C1 Research Computing - Coursework Assignment

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

18 December, 2024

Word Count: 2959

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 mathematical principles behind dual numbers, and the implementation details of the package.

The structure of this report follows that of the coursework problem sheet.

Theoretical Background

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 value*.

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

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

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

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:

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

1 Project Structure and Packaging

1.1 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:

1.1.1 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.

2 Building and Installing the Package

The `pyproject.toml` file is used to manage the configuration and metadata of the project. The key sections are described below:

- `[build-system]`: Specifies the tools required to build the package, including:
  - `setuptools`: For packaging and distribution.
  - `wheel`: For building wheel distributions.
  - `setuptools_scm`: For dynamic versioning based on the repository's state.
  - `build`: A modern tool for building Python packages.
- `[project]`: Contains essential project metadata, including:
  - **Name:** `rsr45_dual_autodiff`.
  - **Description:** *A Python package for forward-mode automatic differentiation using dual numbers.*
  - **Author:** `Raunaq Rai` (`rsr45@cam.ac.uk`).
  - **License:** `MIT License`, specified in the `LICENCE` file.
  - **Dependencies:**
    - `numpy>=1.20.0`: For numerical computations.
    - `pytest>=6.0`: For unit testing.
    - `argparse`: For command-line interface support.
  - **Python Version:** Requires Python 3.9 or higher.
  - **Keywords:** *automatic differentiation, dual numbers, forward-mode.*
  - **URLs:** Link to the gitlab repository.
- `[project.scripts]`: Defines a command-line interface entry point for the package. The `dual_autodiff` command is linked to `dual_autodiff.cli:main`, enabling users to interact with the package via the terminal.

This configuration ensures that the package is documented, modular, and easy to build, install, and distribute. Using `pyproject.toml` aligns the project with modern Python packaging standards, making it accessible to a wide range of users and developers.

3 Implementation of Dual Numbers and Operations

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.

3.1 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.

3.2 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`).

3.3 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.

3.4 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`.

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

- `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.
  - `ensure_dual(value)`: A function that wraps a non-`Dual` value into a `Dual` object with its derivative initialised to zero.

4 Making the Code into a Package

To build and install the package, we made use of the `pyproject.toml` file for configuration, the `requirements.txt` file for dependencies, and the structured organisation of the project.

4.1 Steps to Build and Install the Package

The package was built and installed as follows:

- **Install Build Tools:** Required tools like `setuptools` and `wheel` were installed using:

```
pip install build
```

- **Build Distributions:** Source and wheel distributions were created using:

  ```
  python -m build
  ```

  The outputs (`.tar.gz` and `.whl`) were saved in the `dist/` directory.

- **Editable Installation:** The package was installed in editable mode using:

  ```
  pip install -e .
  ```

  This allows changes to the source code to be directly reflected without reinstallation, streamlining development and testing.

4.2 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.

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/`

Install the package via `pip`:

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

5 Differentiating a Function

5.1 Function Definition

The target function for differentiation is:

$$f(x) = \log(\sin(x)) + x^2 \cos(x), \tag{6}$$

The derivative of this function, computed analytically, is:

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

5.2 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.

5.3 Results

5.3.1 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:

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

5.3.2 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:

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

5.3.3 Comparison

The results obtained using dual numbers closely match the analytically computed values, confirming the correctness of the dual number implementation. The slight discrepancy in the derivative (2×10^{-16}) is attributed to double numerical precision limitations inherent to computers. As explained by Kahan [3], floating-point numbers are represented using a finite number of bits under the IEEE (Institute of Electrical and Electronics Engineers) 754 standard, resulting in small rounding errors during arithmetic operations. While these errors are negligible for most practical purposes, they can accumulate in complex calculations, causing slight discrepancies between computed and exact values.

5.4 Comparison with Numerical Differentiation

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

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \tag{8}$$

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

Figure 1 shows the behavior of the numerical derivative as the step size increases. The red dashed line represents the true derivative obtained using dual numbers.

The numerical derivative converges to the true value for small step sizes (approximately $< 10^{-2}$) but diverges as h increases due to round-off errors. In contrast, the dual number method provides a stable and precise derivative, unaffected by the limitations of finite differences. Moreover, dual numbers compute the derivative in a single step, making the method both computationally efficient and less prone to errors.

