Dual Number Automatic Differentiation: Implementation and Analysis

Phong-Anh Nguyen Trinh (pan31)

December 18, 2024

Word Count: 1516

1 Introduction

The concept of automatic differentiation (AD) has become increasingly important in modern scientific computing, particularly in machine learning and optimization. Dual numbers, an extension of real numbers that include an infinitesimal component, provide an elegant approach to implementing forward-mode automatic differentiation.

The goal of this project is to implement a dual number system in pure Python and optimize performance through Cythonization. We will compare the performance characteristics of the two implementations and analyze the trade-offs between implementation approaches. These techniques are packaged in a Python package following coding good practices.

1.1 Background

Dual numbers, introduced by William Clifford [1], extend real numbers by adding a nilpotent element ϵ with the property $\epsilon^2 = 0$. A dual number takes the form:

$a + b\epsilon$,

where a represents the real part and b represents the dual part.

2 Methodology

2.1 Pure Python Implementation

The pure Python implementation focuses on clarity and maintainability. The functionality of the package is implemented in a base class called Dual. The Dual class takes two arguments: the real and the dual component, which is then passed to the Dual object to be used to perform the operations.

```
class Dual:
    def __init__(self, real, dual=0.0):
        self.real = real
        self.dual = dual
```

Basic operations such as addition, subtraction, multiplication, division, and indices are implemented as methods of the Dual class. These operations were overloaded over standard symbols such as $+$, $-$, $*$, $/$, and $**$. The reverse operators (for when the Dual class is on the right-hand side of the operator), were also defined for completeness. In the case of a Dual number and a non-Dual number, the operator will turn the non-Dual number into a Dual number, with a Dual component of 0. The operations will always return a Dual object.

An example of overloading the + operator:

```
def __add__(self, other) -> "Dual":
# Check if the other object is a Dual number
if not isinstance(other, Dual):
    other = Dual(other)
return Dual(self.real + other.real, self.dual + other.dual)
```

For two dual numbers: $a + b\epsilon$, $c + d\epsilon$, the following mathematical operations were defined:

2.1.1 Addition

Addition is simply just the addition of the real and Dual components.

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon \tag{1}$$

2.1.2 Subtraction

Subtraction is simply the subtraction of the real and Dual components.

$$(a + b\epsilon) - (c + d\epsilon) = (a - c) + (b - d)\epsilon \tag{2}$$

2.1.3 Multiplication

Multiplication of two dual numbers involves multiplying the real parts and applying the distributive property to the dual parts:

$$(a + b\epsilon) \times (c + d\epsilon) = (a \times c) + (a \times d + b \times c)\epsilon \tag{3}$$

2.1.4 Division

Division of two dual numbers involves dividing the real parts and applying the quotient rule to the dual parts:

$$\frac{a + b\epsilon}{c + d\epsilon} = \frac{a}{c} + \frac{b \times c - a \times d}{c^2}\epsilon \tag{4}$$

2.1.5 Exponentiation

Raising a dual number to the power of another dual number involves using the chain rule and logarithmic identities:

$$(a + b\epsilon)^{(c+d\epsilon)} = a^c \left(1 + \left(\frac{b \cdot c}{a} + d \cdot \ln(a)\right)\epsilon\right) \tag{5}$$

A number of useful functions were also added to the base Dual class:

2.1.6 Sine Function

The sine of a dual number is calculated by applying the sine function to the real part and the cosine function to the dual part:

$$\sin(a + b\epsilon) = \sin(a) + b \cdot \cos(a)\epsilon \tag{6}$$

2.1.7 Cosine Function

The cosine of a dual number is calculated by applying the cosine function to the real part and the negative sine function to the dual part:

$$\cos(a + b\epsilon) = \cos(a) - b \cdot \sin(a)\epsilon \tag{7}$$

2.1.8 Tangent Function

The tangent of a dual number is calculated by applying the tangent function to the real part and the derivative of the tangent function to the dual part:

$$\tan(a + b\epsilon) = \tan(a) + b \cdot \sec^2(a)\epsilon \tag{8}$$

2.1.9 Logarithm Function

The logarithm of a dual number is calculated by applying the logarithm function to the real part and dividing the dual part by the real part:

$$\log(a + b\epsilon) = \log(a) + \frac{b}{a}\epsilon \tag{9}$$

2.1.10 Exponential Function

The exponential of a dual number is calculated by applying the exponential function to the real part and multiplying the dual part by the exponential of the real part:

$$e^{(a+b\epsilon)} = e^a\,(1 + b\epsilon) \tag{10}$$

2.1.11 Differentiation

The crucial use of Dual numbers is in being able to efficiently calculate the exact derivative of a function at a certain point. Given a function: $f(x)$, the derivative at point x_0 is calculated with Dual numbers in the following way:

Expand $f(x_0 + \epsilon)$ using a Taylor Expansion

$$f(x_0 + \epsilon) = f(x_0) + f'(x_0)\epsilon + \frac{f''(x_0)}{2}\epsilon^2 \ldots \tag{11}$$

Since $\epsilon^2 = 0$, all of the higher order terms vanish leaving:

$$f(x_0 + \epsilon) = f(x_0) + f'(x_0)\epsilon \tag{12}$$

Therefore, the derivative of $f(x)$ at point x_0 is the coefficient of the dual part of the result, $f'(x_0)\epsilon$

2.2 Cythonised Implementation

Cythonisation is the process of compiling Python code in C, which can significantly speed-up execution, especially for numerical computations and loops.

A separate module called `dual_autodiff_x` was created for the Cython code, where a new base class was created (`dual.pyx`), with the exact same functionality as the pure Python package. In this base class, static type declaration was performed for the class and its functions.

```
cdef class Dual:
    cdef public double real
    cdef public double dual

    def __init__(self, double real, double dual=0.0):
        self.real = real
        self.dual = dual
```

The Cythonisation of the code was performed using the `Cython` library for the build process (compiling the `dual.c` file)

2.3 Build System and Packaging

2.3.1 Packaging

To perform the setup for the package, a `pyproject.toml` was included the code. It contains all of the project information, dependencies, and tools for automatic versioning, and packaging. It allows the package to be installed on any environment by running `pip install -e` from the root folder of the module. The Cythonised package had its own `pyproject.toml` and `setup` file, so that it could be installed as a standalone package if required by the user.

2.3.2 Git, CI/CD and Test Suite

For integration with Git, and to perform Continuous Integration and Continuous Development (CI/CD), the package comes fully tested (testing both valid and invalid inputs). This test suite prevents incorrect changes to the code. This is integrated by creating a automatic pre-commit git hook. Automatic versioning is also implemented, where the version is updated for every push to the main branch. Due to the package structure, the `dual_autodiff_x` does not have automatic versioning, as it would require a fully-built Git repo, which would lead to a Git repo within a Git repo.

2.3.3 Compiling Cython Wheels

`cibuildwheel` was used to create the wheels for specific Linux and Python (3.10, 3.11) installations using different Docker images. Wheels offer many benefits, including pre-compiling binaries (removing the need for compiliers on machines), faster installation, and dependency management. It is especially necessary for Cythonsied code, to have separate wheels as the compilers found on Linux, OSX, and Windows laptops can be different. Without the necessary wheels, the C code will not compile. The `.pyx` (source code) was excluded from the wheels, following good practice.

2.3.4 Documentation

Documentation is vital for users to understand how the package works, and should always be up to date. In this sense, the ability to automatically update documentation whenever there is a change to code is useful. This is implemented using the `Sphinx` library, which detects the fully documented (type-hinting and docstrings) within the code, and outputs them to a `.html` file within the `docs/build` folder. Jupyter notebooks are also integrated into the documentation using the `pandoc` extension. By running `make html` from the root directory, the all necessary dependencies are installed, and the documentation is built. The dependencies for documentation are not included in the wheels or pyproject.toml file, as they are not necessary for the package to run. An improvement would be to create another Git commit hook to update the documentation when changes are mode to the code.

3 Results

3.1 Automatic Differentiation

To demonstrate the automatic differentiation, 3 alternative methods were implemented to solve the following equation:

$$f(x) = \log(\sin(x)) + x^2 \cos(x) \tag{13}$$

3.1.1 Analytical Solution

The analytical solution to Equation 13 is:

$$f'(x) = \frac{\cos(x)}{\sin(x)} + 2x\cos(x) - x^2 \sin(x) \tag{14}$$

which equals -1.9123 at x = 1.5.

