UNIVERSITY OF CAMBRIDGE

C1 Research Computing - Coursework Assignment

Raunaq Rai (rsr45@cam.ac.uk)
Data Intensive Science, Department of Physics, University of Cambridge

18 December, 2024

1 Introduction

This report details the development and implementation of a Python package, `dual_autodiff`, designed for automatic differentiation using dual numbers. The package computes derivatives efficiently while supporting mathematical operations such as trigonometric, logarithmic, and exponential functions.

The approach builds on the concept of forward-mode automatic differentiation, which is essential in fields like optimisation, computational physics, and machine learning. This technique traces its roots to the foundational work by Wengert [1], who introduced a systematic way to compute derivatives using intermediate variables. More recently, Baydin et al. [2] surveyed the use of automatic differentiation in machine learning, emphasising its importance in training deep neural networks.

To enhance performance, a Cython-optimised version, `dual_autodiff_x`, was also developed. This document covers the structure of the project, the mathematical principles behind dual numbers, and the implementation details of the package.

2 Setting Up the Development Environment

Apple Silicon devices primarily use the ARM64 architecture, which can pose challenges when working with scientific computing tools built for x86_64. To ensure compatibility with these tools, I configured the development environment to run in x86_64 mode. This step enabled the execution of packages and tools designed for x86_64 systems.

To achieve this:

- **Rosetta Installation:** Rosetta, an emulation layer, was installed to facilitate running x86_64 binaries on ARM-based devices. This was achieved using:

```
/usr/sbin/softwareupdate --install-rosetta
```

- **Configuring Terminal:** The Terminal application was set to run in Rosetta mode, ensuring compatibility with x86_64 libraries and tools.

- **Creating an x86_64 Conda Environment:** A dedicated Conda environment was created with all required dependencies for developing and testing the package.

This setup allowed for consistent development and ensured the compatibility of tools and libraries required for this project. Further details can be found within dual_autodiff.html within docs.

3 Theoretical Background

3.1 Dual Numbers

Dual numbers can be defined as truncated Taylor series of the form:

$$x = v + \dot{v}\epsilon, \tag{1}$$

where $v, \dot{v} \in \mathbb{R}$, and ϵ is a nilpotent number such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. Here:

- v: Represents the *primal value*.
- $\dot{v}$: Represents the *derivative* or *tangent value*.

As explained by Baydin et al. [2], arithmetic operations with dual numbers align naturally with symbolic differentiation principles:

$$(x_1 + \dot{x}_1\epsilon) + (x_2 + \dot{x}_2\epsilon) = (x_1 + x_2) + (\dot{x}_1 + \dot{x}_2)\epsilon, \tag{2}$$

$$(x_1 + \dot{x}_1\epsilon)(x_2 + \dot{x}_2\epsilon) = x_1x_2 + (x_1\dot{x}_2 + \dot{x}_1x_2)\epsilon, \tag{3}$$

3.2 Automatic Differentiation

Automatic differentiation uses dual numbers to compute derivatives efficiently. For a function $f(x)$, substituting $x = v + \dot{v}\epsilon$ yields:

$$f(x) = f(v + \dot{v}\epsilon) = f(v) + f'(v)\dot{v}\epsilon, \tag{4}$$

The derivative $f'(v)$ is embedded in the coefficient of ϵ, enabling evaluation of function values and derivatives.

This principle extends to composite functions via the chain rule:

$$f(g(v + \dot{v}\epsilon)) = f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon, \tag{5}$$

4 Implementation of Dual Numbers and Operations

4.1 Overview of the `Dual` Class

The `dual.py` file implements the `Dual` class, the core of the `dual_autodiff` package. This class defines dual numbers and supports operations such as addition, subtraction, multiplication, and division.

4.1.1 Arithmetic Operations

The `Dual` class simplifies arithmetic operations by overriding standard operators such as `+`, `-`, `*`, and `/`. This allows `Dual` objects to be used just like regular numbers. For example:

```
x = Dual(2, 1)
y = Dual(3, 2)
print(x + y)  # Output: Dual(real=5, dual=3)
```

4.1.2 Mathematical Functions

The `Dual` class also implements important mathematical functions such as:

- Trigonometric functions (`sin`, `cos`, `tan`).
- Exponential and logarithmic functions (`exp`, `log`).
- Hyperbolic functions (`sinh`, `cosh`, `tanh`).
- Square root (`sqrt`).

