C1 Research Computing Coursework Report

Hania Rezk (hhmmer2)

MPhil Data Intensive Science, Department of Physics

26 November 2024

Word count: [1000]

Contents

1 Introduction

This report discusses the development of a Python package that performs automatic differentiation using dual numbers. Dual numbers, represented as $x = a + b\epsilon$, consist of a real part a, and a dual part b carried by ϵ that satisfies $\epsilon^2 = 0$. These numbers are used to compute derivatives efficiently; this approach, called forward-mode automatic differentiation, is particularly valuable in fields like machine learning as it enables fast training of deep neural networks.

This report describes the step-by-step creation of this package, its test suite, its documentation and finally the process of cynthonising it. This report also contains the answers to the questions of the coursework sheet.

2 `Dual_autodiff` Package Structure and Project Configuration

This package respects the structure seen in the Research Computing course as shown below.

```
.
+-- pyproject.toml                # Configuration
+-- README.md                     # Instructions
+-- dual_autodiff/                # Package folder with codes
|   +-- __init__.py
|   +-- version.py                # Manage package versions
|   +-- dual.py                   # Dual class
+-- dist/                         # Distribution files
+-- docs/                         # Documentation files
|   +-- Makefile                  # Commands to build the documentation
|   +-- requirements.txt       #List of dependencies of the documentation
|   +-- conf.py                   # Configuration of the documentation
|   +-- index.rst                 # The main page of the documentation
|   +-- material/
|   +-- dual_autodiff          #Tutorial notebook of the package
+-- tests/                        # Test folder
|   +-- autodiff_tools.py         # Test suite for Dual class
+-- Question5.py                  # Code for question 5 of the coursework
```

`Pyproject.toml` is a configuration file used by Python packaging tools,

it contains the build system requirements for a project.
This file contains different sections:

1. **A [project] section**: This section includes basic information about the package, such as its name, description, the name of the README file, project dependencies, and the required Python version. I also included a dynamic versioning setup using `dynamic = ["version"]`, as I use Git to tag versions whenever I make significant updates to the package.

2. **A [Build system] section**: This section specifies the build backend (`build-backend`), and the tools required to build the package (`requires`).

3. **A [tool.setuptools.packages.find] section**: This section tells `setuptools` where to find the Python package within the project repository.

   Additionally, I added a [**tools.setuptools_scm**] section, which allows the automatic determination of the version number based on my Git tags.

   I have also decided to include more information, such as the author's name and email address, and the keywords of our package. This will be helpful if I decide to publish this package on PyPi, as they make my package easier to find.

3 `Dual` Class

The main file of this package is called `dual.py`. It contains the core module of the `Dual_autodiff` package: the `Dual` class. This class defines a structure for dual numbers and performs standard operations on them. An instance of this class has two attributes: one to represent the real part of our number and another to represent the dual part of our number.

Automatic differentiation exploits the fact that every computer calculation executes a sequence of elementary arithmetic operations and functions; which highlights the importance of the next part.

3.1 Operators

The `Dual` class redefines the standard operators to make them compatible with dual numbers. The operators that are redefined are: `+`, `-`, `*`, `**`, `/`, `//`, `%`, `==`, `!=`. The redefinitions consider a wide range of cases, illustrated in the

Details section below. The short-hand versions of these operators (`+=`, `-=`, etc.) are also redefined.

3.1.1 Details

- **The `__add__` and `__radd__` Methods**
  These methods redefine the `+` operator as follows:

  1. `y.__add__(x)` is called for the following operations:
     (a) If both x and y are dual numbers, $y = a' + b'\epsilon$ and $x = a + b\epsilon$:
     $$y + x = (a' + a) + (b' + b)\epsilon$$
     (b) If y is a dual number and x is a scalar:
     $$y + x = (a' + x) + b'\epsilon$$
  2. If x is a scalar and y is a dual number, `y.__radd__(x)` is called for the following operation:
     $$x + y = (a' + x) + b'\epsilon$$

- **The `__sub__` and `__rsub__` Methods**
  These methods redefine the `-` operator as follows:

