C1 Research Computing Coursework Report

Steven Ma(ym432)

December 2024

Word Count: 1620

1 Introduction

This report investigates the performance of two implementations of the `dual_autodiff` package for automatic differentiation using dual numbers: a pure Python version and a Cythonized version (`dual_autodiff_x`). The Cythonized version shows a performance improvement of approximately 2 to 2.5 times over the Python implementation. Both versions compute derivatives exactly, matching analytical results. In contrast, the numerical derivative implementation in `SciPy`[1] is significantly slower, with `dual_autodiff` being up to 30 times faster and `dual_autodiff_x` reaching speeds more than 60 times faster than SciPy's approach. This report will explore the reasons behind these performance differences and the implications for numerical differentiation tasks.

2 Basics and Definition

Dual numbers are a mathematical structure similar to complex numbers but with a special property:

$$\epsilon^2 = 0, \tag{1}$$

where ϵ is the dual unit. A dual number x can be expressed as:

$$x = a + b\epsilon, \tag{2}$$

where a is the real part, and $b\epsilon$ is the dual part.

The arithmetic operations for dual numbers are defined as follows:

Addition: For two dual numbers $x = a + b\epsilon$ and $y = c + d\epsilon$, the sum is:

$$x + y = (a + c) + (b + d)\epsilon. \tag{3}$$

Subtraction: The difference is:

$$x - y = (a - c) + (b - d)\epsilon. \tag{4}$$

Multiplication: Using the property $\epsilon^2 = 0$, the product is:

$$x \cdot y = (a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\epsilon. \tag{5}$$

Division: For $x = a + b\epsilon$ and $y = c + d\epsilon$, the division is performed by multiplying by the conjugate of the denominator:

$$\frac{x}{y} = \frac{(a + b\epsilon)(c - d\epsilon)}{(c + d\epsilon)(c - d\epsilon)} = \frac{a}{c} + \frac{bc - ad}{c^2}\epsilon. \tag{6}$$

2.1 Automatic Differentiation Using Dual Numbers

Automatic Differentiation (AD) is a method for efficiently computing derivatives of functions. Dual numbers provide an elegant way to achieve this by leveraging their unique arithmetic properties.

Given a differentiable function $f : \mathbb{R} \to \mathbb{R}$, the input a is extended into a dual number:

$$\hat{a} = a + \epsilon, \tag{7}$$

where the dual part is set to 1 to track derivatives.

The function f is then evaluated at the dual number $\hat{a}$:

$$f(\hat{a}) = f(a + \epsilon). \tag{8}$$

Using the Taylor expansion of f around a:

$$f(a + \epsilon) = f(a) + f'(a)\epsilon + \frac{f''(a)}{2!}\epsilon^2 + \cdots, \tag{9}$$

and since $\epsilon^2 = 0$, higher-order terms vanish, leaving:

$$f(a + \epsilon) = f(a) + f'(a)\epsilon. \tag{10}$$

Dual numbers naturally enforce the product, chain, and quotient rules, automatically propagating derivatives through nested and composite functions.

3 Implementation of `dual_autodiff` and Test Suite

This section corresponds to Questions 1, 2, 3, 4 and 6.

3.1 Structure Overview

The directory structure for `dual_autodiff` package is as follows:

```
dual_autodiff/
|-- dual_autodiff/
|    |-- __init__.py
|    |-- dual.py
|-- tests/
|    |-- __init__.py
|    |-- test_dual.py
|-- pyproject.toml
|-- C1_coursework_main.ipynb
```

Figure 1: Project Repository Structure for `dual_autodiff`

The functionality and contents of each file are described below:

- `dual_autodiff/`: Main package directory.
  - `__init__.py`: Exposes the `Dual` class for external.
  - `dual.py`: Implements the `Dual` class for dual numbers, supporting arithmetic operations, trigonometric, exponential, logarithmic, and other functions for automatic differentiation.
- `tests/`: Directory for unit tests.
  - `__init__.py`: Empty file to initialize the tests package.
  - `test_dual.py`: Unit tests for the `Dual` class, validating initialization, arithmetic operations, and other key functions.
- `pyproject.toml`: Project configuration file specifying build tools, project metadata, and package discovery.
- `C1_coursework_main.ipynb`: Demonstrates differentiation of $f(x) = \ln(\sin(x)) + x^2\cos(x)$, comparing the performance of `dual_autodiff`, its Cythonized version (`dual_autodiff_x`), and SciPy.

