C1 Research Computing - Coursework Report

Jacob Tutt (JLT67)

Department of Physics, University of Cambridge

December 18, 2024

Word Count: 2974

1 Introduction

This report presents **dual_autodiff**, an automatic differentiation package, through discussing the motivation for its development, its core functionalities, and the rationale behind its structure and design. Its primary focus is to provide a critical review of the package's architecture, implementation and preformance while discussing potential future developments.

1.1 Background

Automatic differentiation (AD) has been a well-established area of research in fields such as computational fluid dynamics and atmospheric sciences for many years [2]. Its popularity stems from its ability to compute derivatives of functions with machine precision while significantly reducing the computational cost compared to traditional numerical methods [1]. Recently, its role has become increasingly important in machine learning, where derivatives, particularly gradients and Hessians, are ubiquitous in optimisation and training [2].

The mathematical framework of forward-mode AD is provided by dual numbers, which are defined as:

$$\mathbb{D} = \{a + b\epsilon : a, b \in \mathbb{R}\} \tag{1}$$

where ϵ is a non-zero number with the defining property:

$$\epsilon^2 = 0.$$

When functions are evaluated using dual numbers $a + b\epsilon$, their Taylor expansion can be expressed as:

$$f(a + b\epsilon) = f(a) + f'(a)(b\epsilon) + \frac{f''(a)}{2!}(b\epsilon)^2 + \frac{f'''(a)}{3!}(b\epsilon)^3 + \cdots \tag{2}$$

Since any higher-order terms than ϵ^2 vanish by definition, the result simplifies to:

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

Hence, the result of evaluating $f(a + b\epsilon)$ is a dual number where:

- Real Part ($f(a)$): The function evaluated at a.
- Dual Part ($bf'(a)\epsilon$): The functions first derivative evaluated at a, scaled by b.

This property forms the foundation of automatic differentiation. By evaluating a function using a dual number with $b = 1$, both the function value and its derivative are computed simultaneously. Further proofs and resources for a greater understanding are given in the packages documentation. The **dual_autodiff** package aims to provide a framework to define dual numbers, evaluate them for breadth of functions within Python and finally exploit this structure to provide an automatic differentiation tool.

2 Package Functionalities

2.1 Base Dual Class

At the core of this package is the **Dual** class, which defines the properties of the dual numbers, allowing their initialisation and storing their constituent parts. It also defines the formatting for representation (__repr__) of the dual number as a string. A demonstation of this is shown below:

Listing 1: Example initialisation of the Dual class

```python
x = Dual(2, 1)
print(x.real)
print(x.dual)
print(x)
```

```
Output:
2
1
Dual(real=2.0, dual=1.0)
```

2.2 Dual Numbers' Operations

The package goes on to support a range of basic mathematical operations for dual numbers, such as the addition, multiplication and power operations. This allows the class to integrate with the Python arithmetic operators (+, -, *, /, **) making it intuitive to use and more robust for the user when defining functions. This is achieved by overloading the Python operators in the Dual class eg (__add__ , __mul__ , __pow__). The derivations of these operations in the context of the dual numbers is outlined in Table 1, for the numbers $a + b\epsilon$ and $c + d\epsilon$:

Operation	**Result**
Addition (+)	$(a + c) + (b + d)\epsilon$
Subtraction (-)	$(a - c) + (b - d)\epsilon$
Multiplication (*)	$ac + (ad + bc)\epsilon$
Division (/)	$\frac{a}{c} + \frac{bc-ad}{c^2}\epsilon$
Power (**)	$a^n + na^{n-1}b\epsilon$

Table 1: Dual Numbers Operations

The package additionally supports these operators for a combination of dual numbers and scalars, and defines the reverse opperations for the cummulative operators (eg. __radd__, __rmul__). Without this functionality, the automatic differentiation tool (Section 2.5) would be limited to functions which do not contain scalars.

The inclusion of the equality and comparison operators was less trivial, as dual numbers are not directly comparable. This packages builds on the theory provided by Cheng [3] (1994) which defines equality for dual numbers as the equivalence of both their real and dual components but goes on to make comparisons based solely on their real parts (shown in Table 2).

