Higher-Order Automatic Differentiation and Its Applications

by

Songchen Tan

B.S. in Chemistry, Peking University (2021)
B.S. in Physics, Peking University (2021)

Submitted to the Center for Computational Science and Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Computational Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

© 2023 Songchen Tan. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by:	Songchen Tan Center for Computational Science and Engineering May 12, 2023
Certified by:	Alan Edelman Department of Mathematics Thesis Supervisor
Accepted by:	Kate Nelson Academic Administrator Center for Computational Science and Engineering

Higher-Order Automatic Differentiation and Its Applications

by

Songchen Tan

Submitted to the Center for Computational Science and Engineering
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Science in Computational Science and Engineering

Abstract

Differentiable programming is a new paradigm for modeling and optimization in many fields of science and engineering, and automatic differentiation (AD) algorithms are at the heart of differentiable programming. Existing methods to achieve higher-order AD often suffer from one or more of the following problems: (1) exponential scaling with respect to order due to nesting first-order AD; (2) ad-hoc handwritten higher-order rules which are hard to maintain and do not utilize existing first-order AD infrastructures; (3) inefficient data representation and manipulation that causes significant overhead at lowered-order when compared to nesting highly-optimized first-order AD libraries. By combining advanced techniques in computational science, i.e., aggressive type specializing, metaprogramming, and symbolic computing, we introduce a new implementation of Taylor mode automatic differentiation in Julia that addresses these problems. The new implementation shows that it is possible to achieve higher-order AD with minimal overhead and without sacrificing the performance of lower-order AD and obtain significant speedup in real-world scenarios over the existing Julia AD library. In addition, this implementation automatically generates higher-order AD rules from first-order AD rules, which is a step towards a general framework for higher-order AD.

Thesis supervisor: Alan Edelman

Title: Professor of Applied Mathematics

Acknowledgments

I would like to express my sincere gratitude to my advisors, Professor Alan Edelman and Dr. Christopher Rackauckas, for their invaluable guidance and support throughout my master's study. Their insights, expertise, and encouragement have been instrumental in shaping my research projects in automatic differentiation and pushing me to strive for excellence.

I am also deeply grateful to Shashi Gowda, a Ph.D. student in the Julia lab, for his generous help in using his symbolic computing package. His patient guidance and willingness to share his knowledge have been crucial in helping me navigate through the challenges of the automatic generation of higher-order rules.

I am indebted to members of the Julia community, especially Keno Fischer, for inspiring discussions on theoretical viewpoints of automatic differentiation and their unwavering support for the development of the Julia language. Their passion and dedication to the pursuit of scientific excellence have been a constant source of inspiration for me.

Finally, I would like to express my heartfelt appreciation to my parents for their unwavering support and encouragement during my work for my master's degree. Their sacrifices and belief in me have been a driving force behind my success, and I am forever grateful for their love and guidance.

In conclusion, I would like to extend my sincere thanks to everyone who has contributed to my work in any way. Your support, guidance, and encouragement have been indispensable, and I am deeply grateful for your contributions to my academic and personal growth.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Differentiable programming and automatic differentiation

1.1.1 Differentiable programming

Differentiable programming is an emerging paradigm that unifies computational modeling with gradient-based optimization, allowing the computation of gradients of functions with respect to their input parameters. The program, or the model represented by the program can therefore be optimized with gradient-based methods. This paradigm has become increasingly popular in recent years, as it provides a powerful tool for optimization problems, machine learning, and scientific modeling.[1, 2]

1.1.2 Automatic differentiation

Differentiable programming is based on the fundamental technique of automatic differentiation, which computes exact derivatives of functions through a sequence of algorithmic transformations.

Automatic differentiation is a technique that differs from other differentiation methods, such as symbolic differentiation and numerical differentiation[3]. Symbolic differentiation involves computing the derivative of a function by manipulating symbolic expressions. While symbolic differentiation can be useful for simple functions, it can be cumbersome for complicated functions and may suffer from expression swell, where the expression for the derivative becomes exponentially larger than the original expression. This issue can be mitigated by simplifying the derivative, but the simplification process can be costly[4]. Numerical differentiation, on the other hand, approximates the derivative of a function by computing the difference quotient using finite differences. This method is simple to implement, but it can be prone to numerical errors and may require many function evaluations, which can be computationally expensive[5].

In contrast, automatic differentiation avoids the limitations of both symbolic and numerical differentiation by computing derivatives using a recursive application of the chain rule[6]. This approach is efficient and accurate, and it can compute exact derivatives to machine precision for any differentiable function (at least theoretically), regardless of its complexity.

In summary, differentiable programming offers a powerful paradigm for optimization problems, machine learning, and scientific modeling, and automatic differentiation provides a robust and efficient method for computing derivatives.

1.1.3 Variations of automatic differentiation

Automatic differentiation can be performed in forward mode, reverse mode, or mixed mode. In forward mode, the derivative of a function is computed by calculating the derivative of each variable in the function with respect to the input and propagating through the sequence of operations. In reverse mode, the derivative of a function is computed by calculating the derivative of the output with respect to each variable in the function and propagating reversely through the sequence of operations. In the mixed mode, these two directions may be arbitrarily combined during the computation[4].

In addition, there are two main approaches to implement automatic differentiation: operator overloading and source code transformation.

Operator overloading involves defining new mathematical operations on custom types that contain derivative information, such as addition, multiplication, and elementary functions, so that the derivative of a function can be computed by recursively applying the chain rule to the individual operations. This approach is straightforward to implement and is widely used in programming languages that support operator overloading. However, operator overloading may suffer from issues related to efficiency and type stability, as the resulting code can be less optimized[4].

Source code transformation, on the other hand, involves transforming the source code of a function to a new code that computes the derivative of the function. This approach can be more efficient than operator overloading, as it avoids the overhead of repeatedly calling overloaded operators and allows for more precise control over the numerical computations. However, source code transformation can be more complex to implement than operator overloading, as it requires understanding the semantics of an arbitrarily complicated program and producing the correct code to compute its derivatives[6].

Both operator overloading method and source code transformation method can be utilized to implement forward mode, reverse mode, or mixed mode automatic differentiation[7]. The choice of variation to use depends on the specific requirements of the problem at hand, such as the complexity of the function to be differentiated, the accuracy and efficiency of the differentiation, and the programming language and tools available. For example, operator overloading may be preferred for simple functions and quick prototyping, while source code transformation may be more suitable for complex functions and production-level code.

1.2 Higher-order automatic differentiation

1.2.1 Higher-order derivatives in differentiable programming

In scientific modeling, higher-order derivatives are often needed to compute physical quantities such as acceleration, curvature, and higher moments of a distribution[1, 8, 9]. As a result, many models, especially models expressed as ordinary differential equations (ODE) and partial differential equations (PDE), require efficient higher-order derivative computation via automatic differentiation. Although some automatic differentiation tools allow nesting, naively repeatedly applying first-order automatic differentiation methods to obtain higher-order derivatives can have efficiency and correctness problems[10, 11]. For example, the operator-overloading method suffers from exponential scaling with respect to order, and the source code transform method often produces code that cannot be differentiated efficiently again by itself[12]. Therefore, dedicated higher-order automatic differentiation methods and designs are needed to efficiently compute higher-order derivatives.

1.2.2 Taylor mode automatic differentiation

Insights are obtained if we recognize various automatic differentiation methods as different ways of applying the chain rule. For example, forward mode automatic differentiation can be viewed as applying the chain rule in the forward direction, while reverse mode automatic differentiation can be viewed as applying the chain rule in the reverse direction. In its most simple form (first-order, scalar input, scalar output), the chain rule can be expressed as follows:

$$\frac{d}{dx}f(g(x)) = f'(g(x)) \cdot g'(x), \tag{1.1}$$

where f' and g' are the derivatives of f and g with respect to their inputs, respectively. This expression can be generalized to higher-order derivatives by using Faà di Bruno's formula, which states that the n-th derivative of a composite function is a sum of products of various orders of derivatives of the component functions[13]:

$$\frac{d^n}{dx^n}f(g(x)) = (f \circ g)^{(n)}(x) = \sum_{\pi \in \Pi} f^{(|\pi|)}(g(x)) \cdot \prod_{B \in \pi} g^{(|B|)}(x), \tag{1.2}$$

where Π is the set of all partitions of the set $\{1, \ldots, n\}$, $|\pi|$ is the number of blocks in the partition π, and $|B|$ is the number of elements in the block B.

