# Automatic Differentiation with Dual Numbers

Computing Research for Data Intensive Science

mem97

December 2024

# Contents


**1 Introduction** **1**

**2 Background** **1**
2.1 Dual Numbers & Algebraic Properties . . . . . 1
2.1.1 Addition & Subtraction . . . . . 1
2.1.2 Multiplication & Division . . . . . 1
2.1.3 Power . . . . . 2
2.2 Automatic Differentiation with Dual Numbers . . . . . 2
2.3 Existing Solutions . . . . . 3

**3 Repository Overview** **3**
3.1 Overview of the Python & Cython Packages . . . . . 3
3.1.1 docs . . . . . 3
3.1.2 src . . . . . 4
3.1.3 wheelhouse & wheel_contents . . . . . 5
3.1.4 Other files . . . . . 5
3.2 Structural Difference Between `dual.py` & `dual.pyx` . . . . . 6

**4 Results & Examples** **6**
4.1 Validation . . . . . 6
4.1.1 Simple example . . . . . 6
4.1.2 Test Suit . . . . . 8
4.2 Examples & Applications . . . . . 9
4.3 Performance . . . . . 13

**5 Conclusion** **16**

# Acknowledgment

I would like to acknowledge the invaluable assistance of ChatGPT-4 in the preparation of this project. The tool played a significant role in:

- Debugging and resolving coding challenges encountered during the development of the Python package.
- Providing insights and suggestions for improving the structure and functionality of the code.
- Generating concise and accurate summaries of complex texts to enhance understanding and clarity.

While ChatGPT-4 contributed significantly to streamlining the development process and improving the quality of outputs, all results were rigorously reviewed, tested, and refined to ensure their accuracy, relevance, and alignment with project objectives. The use of this tool reflects a commitment to leveraging innovative technologies for efficient problem-solving and learning.


**Word Count :** 3600

# 1 Introduction

This report presents the development and implementation of a Python package, dual_autodiff, designed to perform automatic differentiation using dual numbers. Automatic differentiation is a cornerstone of modern computational techniques, particularly in machine learning and numerical optimization. This project involves creating a dual number framework in Python, Cythonising the package for performance improvement, and validating the implementation through rigorous testing and comparisons.

# 2 Background

The aim of this section is to provide a basic understanding of dual numbers and how they can be applied for automatic differentiation.

## 2.1 Dual Numbers & Algebraic Properties

Dual numbers[1, 2] are a hypercomplex number system, these are expressed in the form $a + \varepsilon b$, where $a, b \in \mathbb{R}$ , and $\varepsilon$ is the infinitesimal unit which satisfy $\varepsilon^n = 0$, for $n \geq 2$, $n \in \mathbb{N}$, and with $\varepsilon \neq 0$. From this definition, one can derive the following algebraic properties of dual numbers.
Let us define $x$ and $y$ as dual numbers, such that:

$$x = a + \varepsilon b \tag{1}$$

$$y = c + \varepsilon d \tag{2}$$

where $a, b, c, d \in \mathbb{R}$.

### 2.1.1 Addition & Subtraction

Adding dual numbers:

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

Subtracting dual numbers:

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

### 2.1.2 Multiplication & Division

Multiplying dual numbers:

$$xy = yx = ac + \varepsilon(ad + bc) \tag{5}$$

Dividing dual numbers:

$$\frac{x}{y} = \frac{a}{c} + \varepsilon \left( \frac{b}{c} - \frac{ad}{c^2} \right) \tag{6}$$

#### 2.1.3 Power

Dual number to the power of a dual number:

$$x^y = a^c \left[1 + \varepsilon \left(\frac{cb}{a} + \log(a)d\right)\right] \tag{7}$$

Dual number to the power of a real number:
Let $m \in \mathbb{R}$, then

$$x^m = a^m + \varepsilon m a^{m-1} b \tag{8}$$

Real number to the power of a dual number:
Let $m \in \mathbb{R}$, then

$$m^y = a^m[1 + \varepsilon \log(m)d] \tag{9}$$

All of the algebraic results provided are implemented in the dual.py, a Python file containing the Dual class for the package dual_autodiff (same applies for the cythonised version dual_autodiff_x).

