Application of Combinatory Automatic Differentiation in a Python Context

Bachelor Project 2024–25

Department of Computer Science
University of Copenhagen

Kristian Laudrup Hansen
Supervised by Martin Elsman

March 18, 2025

Contents

Abstract

This project provides a Python context implementation of the Caddie tool and explains the underlying theory of Combinatory Automatic Differentiation. A general notion of derivatives and Automatic Differentiation is provided, and explanatory examples of how Combinatory Automatic Differentiation uses point-free notation and monad types to implement symbolic differentiation are presented. A subset is defined for the Python context, and the tool is implemented for a Gradient Descent optimization task to test the tool's potential.

1 Introduction

One of the key elements to the success of Neural Networks is the technique of Automatic Differentiation. Automatic Differentiation automates the computation of partial derivatives of functions, which in Neural Networks are used to optimize the parameters of the network. Derivatives and Automatic Differentiation are also used in many other fields such as finance, physics, and economics, where derivatives are being used to observe changes or sensitivities of complex structures.

This project considers the theory of Combinatory Automatic Differentiation, which uses a point-free compositional approach to symbolic differentiation as the basis for automatically computing derivatives. The tool that implements this theory is called Caddie, and the aim of the tool is to generate efficient derivative code to a domain-specific programming language.

In the project, the underlying theory of Caddie will be explained and modifications to the tool will be implemented to make it work in a Python language context. Python is commonly used to implement Neural Networks and has many frameworks, such as PyTorch and TensorFlow, which provide functionality for implementing neural network architectures. Python is therefore an adequate choice for investigating Caddie's functionality to derive differentiated code.

1.1 Subject and Problem

This project focuses on the program Caddie and the underlying theory of Combinatory Automatic Differentiation.

The project will explain Caddie and the use of point-free notation for automatic differentiation, and test if the program and theory can be used as a tool in a Python context. The method is to extend the testing and tooling of Caddie and to implement examples of usage.

1.2 Contributions

The contributions of this project are the Python context tool Caddiepy, definitions of a Python subset language, explanatory examples of the background theory, automated tests, and a context implementation of the tool to inspect its potential.

The Python context tool *Caddiepy* can differentiate Python programs to linear map derivative code using a forward-mode and a reverse-mode automatic differentiation. Consider the Python function:

```
def f(x): return x[0] * sin(x[1])
```

By using Caddiepy, the linear map derivative of the function is computed as:

```
def f_diff(x1,x2,dx1,dx2): v1 = sin(x2); return ((dx1*v1) +
↪  (x1*(cos(x2)*dx2)))
```

In the background section, contributions of explanatory examples of the theory are provided, such as derivative equations, linear map equations, the partial derivative steps in Automatic Differentiation, translations of variable assignments to point-free notations, and the application of monad types for differentiation rules used in Combinatory Automatic Differentiation.

Furthermore, the project contributes with implemented tests of the Caddiepy tool and shows how Caddiepy can be applied to fit a third-order polynomial to the sine function by using a Gradient Descent optimization, which resembles a simple neural network implementation.

1.3 Structure

In section 2, the theoretical basis of the project is provided. The section is divided into four subsections, where the notions of derivatives, Automatic Differentiation, Combinatory Adjoints and Differentiation, and the tool for Combinatory Automatic Differentiation are explained.

Section 3 focuses on the specifications for using Combinatory Automatic Differentiation in a Python context, and discusses the requirements of the tool for reading and writing Python code.

Section 4 defines a subset of the Python language to be used for the Python context tool, and considers the implementation requirements for obtaining differentiated Python code.

Section 5 explains the implementation of Caddiepy, and shows how the parsing and printing of Python code is implemented to comply with the Python subset language definitions.

Section 6 explains how to install Caddiepy and how to use the tool.

Section 7 assesses the testing of the tool and provides an example of how Caddiepy can be used to fit a third-order polynomial to the sine function with the use of Gradient Descent optimization.

In section 8, the correctness of Caddiepy is discussed together with its current limitations. It is examined how the tool can be improved by extending its functionality, and how Caddiepy compares to JAX, which is another available Python library for automatic differentiation.

The final section 9 concludes the project by addressing the subject of the initial hypothesis and provides suggestions for future development of the Caddiepy tool.

2 Background

2.1 Derivatives

The derivative of a function $f(x)$ describes the rate of change of the function at a given point x. The derivative expresses the sensitivity to change of a function's output with respect to the input. More accurately, the derivative of f gives a *linear approximation* to f around x [1].

The process of finding a derivative is called differentiation [2]. The most common definition of a derivative is defined by the Leibniz notation, named after Gottfried

Wilhelm Leibniz, which is also called the prime notation and is written with the use of the prime mark [2], e.g. $f'(x)$.

The purpose of describing the mathematical definition of derivatives is to work towards an understanding of Automatic Differentiation which is key to understanding the theoretical foundation for Combinatory Automatic Differentiation and its purpose as a programming tool.

2.1.1 One-Dimensional Derivative

The Leibnitz derivative of a simple one-dimensional function $f : \mathbb{R} \to \mathbb{R}$ at a point $x \in \mathbb{R}$ is given by the number $f'(x) \in \mathbb{R}$, that is the slope of the tangent line to f at x. Numerically this slope can be found as an approximated limit by:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{2.1}$$

If the limit exists, it said that f is differential at x. This formula can be rewritten as:

$$f(x+h) \approx f(x) + f'(x)h \tag{2.2}$$

to approximate f [1]. When h is smaller, the better the approximation is to f. It is worth noting that h can also be written as dx, and $f'(x)$ as dy, so that:

$$f'(x) = \frac{dy}{dx}$$

Formally the language describing an infinitesimal change of h is also called a *perturbation* to h, which results in changes to f at x.

By using the Leibnitz derivative equation, the derivative of $f(x) = x^2$ is found at x to be:

$$\begin{aligned} f'(x) &= \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \\ &= \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h} \\ &= \lim_{h \to 0} \frac{x^2 + h^2 + 2xh - x^2}{h} \\ &= \lim_{h \to 0} 2x + h \\ &= 2x \end{aligned}$$

when h goes to zero.

2.1.2 Multivariate Derivative

The Leibnitz derivative is a special case derivative, that works on scalars in one dimension, and it returns a number $\mathbb{R}$. However, it is useful to be able to obtain derivatives of multiple variables. Instead of understanding a derivative as a single number, the derivative can be interpreted as a linear transformation. This is called a *linear map*, which is a function that maps a vector from n to m dimensions. The linear map is written as $f : \mathbb{R}^n \multimap \mathbb{R}^m$. When the derivative is generalized to a linear map, we can think of it as a function that takes argument h, written as $f'(x)(h)$ [1].

To define the derivative as a linear map, the approximation in (2.1) can be extended to:

$$\lim_{||h||\to 0} \frac{||f(x+h) - f(x) + f'(x)h||}{||h||} = 0 \tag{2.3}$$

which defines the derivative of f at x to be a linear map $f'(x) : \mathbb{R}^n \multimap \mathbb{R}^m$ [1], where $||\cdot||$ is the Euclidean norm $||x|| = \sqrt{x_1^2 + ... + x_n^2}$. If the linear map $f'(x)$ exists, it means that f is differentiable at x and f is approximated by (2.2).

When derivatives of linear maps in this form are represented by a $n \times m$ matrix, it is called the *Jacobian* derivative or *Jacobian matrix*. The entries in the Jacobian matrix are the partial derivatives of f at x and they are represented in the form:

$$f'(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{2.4}$$

As an example, consider the function $f : \mathbb{R}^3 \to \mathbb{R}^2$, defined as:

$$f(x) = (x_1 x_2 x_3, x_1 + x_2^2 + x_3^3)$$

The derivative $f'(x)$ with point $x = (x_1, x_2, x_3)$ is:

$$f'(x_1, x_2, x_3) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{pmatrix} = \begin{pmatrix} x_2 x_3 & x_1 x_3 & x_1 x_2 \\ 1 & 2x_2 & 3x_3^2 \end{pmatrix} \tag{2.5}$$

Now, if $f'(x)$ is moved with a change of h, as defined by (2.2), we have:

$$f'(x)(h) = \begin{pmatrix} x_2 x_3 & x_1 x_3 & x_1 x_2 \\ 1 & 2x_2 & 3x_3^2 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} x_2 x_3 h_1 + x_1 x_3 h_2 + x_1 x_2 h_3 \\ h_1 + 2x_2 h_2 + 3x_3^2 h_3 \end{pmatrix}$$

which considers the derivative of f at x to be a linear map $f'(x) : \mathbb{R}^3 \multimap \mathbb{R}^2$.

The derivative of f in equation (2.5) is more accurately said to be the gradient of f. The gradient is used for functions with multiple variables and it is expressed with the symbol nabla as:

$$\nabla f(x_1, x_2, x_3)$$

The Jacobian derivative is restricted in its form $f'(x) : \mathbb{R}^n \to \mathbb{R}^m$, which means that it only works on scalars and vectors over real numbers. Sometimes it is necessary to differentiate functions over inputs that are in high-dimensional vector spaces [3], like in the case of neural networks, which can have millions of parameters in the form of weight matrices and bias vectors, that need to be differentiated. For that, the Fréchet derivative is used.

2.1.3 Fréchet Derivative

For a function $f : V \to W$, that transforms vector space V to vector space W on an input vector $x \in V$, the Fréchet derivative is approximated by:

$$\lim_{||h||_V \to 0} \frac{||f(x+h) - f(x) + f'(x)h||_W}{||h||_V} = 0 \tag{2.6}$$

where $|| \cdot ||_V$ is the norm over the vector space V [4]. If the derivative $f'(x)$ exists, it means that $f'(x)$ is a linear map $f'(x) : V \multimap W$, and that f is differentiable at x. The formula can be rewritten such that f is approximated by:

$$f(x+h) \approx f(x) + f'(x)(h)$$

Like the Jacobian derivative, the Fréchet derivative of a function $f : V \to W$ is a linear map function that is parameterized by h, which can be written as $f'(x)(h)$. The Fréchet derivative linear map can therefore be expressed as a partial function:

$$f' : V \to (V \multimap W) \tag{2.7}$$

which takes a vector $h \in V$ and transforms it into a linear map derivative [3].

This functional notation is important for the understanding of Caddie, since type declarations of the functions in the program, will use a similar notation, which will be described in 2.4.4.

2.2 Automatic Differentiation

2.2.1 Symbolic Differentiation

The method of differentiating functions using the mathematical derivatives described above is called *symbolic differentiation*. With this method, functions are derived, which can be applied to numerical points. In practice, implementing symbolic differentiation is inefficient and results in expression swelling, which will affect the computational power and scaling [5].

For example, if a function is first defined as $f(x) = u(x)v(x)$, and then the derivative is computed as $f'(x) = u'(x)v(x) + u(x)v'(x)$ by the product rule [6]—for the sake of argument, let's say the $u(x)$ and $v(x)$ are very complicated functions, like neural networks—then the implementation of the symbolic differentiation, will evaluate the expression $u(x)$ and $v(x)$ both in f and f', which results in redundant computation [5].

2.2.2 Numerical Differentiation

Instead of first deriving a function f', another option is to approximate the derivative by using the equations from section 2.1 and apply them directly to a point, since the function f is known. This method is called *numerical differentiation* and uses finite or central differences [5]. The derivative is approximated with a small h.

For example, given the function $f(x) = x^3$, the derivative at $x = 2.33$ is $f'(2.33) = 3 \cdot 2.33^2 = 16.2867$[1]. This can be approximated with equation (2.1) to:

$$f'(2.33) = \frac{f(2.33+h) - f(2.33)}{h} \approx 16.28700000$$

[1] Computed in Maple

for $h = 10^{-5}$. This approximation is close to the symbolic result of 16.2867.

