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 optimization, 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-optimized 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 was crucial for enabling seamless execution of packages and tools designed for x86_64 systems.

To achieve this:

- **Rosetta Installation:** Rosetta, an emulation layer by Apple, 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.

3 Theoretical Background

3.1 Dual Numbers

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

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

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:

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

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

3.2 Automatic Differentiation

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

$$f(x) = f(v + \dot{v}\epsilon) = f(v) + f'(v)\dot{v}\epsilon.$$

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

This principle extends to composite functions via the chain rule:

$$f(g(v + \dot{v}\epsilon)) = f(g(v)) + f'(g(v))g'(v)\dot{v}\epsilon.$$

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 overrides arithmetic operators for seamless integration. 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 key 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...)
```

4.2 Utility Functions

To enhance usability:

- `functions.py` provides aliases for mathematical functions.
- `base.py` includes helper functions like:
  - `is_dual_instance(value)`: Checks if a value is a `Dual` instance.
  - `ensure_dual(value)`: Wraps non-`Dual` values into a `Dual` object.

5 Project Structure and Packaging

5.1 Repository Organization

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 organizes the project as follows:

- `dual_autodiff/`: Core implementation, including modules like `dual.py`, `functions.py`, and `base.py`.
- `tests/`: Unit tests for core modules.
- `docs/`: Documentation files, including Sphinx configurations.
- `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.

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 Implementation of `dual.py`

- **Arithmetic Operations:** Operators (`+`, `-`, `*`, `/`) for mathematical operations. For example:

```
x = Dual(2, 1); y = Dual(3, 2)
z = x + y  # Dual(real=5, dual=3)
```

- **Mathematical Functions:** Implements `sin`, `cos`, `log`, `exp`, `sqrt`, etc., extended to dual numbers. For instance:

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

- **Power and Root:** Supports scalar powers and square root computations:

```
x = Dual(4, 1)
result = x.sqrt()  # Dual(real=2.0, dual=0.25)
```

- **Error Handling:** Ensures mathematical operations like `log` and `sqrt` are only applied within valid domains.

Example:

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

can be evaluated directly using:

```
x = Dual(2, 1)
f_x = x.log() + x**2
```

7 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.

7.1 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/`

7.2 Installing the Package

Install the package via `pip`:

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

7.3 Testing the Installation

Verify functionality:

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

8 Differentiating a Function

8.1 Function Definition

The target function for differentiation is:

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

The derivative of this function, computed analytically, is:

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

8.2 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.

8.3 Results

8.3.1 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:

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

8.3.2 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:

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

8.3.3 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.

8.4 Comparison with Numerical Differentiation

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

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

  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.

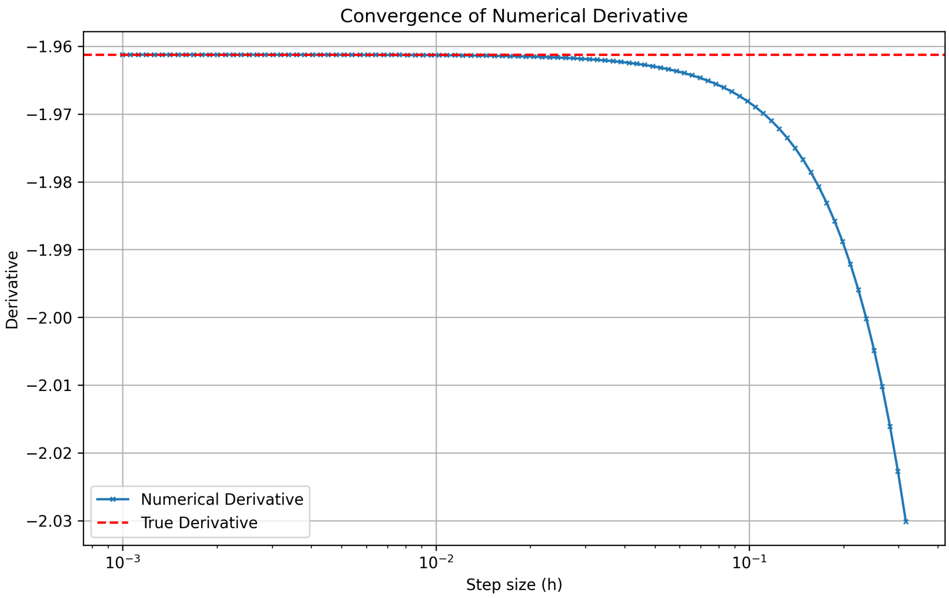

Figure 1: Convergence of the numerical derivative for decreasing step sizes. The red dashed line indicates the true derivative obtained using dual numbers. The numerical derivative converges to the true value for small step sizes, but diverges due to round-off errors as h becomes excessively small.

9 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.

9.1 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.

9.2 Outcome

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

10 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.

10.1 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`: Detailed 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.
  - `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 static assets, search functionality, and individual pages:
    - `index.html`: The main landing page.
    - `dual_autodiff.html`: Detailed API reference.
    - `tutorial.html`: The tutorial section with examples.
    - `apple_silicon_x86_setup.html`: Instructions for configuring the development environment.
    - `_static/`: Contains CSS, JavaScript, and image assets for styling and functionality.

10.2 Generated Output

The generated HTML documentation features:

- **Landing Page:** An overview of the project with links to tutorials and references.
- **API Reference:** Detailed documentation for all modules, classes, and functions in the package.
- **Tutorial:** A step-by-step guide, showcasing practical examples of using the package.

- **Environment Setup Guide:** Instructions for configuring the development environment on Apple Silicon devices.

The Sphinx-generated documentation ensures clarity and accessibility, providing users with a detailed understanding of the `dual_autodiff` package. It combines automatically generated API references with user-friendly tutorials, making it an essential resource for both developers and users.

11 Cythonizing the Package

11.1 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.

11.1.1 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`).

11.1.2 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 .
   ```

