UNIVERSITY OF CAMBRIDGE

C1 Research Computing Coursework

xl628

Department of Physics, University of Cambridge

December 18, 2024

Document Statistics:
Words in text: 2316
Words in headers: 79
Words outside text (captions, etc.): 29
Number of headers: 33
Number of floats/tables/figures: 6
Number of math inlines: 24
Number of math displayed: 13

1 Introduction

Automatic differentiation is a fundamental tool for optimizing learning algorithms in deep neural networks. Forward-mode automatic differentiation using dual number is efficient in scenarios with few input variables and many output variables. This report presents the development of a dual-number-based automatic differentiation package, `dual_autodiff`, along with its Cythonized counterpart, `dual_autodiff_x`.

2 Project Structure and Configuration

2.1 Project Structure

The project directory is organized as follows:

```
xl628/
├── docs/
│   ├── Makefile
│   ├── build/
│   ├── make.bat
│   └── source/
├── dual_autodiff/
│   ├── dual_autodiff/
│   ├── pyproject.toml
│   └── tests/
├── dual_autodiff_x/
│   ├── dual_autodiff_x/
│   ├── pyproject.toml
│   ├── setup.py
│   └── wheelhouse/
```

```
├── notebooks/
├── report/
├── README.md
├── .gitignore
└── requirements.txt
```

The main contents are as follows:

- `docs/`: Contains the Sphinx documentation for the project.
- `dual_autodiff/`: Contains the implementation of the dual-number automatic differentiation package.
  - `dual_autodiff/dual_autodiff/__init__.py`: Initializes the package, imports the `Dual` class, and displays the package version.
  - `dual_autodiff/dual_autodiff/dual.py`: Implements the base class `Dual` and the associated methods required for automatic differentiation.
  - `dual_autodiff/pyproject.toml`: Configuration file defining package dependencies and the build system.
- `dual_autodiff_x/`: Contains the Cythonized version of the `dual_autodiff` package.
- `notebooks/`: Includes example notebooks demonstrating the functionality of the project. Examples include differentiating specific functions and performance comparisons between `dual_autodiff` and `dual_autodiff_x`.
- `report/`: Holds reports and related files.
- `README.md`: The project's main README file.
- `requirements.txt`: Lists project dependencies.
- `.gitignore`: Specifies files and directories to be excluded from version control.

2.2 Configuration

`pyproject.toml` is a configuration file used by packaging tools [8]. There are three possible TOML tables in the configuration file: `build-system`, `project`, and `tool`.

- `build-system`: Specifies the build backend and its required dependencies.
  - In this project, the build backend is `setuptools.build_meta`, and the build dependencies include `setuptools` and `wheel`.
- `project`: Contains project metadata such as the name, version, description, authors, and dependencies.
- `tool`: Holds tool-specific configuration subtables, which are defined by the corresponding tools.
  - For example, we configure `pytest` to run tests under the `[tool.pytest.ini_options]` table.

3 Implementation of Dual Number Automatic Differentiation

3.1 Mathematical Background[1]

Dual Numbers

A dual number is defined as an ordered pair (a, b), typically written as $a + b\epsilon$, where:

- $a, b \in \mathbb{R}$ (the real numbers)
- ϵ is the dual unit with the property $\epsilon^2 = 0$

Basic Operations

The algebraic structure of dual numbers defines the following basic operations:

1. **Addition and Subtraction**:

$$(a + b\epsilon) \pm (c + d\epsilon) = (a \pm c) + (b \pm d)\epsilon$$

2. **Multiplication**:

$$(a + b\epsilon)(c + d\epsilon) = ac + (bc + ad)\epsilon \quad (\text{since } \epsilon^2 = 0)$$

3. **Division** (for $c \neq 0$):

$$\frac{a + b\epsilon}{c + d\epsilon} = \frac{a}{c} + \frac{bc - ad}{c^2}\epsilon$$

4. **Power Rules**:

$$(a + b\epsilon)^n = a^n + nba^{n-1}\epsilon$$

Automatic Differentiation Theory

The key insight of forward-mode automatic differentiation is that dual numbers naturally encode derivative information through the formula. Given $x = a + b\epsilon$, and a function $f(x)$, we have:

$$f(x) = f(a + b\epsilon) = f(a) + f'(a)b\epsilon$$

And so, $f'(a)$ can be extracted from the coefficient of the ϵ term in the dual number $f(a + b\epsilon)$. This formula emerges from the Taylor series expansion and the property of ϵ:

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

The chain rule emerges naturally from dual number arithmetic. For composite functions like $h(x) = f(g(x))$:

$$h(a + b\epsilon) = f(g(a + b\epsilon)) = f(g(a) + g'(a)b\epsilon) = f(g(a)) + f'(g(a))g'(a)b\epsilon$$

where we get $f'(g(a))g'(a)$ as the derivative of the composite function.