However, because of round-off errors in computers when h gets significantly small (around 10^{-11}), there's a limit to this method, and depending on the required precision, this method might not be useful. Also, the computational cost of using numerical differentiation is quadratic ($\mathcal{O}(x^2)$) [5] and therefore doesn't scale well for large computations. However, the method can be useful in practice, since results of implemented derivatives can be verified by hand using numerical differentiation.

2.2.3 Forward-Mode Automatic Differentiation

The most efficient method to compute derivatives is by using *Automatic Differentiation*. The goal of automatic differentiation is to have a computer automatically generate code that implements the derivative calculations as arithmetic steps. In neural networks, that would be the functional forward pass through the network of units and layers. The key idea is to take the code that evaluates a function, augment the code with additional variables, and accumulate the values of the variables by use of the chain rule [7] and partial differentiation to arrive at the final derivative [5].

Automatic differentiation can handle closed-form mathematical expressions, and flow control functions like loops, recursion, and procedure calls [5]. It is therefore a very powerful method. This is also why, automatic differentiation is so fundamental to neural networks, since gradients can be automatically computed efficiently for various network architectures.

Forward-mode automatic differentiation starts with evaluating the innermost functions and then works its way out [1]. It first makes a trace for evaluating the variables and then it propagates through the trace and evaluates the partial derivatives.

For example, consider the function $f : \mathbb{R}^2 \to \mathbb{R}$:

$$f(x_1, x_2) = x_1 x_2 + x_2^2 - \sin(x_1 x_2)$$

The forward-mode evaluation trace is defined as:

$$\begin{aligned}
t_1 &= x_1 \\
t_2 &= x_2 \\
t_3 &= t_1 t_2 \\
t_4 &= t_2^2 \\
t_5 &= \sin(t_3) \\
t_6 &= t_4 - t_5 \\
t_7 &= t_3 + t_6
\end{aligned}$$

To get the derivative of f, we wish to propagate through the trace starting from t_1 and then take the partial derivative in each step and store the results as intermediate tangent variables $\dot{t}_i$. By the chain rule, the intermediate results, with respect to x_1, is accumulated by the expression:

$$\dot{t}_i = \frac{\partial t_i}{\partial x_1} = \sum_{j \in \{pre(i)\}} \frac{\partial t_i}{\partial t_j} \frac{\partial t_j}{\partial x_1} = \sum_{j \in \{pre(i)\}} \dot{t}_j \frac{\partial t_i}{\partial t_j}$$

where $pre(i)$ is the set of predecessors of the variable i.

The process is carried out by first taking the partial derivatives with respect to t_1 as follows:

$$\begin{aligned}
\dot{t}_1 &= \frac{\partial t_1}{\partial t_1} = 1 \\
\dot{t}_2 &= \frac{\partial t_2}{\partial t_1} = 0 \\
\dot{t}_3 &= \frac{\partial t_3}{\partial t_1} = t_1\dot{t}_2 + \dot{t}_1 t_2 \\
\dot{t}_4 &= \frac{\partial t_4}{\partial t_1} = \frac{\partial t_4}{\partial t_2}\frac{\partial t_2}{\partial t_1} = \dot{t}_2 \cdot 2t_2 \\
\hat{t}_5 &= \frac{\partial t_5}{\partial t_1} = \frac{\partial t_5}{\partial t_3}\frac{\partial t_3}{\partial t_1} = \dot{t}_3 \cdot \cos(t_3) \\
\dot{t}_6 &= \frac{\partial t_6}{\partial t_1} = \dot{t}_4 - \dot{t}_5 \\
\dot{t}_7 &= \frac{\partial t_7}{\partial t_1} = \dot{t}_3 + \dot{t}_6
\end{aligned}$$

When accumulating the results in each intermediate tangent variable, the final partial derivative results in:

$$\frac{\partial t_7}{\partial t_1} = t_2 - t_2\cos(t_3)$$

This is called a *single forward pass*. To get the partial derivative for the second parameter x_2, we'll do the same by propagating through the steps and computing the tangent variables with respect to t_2, which will result in:

$$\frac{\partial t_7}{\partial t_2} = t_1 + 2t_2 - t_1\cos(t_3)$$

By using this method the gradient ∇f is:

$$\nabla f(x_1, x_2) = \begin{pmatrix} x_2 - x_2\cos(x_1x_2) \\ x_1 + 2x_2 - x_1\cos(x_1x_2) \end{pmatrix}$$

Forward-mode automatic differentiation is efficient for functions that have few inputs and many outputs [5]. The partial derivatives of multiple functions can be evaluated as a Jacobian matrix, where the index of the columns represents m various functions, and the rows represent the index of the partial derivatives with respect to n variables, as shown in (2.4). In the case where m is much greater than n, the forward-mode method can evaluate the full $m \times n$ Jacobian matrix using n forward passes [5]. In situations where there are a lot of inputs and few outputs, *Reverse-mode* automatic differentiation is preferred.

2.2.4 Reverse-Mode Automatic Differentiation

Reverse-mode automatic differentiation, which is also called *backpropagation*, is similar to the forward-mode method where intermediate variables are augmented for the partial derivatives. In reverse-mode, the tangent variables are called *adjoint variables* [5], and these are computed in reverse order, which means that computation is done

starting from the outermost function calls and worked inwards. This is the reverse process of the forward method. The forward method, in simpler terms, starts from the input and works towards the output. The reverse method starts from the output and works towards the input. By the chain rule, the intermediate adjoint variables is accumulated by:

$$\bar{t}_i = \frac{\partial f}{\partial t_i} = \sum_{j \in \{suc(i)\}} \frac{\partial t_j}{\partial t_i} \frac{\partial f}{\partial t_j} = \sum_{j \in \{suc(i)\}} \frac{\partial t_i}{\partial t_j} \dot{t}_j$$

where $suc(i)$ is the successor of the variable i. As an example, backpropagating the function f and using the trace from before, we'll get the following adjoint variables, written as $\bar{t}_i$:

$$\begin{aligned}
\bar{t}_7 &= \frac{\partial t_7}{\partial t_7} = 1 \\
\bar{t}_6 &= \frac{\partial t_7}{\partial t_6} = 1 \\
\bar{t}_5 &= \frac{\partial t_7}{\partial t_5} = \frac{\partial t_6}{\partial t_5}\frac{\partial t_7}{\partial t_6} = -1 \cdot \bar{t}_6 = -\bar{t}_6 \\
\bar{t}_4 &= \frac{\partial t_7}{\partial t_4} = \frac{\partial t_6}{\partial t_4}\frac{\partial t_7}{\partial t_6} = 1 \cdot \bar{t}_6 = \bar{t}_6 \\
\bar{t}_3 &= \frac{\partial t_7}{\partial t_3} = 1 \\
\bar{t}_2 &= \frac{\partial t_7}{\partial t_2} = \frac{\partial t_3}{\partial t_2}\frac{\partial t_7}{\partial t_3} + \frac{\partial t_4}{\partial t_2}\frac{\partial t_6}{\partial t_4}\frac{\partial t_7}{\partial t_6} + \frac{\partial t_3}{\partial t_2}\frac{\partial t_5}{\partial t_3}\frac{\partial t_6}{\partial t_5}\frac{\partial t_7}{\partial t_6} \\
&\quad = t_1 \cdot \bar{t}_3 + 2t_2 \cdot 1 \cdot \bar{t}_6 - t_1 \cdot \cos(t_3) \cdot -1 \cdot \bar{t}_6 \\
\bar{t}_1 &= \frac{\partial t_7}{\partial t_1} = \frac{\partial t_3}{\partial t_1}\frac{\partial t_7}{\partial t_3} + \frac{\partial t_3}{\partial t_1}\frac{\partial t_5}{\partial t_3}\frac{\partial t_6}{\partial t_5}\frac{\partial t_7}{\partial t_6} \\
&\quad = t_2 \cdot \bar{t}_3 + t_2 \cdot \cos(t_3) \cdot -1 \cdot \bar{t}_6
\end{aligned}$$

When substituting the adjoint variables $\bar{t}_1$ and $\bar{t}_2$ we get the gradient of f to be:

$$\nabla f(x_1, x_2) = \begin{pmatrix} \frac{\partial t_7}{\partial t_1} \\ \frac{\partial t_7}{\partial t_2} \end{pmatrix} = \begin{pmatrix} t_2 - t_2 \cdot \cos(t_1 t_2) \\ t_1 + 2t_2 - t_1 \cos(t_1 t_2) \end{pmatrix}$$

which is the same gradient as in the forward method! If we have more than one output, we'll have to sweep through the reverse pass for all the other outputs as well.

Reverse-mode is a memory-intensive process because the intermediate variables of the trace have to be stored to compute the adjoint variables. In forward-mode, the trace can be computed together with the tangent variables during the pass [5], which reduces the memory overhead in comparison to reverse-mode.

The forward- and reverse-mode processes use a Jacobian matrix, which respectively requires n or m sweeps for an $m \times n$ matrix [1]. Computing the full Jacobian with a minimum number of operations is an NP-complete problem, which is known as the *optimal Jacobian accumulation problem* [8] [1].

This encourages motivation for finding new theoretical methods to efficiently compute derivatives at a large scale. *Combinatory Adjoints and Differentiation* is an attempt at that. The theory seeks to optimize differentiation without the use of Jacobian matrices.

2.3 Combinatory Adjoints and Differentiation

Combinatory Adjoints and Differentiation (CAD) [3] proposes a compositional approach to automatic differentiation. The theory uses the Fréchet derivative, so that derivatives expand to linear functions on abstract vector spaces, instead of being limited to scalars, vectors, matrices, or tensors represented as arrays.

The linear function derivatives are represented in combinatory form, which is a point-free notation. The combinatory differential calculus proposed by CAD is therefore symbolic differentiation. The theory defines rules for differentiating functions into point-free linear maps, which can be applied to numerical points for computing numerical derivatives.

The key of the theory is to avoid Jacobian matrix calculation, which is computationally inefficient when matrices become extremely large (think of tensors in neural networks with millions of entries). For example, instead of keeping a trace and storing intermediate variables when performing reverse-mode automatic differentiation, CAD derives adjoints in symbolic point-free notational form. This form is a function, and can therefore be run using data-parallel computation to optimize the efficiency when it is implemented.

2.3.1 Forward-Mode Differentiation

The forward-mode differentiation in CAD is represented as a point-free term with the type:

$$\text{Term}(V \to W) \to V \to \text{Term}(W \times (V \multimap W)) \tag{2.8}$$

where V and W are types of abstract representations of vector spaces, and $\text{Term}(V \to W)$ is a language for representing functions in combinatory form [3].

The forward-mode functional type takes a function f and a value v and returns the derivative in combinatory form, represented as a point-free notation term $\text{Term}(W \times (V \multimap W))$ containing the value $w = f(v)$, that is the input function f applied to the input value v, and the linear map Fréchet derivative f', which is a function. As explained in section 2.1.3, the Fréchet derivative is written as $f'(x)(h)$, which takes the perturbation h as an argument to the point in which the derivative should be evaluated in.

To get the derivative at a specified point, the CAD derivative is then interpreted as a term of type:

$$\text{Term}(V \multimap W) \to V \to W \tag{2.9}$$

which is the derivative linear map function $f'(v)(dv)$. As input, it takes a tangent differentiable value dv (which corresponds to h in 2.1.3) and a value v, which is the point the derivative is evaluated in, and returns the derivative value $w = f'(v)(dv)$.

2.3.2 Reverse-Mode Adjoint

To obtain a derivative using reverse-mode, CAD applies the theoretical concept of *adjoints*, which comes from the branch of mathematics concerned with functional analysis and vector spaces. The adjoint is the transpose of a linear map [9], and it is formulated by an induced map between dual vector spaces of linear functionals [3] [9], which, in this context, are linear maps[2] from vectors to scalars. When the

[2] also called *linear form*

derivative linear map is $(V \multimap W)$, the notion of the adjoint linear map is $(W \multimap V)$. The adjoint is the derivative transpose and is considered as the reverse-mode process.