11.2 Performance Insights

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

11.2.1 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 analyze gradients of execution time with respect to array length.

11.2.2 Observations

Figure 2 illustrates the performance comparison:

- The Cythonized version exhibited significantly 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.

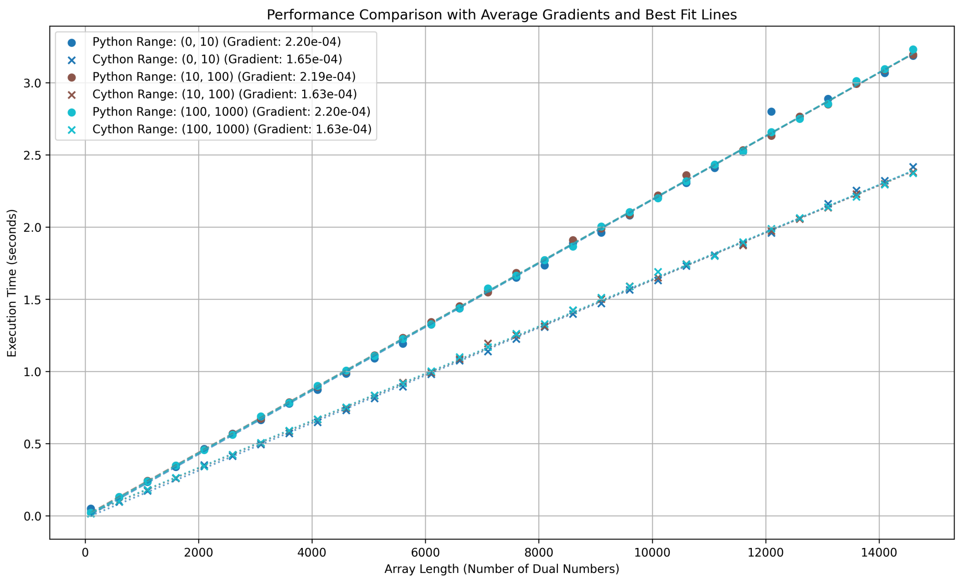

Figure 2: Performance comparison between the pure Python and Cythonized versions of `dual_autodiff`. The gradients indicate the rate of increase in execution time with array length.

11.3 Conclusion

Cythonization reduced execution time and improved scalability by compiling Python code into efficient C extensions. These results align with findings from related works, such as Mortensen and Langtangen [3], which demonstrated that Cythonized code can achieve performance comparable to low-level languages like C++.

11.4 Analysis of Gradients

The gradients of the execution time with respect to the array length, as indicated in the legend of Figure 2, provide a quantitative measure of the computational efficiency of the Python and Cythonized versions. These gradients represent the rate at which execution time increases with the number of dual numbers in the array. The following insights can be drawn from the gradients:

11.5 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, minimizing 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.

12 Building Wheels for Linux

To create specific wheels for the `dual_autodiff_x` package targeting `cp310-manylinux_x86_64` and `cp311-manylinux_x86_64`, the process was carried out manually on the University of Cambridge's CSD3 cluster due to compatibility issues on macOS M4.

12.1 Steps for Building the Wheels

1. **Building the Python 3.10 Wheel:**
   - Python 3.10 was built from source and installed in `$HOME/python310`:

     ```
     ./configure --prefix=$HOME/python310 --enable-optimizations
     make -j$(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 wheelho
```

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 cyth
   ```

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

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

12.2 Contents of the Wheel

The built wheels for Python 3.10 and 3.11 were inspected to ensure they contained the necessary compiled binaries and metadata for distribution. The key contents included:

- **Compiled Binaries:** The `dual_autodiff` directory contained shared object files (`*.so`) for the core modules (`base`, `dual`, and `functions`), ensuring optimized performance without exposing the source code.
- **Metadata:** The `dist-info` directory included essential metadata files such as:
  - `METADATA`: Package details like name, version, and dependencies.
  - `WHEEL`: Compatibility and wheel-specific metadata.
  - `RECORD`: File integrity and hash information.

12.3 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
```

12.3.1 Key Features

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

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

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`.