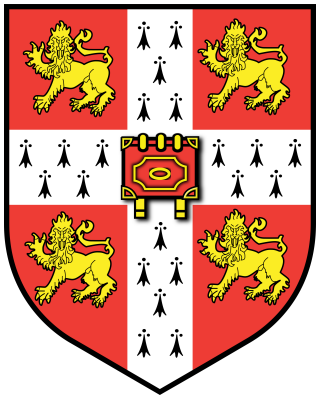
University of Cambridge

Mphil Data Intensive Science

Research computing

Automatic differentiation using dual numbers

Author:
Joshua Roberts



December 14, 2024

Word count: 2,941

1 Introduction

This report documents the design and development of the `dual_autodiff` Python package for the handling of dual numbers. The implementation and distribution of the package is discussed, following best practices in software development, to ensure a robust and maintainable code base. The report also discusses the applications of dual numbers in automatic differentiation and the implementation of Cython to aid computational performance.

2 Dual numbers

Dual numbers are a hypercomplex number system of the form

$$a + b\epsilon, \tag{1}$$

where ϵ satisfies the properties that $\epsilon^2 = 0$ with $\epsilon \neq 0$. The a term is referred to as the real component, while b is the dual component.

A full description of the mathematics behind dual numbers is beyond the scope of this work, but for further reference please see [1]. Common operations applied to dual numbers are summarised in Table 1.

Operation	**Dual result**
Addition $D_1 \pm D_2$	$(a \pm c) + (b \pm d)\epsilon$
Multiplication $D_1 D_2$	$ac + (ad + bc)\epsilon$
Division D_1 / D_2	$\left(\frac{a}{c}\right) + \frac{bc-ad}{c^2}\epsilon$
Dual to a dual power $D_1^{D_2}$	$a^c + a^c \ln(a)(ad + cb)\epsilon$
Dual to a real power D_1^n	$a^n + nba^{n-1}\epsilon$
Real to a dual power a^{D_2}	$a^c + a^c d \ln(a)\epsilon$
$\sin(a + b\epsilon)$	$\sin(a) + b\cos(a)\epsilon$
$\cos(a + b\epsilon)$	$\cos(a) - b\sin(a)\epsilon$
$\log(a + b\epsilon)$	$\ln(a) + \frac{b}{a}\epsilon$
$\exp(a + b\epsilon)$	$e^a + be^a\epsilon$

Table 1: Common operations and functions of dual numbers. Adapted from [1].

3 Development

The repository was built following good coding practices. Its structure is outlined in Figure 1.

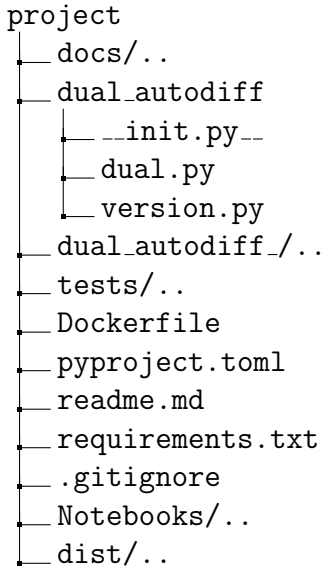


Figure 1: Directory structure of the project.

The core functionality of the package is contained within the `dual_autodiff` folder under the `dual.py` file. This defines the Dual class and its corresponding methods. The `dual_autodiff` folder also contains an `__init.py__` file used in Python to define packages and is automatically run when the package is installed [2]. The `version.py` file denotes the current version of the `dual_autodiff` package and is automatically generated by `setuptools_srm` when the package is built and extracts the current package version from git metadata [3]. This removes the need to constantly update static version variables throughout project development.

The dual_autodiff_x folder contains the source code and wheels for the Cython implementation of the dual_autodiff package. This is discussed in section 8.

The `tests` folder contains Python scripts for testing the accuracy of the `dual_autodiff` package following the pytest framework. This testing procedure is explained in section 6.

Other files in the repository include:

- Dockerfile: provides instructions for the creation of a Docker image, enabling one to run the code in an isolated environment, discussed further in 9.2
- Notebooks: Folder containing example Jupyter Notebooks showcasing the package
- `requirements.txt`: text file containing dependencies required to run the example notebooks.
- Docs: Folder containing the code to build the automated documentation. see 5
- Dist: Folder containing the Python wheels for the distribution of the package. see 9

Finally, there is also the `pyproject.toml` file. This file is integral to the building of the package, it defines build dependencies, package dependencies and metadata for the package. This ensures a robust and reproducible build process.