The reverse-mode method to derive adjoints in CAD is done by applying adjoint calculation rules to the derivative term obtained from the forward-mode process. The adjoint derivative of a function $f : V \to W$ at $v \in V$ is $(f'(v))^*$. This means that rules from Theorem 6.5 [3] are applied to the differential term Term$(V \multimap W)$ in (2.9), to get the adjoint combinatory term.

When the adjoint combinatory term is derived, the term can then be applied to values, just like the forward-mode derivative. For reverse-mode differentiation the flow is 'backward', therefore, the adjoint derivative calculates a function's input differential given a function's output differential. The adjoint linear map $f'(v)(h)$ should therefore take output adjoint values as the input h.

That is, for a function $f : V \to W$, with input $v \in V$ and output $w \in W$:

$$w = f(v)$$

the tangent value to the forward-mode derivative is dv, and the adjoint value for the reverse-mode derivative is dw, so that:

$$\text{Forward-mode: } f'(v)(dv)$$
$$\text{Reverse-mode: } f'(v)(dw)$$

With the notion of the forward- and reverse-mode methods using CAD explained, the next section gives an overview of the tool for Combinatory Automatic Differentiation, which implements this theory.

2.4 Caddie

Caddie is an abbreviation of *Combinatory Automatic Differentiation* [10], and it is the name of the standalone tool that implements the theory of Combinatory Adjoints and Differentiation. The tool implementation is written in the general-purpose functional programming language Standard ML, and it is accessible on the GitHub repository:

github.com/diku-dk/caddie

With Caddie it is possible to differentiate programs by using a forward- or reverse-mode automatic differentiation. The tool aims to create efficient code of differentiated functions, to be used as a source in a high-level language for computing numerical derivatives.

The tool works by first turning an input program into combinatory form. It can then evaluate the zero-order representation of the program, which is the undifferentiated program, and differentiate the form into a linear map representation. The linear map combinatory form can be evaluated according to forward- or reverse-mode specification, which results in a derivative function. The derivative function can then be returned as high-level language code, intended to be run in a domain-specific context. This is called the unlinearized differentiated program.

Currently, the tool can differentiate programs written in a specified let-binding context language that resembles Standard ML. These programs are stored as `.cad` files. The tool is run from a terminal and it accepts various options as command-line arguments.

For example, Caddie can differentiate the program saved in the file `ln.cad`:

```
fun f x = ln(sin(x))
```

With the following command-line arguments:

```
./cad --Pdiff ln.cad
```

the tool gives the output:

```
Differentiated program (linear map expression):
 f x =
    ln(sin(x))
 f' x =
    (pow(-1)(sin(x)) *) :o: (cos(x) *)
 f^ x =
    (cos(x) *) :o: (pow(-1)(sin(x)) *)
```

The output first shows the zero-order representation of the program `f`, then the linear map derivative `f'`, and finally the adjoint linear map `f^`.

2.4.1 Point-Free Notation

The combinatory form used in Caddie is a point-free notational form. Point-free notation comes from *Tacit programming*, which is also called *point-free programming* [11]. It is a programming paradigm where the arguments in function definitions are not directly exposed. The arguments are implicit so that the function definition is expressed in a simplified condensed form. For example, the function:

```
fun f x = ln(x)
```

takes an argument `x` and returns the natural logarithm of `x`. For this function to be in point-free notation it would be written as:

```
fun f = ln
```

Here, the function arguments are explicitly omitted. The reason for expressing functions in point-free notation is to allow multiple functions to be composed into sequential combinations that manipulate arguments.

In mathematics, an example of function composition is:

$$(f \circ g)(x) = f(g(x)) = h(x)$$

In programming, this function composition can be written as:

```
fun h x = f(g(x))
```

The mathematical compositional symbol ∘ allows the functional calls to be evaluated sequentially from the innermost function (right) to the outermost function (left). This is the same for a point-free functional composition in a program. In CAD, the function composition in point-free notation would be written as:

```
let h = f ∘ g
```

Caddie translates expressions to point-free notation before computing the linear map derivative. The translation of expressions to point-free notation is defined by a grammar for point-free combinators and a set of rules for translating variable assignments. This will be explained in detail in section 2.4.3.

2.4.2 Monad Types

Caddie uses monad types to avoid expression swelling of the combinatory forms when deriving linear maps. Expression swelling is when the same variables reoccur in the computational expression several times and cause redundant computation. This is prone to happen with symbolic differentiation, as explained in section 2.2.1.

Monads are special types used in functional programming. A monad is a type constructor, denoted by the type `M`, which is defined by the two operations:

$$\texttt{return:} \quad \alpha \to \alpha\ \texttt{M} \tag{2.10}$$

$$\texttt{bind:} \quad \alpha\ \texttt{M} \to (\alpha \to \beta\ \texttt{M}) \to \beta\ \texttt{M} \tag{2.11}$$

The first operation (2.10), which is also called *unit* [12], returns a monad of type α `M`. The operation takes an input of type α and wraps it into a monadic value.

The second operation (2.11), is a monadic bind. The operation takes a monadic value α `M` and a function f of type $\alpha \to \beta$ `M`. The function f takes a value of type α and changes α to a β type, which is wrapped into a monad. The monadic bind operation therefore takes an input monad α `M`, unwraps the value α from the monad, applies f to it, so that α is bound to a β monad, and returns the monad β `M`.

Monads are used to structure computation as a sequence of steps, where additional information can be wrapped into the context of the monad together with the value. The return operation lifts the values (and information) into a monad context, and the bind operation enables chaining monadic computations [12].

In Caddie, the monads are used to keep track of intermediate variable bindings in the linear maps. The intermediate variable bindings are let-bindings, and therefore the monads in Caddie are called let-binding context monads. In the program implementation, the monad operations are specified in the signature file `val.sig`, as:

```
22  type 'a M
23  val ret         : 'a -> 'a M
24  val >>=         : 'a M * ('a -> 'b M) -> 'b M
```

with the monad bind represented by the value name >>=. The value >>= can then be defined as an infix operator to customize the monadic bind in different contexts of the program implementation.

To differentiate function compositions $f \circ g$, the infix operator >>= is for example used to define the monad context of the CAD rule [3] [3]:

$$\begin{aligned}(g \circ f)^{[1]}(x) = \textbf{let } (fx, f'x) = f^{[1]}(x) \textbf{ in} \\ \textbf{let } (gfx, g'fx) = g^{[1]}(fx) \textbf{ in} \\ (gfx, g'fx \bullet f'x)\end{aligned}$$

by the following implementation in the function `diffM` in `diff.sml`:

[3] the superscript $^{[1]}$ denotes the pair of a function's value and its derivative

```
73                  F.Comp(g,f) =>                  (* g o f *)
74                    D f x >>= (fn (fx,f'x) =>
75                    D g fx >>= (fn (gfx,g'fx) =>
76                    ret (gfx,L.comp(g'fx,f'x))))
```

The `F.Comp` datatype has two functions `g` and `f`. First, the differentiate operator `D` is applied to `f x`, which returns a monadic value. Then the monadic value is bound to a new monadic context using the lambda function `fn (fx, f'x) =>`. Within the scope of the first lambda function, a new monadic sequence for `g fx` is defined with a second lambda function. The second lambda function `fn (gfx, g'fx) =>` returns the monadic bind context by using the unit function `ret`. The final context monad is `(gfx,L.comp(g'fx,f'x))`, where `L.comp` is the linear map context monad binding of the derivative, handled in the `lin.sml` file.

By using monadic binds, the sequence of which the intermediate variable bindings occur, is structurally reliable, since the monadic context ensures that the intermediate variables are correctly bound within the functional scope of the linear map derivatives.

For example, consider the same function f from 2.2.1 defined by:

$$f(x) = u(x)v(x)$$

then by the product rule, it differentiates to:

$$f'(x) = u'(x)v(x) + u(x)v'(x)$$

Let now:

$$\begin{aligned} u(x) &= \sin(x) \\ v(x) &= \cos(x) \cdot x \end{aligned}$$

symbolically $f(x)$ differentiates to:

$$\begin{aligned} f'(x) &= \cos(x)\cos(x) \cdot x + \sin(x)(-\sin(x)x + \cos(x)) \\ &= \cos(x)\cos(x) \cdot x - \sin(x)\sin(x)x + \cos(x)\sin(x) \end{aligned} \tag{2.12}$$

Here, $\cos(x)$ and $\sin(x)$ appear several times in $f'(x)$, which is computationally redundant when implemented in a program. When this function is differentiated using Caddie, the input function defined as:

```
fun f x = sin(x) * (cos(x) * x)
```

leads to the linear map output:

```
f' x dx =
   let v1 = cos(x)
   let v2 = sin(x)
   let v3 = (v1 * x)
   in (((cos(x) * dx) * v3) + (v2 * (((~(sin(x)) * dx) * x) + (v1 * dx))))
```

The output shows, that the linear map derivative function implements three intermediate variable `let`-bindings, `v1`, `v2`, and `v3`. These are the monad context variable bindings. By binding the cosine and sine expressions to variables, expression

swelling in the last line of the linear map is reduced, by using the intermediate variables, instead of evaluating the cosine and sine expression multiple times, as shown in equation (2.12).

In Caddie, the let-binding context monads of the forward-mode and reverse-mode linear maps of a function are defined as:

$$\texttt{D f}: \quad \texttt{V} \Rightarrow (\texttt{W} \times (\texttt{V} \mapsto \texttt{W}))\ \texttt{M} \qquad \text{Forward-mode} \tag{2.13}$$

$$\texttt{Adj(D f)}: \quad \texttt{V} \Rightarrow (\texttt{W} \mapsto \texttt{V})\ \texttt{M} \qquad \text{Reverse-mode} \tag{2.14}$$

Looking back at the previous example of `F.comp`, we now see that the return context monad tuple `(gfx,L.comp(g'fx,f'x))` is of the same type as the return type of `D` in (2.13); the function value is in the first component and the linear map derivative is in the second component, which is wrapped in a monad using `ret`.

2.4.3 Semantics

The semantics of Caddie is specified for function expressions, point-free notation, and linear maps.

Functions in Caddie are defined by unary and binary operators and expressions. The unary operators are symbolized as rho (ρ) and the binary operators are symbolized as diamond ($\diamond$). These operators follow the grammar:

$$\rho := \ln \mid \sin \mid \cos \mid \exp \mid \text{pow}\, r \mid \sim \tag{2.15}$$

$$\diamond := + \mid * \mid - \tag{2.16}$$

where $:=$ means assignment to one of the statements, that are separated by a bar ($\mid$), like match-cases. Here, r denotes the range of all reals $r \in \mathbb{R}$, and $\sim$ is the tilde symbol for negation. Expressions e are defined by the grammar:

$$e := r \mid x \mid \rho\, e \mid e \diamond e \mid (e,e) \mid \pi\, i\, e \mid (e) \mid \text{let } x = e \text{ in } e \tag{2.17}$$

Expressions can be a real, a variable x, a unary or binary operation on expressions, a tuple, a projection (denoted by π where i is an integer index larger than zero), a single expression in parenthesis, or a variable let-binding.

