Implementing Automatic Differentiation Using Dual Numbers

Shahab Yousef-Nasiri (sy475)
Department of Physics, University of Cambridge
Word Count: 2881

Abstract

Automatic differentiation (AD) is a computational technique enabling efficient and precise derivative computation, critical in scientific computing, optimization, and machine learning. This project implements and optimizes forward-mode AD using dual numbers, embedding derivative computation directly into arithmetic operations. A derivative error analysis was conducted, comparing numerical methods, analytical derivatives, and dual numbers differentiation. The dual numbers approach demonstrated machine-level precision, independent of step size, whereas numerical methods exhibited inferior performance due to truncation errors in larger step-size regimes and rounding errors in smaller step-size regimes. Additionally, we developed and bench marked a pure Python implementation and a performance-optimized Cython version of the dual number package and tested performance across arithmetic operations, trigonometric and hyperbolic functions, as well as derivatives of various machine learning functions. Results highlight significant computational efficiency with the Cython implementation, particularly for large-scale datasets.

Contents

1 Introduction

The computation of derivatives is a cornerstone of many fields, including scientific computing, machine learning, and optimization. Gradients are essential for tasks ranging from minimizing loss functions in neural networks to sensitivity analysis in financial models. Traditional methods for derivative computation include symbolic differentiation, which provides exact results but can suffer from computational inefficiency and expression blow-up, and numerical differentiation, which introduces truncation and rounding errors due to finite step sizes. Automatic Differentiation (AD) addresses these limitations by combining the precision of symbolic methods with the efficiency of numerical approaches, enabling the evaluation of exact derivatives to machine precision. Unlike numerical methods, AD avoids step-size errors, and unlike symbolic methods, it scales efficiently to large-scale, complex computations.

2 Theory: Dual Numbers and Automatic Differentiation

2.1 Dual Numbers

2.1.1 Introduction to Dual Numbers

Dual numbers are an extension of real numbers, similar to how complex numbers extend the real numbers [1]. A dual number can be written as:

$$x + \epsilon y, \quad \text{where } x, y \in \mathbb{R}, \quad \epsilon^2 = 0 \text{ (but } \epsilon \neq 0).$$

Here, x is the real part, and y is the dual part. The property $\epsilon^2 = 0$ makes dual numbers particularly useful for evaluating derivatives.

2.1.2 Basic Arithmetic with Dual Numbers

Let $a = x_1 + \epsilon y_1$ and $b = x_2 + \epsilon y_2$ be two dual numbers. The arithmetic operations are defined as follows:

1. **Addition**:
$$a + b = (x_1 + x_2) + \epsilon(y_1 + y_2)$$
2. **Multiplication**:
$$a \cdot b = (x_1 x_2) + \epsilon(x_1 y_2 + x_2 y_1)$$
3. **Division**:
$$\frac{a}{b} = \frac{x_1}{x_2} + \epsilon \frac{y_1 x_2 - x_1 y_2}{x_2^2}, \quad \text{for } x_2 \neq 0.$$
4. **Trigonometric Functions**: Using the Taylor expansion:
$$\sin(a) = \sin(x) + \epsilon y \cos(x), \quad \cos(a) = \cos(x) - \epsilon y \sin(x).$$

2.1.3 Derivatives Using Dual Numbers

Dual numbers provide a simple way to compute derivatives. If $f(x)$ is a differentiable function, we evaluate it at $x + \epsilon$ to obtain:

$$f(x + \epsilon) = f(x) + \epsilon f'(x). \tag{1}$$

Thus, the dual part directly encodes the derivative $f'(x)$.

2.2 Forward-Mode Automatic Differentiation

2.2.1 What is Automatic Differentiation?

Automatic Differentiation (AD) is a computational technique for efficiently and accurately evaluating the derivatives of mathematical functions [2]. This process involves constructing and traversing a computational graph that represents the sequence of operations and dependencies within the function. Each node u_i in the computational graph corresponds to an intermediate variable or operation (e.g. addition, multiplication, or a trigonometric function), while edges capture the flow of information and dependencies between nodes. Each node is annotated with:

$$\text{Value: } u_i, \quad \text{Derivative: } \dot{u}_i.$$

In the context of a computational graph with intermediate variables $u_1, u_2, \ldots, u_m$, where each u_j depends on preceding variables, the derivative of $f : \mathbb{R}^n \to \mathbb{R}$ with respect to an input x_i is propagated as:

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial u_j} \cdot \frac{\partial u_j}{\partial x_i}. \tag{2}$$

Here, $\frac{\partial u_j}{\partial x_i}$ is determined by the specific operation defining u_j, and the computation proceeds sequentially through the graph. This ensures that each intermediate variable is evaluated exactly once.

2.2.2 Using Dual Numbers for Forward-Mode AD

Dual numbers play a central role in forward-mode Automatic Differentiation (AD) because they enable the simultaneous computation of function values and derivatives. By encoding derivatives into dual numbers as a first-order perturbation, forward-mode AD eliminates the need for symbolic differentiation or finite difference approximations, achieving high efficiency and machine-level precision.

Propagation via Dual Arithmetic The forward-mode AD process systematically applies the chain rule by leveraging the arithmetic properties of dual numbers. Consider a composite function $f(x) = g(h(x))$. At each step:

$$f(x + \epsilon) = g(h(x + \epsilon)) = g(h(x) + \epsilon h'(x)) = g(h(x)) + \epsilon g'(h(x))h'(x). \tag{3}$$

This approach is particularly efficient for functions with a small number of inputs relative to outputs. Each input variable independently carries its derivative through the computation, making the method well-suited for scalar functions or low-dimensional problems.

3 Methodology

This section details the implementation of the dual number system through the `Dual` class, its supported mathematical functions, and the derivative computation framework enabled by the `differentiate` function. Additionally, we highlight key utility features and error-handling mechanisms integrated into the class design.

3.1 Mathematical Functions in the `Dual` Class

The `Dual` class supports a wide range of mathematical operations, including arithmetic, trigonometric, hyperbolic, logarithmic, and exponential functions. These operations are implemented using operator overloading and custom methods, enabling the application of standard mathematical operations to dual numbers. Table 1 summarizes the functions supported by the `Dual` class.

