UNIVERSITY OF
CAMBRIDGE

Data Intensive Science
C1 Coursework

Liz Tan (eszt2)
Department of Physics, University of Cambridge
Michaelmas Term 2024

Word count: 2622 words

1 Introduction

Many applications now require computing derivatives of functions defined by programs. Derivatives reveal how a function changes locally, aiding tasks like parameter optimisation in machine learning or approximating integrals in numerical methods [1]. This report discusses the creation of a package that performs automatic differentiation using dual numbers. A dual number consists of a real part and a dual part, similarly to complex numbers. The dual part of the number is carried by ϵ with properties $\epsilon^2 = 0$. Dual numbers can be used to carry out automatic differentiation of functions due to this property.

We produced two packages that have exactly the same functionality: a pure Python package (`dual_autodiff`) and a Cython package (`dual_autodiff_x`). Both packages can perform differentiation and partial differentiation of functions provided the derivative of the function is known. There are performance differences between the packages with the Cython package performing operations faster. A test suite was produced for the pure Python package.

2 Automatic differentiation with dual numbers

Assume we want to find the derivative of a function $f(x)$ at point x_0. By substituting the dual number $d = x_0 + b\epsilon$ into the function and performing a Taylor expansion, we get the result

$$f(x_0 + \epsilon) = f(x_0) + f'(x_0)b\epsilon \tag{1}$$

where f' is the derivative of the function. The higher order terms are reduced to zero because $\epsilon^2 = 0$. Hence, if we set b to unity, the result of the function evaluated at the dual number d is another dual number where the real part is the function evaluated at x_0 and the dual part is the derivative of the function evaluated at x_0. Hence, dual numbers prove themselves useful in differentiation tasks because they provide the exact analytical derivative, as opposed to numerical differentiation where the result is approximate, and are computationally efficient.

3 Usage

3.1 Installation

There are two ways of installing the packages: installing the packages from the or installing from the `.whl` files. To install from the source, run the following in the terminal from the repository to install the pure Python package

```
cd python_package
pip install -e .
```

or to install the Cython package, run

```
cd cython_package
pip install -e .
```

Alternatively, the packages can be installed from the `.whl` files. These files are a binary distribution of the code and it is faster to install the package from these files. To install from the wheels for the pure Python package, run

```
cd python_package/dist
pip install dual_autodiff-0.1.0-py3-none-any.whl
```

There are two versions available for the Cython package specifically built for either Python version 3.10 or Python 3.11. They are both compatible with the Manylinux 2.17 standard, ensuring the package can run on a wide range of Linux systems. To install from these wheels, run

```
cd cython_package/wheelhouse
```

then

```
pip install dual_autodiff_x-0.1.0-cp310-cp310-manylinux_2_17_x86_64.
manylinux2014_x86_64.whl
```

for the compatibility with Python 3.10 or

```
pip install dual_autodiff_x-0.1.0-cp311-cp311-manylinux_2_17_x86_64.
manylinux2014_x86_64.whl
```

for compatibility with Python 3.11.

3.2 Implementation

Both the Cython and Python package contain the same modules and functionality, so for simplicity, we will cover the implementation of the Python package. There are two main modules in the package, a `.dual` module that stores the `Dual` class that handles dual numbers, and a `.tools` module that stores a library of base functions that are compatible with the dual class.

Python

```python
from dual_autodiff.dual import Dual
d = Dual(2, {'x': 1})
```

3.2.1 `Dual` class

The `Dual` class is initialised with two main components: a real component (float or integer) and a dual component (dictionary). By having multiple dual components, we can seamlessly perform automatic partial differentiation, which will be explained later. The initialisation includes type checking to ensure input validity, raising appropriate TypeError exceptions for invalid inputs.