Expressions can be translated into point-free notation denoted by the letter p. The point-free notation grammar is defined by:

$$p := p \circ p \mid \pi\, i \mid K\, e \mid p \times p \mid \triangle \mid Id \mid \rho \mid \diamond \mid (p) \tag{2.18}$$

where $\circ$ is function composition, K is a constant function, $\times$ is element-wise function application, $\triangle$ is duplication (dup), and Id is the identity. The exact definitions are defined in lambda calculus terms as:

$$\begin{aligned}
\triangle: &\quad \lambda x.(x,x) \\
\pi\,1: &\quad \lambda(x,y).x \\
\pi\,2: &\quad \lambda(x,y).y \\
K\,e: &\quad \lambda_.e \\
Id: &\quad \lambda x.x \\
\times: &\quad \lambda(f,g).\lambda(x,y).(f(x),g(y)) \\
\circ: &\quad \lambda(f,g).\lambda x.f(g(x))
\end{aligned}$$

This grammar defines the point-free combinators, which are used in the translation of input expressions to point-free notation. The translation to point-free notation is done with the use of variable assignment rules. The rules define how explicitly declared variables from an expression environment are mapped to point-free notation compositions of projections. That means, that explicit variables are removed from the expression and the projection function is used to project out the variables when the point-free notational form is applied to an environment. The projection function is composed into a point-free composition, just like any other function, as explained in 2.4.1.

The translation rules are defined by:

$$
\begin{aligned}
|x|\delta &= \delta(x) \\
|r|\delta &= K\, r \\
|\rho\ e|\delta &= \rho \circ |e|\delta \\
|e_1 \diamond e_2|\delta &= \diamond \circ (|e_1|\delta \times |e_2|\delta) \circ \triangle \\
|\text{let } x = e_1 \text{ in } e_2|\delta &= |e_2|(\delta \circ \pi\ 2, x : \pi\ 1) \circ (|e_1|\delta \times Id) \circ \triangle \\
|(e_1, e_2)|\delta &= (|e_1|\delta \times |e_2|\delta) \circ \triangle \\
|\pi\ i\ e|\delta &= \pi\ i \circ |e|\delta \\
|(e)|\delta &= |e|\delta
\end{aligned}
$$

where δ denotes the range over variable assignments $x_1 : p_1, ..., x_n : p_n$, and e is an expression environment as defined by (2.17). This means that in the range δ, variable x_1 is assigned point-free notation p_1, and variable x_n is assigned point-free notation p_n. The symbolic definition $|e|\delta$ means that the expression e is translated into point-free notation by having the variables in the expression assigned point-free notation.

To give a better understanding of how the point-free notation translation is applied, consider the expression:

```
let x = 5 in
let y = 8 in
  x + y
```

This is the expression environment e. To translate this to point-free notation, the translation rules are applied to the definition $|e|\delta$. Initially, δ is empty; there have been no assignments. The expression is a let-binding expression, so first the let-binding rule is applied, and then recursively the other rules are applied until the expression is translated to a point-free notation.

It is done by the following:

$$
\begin{aligned}
&|\texttt{let x = 5 in let y = 8 in x + y}|\delta && \delta\{\} \\
=&|\texttt{let y = 8 in x + y}|\delta \circ (|\texttt{5}|\delta \times Id) \circ \triangle && \delta\{x : \pi\ 1\} \\
=&|\texttt{x + y}|\delta \circ (|\texttt{8}|\delta \times Id) \circ \triangle \circ (K\ \texttt{5} \times Id) \circ \triangle && \delta\{x : \pi\ 1 \circ \pi\ 2, y : \pi\ 1\} \\
=&+ \circ (|\texttt{x}|\delta \times |\texttt{y}|\delta) \circ \triangle \circ (K\ \texttt{8} \times Id) \circ \triangle \circ (K\ \texttt{5} \times Id) \circ \triangle && \delta\{x : \pi\ 1 \circ \pi\ 2, y : \pi\ 1\} \\
=&+ \circ (\pi\ 1 \circ \pi\ 2 \times \pi\ 1) \circ \triangle \circ (K\ \texttt{8} \times Id) \circ \triangle \circ (K\ \texttt{5} \times Id) \circ \triangle && \delta\{x : \pi\ 1 \circ \pi\ 2, y : \pi\ 1\}
\end{aligned}
$$

The variable assignments δ are kept on the right, and on the left, the point-free notation translation is applied. First, the let-binding rule is used on the let-binding of x. Then, the let-binding rule is used again on the second binding y. The variables x and y are assigned numerical values, so these values are translated to the constant

function $K\ r$. Finally, the binary operation $+$ is translated, and x and y are translated to projections using the accumulated projections in δ.

The initial expression e is now in point-free notation. It is a function composition, so if an environment or argument is now passed to this point-free notational expression, it will return the result of $x + y$. For example, if 9 is passed to the expression, it gives:

$$\begin{aligned}
&(+ \circ (\pi\ 1 \circ \pi\ 2 \times \pi\ 1) \circ \triangle \circ (K\ 8 \times Id) \circ \triangle \circ (K\ 5 \times Id) \circ \triangle)(9) \\
=&(+ \circ (\pi\ 1 \circ \pi\ 2 \times \pi\ 1) \circ \triangle \circ (K\ 8 \times Id) \circ \triangle \circ (K\ 5 \times Id))(9,9) \\
=&(+ \circ (\pi\ 1 \circ \pi\ 2 \times \pi\ 1) \circ \triangle \circ (K\ 8 \times Id) \circ \triangle)(5,9) \\
=&(+ \circ (\pi\ 1 \circ \pi\ 2 \times \pi\ 1) \circ \triangle \circ (K\ 8 \times Id))((5,9),(5,9)) \\
=&(+ \circ (\pi\ 1 \circ \pi\ 2 \times \pi\ 1) \circ \triangle)(8,(5,9)) \\
=&(+ \circ (\pi\ 1 \circ \pi\ 2 \times \pi\ 1))((8,(5,9)),(8,(5,9)) \\
=&(+)(5,8) \\
=&13
\end{aligned}$$

which is the expected result.

Linear maps denoted by m are defined by the grammar:

$$m := \triangle \mid (+) \mid \sim \mid \pi\ i \mid 0 \mid Id \mid m \oplus m \mid m \bullet m \mid (e \diamond) \mid (\diamond e) \mid (m) \tag{2.19}$$

where $\diamond$ is a bilinear map function, $\oplus$ is the element-wise linear map application corresponding to the point-free $\times$ operator, and $\bullet$ is linear map function composition similar to the point-free $\circ$ operator.

2.4.4 Differentiation

Differentiation in Caddie is defined by the differentiation operator `D` of the following type:

$$\texttt{D} : (\texttt{V} \to \texttt{W}) \Rightarrow \texttt{V} \Rightarrow (\texttt{W} \boxtimes (\texttt{V} \mapsto \texttt{W})) \tag{2.20}$$

Here, the type declaration of `D` is declared in non-monadic form. The declaration uses three different arrows to denote the various expression levels used in Caddie.

The declaration distinguishes between the functions at a *expression* level, denoting the signatures of functions, and functions at a *meta* level, which describes the vector space transformation declarations. The result is a type signature that contains three types of transformations, which are symbolized by the arrows; the functional *expression* transformation, the derivative *linear map* transformation, and the *meta* vector space transformation, given by:

$$\begin{aligned}
\to &\quad \text{expression transformation} \\
\mapsto &\quad \text{linear map transformation}(\multimap) \\
\Rightarrow &\quad \text{vector space meta transformation}
\end{aligned}$$

The reason for this complicated type declaration comes from the Term definition in (2.8), which uses the Fréchet linear map (2.7).

The differentiate operator returns a tuple consisting of the result value `W` of the function $(\texttt{V} \to \texttt{W})$ and the linear map $(\texttt{V} \mapsto \texttt{W})$[4]. $\boxtimes$ denotes a meta-pair of the vector spaces.

[4]The Fréchet linear map (2.7) would in Caddie be written as $\texttt{V} \Rightarrow (\texttt{V} \mapsto \texttt{W})$

Differentiation using the `D` operator is defined on point-free notation according to the definitions from [10] shown in figure 1. Primitive function operators ρ are differentiated by `D` ρ.

The forward-mode automatic differentiation process in Caddie can be defined for a point-free expression at x as:

```
f' x = let (_,m) = D p x
       in m
```

where the linear map `m` is returned by the let-expression.

```
D : (V → W) ⇒ V ⇒ W ⊠ (V ↦ W)

D (g ○ f) x =
  Let (fx, f'x) = D f x
  Let (gfx,g'fx) = D g fx
  In (gfx, g'fx • f'x)

D (K y) x = (y, 0)

D (π i) x = (π i x, π i)

D (f × g) x =
  Let (fx, f'x) = D f (π 1 x)
  Let (gy, g'y) = D g (π 2 x)
  In ((fx,gy), f'x ⊕ g'y)

D (Δ) x = ((x,x), Δ)

D (Id) x = (x, Id)

D (ρ) x = Dρ x

D (◇) x = (◇ x, (+) • ((◇ π 2) ⊕ (π 1 ◇)))

D (+) x = ((+)x, (+))
```

Figure 1: Differentiation definitions, cited from the Caddie repository [10].

The reverse-mode differentiation process uses the adjoint of the derivative linear map, as explained in 2.3.2, by applying the definitions shown in figure 2. The adjoint operator `Adj` is defined by the type:

$$\texttt{Adj} : (\texttt{V} \mapsto \texttt{W}) \Rightarrow (\texttt{W} \mapsto \texttt{V}) \tag{2.21}$$

The adjoint operator takes a linear map derivative as input and transposes the linear map to an adjoint linear map. The reverse-mode differentiation process is then defined as a let-binding operation on a point-free notation with respect to an argument x, as:

```
f^ x = let (_,m) = D p x
       in Adj m
```

where the notation `f^` is the reverse-mode derivative.

2.4.5 Standard ML Implementation

The Caddie tool is written in Standard ML, which is an imperative functional programming language.

```
Adj : (V ↦ W) ⇒ (W ↦ V)

Adj (0) = 0

Adj (Id) = Id

Adj (Δ) = (+)

Adj ((+)) = Δ

Adj (~) = ~

Adj (π 1) = (Id ⊕ 0) • Δ

Adj (π 2) = (0 ⊕ Id) • Δ

Adj (m₁ • m₂) = Adj (m₂) • Adj (m₁)

Adj (m₁ ⊕ m₂) = Adj (m₁) ⊕ Adj (m₂)

Adj ((e*)) = (e*)     if * : ℝ × ℝ → ℝ          (multiplication)
Adj ((*e)) = (*e)     if * : ℝ × ℝ → ℝ          (multiplication)

Adj ((e·)) = (*e)     if · : ℝⁿ × ℝⁿ → ℝ           (dot product)
Adj ((·e)) = (*e)     if · : ℝⁿ × ℝⁿ → ℝ           (dot product)

Adj ((e×)) = (e×)     if × : ℝ × ℝⁿ → ℝⁿ        (scalar product)
Adj ((×e)) = (·e)     if × : ℝ × ℝⁿ → ℝⁿ        (scalar product)
```

Figure 2: Adjoint definitions, cited from the Caddie repository [10].

The Caddie implementation consists of a source folder with eleven different implementation files and signatures which are: `ad`, `ast`, `cad`, `diff`, `exp`, `fun`, `lin`, `prim`, `rel`, `term_val`, and `val`. These modules are the main part of the program. The implementation also uses other files that provide functionality for running the tool in the terminal, and it uses an external library package for the parsing implementation called `sml-parse` [13].

The main program call is implemented in `cad.sml` at the end of the file as:

```
292 fun main () =
293     let val parseRes = parseEval()
294         val compRes = compile parseRes
295         val transRes = translate compRes
296         val diffRes = differentiate transRes
297         val udiffRes = unlinearise diffRes
```

where the flow of the differentiation process is serialized in a let-binding. First, the input is parsed into an abstract syntax tree (AST), then it is compiled to an internal expression for the tool, then the expression is translated to point-free notation and differentiated, and finally translated to an unlinearized result.

Standard ML provides functionality for writing let-expressions. This is an essential feature that enables the monad bindings needed for Caddie.