Elementary Functions

Using the Taylor series expansion and the property of ϵ, we can derive the following formulas for the elementary functions:

1. **Exponential**:

$$\exp(a + b\epsilon) = e^a + be^a\epsilon$$

2. **Logarithm** (for $a > 0$):

$$\log(a + b\epsilon) = \log(a) + \frac{b}{a}\epsilon$$

3. **Trigonometric Functions**:

   - $\sin(a + b\epsilon) = \sin(a) + b\cos(a)\epsilon$
   - $\cos(a + b\epsilon) = \cos(a) - b\sin(a)\epsilon$
   - $\tan(a + b\epsilon) = \tan(a) + \frac{b}{\cos^2(a)}\epsilon$

3.2 Implementation

The `dual.py` file contains the implementation of the base class `Dual` along with the necessary methods for automatic differentiation. The class supports the following functionalities:

- **Basic Arithmetic Operations**
  - Addition: `__add__`, `__radd__`
  - Subtraction: `__sub__`, `__rsub__`
  - Multiplication: `__mul__`, `__rmul__`
  - Division: `__truediv__`
  - Exponentiation: `__pow__`
- **Elementary Functions:**
  - Exponential: `exp`
  - Logarithm: `log`
  - Trigonometric: `sin`, `cos`, `tan`
- **Support Methods:**
  - `__init__`: Initializes a dual number with real and dual components. By default, the dual part is set to 1.0 to facilitate derivative computations.
  - `__repr__` and `__str__`: Provide string representations in the format `Dual(real=x, dual=y)`.
  - `__neg__`: Implements unary negation as $-(a + b\epsilon) = -a + (-b)\epsilon$.

Each method includes proper type checking and error handling, raising appropriate exceptions for invalid operations. The implementation supports operations between dual numbers, and also supports some operations between dual numbers and regular numbers (float/int).

Below are basic examples demonstrating the functionality:

```python
# basic operations
x = Dual(2, 1)
y = Dual(3, 1)
print(x + y)

# trigonometric functions
print(Dual(2, 1).sin())

# alternative way to use the method
x = Dual(2, 1)
print(Dual.sin(x))
```

The corresponding output is:

```
Dual(real=5, dual=2)
Dual(real=0.9092974268256817, dual=-0.4161468365471424)
Dual(real=0.9092974268256817, dual=-0.4161468365471424)
```

3.3 Example: Differentiating a Function

The `notebooks/differentiate_function.ipynb` notebook provides a practical example of using the `Dual` class to compute the derivative of a function:

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

with the analytical derivative:

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

In this example, we compare three derivative computation methods: analytical differentiation, numerical differentiation, and automatic differentiation using the `Dual` class. The numerical derivative is calculated using the central difference formula:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h},$$

We measure the accuracy of the method using relative error and evaluate the derivatives at $x = 1.5$. The results are illustrated in Fig. 1.