4 Implementation

In this work dual numbers are implemented via the Dual class, defined in the `dual.py` file within the `dual_autodiff` folder. The `__init__.py` file also imports the Dual class, allowing users to easily initialise a Dual object after importing the `dual_autodiff` package. This may be achieved via `dual_autodiff.Dual(a, b)` , which initialises a Dual object instance with a real component a and a dual component b.

The package's functionality is contained in the Dual object and its corresponding methods. Arithmetic operations were created using Python's dunder methods allowing standard operators (`+,-,*,/`) to be used between Dual objects. Trigonometric, logarithmic, and exponential operations were also implemented as methods that act on the current Dual instance with all methods following the logic defined in Table 1.

Each method includes extensive type checking and may raise custom exceptions if an unsupported or invalid type is used. This aids the end user in debugging their code and reduces the chance of error.

Dual number operations were also made compatible with Python integers or floats, where the scalar (integer or float) is treated as a dual number with a dual component of 0. Moreover, reverse operations are supported allowing for compatibility even when the scalar is on the left side of the operation. In such cases, Python searches for the relevant method in the scalar's class, and if not found invokes the corresponding reverse methods found in the Dual class.

This package also defines the dunder `__eq__` method to return True when comparing two dual numbers with identical real and dual components. It also evaluates as True when a Dual object with a zero dual component is compared to a scalar equal to the Dual object's real component. However, other comparison operations have been intentionally omitted from the class by design. Some implementations focus solely on the real part of the dual number for all comparison operations [4]. This means that `>, <` only compares the real parts of the two dual numbers. Such implementation increases the risk of end users misinterpreting their results by falsely assuming comparison accounts for both real and dual parts. Therefore these comparisons have not been included. Users may implement comparisons for their specific use case by manually accessing the real and dual components of the Dual objects they wish to compare.

5 Documentation

The Dual package code is documented in the NumPy style docstring format [5]. Documentation was generated via the Sphinx automated documentation tool [6]. Documentation covers use cases, error handling, and examples for all methods of the Dual class as well as an example notebook.

Documentation is typically hosted on the ReadtheDocs platform and a link is provided in the package repository, this allows users to easily access the documentation without the need to build it themselves [7]. However, due to the assessed nature of this work, hosting on ReadtheDocs has not been implemented. Users can generate the documentation locally by following these steps:

1. Clone the repository to their local machine.

2. Install the dependencies listed in the `requirements.txt` file located in the docs folder.

3. Execute the `make html` command from within the docs folder. This will build the documentation in the docs/build folder.

6 Testing

To ensure a robust implementation of the dual numbers package thorough testing was executed via the pytest testing framework [8]. This encompassed 23 separate testing functions across 4 files:

- test_arithmetic - Tested the basic arithmetic operations of the Dual class.
- test_functions - Evaluated various mathematical operations of the package such as trigonometric, exponential and powers.
- test_dual - Evaluated the initialisation, comparison and string representation of the Dual class.
- test_composite - Tested various components of the Dual class in combination to ensure correct implementation across the package. Included testing of numerous edge cases and nuanced scenarios created via generative AI to help ensure a thorough and robust test suit.

The testing suite also verified proper error handling of incorrect types and invalid inputs, confirming that the appropriate error is raised and aligns with the custom exceptions. This minimises potential errors for the end user and promotes robust testing practices.

Finally, a thorough and complete testing suite was ensured through the pytest-cov plugin, which identifies untested sections of the code base [9]. The Dual class achieved 100% test coverage emblematic of a thorough and complete testing procedure.

7 Differentiation

Efficient differentiation of functions is critical to numerous fields, particularly in machine learning where finding the gradient of loss functions is essential.

One approach to calculating the derivative of a function is numerical differentiation. This approach relies on evaluating the target function at

separate points, defining a polynomial that interpolates between these points and then using the gradient of this polynomial to approximate the derivative of the function [10]. One common example of numerical differentiation is the forward difference approach which approximates the derivative $f'(x)$ as

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

Where x represents the point at which the function is evaluated, while h denotes an arbitrary step size. Forward difference differentiation works best when h is small such that the function is approximately linear in the region of interest.