The implementation of Caddie is expressed in compact code, following a functional programming form. It utilizes patterns and anonymous lambda functions to allow for higher-order functionality and programming structure.

Caddie can take `.cad` files as input programs to be differentiated. The `.cad` files are written in an unspecified let-binding context language that resembles a subset

of Standard ML. The `.cad` files allow for specialized operator symbols tailored for Caddie, like scalar multiply `*>`.

With the theoretical notion of derivatives and an overview of the Caddie tool, the following section will address requirements for contextualizing Combinatory Automatic Differentiation with the Python language.

3 Analysis

This section focuses on the modifications that need to be made to Caddie so that the tool will work in a Python context. It is mainly concerned with the programming language specifications of the input and output files to Caddie.

3.1 Python Context Caddie

Currently, Caddie works on `.cad` files, written in a functional style language using let-expressions, that resemble Standard ML. For Caddie to work in a Python context, the tool has to be able to read and derive programs written in the Python language. It is therefore a requirement that the input files and the unlinearized outputs of the modified Caddie tool can be run as Python programs in a Python interpreter.

The modified tool for using Combinatory Automatic Differentiation in a Python context will be called *Caddiepy*.

3.2 Requirements for Reading Python Input

The Python language is a dynamically-typed language, which evaluates variable types during the run of a program. Python is not designed as a functional programming language and it is therefore very different from the Standard ML language that Caddie is built on.

Let-expressions do not exist in Python. Therefore let-expressions have to be written in a different way in Python, which conveys the same grammar. The translation of a functional programming style to an imperative procedural style might cause complications, which should be anticipated as much as possible.

Projection `#` and scalar multiplication `*>` are features that do not exist in Python either. These operators have to be translated to a Python syntax that needs to work with Caddiepy.

The mathematical operators described in 2.4.3, like natural logarithm `ln`, negation $\sim$ and the power function `pow`, also have to be translated to a Python syntax, and changes have to be made to Caddie so that the Python input programs are parsed correctly and compiled to readable internal expressions.

The function declarations in the `.cad` language only allows a single parameter for the input; for example, `fun f x`. To have multiple argument values in the function, projection, symbolized as `#`, is used to project values out from `x`, as in the example `fun x = (#1 x) + (#2 x)`. This limitation has to apply to the Python input context as well.

Many mathematical operators are not native to Python. It, for example, requires external libraries to compute cosine and sine in the interpreter, and vector and matrix operations are not native to Python either. Therefore, the external Python library NumPy [14] will have to be used, to be able to execute the Python programs in a Python interpreter, when mathematical operations are in use.

3.3 Requirements for Deriving Python Output

Caddie can generate various outputs of the differentiation process. The option `--Pdiffu` generates the unlinearized differentiated code output of a program. This is the output that needs to be modified so that the derived output is written in Python.

Caddie outputs a differentiated program as a functional let-expression context language with a similar language definition as the input `.cad` files. For example, the output differentiated programs use `prj1` for projection of element one, instead of `#1` as `.cad` files.

The translation of the let-expression context language to a Python context output will need to be similar to the Caddiepy input requirements. The let-expressions have to be translated to a Python equivalent expression, and mathematical operators will have to follow the syntax of available Python operators.

Ideally, it should be possible to derive a second-order derivative of a program, by running Caddiepy on a first-order differentiated program output; so that a workflow using the tool can follow the sequence: input to a first-order unlinearized differentiated output to a second-order unlinearized differentiated output. To do this, the input and output programs will have to follow the same language definition.

Currently, it is not possible to derive second-order derivatives with Caddie because of the discrepancy of syntax between the input `.cad` files and the unlinearized output. For example, the Caddie unlinearized differentiated programs generate function declarations with multiple parameters; an input function `f xs` might be differentiated to `f' (x1,x2,x3) (dx1,dx2,dx3)`. This differentiated output, cannot be used as input for Caddie, since only a single argument is allowed for the function declaration in input files.

The Python context translation for Caddiepy will have the same issue because the unlinearized output is translated from the linearized differentiated program used internally in Caddie. It is necessary to consider a definition for the Python language used for Caddiepy.

4 Design

This section considers the Python language definition to be used for Caddiepy. The section provides a grammar definition for a subset of Python, which translates the let-expression context language in Caddie to a Python context.

Furthermore, the section describes how differentiated unlinearized Python output code should be generated, and how the generated code should relate to the Python subset grammar definitions.

4.1 Python Subset Language

As discussed in the analysis section 3.2, input files to Caddiepy have to be written in the Python language, but not all language definitions of Python will be able to work with Caddiepy. Therefore, a subset of the Python language that is tailored to Caddiepy will be defined. The subset language defines the input programs for Caddiepy. The requirement is that the Python subset language can run as scripts in a Python interpreter and that it can be differentiated by Caddiepy. The subset language is defined in the sections below.

4.1.1 Functions, Variables, and Parameters

Inputs to Caddiepy are functions since the objective is to differentiate functions. The input files are stored as Python scripts using the `.py` file naming. The `.py` input files should always start with function declarations that follow the form:

$$\texttt{def}\ \ var\texttt{(}param\texttt{):}\quad funexp \tag{4.1}$$

where *var* are variable names, *param* is a single parameter, and *funexp* are function expressions. The `typewriter` font denotes the written program syntax and *italicized* words are context-free grammar definitions.

Variable names *var*, can be any combination of letters, underscore, or digits, that follows the string literal definition of Python [15]. A variable name cannot be a single digit, or start with a digit, but a starting letter can precede a number like `f2`. The same is valid for a parameter. It is defined as:

$$var \quad := \quad \texttt{[a-zA-Z\_][a-zA-Z0-9\_]}* \tag{4.2}$$

$$param \quad := \quad var \tag{4.3}$$

where the symbol := denotes derivation to various definitions of the grammar and the string literal of *var* is defined as a regular expression.

Function expressions are defined by the two definitions:

$$funexp \quad := \quad varbind\texttt{;}\ funexp \tag{4.4}$$

$$\mid\ \texttt{return}\ exp \tag{4.5}$$

The first case is variable bindings *varbind* followed by a semicolon, followed recursively by a function expression. The second case is a return statement followed by an expression *exp*.

Function declarations always have to end with a return statement, and variable-bindings are separated by the semicolon. The full Python language is known for its indentation whitespace syntax to separate statements. For Caddiepy to work, and make the translation from a let-binding context language to a Python subset language, statements are strictly separated by semicolon. Indentation will be ignored by the Caddiepy parser, but it won't be ignored in a Python interpreter. Therefore, to be able to run a Python subset language file in both programs, indentation to separate statements is not allowed.

It is important to note that named function declarations of the Python subset language only take a single parameter as input. This means that if an input consists of multiple arguments, projections will have to be used in the function expression. Projections will be defined further down in the section.

Variable bindings *varbinds* are defined as:

$$varbind \quad := \quad var\ \texttt{=}\ exp \tag{4.6}$$

where variables are assigned the result of an expression. This corresponds to the let-binding context expression, written as `let x = e in e`, as in the definition (2.17). Python doesn't have let-binding definitions in the language. The let-binding context is syntactically translated to the combined use of a variable binding and the semicolon, as shown in *funexp*.

4.1.2 Expressions

Expressions are defined as:

$$
\begin{aligned}
exp &:= num && (4.7)\\
&\mid var && (4.8)\\
&\mid unop\ exp1 && (4.9)\\
&\mid binop\ exp1\ exp2 && (4.10)\\
&\mid (exp, exps) && (4.11)\\
&\mid var[i] && (4.12)\\
&\mid (exp) && (4.13)
\end{aligned}
$$

$$
\begin{aligned}
exps &:= exp,\ exps && (4.14)\\
&\mid exp && (4.15)
\end{aligned}
$$

where *num* are any numbers of reals in $\mathbb{R}$, and *unop* are unary operators taking an expression as argument, and *binop* are binary operators taking two expressions as arguments. A tuple can have two or more expression declarations.

Projections in Caddie are defined with π and an index i excluding zero. In the Python subset language, this is translated to an array-indexing using $var[i]$, where the index i is any natural number $\mathbb{N}_0$, including zero.

4.1.3 Binary and Unary Operators

In Caddie, the unary and binary operators are defined respectively using the symbols ρ and $\diamond$ as explained in 2.4.3. For the Caddiepy Python subset language, the operators are denoted by *unop* and *binop*.

Unary and binary operators are defined as:

$$
\begin{aligned}
unop &:= \texttt{log}(exp) && (4.16)\\
&\mid \texttt{sin}(exp) && (4.17)\\
&\mid \texttt{cos}(exp) && (4.18)\\
&\mid \texttt{exp}(exp) && (4.19)\\
&\mid \texttt{pow}(exp, num) && (4.20)\\
&\mid \texttt{-}\ exp && (4.21)
\end{aligned}
$$

$$
\begin{aligned}
binop &:= exp + exp && (4.22)\\
&\mid exp - exp && (4.23)\\
&\mid exp * exp && (4.24)
\end{aligned}
$$

where `log` is the natural logarithm, `pow` is the power function, `-` is negation, and `exp` is the exponential. For the unary and binary functions to work in a Python interpreter, the Python subset language is extended using the NumPy framework. Some functional operations are not native to Python, such as `log`, `sin`, and `cos`, and therefore need the use of an external library to run in the Python interpreter. Caddiepy does not depend on an external library, however.

4.2 Differentiated Unlinearized Output

The unlinearized Python output of Caddiepy should follow similar grammar definitions as the input, defined in 4.1. However, function expressions should be able to have multiple parameters instead of just one, so that the tangent and adjoint values to the linear maps, as described in 2.3.2, can be passed as arguments. This will allow a differentiated Python function to be declared as `f_diff(x,dx)`.

The function declaration grammar of the unlinearized output is therefore defined as:

$$\texttt{def}\ \mathit{var}(\mathit{params})\texttt{:}\quad \mathit{funexp} \tag{4.25}$$

where parameters are defined as:

$$\begin{aligned} \mathit{params} &:= \mathit{param}\texttt{,}\ \mathit{params} && (4.26)\\ &\mid \mathit{param} && (4.27) \end{aligned}$$

This means that the Python output code can have multiple variables in the function declarations, as `def f(x1,x2,dx1,dx2)`, but the input Python code is only allowed to have a single input variable as `def f(x)`. As discussed in 3.3, differentiated outputs from Caddiepy cannot be re-differentiated, since the function definitions of the input and output differ. Caddiepy is limited to only first-order differentiation.

Python code indentation is not allowed for the output either, and the semicolon is used instead. Unary and binary operators and expressions follow the same grammar as in 4.1.

5 Implementation

This section describes the implementation of Caddiepy. The section shows how the parser of the original program is modified to work with the Python subset language and how the pretty printer is modified to print Python code. The main modifications to the parser are implemented in `ast`, and the printing of the unlinearized Python output is implemented in `term_val`, `prim`, and `cad`.

5.1 Caddiepy

To keep track of the development of Caddiepy, the project is hosted on a GitHub repository:

github.com/kdsoup/caddiepy

It makes the project publicly available and the development process is kept organized by having version control on the project code. New implementations are configured via branches, which are merged into a main branch when implementations are ready to be included in the project.

The source code is taken from the original Caddie project [10]. The core implementations of Caddie have not been altered for Caddiepy, since it is necessary to keep the foundational combinatory automatic differentiation implementation intact.

5.2 Modifying the Parser

Parsing of the Python input program happens in the abstract syntax tree module `ast` when `parse_prg` is called. The input program is first tokenized with the help of the sml-parse `SimpleToken` module, and then parsed with `p_prg`.