Category	**Function**	**Behavior for Dual Numbers**
Arithmetic Operations	Addition (`+`)	Adds real and dual parts.
	Subtraction (`-`)	Subtracts real and dual parts.
	Multiplication (`*`)	Multiplies real parts and applies the product rule to dual parts.
	Division (`/`)	Divides real parts and applies the quotient rule for dual parts.
	Power (`^`)	Raises to a power using the generalized chain rule.
Trigonometric Functions	Sine (`sin`)	Computes $\sin(x)$ and propagates derivative $\cos(x)$.
	Cosine (`cos`)	Computes $\cos(x)$ and propagates derivative $-\sin(x)$.
	Tangent (`tan`)	Computes $\tan(x)$ and propagates derivative $(1+\tan^2(x))$.
Hyperbolic Functions	Sinh (`sinh`)	Computes $\sinh(x)$ and propagates derivative $\cosh(x)$.
	Cosh (`cosh`)	Computes $\cosh(x)$ and propagates derivative $\sinh(x)$.
	Tanh (`tanh`)	Computes $\tanh(x)$ and propagates derivative $(1-\tanh^2(x))$.
	Inverse Sinh (`asinh`)	Computes $\sinh^{-1}(x)$ and propagates derivative $\frac{1}{\sqrt{x^2+1}}$.
	Inverse Cosh (`acosh`)	Computes $\cosh^{-1}(x)$ and propagates derivative $\frac{1}{\sqrt{x^2-1}}$.
	Inverse Tanh (`atanh`)	Computes $\tanh^{-1}(x)$ and propagates derivative $\frac{1}{1-x^2}$.
Logarithmic/Exponential Functions	Exponential (`exp`)	Computes $\exp(x)$ and propagates derivative $\exp(x)$.
	Logarithm (`log`)	Computes $\log(x)$ and propagates derivative $\frac{1}{x}$.
	Square Root (`sqrt`)	Computes $\sqrt{x}$ and propagates derivative $\frac{1}{2\sqrt{x}}$.

Table 1: Summary of arithmetic, trigonometric, hyperbolic, and logarithmic/exponential functions supported by the `Dual` class.

Each of these functions is implemented to return a new `Dual` object, ensuring that operations maintain dual arithmetic consistency.

3.2 Derivative Computation via the `differentiate` Function

The `differentiate` function enables computation of derivatives using dual numbers. The workflow is as follows:

1. Convert the input x into a dual number, initializing the dual part as 1. Mathematically, this represents $x = a + \epsilon$.

2. Evaluate the user-defined function $f(x)$ using the dual number x, which automatically propagates derivatives through the operations in $f(x)$.
3. Extract the derivative from the dual part of the resulting dual number.

Example Usage:

```
def func(x):
    return x ** 2 + x * 3

result = differentiate(func, 2)
# result = 7, as the derivative of f(x) = x^2 + 3x at x = 2 is 7.
```

3.3 Utility Features and Error Handling

The `Dual` class includes utility functions and error-handling mechanisms to enhance usability and prevent invalid operations.

Utility Functions

- `__repr__` and `__str__`: Provide string representations for debugging and display.
- `to_dict` and `from_dict`: Serialize and deserialize dual numbers for storage or communication.
- `abs`, `conjugate`, and rounding methods (`ceil`, `floor`, `__round__`): Extend the functionality of dual numbers to common operations.

Error Handling

- Division by zero is explicitly checked, raising `ZeroDivisionError`.
- Domain violations (e.g., $\log(x)$ for $x \leq 0$) raise descriptive `ValueError` exceptions.
- Inputs are validated to ensure they are numeric, raising `TypeError` if invalid types are passed.

4 Derivative Error Analysis

This section evaluates the errors in numerical differentiation, focusing on truncation and floating-point errors, and compares these with the precision achieved using dual numbers. For our analysis, we compute the derivative of the function:

$$f(x) = \ln(\sin(x)) + x^2 \cos(x),$$

over the domain $x \in [0.5, 2.2]$. The methods compared include numerical differentiation using the central difference formula and dual number differentiation.

4.1 Numerical Differentiation and Error Sources

4.1.1 Truncation Error

The central difference formula for approximating the first derivative is given by [3]:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

This formula is derived from the Taylor series expansion of $f(x+h)$ and $f(x-h)$ about x. Expanding $f(x+h)$ and $f(x-h)$ up to higher-order terms, we have:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + \cdots,$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) - \cdots .$$

Subtracting these expansions, we get:

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2h^3}{6}f'''(x) + \frac{2h^5}{120}f^{(5)}(x) + \cdots .$$

Dividing by $2h$, the central difference approximation becomes:

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{6}f'''(x) + \frac{h^4}{120}f^{(5)}(x) + \cdots .$$

The truncation error is the difference between the true derivative $f'(x)$ and the approximation:

$$\text{Truncation Error} = \frac{h^2}{6}f'''(x) + \frac{h^4}{120}f^{(5)}(x) + \cdots .$$

For sufficiently small h, the leading-order term dominates:

$$\text{Truncation Error} \approx \frac{h^2}{6}f'''(x). \tag{4}$$

Thus, the truncation error scales quadratically with h, ($\mathcal{O}(h^2)$), and is directly proportional to the magnitude of the third derivative $f'''(x)$.

4.1.2 Floating-Point Error

Floating-point error arises from the finite precision of numerical computations. In the central difference formula, the subtraction of nearly equal terms $f(x+h)$ and $f(x-h)$ leads to a loss of significant digits, amplifying rounding errors. Let $\epsilon_{\text{machine}}$ denote the machine epsilon, the smallest representable difference between two floating-point numbers [4]. The floating-point error scales as:

$$\text{Floating-Point Error} \sim \frac{\epsilon_{\text{machine}}}{h}. \tag{5}$$

4.1.3 Total Error Behavior

The total error in numerical differentiation is the sum of truncation and floating-point errors:

$$\text{Total Error} = \text{Truncation Error} + \text{Floating-Point Error}.$$

The truncation error decreases as $\mathcal{O}(h^2)$, while the floating-point error increases as $\mathcal{O}(1/h)$. This results in a U-shaped error curve as a function of h, with an optimal step size h_{optimal} that minimizes the total error.