For example:

```
x = Dual(2, 1)
result = x.sin()
print(result)  # Output: Dual(real=0.9092..., dual=-0.4161...)
```

4.1.3 Error Handling and Special Cases

The `Dual` class ensures that mathematical operations like `log` and `sqrt` are only applied within valid domains, raising appropriate errors when encountering invalid inputs.

4.2 Example of Using Dual Numbers

Consider the function:

$$f(x) = \log(x) + x^2 \implies f'(x) = \frac{1}{x} + 2x. \tag{6}$$

This can be evaluated directly using the `Dual` class:

```
x = Dual(2, 1)
f_x = x.log() + x**2
print(f_x)  # Output: Dual(real=4.0, dual=4.5)
```

4.3 Utility Functions

To enhance usability and simplify mathematical operations, two utility modules are provided as part of the package:

- `functions.py`: This module provides aliases for commonly used mathematical functions, such as `sin`, `cos`, `log`, and `sqrt`. These aliases act as wrappers around the corresponding methods of the `Dual` class, allowing users to apply these functions directly to `Dual` instances or standard numerical values. For example:

```
from dual_autodiff.functions import sin, cos, log
x = Dual(2, 1)
result = sin(x) + log(x)
```

  These aliases improve the user experience by making the library's API intuitive and easy to use, especially when working with dual numbers in complex expressions.

- `base.py`: This module includes helper functions to streamline operations on `Dual` instances:
  - `is_dual_instance(value)`: A utility function that checks whether a given value is an instance of the `Dual` class. This is useful for validation, ensuring that mathematical operations are applied only to compatible types.
  - `ensure_dual(value)`: A function that wraps a non-`Dual` value into a `Dual` object with its derivative initialised to zero. This ensures that all mathematical functions can seamlessly handle both standard numerical values and dual numbers.

These utility functions simplify the use of the package by enabling both explicit handling of dual numbers and implicit conversions when required, making the library more accessible for a wide range of users.

5 Project Structure and Packaging

5.1 Repository Organisation

The repository adheres to established best practices for Python projects to ensure clarity, maintainability, and modularity. Below is an overview of its structure:

5.1.1 Top-Level Directory

The top-level directory organises the project as follows:

- `dual_autodiff/`: Core implementation, including modules like `dual.py`, `functions.py`, and `base.py`.
- `dual_autodiff_x/`: Cythonised implementation of the package for enhanced performance, including `.pyx` source files and compiled `.so` binaries.
- `tests/`: Unit tests for core modules. Use the `pytest` command to run the tests.

- `docs/`: Documentation files, including Sphinx configurations and example notebook.
- `report/`: LaTeX report and related files.
- `dist/`: Package distribution files (wheel and source archives).
- `build/`: Temporary build files.
- `pyproject.toml`: Modern Python project configuration.
- `requirements.txt`: Python dependencies.
- `environment.yaml`: Conda environment definition.
- `README.md`: Project overview and instructions.
- `LICENCE`: Contains the MIT License under which the project is distributed.

5.2 Building and Installing the Package

The `pyproject.toml` file is used to manage the configuration and metadata of the project. It follows the modern Python packaging standards and includes the following sections:

- `[build-system]`: Specifies the tools required for building the package, such as `setuptools` and `wheel`.
- `[project]`: Contains metadata, including the project name, version, author, and dependencies.
- `[tool.setuptools_scm]`: Enables dynamic versioning based on the state of the repository.

To build and install the package, the following steps were performed:

1. Install build tools: `pip install build`.
2. Build distributions: `python -m build`.
3. Install in editable mode: `pip install -e .`.

This approach ensures the package is properly configured, packaged, and ready for distribution or further development.

6 Publishing to PyPI

To make the `dual_autodiff` package publicly available, it was uploaded to the Python Package Index (PyPI). The following steps outline the publishing process and installation instructions.