Since Faà di Bruno's formula is a generalization of the chain rule used in first-order automatic differentiation, one can directly apply this formula to achieve higher-order automatic differentiation. Compared to the naive approach of repeatedly applying first-order automatic differentiation methods, Faà di Bruno's formula provides a more efficient and accurate way to compute higher-order derivatives by avoiding a lot of redundant computations. The resulting method is called Taylor mode automatic differentiation, as it can be viewed as applying the chain rule in the Taylor series representation of a function[10].

In summary, higher-order derivatives are important in scientific modeling, but repeatedly applying first-order automatic differentiation methods to obtain them can be inefficient and numerically unstable. Faà di Bruno's formula provides a more efficient and accurate way to compute higher-order derivatives by generalizing the chain rule used in first-order automatic differentiation, resulting in the Taylor mode automatic differentiation method.

1.2.3 Problems with existing implementations of Taylor mode automatic differentiation

However, the existing algorithms for implementing Taylor-mode automatic differentiation have some significant limitations. Within the Julia language community, TaylorSeries.jl[14] provides a systematic treatment of Taylor polynomials in one and several variables, but it only supports a very limited set of primitives (via ad-hoc handwritten rules). In addition, its mutating and scalar code isn't great for speed, and therefore despite the theoretical advantage over nested first-order derivatives, it is often slower compared to packages like ForwardDiff[15] in orders <4. Similarly, in the Python community, `jax.jet` in the JAX framework[10] is an experimental implementation of Taylor-mode automatic differentiation, but it suffers from the same problem of handwritten higher-order rules, and is not composable with other parts of JAX. Other implementations of Taylor-mode automatic differentiation, such as the one in the Tapenade tool[16], have similar problems.

1.3 Julia language and related implementations

1.3.1 Julia language for differentiable programming

Julia is a high-level, dynamic programming language designed for numerical and scientific computing[17, 18]. It is designed to be fast and expressive, with syntax that is familiar to users of other technical computing environments such as MATLAB and Python. Julia's features include a just-in-time (JIT) compiler, multiple dispatch, type inference, and a flexible macro system. These features make Julia well-suited for differentiable programming, which requires a combination of performance and expressiveness.

One of the strengths of Julia is its ability to support multiple dispatch, which allows for efficient, generic programming. In particular, Julia's support for multiple dispatch makes it possible to define overloads for custom types, which is essential for building operator-overloading based first- and higher-order automatic differentiation tools.[19, 20]

1.3.2 First-order implementations

Julia has several implementations of first-order automatic differentiation, including:

- ForwardDiff.jl[15]: This is a fast and reliable implementation of forward-mode automatic differentiation in Julia. It is based on operator overloading and supports scalar and vector-valued functions. ForwardDiff.jl is widely used in the Julia community and is the recommended package for forward-mode automatic differentiation.

- ReverseDiff.jl[21]: This is an implementation of reverse-mode automatic differentiation in Julia. It is based on operator overloading and supports a variety of types.

- Zygote.jl[12]: This is a source-to-source automatic differentiation tool in Julia which supports reverse-mode automatic differentiation.

- Enzyme.jl[22, 23]: This is a source-to-source automatic differentiation tool in Julia that supports both forward- and reverse-mode automatic differentiation. Unlike Zygote.jl which works on the untyped Julia IR, Enzyme.jl works on the LLVM IR, which allows it to efficiently handle scalar operations, control flow and other complex features.

- AbstractDifferentiaton.jl[24]: This is an abstraction over different implementations of automatic differentiation in Julia. It provides a common interface for forward- and reverse-mode automatic differentiation and supports scalar and vector-valued functions.

1.3.3 Higher-order implementations

In addition to these first-order automatic differentiation packages, Julia also has several implementations of higher-order automatic differentiation, including:

- TaylorSeries.jl[14]: This is a package for performing automatic differentiation using truncated Taylor series expansions in one or many variables.

- Diffractor.jl[25]: This is a next-generation source-code transformation-based forward-mode and reverse-mode AD package, but its higher-order functionality is currently only a proof-of-concept.

Overall, Julia's flexibility and performance make it a powerful tool for differentiable programming, with a variety of options for performing first- and higher-order automatic differentiation.

1.3.4 Potential applications of higher-order automatic differentiation in the Julia community

Although there hasn't been a satisfactory implementation of Taylor mode automatic differentiation in Julia or any other language, higher-order automatic differentiation has a wide range of potential applications in the Julia community.

One application is in the field of physical modeling, where higher-order derivatives are needed to accurately model complex physical phenomena. For example, the NeuralPDE.jl[26] package uses higher-order automatic differentiation to solve differential equations for complex systems in physics and biology.

In addition, higher-order automatic differentiation can be used for uncertainty quantification, where higher-order derivatives can be used to compute error bounds and sensitivity analysis. This has applications in finance, machine learning, and scientific modeling.

Overall, the potential applications of higher-order automatic differentiation in the Julia community are vast and varied, and the availability of efficient and flexible automatic differentiation tools in Julia is likely to accelerate progress in many areas of research and development.

1.4 Goals of this work

The goal of this work is to develop a new implementation of Taylor-mode automatic differentiation in Julia that is efficient, composable, and easy to use. The detailed goals are as follows:

- Reasonable scaling with the order of differentiation (while naively composing first-order differentiation would result in exponential scaling)
- Automatic generation of higher-order derivative rules to avoid the need for manual implementation of higher-order rules
- Same performance with ForwardDiff.jl[15] on first order and second order, so there is no penalty of abstraction caused by the implementation of Taylor mode
- Composable with other automatic differentiation systems like Zygote.jl[12], so that the user can use it together with other tools for the computation of gradients (e.g. compute the physical derivatives with Taylor mode and then optimized with gradient-based optimization techniques with Zygote.jl)

- Easy to use for both experts and non-experts in automatic differentiation

By combining advanced techniques in computational science and engineering, i.e. aggressive type specializing, metaprogramming and symbolic computing, this work will meet the above goals and provide a new perspective for the Julia community.

Chapter 2

Theory

2.1 Geometric formulation of higher-order automatic differentiation

2.1.1 Taylor series representation

In first-order forward-mode automatic differentiation, the computation of a function's derivative at a point is performed by evaluating the function on a **dual number**. A dual number is a mathematical construct of the form $x_0 + x_1\varepsilon$, where x_0 and x_1 are real numbers and ε is a non-zero number satisfying the property $\varepsilon^2 = 0$. The value x_0 represents the function's value at a given point, while the value x_1 represents the function's derivative at that point. If we extend the domain of elementary functions f on dual numbers, we will get:

$$f(x_0 + x_1\varepsilon) = f(x_0) + f'(x_0)x_1\varepsilon + \frac{1}{2}f''(x_0)x_1^2\varepsilon^2 + \cdots = f(x_0) + f'(x_0)x_1\varepsilon, \tag{2.1}$$

since the higher-order Taylor expansion terms are zero. Therefore, by computing a function's value on a dual number instead of a real number, we can automatically compute its derivative using the properties of dual numbers.

In programmatic implementations, dual numbers are often stored as number pairs (x_0, x_1), and we need to define a set of rules for elementary functions and arithmetic operations on dual numbers to ensure that the derivative of a function is computed correctly. For example, function sin on the number pair is defined as:

$$\sin(x_0, x_1) = (\sin(x_0), \cos(x_0)x_1), \tag{2.2}$$

and the product of two dual numbers (x_0, x_1) and (y_0, y_1) is computed as:

$$(x_0, x_1) \cdot (y_0, y_1) = (x_0y_0, x_0y_1 + x_1y_0). \tag{2.3}$$

Therefore, in an arbitrarily complicated function, the input dual number goes through a series of arithmetic operations and elementary functions, and the output dual number contains the derivative of the function at the input point.