  1. `y.__sub__(x)` is called for the following operations:
     (a) If both x and y are dual numbers, $y = a' + b'\epsilon$ and $x = a + b\epsilon$:
     $$y - x = (a' - a) + (b' - b)\epsilon$$
     (b) If y is a dual number and x is a scalar:
     $$y - x = (a' - x) + b'\epsilon$$
  2. If x is a scalar and y is a dual number, `y.__rsub__(x)` is called for the following operation:
     $$x - y = (x - a') - b'\epsilon$$

- **The `__mul__` and `__rmul__` Methods**
  These methods redefine the `*` operator as follows:

  1. `y.__mul__(x)` is called for the following operations:

(a) If both x and y are dual numbers, $y = a' + b'\epsilon$ and $x = a + b\epsilon$:

$$y * x = (a' * a) + (b' * a + b * a')\epsilon$$

(b) If y is a dual number and x is a scalar:

$$y * x = (a' * x) + (b'\epsilon * x)$$

2. If y is a dual number and x is a scalar, `y.__rmul__(x)` is called for the following operation:

$$x * y = (a' * x) + (b'\epsilon * x)$$

- **The `__truediv__` and `__rtruediv__` Methods**
These methods redefine the y/x operation as follows:

1. `y.__truediv__(x)` is called for the following operations:

(a) If both x and y are dual numbers, $y = a' + b'\epsilon$ and $x = a + b\epsilon$:

$$y/x = \frac{a'}{a} + \frac{a * b' - b * a'}{a^2}\epsilon$$

Explanation of this formula:

$$\frac{a + b\epsilon \cdot (a' - b'\epsilon)}{(a' + b'\epsilon)(a' - b'\epsilon)} = \frac{a \cdot a' + b \cdot a\epsilon - a \cdot b'\epsilon - bb' \cdot \epsilon^2}{a'^2 - b'^2\epsilon^2} = \frac{a}{a'} - \frac{b' * a - b * a'}{a'^2}\epsilon.$$

and returns a `ZeroDivisionError` if the real part of x is zero.

(b) If y is a dual number and x is a scalar:

$$y/x = \frac{a'}{x} + \frac{b'}{x}\epsilon$$

and returns a `ZeroDivisionError` if x is zero.

2. If x is a scalar and y is a dual number, `y.__rtruediv__(x)` is called for the operation:

$$x/y = \frac{x}{a'} - \frac{b' * x}{a'^2}\epsilon$$

Explanation of this formula:

$$\frac{x}{a' + b'\epsilon} = \frac{x \cdot (a' - b'\epsilon)}{(a' + b'\epsilon)(a' - b'\epsilon)} = \frac{x \cdot a' - b' \cdot x\epsilon}{a'^2 - b'^2\epsilon^2} = \frac{x}{a'} - \frac{b'x}{a'^2}\epsilon.$$

and returns a `ZeroDivisionError` if y is zero.

Note: The `__floordiv__` (`//`) and `__mod__` (`%`) operators are defined using the same logic as above, by replacing `/` with `//` for the floor division and `%` for the modulus operation.

- **The `__pow__` Method**
  This method redefines the `**` operator as follows:

  1. If both x and y are dual numbers, $y = a' + b'\epsilon$ and $x = a + b\epsilon$, `y.__pow__(x)` is defined as:
     (a) If both x and y are dual numbers, $y = a' + b'\epsilon$ and $x = a + b\epsilon$, `y.__pow__(x)` is defined as:
        i. If $b = 0$:
           A. If $a' = 0$ and $a = -1$: raises a `ZeroDivisionError`.
           B. If $a = 0$: returns 1.
           C. Otherwise:
           $$y^x = y^a = a'^a + \epsilon \cdot a \cdot a'^{(a-1)} \cdot b'.$$
        ii. If $b' \neq 0$: raises a `TypeError`.
     (b) If y is a dual number and x is a scalar:
        i. If $a' = 0$ and $x = -1$: raises a `ZeroDivisionError`.
        ii. If $x = 0$: returns 1.
        iii. Otherwise:
        $$y^x = a'^x + x \cdot a'^{(x-1)} \cdot \epsilon.$$