6.1 Publishing to PyPI

To publish the `dual_autodiff` package on PyPI, the following steps were followed:

1. **Create Distributions:** Build source and wheel distributions as done previously:
2. **Upload to PyPI:** Use `twine` to securely upload distributions.
   Authentication with PyPI credentials was required.
3. **Verify Upload:** `https://pypi.org/project/rsr45-dual-autodiff/`

6.2 Installing the Package

Install the package via `pip`:

```
pip install rsr45-dual-autodiff
```

6.3 Testing the Installation

Verify functionality:

```
import dual_autodiff as df
x = df.Dual(2, 1)
print(x.sin())
```

7 Differentiating a Function

7.1 Function Definition

The target function for differentiation is:

$$f(x) = \log(\sin(x)) + x^2 \cos(x), \tag{7}$$

The derivative of this function, computed analytically, is:

$$f'(x) = \frac{\cos(x)}{\sin(x)} - x^2 \sin(x) + 2x \cos(x), \tag{8}$$

7.2 Using Dual Numbers for Differentiation

To compute $f'(x)$ at $x = 1.5$ using dual numbers:

- Represent x as a dual number: $x = 1.5 + 1\epsilon$, where the real part is 1.5 and the dual part represents the derivative.
- Substitute x into $f(x)$ and use the dual number arithmetic to compute $f'(x)$ from the dual part of the result.

7.3 Results

7.3.1 Using Dual Numbers

The function $f(x)$ and its derivative $f'(x)$ were computed at $x = 1.5$ using dual numbers. The results are as follows:

$$f(1.5) = 0.15665054756073515, \quad f'(1.5) = -1.9612372705533612$$

7.3.2 Using Manual Computation

The analytical expression for $f(x)$ and $f'(x)$ was used to compute the same values at $x = 1.5$. The results are:

$$f(1.5) = 0.15665054756073515, \quad f'(1.5) = -1.9612372705533614$$

7.3.3 Comparison

The results obtained using dual numbers closely match the manually computed values, confirming the correctness of the dual number implementation. The slight discrepancy in the derivative (2×10^{-13}) is attributed to floating-point precision errors inherent in numerical computations.

7.4 Comparison with Numerical Differentiation

- **Numerical Differentiation:** The central difference formula was used:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \tag{9}$$

  This was evaluated for step sizes decreasing logarithmically from $h = 10^{-0.5}$ to $h = 10^{-3}$.

Figure 1 illustrates the behavior of the numerical derivative as the step size decreases. The red dashed line represents the true derivative obtained using dual numbers, which serves as the reference value.

- **Accuracy:** For moderate values of h, the numerical derivative closely matches the true value. However, as h increases further, round-off errors lead to divergence.
- **Dual Numbers:** The dual number method provides a stable and precise derivative, unaffected by the limitations of finite differences.
- **Efficiency:** Unlike numerical differentiation, dual numbers compute the derivative in a single step, making the method both computationally efficient and less error-prone.

8 Tests and Validation

The `tests/` directory contains unit tests designed to validate the functionality of the `dual_autodiff` package. These tests ensure the correctness of mathematical operations, dual number functionality, and integration with various functions like trigonometric, logarithmic, and exponential operations.