4.2 Comparison of Derivative Errors: Numerical vs Dual Numbers

Figure 1 presents an analysis of the absolute errors of derivatives computed numerically and using dual numbers. For numerical differentiation, errors are plotted for various step sizes h, showcasing distinct behaviors in the truncation and floating-point precision regimes. In contrast, dual number differentiation achieves machine-level precision, as its errors are independent of h.

In the **truncation error regime** ($h \gtrsim 10^{-5}$), the errors are dominated by truncation effects, which scale quadratically with h, as derived earlier in Equation (4). Specifically, these errors are proportional to higher-order derivatives of the function, primarily the third derivative $f^{(3)}(x)$. Dips in the truncation error occur where $f^{(3)}(x)$ or higher-order derivatives like $f^{(5)}(x)$ approach zero, creating characteristic troughs. These troughs are confirmed by Figure 2, which shows the alignment of numerical error dips with the zero crossings or local minima of the third and fifth derivatives.

As h decreases further ($10^{-11} \lesssim h \lesssim 10^{-5}$), the **floating-point precision regime** begins to dominate. In this regime, errors increase inversely with h, as shown in Equation (5). This behavior arises due to round-off errors, which are a consequence of the finite precision of floating-point arithmetic. The subtraction $f(x+h) - f(x-h)$ in the central difference formula involves values that are increasingly close in magnitude as h becomes smaller. This subtractive cancellation amplifies numerical imprecision, leading to a loss of significant digits.

Furthermore, the division by $2h$ in the central difference formula exacerbates this effect, as smaller h magnifies the noise introduced by the subtraction step. This highlights a fundamental limitation of numerical differentiation: as h becomes exceedingly small, round-off errors render the method unreliable regardless of the smoothness or structure of the underlying function.

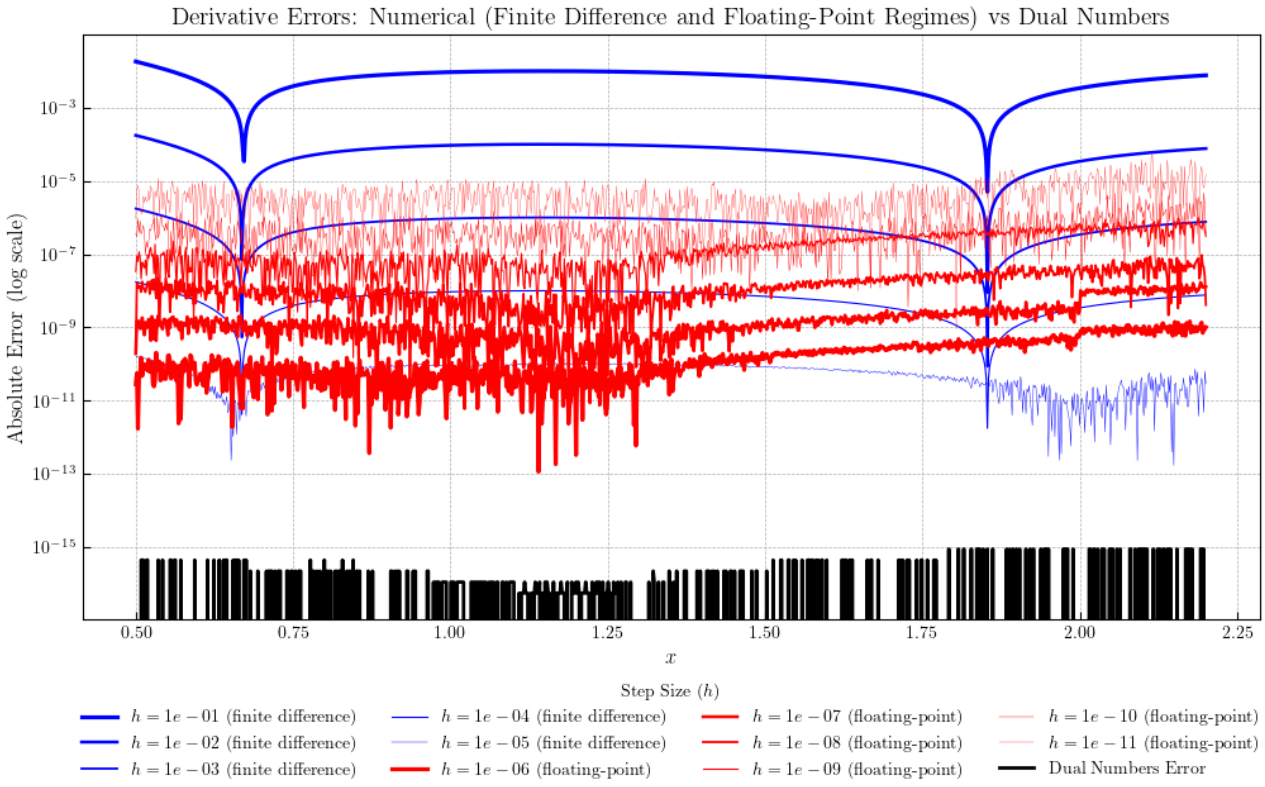

Figure 1: Absolute errors of derivatives computed numerically (finite difference and floating-point regimes) and using dual numbers. Numerical errors for $h \gtrsim 10^{-5}$ are dominated by truncation error, while $h \lesssim 10^{-5}$ errors are dominated by floating-point precision limitations. Dual numbers exhibit consistently low errors independent of h.