Operation	Result
Equality (==)	$(a == c) \land (b == d)$
Inequality (!=)	$(a! = c) \land (b! = d)$
Less Than (<)	$a < c$
Greater Than (<)	$a > c$
Less Than or Equal To (≤)	$a \leq c$
Greater Than or Equal To (≥)	$a \geq c$

Table 2: Dual Numbers Comparison

A small range of examples is provided below, with a much more extensive list available in the packages documentation and notebook.

Listing 2: Example of Dual Number Operations

```
x = Dual(2, 1)
y = Dual(3, 2)
print(x + y)
print(x * y)
print(x ** 2)
print(x == y)
print(x < y)
```

```
Output:
Dual(real=5.0, dual=3.0)
Dual(real=6.0, dual=7.0)
Dual(real=4.0, dual=4.0)
False
True
```

2.3 Dual Numbers' Functions

The package also defines more complex mathematical functions within the dual numbers class, a full list of which is defined in Table 3. These functions $f(a + b\epsilon)$ are implemented using the chain rule property defined in Equation 3, allowing the real and dual parts to be computed independently.

Function	Real Part	Dual Part, ϵ
$\sin(a + b\epsilon)$	$\sin(a)$	$b\cos(a)$
$\cos(a + b\epsilon)$	$\cos(a)$	$-b\sin(a)$
$\tan(a + b\epsilon)$	$\tan(a)$	$b\sec^2(a)$
$\sinh(a + b\epsilon)$	$\sinh(a)$	$b\cosh(a)$
$\cosh(a + b\epsilon)$	$\cosh(a)$	$b\sinh(a)$
$\tanh(a + b\epsilon)$	$\tanh(a)$	$b\,\text{sech}^2(a)$
$\arcsin(a + b\epsilon)$	$\arcsin(a)$	$\frac{b}{\sqrt{1-a^2}}$
$\arccos(a + b\epsilon)$	$\arccos(a)$	$-\frac{b}{\sqrt{1-a^2}}$
$\arctan(a + b\epsilon)$	$\arctan(a)$	$\frac{b}{1+a^2}$
$\exp(a + b\epsilon)$	$\exp(a)$	$b\exp(a)$
$\log(a + b\epsilon)$	$\log(a)$	$\frac{b}{a}$
$\sqrt{a + b\epsilon}$	$\sqrt{a}$	$\frac{b}{2\sqrt{a}}$
$\text{pow}(a + b\epsilon, n)$	a^n	$na^{n-1}b$

Table 3: Dual Numbers Functions

These are initially defined as member functions of the `Dual` class, and only callable using `Dual.func()`. This would require the user to define compound functions in a non-intuitive way. Hence, the package was extended to include these functions as independent classes within the **autodiff_tools** module. These functions are defined to be able to take scalars, dual numbers, or arrays of either (discussed further in Section 2.4). Examples of single input functions are shown below:

Listing 3: Example of Dual Number Functions

```
x = Dual(2, 1)
print(x.sin())
print(sin(x))
print(log(x))
print(tan(x) + cos(x) * pow(x, 2))
```

```
Output:
Dual(real=0.9093..., dual=-0.4161...)
Dual(real=0.9093..., dual=-0.4161...)
Dual(real=0.6931..., dual=0.5)
Dual(real=-3.8496..., dual=0.4726...)
```

2.4 Numpy Integration

In fields such as machine learning, it is often necessary to differentiate functions across batches or large arrays of data points, highlighted by ongoing research into the parallelisation of automatic differentiation [5]. In order to address this need (at a elementary level) and enhance the package's utility in modern applications, Version 1.1 of `dual_autodiff` introduced seamless integration with Numpy arrays. This implementation is achieved by exploiting Numpy's ability to identify over-written operators and functions. The implications of this approach on computational efficiency are further discussed in Section 2.7. Its practical application is shown below:

Listing 4: Example of Numpy Integration

```
x = np.array([Dual(2, 1), Dual(3, 2)], ...)
print(log(x))
print(tan(x) + cos(x) * pow(x, 2))
```

```
Output:
[Dual(real=0.6931, dual=0.5000), Dual(real=1.0986, dual=0.6667), ...]
[Dual(real=-3.8496, dual=0.4726), Dual(real=-9.0525, dual=-12.3794), ...]
```

2.5 Automatic Differentiation

Using the Dual number's framework described in the previous sections, an automatic differentiation function (`auto_diff`) is introduced to the `autodiff_tools` module.