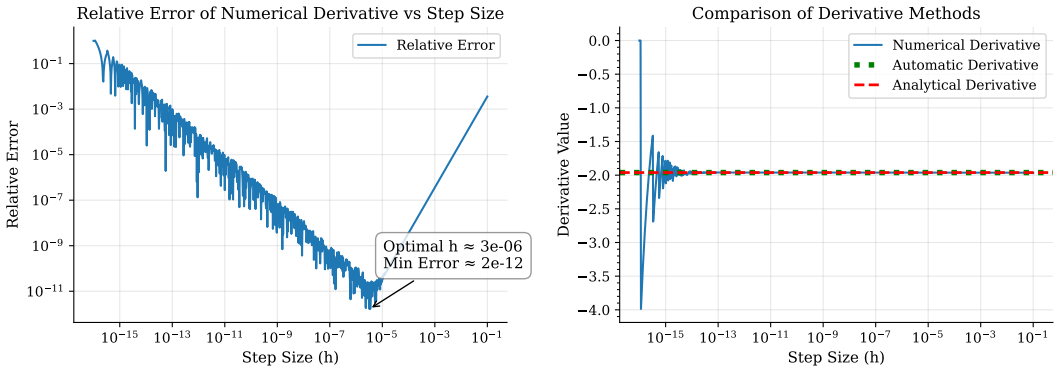

Figure 1: Comparison of Derivative Methods

Automatic differentiation using `Dual` class provides a more accurate result than numerical differentiation, with significantly smaller relative errors:

```
Analytical derivative at x=1.5: -1.9612372705533612
Value at x=1.5: 0.15665054756073515
AutoDiff derivative at x=1.5: -1.9612372705533612
AutoDiff relative error: 0.0
```

When reducing the step size, the error in the numerical derivative does not decrease as expected but instead oscillates and increases. This instability suggests that numerical differentiation is sensitive to step size and may fail for singular or non-differentiable functions. Conversely, increasing the step size beyond a certain point also results in larger errors due to loss of precision.

From the figure, the optimal step size for this function is approximately 3×10^{-6}.

4 Package Development

4.1 Development Installation

To install the package for development, navigate to the root directory (`xl628/dual_autodiff/`) and run the following command:

```
pip install -e .
```

This will install the package in editable mode.

To verify the installation, open a Python shell, import the `Dual` class, and create a `Dual` object:

```
>>> from dual_autodiff.dual_autodiff.dual import Dual
>>> x = Dual(2.0, 1.0)
>>> print(x)
Dual(real=2.0, dual=1.0)
```

If the installation is successful, no errors should occur.

4.2 Testing Strategy

The test suite is implemented using `pytest`. Before running the tests, ensure that all development requirements, including `pytest` and `pytest-cov`, are installed. Run the following command at the root directory (e.g., `xl628/`):

```
pip install -r requirements.txt
```

To execute the tests, use:

```
pytest
```

Run this command from the root directory of the package, i.e., `xl628/dual_autodiff/`.
The test suite includes:

- Basic initialization and string representation checks
- Arithmetic operations, including proper error handling
- Verification of elementary functions (e.g., trigonometric, exponential, logarithmic)
- Validation of chain rule implementation

Both the real and dual parts of calculations are tested for accuracy. Numerical results are validated using `pytest.approx()` with a relative tolerance of 1e-5. For details, refer to the source code in `xl628/dual_autodiff/tests/test_dual.py`.

To ensure conciseness and maintainability, we use `pytest`'s `parametrize` decorator to test multiple cases efficiently.

Test coverage reports are automatically generated using `pytest-cov` whenever the tests are executed.

All tests passed in my experiment.

5 Documentation

The documentation is built using `Sphinx` and `nbsphinx` with the `readthedocs` theme. It provides comprehensive details, including:

- Installation instructions
- Usage examples
- API reference
- A mathematical introduction to dual numbers and automatic differentiation

`Sphinx` is a powerful tool for generating structured and beautiful documentation using reStructuredText as its markup language [7]. The `nbsphinx` extension allows integration of Jupyter notebooks into the documentation [5].

5.1 Building the Documentation

To build the documentation, install the development requirements and run the following commands from the `docs` directory:

```
pip install -r requirements.txt
cd docs
make html
```

The generated HTML documentation will be available in the `docs/build/html` directory. To clean the build, use:

```
make clean
```

5.2 Documentation Examples

The *Getting Started* section includes basic examples demonstrating the use of the package for derivatives, mathematical operations, and elementary functions. More advanced examples are provided in the *Applications* section, where the `Dual` class is used to compute partial derivatives and gradients.

The gradient example serves as a solid foundation for further applications such as optimization, neural networks, and other numerical techniques.

6 Cythonization and Performance Analysis

6.1 Cythonized Package

The `dual_autodiff_x` package is the cythonized version of `dual_autodiff`. It is built using the `Cython` extension. [4][3]

Cython is a programming language designed to simplify the creation of C extensions for Python. It serves as a superset of Python, offering features like optional static type declarations and seamless integration with external C libraries.

Cython translates Python-like source code into optimized C/C++ code, which is then compiled into Python extension modules. This process results in significantly faster execution while maintaining Python's high-level functionality and developer-friendly syntax.

