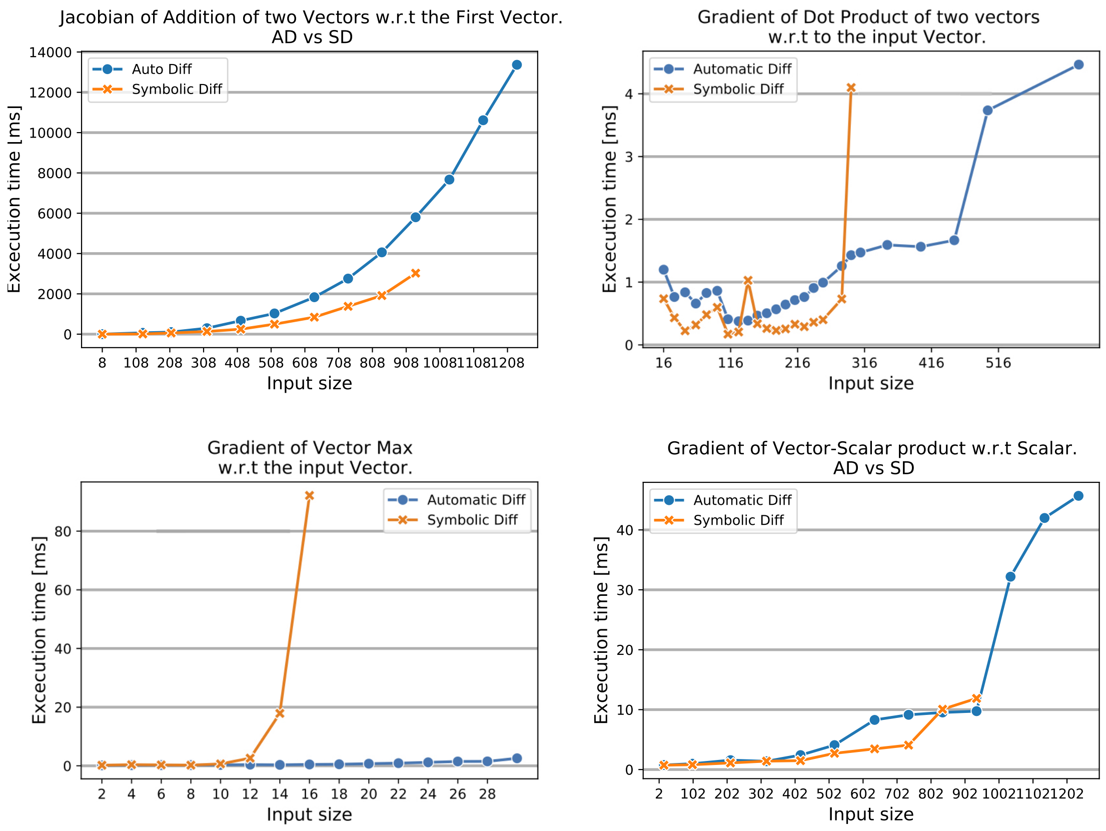


Figure 6.3: Time taken to run the programs resulting from symbolic differentiation (SD) and forward mode AD (AD), as a function of the size of the input vectors. Exact numerical measurements can be found in Appendix B.

vectors when compared to SD. This analysis further supports the argument that AD is, in fact, superior to symbolic differentiation.

This behaviour is due to the fact that symbolic differentiation is dependent on the vector size. The implementation for SD is such that, unlike AD, it needs to know the vector length as it has to sequentially differentiate each vector element. So as the vector size grows, so does the number of elements that SD has to compute on and as a consequence, the run-time also grows. This property results in a dramatically increasing run-time. On the other hand, as explained in section 5.7, AD performs similarly regardless of the vector size. Therefore, AD computes the derivative in constant time.