3.1.2 Automatic Differentiation Solution

To perform the automatic differentiation, the `Dual` class was used. This yielded the a value of -1.9123 at x = 1.5, which is the same as the analytical solution.

3.1.3 Numerical Solution

To perform the numerical differentiation, the central difference method was used. This was performed for a variety of step sizes, and the results are shown in Figure 1. This method is defined as:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} \tag{15}$$

The results compared to the other methods are shown in Figure 1.

3.1.4 Timimg Comparison

A benchmark was performed to compare the time taken (automatic and numerical differentiation), by performing the differentiation 1000 times, and averaging the time taken.

Method	Time (s)
Numerical	0.003912
Automatic	0.008927

Table 1: Performance comparison of numerical and automatic differentiation

3.2 Cython Performance

A benchmark study of the four basic operations was performed, with 1 million iterations between the Python and Cython versions. Figure 2 shows the speedup of the basic operations, and Figure 3 shows the memory use.

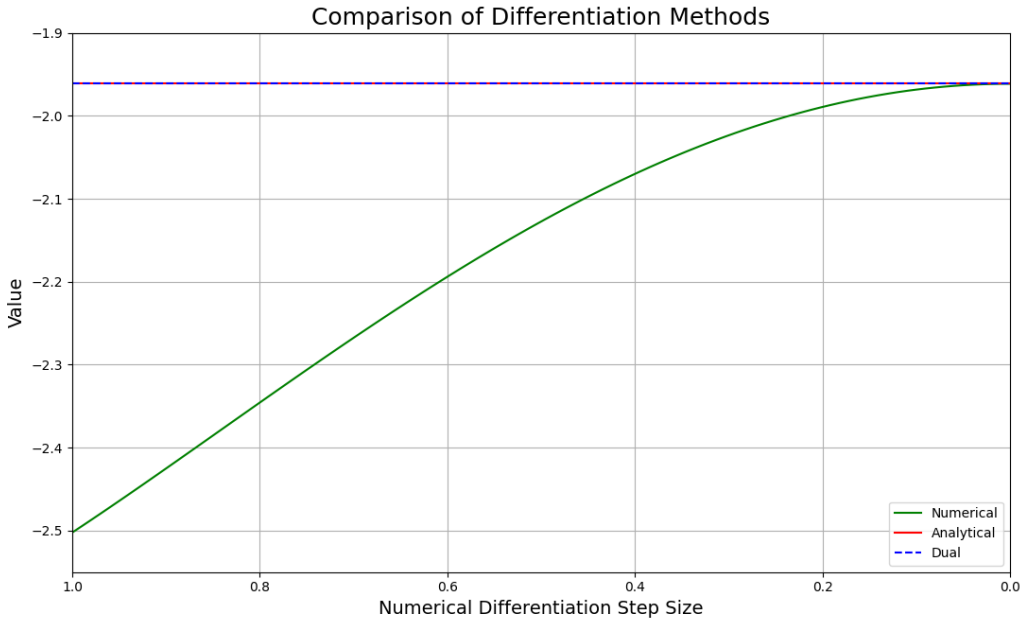

Figure 1: Comparison of Differentiation Methods

Method	Python Time (ms)	Cython Time (ms)	Python Memory (B)	Cython Memory (B)
Addition	0.007000	0.012000	912	848
Multiplication	0.013900	0.014400	848	752
Division	0.024000	0.007200	880	720
Derivative	0.049900	0.007700	1120	784

Table 2: Time and Memory Comparison of Basic Operations

4 Discussion

4.1 Automatic Differentiation vs. Numerical Differentiation

As shown by Figure 1, numerical differentiation heavily depends on the step size given, and theoretically can never find the true solution (as step size must be finite). Automatic Differentiation, however, finds the exact solution, which is incredibly useful. In practice however, with a small enough step-size, the numerical differentiation will reach the solution to within floating-point error, which is the same as any method was achieved. Also, the differentiation method used was central difference, which is a relatively crude method compared to the methods available today.