In higher-order automatic differentiation, **truncated Taylor series** extends the idea of dual numbers. A truncated Taylor series is a mathematical construct of the form $x_0+x_1t+x_2t^2+\cdots+x_nt^n$, where t is a non-zero number satisfying $t^{n+1} = 0$, where n is the order of differentiation of interest. The value x_0 represents the function's value at a given point, while the value x_1 represents the function's first derivative at that point, and so on. If we extend the domain of elementary functions f on truncated Taylor series, we will get:

$$f(x_0 + x_1t + x_2t^2 + \cdots) = f(x_0) + f'(x_0)x_1t + [f''(x_0)x_2 + f'^2(x_0)x_1^2]t^2 + \cdots \tag{2.4}$$

In programmatic implementations, we also define a set of rules accordingly. For example,

$$(x_0, x_1, \cdots, x_n) \cdot (y_0, y_1, \cdots, y_n) = (x_0y_0, x_0y_1 + x_1y_0, x_0y_2 + x_1y_1 + x_2y_0, \cdots) \tag{2.5}$$

Therefore, by computing a function's value on a truncated Taylor series instead of a real number, we can automatically apply the Faà di Bruno's formula[13] and thereby extract its derivatives of orders up to n.

2.1.2 From numbers to manifolds

The truncated Taylor series representation of higher-order automatic differentiation is intuitive and follows nicely from elementary calculus. However, in the context of differentiable programming, we care not only about differentiating on a predefined set of types, but also arbitrary custom types in arbitrary programmatic constructs[1]. In fact, there is a more general framework for understanding higher-order automatic differentiation, which is based on the theory of differential geometry.

Differential geometry is the branch of mathematics that studies geometric properties and structures for smooth (differentiable) shapes and smooth spaces. In the context of automatic differentiation, differential geometry provides a framework for understanding the behavior of arbitrary types that undergoes automatic differentiation, as long as the type corresponds to a differentiable Riemannian manifold. Although the idea has been explored by various texts[27, 28, 29, 30, 31, 32], it has not been fully elaborated with a general form for arbitrary order and programmatic types.

To begin with, we assume the space of possible values of a type `T` can be viewed as a high-dimensional **differentiable manifold** M, with each point on the manifold representing a different value. The **tangent space** at a point x_0 on the manifold, $T_{x_0}M$, represents the space of first-order derivatives. The **tangent bundle** is defined as:

$$\begin{aligned} TM &= \bigcup_{x_0 \in M} \{x_0\} \times T_{x_0}M \\ &= \{(x_0, x_1) | x_0 \in M, x_1 \in T_{x_0}M\}. \end{aligned} \tag{2.6}$$

For a differentiable function $f : M \to N$, the **pushforward** f_* is a mapping from the tangent space $T_{x_0}M$ to the tangent space $T_{f(x_0)}N$ that maps the tangent vector $x_1 \in T_p x$ to the tangent vector $f_*(x_1) \in T_{f(x_0)}N$. The pushforward f_* is defined as:

$$f_*(x_1) = \left.\frac{d}{dt}\right|_{t=0} f(\gamma(t)), \tag{2.7}$$

where γ is a curve on M such that $\gamma(0) = x_0$ and $\gamma'(0) = x_1$. Therefore, by defining pushforward rules for relevant types and functions, we can achieve the first-order (forward-mode) automatic differentiation.

We can also extend the above discussion to higher order. The **Taylor spaces** at a point x_0 on the manifold, $T^n_{x_0}M$, represents the space of n-th order derivatives. The **Taylor bundle** is defined as:

$$T^nM = \{(x_0, x_1, x_2, \cdots, x_n) | x_0 \in M, x_i \in T^i_{x_0}M\}. \tag{2.8}$$

For a differentiable function $f : M \to N$, the **higher-order pushforward** f_* can also be similarly defined to map the derivatives to the corresponding spaces $T^n_{x_0}N$. Therefore, by defining higher-order pushforward rules for relevant types and functions, we can achieve higher-order automatic differentiation.

The geometry of these tangent spaces plays an important role in the design and implementation of automatic differentiation algorithms. For example, the ChainRules.jl package[33] uses a combination of natural tangents, structural tangents and special tangents to write efficient rules for a variety of Julia types, and handles lazy evaluation and control flow elegantly.

In summary, differential geometry provides a powerful framework for understanding the relationships between higher-order derivatives of types that correspond to various spaces, and this understanding can be used to develop more general and robust program transformations for automatic differentiation.

2.2 Bundle representation

We further assume the elements in the Taylor bundle also correspond to some type in the programming language. In the context of Julia's type system, we can define a **bundle type** for the Taylor bundle T^nM as `B = Tuple{T, T1, T2, ..., Tn}`, where each value of this type correspond to a mathematical tuple $x = (x_0, x_1, \cdots, x_n)$.

For many practical types such as scalars and arrays, there is a natural isomorphism between the tangent space and the original space, so programmatically all components of the bundle can be stored in a value of original type `T`. Therefore we can conveniently express the bundle type in a parameterized composite type:

Listing 2.1: Bundle type for primal types with natural isomorphism

```julia
struct Taylor{T, N}
    value::NTuple{N, T}
end
```

where `NTuple` is the Julia `Tuple` type of several components in the same type, and tuple size `N` is the order of the Taylor series plus 1 ($N = n+1$). For example, the Taylor bundle for a scalar `Float64` with order 3 is represented as `Taylor{Float64, 4}`. While this form cannot represent the Taylor bundle for types that do not have a natural isomorphism, it is sufficient for many practical use cases. Therefore, in this work, we will focus on bundle types with the form shown in Listing 2.1.

These type constructs have complete inferred information about the type and order, therefore it automatically unwraps and unrolls to the corresponding original type at compile time, and therefore the bundle representation does not incur any runtime overhead. In contrast, TaylorSeries.jl[14] use `Vector{T}` to represent the Taylor bundle, which incurs a runtime overhead for indexing and heap allocation, and causes mutating array issues when composed with other packages like Zygote.jl. We will see the consequences of this design choice in chapter 4.

Furthermore, since ForwardDiff.jl also uses `NTuple` to represent dual numbers, it is expected that they will have similar performance at lower-order. Therefore, users won't be punished at lower order if they would like the benefits of higher-order automatic differentiation.

2.3 Pushforward rules

2.3.1 Requirements

For the purpose of this work, we will use an operator-overloading approach, although this may change in the future (see chapter 5). In Julia language terminology, for each manifold M represented by primal type `T` that has a method `f(x::T)` defined, we define a corresponding method for the bundle type `Taylor{T, n + 1}` that represents the bundle $x = (x_0, x_1, \cdots, x_n)$ for each order n, i.e. `f(x::Taylor{T, n + 1})`, and this method is called a **pushforward rule**. Similarly, for each pair of primal type `T, S` that has a method `f(x::T, y::S)` defined, we need to define methods accordingly. Since there is a natural projection from the bundle type to the primal type, for functions with more than one input, we only need to define the method where all inputs are bundle types, and promote the primal type to the bundle type when the inputs are a mixture of primal types and bundle types.

2.3.2 Manual implementation of basic pushforward rules

The pushforward rule for four basic arithmetic operations can be derived by making an analogy to the higher-order addition, subtraction, multiplication and division rules that are well-known in calculus:

$$\begin{aligned}
(f+g)^{(n)} &= f^{(n)} + g^{(n)} \\
(f-g)^{(n)} &= f^{(n)} - g^{(n)} \\
(f\times g)^{(n)} &= \sum_{k=0}^{n} \binom{n}{k} f^{(k)} g^{(n-k)} \\
\left(\frac{f}{g}\right)^{(n)} &= \frac{1}{g}\left[f^{(n)} - \sum_{k=0}^{n-1} \binom{n}{k} \left(\frac{f}{g}\right)^{(k)} g^{(n-k)}\right]
\end{aligned} \tag{2.9}$$

where $f^{(n)}$ is the n-th derivative of f. These rules need to be implemented once for the parameterized bundle type `Taylor{T, N}` and can be used for any type `T` that has corresponding arithmetic operations defined.