6.5 Performance of Running Differentiated Programs

In this section we compare the run times of differentiated programs resulting from AD and symbolic differentiation (SD). These differentiated programs are obtained in section 6.4. The numerical results for these are plotted in Figure 6.3. The reader should recall that in section 6.4, we saw how SD can only compute on vectors up to a certain size. As a direct consequence, in the evaluation in this section, it has not been possible to run symbolically differentiated programs for input vectors as large as for automatically differentiated programs.

Overall, the plots show that AD and SD show comparable performances in the beginning. However, as the vector size grows, SD starts overtaking AD, giving a much worse run-time. AD is capable of operating on much larger size vectors, and as expected shows an increasing run-time as the vector size grows.

In the case of evaluating the addition of two vectors w.r.t. the first vector, the vector input sizes vary between 8 and 1236. The result of this operation is a Jacobian matrix which is a square matrix. The plot shows how gradient computation scales similarly for forward mode AD and symbolic differentiation.

For the gradient of dot product of vectors w.r.t. the first vector, the input size of vectors used to evaluate this benchmark was between 16 and 636. As illustrated in section 6.2, the result is a Jacobian matrix which is a row vector. Therefore, forward mode AD needs to iterate over each column to compute the corresponding derivative value. The plot shows how gradient computation scales similarly for forward mode AD and symbolic differentiation in the beginning with AD giving a much better run-time as the vector size grows.

As for the gradient of vector max $w.r.t.$ the input vector, the vector input size only varies between 2 and 30. SD performs noticeably worse in this benchmark than others. This outcome was predicted in section 5.6 and section 5.7. In case of symbolic differentiation, the program is a vector of complicated, nested If_Else statements. Each If_Else statement is evaluated individually. The run-time increases dramatically as complexity of the statements increases with the increasing vector size. On the other hand, the AD-differentiated program is a simple map operation which is more straightforward to evaluate given an input vector. Therefore, in the case of AD, even though the run-time does increase with increasing vector size, it happens at a constant rate.

Finally, for the gradient of vector-scalar product w.r.t. the scalar, the vector input size varies between 16 and 1236. In this case, the result is a Jacobian matrix with a single column and the forward mode AD computes the whole Jacobian matrix in just one forward pass. The performances of forward mode AD for fixed-size vectors and SD closely resemble each other. It starts with AD having a slightly higher run-time than SD, with SD eventually performing similarly to AD until SD can no longer process larger vectors.

6.6 Summary

The evaluation has identified the following strengths of dd$\tilde{x}$:

- The Evaluator has a 100% accuracy in terms of correctness and works as expected, i.e. its run-time is directly proportional to the size of the vectors.
- The run-time for differentiating a program using AD outperforms that of SD with a significant margin. AD has an almost constant run-time and SD has a

dramatically increasing run-time.

- An automatically-differentiated dd$\tilde{x}$ program has an execution time as good as or better than a symbolically differentiated dd$\tilde{x}$ program. This supports the argument that AD has a constant time overhead of run-time of differentiated programs when compared with symbolic differentiation. (Shaikhha et al., 2019; van Merriënboer et al., 2017)

Although the evaluation itself has not identified any weaknesses of dd$\tilde{x}$, there is still much room for further improvement. The same is discussed in Chapter 7.

Chapter 7

Conclusion

This project explains the fundamentals underlying automatic differentiation and then demonstrates how those could be applied to implement an AD tool in a functional setting. This chapter gives a summary of contributions, a critical analysis of the project and future work.

7.1 Summary of Contributions

The main deliverable of this project is a functional array-processing language, dd$\tilde{x}$, with differentiation programming capabilities. As demonstrated by examples, the implementation is expressive enough for successfully evaluating complex algebraic programs and their derivatives, which also includes computing Jacobian matrices.

Just like most other standard functional languages, dd$\tilde{x}$ has a type checker and an interpreter. Its higher-order built-in primitives are what give it the power to facilitate moderately complex computations. Its simple design ensures that more such useful primitives can be added to the language with ease. This enables dd$\tilde{x}$ to cover an even broader range of algebraic expressions and also deliver practical applications in fields like machine learning in the near future.