The primary use of Automatic Differentiation would be in machine learning (especially important for the back-propagation process for neural networks), and that should be the context in which we evaluate the different methods. In terms of implementation, numerical differentiation step-size would effectively become a hyperparameter, where the goal would be to make it as small as possible. Without a sufficiently small enough step-size, the differentiation would be inaccurate, and propagate the errors to the learning process, leading to ineffective learning. Also, central differences in higher dimensions is computationally expensive, which is an issue for modern neural networks.

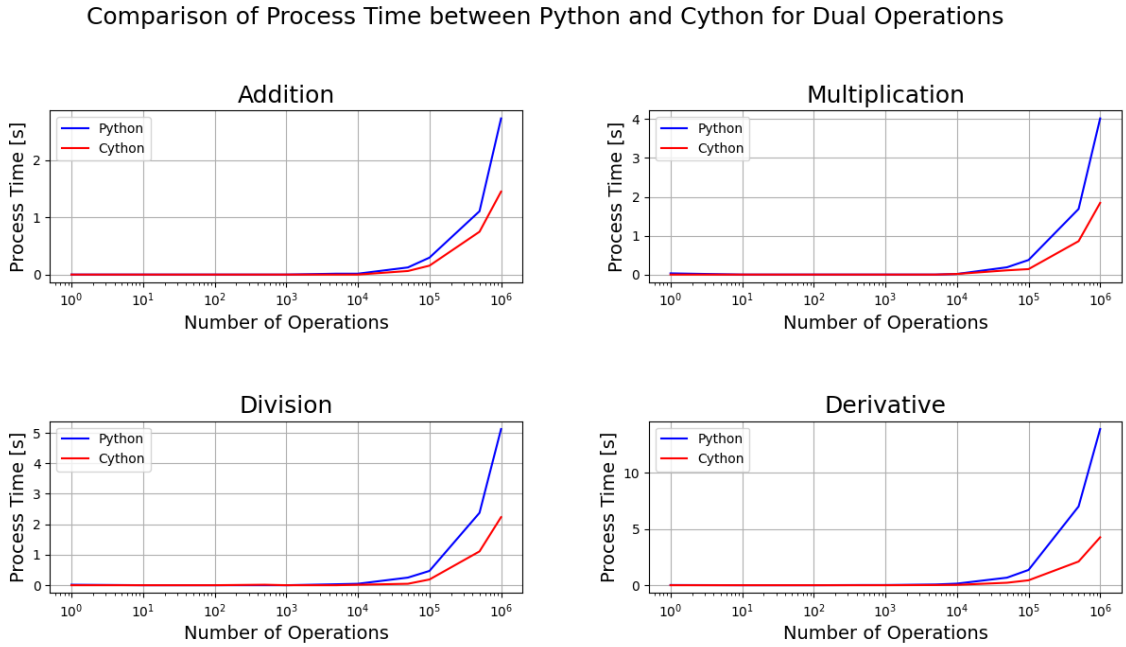

Figure 2: Python vs. Cython Speedup for 1 million operations

In terms of performance the numerical differentiation is faster than the automatic differentiation, as it is a simpler method, but the negatives of numerical differentiation outweigh the positives for the specific use case of machine learning.

4.2 Python vs. Cython Trade-offs

4.2.1 Development Complexity

The Cythonized implementation introduces additional complexity, requiring: separate build process, platform-specific compilation (wheels), and increased maintenance overhead.

4.2.2 Performance Benefits

For use in machine learning, the reduced memory usage of Cython is extremely useful, as models get larger. However, the memory usage for derivatives show no improvement, which is the most critical use of Dual numbers. In Figure 3, the memory usage for derivatives is the same as the Python implementation, which implies that the C optimization may not be as impactful as the amount of data being processed overshadows the memory optimizations. In terms of speed, Cython makes a noticeable improvement, especially for large numbers of operations, which is useful for machine learning. This is likely due to Cython's optimisations for running loops and computations, but the difference is negligible when the number of operations is small. The performance improvements justify the added complexity.