2.3.3 Automatic generation of other pushforward rules

In a comprehensive differentiable programming system, numerous rules (often called "primitives") are needed for efficient and flexible computation of derivatives, and it is often beyond the ability of the author of AD libraries to implement all of them. Many AD libraries have a mechanism for separating AD engine and AD rules, for example,

- In ForwardDiff.jl[15], the AD rules come from DiffRules.jl[34].
- In Zygote.jl[12], the AD rules come from ChainRules.jl [33], which is powered by ChainRulesCore.jl rule definition system[35]; in addition, it also has its own rule definition system ZygoteRules.jl[36].
- In Difffractor.jl[25], the AD engine is completely supported by ChainRules.jl.

With the presence of many well-written first-order AD rule sets, it seems redundant to implement the higher-order AD rules for other functions manually (as shown in TaylorSeries.jl[14]). Instead, it is desirable to have a mechanism to automatically generate higher-order AD rules for other functions from (1) well-written first-order AD rule sets and (2) a limited set of higher-order AD rules for basic arithmetic operations. In this section, we will show how to automatically generate pushforward rules for other functions from the pushforward rules for basic arithmetic operations.

To begin with, one can observe that the higher-order multiplication rule and division rule can be used to partially unroll the higher-order chain rule:

$$f(g(x))^{(n)} = [f'(g(x))g'(x)]^{(n-1)} = \sum_{i=0}^{n-1} \binom{n-1}{i} [f'(g(x))]^{(i)} g^{(n-i)}. \tag{2.10}$$

Therefore, we have a three-step approach to generate higher-order pushforward rules for f:

1. Obtain the form of f' from ChainRules.jl
2. Recurse to get the $[f'(g(x))]^{(i)}$ part, where i is the order of differentiation, up to $n-1$
3. Sum up via product rule as shown in equation 2.10

This recursion process doesn't seem to be efficient at first glance, but in most cases the form of f' is closely related to f itself, or can be built by simple arithmetic expression of the input variable. In either case, the recursion process can be done in a single pass by reusing the lower-order results and the complexity is at most quadratic to the order of differentiation n.

For example, consider the pushforward rule for exp function, which converts the bundle $x = (x_0, x_1, \cdots, x_n)$ to the bundle $y = (y_0, y_1, \cdots, y_n)$:

- $y_0 = \exp(x_0)$;
- $y_1 = \exp(x_0)x_1$;
- For each higher order, observe that in equation 2.10 the f' part is exp itself, therefore we can reuse the lower-order result $y_0, y_1, \cdots$ and get

$$y_i = \sum_{k=0}^{i-1} \binom{i-1}{k} y_k x_{i-k}.$$

Another example would be the pushforward rule for log function:

- $y_0 = \log(x_0)$;
- $y_1 = x_1/x_0$;
- For each higher order, observe that in equation 2.10 the f' part is $1/g$, therefore we can use the division rule and get

$$y_i = \frac{1}{x_0}\left(x_i - \sum_{k=0}^{i-1} \binom{i-1}{k} y_k x_{i-k}\right).$$

The complexity of both of these processes is clearly $O(n^2)$ from the description. While exponentiation and logarithm are extreme cases of reusing lower-order results, many other functions can be handled similarly. For example, the higher-order derivative of $\tan(g(x))$ can be related to the lower-order results of $1 + \tan^2(g(x))$, which in turn is related to itself via arithmetic operations; the higher-order derivative of $\arctan(g(x))$ can be related to the lower-order results of $1 + g^2(x)$, which in turn is related to $g(x)$ via arithmetic operations. While not proven for arbitrary functions, all functions in DiffRules.jl (which is a large set with almost all (100+) built-in scalar mathematical functions) can be handled either of these two ways with complexity $O(n^2)$. In the text below, we will call these functions to have **simple derivative expressions**.

In practice, we will use Symbolics.jl[37] to obtain the form of f' from ChainRules.jl and automatically discover its possibilities of reusing lower-order results. If the form of f' is not recognized, we will fall back to the recursion process described above.

2.4 Complexity analysis

We will now formalize the complexity analysis of the pushforward rules described in the previous section, and give proofs for linear and non-linear unary functions.

Theorem 1 *Assume we have a unary function f mapping from a differentiable manifold M to another manifold N, and they are programmatically represented by a function method of `f` with input type `T` and output type `S` respectively. We additionally require M and N to have natural isomorphism to their tangent spaces, so the data structure in listing 2.1 can be used. For a particular input `x` which is a value of type `T`, it takes X steps on a Turing machine to compute `f(x)`, then we have:*

- *If f is linear (as defined by the linear space underlying the manifold), there is an algorithm to implement the pushforward rule which, upon the input bundle containing `x`, finishes within $O(nX)$ steps;*
- *If f is non-linear and has a simple derivative expression, there is an algorithm to implement the pushforward rule which, upon the input bundle containing `x`, finishes within $O(n^2 + nX)$ steps.*

Proof 1 *We will first prove the theorem for linear functions, then for non-linear functions with simple derivative expressions.*

- *If f is linear, then guaranteed by the isomorphism, we only need to apply the function f on each component of the bundle type `Taylor{T, N}`, which takes*

$O(nX)$ steps. This result is a simple extension of the linearity of higher-order derivatives in a programmatic context.

- *If f is non-linear and has a simple derivative expression, then the pushforward rule can be implemented by reusing lower-order results as described in the previous section.*
  - *For each higher order, we need to construct f' from the simple expression, which, according to the complexity of first-order AD[38], takes $O(cX)$ steps, where c is a constant independent of n;*
  - *For each higher order, we need to reuse lower-order results and assemble them by the equation 2.10, which takes at most $O(n)$ steps;*

  The complexity of this process is $O(n^2 + nX)$.

In many applications, the complexity of the function to differentiate is dominated by linear operations (such as in neural networks)[26, 39, 40]; in addition, non-linear functions like exp often takes much more steps than simple addition and multiplication required by 2.10. Therefore, in practice the $O(nX)$ term always dominates the complexity of the pushforward rule, and the $O(n^2)$ term is negligible. It is not surprising to observe a linear scaling of the complexity with respect to the order of differentiation n in the following experiments in chapter 4.

Chapter 3

Implementation

3.1 General information

The algorithm is written as a Julia package, namely TaylorDiff.jl[41], in order to be easily accessible to the Julia community. The package is available on the Julia registry and can be installed by running the following command in the Julia REPL:

```
julia> ] add TaylorDiff
```

3.2 Type definition and type system integration

In addition to the core definition in listing 2.1, the package also defines a number of methods to organically integrate it into the type system, so that the type behaves like a number. For example, the type allows construction from primal type or (first-order) tangent bundle:

Listing 3.1: Constructors

```
"""
    Taylor{T, N}(x::T) where {T, N}

Construct a Taylor bundle with zeroth order coefficient.
"""
```

```julia
@generated function Taylor{T, N}(x::T) where {T, N}
    return quote
        $(Expr(:meta, :inline))
        Taylor((T(x), $(zeros(T, N - 1)...)))
    end
end

"""
    Taylor{T, N}(x::T, d::T) where {T, N}

Construct a Taylor bundle with zeroth and first order
coefficient, acting as a seed.
"""
@generated function Taylor{T, N}(x::T, d::T) where {T, N}
    return quote
        $(Expr(:meta, :inline))
        Taylor((T(x), T(d), $(zeros(T, N - 2)...)))
    end
end
```

Notice that generated functions[17] are used to allow the compiler to specialize the code for different types. This is necessary because the number of zeros to pad is not known until compile time. The type also has number-like conversions and promotions:

Listing 3.2: Conversions

```julia
function promote_rule(::Type{Taylor{T, N}},
    ::Type{S}) where {T, S, N}
    Taylor{promote_type(T, S), N}
end