  2. **The `__eq__` and `__ne__` Methods**
     These methods redefine the `==` and `!=` operators as follows:
     (a) If both x and y are dual numbers, $y = a' + b'\epsilon$ and $x = a + b\epsilon$, the equality is evaluated as:
     $$\text{if } (a == a') \text{ and } (b == b'),$$
     it returns `True` for `==` and `False` for `!=`.
     (b) If y is a dual number and x is a scalar:

if $(b' == 0)$:	if $(x == a')$,	returns `True` for `==` and `False` for `!=`.
	if $(x \neq a')$,	returns `False` for `==` and `True` for `!=`.
if $(b' \neq 0)$:		returns `False` for `==` and `True` for `!=`.

  3. **The `__gt__`, `__ge__`, `__le__`, and `__lt__` Methods**
     These methods redefine the `>`, `>=`, `<`, and `<=` operators respectively. The behavior for the operation $x < y$ (and analogously for the other operators) is defined as follows:

   (a) If both x and y are dual numbers with a null dual parts:
   The operator returns the comparison between their real parts:

   $$\text{if } x = a \text{ and } y = a', \quad \text{returns the result of } (a < a').$$

   (b) If y is a scalar and x is a dual number with a null dual part:
   The operator returns the comparison between the real part of x and y:

   $$\text{if } x = a \text{ and } y = y, \quad \text{returns the result of } (a < y).$$

   (c) If x and y are dual numbers with non-null dual parts, or if y is a scalar and x is a dual number with a non-null dual part:
   The operator signals a warning indicating that the comparison is not defined for dual numbers. However, the operator ultimately returns the comparison between the real parts (as a personal choice)

4. **The `__neg__` Method**
   This method redefines the `-` operator by negating the real and dual parts of x, changing the real part to $-a$ and the dual part to $-b$.
5. **Shorthand operators**
   The method `__iadd__` redefines the shorthand operator `+=` using the same logic as `__add__`. Similarly, the methods `__isub__`, `__imul__`, `__itruediv__`, `__ipow__`, `__ifloordiv__` and `__imod__` redefine the shorthand operators `-=`, `*=`, `/=`, `**=`, `//=` and `%=`respectively, using the same logic as their corresponding binary methods (`__sub__`, `__mul__`, `__truediv__`, `__pow__`).

3.2 Functions

The `Dual` class also defines essential functions to make them compatible with dual numbers. The essential functions defined are: exponential, sine, cosine, tangent, logarithm, ceil, floor, inverse, square, and abs. Like the case of our operators, the definition of the functions also considers illegal calls for the functions: inverse, logarithm and tangent who raise `ZeroDivisionError` to prevent a division by zero.

3.2.1 Details

The definition of essential functions lies completely on the f(a + bϵ) formula that was provided for us in the coursework sheet, which provides us with the formulas for the real part and the dual part of each of the functions:

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

The real part corresponds to $f(a)$, and the dual part corresponds to $f'(a)b$. The only work we have to do is compute the derivative of each essential functions we defined.

– Derivative of $\exp(x)$ is $\exp(x)$.
– Derivative of $\log(x)$ is $1/x$, raising a `ZeroDivisionError` when the real part of $x = 0$.
– Derivative of $\tan(x)$ is $1/\cos^2(x)$, raising a `ZeroDivisionError` when $\cos(x.real) \approx 0$ (detected using `np.isclose()`).
– Derivative of $\cos(x)$ is -$\sin(x)$.
– Derivative of $\sin(x)$ is $\cos(x)$.
– Derivative of $square(x)$ is $2x$.
– Derivative of $inverse(x)$ is -$1/x^2$, raising a `ZeroDivisionError` when the real part of $x = 0$.

For `abs(x)`, `floor(x)`, and `ceil(x)`, we just apply these to the real part and the dual part of the dual number x.

4 Test Suite

To ensure the reliability of our `Dual` class, a comprehensive test suite was developed. The `autodiff_tools.py` file in the `tests` folder executes 51 tests on our class, with each test corresponding to a specific function in the dual.py file.