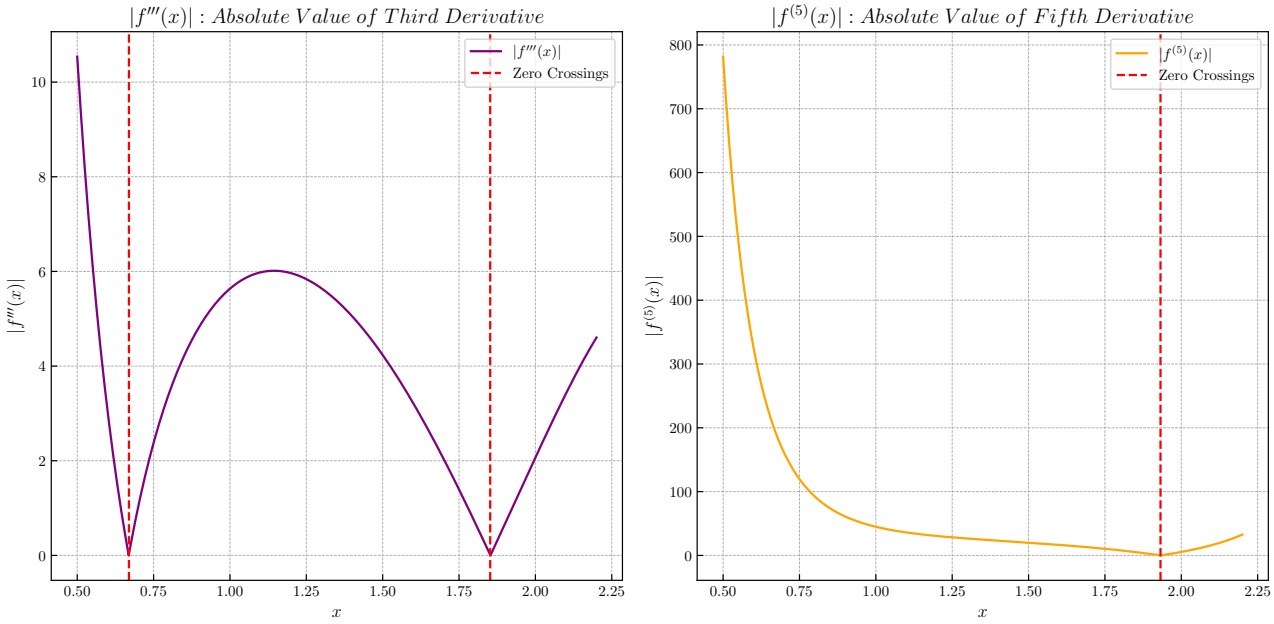

Figure 2: Magnitudes of the third and fifth derivatives, $|f^{(3)}(x)|$ and $|f^{(5)}(x)|$, across the domain $x \in [0.5, 2.2]$. Numerical truncation error dips align with zero crossings or local minima of these derivatives.

4.3 Modeling Average Absolute Numerical Derivative Error and Optimal Step Size

To investigate the optimal step size h_{optimal} for minimizing the total derivative error, we model the average absolute error $E(h)$ across $x \in [0.5, 2.2]$ as a piecewise linear function in log-log space. The total error is expressed as the sum of truncation and round-off errors:

$$E_{\text{total}}(h) = E_{\text{truncation}}(h) + E_{\text{round-off}}(h)$$

where

$$E_{\text{truncation}}(h) \propto h^2, \quad E_{\text{round-off}}(h) \propto \frac{\epsilon_{\text{machine}}}{h}.$$

Error Behavior in Log-Log Space

Taking the logarithm of both components:

$$\log(E_{\text{truncation}}) = 2\log(h) + \text{constant},$$

$$\log(E_{\text{round-off}}) = -\log(h) + \text{constant}.$$

In log-log space, these errors form two straight lines with slopes $+2$ (truncation error) and -1 (round-off error), intersecting at h_{optimal}, where the total error is minimized. The total error E_{total} in log-log space forms a V-shaped curve:

$$\log(E(h)) = \begin{cases} m_1(\log(h) - \log(h_{\text{optimal}})) + \log(E_{\text{optimal}}), & \log(h) \leq \log(h_{\text{optimal}}), \\ m_2(\log(h) - \log(h_{\text{optimal}})) + \log(E_{\text{optimal}}), & \log(h) > \log(h_{\text{optimal}}), \end{cases} \tag{6}$$

where:

- $m_1 = 2$: slope in the truncation error region,
- $m_2 = -1$: slope in the round-off error region,

- $\log(h_{\text{optimal}})$: breakpoint of the V (optimal step size),
- $\log(E_{\text{optimal}})$: minimum error at h_{optimal}.

Fitting the Model

To determine h_{optimal}, we fit the above piecewise linear model to the observed average error data in log-log space using nonlinear least squares optimization. The fitting process yields:

1. The best-fit slopes m_1 and m_2,
2. The breakpoint $\log(h_{\text{optimal}})$,
3. The minimum error $\log(E_{\text{optimal}})$.

The resulting model captures the error behavior and identifies h_{optimal} as the step size that minimizes the total error. The theoretical optimal step size [5] is given by the cube root of the machine epsilon, $h_{\text{opt}} = \epsilon^{1/3}$. For double precision, this evaluates to $h_{\text{opt}} \approx 6.055 \times 10^{-6}$. Our experimentally determined optimal step size is 7×10^{-6}, which aligns closely with the theoretical prediction.