### 2.2 Automatic Differentiation with Dual Numbers

Dual number can be applied for automatic differentiation, here is how. Consider any $n$ degree polynomial[3], $P$, whose domain is $x$;

$$P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n \tag{10}$$

Now, allowing $x$ to have a real and dual part, i.e. let $x = a + \varepsilon b$, then

$$P(a + \varepsilon b) = p_0 + p_1(a + \varepsilon b) + p_2(a + \varepsilon b)^2 + \cdots + p_n(a + \varepsilon b)^n$$

Using equation (8), one can expand the above polynomial, and then group terms together as follows;

$$\begin{aligned} P(a + \varepsilon b) &= p_0 + p_1 a + p_2 a^2 + \cdots + p_n a^n + \varepsilon(p_1 b + 2p_2 ab + \ldots np_n a^{n-1} b) \\ &= P(a) + \varepsilon b(p_1 + 2p_2 a + \ldots np_n a^{n-1}) \end{aligned}$$

This reduces to

$$P(a + \varepsilon b) = P(a) + \varepsilon b P'(a) \tag{11}$$

One can extend the same principle to general functions (real and analytical) using the Taylor series expansion[3],

$$f(a + \varepsilon b) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a) b^n \varepsilon^n}{n!} = f(a) + \varepsilon b f'(a) \tag{12}$$

This is a very powerful result that is at the base of automatic differentiation with dual numbers. If you notice, one can construct the domain of a function based on dual numbers such that $b = 1$ hence Eq.(12) reduces to

$$f(a + \varepsilon) = f(a) + \varepsilon f'(a) \tag{13}$$

Thus, for a real and analytical function defined on dual numbers, the real part of the function corresponds to the function itself evaluated at $a$, while the dual part represents the derivative of the function evaluated at $a$.

### 2.3 Existing Solutions

There are several packages that have adopted automatic differentiation, the most common and used are TensorFlow and JAX, both are used for machine learning, in particular for training neural networks, where methods like back-propagation, stochastic gradient descent (SGD), ADAM, and many more require derivative computation, making automatic differentiation a crucial tool for machine learning (For more information see TensorFlow and JAX).

## 3 Repository Overview

This section highlights the package structure and main features. The primary objective was to develop a Python package, **dual_autodiff**, for automatic differentiation using dual numbers. After completing the Python implementation, the package was optimized by converting it into Cython. Cythonising involves translating Python code into C or C++ extensions, improving performance through low-level optimisations and static typing. This process results in faster execution by compiling the code into machine-readable formats like `.so` or `.pyd` files, which can be seamlessly imported into Python.

Additionally, a numerical differentiation module was developed and also Cythonised to enhance its efficiency. The full package structure and details can be explored on the GitHub repository. Furthermore, wheels for Linux systems were created to facilitate easy installation.

Comprehensive documentation for the package was generated using Sphinx, a tool for creating well-structured and navigable documentation. Written in reStructuredText or Markdown, Sphinx converts documentation into formats like HTML, PDF, and ePub. For this project, HTML documentation includes all API details, enriched with meaningful examples and comments extracted from docstrings. While the repository is named **dual_autodiff** on GitHub, it is referred to as **dual_autodiff** consistently throughout this report.

### 3.1 Overview of the Python & Cython Packages

The repository containing the packages is divided into five directories, docs, src, wheel_contents and wheelhouse, and five files namely, .gitignore, LICENSE, README.md, pyproject.toml and setup.py. Now a quick overview of each file and directory including their objectives.

#### 3.1.1 docs

The directory docs contains the required files for automatically generate documentation, this consists of:

- The directory _build contains `html` and `doctrees`, generated using `make clean` (to remove old documentation) and `make html` (to build the documentation). These commands should be run in an active virtual environment within the `docs` directory (see details in the `README.md`).
- The `pag` directory includes documentation pages in `.rst` format, such as `introduction.rst`, `quick_start.rst`, and `api_reference.rst`. These plain-text files are used for structured documentation.

- `Makefile` automates tasks like building or cleaning documentation with rules for compilation and dependencies.
- `index.rst` serves as the main file for organizing documentation pages.
- `dual_autodiff.ipynb` is a Jupyter Notebook with examples and tutorials for using `dual_autodiff` and `dual_autodiff_x`, including exercises.
- `Solutions.ipynb` provides answers to exercises in `dual_autodiff.ipynb`.
- `conf.py` configures Sphinx with metadata, extensions, output formats (e.g., HTML), and paths for source files and templates.