```

```
convert(::Type{Taylor{T, N}}, x::Taylor{T, N})
    where {T, N} = x
convert(::Type{Taylor{T, N}}, x::S)
    where {T, S, N} = Taylor{T, N}(convert(T, x))
```

3.3 Handwritten rules

The package includes basic definitions of arithmetic operations as well as the recursion method described in 2.10 in order to serve as the basis of code generation. For example, the definition of multiplication and division is as follows:

Listing 3.3: Multiplication and division

```
@generated function *(a::Taylor{T, N}, b::Taylor{T, N})
    where {T, N}
    return quote
        va, vb = value(a), value(b)
        Taylor($([:(+($([:($(binomial(i - 1, j - 1))
        * va[$j] * vb[$(i + 1 - j)]) for j in 1:i]...)))
        for i in 1:N]...))
    end
end

@generated function /(a::Taylor{T, N}, b::Taylor{T, N})
    where {T, N}
    ex = quote
        va, vb = value(a), value(b)
        v1 = va[1] / vb[1]
    end
    for i in 2:N
        ex = quote
```

```
            $ex
            $(Symbol('v', i)) = (va[$i] -
            +($([:($(binomial(i - 1, j - 1)) * $(Symbol('v', j)) *
            vb[$(i + 1 - j)]) for j in 1:(i - 1)]...))) / vb[1]
        end
    end
    ex = :($ex; Taylor($([Symbol('v', i) for i in 1:N]...)))
    return :(@inbounds $ex)
end
```

3.4 Code generation

A comprehensive set of rules is generated using the code generation method described in chapter 2, with the help of Julia symbolics[37]. The code generation is done by the following function:

Listing 3.4: Code generation

```
for func in RULE_LIST
    F = typeof(func)
    # base case
    @eval function (op::$F)(t::Taylor{T, 2}) where {T}
        t0, t1 = value(t)
        Taylor{T, 2}(frule((NoTangent(), t1), op, t0))
    end
    der = frule(dummy, func, t)[2]
    term, raiser = der isa Pow && der.exp == -1 ?
        (der.base, raiseinv) : (der, raise)
    # recursion by raising
    @eval @generated function (op::$F)(t::Taylor{T, N})
        where {T, N}
        der_expr = $(QuoteNode(toexpr(term)))
        f = $func
        quote
            $(Expr(:meta, :inline))
            t = Taylor{T, N - 1}(t)
            df = $der_expr
            $$raiser($f(value(t)[1]), df, t)
        end
    end
end
```

The `raiser` function is an implementation of 2.10 to compute the Taylor bundle of order `N` from lower orders. During the compilation step, the expressions generated by the `@generated` function undergo a number of optimizations including common subexpression elimination, which reuses computation from lower orders, and therefore the scaling mentioned in theorem 1 is achieved.

3.5 Interface

The package also provides a number of convenience functions to allow easy use of the Taylor bundle type to compute derivatives. For example, the following functions compute either the derivative of a function at a point, or the directional derivative of a vector-input function at a point in a given direction:

Listing 3.5: Derivative

```
function derivative end

@inline function derivative(f, x::T,
    order::Int64) where {T <: Number}
    derivative(f, x, Val{order + 1}())
end
```

```julia

@inline function derivative(f, x::V, l::V,
    order::Int64) where {V <: AbstractVector{<:Number}}
    derivative(f, x, l, Val{order + 1}())
end

@inline function derivative(f, x::T,
    ::Val{N}) where {T <: Number, N}
    t = Taylor{T, N}(x, one(x))
    return extract_derivative(f(t), N)
end

make_taylor(t0::T, t1::T, ::Val{N}) where {T, N}
    = Taylor{T, N}(t0, t1)

@inline function derivative(f, x::V, l::V,
    vN::Val{N}) where {V <: AbstractVector{<:Number}, N}
    t = map((t0, t1) -> make_taylor(t0, t1, vN), x, l)
    return extract_derivative(f(t), N)
end
```

Note that the two of the interfaces nicely wrap around the underlying types that are specialized for each order, and instead allow an integer input to specify the order. This is possible because the compiler can infer the type of the input at compile time, and thus the order can be used to select the correct method. The function `extract_derivative` is used to extract the derivative from the Taylor bundle, and is defined as follows:

Listing 3.6: Extract derivative

```julia
extract_derivative(t::Taylor, i::Integer) = t.value[i]
```

3.6 Composability

Finally, the package is designed to be composable with other packages. For example, one should be able to compute several orders of derivative in forward mode using this package, and then compute an additional order of gradient in the reverse mode using packages like Zygote.jl[12] or Diffractor.jl[25]. This is achieved by making the types understandable by the `rrules` in ChainRules.jl, as shown in defining the following methods:

Listing 3.7: ChainRules

```
function rrule(::typeof(value), t::Taylor{T, N}) where {N, T}
    value_pullback(v::NTuple{N, T}) = NoTangent(), Taylor(v)
    # for structural tangent, convert to tuple
    function value_pullback(v::Tangent{P, NTuple{N, T}}) where {P}
        NoTangent(), Taylor{T, N}(backing(v))
    end
    value_pullback(v) = NoTangent(),
        Taylor{T, N}(map(x -> convert(T, x), Tuple(v)))
    return value(t), value_pullback
end

function rrule(::typeof(extract_derivative), t::Taylor{T, N},
               i::Integer) where {N, T <: Number}
    function extract_derivative_pullback(d)
        NoTangent(), Taylor{T, N}(ntuple(j -> j === i ?
            d : zero(T), Val(N))),
        NoTangent()
    end
    return extract_derivative(t, i), extract_derivative_pullback
end
```

Chapter 4

Results

4.1 Open source contributions

The code implementation described in chapter 3 is open-sourced on GitHub[41]. The code is available under the MIT license, and is free to use for both academic and commercial purposes. At the time of writing, the package has received 42 stars on GitHub, 36 issues and pull requests, and has been used in several other research projects and applications.

4.2 Correctness

The correctness of automatic differentiation can be tricky due to the complexity of supporting a variety of different programs. In the package, we use FiniteDifferences.jl[42], which is a Julia package to calculate derivatives with finite difference methods, to automatically verify the correctness of all primitives involved. The test suite is run on every pull request and commit to ensure that the package is always correct.

4.3 Performance results

Below we demonstrate the performance of the package on a variety of benchmarks. Similar to tests, some performance benchmarks are also automatically run on every pull request and commit to study the influence of various design choices. The continuous benchmarking results are available on a dedicated website[43].

All benchmarks are run with Julia 1.8.5, and the versions of third-party packages are detailed at GitHub repository. Continuous benchmarks are run exclusively on Julia community machine "AMDCI-3.0" with a 1.5 GHz AMD EPYC 7502 32-core processor and 512 GB of RAM, and other benchmarks are run on a 2018 MacBook Pro with a 2.6 GHz Intel Core i7 6-core processor and 16 GB of RAM. All benchmarks are run by the Julia standard benchmarking package BenchmarkTools.jl[44, 45] to average over multiple runs and reduce the influence of noise, as well as to exclude the time of compilation.

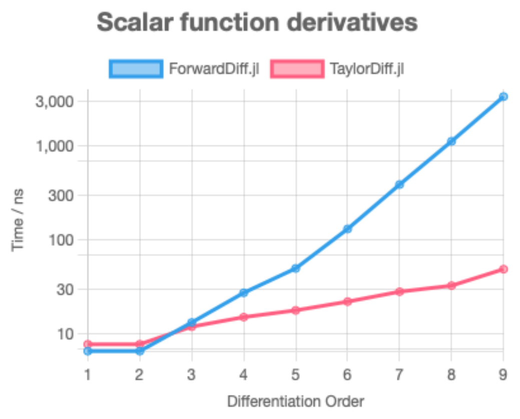
4.3.1 Scaling behavior

In figure 4-1, we demonstrate the scaling behavior of TaylorDiff.jl on two benchmarks. The first benchmark is the computation of higher-order derivatives of the sin function, and the second benchmark is the computation of the directional derivative of a multi-layer perceptron. The formula for the perceptron is:

$$f(x) = \sum_{i=1}^{n} W_i^{(2)} \sigma \left(\sum_{j=1}^{m} W_{ij}^{(1)} x_j + b_i^{(1)} \right) + b^{(2)}, \tag{4.1}$$

where σ is the sigmoid function, $W_{ij}^{(1)}$, $b_i^{(1)}$, $W_i^{(2)}$ and $b^{(2)}$ are random weights and biases of the perceptron.