We test a variety of dual numbers with both integer and float values for their real and dual parts. For functions and operations that raise errors (`ZeroDivisionError` and `TypeError`), invalid inputs were also tested to ensure the correct errors were raised.

For the operators, there is a test for every scenario:

1. Operators between two dual numbers.
2. Operators between a dual number and a scalar.

3. Operators between a scalar and a dual number.

These tests cover all possible combinations for each operator to ensure that the class behaves correctly each case.

To run the tests with pytest, after having installed the package, type the following command in your terminal from the project repository:

```
pytest -s tests/*
```

Note: If pytest is not installed in your environment, install it by running the following command:

```
pip install pytest
```

5 Differentiating a function

In this section, we use dual numbers to compute the derivative of the function $f(x) = \log(\sin(x)) + x^2\cos(x)$ (question 5 of the coursework). Three functions are defined in the `question5.py` file:

1. A function to compute the numerical derivative of $f(x)$ with a step size h,

$$f\_numerically(x) = \frac{f(x+h) - f(x)}{h}.$$

2. A function that evaluates the analytical derivative of the function at a point x (which I found manually):

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

3. A function that finds the derivative of f(x) using dual numbers,

$$f\_AD(x) = (x.\text{cos}()*x.\text{square}() + (x.\text{sin}()).\text{log}()).$$

We simply initialize a dual number $x_1 = \text{Dual}(x, 1)$, since $b = 1$, the derivative with respect to x is the dual part of $f\_AD(x1)$, as shown by the formula provided in the coursework:

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

5.1 Observations

We observe that the analytical derivative and the derivative computed through dual numbers are exactly the same, with the same values up to the 16th decimal point, which is the maximum precision for numpy (since the Dual class uses `float64` by default for both the real and dual parts of the dual numbers).

I have also defined a time decorator to see how long each function takes to execute. Both the analytical derivative and the dual numbers derivative are computed in a time order of 1e-5 seconds. The analytical one seems to compute slightly faster, which is expected as the analytical approach involves pre-computed derivatives, while dual numbers compute the derivative from scratch, using function evaluation. This illustrates the usefulness of dual numbers, as they save a lot of effort and eventually time when differentiating.

Figure 1 below illustrates that as the step size gets smaller, the numerical derivative approaches the true analytical value (the plot was obtained using the `numerical_values()` function in `question5.py`). Additionally, we observe that beyond a certain threshold, when the step size becomes too small, the numerical derivative values deviate from the analytical value.

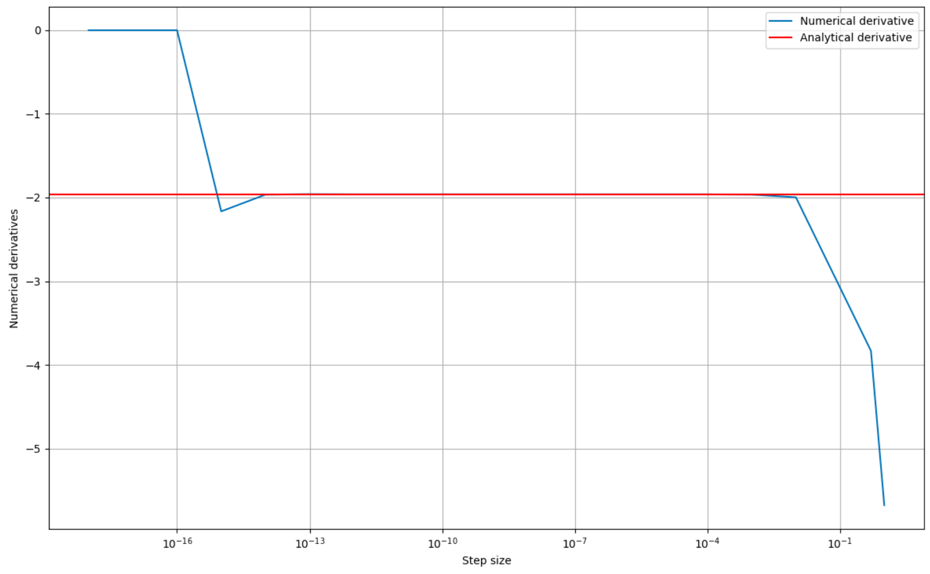

Figure 1: Analytical and Numerical Derivatives for Different Step Sizes

More precisely, the `interval()` function examines what happens in the interval of steps $[1e-16, 1e-13]$. Figure 2 shows that for smaller step sizes than $1e-13$, the numerical derivatives start to oscillate heavily, which is caused by the limitations of floating-point precision and round-off errors that become more pronounced as the step size gets infinitely small.