The theme adopted in the documentation is from Read the Docs.

### 3.1.2 src

The directory `src` contains the packages `dual_autodiff` and `dual_autodiff_x` (Python and Cython respectively).

| dual_autodiff | dual_autodiff_x | tests |
|---|---|---|
| __init__.py | __init__.py | __init__.py |
| dual.py | dual.pyx | test_dad.pyx |
| n_diff.py | n_diff.pyx | test_ndiff.pyx |
| version.py | version.py | / |

Table 1: This table aims to illustrate what files the directories `dual_autodiff`, `dual_autodiff_x` and `tests` contain.

- `__init__.py`: Marks directories as Python packages for import.
- `dual.py` and `dual.pyx`: Contain the `Dual` and `DualX` classes, which perform dual number algebra and support operations like trigonometric, hyperbolic, exponential, and logarithmic functions. They accept inputs as `int`, `float`, or `array`.
- `n_diff.py` and `n_diff.pyx`: Define the `NumDiff` and `NumDiffX` classes for numerical differentiation. They support methods like forward, central, backward difference, and second-order differentiation.
- `version.py`: Manages the package or project version.
- `test_dad.py` and `test_ndiff.py`: Run test suites for `Dual` and `NumDiff` classes to validate functionality.
- `__init__.py`: Converts test files into packages, enabling test suite execution in Jupyter Notebooks or via terminal commands like `pytest-s tests/*`.

Please note, there is not test suits for the Cython packages, this is beyond the scope of this project.

### 3.1.3 wheelhouse & wheel_contents

The **wheelhouse** is the directory where build wheel files (`.whl`) are stored. When using tools like `cibuildwheel, pip wheel`, or other Python packaging tools, the generated wheels, in most cases, will be automatically stored in the directory **wheelhouse** by convention. The main purposes of this directory is to store compiled wheels for distribution or installation and to provide a centralised location for all binary file for a package.

The **wheel_contents** is a directory, containing compiled files (in this project `.so` files), automatically generated when "unzipping" the wheelhose directory. Evidence of wheels working:

```
root@T800:/mnt/c/Users/mmanc# cd /root
root@T800:~# mkdir wheels_testing
root@T800:~# cd /root/wheels_testing/
root@T800:~/wheels_testing# git clone
https://gitlab.developers.cam.ac.uk/phy/data-intensive-science-
mphil/assessments/c1_coursework1/mem97.git
Cloning into 'mem97'...
Username for 'https://gitlab.developers.cam.ac.uk': mem97
Password for 'https://mem97@gitlab.developers.cam.ac.uk':
remote: Enumerating objects: 441, done.
remote: Counting objects: 100% (441/441), done.
remote: Compressing objects: 100% (289/289), done.
remote: Total 441 (delta 181), reused 374 (delta 139), pack-reused 0
(from 0)
Receiving objects: 100% (441/441), 8.77 MiB | 13.21 MiB/s, done.
Resolving deltas: 100% (181/181), done.
root@T800:~/wheels_testing# python3 -m venv new_env
root@T800:~/wheels_testing# source new_env/bin/activate
(new_env) root@T800:~/wheels_testing# cd mem97
(new_env) root@T800:~/wheels_testing/mem97# pip install
wheelhouse/dual_autodiff_x-0.1.0-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl
Processing ./wheelhouse/dual_autodiff_x-0.1.0-cp312-cp312-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl
Installing collected packages: dual-autodiff-x
Successfully installed dual-autodiff-x-0.1.0
(new_env) root@T800:~/wheels_testing/mem97# python
```

**Note!** The wheels were generated with Python 3.12, this is not compatible with CSD3 Python version, hence I had to test it on WSL. To generate wheels Docker was not implemented, wheels were directly generated in a Linux system (i.e. by using `wheel` instead of `cbuildwheel`.)

### 3.1.4 Other files

`README.md` provides an introduction and essential details about the repository.
`LICENSE` specifies usage, modification, and distribution terms, with this project using the `UNILICENSE` (see unilicense.org).
`.gitignore` lists files and directories for Git to ignore, such as temporary files or build

artifacts (e.g., adding *.pyx prevents tracking of such files; see `.gitignore`).
`pyproject.toml`, introduced in PEP 518, defines build system requirements and settings for Python projects, standardizing metadata and dependencies (see `pyproject.toml`).
`setup.py` manages project metadata, dependencies, and build instructions, acting as a core file for Python packaging with tools like `setuptools` (see `setup.py`).
Both `pyproject.toml` and `setup.py` ensure all dependencies are installed when running `pip install .` or `pip install -e .`, with `-e` enabling editable installations. Follow the repository setup instructions in the `README.md`.