A distinctive contribution of this project is the proof of concept that it presents. It shows how AD is much more efficient than its counterpart symbolic differentiation in a functional setting similar to $\tilde{F}$. This property, although always mentioned in studies concerning AD, is usually just simply accepted without much evidence shown and comparisons are rarely ever made. Therefore, it is certainly valuable to have more supporting evidence. Moreover, this feature makes dd$\tilde{x}$ suitable for educational purposes as it has the potential to demonstrate the caveats which concern symbolic differentiation in computer programs and how automatic differentiation overcomes those. Especially with the increasing demand for efficient automatic gradient computation, dd$\tilde{x}$ has the simplicity to be introduced to learners in early stages of their education.

7.2 Critical Analysis

The project successfully achieved its set goals. However, there is still much room improvement.

dd$\tilde{x}$ is powerful enough to evaluate a range of algebraic expressions, moderately-large vectors and their derivatives. However, the current AD mechanism provided by dd$\tilde{x}$ only supports forward mode AD. The evaluation in Chapter 6 shows how this implementation outperforms SD. However, it would certainly be interesting to also be able to compare the performance of reverse mode AD in relation with forward mode AD. Unfortunately, the limited time did not allow for extending dd$\tilde{x}$ further to support reverse mode AD. Nevertheless, it does not rule out the possibility for future extension of dd$\tilde{x}$ to provide reverse mode AD functionality.

dd$\tilde{x}$ is not designed to provide another tool useful for real-world software development. However, as mentioned already in section 7.1, dd$\tilde{x}$ could undoubtedly be used for educational purposes and presenting historical results surrounding AD. This could be rather cumbersome given dd$\tilde{x}$ does not provide a user interface at the moment. Therefore, there is a need to make dd$\tilde{x}$ more accessible such that it provides a user-friendly interface for any such activities.

7.3 Future Work

Potential for future work has been mentioned throughout the report. The priority, of course, is to enable dd$\tilde{x}$ to perform reverse mode AD. Reverse mode AD is much more efficient than forward mode when it comes to evaluating functions with a large number of inputs compared to the outputs, such as the Helmholtz free energy function (Baydin et al., 2015a). This feature would enhance dd$\tilde{x}$'s expressivity significantly and even allow it to be able to efficiently benchmark on many practical applications in the field of numerical engineering. Furthermore, it would be especially interesting to introduce novel techniques that combine features of both forward mode and reverse mode and increase the efficiency of AD. Shaikhha et al. (2019) could be taken as a starting point for this particular feature.

dd$\tilde{x}$ is powerful enough to evaluate moderately large vectors. Distributed computing techniques like MapReduce (Dean and Ghemawat, 2008) could be intergrated in the dd$\tilde{x}$'s functionality. This would allow for computing programs with significantly larger vectors with minimal performance overhead.

This project only implements a simple interpreter. However, dd$\tilde{x}$ could be extended to compile differentiated programs to a real-world language like C. This feature will enable the user to write programs and compile them once before using them multiple times. Moreover, techniques like loop fusion could be integrated with the compiler to allow for optimising the differentiated programs (Shaikhha et al., 2019).

Finally, a unique addition to the project would be adding an interactive user interface. This tool could be built such that it provides the user with a way to perform differentiation using AD in a step-by-step manner. This feature would be especially useful for educational purposes and could even be used by non-specialist programmers to explore the capabilities of AD in a more concrete manner.

Bibliography

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning, 2016.

A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey, 2015a.

A. G. Baydin, B. A. Pearlmutter, and J. M. Siskind. Diffsharp: Automatic differentiation library, 2015b.

J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-farley, and Y. Bengio. Theano: A cpu and gpu math compiler in python. In *Proceedings of the 9th Python in Science Conference*, pages 3–10, 2010.