3.2 `Dual` Class

The `Dual` class represents dual numbers for automatic differentiation. The class supports arithmetic operations, trigonometric functions, exponentiation, and logarithmic functions, enabling derivative calculations by extracting the dual part.

Core Operations: Four basic arithmetic operations are implemented: addition, subtraction, multiplication, and division. Dual-to-dual, dual-to-scalar, and scalar-to-dual interactions are all handled.

Advanced Functions: The class includes methods for trigonometric (`sin`, `cos`, `tan`), exponential (`exp`), logarithmic (`log`), and square root (`sqrt`) computations, leveraging dual number rules to propagate derivatives. The power operation (`__pow__`) supports scalar and dual exponents.

An example of the addition operation and the `sin` function in the `Dual` class is shown below (for more mathematical definition of the functions, refer to Table 3 in the appendix).

```python
import math

class Dual:
    def __init__(self, real: float, dual: float = 1.0):
        self.real = real
        self.dual = dual

    def __add__(self, other):
        if isinstance(other, Dual):
            return Dual(self.real + other.real, self.dual + other.dual)
        elif isinstance(other, (float, int)):
            return Dual(self.real + other, self.dual)

    __radd__ = __add__

    def sin(self):
        return Dual(math.sin(self.real), math.cos(self.real) * self.dual)
```

Figure 2: Initialization and a Method Definitions for the Dual Class (error handling is omitted)

Example: Addition and Logarithm Here, `Dual + float` resolves to a dual addition. The `log` method correctly computes the derivative as $\frac{1}{2}$.

```python
x = Dual(2.0, 1.0)
y = x + 3.0          # Dual + float
z = 3.0 + x          # float + Dual
print(y)             # Dual(real=5.0, dual=1.0)
print(z)             # Dual(real=5.0, dual=1.0)

w = x.log()
print(w)             # Dual(real=0.6931, dual=0.5)
```

Figure 3: Example Usage of the Dual Class

3.3 Test Suite

The testing framework for `dual_autodiff` is implemented using Python's `unittest` library. The tests are located in the `tests/` directory, with the main test file being `test_dual.py`. This file includes comprehensive unit tests to validate the functionality of the `Dual` class, covering initialization, arithmetic operations (addition, subtraction, multiplication, division), power, trigonometric functions (`sin`), exponential (`exp`), logarithmic (`log`), square root (`sqrt`), and negation. All dual-to-dual, dual-to-scalar, and scalar-to-dual operations are thoroughly tested. Below are the tests to test addition.

```python
import unittest
from dual_autodiff.dual import Dual
import math

class TestDual(unittest.TestCase):

    def test_initialization(self):
        x = Dual(2.0, 1.0)
        self.assertEqual(x.real, 2.0)
        self.assertEqual(x.dual, 1.0)

    def test_addition_dual_dual(self):
        x = Dual(2.0, 1.0)
        y = Dual(3.0, 2.0)
        z = x + y
        self.assertEqual(z.real, 5.0)
        self.assertEqual(z.dual, 3.0)

    def test_addition_dual_scalar(self):
        # skip for brevity

    def test_addition_scalar_dual(self):
        # skip for brevity

    def test_sin(self):
        x = Dual(math.pi/2, 1.0)
        y = x.sin()
        self.assertAlmostEqual(y.real, math.sin(math.pi/2))
        self.assertAlmostEqual(y.dual, math.cos(math.pi/2))
```

Figure 4: Unit Test Code for the Dual Class

4 Implementation of `dual_autodiff_x`

This section corresponds to Question 8.