## 3.2 Structural Difference Between `dual.py` & `dual.pyx`

The files `dual.py` and `dual.pyx` initialise the class `Dual` for the Python and Cython packages respectively. The Python and Cython implementations of the `Dual` class differ primarily in their focus on performance and usability. The Python version is simple, dynamically typed, and easy to understand, making it ideal for development, prototyping, and use cases where performance is not critical. In contrast, the Cython version introduces static typing (`cdef`) and compiles to C, which significantly enhances execution speed and reduces runtime overhead, making it suitable for performance-critical applications. Additionally, the Cython version uses properties for controlled attribute access, whereas the Python version directly accesses attributes without encapsulation. While the Python implementation is portable and runs on any standard interpreter, the Cython implementation requires a Cython compiler and additional setup. Ultimately, the Python version prioritizes readability and ease of use, while the Cython version focuses on efficiency and optimized numerical computation.

# 4 Results & Examples

In this section we want to demonstrate the accuracy of the implementation of the packages by comparing their results with known derivatives, discuss performance and potential application.

## 4.1 Validation

To demonstrate the packages accuracy and consistency, in the `dual_autodiff.ipynb` notebook, there are a full demonstration on how to use the packages from the simplest tasks, i.e. constructing a dual number, to more complex ones such as, returning the dual part of a function constructed with dual variables.

### 4.1.1 Simple example

Starting off with the simple task that one expects from both Python and Cython packages, input two numbers, output the numbers with allocation to real and dual part.

```python
# Importing required package

# Pure Python
from dual_autodiff import Dual
from dual_autodiff import NumDiff
```

```

# Cythonised
from dual_autodiff_x import DualX
from dual_autodiff_x import NumDiffX
```

Check if allocation task works correctly

```
x = Dual(2,1)
xc = DualX(2,1)

print('Pure Python:')
print(f'x = {x}')
print(f'The real part of x is {x.real}')
print(f'The dual part of x is {x.dual}')
print('')
print('Cythonised:')
print(f'xc = {xc}')
print(f'The real part of xc is {xc.real}')
print(f'The dual part of xc is {xc.dual}')
```

Output:

```
Pure Python:
x = Dual(real = 2, dual = 1)
The real part of x is 2
The dual part of x is 1

Cythonised:
xc = Dual(real = 2, dual = 1)
The real part of xc is 2
The dual part of xc is 1
```

From the above output it is clear that both packages are executing the simple task correctly.

### Division and Power example

Let $x = 2 - \varepsilon 5$ and $y = 4 + \varepsilon 7$, our objective is to compute $x/y$ and $y^x$ coding equations (6, 7) in Python, use the result as a benchmark to compare and the packages output when executing the same tasks.

```
import numpy as np
# Defining eq. (6)
def dual_div(a,b,c,d):
    return f"real={a/c}, dual={b/c - (a*d)/(c**2)}"

# Defining eq. (7)
def dual_pow(a,b,c,d):
    return f"real={a**c}, dual={a**c*(c*b/a + np.log(a)*d)}"

# Example with a=2, b=-5, c=4 and d=7
```

```python

# Computation with the above functions

div_res = dual_div(2,-5,4,7)
pow_res = dual_pow(4,7,2,-5)

# Using Dual and DualX packages
# Assignig vales
x,y=Dual(2,-5),Dual(4,7) # Python
xc,yc=DualX(2,-5),DualX(4,7) # Cython

# Printing results and comparing
print('Division Results (x/y):')
print('Result using eq 6', div_res)
print('Python', x/y)
print('Cython', xc/yc)
print('')
print('Power Results (y**x):')
print('Result using eq 7', pow_res)
print('Python', y**x)
print('Cython', yc**xc)
```

Output:

```
Division Results (x/y):
Result using eq 6 real=0.5, dual=-2.125
Python Dual(real = 0.5, dual = -2.125)
Cython Dual(real = 0.5, dual = -2.125)

Power Results (y**x):
Result using eq 7 real=16, dual=-54.90354888959125
Python Dual(real = 16, dual = -54.90354888959125)
Cython Dual(real = 16, dual = -54.90354888959125)
```

The results provided from the defined functions `dual_div` and `dual_pow` and the two packages are the same. Unfortunately, testing every single function in the packages cannot be presented in this report. However, the reader is strongly advised to use the notebooks available and do the exercises.