Analytical differentiation is a different approach that uses calculus techniques to derive a closed-form expression for the derivative. Once derived this expression for the derivative may be evaluated at chosen points, whose accuracy is only limited by the storage capabilities of the machine used for calculation.

Dual numbers may also be used to calculate the derivative of a function through a process known as automatic differentiation [11]. For some function $f(x)$ substitution of a dual number $a + b\epsilon$ yields the result

$$f(a + b\epsilon) = f(a) + f'(a)b\epsilon. \tag{3}$$

Therefore by choosing a to be the point of interest and $b = 1$, the result is a dual number whose real part is the function evaluated at a and whose dual part is the derivative of the function evaluated at a. This allows for the simultaneous evaluation of the function's value and derivative to machine-level accuracy.

To illustrate the benefit of using dual numbers in automatic differentiation consider the derivative of the function

$$f(x) = \log(\sin(x)) + x^2 \cos(x), \tag{4}$$

evaluated at $x = 1.5$. Table 2 compares the result of this derivative calculated via the 3 previously mentioned methods.

Differentiation methods	Derivative of f(x) at x=1.5
Analytical	-1.9612372705533612
Dual Number	-1.9612372705533612
Forward difference method	-1.996345877880931

Table 2: Derivative of f(x) evaluated at x=1.5, via an analytical, numerical and dual number approach, using a step size of 0.01

Table 2 shows how differentiation via dual numbers produces the same result as the analytical approach to machine-level accuracy. Moreover, Table 2 also highlights the loss of accuracy when calculating the derivative through the forward difference method.

The accuracy of the forward difference approach may be examined further by consideration of the method's accuracy as a function of step size, shown in Figure 2.

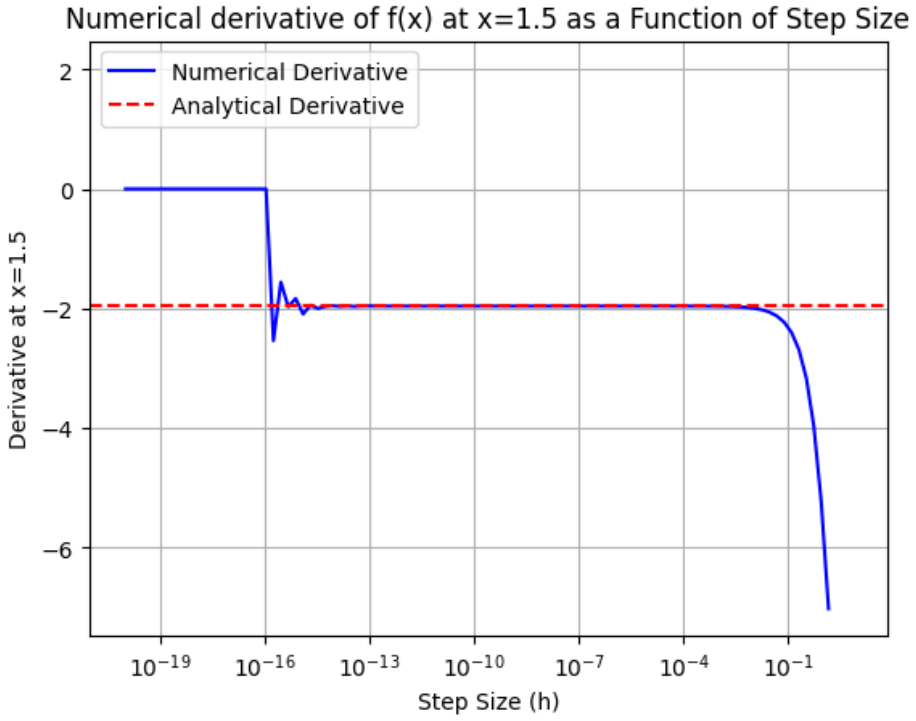


Figure 2: Derivative of f(x) evaluated using forward difference differentiation as a function of step size.

Figure 2 highlights the significance of choosing an appropriate step size for the numerical derivative calculation. When step sizes are too large, the numerical derivative diverges from the true value, while extremely small step sizes lead to significant oscillations before approaching zero. This behaviour may be better understood by considering the relative error of the numerical derivative as a function of step size on a log-log plot shown in Figure 3.