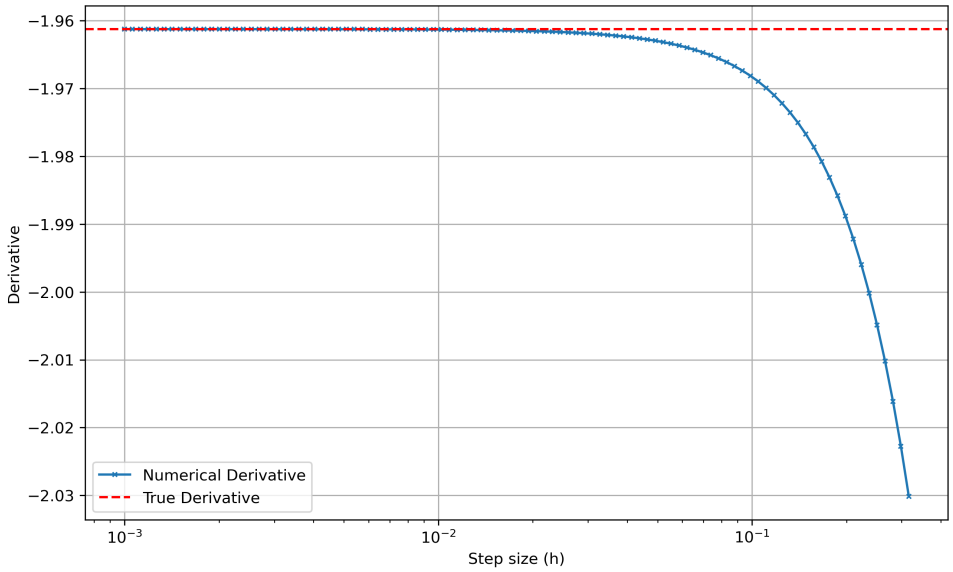

Figure 1: Numerical derivative for increasing step sizes (h). The red dashed line indicates the true derivative obtained using dual numbers.

6 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.

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.

The tests validate that the `dual_autodiff` package functions as expected under various scenarios. They also confirm that dual numbers provide accurate derivatives. In total 18 tests were created - all passing successfully.

7 Project Documentation with Sphinx

The `dual_autodiff` package documentation was generated using Sphinx, providing clear explanations, examples, an API reference, and a tutorial notebook.

7.1 Setup and Configuration

The `docs/` directory was created using `sphinx-quickstart`, which generated configuration files such as `Makefile`, `conf.py`, and initial templates for reStructuredText (`.rst`) files. Key configurations included enabling extensions like `sphinx.ext.autodoc` (for API

generation), `sphinx.ext.napoleon` (for docstring style), and `sphinx.ext.viewcode` (for linking code). Additionally, `nbsphinx` was added to integrate the tutorial notebook.

7.2 Documentation Structure

The `source/` directory contains:

- `index.rst`: Main landing page linking all sections.
- `dual_autodiff.rst`: Auto-generated API documentation.
- `modules.rst`: Auto-generated list of modules.
- `tutorial.rst`: User guide linked to the tutorial notebook.
- `dual_autodiff.ipynb`: Notebook showcasing examples and usage.
- `implementation.rst`: Theoretical background and implementation details.
- `apple_silicon_x86_setup.rst`: Setup steps for Apple Silicon devices.

7.3 Generating the Documentation

HTML documentation was generated by running:

```
make html
```

This compiled the `.rst` files and `.ipynb` notebook into structured HTML files in the `build/html/` directory.

7.4 Output and Features

The final documentation includes:

- **Landing Page:** Overview with links to all sections.
- **API Reference:** Detailed documentation of modules and functions.
- **Tutorial Page:** Practical examples and usage guide.
- **Implementation Page:** Theoretical background and key concepts.

The integration of `autodoc` and `nbsphinx` ensures docstrings from the codebase and the tutorial notebook are included, enhancing accessibility for users.

8 Cythonizing the Package

8.1 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.

8.1.1 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`).