The beginning of the input Python program starts with a function declaration, as defined in 4.1.1. `p_prg` is implemented by expecting `def` as the first identifier to be parsed, and then parsing the remaining function declaration and body. It is implemented as:

```
552 val rec p_prg : rprg p =
553     fn ts =>
554        (  (((((((p_kw "def" ->> p_var) >>- p_symb "(") >>> p_var) >>- p_symb
           ↪  ")") >>- p_symb ":") >>> p_e) oor (fn (((f,x),e),r) => [(f,x,e,r)]))
           ↪  ??* p_prg) (op @)
555        ) ts
```

The functions `p_kw`, `p_var`, and `p_symb` parses keywords, variables, and symbols, and custom operators are used from the sml-parse library to combine the parsed results[5]. The function `p_e` parses the remaining body of the program using several helper functions in recursive calls.

The operators for addition, subtraction, and multiplication remain implemented from the source code. The simultaneous recursive parse function `p_ae` is implemented to parse expressions according to the Python subset language definitions:

```
491 and p_ae : rexp p =
492     fn ts =>
493        (    ((p_kw "return") ->> p_e)
494          || ((p_var >>> ((p_symb "=" ->> p_e) >>> (p_symb ";" ->> p_e))) oor
           ↪  (fn ((v,(e1,e2)),r) => Let(v,e1,e2,r)))    (* Variable bindings *)
495          || ((p_e_prj >>> ((p_symb "[" ->> p_index) >>- p_symb "]")) oor (fn
           ↪  ((e,i),r) => Prj(i,e,r)))
496          || ((p_var >>> p_ae) oor (fn ((v,e),r) => App(v,e,r)))
497          || (((((p_kw "pow" ->> p_symb "(") ->> p_ae) >>- p_symb ",") >>>
           ↪  (p_real >>- p_symb ")")) oor (fn ((e,f),r) => Pow(f,e,r)))
498          || ((((p_kw "log" ->> p_symb "(") ->> p_e) >>- p_symb ")") oor (fn
           ↪  (e,r) => App("ln",e,r)))
499          || (p_var oor Var)
500          || (p_zero oor (fn ((),i) => Zero i))
501          || (p_int oor Int)
502          || (p_real oor Real)
503          || ((p_seq "(" ")" p_e) oor (fn ([e],_) => e | (es,r) => Tuple
           ↪  (es,r)))
504       ) ts
```

For projection, two additional helper functions `p_e_prj` and `p_index` are implemented to specifically parse projections of input arguments. The `p_index` function maps the zero-indexing of arrays to one-indexing of projections for the internal tool expressions and prevents indexes from being negative integers. The functions are implemented as:

[5] see `PARSE.sig` for explanation on the operators.

```
and p_e_prj : rexp p =
  fn ts =>
      ( (p_var oor Var)
      ) ts
```

```
val p_index : int p =
 fn ts =>
    case ts of
        (T.Num n,r)::ts' =>
        (case (Int.fromString n, List.exists (fn c => c = #"." orelse c = #"-"
        ↪  ) (String.explode n)) of
             (SOME n, false) => OK (n+1,r,ts')
           | _ => NO(locOfTs ts, fn () => "non-negative int"))
      | _ => NO(locOfTs ts, fn () => "non-negative int")
```

For handling negation, which uses the same symbol as subtraction, a new parsing grammar `p_e1` is added and is implemented as:

```
and p_e1 : rexp p =
    fn ts =>
       ( (((p_symb "-") ->> p_e1) oor (fn (e,r) => Sub (Zero r,e,r))) || p_ae
       ) ts
```

The parsing of negation is implemented as a grammar level between multiplication in `p_e0` and the atomic expressions in `p_ae`. The reason for this is to avoid ambiguous parsing of the negation symbol and subtraction symbol. Negation is evaluated as a subtraction operator, where the negated expression is subtracted from zero.

The additional reserved keywords for Caddiepy are: `def`, `return`, `log`, and `pow`, implemented in the parser as:

```
val kws_py = ["def", "return", "log", "pow"]
```

Keywords for the operators: `cos`, `sin`, and `exp` are already part of the parse implementation elsewhere in the `ast` file, so modifications of the operators are not needed for parsing the Python input syntax; it is the same.

5.3 Modifying the Unlinearized Output

To generate the unlinearized differentiated output, the pretty printer in `term_val` is modified. The `term_val` module evaluates the entire differentiated expression and prints all the intermediate computations needed to generate the output program code. The pretty printing function specifically writes the string output of the unlinearized code. The pretty printer for Caddiepy is named `pp_py`.

The `pp_py` function is called in the `unlinearised` function, as part of the main program code in `cad.sml`, as shown in 2.4.5. In `unlinearised`, the Python function declaration is printed and `pp_py` is called to evaluate the body of the function. It is implemented as:

```
278         val pling = if rad_p() then "_diff_reverse" else "_diff"
279         val () =
280             if print_diff_unlinearised_p() then
281               ( println "# Unlinearised differentiated program (python):"
282               ; List.app (fn (f,arg,d,gM,_) =>
283                             ( println_py ("def " ^ f ^ pling ^ "(" ^ V.pp_py
                              ↪  arg ^ "," ^ V.pp_py d ^ ")" ^ ":")
284                             ; println (V.ppM_py "" V.pp_py gM)
285                             ; println "")
286                          ) prg'
```

The pp_py function evaluates the internal value expressions of the program and writes strings of Python code for the output, as defined in section 4.2:

```
77  fun pp_py v =
78      case v of
79          R r => real_to_string r
80        | T vs => "" ^ String.concatWith "," (map pp_py vs) ^ ""
81        | Uprim(p,v) => Prim.pp_uprim_py (p, pp_py v)
82        | Add(v1,v2) => "(" ^ pp_py v1 ^ " + " ^ pp_py v2 ^ ")"
83        | Bilin(p,v1,v2) => "(" ^ Prim.pp_bilin_py p (pp_py v1) (pp_py v2) ^ ")"
84        | Var v => v
85        | Z => "0"
86        | Prj(i,v) => "(" ^ pp_py v ^ "[" ^ Int.toString (i-1) ^ "]" ^ ")"
87        | _ => die (pp v ^ " not defined for Caddiepy!")
```

For Caddiepy, that is the values of reals, tuples, primitive functions, addition, bilinear functions, variables, projection, and zero.

A function pp_uprim_py is added for printing primitive functions in compliance with the Python subset language. The primitive functions are the operators `sin`, `cos`, `log`, `exp`, `-` (negation), and `pow`. The function is implemented in `prim` as:

```
25    fun pp_uprim_py (p: uprim, v: string) =
26        case p of
27            Sin => "sin(" ^ v ^ ")"
28          | Cos => "cos(" ^ v ^ ")"
29          | Ln => "log(" ^ v ^ ")"
30          | Exp => "exp(" ^ v ^ ")"
31          | Neg => "-" ^ v
32          | Pow r => "pow(" ^ v ^ "," ^ real_to_string r ^ ")"
```

The bilinear functions are the linear algebra operators for scalar and vector computations. Only multiplication is implemented for the bilinear operators in Caddiepy. The Python syntax is generated with pp_bilin_py as:

```
48    fun pp_bilin_py b v1 v2 =
49        case b of
50            Mul => v1 ^ "*" ^ v2
```

In addition to pp_py the function ppM_py and ppM0_py are modified too in `term_val`. ppM0_py implements the function scope of defined function declarations so that variable bindings are separated by semicolon and that function expressions end with the

explicit `return` statement before the return expression, as defined by the grammar in 4.1.1:

```
69 fun ppM0_py (ind:string) (pp:v->string) (pp0:'a -> string) ((x,bs): 'a M) :
   ↪ string =
70     case bs of
71         nil => ind ^ "return " ^ pp0 x
72       | _ => let val bs = List.map (fn (var,v) => ind ^ "" ^ var ^ " = " ^ pp v
         ↪  ^ ";") bs
73              in String.concatWith " " bs ^ " " ^ ind ^ "return " ^ pp0 x
74              end
```

The functions `ppM_py` and `ppM0_py` are the pretty printer of the sequence of monad bindings from the internal program.

5.4 Line Comments

The handling of Python comment lines is implemented in the `SimpleToken.sml` file, that is part of the external `sml-parse` library. Python comment lines start with `#` and end with a newline `\n`. All characters in the comment line are not tokenized.

The reason for implementing it in the tokenizer and not in the parser is that the parser parses the program from tokens. If the comment line is tokenized, characters in the comment are processed as either symbol, id, or number tokens. This is undesirable since the tokens would have to be discarded or ignored so that they are not included in the parse tree, which over-complicates the process. It is therefore better to modify the tokenizer to ignore the Python line comments.

The tokenizer in `SimpleToken` handles a Python comment by implementing the comment state called `CommentP`. The state iterates through a Python comment string until a newline is reached. The implementation is shown below:

```
53             | CommentP (l0,"#") => (l',CommentP(l0,""),ts)
54             | CommentP (l0, "") =>
55               if c = #"\n" then (l',BeginS,ts)
56               else (l',CommentP(l0,""),ts)
57             | BeginS =>
58               if c = #"#" then (l',CommentP (l,"#"),ts)
```

6 User Guide

Guidance on how to setup Caddiepy and use the tool is provided in Appendix A. The guide is also available on the repository: github.com/kdsoup/caddiepy.

7 Evaluation

7.1 Testing

To verify that the Caddiepy program works as expected, a series of tests are implemented. There are three kinds of testing: reading of input files, writing of output

files, and unit tests of the differentiated Python functions. The tests are organized in the folder `test` into the subfolders: `input`, `output`, and `unit`.

7.1.1 Input Tests

It is tested if Caddiepy can read and evaluate Python script input files, and if the Python scripts are evaluated correctly. The Python files are written in the Python subset language defined in section 4.

The testing is done by using a Makefile setup that automates the process with bash commands. The Python subset scripts are passed as inputs to Caddiepy, and Caddiepy evaluates the scripts by using the `--Pexp` and `--Ptyped` options. These commands instruct Caddiepy to return the evaluated expression and type of the input script.

The printed output from Caddiepy is written to a `.out` file. The automated testing checks if the evaluated `.out` file is the same as the expected output stored in the `.out.ok` files. The bash command `diff` is used to check for differences. If the files match, `Test <filename>: OK` is written to a `.res` file. If there is a mismatch, the error is written to the `.res` file. A `complog.txt` file is added with the result output from the testing. When running Makefile in the terminal, a printed test overview is written to the terminal.

The Makefile test setup is modified from the Caddie tool [10], which has a similar test setup for verifying the `.cad` input files.

7.1.2 Output Tests

To verify that Caddiepy prints the correct unlinearized output, automated tests are implemented using Makefile. The test setup is similar to the input tests.

The Makefile bash commands generate unlinearized differentiated outputs of the input test files, using the Caddiepy option `--Pdiffu`. The differentiated results are written to `.py` files and the generated output Python files are compared to `.ok` files, to check for differences in the files. The `.ok` files are the expected differentiated output. If there are differences between the generated and expected files, errors will be logged to a `complog.txt` file. A brief test report overview is printed on the terminal as well.

7.1.3 Unit Tests

The Python module *Unittest* [16] is used to test that the input Python programs and the differentiated code from Caddiepy, can run as Python scripts and that they run correctly. This is implemented as unit tests, which are run with numerical values on the input functions and on the differentiated linear map output functions. Each unit test is composed into the three parts of *arrange*, *act*, and *assert*, to make the tests legible [17].

The automated unit tests verify that all written input Python test functions—from the folder `../test/input`—can run as Python programs and compute numerical results. An expected manually calculated result is compared to the Python input function result using `assertAlmostEqual`. The assert statement checks that the numerical results are within a six decimal accuracy of the expected result [6]. The unit tests for the Python input functions are implemented in `test_input.py`.

[6] `assertEqual` is not able to check decimal approximation, therefore `assertAlmostEqual` is preferred.