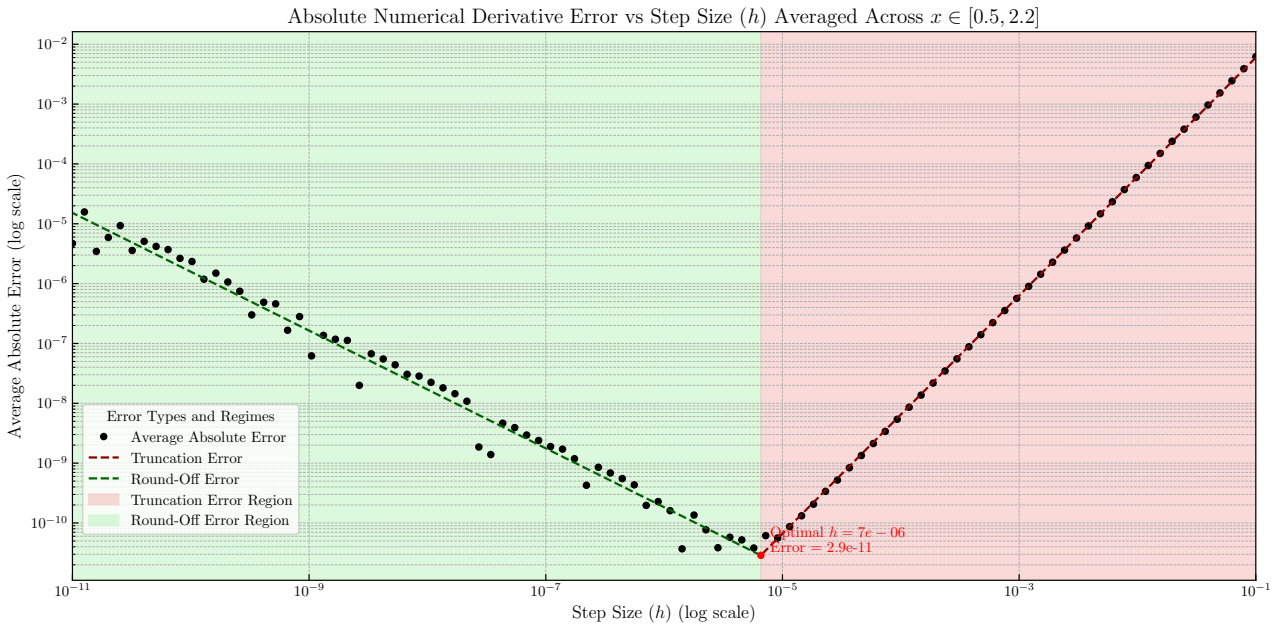

Figure 3: Fitted log-log model of average absolute derivative error as a function of h. The optimal step size h_{optimal} corresponds to the bottom of the V-shaped curve, balancing truncation and round-off errors.

5 Benchmarking

This section evaluates the computational performance of derivative computation methods and dual number implementations across various datasets. The benchmarking is divided into three parts: execution time comparison for the three derivative methods, performance comparison of Python and Cython for basic operations and derivatives, and scaling analysis for Python and Cython implementations.

5.1 Execution Time for Derivative Methods

Here, we compare the execution time of analytical, numerical, and dual number derivative methods for 10000 points equally spaced over $x \in [1.1, 2.5]$. The results in Figure 4 show that analytical derivatives are the fastest (0.14 s), followed by numerical derivatives (0.26 s), with dual number methods being the slowest (0.69 s for Python and 0.62 s for Cython).

The **analytical method** achieves the best performance by directly evaluating pre-derived expressions, avoiding iterative or memory-intensive computations. This makes it the most efficient approach in terms of execution time.

The **numerical method** exhibits consistent execution times across the entire range of h, with only minor fluctuations. Unlike analytical derivatives, which evaluate pre-derived expressions directly, the numerical method relies on the central difference formula, requiring two function evaluations for each derivative computation. This additional overhead makes it slower than the analytical method. Despite varying h, the execution time remains largely unaffected.

The **dual number method** incurs the highest computational overhead, with Python-based implementation taking 0.69 s and the Cython-optimized version reducing this to 0.62 s. The overhead arises from:

- **Object Instantiation**: Each input is represented as a dual number, requiring additional memory and initialization of real and dual components.
- **Arithmetic Overloading**: Operations such as addition, multiplication, and trigonometric evaluations are overloaded to handle dual arithmetic, increasing computational complexity.
- **Memory Management**: Accessing and storing dual components adds further overhead.

The Cythonized implementation achieves a noticeable speedup compared to the Python version by optimizing arithmetic operations and memory handling at the compiled code level [6], though it still lags behind analytical and numerical methods due to the inherent complexity of dual arithmetic.

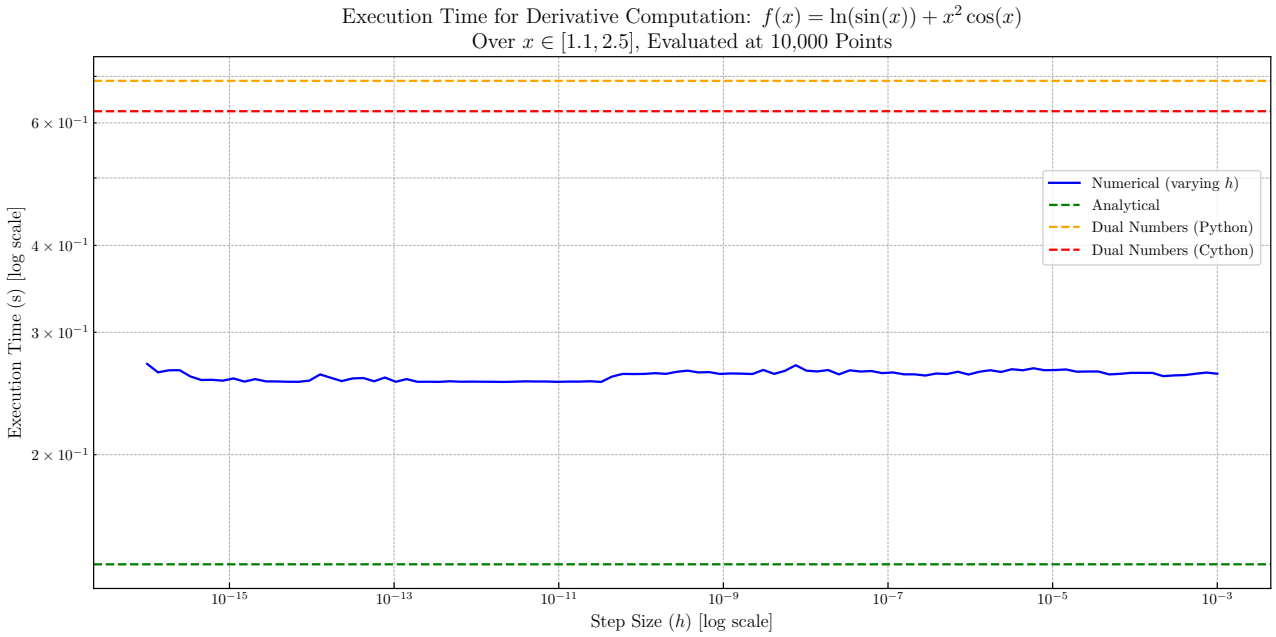

Figure 4: Execution time comparison for derivative computation methods over $x \in [1.1, 2.5]$. Analytical derivatives (0.14 s) are the fastest, followed by numerical derivatives (0.26 s). Dual number methods are the slowest, with Python implementation taking 0.69 s and Cython optimization reducing it to 0.62 s.