In both cases, we compare the performance of TaylorDiff.jl with ForwardDiff.jl, which is the state-of-the-art and well-optimized forward-mode automatic differentiation package in Julia. We can see that ForwardDiff.jl demonstrates exponential scaling (a line in the log-scaled y-axis) in the sin example, while TaylorDiff.jl has mostly linear scaling. Similarly, in the multi-layer perceptron example, ForwardDiff.jl has exponential scaling, while TaylorDiff.jl grows much slower. This is due to the fundamental capability of the Taylor mode of automatic differentiation in computing higher-order derivatives. We note that while ForwardDiff.jl is well-optimized, the performance of TaylorDiff.jl still has a lot of room to improve due to a lack of detailed optimizations, so we can expect the advantage to grow with implementation refinements. We leave this as future work.



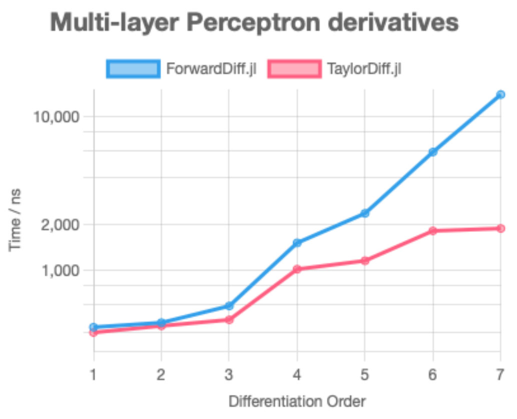


Figure 4-1: From the continuous benchmark set. Scaling behavior of TaylorDiff.jl with comparison to ForwardDiff.jl. Left: the time to compute the higher-order derivative of sin function with respect to the order of derivatives. Right: the time to compute the gradient of a multi-layer perceptron with respect to the order of the derivatives.

More noticeably, TaylorDiff.jl has demonstrated comparable performance with ForwardDiff.jl at lower orders (first and second order), ensuring that there is no "penalty of abstraction", allowing drop-in replacement. This is a significant advantage over other packages like TaylorSeries.jl, which often have a much higher overhead.

4.3.2 Comparison with existing higher-order automatic differentiation packages

In table 4.1, we compare the performance of TaylorDiff.jl with existing higher-order automatic differentiation packages like TaylorSeries.jl. We examine their capabilities to compute Taylor expansions to a relatively high order (20 is used in these tests), and compare their performance in terms of time, heap memory allocation amount and number of allocations. In each case, Taylor expansion is computed on a real function in the neighbor of 0, where the input variable is represented by `Float64` in Julia. To compute the expansion, the input Taylor bundle is constructed as `Taylor1(Float64, 20)` in TaylorSeries.jl and `Taylor{Float64, 21}(0.0, 1.0)` in

TaylorDiff.jl (the `N` in the type definition 2.1 is the length of the tuple, which is 1 plus the order). The functions to expand are a mixture of some common functions that have meaningful Taylor series and some real use cases collected from Julia community (provided in the appendix A).

Table 4.1: Comparison of TaylorDiff.jl with TaylorSeries.jl on a variety of benchmarks, in terms of time, memory allocation numbers, and memory usage.

Test case	TaylorDiff.jl			TaylorSeries.jl		
	time (μs)	alloc (KiB)	number	time (μs)	alloc (KiB)	number
sin	0.145	0.176	1	0.708	0.704	4
arcsin	0.594	0.176	1	10.4	24.1	144
log1p	0.233	0.176	1	0.665	1.23	9
Bell	0.262	0.528	3	0.928	1.70	12
$\sqrt{1+x^2}$	0.394	0.528	3	0.931	1.70	12
case1	3.55	0.176	1	8.86	23.0	106
case2	27.2	0	0	95.7	92.0	426

While TaylorSeries.jl achieved AD via dynamic mutating array-style code, TaylorDiff.jl achieves fully static and non-allocating code by exploiting the structure of Taylor polynomials and using advanced techniques in computational science, such as aggressive type specializing and metaprogramming. This results in significant improvements in memory allocation and speed. We also notice that TaylorSeries.jl expands arcsin much slower than sin, which is due to the fact that it has a handwritten rule for sin but fallback to recursion for arcsin. In the case of TaylorDiff.jl, the expansion of arcsin is on the same order of magnitude as sin, since the auto-generated higher-order rule for arcsin has little overhead.

4.3.3 Neural PDE case study

Furthermore, this project has significant potential applications in scientific models where higher-order derivatives are required to be calculated efficiently, such as solving ODEs and PDEs with neural functions, which is also known as physics-informed neural networks (PINNs)[46]. The efficient computation of higher-order derivatives enables faster and more accurate optimization of these models, when compared to numerical finite differences that are currently used in such applications, leading to more reliable predictions and better scientific insights.

As a case study, we demonstrate the performance of TaylorDiff.jl on a PINN model for solving Poisson's equation, which is taken from NeuralPDE.jl tutorial. The Poisson's equation is a fundamental model in physics, and it is defined as:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -\sin(\pi x)\sin(\pi y), \tag{4.2}$$

where u is the physical quantity over a space domain $x \in [0,1], y \in [0,1]$, x and y are the positions. The boundary conditions are:

$$\begin{aligned} u(x,0) &= 0 \\ u(x,1) &= 0 \\ u(0,y) &= 0 \\ u(1,y) &= 0. \end{aligned} \tag{4.3}$$

The PINN model is defined as:

$$\varphi(x,y) = x(1-x)y(1-y)\mathrm{NN}(x,y), \tag{4.4}$$

where $x(1-x)y(1-y)$ is the part to ensure the boundary condition, and NN is the neural network function with 2 inputs and 1 output that is later trained to approximate the Poisson's equation. The neural network function NN is built with Flux.jl[47, 48], defined as a multi-layer perceptron with 2 hidden layers, and the activation function is exp, and the number of neurons in each layer is 16. The training is done by minimizing the loss function:

$$\mathcal{L} = \sum_{i=1}^{N} \left(\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} + \sin(\pi x)\sin(\pi y) \right)^2, \tag{4.5}$$

where N is the number of training data points randomly sampled from the domain $x \in [0,1], y \in [0,1]$.

The numerical way to compute the derivative in the loss function is:

$$\frac{\partial^2\varphi}{\partial x^2}+\frac{\partial^2\varphi}{\partial y^2}=\frac{\varphi(x+h,y)+\varphi(x-h,y)+\varphi(x,y+h)+\varphi(x,y-h)-4\varphi(x,y)}{h^2}, \tag{4.6}$$

where h is the step size chosen as 0.001. The automatic differentiation way to compute the derivative in TaylorDiff.jl is to utilize the directional derivative interface defined in listing 3.5.

The loss function is minimized using the stochastic gradient descent optimizer with the gradient of neural network parameters calculated by Zygote.jl[36]. The training is done in single precision floating point number (`Float32` in Julia) for 100 epochs with a learning rate of 0.001. We note that in this case we don't include TaylorSeries.jl for comparison because its implementation is based on mutating arrays, which cannot be differentiated by Zygote.jl.

Based on a random sample of training data points, we obtain the performance of forward evaluation and gradient evaluation of the loss function with finite differences methods and TaylorDiff.jl in figure 4-2.