Cython is widely used in performance-critical research computing libraries, such as `NumPy`, `SciPy`, `Pandas`, and `scikit-learn` [6]. The key performance improvements stem from:

- **Static Typing:** Variables can be explicitly declared with C types to eliminate Python's dynamic type overhead.
- **Reduced Overhead:** Cython avoids Python's interpreter overhead, enabling faster function execution.
- **Compiler Optimizations:** Cython leverages C-level optimizations for loops and arithmetic operations.

In the `dual_autodiff_x` package, Cython source files (`.pyx`) are compiled to generate intermediate files (`.c` and `.cpp`) and shared object files (`.so`). Developers can optimize the `.pyx` files further by incorporating static type declarations and C libraries.

6.2 Performance Analysis

Performance comparisons between the `dual_autodiff` (Python version) and `dual_autodiff_x` (Cython version) are conducted in the `notebooks/performance_comparison.ipynb` notebook. The evaluation considers multiple aspects, including:

- Basic operations
- Mathematical functions
- Scaling behavior
- Memory usage
- Real-world applications

The summarized results are presented in Table 1. Overall, the cythonized version demonstrates substantial improvements in both speed and memory efficiency.

Table 1: Overall Performance Summary.

Category	Details
1. Basic Operations:	
Average speedup	1.91x
Best performing	Power (2.44x)
Average improvement	46.7%
2. Mathematical Functions:	
Average speedup	2.07x
Best performing	Logarithm (2.23x)
Average improvement	51.7%
3. Scaling Analysis:	
Maximum speedup	3.11x at chain length 100
Speedup trend	Increases with chain length
4. Memory Usage:	

Category	Details
Average memory ratio (Python/Cython)	1.87
Maximum memory savings	50.5%
5. Real-world Applications:	
Average speedup	2.31x
Best performing	Gradient Computation (2.61x)
Average improvement	56.3%
Overall Assessment:	
Global average speedup	2.33x
Performance improvement range	1.7x - 3.1x

Basic Operations

We evaluate the performance of basic operations, including creation, addition, multiplication, division, and exponentiation. The `measure_time` function from `performance_benchmark.py` is used with `n_runs=1,000,000` to collect execution time statistics.

The results are summarized in Fig. 2.

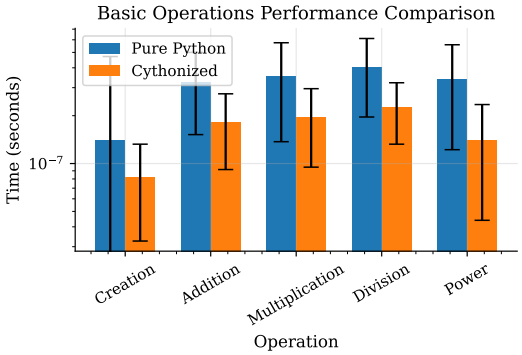

Figure 2: Performance of Basic Operations

The results show that the cythonized version significantly outperforms the Python version, achieving an average speedup of 1.91x. Among all operations, the power operation demonstrates the greatest improvement, with a speedup of 2.44x. This is likely because, unlike other operations, the implementation of the power function explicitly accepts `int` or `float` types but not `Dual` types. As a result, static typing using `double` could be applied in the Cython implementation, allowing the compiler to determine variable types at compile time, thereby eliminating runtime checks and improving computational efficiency.

Mathematical Functions

We also analyze the performance of mathematical functions, including the exponential, sine, cosine, and logarithm functions. Using the same `measure_time` function with `n_runs=1,000,000`, we collect execution time statistics.

The results are shown in Fig. 3.

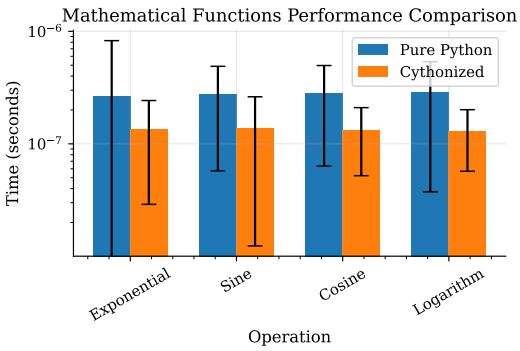

Figure 3: Performance of Mathematical Functions

The cythonized version shows strong performance improvements, with an average speedup of 2.07x. Among all functions, the logarithm function achieves the highest speedup at 2.23x. In addition to static typing, these improvement are due to the use of the C math library, which optimizes the computation of these mathematical functions.