5.2 Python vs Cython: Basic Operations and Differentiation

Table 2 highlights the performance gains achieved with Cythonized implementations over Python. Each operation/function derivative was calculated 1 million times and 5 experiments were repeated with the results being aggregated. Median performance times of aggregated results were used to quantify performance increase percentages.

For **arithmetic, logarithmic, and exponential functions**, speedups range from 15.06% ($\log(x)$) to 37.26% ($x+y$), reflecting reduced overhead from Python's dynamic type handling in basic operations.

Trigonometric and hyperbolic functions achieve speedups between 18.03% ($\tan(x)$) and 23.10% ($\sinh(x)$), demonstrating moderate gains from the optimization of computationally intensive transcendental evaluations.

For **differentiation of machine learning (ML) functions**, the speedups are more substantial, ranging from 23.51% ($f(x) = \tanh(x)$) to 32.01% ($f(x) = \text{ReLU}$), highlighting the effectiveness of Cython in accelerating operations that involve complex composite functions and derivative propagation.

Category	**Operation**	**Performance Increase (%)**
Arithmetic, Logarithmic, and Exponential	$x + y$	37.26
	$x - y$	30.65
	$x \times y$	31.79
	x/y	23.24
	$\exp(x)$	23.37
	$\log(x)$	15.06
Trigonometric Functions	$\sin(x)$	19.38
	$\cos(x)$	18.55
	$\tan(x)$	18.03
	$\sinh(x)$	23.10
	$\cosh(x)$	20.92
	$\tanh(x)$	21.02
Differentiation of Functions	$f(x) = \frac{1}{1+e^{-x}}$ (Sigmoid)	27.78
	$f(x) = \tanh(x)$ (Tanh)	23.51
	$f(x) = \max(0, x)$ (ReLU)	32.01
	$f(x) = \log(1 + e^x)$ (SoftPlus)	27.35
	$f(x) = (x - 1)^2$ (MSE)	28.86
	$f(x) = -[y \log(x) + (1-y) \log(1-x)]$ (BCE)	28.93

Table 2: Summary of arithmetic, trigonometric, logarithmic/exponential functions, and derivatives of machine learning (ML) functions. Performance increases for Cython implementations over Python are shown as percentages.

5.3 Python vs Cython: Data Scaling Analysis

This section investigates the performance of Python and Cython implementations under increasing dataset sizes. The benchmarks evaluate execution time for both basic operations and derivative computations to understand their scaling behavior. The benchmarking process was designed to ensure fair and consistent evaluation. For basic operations (e.g., addition, multiplication, trigonometric, and logarithmic functions), random input values were generated within predefined ranges to simulate realistic workloads. Each operation was applied repeatedly across datasets of varying sizes to measure execution times. For derivative computations, random real values were similarly generated, and the derivative of pre-specified functions was calculated using dual numbers. To account for variability, multiple repetitions were averaged for each dataset size. Both Python and Cython implementations were continuously benchmarked on the same datasets to ensure fairness in the comparisons.

5.3.1 Scalability of ML Function Derivatives

To evaluate the performance of Python and Cython implementations in computing derivatives of machine learning (ML) functions, we analyzed the relationship between execution time $T(N)$ and dataset size N. For such operations, the execution time is expected to scale linearly with dataset size, expressed as:

$$T(N) = K \cdot N + C \tag{7}$$

where:

- K represents the per-element computation cost, quantifying how execution time increases with the number of elements.
- C represents constant overhead, encompassing fixed costs such as initialization, memory allocation, and interpreter overhead.

The $O(N)$ scaling of derivative computations and basic arithmetic operations arises naturally when using dual numbers. In this approach, derivatives are computed by evaluating the function at a dual number and extracting the dual part. This method leverages the intrinsic structure of dual numbers, where the primary computation of the function value (the real part) and its derivative (the dual part) occur simultaneously through simple arithmetic operations. Each operation, whether evaluating a basic arithmetic function or computing a derivative, processes the dataset independently and does not require interactions between elements, preserving linear scaling.

For arithmetic operations such as addition, subtraction, and multiplication, the dual number formalism ensures that both the function evaluation and its derivative require a fixed, element-wise cost. Similarly, transcendental functions (e.g., $\exp(x)$, $\log(x)$, or sigmoid-like operations) directly extend to dual numbers with constant overhead per element. Consequently, the overall complexity of evaluating derivatives and performing arithmetic operations is proportional to the dataset size.

We fit linear regression models to execution time data, extracting K (gradient) and C (constant overhead) for both Python and Cython implementations. The fitted parameters were then used to quantify the efficiency improvements provided by Cython. The linear relationship between execution time and dataset size was verified experimentally, as shown in Figure 5.