8.1.2 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 .
```

The `.so` and `.c` files are excluded from the GitLab repository because they are platform-specific and can be regenerated from the `.pyx` source files using Cython. Including these files would unnecessarily increase the repository size and could lead to compatibility issues, as the compiled files depend on the system architecture and compiler settings of the development environment.

9 Performance Insights

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

9.1 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.

9.2 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.

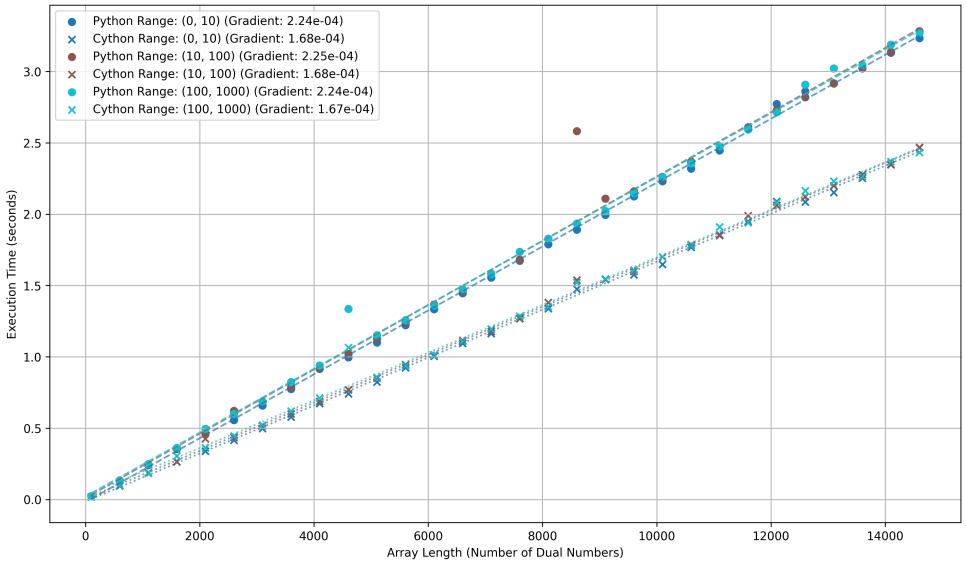

Figure 2: Performance comparison between the pure Python and Cythonized versions of `dual_autodiff`. The gradients indicate the rate of increase in execution time with array length.

Figure 2 shows how execution time scales with array length. Python implementation gradients are higher (2.20×10^{-4}), showing faster growth in execution time, unaffected by the range of real parts. In contrast, Cythonized gradients (1.63×10^{-4}) are lower, showing better scalability and suitability for larger datasets.

9.3 Conclusion

Cythonization improves the scalability and efficiency of the `dual_autodiff` package by converting Python code into optimised C extensions. The reduced overhead and compiled nature of Cython minimise dynamic type-checking, leading to lower execution times and better handling of computationally intensive tasks.

Applications like machine learning, requiring differentiation of large parameter arrays, and computational physics, involving derivatives of discretised data points, benefit from this improved performance. Faster computation reduces runtime and resource usage, making the Cythonized version ideal for large-scale problems.

These results align with prior studies [4], demonstrating that Cython achieves performance comparable to low-level languages like C++. The findings validate the advantages of Cythonization for real-world, high-performance applications.

10 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 used 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 produced linux-compatible wheels, 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.

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 employs Docker (running in the background) to efficiently build compatible wheels, outputting the generated wheels to the `wheelhouse/` directory.

The resulting wheels were named:

- rsr45_dual_autodiff_x-0.1.16-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
- rsr45_dual_autodiff_x-0.1.16-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl

The naming convention of the wheels is determined automatically by `cibuildwheel`:

- **Package Name:** `rsr45_dual_autodiff_x`, which uniquely identifies the package.
- **Version:** `0.1.16`, reflecting the current version of the package.
- **Python Tag:** `cp310` and `cp311`, denoting compatibility with Python 3.10 and 3.11.
- **ABI Tag:** Repeated `cp310` or `cp311`, indicating compatibility with the specific Python Application Binary Interface (ABI).
- **Platform Tag:** `manylinux_2_17_x86_64.manylinux2014_x86_64`, signifying compliance with the `manylinux2014` standard for Linux distributions.

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.

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 Uploading Wheels to GitLab Repository

The `wheelhouse/` directory contains the wheels built for Python 3.10 and 3.11, specifically targeting the `cp310-manylinux_x86_64` and `cp311-manylinux_x86_64` platforms. To upload these wheels to the GitLab repository, I first ensured they were properly located in the `dual_autodiff_x/wheelhouse/` directory. I then staged the files using `git add`, committed the changes with a descriptive message indicating the addition of the manylinux wheels, and pushed the changes to a dedicated branch.

This approach allowed me to validate the wheels and ensure their correctness before merging the branch into the main repository. Once I confirmed the wheels worked as expected and passed all tests, I merged the branch into the main repository, maintaining a reliable and well-documented development history.

11.1 Installation and Validation

After downloading the wheels from the GitLab repository, the package can be installed on any Linux system, such as CSD3 or a local laptop, using the following command:

```
pip install rsr45_dual_autodiff_x_<name_of_wheel>.whl
```

The prefix `rsr45` in the wheel name reflects the unique naming convention required to identify this package as part of my coursework submission, distinguishing it from any existing or similarly named packages on PyPI or other systems. This ensures that the package is easily attributable to me and avoids potential conflicts with other versions of `dual_autodiff_x`.

To verify the installation, the examples provided in the tutorial notebook `dual_autodiff.ipynb` were executed.

12 Conclusion

This project successfully developed the `dual_autodiff` package, implementing forward-mode automatic differentiation using dual numbers. Key objectives, including creating the `Dual` class, enhancing performance through Cythonization, and packaging with `cibuildwheel`, were accomplished. The resulting packages, `rsr45-dual-autodiff` and `rsr45-dual-autodiff-x`, are publicly available on PyPI for easy installation and use.

The Python implementation prioritised ease of use, while the Cythonized version (`dual_autodiff_x`) improved performance which was quantitatively evaluated. Best practices, such as unit testing, detailed documentation via Sphinx, and a tutorial notebook, ensured the package's robustness and practical applicability.

This coursework strengthened skills in computational physics, software optimisation, and problem-solving. Future work could expand the package to support higher-order derivatives, alternate differentiation modes, and additional architectures, such as ARM-based platforms. Overall, the `dual_autodiff` package is a versatile tool with applications in optimisation, machine learning, and computational physics.

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] W. Kahan. *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*. Work in Progress: October 1, 1997, 3:36 am. Berkeley CA 94720-1776, 1997.

[4] 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`.

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, understanding and engagement with lecture and supervision material. The work presented here is entirely my own.

The following AI tools were used for support purposes:

- **ChatGPT:** Used to refine code snippets, assist with LaTeX formatting, and improve the clarity and structure. All content generated by this tool was critically evaluated and adapted.
- **GitHub Copilot:** Used to suggest coding snippets and assist the implementation of functions. All suggested code was reviewed, modified, and tested.