The automatic differentiation function takes a function (composed of the arithemetic operations and functions defined in Tables 1 and 3) and a scalar value(s) at which to evaluate it and its derivative. Due to the nature of dual numbers this can be simply implemented by:

1. Accepting a function and a scalar value (x) as input.
2. Initialising a dual number ($\mathbb{D} = x + \epsilon$) from the scalar value.
3. Evaluating the function at a $\mathbb{D}$, $f(\mathbb{D})$.
4. Return Function Value: $f(a) = real(f(\mathbb{D}))$.
5. Return Derivative Value: $f'(a) = dual(f(\mathbb{D}))$.

An example of its application is shown below:

Listing 5: Example of `autodiff` Function

```
def f(x):
  return x ** 2 + 2 * x + 1

x = 2
value, derivative = auto_diff(f, x)
print(value)
print(derivative)

# Similairly for an array of values
x_array = np.array([2, 3, 4])
value_array, derivative_array = auto_diff(f, x_array)
print(value_array)
print(derivative_array)
```

```
Output for single input:
Value: 9.0
Derivative: 6.0
Output for array of inputs:
Value: [9.0, 16.0, 25.0]
Derivative: [6.0, 8.0, 10.0]
```

This functionality was further expanded in the function `multi_auto_diff` to support the extended case in which the user wishes to evaluate multiple functions and their derivatives simultaneously. This can be achieved by passing a list of functions as well as the scalar values at which to evaluate them. The performance of the automatic differentiation is demonstrated below in Section 2.6.

2.6 Validation of Automatic Differentiation

The validity of the package's automatic differentiation was tested by comparing its results to the analytical derivative for a range of functions. These functions (of varying complexity) and their analytical derivatives are defined in Table 4.

Name	Function	Analytical Derivative
f_1	$x^2 + x$	$2x + 1$
f_2	$\sin(x)$	$\cos(x)$
f_3	$\log(\sin(x)) + x^2 \cos(x)$	$\frac{\cos(x)}{\sin(x)} + 2x\cos(x) - x^2 \sin(x)$

Table 4: Functions and their analytical derivatives used to verify the packages automatic differentiation ability

The functions and their analytical derivatives were evaluated over the range $x \in (0, \pi)$ and compared to the results from automatic differentiation, using the `multi_auto_diff` function. More examples of the exact implementation can be found in the packages documentation and notebook. The results of this comparison are shown in Figure 1.