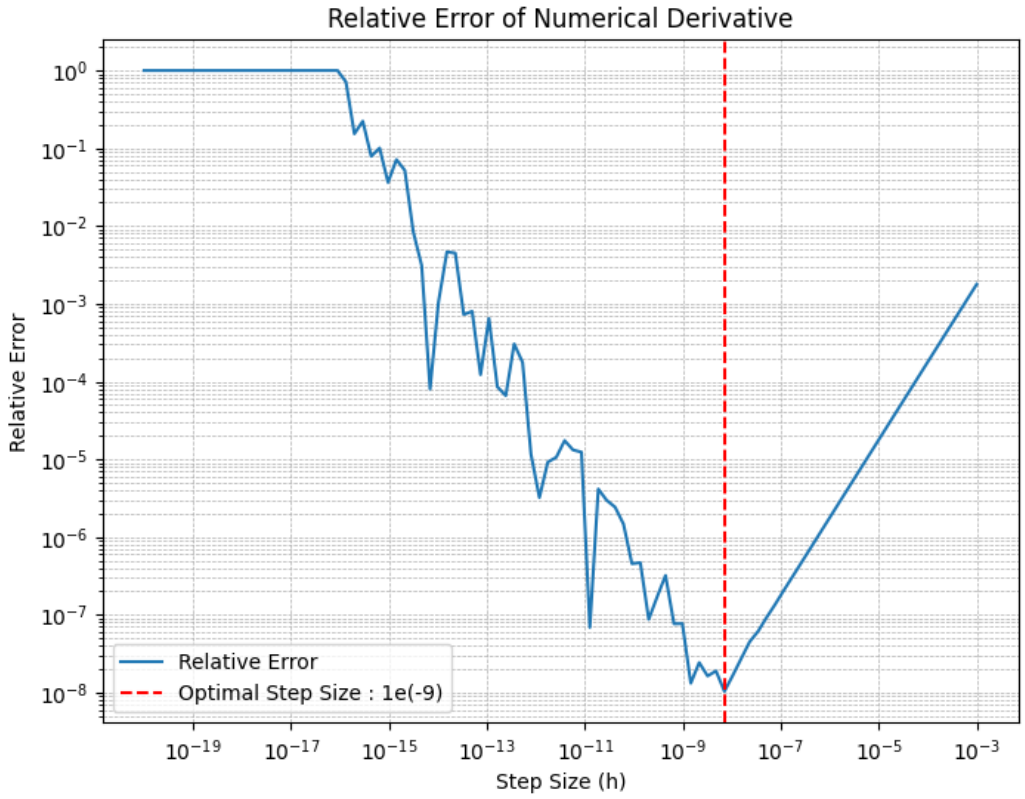


Figure 3: Relative error of numerical derivative from forward difference approach with respect to step size.

Figure 3 indicates two distinct regions of relative error with respect to step size. To the right of the optimal step size, truncation error is the dominant source of error associated with the numerical derivative. Truncation error is the error deriving from the forward difference assumption that the function is linear in the region from $f(x)$ to $f(x+h)$. When the step size grows large this assumption breaks down introducing increasingly larger errors [12].

As the step size gets smaller the dominant source of error instead derives from the limitations of the storage of floating point numbers. Computers have a finite memory and thus floating-point digits may only be stored to some finite precision with any further components being rounded off. When subtracting $f(x) - f(x+h)$ in the forward difference approach as h becomes smaller this difference becomes increasingly minute and the rounding error begins to become more significant [12]. Eventually $f(x)$ and $f(x+h)$ become indistinguishable within floating point precision and their difference becomes zero.

8 Cython

Cython is a superset of the Python programming language that compiles to C. Cython code looks extremely similar to Python however allows for static typing, reduces Python overhead and introduces compiler optimisations. This combines the ease of writing Python code with the performance increase of a compiled language [13].

8.1 Implementation

The `dual_autodiff` package has been Cythonised to form the Cython package `dual_autodiff_x`. The process used to cythonise the package was:

1. Copy the Python source code and `pyproject.toml` file into a new folder entitled `dual_autodiff_x`.
2. Convert the `.py` files to `.pyx` files.
3. Ensure Cython is installed and add it to the build system requirements in the `pyproject.toml` file.
4. Add a `setup.py` file to define the compilation settings for the package.
5. Build the package via `python setup.py build_ext --inplace` from within the `dual_autodiff_x` folder.