4.3 Computational Efficiency

The operations on Dual numbers have constant time complexity:

- Addition/Subtraction: $O(1)$ - requires two basic arithmetic operations
- Multiplication: $O(1)$ - requires three multiplications and one addition

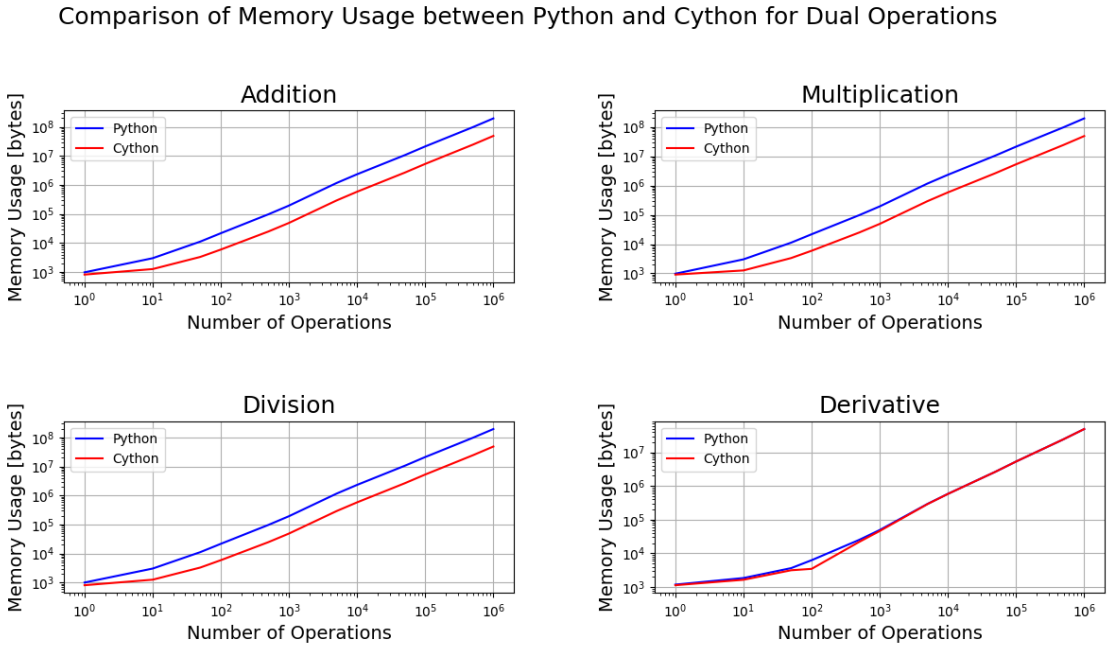

Figure 3: Python vs. Cython Memory Use for 1 million operations

- Division: $O(1)$ - requires three divisions, one multiplication, and one subtraction
- Elementary Functions: $O(1)$ - requires evaluation of the function and its derivative at a point

This constant-time complexity makes Dual numbers particularly efficient for automatic differentiation, as the computational cost of each operation remains bounded regardless of the complexity of the expression being differentiated. The constant-time complexity ($O(1)$) of Dual number operations represents a significant advantage over alternative differentiation methods. This efficiency stems from the elegant mathematical properties of dual numbers, where $\epsilon^2 = 0$ eliminates the need for higher-order terms in calculations.

When compared to other differentiation methods:

- **Numerical Differentiation:** Requires multiple function evaluations for finite differences and can suffer from round-off errors and step-size sensitivity.
- **Dual Numbers:** Maintains constant-time complexity regardless of function complexity, with exact derivatives computed alongside function evaluation.

This efficiency is particularly valuable in machine learning applications, where gradient computations are performed repeatedly during optimization. The Cythonized implementation further enhances this advantage by reducing the overhead of Python's dynamic typing, resulting in performance approaching that of pure C implementations while maintaining the convenience of Python's syntax.

4.4 Future Work

To build on this package, futher improvements include: an implementation of reverse-mode differentiation, which is more efficient for nested functions. This would allow for more complex models to be differentiated, and would be a significant improvement for machine learning.

Extended support for complex mathematical functions, such as inverse trigonometric functions, which are not currently supported. Support for multivariate dual numbers and operations. More comprehensive testing and documentation, such as linting, code coverage and type hinting

References

[1] William Kingdon Clifford. Preliminary sketch of biquaternions. *Proceedings of the London Mathematical Society*, 4:381–395, 1873.