The same unit test method is used for testing the differentiated Python functions. These functions are the differentiated programs in the folder `../test/output`. The unit tests verify that both the derivative and adjoint code can run as Python scripts and that the numerical differentiated results are correct. The numerical results of the derivative and adjoint linear map functions are compared against an expected manually calculated result.

The test results of the unit tests are written to a log file and printed to the terminal when the tests are run using `Makefile`. If the tests are run in the terminal with Python, the results are only printed to the terminal output.

7.1.4 Results

The functionality of Caddiepy is tested with Python programs, that follow the definitions of the grammar in 4.1. Thirteen test files cover the testing of the binary- and unary operators, function declarations, variable bindings, and expressions. These are the `.py` files located in the `../test/input` folder.

The results of the input, output, and unit tests, all show successful results. For the input and output tests, 13 out of 13 tests complete successfully, and all of the 26 unit tests pass successfully.

A full coverage of the test results is available in Appendix B. Test results can be reproduced by running the makefile command: `make test` from the `src` folder in the terminal. The Python unit tests can be run independently from the `../test/unit` folder with the command: `python -m unittest -v`.

7.1.5 Continuous Integration Testing using GitHub Action

To make sure that new changes to the Caddiepy tool are working correctly, it is possible to configure continuous integration testing using GitHub Actions. This will ensure that code that is pushed to branches on the project on GitHub first will have to be verified by a test flow implemented using `.yml` files [18]. This is not implemented for the project, but it is good practice to enable test runs like this, to ensure that new implementations will not break the main project code when it is merged in.

7.2 Gradient Descent Implementation

It is desirable to know if Caddiepy can be used in a programming context beyond the unit tests and standalone tool control in the terminal. To test that Caddiepy can be used in a Python context program, a simple optimization implementation is provided, available in the `../src/sine` folder.

The Python program fits a third-order polynomial to the sine function using a supervised learning approach, that resembles the training of a simple neural network.

By using a basic Gradient Descent algorithm [19], the coefficients of the third-order polynomial $a + bx + cx^2 + dx^3$ are optimized to minimize the loss between the output values of the sine function y and the observed values from the approximated polynomial $\hat{y}$.

The coefficients a, b, c, d are first initialized to random values (weights), and then Gradient Descent is applied for 2000 iterations. The main implementation is taken from the PyTorch website [20]. However, instead of using the standard automatic differentiation steps, as explained in 2.2, Caddiepy is employed to derive the adjoint of the loss function, to compute the gradients of the coefficients for the Gradient Descent algorithm.

By declaring the loss function as:

```
def l(x): a = x[0]; b = x[1]; c = x[2]; d = x[3]; t = x[4]; return pow(((a + b
↪  * t + c * t * t + d * t * t * t) - sin(t)), 2.0)
```

with Caddiepy, the reverse-mode adjoint is computed to:

```
def l_diff_reverse(x1,x2,x3,x4,x5,dy): v1 = (x3*x5); v2 = (x4*x5); v3 =
↪  (v2*x5); v4 = ((2*((((x1 + (x2*x5)) + (v1*x5)) + (v3*x5)) + -sin(x5)))*dy);
↪  v5 = (v4*x5); v6 = (x2*v4); v8 = (v5*x5); v9 = (x3*v5); v10 = (v1*v4); v13
↪  = (v8*x5); v14 = (x4*v8); v15 = (v2*v5); v16 = (v3*v4); v17 =
↪  (cos(x5)*-v4); v18 = (((v6 + (v9 + v10)) + ((v14 + v15) + v16)) + v17);
↪  return v4,v5,v8,v13,v18
```

The differentiated loss function is then applied to compute the gradients for optimizing the coefficients, as seen on line 32 in the implementation:

```
20      for t in range(2000):
21          # Forward pass: compute predicted y
22          # y = a + b x + c x^2 + d x^3
23          X = [a, b, c, d, x]
24          y_pred = a + b * x + c * x ** 2 + d * x ** 3
25
26          # Compute and print loss
27          loss = np.square(y_pred - y).sum()
28          if t % 100 == 99:
29              print(t, loss)
30
31          # compute gradients of a, b, c, d with respect to loss using adjoint
            ↪  CAD
32          grad_a, grad_b, grad_c, grad_d, _ =
            ↪  lossdx.l_diff_reverse(a,b,c,d,x,1.0)
33
34          # Update weights
35          a -= learning_rate * grad_a.sum()
36          b -= learning_rate * grad_b.sum()
37          c -= learning_rate * grad_c.sum()
38          d -= learning_rate * grad_d.sum()
```

The polynomial fitted to the sine function gives the result:

```
Result: y = 0.03298233259390022 + 0.8545375859770113 x + -0.005690001200831127
↪  x^2 + -0.09301699490471449 x^3
```

and it is visualized in Figure 3 with the sine function.

8 Discussion

This part discusses the correctness of the Caddiepy tool, where the implementation lacks functionality, and how the tool can be improved. Caddiepy is briefly compared to JAX, which uses Jacobian matrices and traces.

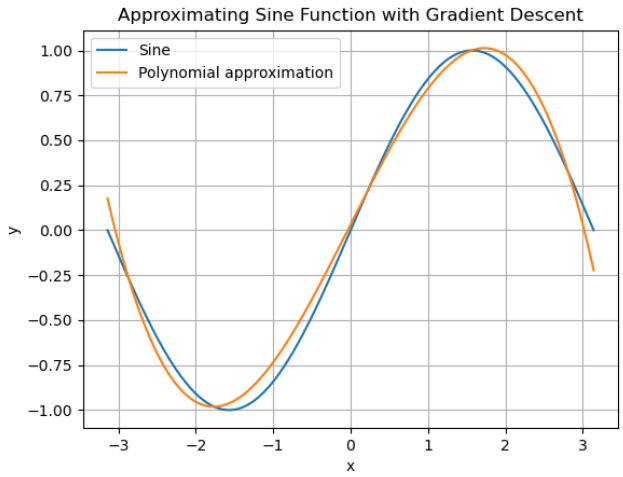

Figure 3: Third-order polynomial fitted to the sine function using Caddiepy for computing gradients.

8.1 Correctness

From the tests conducted in the evaluation, Caddiepy seems to work reliably. The Gradient Descent implementation shows that the tool can be used in a Python programming context.

Currently, the discrepancy between the allowed parameters of the input function declarations and the differentiated output function declarations breaks the consistency of the Python subset language. The input programs can only have a single parameter, and the output program can have multiple parameters. It would be better to allow for multiple parameters in the function declarations of the input programs as well. This will allow for higher-order derivatives (programs can be re-differentiated) and it will be easier to implement functions into a Python programming context, which can be differentiated with Caddiepy.

8.2 Limitations

Caddiepy, so far only works on scalar operations, which is shown in the Gradient Descent example. To fully take advantage of the mathematical theory of vector space differentiation inherent in Combinatory Automatic Differentiation, Caddiepy should be able to operate on linear algebra between vectors and matrices.

In neural network implementations, where tensors of weights and bias are represented as array matrices, it is crucial to be able to perform matrix multiplication using dot product operators. The Python subset language should therefore be extended to include operator definitions such as `dot` for dot-product and `cross` for the cross-product. This can be done in a Python context with functionality from NumPy. Furthermore, options to be able to perform summation on vector elements or sequentialized products will improve the functionality of Caddiepy, and allow for implementations of fully connected neural network architectures, where for example, summation is a crucial operation between the layers.

8.3 Extending the Functionality

8.3.1 Bilinear Operators

As discussed in the limitations of Caddiepy, the tool should be extended to include bilinear operators, such as the dot-product and cross-product, which can be used in a Python programming context.

The Python subset language can be extended to include definitions for the bilinear operators, such that a grammar might be defined as:

$$\begin{aligned} \mathit{bilinop} &:= \texttt{cross}(\mathit{exp}1\texttt{, }\mathit{exp}2) && (8.1)\\ &\mid \texttt{dot}(\mathit{exp}1\texttt{, }\mathit{exp}2) && (8.2) \end{aligned}$$

Furthermore, it would be useful to allow for computations such as the Euclidean norm and transpose of a matrix.

8.3.2 Lambda Functions

Instead of defining every function as a named function for Caddiepy, it is beneficial to extend the Python subset language to include anonymous function expressions.

Anonymous functions in Python are declared with the term `lambda`. The lambda function doesn't need the return statement, they'll just return the expression. The lambda functions can take one or more parameters as arguments and a parameter can also be a variable binding. The lambda function is defined as:

$$\texttt{lambda}\ \mathit{lambda\_params}\ \texttt{:}\ \ \mathit{exp} \tag{8.3}$$

where the parameters are defined as:

$$\begin{aligned} \mathit{lambda\_params} &:= \mathit{param}\texttt{, }\mathit{lambda\_params} && (8.4)\\ &\mid \mathit{varbind}\texttt{, }\mathit{lambda\_params} && (8.5)\\ &\mid \mathit{param} && (8.6) \end{aligned}$$

However, the anonymous lambda function doesn't allow variable bindings in the body of the function, as the named functions do. Therefore, the lambda function cannot be used as a substitute for a let-binding expression. It is not possible to write something like:

```
lambda x, y: y = 2; x + y
```

The assignment of `y = 2` in the expression body is not allowed.

There is a workaround to this where lambda functions are nested, to make variable bindings in the expressions as a form of let-binding, but quickly this kind of programming becomes difficult to write and read. It is better to avoid it and use named function declarations instead of variable bindings in the function scope.

8.3.3 Array Functions and Conditionals

Definitions of if-else conditionals and map-functionality are proposed on the Combinatory Automatic Differentiation repository page [10], which is not implemented

in Caddiepy, and it is not defined for the Python subset language. With if-else conditionals, more complex algorithmic programming structures can be implemented. It is therefore desirable to implement differentiation of the conditionals and extend the parser and pretty printer in Caddiepy to handle the if-else syntax, and even expand it to handle if-then-else too.

Array functionalities such as map and reduce, are relevant when targeting Caddiepy to work in a neural network implementation. Reduce can be employed as a summation or sequentialized product. Definitions for arrays are not defined for Caddiepy, so instead tuples can be used as iterable representations in Python, and the array functions can be defined as:

$$
\begin{aligned}
arrfun &:= \texttt{tuple(range(}i\texttt{))} && (8.7)\\
&\mid \texttt{tuple(map(}fun\texttt{, }arr\texttt{))} && (8.8)\\
&\mid \texttt{reduce(}fun\texttt{, }arr\texttt{, }num\texttt{)} && (8.9)
\end{aligned}
$$

where i is a positive zero-index and `range` is the iota function which initializes a range from zero up to i. For `map` and `reduce`, arr is a tuple representation of an array, and fun is a named function or lambda function expression. However, to extend the Python context to include these operators, differentiation and point-free notation of the operators have to be defined and implemented as well.

8.3.4 Complex Numbers

It is worth considering if Caddiepy should be able to handle complex numbers. For example, the result of the equation below is a complex number, when the natural logarithm is applied to a negative number:

$$
\begin{aligned}
\ln(3.0 \cdot \cos(2.0)) &= \ln(3.0 \cdot -0.4161468365)\\
&= 0.2218951804 + 3.141592654\ I
\end{aligned}
$$

A way of handling this would be to implement complex numbers as tuples, with the first element representing the real part and the second element the imaginary. This makes it possible to differentiate a function with Caddiepy, where the complex numbers are treated as tuples of scalars. The Pretty Printer in Caddiepy can then be extended to print the complex number tuple representation to num + num`j`, which can be used in a Python context. However, as suggested in the paper by [21] linearity cannot be taken for granted in complex arithmetic, and therefore different assumptions hold for differentiation with complex numbers. Further study of how differentiation with complex numbers should be implemented with Combinatory Automatic Differentiation might therefore be needed.