In the process of Cythonisation the Python code was converted to Cython however, no other alterations were made to the source code. This means that the full performance increase of Cython, such as improvements from static typing, could not be utilised.

8.2 Performance

The performance of the pure Python `dual_autodiff` package and the Cythonised `dual_autdiff_x` package was analysed using the timeit library [14].

The runtime of various methods of the Dual class was measured for the Python and Cython implementation. This was done by finding the total time to complete a loop of 1,000,000 calls to each method, this time was then averaged across 100 repeats and this average was then divided by 1,000,000 to find the average time to complete the method. As the methods are all relatively lightweight by timing over 1,000,000 loops the relative error of overheads associated with background processes on the machine is reduced.

The measured times for both the Python and Cython implementation are shown in Figure 4. Errors have been omitted as they were negligible in size.

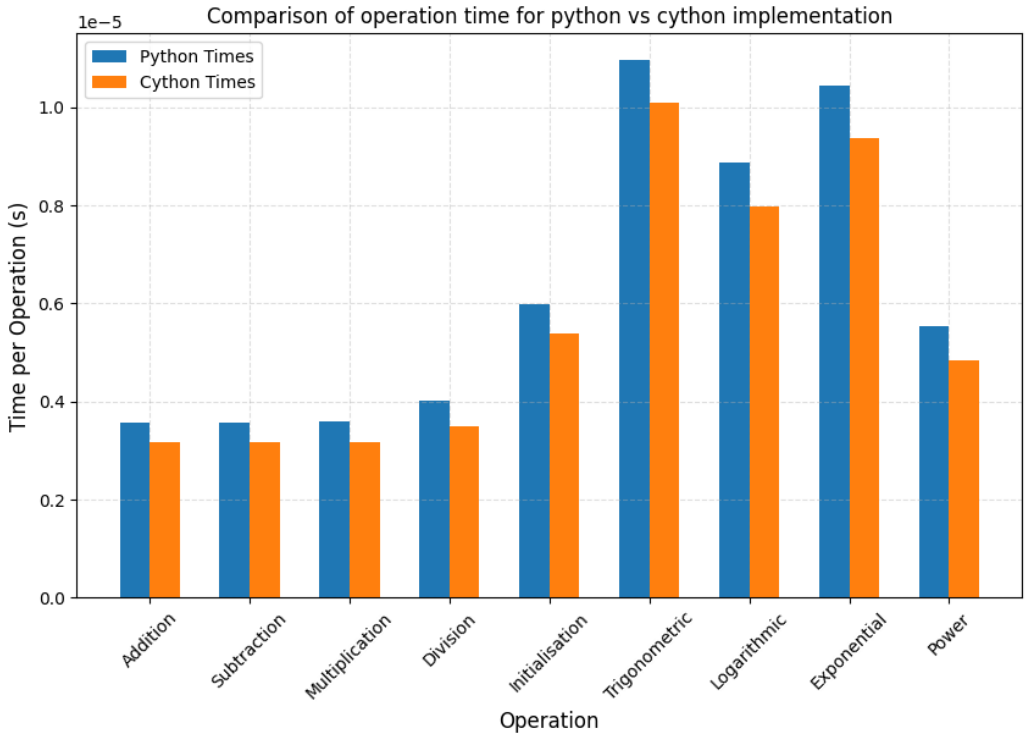


Figure 4: Comparison of average time taken for methods of the Dual class between Python and Cython implementation

Figure 4 demonstrates that for all methods Cython is quicker than the

Python implementation. This likely stems from Cython code being run through the Cython compiler resulting in faster execution.[15] Moreover, Figure 4 also illustrates that in both languages the arithmetic operations addition, subtraction and multiplication are close in time, with division taking slightly longer. This is likely due to the division being the most complex of the 4 arithmetic operations and the most difficult to parallelise thus taking longer to be calculated via the CPU [16]. Alternatively, the extra time may be a result of the additional checks in the division method ensuring the denominator is non-zero.

There is also the trend that the more complex functions such as trigonometric, logarithmic and exponential take greater time than arithmetic operations. This is to be expected due to their greater computational complexity.

To better visualise the increase in speed between Python and Cython consider Figure 5 showing the ratio between Python and Cython times for different operations.