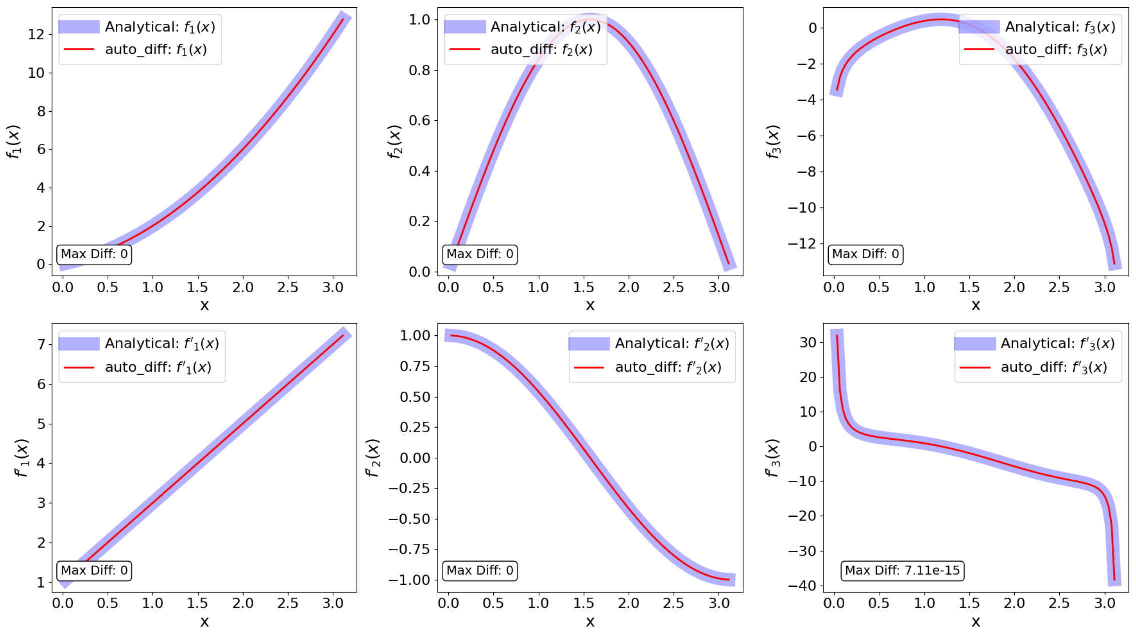


Figure 1: Comparison of the analytical and automatic differentiation results for the functions defined in Table 4, showing both the function and their derivatives' values as well as the maximium difference between them

For all functions, the automatic differentiation results were consistent with the analytical derivatives, with $f_1(x)$ and $f_2(x)$ having zero deviation from the analytical form and the maximum deviation for $f_3(x)$ being 7.11×10^{-15}. This falls within the expected numerical error for floating-point calculations and demonstrates the package's ability to produce exact derivatives through automatic differentiation.

2.7 Comparison with Numerical Differentiation

To justify the use of automatic differentiation, the package's performance is compared with traditional numerical methods, using accuracy, speed, and memory usage as metrics. Both forward difference (Eq 4) and central difference (Eq 5) methods are comparison against. The benchmark that these are each evaluated for is differentiating $f_3(x)$ from Table 4 for 100 data points over the range of $x \in (0, \pi)$. Each metric provided is averaged for 100 runs.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{4}$$

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{5}$$

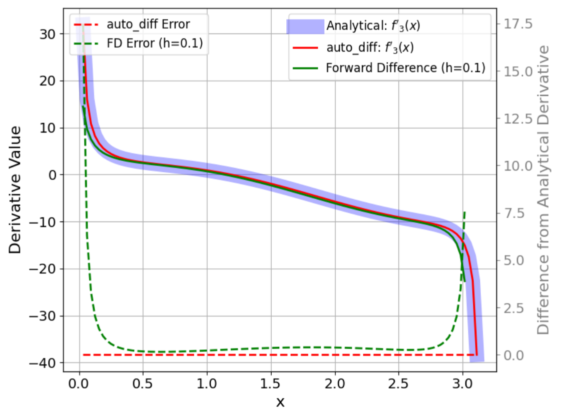
Firstly, we compare the method's computational overheads (Table 5), and can see a clear advantage in terms of speed and memory usage for the numerical approaches. Typically both numerical methods are two orders of magnitude faster and use on average $\approx 78\%$ of the memory. Although initially unexpected, this can be atributed to each of the methods being preforming simultaneously on 100 data points for each iteration. Hence the numerical derivates, which fully exploit Numpy's C backend, benefit from significant performance improvements through vectorisation. Improving the package's efficiency by further increasing its C backend usage is discussed in Section 3.