4.1 Structure Overview

```
dual_autodiff_x/
|-- wheelhouse/
|-- __init__.py
|-- dual.c
|-- dual.cp39-win_amd64.pyd
|-- dual.pyx
|-- pyproject.toml
|-- setup.py
```

Figure 5: Project Repository Structure for `dual_autodiff_x`

- `dual.pyx`: Cython source file defining the `Dual` class and its operations.
- `setup.py`: Script to build and install the Cythonized extension module.
- `dual.c`: C file generated by Cython from the `dual.pyx` source file.
- `dual.cp39-win_amd64.pyd`: Compiled Python extension module for Windows 64-bit systems (Python 3.9).
- `wheelhouse/`: Directory containing built distribution wheels for the package(will be discussed in section 7.1).

4.2 Cython `Dual` Class

This class defines all the methods described in Section 3.2 for the Python `Dual` class, but with Cython and Python syntax differences. Below is the definition of the initialization, addition, and `sin` methods in Cython, which are functionally consistent with the Python version.

```
from libc.math cimport sin, cos

cdef class Dual:
    cdef double _real
    cdef double _dual

    def __init__(self, double real, double dual=1.0):
        self._real = real
        self._dual = dual

    @property
    def real(self):
        return self._real

    @real.setter
    def real(self, value):
        self._real = value

    # @property and @dual.setter skipped for brevity

    def __add__(self, other):
        if isinstance(other, Dual):
            return Dual(self.real + other.real, self.dual + other.dual)
        elif isinstance(other, (float, int)):
            return Dual(self.real + other, self.dual)

    def __radd__(self, other):
        return self.__add__(other)

    cpdef sin(self):
        cdef double sin_real = sin(self.real)
        cdef double cos_real = cos(self.real)
        return Dual(sin_real, cos_real * self.dual)
```

Figure 6: Initialization and Method Definitions for the Cython Dual Class (error handling is omitted)

It can be observed from Fig.2 and Fig.5 that there are three main differences between the two versions of the `Dual` class:

- `cdef`: In the Cython version, `cdef` is used to define the class and member variables with explicit types (e.g., `cdef double _real`), avoiding the performance loss caused by Python's dynamic typing.
- `@property` **and** `@setter`: These decorators are required in Cython to maintain Python-style property access while optimizing performance.
- `cpdef`: `cpdef` enables methods to be called both from Python and Cython, with Cython achieving better performance.

5 Comparison: `dual_autodiff` vs. `dual_autodiff_x` vs. `SciPy`

This section corresponds to Questions 5 and 9.

The function of interest is:

$$f(x) = \ln(\sin(x)) + x^2 \cos(x) \tag{11}$$

The analytical derivative is:

$$f'(x) = \frac{\cos(x)}{\sin(x)} + 2x\cos(x) - x^2\sin(x) \tag{12}$$

5.1 Derivative of $f(x) = \ln(\sin(x)) + x^2\cos(x))$

In this section, the derivative of the function $f(x)$ at $x = 1.5$ is computed using numerical differentiation with the central difference method. This method provides second-order accuracy, meaning the error scales as $O(h^2)$ [**?**], which makes it more accurate than forward or backward differences. The formula for the central difference method is given by:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{13}$$

This method strikes a balance between performance and computational cost. Numerical differentiation is performed using various step sizes and the results is as follows:

Step Size h	**Error**
10^{-10}	-1.5191×10^{-7}
10^{-9}	-1.7967×10^{-7}
10^{-8}	9.0704×10^{-9}
10^{-7}	-1.1992×10^{-9}
10^{-6}	2.1633×10^{-10}
10^{-5}	-8.7597×10^{-11}
10^{-4}	-7.0574×10^{-9}
10^{-3}	-7.0578×10^{-7}
10^{-2}	-7.0577×10^{-5}
10^{-1}	-7.0412×10^{-3}

Table 1: Error of numerical derivative at different step sizes.