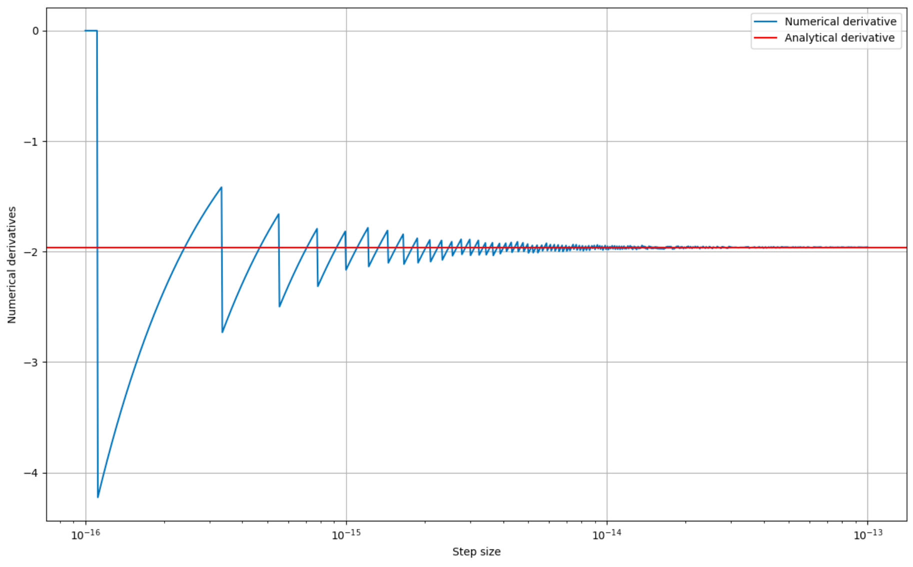

Figure 2: Zoom on the Analytical and Numerical Derivatives for Different Step Sizes

6 Documentation

Since the first file created in my package, the code was documented using docstrings. Each function includes a docstring that describes its purpose, parameters, returns values and potential errors raised, which was very helpful when generating the Sphinx documentation. The structure mirrors the `Read the Docs` page of the C1 course, as I used a theme provided by Read the Docs (`sphinx_rtd_theme`).

To generate the html pages of the documentation, after having installed the package, please run the following command in your terminal from docs folder:

```
make install
make clean
make html
```

The documentation can be viewed by opening docs/build/html/index.html in a web browser.

The `make html` command installs all the required dependencies for building the documentation from the `requirements.txt` file.

7 Cythonized version of the package

After Cythonizing the package, we compared the performance of the Cythonized version (`dual_autodiff_x`) and the normal version (`dual_autodiff`) in the `dual_autodiff.ipynb` notebook.

First, I initialized the same dual numbers using both `dual_autodiff_x` and `dual_autodiff`. Then, I tested the same functions on both versions and verified key equalities to ensure they produced identical results.

To compare the performance of operators and functions, I used the `%timeit` magic command. The results (shown in the figures) demonstrate that the Cythonized version performs almost twice as fast for our operators and other functions. Additionally, I defined:

– A function `f()` that loops 1,000 times, applying various functions.

– A loop that performs 10,000 multiplications.

To see how the performance of the two packages changes with the number of iterations, I used Python's `timeit` module and iterated over different loop counts to measure execution times for both versions. The results are displayed in Figures 3 and 4 (reproducible from the `dual_autodiff` notebook).