Method	**Avg Run Time (s)**	**Avg Memory (kb)**
AutoDiff	0.006733	6.10
Forward Difference (h=0.1)	0.000019 (0.28%)	4.34 (71%)
Forward Difference ($h = 1 \times 10^{-10}$)	0.000017 (0.25%)	4.34 (71%)
Central Difference (h=0.1)	0.000023 (0.34%)	5.21 (85%)
Central Difference ($h = 1 \times 10^{-10}$)	0.000019 (0.28%)	5.20 (85%)

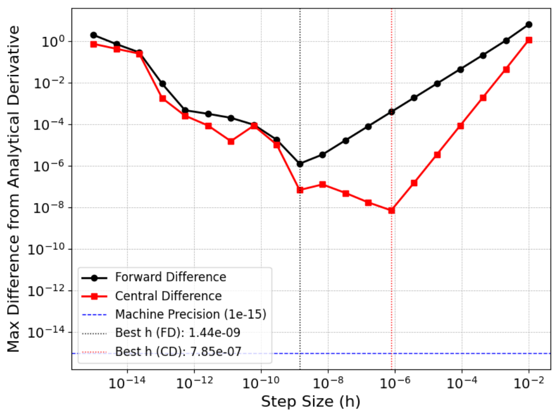
Table 5: Comparison of Average Run Time and Memory Usage for Different Methods

The second factor to consider is the accuracy of numerical methods in comparison with automatic differentiation, which was shown to be exact (within machine error) in Section 2.6. Figure 2a shows the preformance of the forward difference method with $h = 0.1$ as a demonstration of where the numerical methods fails to provide accurate results. For the case of $f_3(x)$, this can be seen as x approaches 0 and π, corresponding to where the function is discontinuous and hence has very large second derivatives. Therefore automatic differentiation shows advantages towards ill-defined regions, as it is able to provide exact derivatives for these points, whereas numerical methods are limited by the step size's relation to the function's second derivative.

Figure 2 shows the maximium deviation from the analytical derivative, for the forward and central difference methods, across varying step sizes. It can be seen that although the central difference method is more accurate than the forward difference method, both are significanty effected by the step size. Notably, it suggests that there is an optimal step size, h, at which the methods reaches a maximium accuracy before the error increases again. This can be attributed to numerical integrations requirement to balance the truncation errors with numerator's round off error (catastrophic cancellation [4]). The optimal step size for the central difference method was found to be $h = 7.85 \times 10^{-7}$ with a maximum error of 1.27×10^{-6} and similairly for the forward difference method $h = 1.44 \times 10^{-9}$ with a maximum deviation of 7.18×10^{-9}. Although important to consider that this is an extreme case, the automatic differentiation shows clear advantages in accuracy, with numerical methods limited to an error at least 6 order of magnitude greater than machine precision. Thus in applications where very high degrees of accuracy are required, such as scientific computing and machine learning, automatic differentiation would be prefered.



(a) Comparison of accuracy of automatic differentiation and numerical methods



(b) Accuracy against step size for forward and central method differentiation

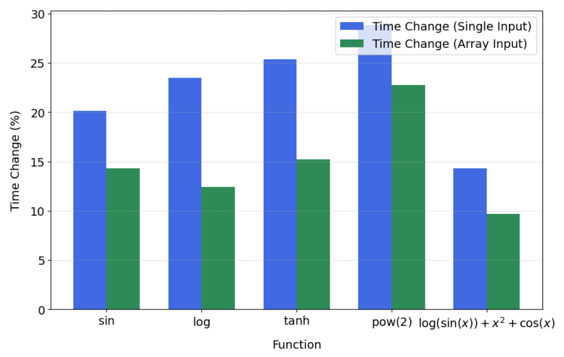
Figure 2: Comparison of accuracy for numerical differentiation across step sizes

3 Exploiting C's Optimised Performance

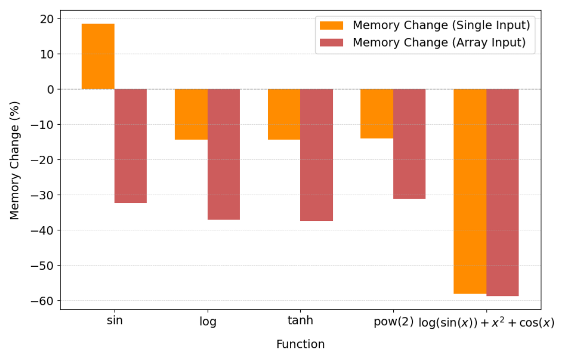
As highlighted in Section 2.7, the performance is constrained by Python's interpreted execution, which is inherently slower than C's compiled approach. This section discusses the steps taken to maximise the package's performance by exploiting C. Firstly, through Cythonisaton and then through future developemnt of the packages use of numpy's C backend.