3.2.2 Basic operations

The class implements fundamental arithmetic operations through operator overloading:

- Addition and subtraction
  - When adding dual numbers, the real parts and dual parts are added component-wise.
  - Maintains component-wise addition for dual parts using dictionary operations.
- Multiplication
  - Implements dual number multiplication: $(a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\epsilon$
  - If a dual number is multiplied by a scalar, the dual and real components are multiplied component-wise.
- Division
  - Implements dual number division: $(a + b\epsilon)(c + d\epsilon) = a/c + [(bc - ad)/c^2]\epsilon$
  - If the dual number is divided by a scalar, the dual and read components are each divided component-wise.
  - A scalar divided by a dual number results: $k/(a + b\epsilon) = k/a + (-kb/a^2)\epsilon$
  - The denominators real component must be non-zero, otherwise ZeroError is raised.
- Powers
  - Implements dual numbers to the power of scalars: $(a + b\epsilon)^n = a^n + (na^{n-1}b)\epsilon$.

- Implements dual numbers to the power of dual numbers: $(a + b\epsilon)^{(c+d\epsilon)} = a^c + a^{(bc/a)+d\ln a}\epsilon$
- Implements scalars to the power of dual numbers: $k^{a+b\epsilon} = k^a + k^{a\ln(k)b}\epsilon$
- The base must have a positive real component, otherwise ValueError is raised.

These binary operators handle both dual-dual operations and dual-scalar operations, with reverse operations defined for cases where scalars interact with dual numbers. Appropriate errors are raised for special cases like division by zero and invalid power operations. When operations are performed between dual numbers, each dual component in the dictionary is processed according to the appropriate arithmetic rules, with the dictionary structure automatically managing the combination of like terms.

3.2.3 `Numpy` integration

Through the `array_ufunc` method, `Dual` objects can be passed through `Numpy` functions while maintaining proper dual number arithmetic and derivative calculations. The `.tools` module that stores function-derivative pairs for various mathematical operations. Each function in the `tools_store` dictionary consists of two components: the function itself and its derivative. For instance, the sine function is paired with cosine as its derivative, while the exponential function is paired with itself. This pairing system allows for automatic computation of derivatives when applying these functions to dual numbers, as in Equation 1.
The base implementations include the trigonometric functions, `np.sin, np.cos, np.tan`, the hyperbolic functions, `np.sinh, np.cosh, np.tanh`, exponential and logarithmic functions `np.exp, np.log, np.sqrt` and the inverse trigonometric functions `np.arcsin, np.arccos, np.arctan`.

3.2.4 User-added custom functions

Users can add their own function-derivative pairs to the `tools_store` by using the `add_function` method, allowing the package to handle custom mathematical operations while maintaining automatic differentiation capabilities. This extensibility makes the implementation particularly versatile for various mathematical applications. Users can also remove functions with the `remove_function` method.

Python

```
add_function('name', function, function_derivative)
```

3.2.5 Partial differentiation

The `Dual` class handles partial differentiation through its dictionary-based structure of dual components. Each key in the dual dictionary corresponds to a different variable with respect to which we want to compute partial derivatives. Assume we want to compute the partial derivatives of a function, $f(x, y)$ at point (x_0, y_0), we first initialise two dual numbers

$$d_x = x_0 + \epsilon_x \tag{2}$$

$$d_y = y_0 + \epsilon_y \tag{3}$$

where ϵ_x and ϵ_y represent two different dual components and cannot be combined. If we evaluate the function with these two dual numbers, the result would be a dual number with two dual components.

$$f(x = d_x, y = d_y) = f(x_0, y_0) + f_x(x_0, y_0)\epsilon_x + f_y(x_0, y_0)\epsilon_y \tag{4}$$

where f_x is the partial derivative of f with respect to x and f_y is the partial derivative of f with respect to y. Hence, the coefficient of ϵ_i gives the partial derivative of f with respect to variable i. This method generalises to any number of variables.