M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications and Tools*, Philadelphia, PA, 1996. SIAM. ISBN 0–89871–385–4.

C. Bischof, P. Khademi, A. Mauer, and A. Carle. Adifor 2.0: Automatic differentiation of fortran 77 programs. *Computational Science Engineering, IEEE*, 3:18 – 32, 02 1996. doi: 10.1109/99.537089.

C. Bischof, G. Corliss, A. Griewank, M. Berz, K. Makino, K. Shamseddine, G. Hoffstatter, and W. Wan. Cosy infinity and its applications in nonlinear dynamics. 12 1997.

C. H. Bischof, H. M. Bucker, B. Lang, A. Rasch, and A. Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for matlab programs. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 65–72, 2002.

O. Breuleux and B. van Merriënboer. Automatic differentiation in myia. 2017.

B. Dauvergne and L. Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In *International Conference on Computational Science, ICCS 2006, Reading, UK*, 2006.

J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

C. Elliott. Beautiful differentiation. volume 44, pages 191–202, 08 2009. doi: 10.1145/1631687.1596579.

R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*, volume 36. 01 2002. doi: 10.1287/mnsc.36.5.519.

A. Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12 (4):30–43, Apr. 2014. ISSN 1542-7730. doi: 10.1145/2611429.2617811. URL `https://doi.org/10.1145/2611429.2617811`.

C. Grelck, K. Hinckfuß, and S.-B. Scholz. With-loop fusion for data locality and parallelism. pages 178–195, 09 2005. doi: 10.1007/11964681_11.

A. Griewank. A mathematical view of automatic differentiation. 2003.

L. Hascoet and V. Pascual. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3), May 2013. ISSN 0098-3500. doi: 10.1145/2450153.2450158. URL `https://doi.org/10.1145/2450153.2450158`.

P. Haukkanen, J. Merikoski, M. Mattila, and T. Tossavainen. The arithmetic jacobian matrix and determinant. *Journal of Integer Sequences*, 20, 09 2017.

R. J. Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Trans. Math. Softw.*, 40(4), July 2014. ISSN 0098-3500. doi: 10.1145/2560359. URL `https://doi.org/10.1145/2560359`.

M. E. Jerrell. Automatic differentiation and interval arithmetic for estimation of disequilibrium models. *Computational Economics*, 10(3):295–316, 1997.

J. Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14:35–57, 2001.

F. Luporini. Generalizing loop-invariant code motion in a real-world compiler. 2015.

D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in numpy.

C. C. Margossian. A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4):e1305, 2019.

B. Pearlmutter and J. Siskind. *Using Programming Language Theory to Make Automatic Differentiation Sound and Efficient*, volume 64, pages 79–90. 08 2008a. doi: 10.1007/978-3-540-68942-3_8.

B. A. Pearlmutter and J. M. Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2), Mar. 2008b. ISSN 0164-0925. doi: 10.1145/1330017.1330018. URL `https://doi.org/10.1145/1330017.1330018`.

S. Peles and S. Klus. Sparse automatic differentiation for large-scale computations using abstract elementary algebra. *ArXiv*, abs/1505.00838, 2015.

J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in julia, 2016.

A. Shaikhha, A. Fitzgibbon, D. Vytiniotis, and S. Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–30, Jul 2019. ISSN 2475-1421. doi: 10.1145/3341701. URL `http://dx.doi.org/10.1145/3341701`.

J. M. Siskind and B. A. Pearlmutter. Perturbation confusion and referential transparency correct functional implementation of forward-mode ad, 2005.

J. M. Siskind and B. A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6): 1288–1330, Sep 2018. ISSN 1029-4937. doi: 10.1080/10556788.2018.1459621. URL `http://dx.doi.org/10.1080/10556788.2018.1459621`.

F. Srajer, Z. Kukelova, and A. Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*, 33(4-6):889–906, Feb 2018. ISSN 1029-4937. doi: 10.1080/10556788.2018.1435651. URL `http://dx.doi.org/10.1080/10556788.2018.1435651`.