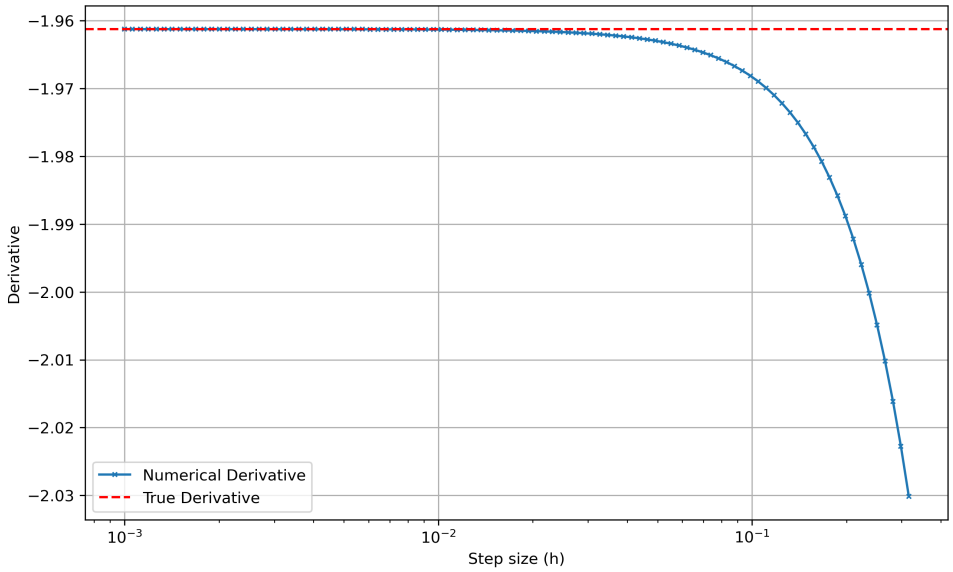

Figure 1: Divergence of the numerical derivative for increasing step sizes. The red dashed line indicates the true derivative obtained using dual numbers. The numerical derivative converges to the true value for small step sizes (approximately $< 10^{-2}$), but diverges due to round-off errors as h becomes larger.

8.1 Structure of the `tests/` Directory

The directory includes the following key test files:

- `test_dual.py`: Validates the core `Dual` class, including arithmetic operations and function implementations.
- `test_functions.py`: Tests global mathematical functions like `sin`, `cos`, and `log`.
- `test_base.py`: Ensures utility functions such as `is_dual_instance()` and `ensure_dual()` work correctly.

8.2 Outcome

The tests validate that the `dual_autodiff` package functions as expected under various scenarios. They also confirm that dual numbers provide accurate derivatives.

9 Project Documentation with Sphinx

The `docs/` directory contains all the files required to generate the HTML documentation for the `dual_autodiff` package using Sphinx. After running `make html` in the terminal, Sphinx processes the source files and generates structured HTML documentation, which can be found in the `build/html/` directory.

9.1 Structure of the `docs/` Directory

- `Makefile` and `make.bat`: Used to build the documentation. The `Makefile` is for Unix-based systems, while `make.bat` is for Windows.

- `source/`: Contains the source files for the documentation:
  - `index.rst`: The main landing page of the documentation, linking to other sections.
  - `dual_autodiff.rst`: API reference for the package, generated using the `autodoc` extension.
  - `modules.rst`: Lists all the modules included in the `dual_autodiff` package.
  - `tutorial.rst`: A guide for using the package, linking to the tutorial notebook.
  - `dual_autodiff.ipynb`: A Jupyter notebook providing hands-on examples of the package's features. This also includes a comparison with the package's cythonized version.
  - `implementation.rst`: Documentation of the theoretical background and implementation details of the package.
  - `apple_silicon_x86_setup.rst`: A section explaining how to set up the development environment on Apple Silicon devices.
  - `conf.py`: The Sphinx configuration file, which defines project settings, extensions, and theme configurations.
- `build/`: Stores the generated documentation:
  - `build/doctrees/`: Contains intermediate files generated during the build process.
  - `build/html/`: The final HTML output, including:
    - `index.html`: Main landing page.
    - `dual_autodiff.html`: API reference.
    - `tutorial.html`: Examples and usage guide.
    - `apple_silicon_x86_setup.html`: Environment setup instructions.
    - `implementation.html`: Theoretical background and implementation.
    - `modules.html`: Package modules overview.
    - `_images/`: Documentation figures (`dual_autodiff_13_1.png`, `dual_autodiff_23_1.png`)
    - `_modules/`: Generated module docs (`dual_autodiff/base.html`, `dual_autodiff/dual.` etc.).
    - `_sources/`: Source files (`index.rst.txt`, `tutorial.rst.txt`, etc.).
    - `_static/`: Static assets (CSS, JavaScript).
    - `search.html`, `genindex.html`, `searchindex.js`: Search and index functionality.

9.2 Generated Output

The generated HTML documentation includes:

- **Main Landing Page:** Provides an overview of the project with links to tutorials, references, and key sections.
- **API Documentation:** Comprehensive details for all modules, classes, and functions in the package.

- **User Tutorial:** A practical guide with examples demonstrating how to use the package effectively.
- **Setup Guide:** Step-by-step instructions for configuring the development environment, including Apple Silicon devices.
- **Implementation Details:** An in-depth explanation of the internal workings, including key algorithms and design principles.