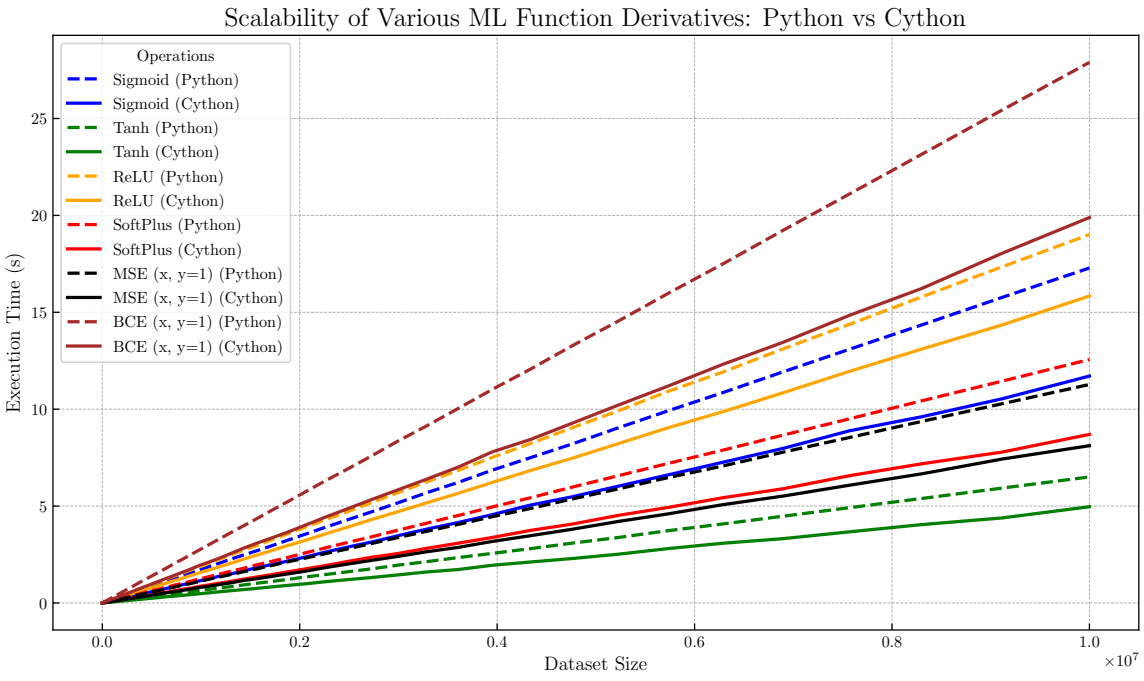

Figure 5: Linear Scaling of Derivative Computations: Python vs Cython.

Table 3 summarizes the fitted values of K and C for Python and Cython implementations. The relative reductions in K and C demonstrate the efficiency gains achieved by Cython across various derivative computations.

Function	**Python**		**Cython**		**Reduction (%)**	
	K ($\times 10^{-6}$)	C ($\times 10^{-5}$)	K ($\times 10^{-6}$)	C ($\times 10^{-5}$)	K	C
Sigmoid ($f(x) = \frac{1}{1+e^{-x}}$)	1.729	-0.9134	1.158	0.3980	33.03	143.57
Tanh ($f(x) = \tanh(x)$)	0.6494	-3.1893	0.4844	-0.8128	25.41	74.51
ReLU ($f(x) = \max(0, x)$)	1.901	-1.9276	1.575	1.680	17.15	108.72
SoftPlus ($f(x) = \log(1+e^x)$)	1.255	-2.7722	0.8558	0.8434	31.84	419.01
MSE ($f(x) = (x-1)^2$)	1.127	-1.0733	0.8020	-0.6196	28.86	42.26
BCE ($f(x) = -[y \log(x) + (1-y)\log(1-x)]$)	2.788	-5.3759	1.954	0.8664	29.91	116.12

Table 3: Scaling parameters K (gradient) and C (intercept) for Python and Cython implementations. Reductions in K and C highlight the efficiency improvements achieved by Cython.

Key Insights.

- **Reductions in Constant Overhead (C):**
  - Substantial reductions in C were observed for *SoftPlus* (419.01%) and *Sigmoid* (143.57%), indicating that Cython excels at reducing fixed computational costs.
  - Negative C values for Python in certain cases suggest initialization delays or noise for smaller datasets, which are (mostly) resolved in Cython.
- **Improvements in Per-Element Computation Cost (K):**
  - Cython consistently reduced K across all functions, with improvements ranging from 17.15% (*ReLU*) to 33.03% (*Sigmoid*).
  - This demonstrates the ability of Cython to optimize arithmetic operations and memory access patterns.
- **Behavior of Computationally Lightweight Functions:**
  - Functions like *ReLU* and *Tanh* exhibited smaller improvements in K, reflecting their already low computational complexity in Python.
- **Loss Functions:**
  - Both *MSE* and *BCE* showed consistent reductions in K and C, suggesting that Cython optimizations generalize effectively to both activation and loss functions.

5.3.2 Relative Performance Analysis

Figure 6 presents the relative performance improvement of Cython over Python for various operations, including arithmetic, trigonometric, and derivative computations, as a function of dataset size.

The analysis demonstrates that while the relative performance advantage of Cython slightly decreases for very large dataset sizes due to the diminishing impact of fixed overheads, the overall performance improvement remains consistent and substantial across all dataset sizes.