B. van Merriënboer, A. B. Wiltschko, and D. Moldovan. Tangent: Automatic differentiation using source code transformation in python, 2017.

B. van Merriënboer, O. Breuleux, A. Bergeron, and P. Lamblin. Automatic differentiation in ml: Where we are and where we should be going, 2018.

F. Wang, J. Decker, X. Wu, G. Essertel, and T. Rompf. Backpropagation with continuation callbacks: Foundations for efficient and expressive differentiable programming. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 10201–10212, Red Hook, NY, USA, 2018. Curran Associates Inc.

M. J. Weinstein and A. V. Rao. Algorithm 984: Adigator, a toolbox for the algorithmic differentiation of mathematical functions in matlab using source transformation via operator overloading. *ACM Trans. Math. Softw.*, 44(2), Aug. 2017. ISSN 0098-3500. doi: 10.1145/3104990. URL `https://doi.org/10.1145/3104990`.

Appendix A

Implementation: AD scalar operation example

```
def forwardPrimalTrace(e: Expr): Unit = {
    e match {
        .
        .
        .
        case FunctionCall(FunctionCall(_: MultiplyDouble, arg1), arg2) =>
            {
            (arg1, arg2) match {
                 //both args are parameters in this case, arg1: x, arg2: y
                 case (_: Param, _: Param) =>
                   // yields x =[(v_0, a)]
                   x.addOne((Param("v_".concat(counter.toString))), arg1)
                   hm.put(arg1, Param("v_".concat(counter.toString)))
                   counter = counter + 1 // counter = 1
                   // yields x =[(v_0, a), (v_1, b)]
                   x.addOne((Param("v_".concat(counter.toString))), arg2)
                   hm.put(arg2, Param("v_".concat(counter.toString)))
                   counter = counter + 1 // counter = 2
                   //yields x =[(v_0, a), (v_1, b), (v_2, a*b)]
                   x.addOne(Param("v_".concat(counter.toString)),
                       ((x.apply(x.knownSize-1)._1) * ((
                       x.apply(x.knownSize-2)._1))))
                   hm.put(((x.apply(x.knownSize-3)._2) * ((
                       x.apply(x.knownSize-2)._2))),
                       (Param("v_".concat(counter.toString))))
                   counter = counter + 1 //counter = 3
               .
               .
               .
                //handles nested operations by recursively computing the
                    primals
                case (_, _) =>
                   forwardPrimalTrace(arg1)
                   forwardPrimalTrace(arg2)
                   x.addOne(Param("v_".concat(counter.toString)),
                       hm(arg1)*hm(arg2))
                   hm.put(arg1 * arg2,
                       Param("v_".concat(counter.toString)))
                   counter = counter + 1
        }
    }
}
```

Listing A.1: Computing primals for the forward primal trace of $f = a \times b$ w.r.t. a in dd$\tilde{x}$

Appendix B

Evaluation: Numerical Results

Vector size	**Diff: AD**	**Diff: SD**	**Eval: AD**	**Eval: SD**
8	0.2560	0.22946	0.9291	0.3043
128	0.2377	19.0383	66.5092	9.6787
212	0.2418	68.8016	102.5071	58.2684
318	0.1637	224.0104	296.0637	133.6428
418	0.1974	562.0310	667.3266	246.8746
518	0.2179	1176.6695	1028.1939	496.0434
636	0.2308	1828.0116	1833.5542	843.4450
736	0.2370	3092.9064	2757.1777	1380.3548
836	0.2714	4422.3525	4059.9162	1915.7682
936	0.1574	5952.9072	5800.2073	3026.7400
1036	0.3197	N/A	7669.4046	N/A
1136	0.2801	N/A	10613.7956	N/A
1236	0.2263	N/A	13365.8888	N/A

Table B.1: Numerical results for addition of two vectors w.r.t. the first vector. The time is measured in milliseconds. "Diff" corresponds to the evaluations in section 6.4 and "Eval" corresponds to section 6.5.