3.1 Cythonisation

A Cythonised sub-package (dual_autodiff_x) was then created, a proccess in which the package is converted to C allowing it to be precomplied, thus improving its run-time performance. The structure of the package is further discussed in Section 4. The performance of the Cythonised package (dual_autodiff_x) was then compared with the original Python package (dual_autodiff), by evaluating the automatic derivative for a range of functions (detailed in figure 3). This was performed for single inputs ($x = 1$) and an array of 8 inputs, each ($x = 1$) for comparability.



(a) Percentage change in runtime for automatic differentiation in Python and Cython



(b) Percentage change in memory usage for automatic differentiation in Python and Cython

Figure 3: Comparison of Cython and Python package's computational overheads for automatic differentiation for both single and array inputs

The cythonised package shows consistent decreases in runtime across all functions and input types, in the range of 14-29% for single-value inputs. Although significant, the majority of the original Python package's functions are implemented through the C depenedent 'math' library and thus already exploit C's advantages and resulting in a marginal decrease. Notably, the array inputs show a smaller fractional change, typically 2/3 of their single input counter parts. This supports the packages success in vectorisation (section 2.4) as the package is already exploiting Numpy's C backend more, making its performance less changeable for Cythonisation.

On the other hand the memory usage of the Cythonised package typically shows a significant increase, upwards of 50% for complex functions such as $f_3(x)$. This is expected due to C's use of non-dynamic memory allocation, which although causes greater memory demand, is one of the features which assists the improved computational performance.

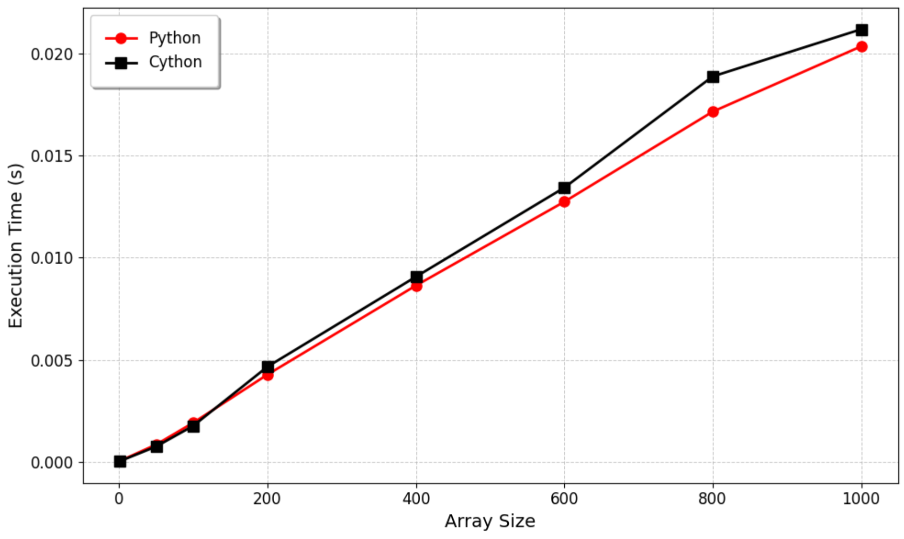


Figure 4: Scaling of automatic differentiation of the function $\sin(x)$ with array size

Finally, the scaling of run time with array size for the automatic differentiation of $\sin(x)$ is shown in Figure 4. It shows a linear trend for both the Python and Cythonised sub-packages, with the former consistently ($\approx 10\%$) higher. Therfore, both show a computational complexity of $O(n)$. Contrary to before this shows that the packages are performing the calculations within the array sequentially rather than in parallel, and thus the packages vectorisation still requires further development.