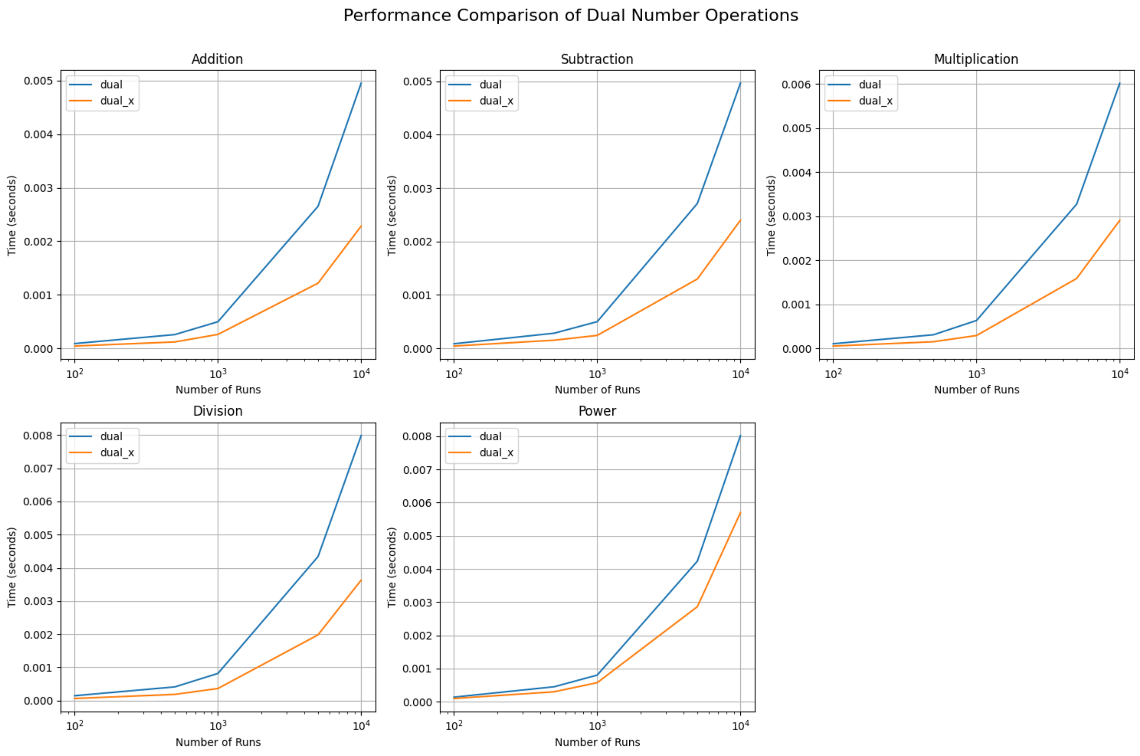

Figure 3: Comparison of the Performance of Different Operators Applied on `Dual_autodiff` and `Dual_autodiff`$_x$