Python

```python
def multi_variate_f(x,y):
    return result
d_x = (x0, {'x':1})
d_y = (y_0, {'y':1})

f_x = multi_variate_f(d_x, d_y).dual['x']
f_y = multi_variate_f(d_x, d_y).dual['y']
```

3.2.6 Demonstration

For an in-depth demonstration on how to use the package, including examples on adding custom functions, differentiation and partial differentiation, please see the notebook `dual_autodiff_demo.ipynb` located in the `python_package/docs/source/demos` directory.

4 Comparison to the numerical derivative

In the Jupyter notebook, `dual_autodiff_demo.ipynb`, we demonstrate the functionality of the class by differentiating the function

$$f(x) = \log(\sin(x)) + x^2 \cos(x) \tag{5}$$

at $x = 1.5$ using both the analytical derivative, the `Dual` class and numerical methods. The numerical methods used are the forward difference approximation, defined by

$$f'_f(x) \approx \frac{f(x+h) - f(x)}{h} \tag{6}$$

and the central difference approximation, defined by

$$f'_c(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{7}$$

The derivative found using the analytical derivative and using dual numbers was exactly the same, as was to be expected from Equation 1. Figure 1 shows the comparison of the numerical derivative of function 6 using the forward difference approximation and central difference approximation for difference step sizes, h.
The central difference approximation is more accurate, because the truncation error scales as $O(h^2)$ whereas for the forward difference approximation, the error scales as $O(h)$.

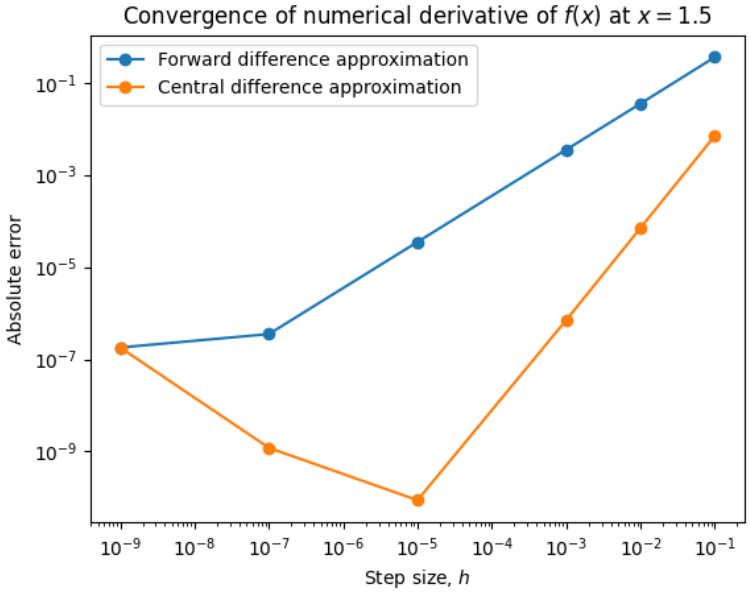

Figure 1: Comparison of the numerical derivative of function 6 using the forward difference approximation and central difference approximation for difference step sizes.

Smaller values of h will decrease the truncation error, as shown in the plot above. However, at smaller levels of h, the error seems to increase again. This is due to the way the computer deals with floating point numbers: the numerator $f(x+h) - f(x)$ or $f(x+h) - f(x-h)$ becomes a very small number leading to catastrophic cancellation. This error scales as $O(1/h)$, hence is larger for small h. To find optimum h, you need a balance between these two errors.

5 Development

5.1 Environment and dependencies

Python v3 was used in package development and a `conda` environment was used to manage the dependencies. The `requirements` directory contains `.txt` files of the libraries used in the development of the code.

5.2 Repository structure

The repository is structured as follows

- `python_package/`: contains source code for the pure Python `dual_autodiff` package.
  - `python_package/dist/`: contains `.whl` files for the Python package.
  - `python_package/tests/`: contains `pytest` suite for the Python package.
  - `python_package/docs/build/html/`: contains documentation for the package created using Sphinx.