The Sphinx-generated documentation ensures clarity and accessibility. It combines automatically generated API references with user-friendly tutorials, making it an important resource for both developers and users.

10 Cythonizing the Package

10.1 Configuration and Implementation

To Cythonize the `dual_autodiff` package, a separate directory named `dual_autodiff_x` was created. This included necessary configurations to ensure efficient compilation and distribution of the Cythonized version.

10.1.1 Key Configuration Files

- `setup.py`**:** Defined Cython modules to be compiled (e.g., `dual.pyx`, `functions.pyx`) and metadata for the package.
- `pyproject.toml`**:** Declared build dependencies (`Cython`, `setuptools`, `wheel`) and Python version compatibility.
- `MANIFEST.in`**:** Included essential files (`README.md`, compiled `.so` files) while excluding unnecessary source files (`.pyx`, `.py`).

10.1.2 Cythonization Process

1. **Code Preparation:** Python files (`.py`) in the original `dual_autodiff` directory were copied into `dual_autodiff_x` and renamed to `.pyx` to allow Cython compilation.
2. **Compilation:** The source files were compiled into shared object files (`.so`) using:

```
python setup.py build_ext --inplace
```

3. **Installation:** The package was installed in editable mode for testing and further development:

```
pip install -e .
```

10.2 Performance Insights

To evaluate the effectiveness of Cythonization, we compared the performance of the pure Python and Cythonized implementations.

10.2.1 Experimental Setup

Execution times were measured for arrays of dual numbers with lengths ranging from 100 to 14,000. Three ranges of real parts were considered: $(0, 10)$, $(10, 100)$, and $(100, 1000)$. Each experiment was repeated 100 times, and linear regression was applied to analyse gradients of execution time with respect to array length.

10.2.2 Observations

Figure 2 illustrates the performance comparison:

- The Cythonized version exhibited lower execution times across all scenarios.
- Gradients for the Cythonized implementation were consistently smaller, highlighting better scalability.
- Performance improvements were particularly notable for larger arrays, validating the computational efficiency of Cython.

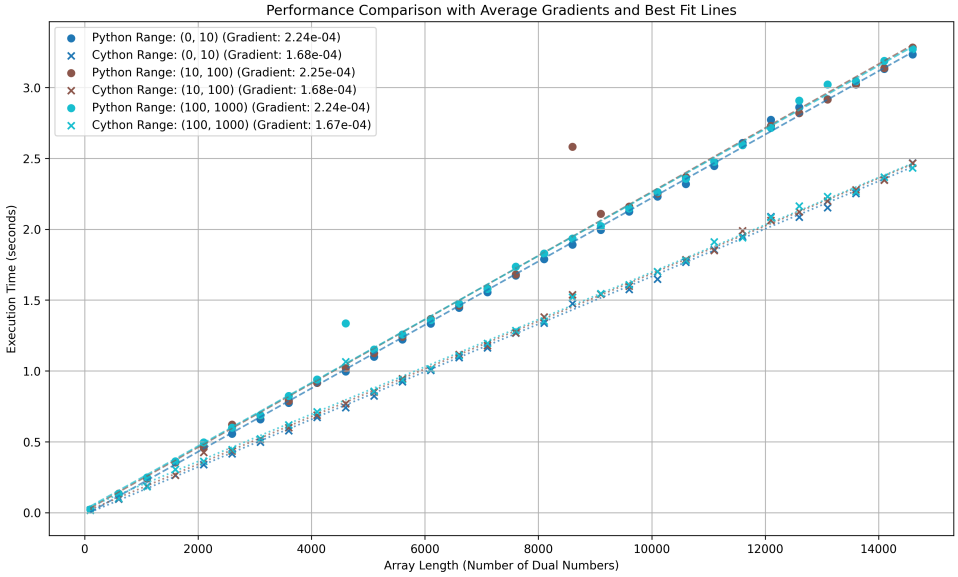

Figure 2: Performance comparison between the pure Python and Cythonized versions of `dual_autodiff`. The gradients indicate the rate of increase in execution time with array length.