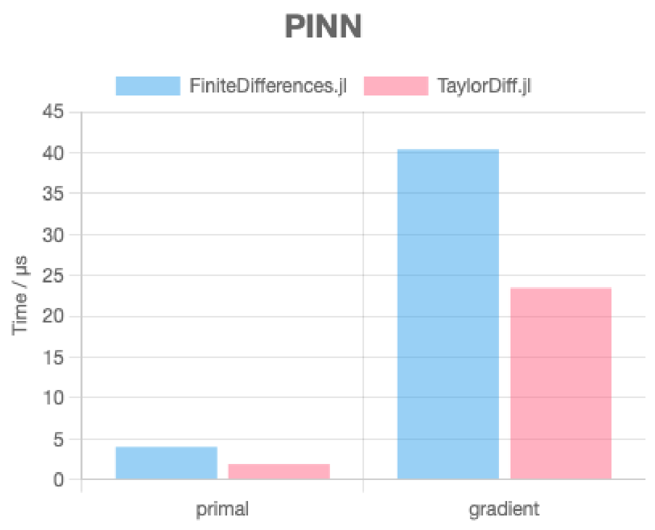


Figure 4-2: From the continuous benchmark set. The averaged computation time of the loss function on a training data point with finite differences methods and automatic differentiation methods with TaylorDiff.jl. Left: forward evaluation. Right: gradient evaluation.

We can see that the forward evaluation is 2.1 times faster, and the gradient evaluation is 1.7 times faster. The performance improvement is significant, and it is expected to be more significant when the model is more complex. We note that TaylorDiff.jl has a lot of room to improve, as it is still in the early stage of development, and we expect the advantage to grow with implementation refinements.

Moreover, while the accuracy of the finite differences method is generally enough, we observed a significant improvement in terms of smooth training curve when using TaylorDiff.jl, which is shown in figure 4-3. This is due to the more accurate derivatives computed by automatic differentiation algorithms, and may lead to faster convergence and better generalization of the model in more complicated cases.

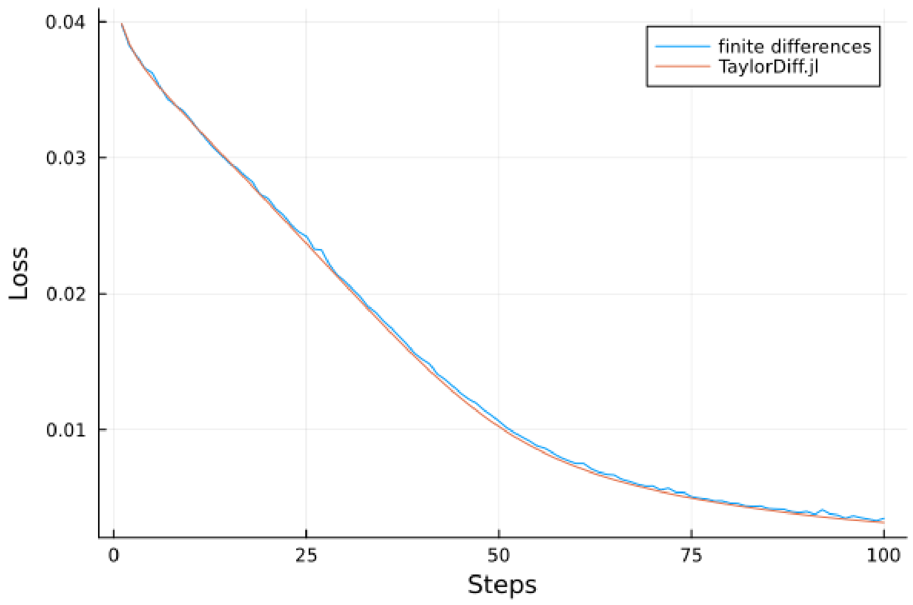


Figure 4-3: Averaged loss function in the training progress, on a training data point with finite differences methods and automatic differentiation methods with TaylorDiff.jl.

In summary, in the neural PDE case study, we demonstrate that TaylorDiff.jl can be used to efficiently compute higher-order derivatives in scientific models, and it has significant potential applications in scientific models where higher-order derivatives are required to be calculated efficiently, such as solving ODEs and PDEs with neural functions.

Chapter 5

Conclusion

5.1 Summary

Overall, this work on higher-order differentiation algorithms represents a contribution to the field of automatic differentiation. Through the development of advanced algorithms and techniques, I have overcome the challenges associated with higher-order automatic differentiation and developed a package that is efficient, accurate, and easy to use. The key contributions of my research are as follows:

- Taylor-mode automatic differentiation algorithms which can compute the n-th order derivative of most functions in effectively $O(n)$ time;
- Comparable performance with ForwardDiff.jl at lower order in simple scalar and vector routines;
- Greatly improved performance over TaylorSeries.jl for various higher-order computations, in terms of both time and memory;
- Generated higher-order rules from first-order rule sets in ChainRules.jl and a very limited set of handwritten higher-order rules;
- Composable with other packages in the Julia ecosystem, including Zygote.jl and Flux.jl, demonstrated by the use of TaylorDiff.jl in a neural PDE training example;

5.2 Future work

I am excited about the potential impact of TaylorDiff.jl on the Julia community and look forward to further developing the package in the future. As the package is still in the early alpha stage, there is a lot of room for growth and potential improvements, and I am going to continue working on this in my future research. Some areas for future work include:

- Improving the performance of the package by optimizing the generated code and the implementation of the algorithms;
- Integrate TaylorDiff.jl with the new machine learning framework Lux.jl[49], which is designed to be a more explicitly parameterized and performant alternative to Flux.jl;
- Developing a more robust and efficient algorithm for source-code transformation-based higher-order automatic differentiation;
- Integrate this package as a general differentiation method in NeuralPDE.jl, allowing compilation of symbolic user-defined models to calls to TaylorDiff.jl;

Additionally, it would be valuable to explore potential applications of TaylorDiff.jl in other areas of computational science and engineering, such as financial modeling and operational research.

Appendix A

Benchmarks used to compare TaylorSeries.jl and TaylorDiff.jl

Case 1

This case solves the Liouville equation $y' = x^2 + y^2$ using Picard integration method.

```
using TaylorSeries, TaylorDiff
using BenchmarkTools
import TaylorSeries: integrate

@generated function integrate(t::TaylorScalar{T, N}) where {T, N}
    return quote
        $(Expr(:meta, :inline))
        v = TaylorDiff.value(t)
        TaylorScalar((zero(T), $([:(v[$i] / $i)
            for i in 1:N-1]...)))
    end
end

function taylor_step(f, u0, t)
```

```julia
    u = u0
    unew = u0 + integrate(f(u, t))
    for i = 1:20
        u = unew
        unew = u0 + integrate(f(u, t))
    end
    return u
end

ts = Taylor1([1.0], 20)
td = TaylorScalar{Float64, 21}(1.0)

f(x, t) = x^2 + t^2

@btime taylor_step(f, ts, 1.)
@btime taylor_step(f, td, 1.)
```

Case 2

This case calculates the Taylor expansion of a polynomial after a change of variable.

```julia
using TaylorSeries, TaylorDiff
using BenchmarkTools

function my_calculation(t, p, alpha, s)
    x = 1.0 / (1.0 - s * (t + 1) / (t - 1))
    rez = zero(x)
    for i in eachindex(p)
        rez += p[i] * x^alpha[i]
    end
    return rez * sqrt(2) / (1 - t)
end

N, m = 100, 20
p, alpha, s = rand(N), rand(N), rand()
p ./= sum(p)
t_ts = Taylor1(eltype(p), m)
t_td = TaylorScalar{eltype(p), m + 1}(0.0, 1.0)

@btime my_calculation($t_ts, $p, $alpha, $s)
@btime my_calculation($t_td, $p, $alpha, $s)
```

Bibliography

[1] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B. Shah, and Will Tebbutt. A Differentiable Programming System to Bridge Machine Learning and Scientific Computing, July 2019. arXiv:1907.07587 [cs].

[2] Christopher M. Bishop and Nasser M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[3] Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 9(4):e1305, 2019. Publisher: Wiley Online Library.

[4] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[6] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018. Publisher: Microtome Publishing.

[7] Jarrett Revels, Tim Besard, Valentin Churavy, Bjorn De Sutter, and Juan Pablo Vielma. Dynamic automatic differentiation of GPU broadcast kernels. *arXiv preprint arXiv:1810.08297*, 2018.