### 4.1.2 Test Suit

As mentioned in the previous section, there is a test suit for the `Dual` and `NumDiff`. To run the test suit on your bash terminal (make sure you are in the right directory (`src`) and you virtual environment is active), execute the command `pytest-s tests/*`.

```
(dad_venv_2) root@T800:~/Document/C1_Cw/mem97# cd src
(dad_venv_2) root@T800:~/Document/C1_Cw/mem97/src# pytest -s tests/*
======================== test session starts ========================
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /root/Document/C1_Cw/mem97
configfile: pyproject.toml
```

```
collected 42 items

tests/__init__.py .....................
tests/test_dad.py .....................
tests/test_ndiff.py ...................

====================== 42 passed in 0.39s ========================

(dad_venv_2) root@T800:~/Document/C1_Cw/mem97/src#
```

Alternatively, on the Jupyther Notebook

```python
import ipytest
from tests import TestDual, TestNumDiff

TestDual
TestNumDiff
ipytest.run()
```

Output:

```
=========================== test session starts ===========================
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /root/Document/C1_Cw/mem97
configfile: pyproject.toml
collected 19 items

t_3e6bf4223a4e450cbc4d5a068d47bc83.py ...................           [100%]

=========================== 23 passed in 0.09s ===========================
```

**Note!** The output number of successful tests displayed is different from the test executed in the terminal and the test executed in the notebook, this is because in the terminal the test output displays each `assert`[1] in the classes `TestDual` and `TestNumDiff`, i.e. 43 `assert` overall. While the test executed in the notebook, displays the number of functions in each class that passed the test, i.e. 4 functions in `TestNumDiff` and 19 functions in `TestDual`.

## 4.2 Examples & Applications

Having demonstrated the package validation, it is now time to apply it for automatic differentiation. To perform automatic differentiation using the classes `Dual` and `DualX`, we are going to present a simple example:

Consider the function $f(x) = \sin x + \sinh x$, we want to compute the derivative of $f(x)$ at $x = 2$:

[1] In Python, `assert` is a statement used for debugging that tests whether a condition is `True`. If the condition evaluates to `False`, it raises an `AssertionError` with an optional error message.

```
# Initialise variable
# Set dual part equal to 1, set real part x.real=2, differentiating at x=2
x=Dual(2,1) # Python
xc=DualX(2,1) # Cython

# Contruct function with dual variables
f_x = x.sin() + x.sinh() # Python
f_xc = xc.sin() + xc.sinh() # Cython

# Computing derivatives
analytical_der = np.cos(2) + np.cosh(2)

# Printing results
print('Derivative of f(x) at x=2:')
print(f'Analytical result df/dx={analytical_der}')
print(f'Automatic derivative with Python df/dx={f_x.dual}') # return dual
# part for derivative
print(f'Automatic derivative with Cython df/dx={f_xc.dual}')
```

Output:

```
Derivative of f(x) at x=2:
Analytical result df/dx=3.346048854536489
Automatic derivative with Python df/dx=3.346048854536489
Automatic derivative with Cython df/dx=3.346048854536489
```

If the user wants to visualise the derivative over a range of values, this is how is done:

Consider the same function as before, $f(x) = \sin x + \sinh x$, instead of computing the derivative at a single point, we now want to compute it over a range of $x$-values, and plot the results. Output:

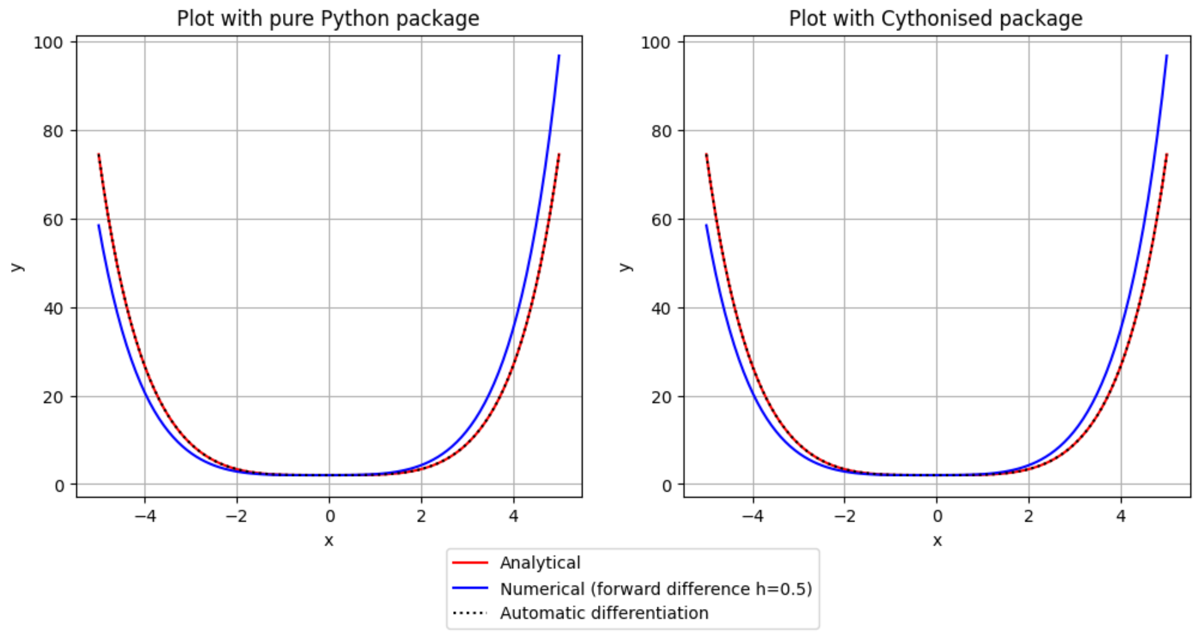


Figure 1: Plot, analytical derivative, automatic derivative, and Numerical derivative, using Python and Cython separately (see Notebook)

**Task 5** One of our task was for this project was to compute the derivative of the function $f(x) = \log(\sin(x)) + x^2 \cos(x)$ at the point $x = 1.5$, using three different methods, analytical derivative, numerical derivative and automatic differentiation, to then compare the results from all three methods.

By first computing the derivative of $f(x)$, for a fixed $h$-value (step size), and presented the results in a tabular format using `pandas`.

Results:

| **Method** | **Results** | **Ratio** | **Absolute Error** |
|---|---|---|---|
| Automatic Differentiation | -1.961237 | 1.000000 | 0.000000 |
| Forward Differentiation | -1.996346 | 1.017901 | 0.035109 |
| Central Differentiation | -1.961308 | 1.000036 | 0.000071 |
| Backward Differentiation | -1.926270 | 0.982171 | 0.034967 |

Table 2: Comparison of Differentiation Methods(code in the Notebook)

Plotting the ration over $x \in [1, 3]$:

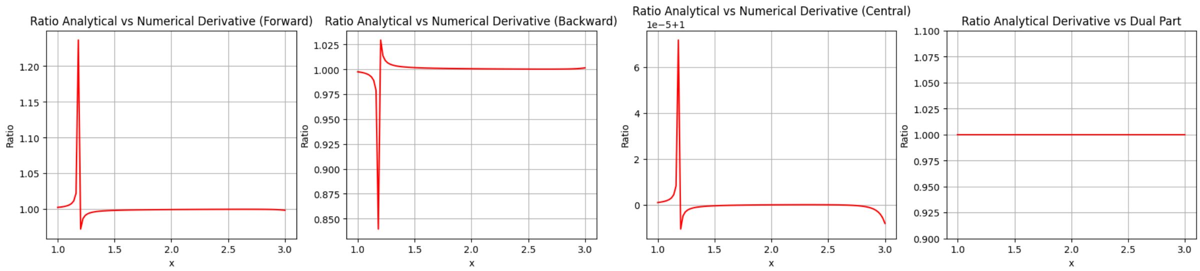


Figure 2: Shows difference in ratio, between analytical solution, and all the numerical methods including automatic differentiation. One can observe from the last plot on the right, that the automatic derivative is exact. (see code in Notebook)

One can observe from Figure 2, that the automatic differentiation with dual numbers is exact. The next plot is a `loglog`[2]-plot, where we are comparing the accuracy level of the numerical derivatives methods as the step size, $h$, value varies.