Scaling Analysis

To assess scaling performance, we analyze computations with increasing chain lengths. This is particularly relevant in neural network backpropagation, where chain derivatives are common. For this experiment, `n_runs=100,000` is used to collect execution time statistics.

In this experiment, we construct a chain of functions based on $f(x) = \sin(\exp(\log(x)))$, where the chain is defined as $f(f(...f(x)...))$. We measure the performance for varying chain lengths.

```
def chain(x):
   result = impl.Dual(x, 1.0)
   for _ in range(n):
      result = result.sin().exp().log()
   return result
```

The chain lengths tested are `[1, 2, 5, 10, 20, 50, 100]`.

The results are presented in Fig. 4.

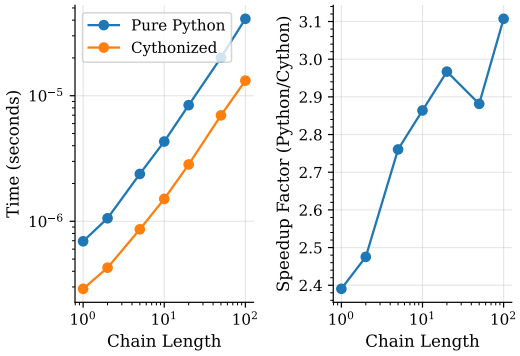

Figure 4: Scaling Performance with Increasing Chain Length

The analysis reveals that the speedup increases with chain length, reaching a maximum of 3.11x at the longest chain. This trend suggests that the optimized implementation scales efficiently with larger inputs, likely due to reduced overhead and improved data handling at scale.

Memory Usage

We measure memory usage by creating arrays of `Dual` objects of varying sizes. The `memory_usage` function from `performance_benchmark.py` is used for this purpose. The array sizes tested are:

```
array_sizes = [1000000, 5000000, 10000000]

def create_array(size, impl):
```

```
    return [impl.Dual(x_val, 1.0) for _ in range(size)]
```

The results are illustrated in Fig. 5.

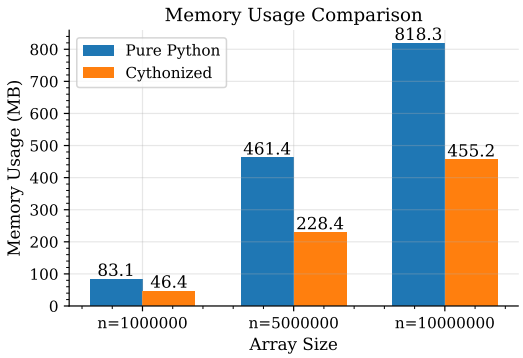

Figure 5: Memory Usage for Array Creation

The cythonized version significantly reduces memory usage, achieving a maximum reduction of 50.5%. This improvement can be attributed to more efficient memory management in Cython.

Applications

To assess real-world applicability, we benchmark three scenarios where derivatives are useful:

- Neural network forward propagation
- Gradient computation
- Physics simulation

Execution times are measured using the `measure_time` function with `n_runs=1,000,000`. The results are shown in Fig. 6:

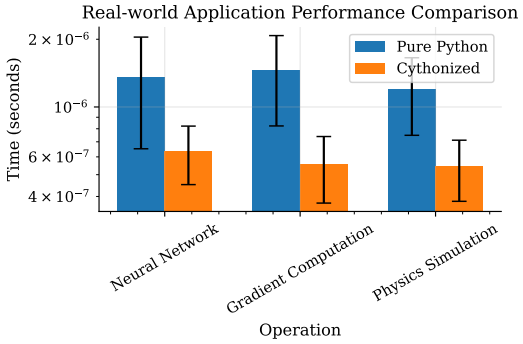

Figure 6: Application Performance Comparison

The results demonstrate significant performance gains in practical applications. Gradient computation achieves the largest improvement, with a speedup of 2.61x, underscoring the advantages of the optimized Cython implementation for computationally intensive tasks.

7 Linux Wheel Generation

We use `cibuildwheel` to generate the wheel for the `dual_autodiff_x` package.

`cibuildwheel` supports building wheels for CPython and PyPy across various platforms, including manylinux, musllinux, macOS, and Windows [2].

7.1 Wheel Generation

The wheel is generated using the following command:

```
CIBW_BUILD="cp311-manylinux_x86_64" CIBW_ARCHS="x86_64" cibuildwheel --platform linux
```

we can specify the Python version, the platform, and the architecture we want to build for.