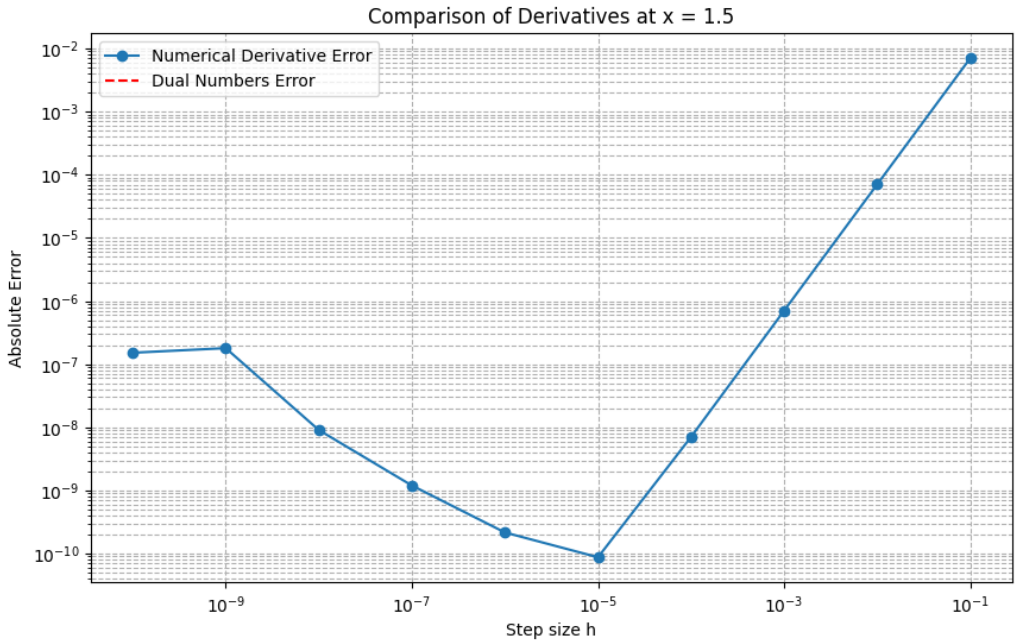

Figure 7: Error of numerical derivative vs. step size h (loglog scale)

It can be observed from Fig. 7 that the error at the smallest step size $h = 10^{-10}$ is not as small as expected from the truncation error $O(h^2)$. The error reaches its minimum at $h = 10^{-5}$, which represents the optimal step size, and then steadily increases for larger step sizes up to 10^{-1}. The behavior can be explained as follows:

- **Truncation Error:** Truncation error arises from the approximation in numerical methods, such as the central difference formula. For this method, the truncation error scales as $O(h^2)$. When h is large, truncation error dominates and decreases rapidly as h reduces.
- **Rounding Error:** Rounding error originates from finite precision in floating-point arithmetic, particularly in computing $f(x+h) - f(x-h)$. When h is extremely small, the two values are very close, leading to significant loss of precision. Rounding error scales as $O\left(\frac{1}{h}\right)$ and becomes dominant at small h.

For large h, truncation error dominates, causing the total error to decrease as h reduces. For very small h, rounding error dominates, causing the total error to increase.

5.2 Performance Comparsion

In this subsection, the performance of three methods for computing the derivative of $f(x) = x^2 \sin(x)$ at $x = 1.5$ is evaluated. The methods include:

- `dual_autodiff`: Python implementation of dual numbers.
- `dual_autodiff_x`: Cython-optimized implementation of dual numbers.
- `SciPy`: Numerical differentiation using the `approx_fprime` function, which is more general compared to the previously used central difference method.

The performance is measured using the `timeit` function, with the results averaged over multiple evaluations. The table and plot below summarize the time per evaluation and corresponding errors for each method.

The performance is measured using the `timeit` function, with the results averaged over multiple evaluations. The table and plot below summarize the time per evaluation and corresponding errors for each method. The value 1.49×10^{-9} represents the square root of machine precision, a small number that is numerically stable this task.