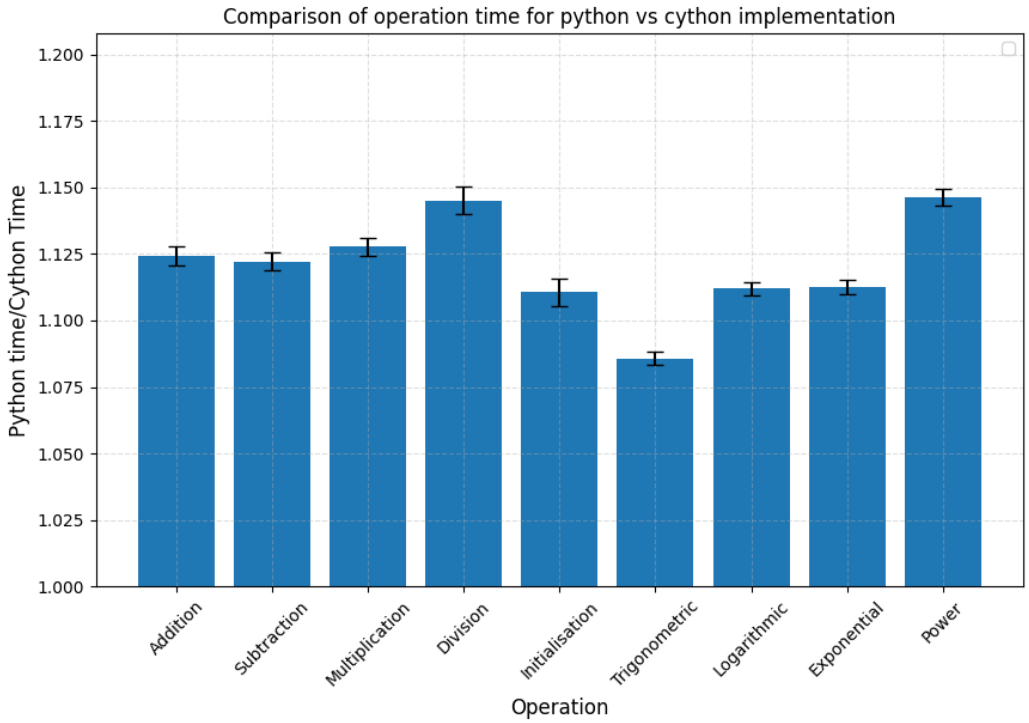


Figure 5: Ratios of average time taken for methods of the Dual class between Python and Cython implementations

Figure 5 shows the percentage increase in speed is only approximately 10%, a far lower performance increase than that reported in other projects [13]. This may be due to the Cython package source code mirroring that of the Python source code, meaning that the performance increase is coming from the Cython compiler efficiency, however, there is no benefit introduced from static type checking.

Moreover, due to the relative simplicity of these operations, there is no time saved from more advanced Cython optimisations such as loop unrolling or SIMD operations as is the case with loops and array calculations.

Furthermore in both implementations type checks were frequently used for error handling. This likely serves as a bottleneck to performance as such checks do not benefit from Cythonisation. Finally, Figure 5 demonstrates that trigonometric, logarithmic and exponential functions have the lowest performance increase. This stems from the fact that in Python these methods were implemented using the NumPy package, which itself is already compiled in C and thus little performance gain is to be expected from Cythonisation.

9 Package distribution and utilisation

Best practices in software development have been followed to ensure correct packaging and deployment of the `dual_autodiff` package.

9.1 Packaging

The packaging and distribution differ between the Python and Cython implementation.

9.1.1 Python

For the Python implementation of `dual_autodiff` a `pyproject.toml` file has been provided in the main project folder specifying package dependencies, metadata, and build requirements. The source code for the Python package is located in the `dual_autodiff` folder. To build the Python package:

1. Clone the repository to your local machine.
2. In the main project repository run `pip install .`

This will install the Python package in your current environment. Alternatively, the wheel for the Python `dual_autodiff` package is provided in the dist folder. Wheels are binary distributions of the package and can be installed easily using `pip install <wheelname>` without the requirement to clone the repository. Wheels for pure Python packages are independent of the system architecture, with execution managed by the Python interpreter [17].

9.1.2 Cython

Cython package's may be built in a similar way to the pure Python implementation.