Here:

- `cp` denotes CPython, and `311` specifies Python version 3.11.
- `manylinux` indicates the manylinux platform, a standardized environment for building Linux wheels.
- `x86_64` refers to the 64-bit x86 CPU architecture.

In this case, we build wheels for Python 3.10 and Python 3.11 on the manylinux platform with x86_64 architecture. The resulting wheels are stored in the `dual_autodiff_x/wheelhouse` directory.

Additionally, the `setup.py` file is configured as follows:

- `package_data` and `exclude_package_data` ensure that only the `.so` files are included, while the `.pyx` and `.py` source files are excluded to prevent their distribution.
- `zip_safe=False` is specified for wheel generation.

7.2 Wheel Installation on CSD3

To verify the generated wheel, we install it on the CSD3 server as follows:

```
(venv_EC1_311) [xl628@login-q-1 wheelhouse]$ pip install
    dual_autodiff_x-0.0.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
Processing
    ./dual_autodiff_x-0.0.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
Installing collected packages: dual-autodiff-x
Successfully installed dual-autodiff-x-0.0.1
```

Next, we validate the installation by importing the `Dual` class and performing a basic computation:

```
(venv_EC1_311) [xl628@login-q-2 ~]$ python3
Python 3.11.9 (main, Sep 24 2024, 09:39:15) [GCC 8.5.0 20210514 (Red Hat 8.5.0-22)]
    on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import dual_autodiff_x as df
dual_autodiff_x package version: 0.0.1
>>> x = df.Dual(2, 1)
>>> sinx = x.sin()
>>> print(sinx)
Dual(real=0.9092974268256817, dual=-0.4161468365471424)
```

As demonstrated, the `Dual` class operates correctly on the CSD3 server, confirming that the wheel has been built successfully.

Furthermore, it has been verified that the package runs well with the example notebooks provided in the `docs` directory on the CSD3 server.

8 Conclusion

In summary, the development of the `dual_autodiff` and `dual_autodiff_x` packages illustrates the practicality and efficiency of forward-mode automatic differentiation using dual numbers. The Cythonized version, `dual_autodiff_x`, achieves significant performance improvements in execution speed and memory efficiency.

A Use of AI Tools

The following describes how AI tools were utilized in the preparation of this report:

- **ChatGPT 4o**
  - **Drafting** - Used for drafting certain sections of the report, including the Mathematical Background section, Performance Analysis section, Introduction, Conclusion, Appendices. Also used for drafting captions of figures and tables.
  - **Proofreading** - Reviewed grammatical correctness, improved sentence structure, and suggested alternative wordings for clarity across the report.
  - **LaTeX Assistance** - Helped resolve issues encountered during the LaTeX formatting process, such as debugging compilation errors, optimizing figure placement, and improving overall document layout.

B Template Information

This report was formatted using the LaTeX template from the given Example Coursework (by Michal Dorko). Some modifications were made.

References

[1] *Automatic differentiation - Wikipedia*. URL: https://en.wikipedia.org/wiki/Automatic_differentiation. (accessed: 18.12.2024).

[2] *cibuildwheel*. 2024. URL: https://cibuildwheel.pypa.io/en/stable/. (accessed: 17.12.2024).

[3] *Cython - an overview — Cython 3.1.0a1 documentation*. URL: https://cython.readthedocs.io/en/latest/src/quickstart/overview.html. (accessed: 13.12.2024).

[4] *Cython: C-Extensions for Python*. URL: https://cython.org/. (accessed: 13.12.2024).

[5] *nbsphinx: A Sphinx Extension for Jupyter Notebooks*. URL: https://nbsphinx.readthedocs.io/en/0.9.5/. (accessed: 13.12.2024).

[6] *Research Computing and Software Development — UoC MPhil Data Intensive Science and MPhil Economics and Data Science beta documentation*. URL: https://researchcomputing.readthedocs.io/en/latest/material/part15/notebook.html. (accessed: 13.12.2024).

[7] *Sphinx · PyPI*. URL: https://pypi.org/project/Sphinx/. (accessed: 13.12.2024).

[8] *Writing your pyproject.toml - Python Packaging User Guide*. URL: https://packaging.python.org/en/latest/guides/writing-pyproject-toml/. (accessed: 13.12.2024).