  - `python_package/docs/source/demos/`: contains Jupyter notebooks with a tutorial on how to use the package as well as a comparison between the Cython and Python package.

- `cython_package/`: contains source code for the Cython `dual_autodiff_x` package.
  - `cython_package/wheelhouse/`: contains `.whl` files for the Python package.
- `Docker_testing/`: contains a Dockerfile that can be used to create a docker image used to test the package. The docker image downloads the `.whl` distributions of the Python and Cython packages and runs the tutorial and Cython/Python analysis on Jupyter notebooks.
- `requirements/`: contains `.txt` files with the external libraries used in the development of the package.

5.3 Version control

We used `git` as a version control system in the development of the packages. Most of the development was completed on a `development` branch, which was merged to the main branch once the packages were completed.

5.4 Documentation

Extensive documentation on how to use the package as well as a comparison of the performance on the Python and Cython packages were generated using Sphinx. We used the extension `sphinx.ext.napoleon` to generate documentation automatically from the docstrings in the source code. Additionally, Pandoc facilitated the integration of Jupyter notebooks into the HTML documentation, providing interactive examples that demonstrate the package's functionality and usage patterns.

5.5 Comprehensive `pytest` test suite

The test suite, built using the `pytest` framework, implements extensive coverage of both the dual number implementation and its supporting tools. For the `Dual` class, tests verify correct initialisation, basic arithmetic operations (including reverse operations with scalars), power operations, and integration with `numpy` universal functions. The suite leverages pytest's assertion system and fixtures to test edge cases like division by zero and invalid power operations with negative bases. For the `.tools` module, tests ensure proper management of the function store, including adding, removing, and retrieving mathematical functions. Each mathematical function is validated against `numpy` implementations at characteristic points, with special attention to trigonometric, hyperbolic, exponential, and logarithmic functions. The tests employ pytest's approx comparisons for floating-point arithmetic and proper exception handling for error cases. To run the test suite, execute the following in the terminal.

```
cd python_package
pytest tests
```

5.6 Converting to Cython

The original Python package was optimized using Cython, a programming language that extends Python with C-like static typing. To Cythonise the package, we first converted the Python files (.py) to Cython files (.pyx) and added static type declarations using keywords such as `cdef` for variables and class attributes. Static type declarations are essential for performance optimisation because they allow the compiler to generate efficient C code by eliminating Python's dynamic type checking overhead. When variables are statically typed (e.g., declaring `cdef double x` instead of just x), Cython can directly use C-level data types and operations, avoiding the need for type checking and object boxing/unboxing. This is particularly important for mathematical operations where we frequently access and modify numeric values - instead of treating these values as full Python objects that need type verification at each operation, the compiler can generate direct C arithmetic operations. For example, by declaring dual number components as `cdef double real` and numeric variables in loops as `cdef double val`, we enable the compiler to use native C floating-point arithmetic, which is significantly faster than Python's object-oriented number handling.
The package structure was maintained with separate modules for dual numbers and mathematical tools, but with added Cython-specific syntax and C-level operations. We also created a build configuration using `setup.py` and `pyproject.toml` to compile the Cython code into efficient C extensions. This optimisation process allows the package to execute mathematical operations more quickly while maintaining the same functionality and interface as the original Python version.

6 Python vs. Cython analysis

For performance evaluation, we tested the Cythonised implementation against the original Python package across multiple operation types and input sizes. The analysis can be found in the Jupyter notebook `dual_autodiff_analysis.ipynb` located in the `python_package/docs/source/demos/` directory.