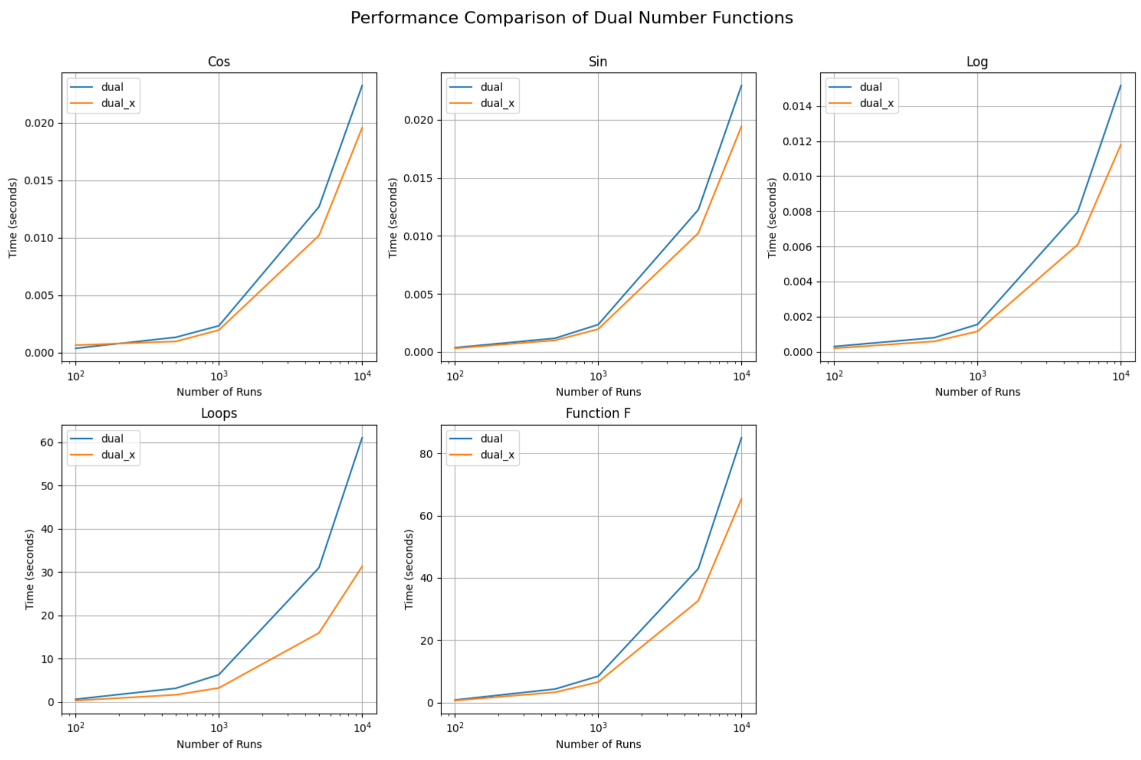

Figure 4: Comparison of the Performance of Different Functions Applied on `Dual_autodiff` and $\texttt{Dual\_autodiff}_x$

We observe that, as the number of iterations increases, the performance gap widens dramatically for the operators. This increase is less pronounced for functions but still evident.

After having completed the C1 course this semester, we can explain the reasons behind these differences in performance:

1. **For the functions (e.g., `sin()`, `cos()`, etc.):** The NumPy module is used in `dual.py`. In the pure Python version, Python performs type checking and memory management before calling the NumPy functions. In contrast, the Cythonized version compiles these calls directly to C at runtime, reducing overhead and improving execution time.
2. **For the operators:** The Cythonized version executes these operations as pure C operations, avoiding Python's overhead. This makes the Cythonized operators approximately twice as fast.

This performance improvement can be particularly useful in machine learning, where automatic differentiation can be performed significantly faster using the Cythonized version.

8 Summary

To summarise, this report described the steps in the creation of the `dual_autodiff` package that can successfully perform automatic differentiation using dual numbers, which was possible by the redefinitions of basic arithmetic operators and functions (section 3). The package is documented and reliable, as it passes a wide range of tests. Finally, after cynthonising the package, we noticed a significant difference in the performance of the functions and operators; which can be particularly useful in machine learning to enable fast training of deep neural networks.

9 Declaration of AI Tools

ChatGPT was the only AI tool used in the creation of this project. It was only used to correct spelling mistakes in this report, and correct my .tex file (source of this LATEX report).

10 Important Notes:

- Please make sure you are using Python 3.9 or a newer version when importing and using the `dual_autodiff` package.
- If you are creating a virtual environment to download the package, please ensure that Python 3.9 or newer is used to create the virtual environment. Also, make sure to upgrade `pip` by running the following command inside your environment:

  ```
  python3.9 -m pip install --upgrade pip
  ```

  (Older versions of `pip` cannot install your package in editable mode without a `setup.py` file.)

- If you are in the directory that directly contains the main project repository (where the `dual_autodiff` package is located), use the following command in Python to import the package:

  ```
  from dual_autodiff.dual import dual
  ```

  in Python to import the package instead of `import dual_autodiff as df`. This is because the system treats `dual_autodiff` as a folder rather than a package in this directory.

11 References

1. `https://en.wikipedia.org/wiki/Automatic_differentiation`
2. `https://github.com/borisbolliet/ResearchComputing/tree/main/docs`