3.2 Future Developments for C

Version 1.2 will aim to further exploit C's capabilities by redefining the __array_ufunc__ method to enable full vectorization of operations. The correct overwritten operators can be found within the `Future_Development` folder of the package's repository (note this would typically not be included in the package distribution but has been for assesment purposes). Although this method has been shown to work for well defined cases, the Dual class's initialisation requires rebuilding to support all cases robustly, so is left for future development rather than compromising the packages current functionality. A simplified example of the implementation of this method is shown below for np.sin.

Listing 6: Example of __array_ufunc__ Method

```
def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
    if ufunc == np.sin:
        real_result = np.sin(real_parts)
        dual_result = np.cos(real_parts) * dual_parts
    return Dual(real_result, dual_result)
```

4 Development of Package

This section provides a review of the software development practises followed during the coursework and provides the motivation behind some of the decisions made.

4.1 Error Handeling

Extensive error handling was implemented throughout to ensure robustness as well as clear and informative error messages making the package more user friendly and accessible. Examples of these error messages can be found in the packages notebook as well as being provided within the API references (including examples) in the packages documentation.

4.2 Package Structure

The package was structured into two modules, `dual_autodiff` and `autodiff_tools`, to separatate the core dual number functionality from the automatic differentiation tools. Despite this the `__init__.py` was configured to expose all core classes and functions to the user from the package's root allowing for intuitive imports, such as:

Listing 7: Example of Package Imports

```python
from dual_autodiff import Dual, Sin, auto_diff, ...
#or
from dual_autodiff import *
```

Additionally, the package contains both the Python and Cython versions, `dual_autodiff` and `dual_autodiff_x`, as submodules, allowing users to easily import either version for comparison. Both submodules have been included in the wheel distribution for assessment purposes however, in practice, only the optimized Cython version would typically be distributed.

Notably, if both versions are imported simultaneously, it can be difficult to distinguish between identical functions, such as `auto_diff`. To avoid this issue, the coursework renames the Cythonised functions on import (typically not required but was included for comparison).

Listing 8: Referencing Cythonised Functions on Import

```python
from dual_autodiff import auto_diff
from dual_autodiff_x import auto_diff as auto_diff_x
```

Note, when wheel files are used, the user may run into conflicts between the installed and local version of `dual_autodiff_x` in the terminal - discussed in Section 4.4.1.

4.3 Package Distribution and Wheels

The package includes wheels for linux architecture running either Python 3.10 and 3.11 in the `wheelhouse` directory (cp310-manylinux_x86_64 and cp311-manylinux_x86_64). These allows the binary distribution of the package, containing both `dual_autodiff` and `dual_autodiff_x` subpackages. User can thus install the package without preforming the Cythonisation and compilation locally. Note, the Python version would not be typically included, but was included for assessment purposes.

This package also supports local installation (`pip install -e .`), which generates and compiles the current architecture's necessary files locally, such as `.so` files (shared object files). As such these files are not included outside of the wheels, as they are unique to the users architecture.

4.4 Package dependencies

The package defines both essential or optional dependencies, all of which are defined within `pyproject.toml` file for easy installation. This distinction was made to minimise the dependencies installed on the users device and allow only the required dependencies to be installed. The classes of dependencies are defined in Table 6. Installing these dependencies with the wheels is equally trivial and outlined in the `README` and documentation.

Name	**Purpose**	**Installation**
Essential	Core dependencies	`pip install -e .`
Tutorial	For example notebooks	`pip install -e ".[tutorial]"`
Testing	For running test suites	`pip install -e ".[testing]"`
Documentation	Building documentation	`pip install -e ".[docs]"`

Table 6: Package Dependencies

4.4.1 Neich Conflict for Coursework

A potential conflict arises when the user installs the package from the wheels but attempts to run the Cythonised version from the terminal in the package's directory. In such cases, the locally available package directory is prioritised over the installed version, leading to the program searching for the `.so` files in the local directory, which are only generated during a local installation using `pip install -e .`. This problem is unique to the coursework as wheels are normally distributed independently to the source code.