The speeds of basic arithmetic operations as well as exponentials, sine, hyperbolic cosine and sigmoid were tested for 100, 1000, 5000, 10,000 and 50,000 iterations. The power operation demonstrated the most significant improvement with a 2.04x speedup. Mathematical functions including sine (1.53x), hyperbolic cosine (1.51x), exponential (1.58x), and sigmoid (1.89x) all showed notable performance gains. We also evaluated the computation of partial derivatives using input sizes ranging from 0 to 50,000 evaluations, which revealed a consistent speedup of 1.83x, with the performance difference becoming more pronounced at larger input sizes. These results demonstrate that while the Cythonised implementation provides moderate performance improvements for simpler operations, it offers more substantial benefits for computationally intensive operations and larger-scale calculations.

7 Packaging and distribution

The Python and Cython packages can be installed from the source, as mentioned in the Usage section previously. Ensure that the requirement dependencies are installed from the file by running

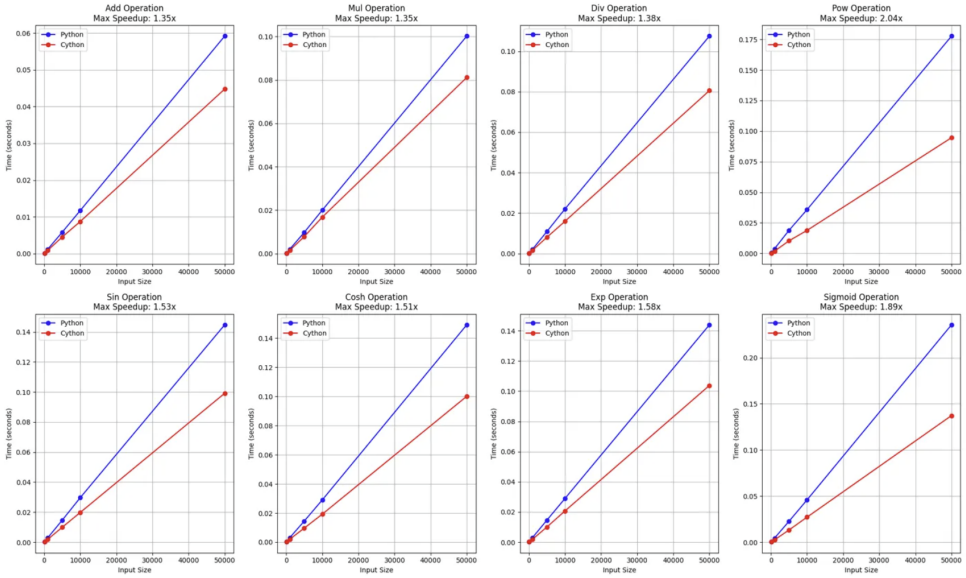

Figure 2: Performance comparison between Python and Cython implementations for various mathematical operations. The graphs show execution time against input size (100, 1,000, 5,000, 10,000 and 50,000) for basic operations (addition, multiplication, division, power) and mathematical functions (sine, hyperbolic cosine, exponential, sigmoid). Each subplot displays the maximum speedup achieved by the Cythonised version.

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

in a virtual environment before using the package. Installing the package will install `numpy` and `matplotlib` as specified in the build requirements in the `.toml` file, but extra dependencies such as `ipykernel` are required to use the Jupyter notebooks.
A Dockerfile is provided, if the user does not want to create and modify virtual environments. To create a docker image, execute

```
cd Docker_testing
docker build -t test-dual-autodiff .
```

To run the Docker image and create a Docker container,

```
docker run -p 8888:8888 test-dual-autodiff
```

This command starts the container and maps port 8888 of the container to port 8888 on the host machine. A URL will be provided in the terminal, which can be copied into a browser to access and interact with the running container. The container installs the packages from the `.whl` files and contains the tutorial Juypter notebook and the Cython analysis Juypter notebook.