Figure 3: As one can expect, for all methods, as $h$ decreases so does the error (see code in Notebook). Note the error is evaluated on automatic differentiation using the class `Dual`.

### Comments on Figure 3

1. Forward Difference (Red):The error decreases linearly as h decreases, indicating first-order accuracy i.e. $\mathcal{O}h$.

[2] A log-log plot is a type of plot that uses logarithmic scales for both the x-axis and the y-axis. It is useful for visualizing relationships where both variables span several orders of magnitude, such as power-law relationships.

2. Central Difference (Purple): The error decreases at a steeper rater compared to the other two difference methods, showing second-order accuracy i.e. $\mathcal{O}h^2$.

3. Backward Difference (Blue): Similarly to the Forward Difference method, the error decreases linearly, indicating first-order accuracy i.e $\mathcal{O}h$.

**Comments**

- Central Difference is the most accurate method as it achieves faster error reduction with decreasing $h$.

- Forward Difference and Backward Difference are less accurate but exhibit comparable performance.

## 4.3 Performance

**Comparing Performance of the Pure Python Version and the Cythonised Version**

To compare performance between the pure Python vs the Cythonised versions of the package, we are going to use the module `timeit`.
The `timeit` module in Python is a standard library tool used to measure the execution time of small code snippets. It is particularly useful for benchmarking and identifying performance bottlenecks.

## Key Features of `timeit`

1. **Accurate Timing**:

   - `timeit` disables unnecessary collections and runs code multiple times to provide a more accurate measurement of execution time.
   - By default, `timeit` outputs an estimate of time per loop, including the mean time and its standard deviation.

2. **Easy to Use**:

   - You can measure the execution time of code directly in a script or interactively in the Python shell.

3. **Adaptive**:

   - You can control the number of repetitions and iterations to balance accuracy and performance.

For our purposes, we do not need to customize `timeit`; instead, we are going to use its default features. For more information on how to use `timeit`, please see the official documentation.

## Comparison Objectives

We want to compare the following performances:
`Dual` vs `DualX`
Aim to perform the comparison twice (this will also be b):

- Using the simplest function (i.e., returning a dual number).
- Using a more complex function (taking the dual and real part separately of a function).

After performing tests with 10 `runs` and $10^7$ `loops`, this is the output:

```
Simple Test
Python:
148 ns ± 3.99 ns per loop (mean ± std. dev. of 10 runs, 10,000,000 loops each)
Cython:
88.1 ns ± 2.02 ns per loop (mean ± std. dev. of 10 runs, 10,000,000 loops each)

Complex Test
Python:
10.1 s ± 65.3 ns per loop (mean ± std. dev. of 10 runs, 10,000,000 loops each)
Cython:
9.39 s ± 1.29 s per loop (mean ± std. dev. of 10 runs, 10,000,000 loops each)
```

This experiment was repeated for a fixed number of `runs` (default 7 runs), and a range of values for `loops`, i.e. `loops` $= [1000, 5000, 10000, 50000, 100000, 500000, 1000000]$.

Output:

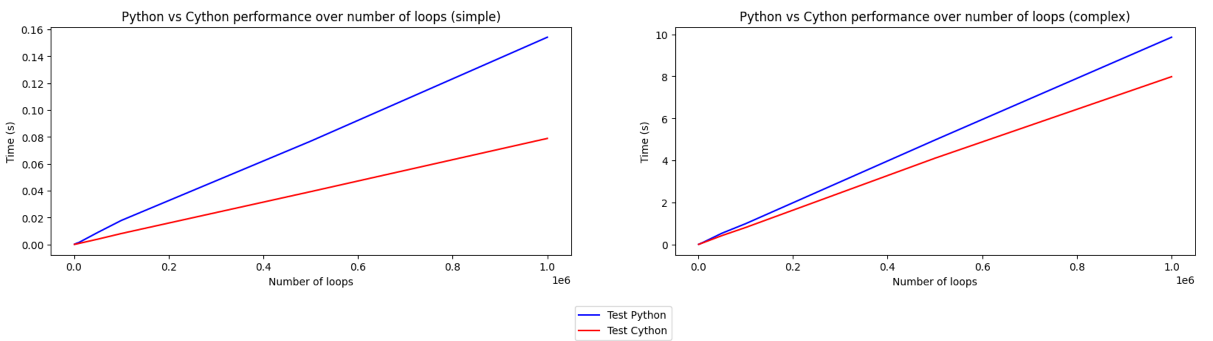


Figure 4: As one can expect, the Cythonised package is quicker that the pure Python (see code in Notebook). Note the error is evaluated on automatic differentiation using the class `Dual`.