1. Clone the repository to your local machine
2. In the `dual_autodiff_x` folder run `pip install .`

Wheels have also been provided for the Cython implementation located in `dual_autodiff_x/wheelhouse`. However, for packages containing non-Python code, the package wheels contain compiled binaries that must be compatible with the architecture of the target platform, meaning the Wheels are now platform-dependent. In this work, two wheels were created

- cp310-manylinux_x86_64
- cp311-manylinux_x86_64

These wheels correspond to the Python 3.10 and 3.11 versions on the standard Linux distribution following 64 bit x84 architecture.

9.2 Docker

A Dockerfile has also been provided to ensure the usability of the packages regardless of the user's hardware. Docker is a containerisation service that allows isolated operating system environments to be built with their own software system [18].

The Dockerfile contains instructions and commands for installing the relevant dependencies and to initialise the environment. A Docker image is a snapshot of the resulting system built using the Dockerfile and finally the Docker container runs the Docker image isolated from the host system.

The repository contains a Dockerfile for building a container with Python 3.10, as well as the Python and Cython implementation of the `dual_autodiff` package. Additional dependencies from the `requirements.txt` are also installed and both the tests and notebook folders are transferred to the container.

To build the docker container

1. Download Docker.
2. Clone the repository
3. From the project folder, build the docker image using `docker build -t dual_autodiff_image`

There are two ways to run the docker container.

1. For terminal access
    - Run `docker run -it -p 8888:8888 my-jupyter /bin/bash`
    - This provides access to the Linux environment, from here pytests may be run via `pytest test/*` or Python may be run in the terminal where both `dual_autodiff` and `dual_autodiff_x` will already have been installed.
2. For Notebook access
    - Run `docker run -p 8888:8888 dual_autodiff_image`
    - Enter `http://localhost:8888/` in the browser and when prompted enter the authentication key that is present in the terminal.
    - Access is now provided to all example notebooks with all required packages installed via their wheels.

10 Summary

This work has implemented the `dual_autodiff` package for the manipulation and computation of dual numbers following good coding practices. A robust and thorough testing suite was implemented to ensure correct implementation along with detailed docstrings allowing for the creation of exhaustive

automated documentation via Sphinx. The package was made available for distribution via Python wheels along with a cythonised version of the package. Performance between the two was compared showing the benefits of Cython. Finally, a docker image has been made available for the running of the 2 packages as well as all example notebooks.

References

[1] University of Texas at San Antonio. Dual numbers for first order sensitivity analysis, April 2023. Accessed: 2024-12-13.

[2] GeeksforGeeks. What is __init__.py file in python?, 2024. Accessed: 2024-12-13.

[3] The setuptools-scm Development Team. setuptools-scm: Manage python package versions using scm tags, 2024. Accessed: 2024-12-13.

[4] Sam Ritchie. Dual numbers and automatic differentiation, n.d. Accessed: 2024-12-13.

[5] The NumPy Community. Numpy documentation: Docstring standard, 2024. Accessed: 2024-12-13.

[6] The Sphinx Documentation Team. *Sphinx Documentation*, 2024. Accessed: 2024-12-13.

[7] Read the Docs Community. *Read the Docs Documentation*, 2024. Accessed: 2024-12-13.

[8] The pytest Development Team. *pytest: Helps you write better programs*, 2024. Accessed: 2024-12-13.

[9] The pytest Development Team. pytest-cov: Pytest plugin for measuring coverage, 2024. Accessed: 2024-12-13.

[10] 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.

[11] Martin Neuenhofen. Review of theory and implementation of hyper-dual numbers for first and second order automatic differentiation. *CoRR*, abs/1801.03614, 2018.

[12] Anders Holm. What is automatic differentiation?, 2023. Accessed: 2024-12-13.

[13] 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, 2011.

[14] Python Software Foundation. *The Python Standard Library: timeit — Measure execution time of small code snippets*, 2024. Accessed: 2024-12-13.

[15] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13:31 – 39, 05 2011.

[16] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2nd edition, 2010.

[17] Python Packaging Authority. Pep 427 – the wheel binary package format 1.0. Python Enhancement Proposal (PEP), 2013. Accessed: 2024-12-13.

[18] Inc. Docker. Docker: Enterprise container platform. Website, 2013. Accessed: 2024-12-13.