10.3 Analysis of Gradients

The gradients in Figure 2 provide a quantitative measure of how execution time scales with the array length:

- **Python Implementation:** The gradients are consistently higher, around 2.20×10^{-4}, across all ranges of dual numbers. This indicates that the execution time for the Python implementation increases more rapidly with the number of dual numbers. The similarity of gradients across ranges suggests that the range of real parts has minimal impact, and the computational overhead primarily depends on array length.

- **Cythonized Implementation:** The gradients are significantly lower, approximately 1.63×10^{-4}, for all ranges. This slower rate of increase in execution time highlights the efficiency of the Cythonized implementation. A lower gradient indicates that the Cythonized version scales better with increasing array lengths, making it more suitable for handling larger datasets.

Scalability and Impact of Lower Gradients:
The lower gradient for the Cythonized implementation demonstrates its superior scalability. As the array length grows, the execution time for the Cythonized version increases at a much slower rate compared to the Python implementation. This efficiency arises from the reduced overhead in Cython, where the code is compiled into C, minimising dynamic type-checking and interpretation, which are inherent to Python. Consequently, the Cythonized version is better equipped to handle larger and more computationally intensive tasks efficiently.

10.4 Conclusion

Cythonization significantly reduced execution time and enhanced scalability by converting Python code into optimized C extensions. These improvements align with findings from prior research, such as Mortensen and Langtangen [3], which highlight that Cythonized code can achieve performance levels comparable to low-level languages like C++. While additional experiments could provide further insights, Figure 2 clearly illustrates that the Cythonized implementation consistently outperforms the Python version, with the performance gap widening as array lengths increase.

11 Building Wheels for Linux

To create specific wheels for the `dual_autodiff_x` package targeting `cp310-manylinux_x86_64` and `cp311-manylinux_x86_64`, I initially attempted the process manually on the University of Cambridge's CSD3 cluster due to compatibility issues on macOS M4. Later, I streamlined the process by utilizing a family member's laptop, which allowed me to generate wheels in the required manylinux format with a single command.

The initial steps on CSD3 taught me valuable lessons about creating environments, understanding computer architecture, and troubleshooting compatibility challenges. Although these steps did not produce manylinux-compatible wheels, the experience deepened my technical understanding and problem-solving skills. Borrowing an older laptop ultimately provided a simpler and more effective solution, highlighting the importance of adaptability in software development.

Below, I outline my approach to building wheels manually on CSD3 without using `cibuildwheel`. While the resulting wheels, included in my `wheelhouse` directory, are not in manylinux format, the process was a valuable learning experience. In section 11.2, I outline how I was able to build the wheels more easily and in the correct format.

I have included four `.whl` files in the `wheelhouse` directory. However, the two essential files are those in the manylinux format, as they meet the required compatibility standards and are required for the coursework.

11.1 Steps for Building the Wheels on CSD3

1. **Building the Python 3.10 Wheel:**

   - Python 3.10 was built from source and installed in `$HOME/python310`:

     ```
     ./configure --prefix=$HOME/python310 --enable-optimisations
     make -j$(nproc) && make install
     ```

   - The wheel was created and saved in the `wheelhouse` directory:

     ```
     /home/rsr45/python310/bin/python3.10 setup.py bdist_wheel
     --dist-dir wheelhouse
     ```

2. **Building the Python 3.11 Wheel:**

   - Verified Python 3.11 was installed on CSD3 and prepared the environment:

     ```
     python3.11 -m pip install --user --upgrade pip setuptools
     wheel cython
     ```

   - The wheel was created and saved in the `wheelhouse` directory:

     ```
     python3.11 setup.py bdist_wheel --dist-dir wheelhouse
     ```

11.2 Final Method and Contents of the Wheel

By utilising a family member's laptop, I generated both wheels in the manylinux format with a single command:

```
CIBW_BUILD="cp310-manylinux_x86_64 cp311-manylinux_x86_64" cibuildwheel
 --output-dir wheelhouse
```

The `CIBW_BUILD` environment variable specifies that the wheels should be built exclusively for Python 3.10 and Python 3.11 on the manylinux_x86_64 platform. This command uses `cibuildwheel`, which uses Docker (running in the background) to build compatible wheels efficiently, and outputs the generated wheels to the `wheelhouse` directory.