# Performance Comparison Results

From the above results, one can notice that the Cythonised version of the package is faster for both tests (as one might expect). In the simple test, the Cythonised version is significantly faster than the pure Python version, but there is not much difference between the two for the complex test in terms of performance.

## Difference Between the Two Tests

**Simple Test**: In the simple test, we constructed a function that runs the simplest task. It takes two variables as input, which are allocated as the real and dual parts of a number, i.e., `Dual(a,b)` for the pure Python version and `DualX(a,b)` for the Cythonised version. Here, the Cythonised version is much faster.

**Complex Test**: In the complex test, we constructed a function that takes two variables as input and allocates them into the real and dual parts. There are two major differences between the complex and simple tests:

1. In the simple test, we allocate input into the real and dual parts. In the complex case, we allocate the input to a dual number $x$. Then we evaluate the function:

$$f(x) = \sin x + \cos x + x^2 e^x + \tanh x$$

with the dual number $x$.

2. After evaluating $f(x)$ at $x = a + \epsilon b$, the functions `test_dual_complex` and `test_dual_x_complex` return the real and dual parts separately. This is not a problem for the pure Python version as no major modifications were required to return `.real` and `.dual` separately. However, this caused some issues when Cythonising the package, i.e., Cython did not recognise `.real` and `.dual`. To overcome this problem, two extra functions were implemented using the `@property` decorator:

   - `def real(self)`: to return `.real`.
   - `def dual(self)`: to return `.dual`.

   This additional implementation might be one of the reasons why, for the complex test, the Cythonised version outperforms the pure Python version by only $0.71\,\mu s$ (this is the result obtained when running the code; the user may get a different answer). Furthermore, when defining input variables with `cdef`, `object` was utilised instead of `double`. This significantly slows down performance but allows more flexibility, e.g., accepting arrays as inputs.

## Difference Between `double` and `object`

`double`:

`double` represents a C-style double-precision floating-point number. This is used when high-performance numerical computation is required, and it works directly with C-style numeric types. `double` has a fixed size (usually 8 bytes) and does not involve Python's memory management, making it much faster than Python's `float` (which is an `object` type).

**Limitations of** `double`:

- Cannot handle Python's arbitrary-precision floats.
- Cannot be `None` or any other non-numerical value.
- Less flexible than `object`; for example, `double` cannot handle arrays.

`object`:

`object` represents a generic Python object and can hold any type of Python data (e.g., `int`, `float`, `list`, `str`, or custom objects). It is generally used for its flexibility, as it allows interaction with Python's dynamic typing system. However, in terms of memory usage, `object` is more expensive than `double` because it uses Python's heap memory and is managed by Python's garbage collector.

**Limitations of `object`:**

- Significantly slower than `double`.

# 5 Conclusion

The development of the dual_autodiff package successfully demonstrates the applicability of dual numbers in automatic differentiation, providing precise and efficient derivative computations. By implementing a Python-based solution and enhancing it with Cython, the project highlights the performance trade-offs between simplicity and computational speed.

Validation through examples and rigorous testing confirms the package's accuracy and robustness. Additionally, the performance analysis emphasizes the benefits of Cython for performance-critical applications. Packaging the project with wheels ensures compatibility and ease of distribution.

This work not only fulfils the project objectives but also lays a foundation for potential extensions in areas like neural network training and advanced numerical optimization tasks.

# References


[1] Vladimir Brodsky and Moshe Shoham. "Dual numbers representation of rigid body dynamics". In: *Mechanism and machine theory* 34.5 (1999), pp. 693–718.

[2] E Pennestrı and R Stefanelli. "Linear algebra and numerical algorithms using dual numbers". In: *Multibody System Dynamics* 18 (2007), pp. 323–344.

[3] Wikipedia contributors. *Dual number — Wikipedia, The Free Encyclopedia*. 2002. URL: https://en.wikipedia.org/wiki/Dual_number.