Method	**Error**	**Time per Evaluation**
`dual_autodiff` (Python)	0.0000	1.24 µs ± 17.1 ns
`dual_autodiff_x` (Cython)	0.0000	549 ns ± 15.4 ns
`SciPy` ($h \approx 1.49 \times 10^{-8}$)	8.03×10^{-11}	38.5 µs ± 1.49 µs

Table 2: Error and time benchmarking results for Python, Cython, and SciPy implementations.

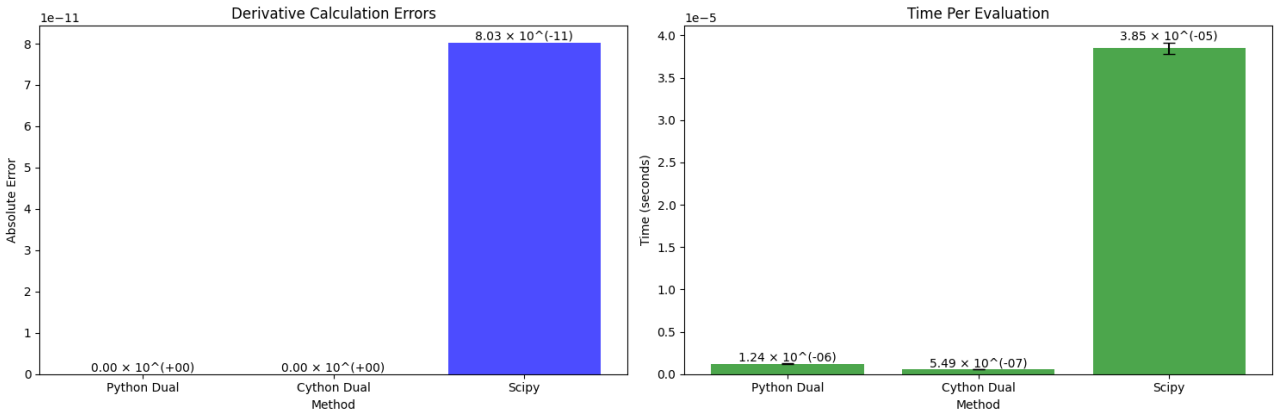

Figure 8: Error and time taken for different methods

From Table 2 and Fig. 8, it is clear that the `dual_autodiff_x` (Cython implementation) significantly outperforms the other methods in terms of both execution time and accuracy, achieving the fastest computation. Both `dual_autodiff` and `dual_autodiff_x` produce zero error at this level of precision, demonstrating their accuracy.

In terms of execution time, `dual_autodiff_x` is approximately 2.2 times faster than `dual_autodiff`. Compared to the `SciPy` method, `dual_autodiff` achieves a speedup of 30 times, while `dual_autodiff_x` achieves a remarkable 70 times improvement, making it the most efficient approach among the three.

6 Documentation with Sphinx

This section corresponds to Question 7.

```
docs/
|-- build/
|   |-- doctrees/
|   |-- html/
|-- source/
|   |-- conf.py
|   |-- dual_autodiff.ipynb
|   |-- index.rst
|   |-- modules.rst
|-- make.bat
|-- Makefile
```

Figure 9: Directory Structure for docs

The project uses `Sphinx` for automated documentation generation, integrating source code, module explanations, and examples. Key configurations in `conf.py` include:

- **autodoc**: Automatically generates documentation from docstrings.
- **napoleon**: Supports Google Style docstrings[2](Implemented in this work).
- **nbsphinx**: Embeds Jupyter Notebook tutorials as documentation content.

The documentation generates clear HTML outputs that include Jupyter Notebook content.

7 Wheels for Linux and Docker

This section corresponds to Questions 10 and 11.

7.1 Wheels

In this work Wheels are used to distribute Python packages as pre-compiled binaries.