This method simplified the process and ensured compliance with `manylinux` standards, resulting in wheels suitable for distribution across most Linux environments. The wheels contain only the necessary compiled files (`.so` and `.pyd`), with source files (`.pyx`) excluded, as verified by inspecting the wheel contents.

The wheels are stored in the `wheelhouse` directory, and their contents were verified by extracting each into the `wheel_contents` directory:

- **Compiled Binaries:** Shared object files (`*.so`) for the core modules (`base`, `dual`, `functions`, and `version`), compiled for both `x86_64-linux-gnu` (manylinux-compatible) and `darwin` (macOS). These binaries ensure platform compatibility.
- **Metadata:** The `dist-info` directory contains essential metadata files, including:
  - `METADATA`: Provides details about the package, such as name, version, and dependencies.
  - `WHEEL`: Specifies compatibility standards and wheel-specific metadata.
  - `RECORD`: Contains file integrity information, including hashes and file paths.

11.3 PyPI Upload

The `dual_autodiff_x` package was uploaded to PyPI under the name `rsr45-dual-autodiff-x`, allowing users to install it easily via:

```
pip install rsr45-dual-autodiff-x
```

The package is available at the following link: https://pypi.org/project/rsr45-dual-autodiff-x/

11.3.1 Key Features

- Efficient dual number arithmetic.
- Comprehensive mathematical functions.
- Automatic differentiation.
- Performance optimisation with Cython for enhanced speed.

This ensures the package is accessible for scientific and computational tasks, promoting usability and reproducibility.

12 Conclusion

This project successfully implemented the `dual_autodiff` package, providing a method for forward-mode automatic differentiation using dual numbers. Through a structured approach, key objectives such as developing the `Dual` class, performing differentiation, creating a Cythonized version, and packaging the library for Linux via `cibuildwheel` were accomplished.

The pure Python implementation of the `dual_autodiff` package offered ease of use and flexibility, while the Cythonized version (`dual_autodiff_x`) significantly enhanced performance, as demonstrated in the quantitative comparisons. By leveraging `cibuildwheel`, Linux-compatible manylinux wheels were created, ensuring broad usability across platforms. The package was also uploaded to PyPI, simplifying installation and distribution.

The project emphasised best practices in software development, including testing with `pytest`, documentation via Sphinx, and the use of modular project structures. Practical examples and performance insights were integrated into a tutorial notebook, ensuring the package's utility in real-world scenarios.

This coursework reinforced critical skills in computational physics, software optimisation, and development workflows. The experience highlighted the importance of adaptability and problem-solving, particularly when addressing compatibility challenges and optimising performance.

Future work could extend the package's capabilities to support higher-order derivatives or alternate differentiation modes. Additionally, further optimisation and support for additional architectures, such as ARM-based platforms, would expand its applicability.

In conclusion, the `dual_autodiff` package is an efficient and versatile tool for automatic differentiation, with potential applications in optimisation, machine learning, and computational physics.

Declaration of Use of AI Tools

I declare that all ideas, methods, and approaches used in this project were conceived and developed independently through my own research and understanding. The work presented here is entirely my own.

The following AI tools were used solely for support purposes:

- **ChatGPT:** Used to refine code snippets, assist with LaTeX formatting, and improve the clarity and structure of the written report. All content generated by this tool was critically evaluated and adapted as necessary to ensure accuracy and alignment with my original ideas.
- **GitHub Copilot:** Used to suggest coding snippets and assist the implementation of functions. All suggested code was reviewed, modified, and tested to meet the specific requirements of this project.

These tools were used strictly to enhance productivity and ensure clarity. They did not contribute to the conceptualisation of the project, the development of ideas, or the formulation of methods, all of which were my own work.

References

[1] RE Wengert. "A Simple Automatic Derivative Evaluation Program". In: *Communications of the ACM* 7.8 (1964), pp. 463–464.

[2] Atılım Güneş Baydin et al. "Automatic Differentiation in Machine Learning: A Survey". In: *Journal of Machine Learning Research* 18.153 (2018), pp. 1–43.

[3] Mikael Mortensen and Hans Petter Langtangen. "High Performance Python for Direct Numerical Simulations of Turbulent Flows". In: *arXiv preprint arXiv:1602.03638* (2016). URL: `https://arxiv.org/abs/1602.03638`.