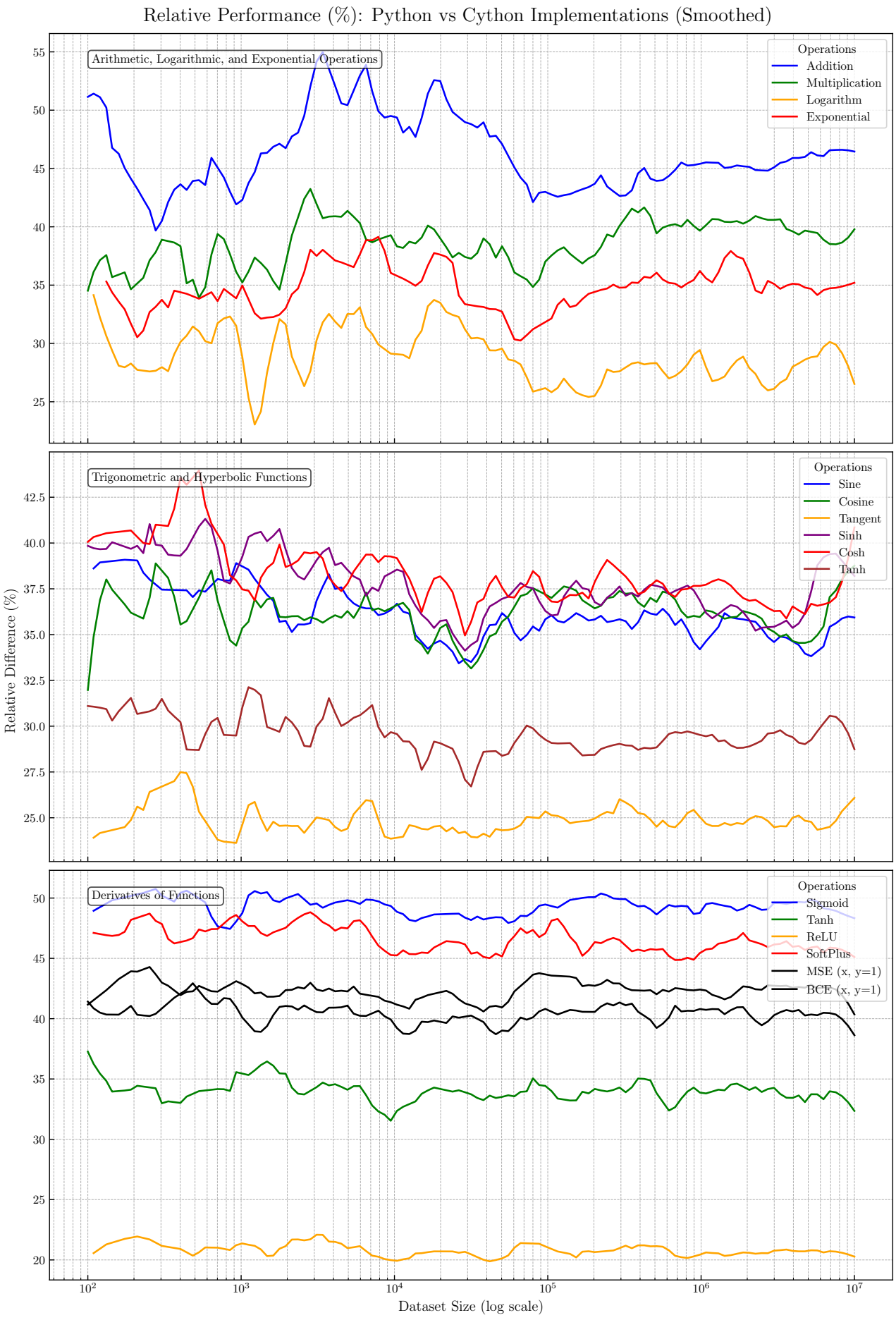

Figure 6: Relative Performance (%): Python vs Cython Implementations. Smoothing techniques were applied for clarity.

6 Discussion

6.1 Summary of Results

The results consistently demonstrate the computational advantages of dual numbers, particularly in the context of automatic differentiation. Numerical differentiation methods showed clear limitations, with truncation and floating-point errors contributing to a U-shaped error behavior as step size varied. In contrast, dual number-based differentiation achieved machine-level precision, independent of step size. The Cythonized implementation of the `Dual` class highlighted significant performance gains over the Python implementation. Basic operations such as addition and trigonometric evaluations showed consistent improvements, while differentiation of machine learning functions benefited substantially, underscoring the scalability of Cython optimizations, especially for large datasets.

6.2 Future Directions: Enhancing the `Dual` Class

While the current implementation of the `Dual` class has proven effective, several improvements can further enhance its utility and performance:

1. **Vectorization of the `Dual` Class:** Current operations on dual numbers are scalar-based, which can be computationally expensive for large datasets. Vectorizing the `Dual` class would allow operations to be applied element-wise on arrays of dual numbers, leveraging libraries such as NumPy for optimized performance.
    - **Advantages of vectorization:**
        - Significant speedup due to parallelized computation.
        - Reduced memory overhead by avoiding repeated object instantiation.
        - Seamless integration with machine learning frameworks that handle data in vectorized formats.
    - Vectorization would make the `Dual` class more applicable to high-dimensional problems, such as gradient computations for large-scale optimization and neural network training.
2. **Support for Higher-Order Derivatives:** Extending the current implementation to compute second-order and higher-order derivatives would enhance its applicability in areas like Hessian computations and second-order optimization algorithms.
    - **Approach:**
        - Introduce nested dual numbers or alternative representations such as Taylor series coefficients.
        - Optimize the implementation to maintain computational efficiency, especially for higher dimensions.
3. **Integration with GPU Acceleration:** The current implementation is CPU-bound. Incorporating GPU-based computation via frameworks like CuPy or PyTorch would enable faster differentiation for extremely large datasets or real-time applications.

7 Conclusion

This project successfully implemented forward-mode automatic differentiation using dual numbers and demonstrated that dual numbers effectively overcome the limitations of numerical differentiation, providing exact derivatives without step-size-related errors. Additionally, the Cythonized implementation delivered significant computational performance gains over the pure Python implementation, particularly for large datasets and complex functions.

References

[1] Jeffrey A Fike and Juan J Alonso. Automatic differentiation through the use of hyper-dual numbers for second derivatives. In *Recent Advances in Algorithmic Differentiation*, pages 163–173. Springer, 2012.

[2] Birthe van den Berg, Tom Schrijvers, James McKinna, and Alexander Vandenbroucke. Forward-or reverse-mode automatic differentiation: What's the difference? *Science of Computer Programming*, 231:103010, 2024.

[3] Richard L Burden and J Douglas Faires. Numerical analysis, 2010.

[4] Timothy Sauer. *Numerical analysis*. Addison-Wesley Publishing Company, 2011.

[5] Knut Mørken. Numerical algorithms and digital representation. *Oslo, Norway: University of Oslo.(https://www. uio. no/studi er/emner/matnat/math/MAT-INF11 00/h17/kompe ndiet/matin f1100. pdf) Accessed*, 1(29):2021, 2017.

[6] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2010.

Appendix

A Declaration of Autogenerative Tools

In this project, I used GitHub Copilot as a coding assistant. I mainly used it to generate the plots in Figure 1, 2, 3, 4, 5 and 6 and the Tables 1, 2, and 3. I also used it within my code to create comments, docstring my functions, debug and format markdown cells explaining the different sections in the notebook.

In regards to the report, I used the Overleaf AI which uses ChatGPT to help write the mathematics in latex format, and also used it to correctly format various sections of the report as I am quite new to LaTeX.