7.1 Creating the `.whl` files

The wheel building process for both Cython and pure Python packages involves different approaches. For the Cython package (`dual_autodiff_x`), we used `cibuildwheel` which can handle the Cythonised package. It creates platform-specific wheels containing only the compiled binaries (.pyd files for Windows, .so files for Linux) while excluding the

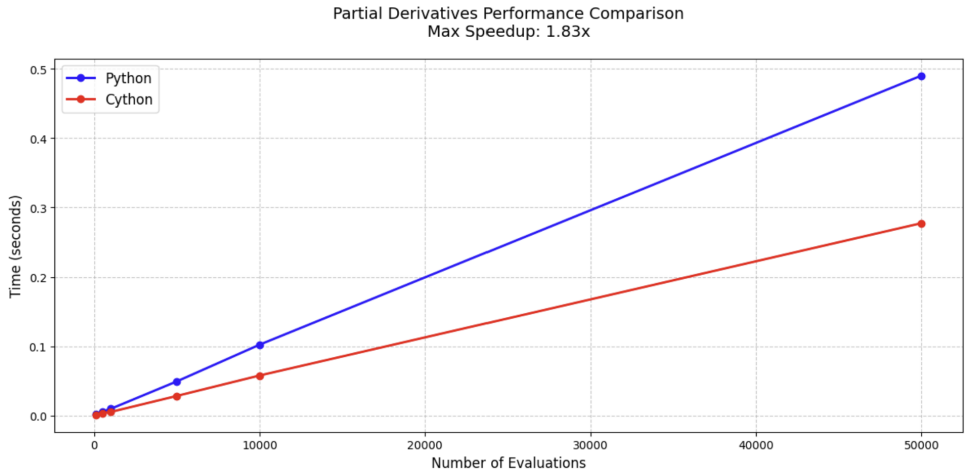

Figure 3: Comparison of partial derivatives computation performance between Python and Cython implementations. The graph shows execution time against number of evaluations (100, 1,000, 5,000, 10,000 and 50,000), demonstrating a maximum speedup of 1.83x for the Cythonised version. The diverging trend lines indicate better scaling behavior of the Cythonised implementation at larger input sizes.

source files (.pyx, .py) through configuration in `setup.py` using `exclude_package_data`. For the pure Python package (dual_autodiff), we used the standard build tool since no compilation is needed. Both packages required configuration through `pyproject.toml` and `setup.py` files to specify dependencies, metadata, and build requirements. The `.whl` files for `dual_autodiff_x` are stored in the `cython_package/wheelhouse/` directory, while pure Python wheels are stored in the the `python_package/dist/` directory. We verified the contents of the Python 3.10 `.whl` to ensure proper packaging - only necessary files were included (compiled binaries and `init.py`), while source code files were excluded to protect the source code. The contents of this `.whl` file is stored in `wheelhouse/wheel_contents/`.

8 Conclusion

This project delivered two implementations of an automatic differentiation package using dual numbers: a pure Python version and a Cython version. The packages provide exact analytical derivatives, avoiding the approximation errors inherent in numerical differentiation methods. The implementation supports both single-variable and partial differentiation, along with a comprehensive set of mathematical operations and functions. Performance analysis demonstrated significant speedups in the Cython implementation. The packages are well-documented, thoroughly tested, and distributed with proper packaging that ensures easy installation across different platforms.

8.1 Large Language Model (LLM) Usage

Claude 3.5 Sonnet and **ChatGPT 4.0** were used in programming and report writing. **Coding:** LLMs were primarily used for plotting, adding docstrings, static typing code, big fixing and writing README.md files. In particular, Claude was used to add static typing to the `.pyx` files. Example prompts:

- '`.py code` Please add static typing to variables so we can Cythonise the package.'
- 'What tests should I implement for my dual class?'

Report Writing: LLMs were used to decrease word count, optimise readability and correct grammatical errors. Example prompts:

- 'How do I add a code block to LATEX?'
- 'Please correct any grammatical or spelling mistakes `INSERT TEXT EXCERPT FROM REPORT`'

References

[1] Tom J. Smeding and Matthijs I. L. Vákár. "Dual-Numbers Reverse AD, Efficiently". In: ().