The directory structure of this subsection corresponds to the `wheelhouse` folder within the structure presented in Fig. 5.

```
wheelhouse/
|-- dual_autodiff_x...cp310...manylinux2014_x86_64.whl
|-- wheel_contents/
|   |-- cp310/
|   |   |-- dual_autodiff_x/
|   |   |   |-- __init__.py
|   |   |   |-- dual.cpython-310-x86_64-linux-gnu.so
|   |   |   |-- setup.py
|   |   |-- dual_autodiff_x-0.1.0.dist-info/
|   |       |-- METADATA
|   |       |-- RECORD
|   |       |-- top_level.txt
|   |       |-- WHEEL
```

Figure 10: Directory Structure of Generated Wheel Contents (some content omitted and file name shortened for brevity)

The key files worth highlighting are:

- `dual_autodiff_x ...cp310...manylinux2014_x86_64.whl`: Pre-compiled wheel file for Linux systems (Python 3.10), build from `cibuildwheel`, containing the built package and its dependencies. This file is designed to be installed within a Docker environment or other Linux-based systems.
- `cp310`: Contains the unzipped content of the `.whl` file, including the compiled shared object file and metadata required for package installation.

7.2 Docker

In this work, Docker is used to provide a consistent, isolated, and reproducible environment for building and testing software.

```
dual_autodiff_docker/
|-- Dockerfile
|-- dual_autodiff_x_image.tar
|-- dual_autodiff_x...-cp310-...-manylinux2014_x86_64.whl
|-- dual_autodiff_x...-cp311-...-manylinux2014_x86_64.whl
```

Figure 11: Directory Structure of the Docker and Wheel Files (file names shortened for brevity)

The key files worth highlighting are:

- `Dockerfile`: Defines the instructions to build the Docker image, ensuring a consistent environment for compiling and testing the `dual_autodiff_x` package.
- `dual_autodiff_x_image.tar`: A pre-configured Docker image that contains the fully set up environment, including the installed `dual_autodiff_x` package.

8 Summary

This report demonstrates the implementation and performance analysis of the `dual_autodiff` package for automatic differentiation using dual numbers. The pure Python version and its Cython-optimized counterpart (`dual_autodiff_x`) were compared, highlighting significant performance improvements in the Cython version. Benchmarks showed that `dual_autodiff_x` achieves a speedup of over 2 times compared to the Python version and up to 70 times compared to SciPy's numerical differentiation. Both implementations compute derivatives exactly, matching analytical results, while SciPy's numerical approach introduces small errors and suffers from rounding limitations.

To ensure reproducibility, Docker was employed to create a consistent and isolated build environment, while wheels were used for efficient package distribution.

References

[1] Virtanen, P., Gommers, R., Oliphant, T. E., et al., *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*, Nature Methods, 17, pp. 261–272, 2020.

[2] Google, *Google Python Style Guide*, Available at: `https://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings`, (Accessed: 18.12.2024).

[3] Messelmi, F., *Analysis of Dual Functions*, Annual Review of Chaos Theory, Bifurcations and Dynamical Systems, Vol. 4, pp. 10-12, 2013.

Appendix

Function	Expression	Result
`sin()`	$\sin(a+b\epsilon)$	$\sin(a)+\cos(a)\cdot b\epsilon$
`cos()`	$\cos(a+b\epsilon)$	$\cos(a)-\sin(a)\cdot b\epsilon$
`tan()`	$\tan(a+b\epsilon)$	$\tan(a)+\frac{b}{\cos^2(a)}\epsilon$
`exp()`	$\exp(a+b\epsilon)$	$\exp(a)+\exp(a)\cdot b\epsilon$
`log()`	$\log(a+b\epsilon)$	$\log(a)+\frac{b}{a}\epsilon$
`__pow__()`	$(a+b\epsilon)^{c+d\epsilon}$	$a^c+a^c\left(c\cdot\frac{b}{a}+d\ln(a)\right)\epsilon$

Table 3: Mathematical Formulas of Dual Number Functions Defined in the `Dual` Class[3]

Declaration of Auto-Generation Tool

Based on the original text and instructions provided by me, some content in this report was generated by ChatGPT 4o. The tool also assisted in formatting and refining the LaTeX code. All analysis and interpretations remain my own.