8.4 Other Tools

JAX is a tool for Python that is designed for optimal array computation in large-scale machine learning, and it has functionality for automatic differentiation. JAX uses Jacobian-Vector products (JVP) for its computation of forward-mode automatic differentiation and Vector-Jacobian products (VJP) for its reverse-mode automatic differentiation [22]. The automatic differentiation is implemented with the use of traces for intermediate representations in the process, as explained in 2.2, and thereby increases the memory usage when the depth of the Jacobian matrices expands.

When looking at the type definition of the forward-mode VJP, expressed in Haskell-like signatures [22]:

```
jvp :: (a -> b) -> a -> T a -> (b, T b)
```

it represents a linear map, where `T a` is the tangent variable, which is very similar to the forward-mode differentiation in Caddiepy, defined in (2.20). This suggests potential similarities between the two tools, which might be worth studying, especially since the Combinatory Automatic Differentiation approach avoids the memory-intensive Jacobian computations and refrains from using traces, also called *tapes* by [23], to store intermediate values for the backpropagation process.

Another approach is the Futhark implementation for automatic differentiation with nested parallelism, which is introduced in [24]. The theory proposes reverse-mode automatic differentiation as VJP, which eliminates the use of tapes by exploiting nested function scopes. The technique enables differentiation of loop-conditionals, and the map and reduce operators, which is relevant for Combinatory Automatic Differentiation.

9 Conclusion

This project demonstrates that the theory and implementation of Combinatory Automatic Differentiation can be used in a Python context and that the tooling has potential for neural network implementations in Python.

It was explained, how the theory of the Fréchet derivative was implemented in Caddie, and how the use of monad types and point-free notation allowed symbolic differentiation to take a compositional form to avoid expression swelling and provide efficient computation.

By changing the parser and the pretty printer in Caddie, the tool was modified to work in a Python context. The modified tool is called Caddiepy, and the Python context was defined as a subset of the Python language to comply with functionality restrictions as part of a development and design process.

The tool was tested with unit tests to verify the functionality, and a context implementation was provided to demonstrate how Caddiepy can be utilized to compute the gradients in a simple supervised optimization task.

9.1 Future Work

Caddiepy should be extended to work with bilinear operators, such as the dot product so that the tool can integrate in neural network implementations with multiple layers and activations functions, where parameters are represented as matrices.

The Python subset language should be extended so that multiple arguments are permitted in the input functions to allow for the computation of second-order derivatives, and to make it more seamless to implement functions in a Python programming context that can be differentiated with Caddiepy. Furthermore, it would be useful to extend Caddiepy with array functionality operators such as map and reduce, to be able to serialize equations.

There's great potential for the tool, and it would be exciting to see Caddiepy being used in more involved differentiation tasks for computing gradients in complex algorithmic structures.

References

[1] E. B. Halvorsen, "Calculating key ratios for financial products using automatic differentiation and monte carlo simulation," *Student Project, Department of Computer Science, University of Copenhagen (DIKU)*, 2012. [Online]. Available: http://hiperfit.dk/pdf/ad_esben.pdf 5, 6, 7, 9, 11

[2] Wikipedia, "Derivative - wikipedia," 2025, last accessed 27 January 2025. [Online]. Available: https://en.wikipedia.org/wiki/Derivative 5, 6

[3] M. Elsman, F. Henglein, R. Kaarsgaard, M. K. Mathiesen, and R. Schenck, "Combinatory adjoints and differentiation," *Ninth Workshop on Mathematically Structured Functional Programming (MSFP 2022), Munich, Germany*, 2022. [Online]. Available: https://elsman.com/pdf/msfp22.pdf 7, 8, 12, 13, 15

[4] Wikipedia, "Fréchet derivative - wikipedia," 2025, last accessed 12 March 2025. [Online]. Available: https://en.wikipedia.org/wiki/Frechet_derivative 8

[5] C. M. Bishop and H. Bishop, *Deep Learning - Foundations and Concepts*, S. Cham, Ed., 2023. 8, 9, 10, 11

[6] Wikipedia, "Product rule - wikipedia," 2025, last accessed 27 January 2025. [Online]. Available: https://en.wikipedia.org/wiki/Product_rule 8

[7] ——, "Chain rule - wikipedia," 2025, last accessed 28 January 2025. [Online]. Available: https://en.wikipedia.org/wiki/Chain_rule 9

[8] ——, "Automatic differentiation - wikipedia," 2025, last accessed 28 January 2025. [Online]. Available: https://en.wikipedia.org/wiki/Automatic_differentiation 11

[9] ——, "Transpose of a linear map - wikipedia," 2025, last accessed 1 March 2025. [Online]. Available: https://en.wikipedia.org/wiki/Transpose_of_a_linear_map 12

[10] D. of Computer Science at the University of Copenhagen, "Caddie," 2025, last accessed 1 March 2025. [Online]. Available: https://github.com/diku-dk/caddie 13, 20, 21, 26, 31, 35

[11] Wikipedia, "Tacit programming - wikipedia," 2025, last accessed 29 January 2025. [Online]. Available: https://en.wikipedia.org/wiki/Tacit_programming 14

[12] ——, "Monad (functional programming) - wikipedia," 2025, last accessed 2 February 2025. [Online]. Available: https://en.wikipedia.org/wiki/Monad_(functional_programming) 15

[13] D. of Computer Science at the University of Copenhagen, "sml-parse," 2025, last accessed 1 March 2025. [Online]. Available: https://github.com/diku-dk/sml-parse 21

[14] NumPy, "Numpy," 2025, last accessed 1 March 2025. [Online]. Available: https://numpy.org/ 22

[15] Python, "2. lexical analysis — python 3.13.2 documentation," 2025, last accessed 15 March 2025. [Online]. Available: https://docs.python.org/3/reference/lexical_analysis.html#string-literal-concatenation 24

[16] ——, "unittest — unit testing framework — python 3.13.2 documentation," 2025, last accessed 26 February 2025. [Online]. Available: https://docs.python.org/3/library/unittest.html 31

[17] G. M. Hall, *Adaptive Code: Agile coding with design patterns and SOLID principles, 2nd Edition*. Microsoft Press, 2017. 31

[18] GitHub, "Github actions documentation - github docs," 2025, last accessed 24 January 2025. [Online]. Available: https://docs.github.com/en/actions 32

[19] R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein, *Introduction to Algorithms, fourth edition*. The MIT Press, 2022. 32

[20] PyTorch, "Learning pytorch with examples — pytorch tutorials 2.6.0+cu124 documentation," 2025, last accessed 26 February 2025. [Online]. Available: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html 32

[21] N. Krämer, "A tutorial on automatic differentiation with complex numbers," 2024. [Online]. Available: https://arxiv.org/abs/2409.06752 36

[22] JAX, "The autodiff cookbook — jax documentation," 2025, last accessed 5 March 2025. [Online]. Available: https://docs.jax.dev/en/latest/notebooks/autodiff_cookbook.html 36

[23] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. Siam, 2008. 37

[24] R. Schenck, O. Rønning, T. Henriksen, and C. E. Oancea, "Ad for an array language with nested parallelism," Los Alamitos, CA, USA, nov 2022. [Online]. Available: https://futhark-lang.org/publications/sc22-ad.pdf 37

A Caddiepy Setup and Use Guide

A.1 Setup

To be able to run Caddiepy, the Standard ML compiler toolkit MLKit is required. The toolkit is available at github.com/melsman/mlkit. The MLKit repository provides instructions for installing the toolkit. For macOS ARM computers, MLKit has to be installed using Rosetta.

When MLKit is installed, download or clone the Caddiepy repository at github.com/kdsoup/caddiepy. In the terminal, navigate to the `caddiepy` folder and type the command `make`. The Makefile command will compile the Caddiepy source code to an executable program using MLKit. The executable named `./cad` is available in the `src` folder.

Python and NumPy are required to be able to run the Python scripts.

A.2 How to Use

To run the program, navigate to the `src` folder in the terminal, and type:

```
./cad --Pdiffu some-file.py
```

With this command, Caddiepy will compute the unlinearized linear map derivative of the input python file `some-file.py`, and print the result to the terminal. To get the adjoint linear map of an input program, type:

```
./cad -r --Pdiffu some-file.py
```

where `-r` is the reverse-mode option command.

Caddiepy has the following options, which are listed by typing `./cad --help`:

```
-r
  Apply reverse mode AD.
--verbose
  Be verbose.
-e ()
  Expression to be evaluated after loading of program files.
--help
  Print usage information and exit.
--version
  Print version information and exit.
--Ptyped
  Print program after type inference.
--Pexp
  Print internal expression program.
--Ppointfree
  Print point free internal expression program.
--Pdiff
  Print differentiated program.
--Pdiffu
  Print unlinearised differentiated program.
```

B Test Report

```
-------T E S T --- R E P O R T-------

-------------- I N P U T --------------
Combinatory AD (CAD) for Python v0.0.1

Test add: OK
Test cos: OK
Test exp: OK
Test ln: OK
Test mul: OK
Test neg: OK
Test pow: OK
Test proj: OK
Test simple1: OK
Test simple2: OK
Test simple3: OK
Test sin: OK
Test sub: OK

Tests succeeded:         13 /      13
Test errors:              0 /      13
See complog.txt
----------------------------------------

------------ O U T P U T --------------
Combinatory AD (CAD) for Python v0.0.1

Test add: OK
Test cos: OK
Test exp: OK
Test ln: OK
Test mul: OK
Test neg: OK
Test pow: OK
Test proj: OK
Test simple1: OK
Test simple2: OK
Test simple3: OK
Test sin: OK
Test sub: OK

Tests succeeded:         13 /      13
Test errors:              0 /      13
See complog.txt
----------------------------------------

-------------- U N I T ----------------
test_add (test_input.TestPyInputFiles.test_add) ... ok
test_cos (test_input.TestPyInputFiles.test_cos) ... ok
test_exp (test_input.TestPyInputFiles.test_exp) ... ok
test_ln (test_input.TestPyInputFiles.test_ln) ... ok
```

```
test_mul (test_input.TestPyInputFiles.test_mul) ... ok
test_neg (test_input.TestPyInputFiles.test_neg) ... ok
test_pow (test_input.TestPyInputFiles.test_pow) ... ok
test_proj (test_input.TestPyInputFiles.test_proj) ... ok
test_simple1 (test_input.TestPyInputFiles.test_simple1) ... ok
test_simple2 (test_input.TestPyInputFiles.test_simple2) ... ok
test_simple3 (test_input.TestPyInputFiles.test_simple3) ... ok
test_sin (test_input.TestPyInputFiles.test_sin) ... ok
test_sub (test_input.TestPyInputFiles.test_sub) ... ok
test_add (test_output.TestPyOutputFiles.test_add) ... ok
test_cos (test_output.TestPyOutputFiles.test_cos) ... ok
test_exp (test_output.TestPyOutputFiles.test_exp) ... ok
test_ln (test_output.TestPyOutputFiles.test_ln) ... ok
test_mul (test_output.TestPyOutputFiles.test_mul) ... ok
test_neg (test_output.TestPyOutputFiles.test_neg) ... ok
test_pow (test_output.TestPyOutputFiles.test_pow) ... ok
test_proj (test_output.TestPyOutputFiles.test_proj) ... ok
test_simple1 (test_output.TestPyOutputFiles.test_simple1) ... ok
test_simple2 (test_output.TestPyOutputFiles.test_simple2) ... ok
test_simple3 (test_output.TestPyOutputFiles.test_simple3) ... ok
test_sin (test_output.TestPyOutputFiles.test_sin) ... ok
test_sub (test_output.TestPyOutputFiles.test_sub) ... ok

----------------------------------------------------------------------
Ran 26 tests in 0.001s

OK
make[1]: *** No rule to make target `-i', needed by `all'.  Stop.
make: [test] Error 2 (ignored)
```