[8] Suyong Kim, Weiqi Ji, Sili Deng, Yingbo Ma, and Christopher Rackauckas. Stiff neural ordinary differential equations. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 31(9):093122, 2021. Publisher: AIP Publishing LLC.

[9] Christopher Rackauckas and Qing Nie. Differentialequations. jla performant and feature-rich ecosystem for solving differential equations in julia. *Journal of open research software*, 5(1), 2017. Publisher: Ubiquity Press.

[10] Jesse Bettencourt, Matthew J. Johnson, and David Duvenaud. Taylor-Mode Automatic Differentiation for Higher-Order Derivatives in JAX. July 2022.

[11] Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew Fitzgibbon. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–30, 2022. Publisher: ACM New York, NY, USA.

[12] Michael Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs, March 2019. arXiv:1810.07951 [cs].

[13] Barnaba Tortolini. *Annali di scienze matematiche e fisiche*. Tip. delle bella arti, 1855. Google-Books-ID: ddE3AAAAMAAJ.

[14] Luis Benet and David Sanders. TaylorSeries.jl: Taylor expansions in one and several variables in Julia. *Journal of Open Source Software*, 4(36):1043, April 2019.

[15] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv preprint arXiv:1607.07892*, 2016.

[16] Laurent Hascoet and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):1–43, 2013. Publisher: ACM New York, NY, USA.

[17] Jeffrey Werner Bezanson. *Abstraction in technical computing.* Thesis, Massachusetts Institute of Technology, 2015. Accepted: 2015-11-09T19:50:22Z.

[18] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017. Publisher: Society for Industrial and Applied Mathematics.

[19] Jeff Bezanson, Jake Bolewski, and Jiahao Chen. Fast flexible function dispatch in Julia. *arXiv preprint arXiv:1808.03370*, 2018.

[20] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, 132:29–46, June 2019.

[21] Jarrett Revels. ReverseDiff.jl - Reverse Mode Automatic Differentiation for Julia, May 2023. original-date: 2016-06-24T17:56:47Z.

[22] William Moses and Valentin Churavy. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020.

[23] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. Reverse-mode automatic differentiation and optimization of GPU kernels via Enzyme. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–16, 2021.

[24] Frank Schäfer, Mohamed Tarek, Lyndon White, and Chris Rackauckas. AbstractDifferentiation.jl: Backend-Agnostic Differentiable Programming in Julia, February 2022. arXiv:2109.12449 [cs].

[25] Keno Fischer. Diffractor.jl - Next Generation AD, May 2023. original-date: 2021-05-13T14:28:09Z.

[26] Kirill Zubov, Zoe McCarthy, Yingbo Ma, Francesco Calisto, Valerio Pagliarino, Simone Azeglio, Luca Bottero, Emmanuel Luján, Valentin Sulzer, Ashutosh Bharambe, Nand Vinchhi, Kaushik Balakrishnan, Devesh Upadhyay, and Chris Rackauckas. NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations, July 2021. arXiv:2107.09443 [cs].

[27] Keno Fischer. P .jl terminology and notation guide. Technical report, 2020.

[28] Mathieu Huot, Sam Staton, and Matthijs Vákár. Higher Order Automatic Differentiation of Higher Order Functions. *Logical Methods in Computer Science*, Volume 18, Issue 1:7106, March 2022.

[29] Michael Betancourt. A Geometric Theory of Higher-Order Automatic Differentiation, December 2018. arXiv:1812.11592 [stat] version: 1.

[30] Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. A general construction for abstract interpretation of higher-order automatic differentiation. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1007–1035, 2022. Publisher: ACM New York, NY, USA.

[31] Martin Berz. Algorithms for higher order automatic differentiation in many variables with applications to beam physics. *Automatic Differentiation of Algorithms: Theory, Implementation and Application, A. Griewank and G. F. Corliss Eds*, page 147, 1991. Publisher: Citeseer.

[32] Jason Abate, Christian Bischof, Lucas Roh, and Alan Carle. Algorithms and design for a second-order automatic differentiation module. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 149–155, 1997.

[33] Frames White, Michael Abbott, Miha Zgubic, Jarrett Revels, Seth Axen, Alex Arslan, Simeon Schaub, Nick Robinson, Yingbo Ma, Gaurav Dhingra, Will Tebbutt, David Widmann, Niklas Heim, Niklas Schmitz, Christopher Rackauckas, Carlo Lucibello, Rainer Heintzmann, frankschae, Andreas Noack, Keno Fischer, Alex Robson, cossio, Jerry Ling, mattBrzezinski, Rory Finnegan, Andrei Zhabinski, Daniel Wennberg, Mathieu Besançon, and Pietro Vertechi. JuliaDiff/ChainRules.jl: v1.49.0, April 2023.

[34] DiffRules, March 2023. original-date: 2017-08-23T23:58:49Z.

[35] Frames White, Miha Zgubic, Michael Abbott, Jarrett Revels, Nick Robinson, Alex Arslan, David Widmann, Simeon Schaub, Yingbo Ma, Will Tebbutt, Seth Axen, Pietro Vertechi, Christopher Rackauckas, Keno Fischer, BSnelling, st, Ben Cottier, Hendrik Ranocha, Brian Chen, Jutho, Marco, Mathieu Besançon, Niklas Schmitz, Curtis Vogt, Fernando Chorney, Gaurav Dhingra, Guillaume Dalle, James Bradbury, Jeffrey Sarnoff, and Julia TagBot. JuliaDiff/ChainRulesCore.jl: v1.16.0, May 2023.

[36] ZygoteRules.jl, February 2023. original-date: 2019-07-30T14:47:29Z.

[37] Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwózd, Viral B. Shah, Alan Edelman, and Christopher Rackauckas. High-performance symbolic-numerics via multiple dispatch. *ACM Communications in Computer Algebra*, 55(3):92–96, September 2021.

[38] Andreas Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, 12:321–398, 2003. Publisher: Cambridge University Press.

[39] Chris Rackauckas, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. Diffeqflux. jl-A julia library for neural differential equations. *arXiv preprint arXiv:1902.02376*, 2019.

[40] Chris Rackauckas, Maja Gwozdz, Anand Jain, Yingbo Ma, Francesco Martinuzzi, Utkarsh Rajput, Elliot Saba, Viral B. Shah, Ranjan Anantharaman, Alan Edelman, Shashi Gowda, Avik Pal, and Chris Laughman. Composing Modeling And Simulation With Machine Learning In Julia. In *2022 Annual Modeling and Simulation Conference (ANNSIM)*, pages 1–17, July 2022.

[41] Songchen Tan. TaylorDiff.jl, May 2023. original-date: 2022-11-09T17:21:07Z.

[42] Frames Catherine White, Will Tebbutt, Miha Zgubic, Wessel, Rogerluo, Nick Robinson, Alex Arslan, Seth Axen, Simeon Schaub, David Widmann, Rory Finnegan, Alex Robson, Benoît Richard, Curtis Vogt, Daniel Karrasch, Eric Davies, José E. C. Serrallés, Julia TagBot, and Viral B. Shah. JuliaDiff/FiniteDifferences.jl: v0.12.26, January 2023.

[43] Songchen Tan. Continuous Benchmarking, January 2023.

[44] BenchmarkTools.jl, May 2023. original-date: 2016-02-23T20:54:02Z.

[45] PkgBenchmark, May 2023. original-date: 2016-09-14T06:50:17Z.

[46] C. Rackauckas, A. Edelman, K. Fischer, M. Innes, E. Saba, V. B. Shah, and W. Tebbutt. Generalized physics-informed learning through language-wide differentiable programming. *MIT web domain*, November 2021. Accepted: 2021-11-04T11:58:19Z.

[47] Mike Innes. Flux: Elegant machine learning with Julia. *Journal of Open Source Software*, 3(25):602, May 2018.

[48] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable Modelling with Flux, November 2018. arXiv:1811.01457 [cs].

[49] Avik Pal. Lux: Explicit Parameterization of Deep Neural Networks in Julia, May 2022. original-date: 2022-03-21T04:53:14Z.