4.5 Testing

A comprehensive test suite of all functionalities and edge cases was implemented in the `tests` directory, using `pytest`. This includes 43 tests encompassing 258 statments over 7 files, each focused on testing a specific feature. This allows the user to verify the package's installation, and also stop future developers from introducing conflicts. This was introduced into the continuous integration as a pre-commit hook which is discussed in Section 4.8.

A report on the test coverage was generated using the `coverage` utility, showing a coverage of 100% for the package, shown in Table 7. Note the 82% coverage for the `version.py` file is neglected as it simply contains metadata such as the version number rather than core features.

Name	**Stmts**	**Miss**	**Cover**	**Missing**
dual_autodiff/__init__.py	5	0	100%	-
dual_autodiff/autodiff_tools.py	103	0	100%	-
dual_autodiff/dual.py	139	0	100%	-
dual_autodiff/version.py	11	2	82%	5-6
TOTAL	258	2	99%	-

Table 7: Test coverage report for the `dual_autodiff` package

4.6 Documentation

Documentation on the package was auto-generated using `sphinx`, and is hosted on `readthedocs` which relies on the `.readthedocs.yml` file. This provides a comprehensive guide to the package's functionality, including the API references, examples and context. The documentation can also be built locally using the `make html` command in the `docs` directory, once the `docs` dependencies have been installed.

4.7 Version Control

The package was developed using Git for automatic version control and exploiting `setuptools_scm` for dynamic tracking. Once the first usable version [1.1.0] was released, new features where developed on separate feature branches and merged into the main branch once completed, thus not compremising the packages functionalities.

4.8 Continuous Integration

This package made use of the following continuous integration (CI) methods:

- **Pre-commit Hooks for Testing**: `pytest` was configured to ensure all tests were passed before allowing commits, ensuring updates would not corrupt the package. Future developers are encouraged to do the same before submitting a pull request, highlighted in the `README`.
- **Sychronisation of Cython**: The `setup.py` was configured to automatically duplicate `.py` files to `.pyx` files before running the build, ensuring the Cythonised package was always up to date.
- **Documentation**: `Readthedocs` was configured to automatically build the documentation on each push to the main branch, ensuring the documentation was always up to date.

5 Summary

Overall, the `dual_autodiff` package provides a framework to preform automatic differentiation, supporting a wide range of functions. The package was developed following good software development practices, and shows strong increases in the accuracy of automatic differentiation over numerical methods. Future versions aim to further exploit C's optimised performance improve the packages vectorisation.

5.1 Declaration of Use of Autogeneration Tools

This project made use of Large Language Models (LLMs), primarily ChatGPT and Co-Pilot, to assist in the development of the package. These tools have been employed for:

- Helping generate docstrings for the repository's documentation.
- Formatting plots to enhance presentation quality.
- Performing iterative implementation to predefined code.
- Debugging code and identifying issues in implementation.
- Helping develop tests for the package.
- Supporting generation of metadata files
- Identifying spelling and punctuation inconsistencies within the report.
- Suggesting more concise phrasing to reduce word count.

References

[1] Michael Bartholomew-Biggs et al. "Automatic differentiation of algorithms". In: *Journal of Computational and Applied Mathematics* 124.1 (2000). Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations, pp. 171–190. ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-0427(00)00422-2. URL: https://www.sciencedirect.com/science/article/pii/S0377042700004222.

[2] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: 1502.05767 [cs.SC]. URL: https://arxiv.org/abs/1502.05767.

[3] H. H. Cheng. "Programming with Dual Numbers and its Applications in Mechanisms Design". In: *Engineering with Computers, An International Journal for Computer-Aided Mechanical and Structural Engineering* 10.4 (1994), pp. 212–229. URL: https://iel.ucdavis.edu/publication/journal/j_EC1.pdf.

[4] Annie Cuyt et al. "A Remarkable Example of Catastrophic Cancellation Unraveled". In: *Computing* 66.3 (May 2001), pp. 309–320. ISSN: 1436-5057. DOI: 10.1007/s006070170028. URL: https://doi.org/10.1007/s006070170028.

[5] William S. Moses et al. "Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation". In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2022, pp. 1–18. DOI: 10.1109/SC41404.2022.00065.