Vector size	**Diff: AD**	**Diff: SD**	**Eval: AD**	**Eval: SD**
16	0.1286	1.4645	1.1979	0.7352
32	0.0986	3.1815	0.7628	0.4312
48	0.0780	1.8243	0.8371	0.2244
64	0.0725	7.1486	0.6582	0.3175
80	0.0633	4.8248	0.8264	0.4804
96	0.0531	6.8670	0.8612	0.5975
112	0.0854	14.4993	0.4083	0.1703
128	0.0506	15.3678	0.3758	0.2052
142	0.0487	12.6503	0.3867	1.0282
156	0.04054	32.5518	0.4652	0.3361
170	0.0384	32.8103	0.5041	0.2610
184	0.0324	74.6749	0.5670	0.2296
198	0.0350	93.8332	0.6423	0.2544
212	0.0285	38.9946	0.7136	0.3281
226	0.0274	64.6966	0.7623	0.2902
240	0.0297	95.4356	0.9065	0.3602
254	0.0243	89.6133	0.9917	0.4004
282	0.0290	163.6741	1.2565	0.7307
296	0.0259	227.4465	1.4289	1.3826
310	0.0241	200.5902	1.4715	4.1005
342	0.0243	N/A	1.5923	N/A
400	0.0219	N/A	1.5635	N/A
450	0.0233	N/A	1.6667	N/A
500	0.0269	N/A	3.7361	N/A
636	0.0169	N/A	4.4654	N/A

Table B.2: Numerical results for differentiation of dot of two vectors w.r.t. the first vector. The time is measured in milliseconds. "Diff" corresponds to the evaluations in section 6.4 and "Eval" corresponds to section 6.5.

Vector size	Diff: AD	Diff: SD	Eval: AD	Eval: SD
2	0.0776	0.0209	0.1611	0.2052
4	0.0997	0.0681	0.2306	0.3910
6	0.0916	0.1075	0.2360	0.3153
8	0.0739	0.1888	0.2402	0.2302
10	0.0826	0.0412	0.3096	0.6559
12	0.0764	0.0201	0.4070	2.6170
14	0.0727	0.0187	0.3317	24.2467
16	0.0766	0.0202	0.4725	71.2307
18	0.0817	0.0227	0.5355	300.5129
20	0.0531	0.0232	0.7281	1319.0039
22	0.0803	N/A	0.9069	N/A
24	0.0634	N/A	1.1902	N/A
26	0.0578	N/A	1.4834	N/A
28	0.0524	N/A	1.5134	N/A
30	0.0477	N/A	2.5685	N/A
212	0.0517	N/A	68.7847	N/A

Table B.3: Numerical results for differentiation of max of vector w.r.t. the vector itself. The time is measured in milliseconds. "Diff" corresponds to the evaluations in section 6.4 and "Eval" corresponds to section 6.5.

Vector size	Diff: AD	Diff: SD	Eval: AD	Eval: SD
16	0.0863	0.7671	0.7671	0.7289
100	0.0746	1.5786	0.9733	0.8098
212	0.0650	0.5199	1.5700	1.1090
318	0.0610	0.9395	1.3810	1.4061
418	0.0635	1.5456	2.4120	1.4859
518	0.0533	1.6020	4.0801	2.7129
636	0.0506	2.8060	8.2864	3.4534
736	0.0477	8.8146	9.1391	4.0777
836	0.0767	5.0974	9.5263	10.0654
936	0.0543	21.8943	9.7466	11.8791
1036	0.0438	N/A	32.1916	N/A
1136	0.0441	N/A	41.9909	N/A
1236	0.0451	N/A	45.6914	N/A

Table B.4: Numerical results for vector scalar product w.r.t. scalar. The time is measured in milliseconds. "Diff" corresponds to the evaluations in section 6.4 and "Eval" corresponds to section 6.5.