1 Introduction

This report discusses development of the package designed to implement automatic differentiation using dual numbers in Python.

Dual numbers are an extension of real numbers, typically used for automatic differentiation in areas such as research computing and machine learning.

A dual number is defined like so:

$$\text{Dual} = a + b\varepsilon \tag{1}$$

Where a represents the real part, and $b\varepsilon$ represents the dual part. a and b are real numbers, and ε represents an infinitesimally small number such that $\varepsilon^2 = 0$ but $\varepsilon \neq 0$.

Their utility lies in the fact that a dual number automatically encodes the differentiation of the real part in the dual part. This makes them especially useful in machine learning, where derivatives are frequently calculated but can be expensive to obtain.

2 Repository

2.1 Structure

Good practice suggests the following directory structure:

```
dual_autodiff
├── report
│   └── ...
├── dual_autodiff
│   ├── __init__.py
│   └── dual.py
├── tests
│   └── test_dual.py
├── docs
│   └── ...
├── build
│   └── ...
├── source
│   └── ....2 pyproject.toml
├── setup.py
├── README.md
├── Makefile
└── LICENSE
```

2.2 Project configuration

In order to compile this package, a `pyproject.toml` file was used. The following is its contents:

Listing 1: pyproject.toml

```
[build-system]
requires = ["setuptools", "wheel", "setuptools_scm"]
build-backend = "setuptools.build_meta"

[project]
name = "dual_autodiff"
version = "0.0.1"
description = "A Python package for automatic
   differentiation using dual numbers."
readme = "README.md"
requires-python = ">=3.10"
license = { file = "LICENSE" }
authors = [
    { name = "Laura Just Fung", email = "lj441@cam.ac.uk"
       }
]

dependencies = [
    "numpy",
    "mpmath"
]

[tool.setuptools_scm]
write_to = "dual_autodiff/version.py"
version_scheme = "post-release"
local_scheme = "no-local-version"

[tool.setuptools.packages.find]
where = ["."]

[project.urls]
Homepage = "https://github.com/ljf441/dual_autodiff"
Issues = "https://github.com/ljf441/dual_autodiff/issues"
```

3 Implementation

In order to create a package that implements dual numbers, a class that defines what a dual number is must first be created. Afterwards, the usual binary operations (addition, subtraction, multiplication, division, exponentiation, etc.) in Python are overloaded with the

dual implementations. After that, additional functions can be overloaded, including `NumPy` and `mpmath` trigonometric functions, so that dual numbers can be handled as well as real numbers.

3.1 Dual class

The `Dual` class is defined and initialised like so:

Listing 2: Dual class

```
class Dual:
    def __init__(self, real, dual):
        self.real = real
        self.dual = dual
```

The `real` and `dual` variables are defined as attributes of the `Dual` class that can be accessed publicly.

A dual number can be initialised as follows:

Cell 02

```
# initialise dual number
x = df.Dual(2, 1)
print(f"x.real = {x.real}, x.dual = {x.dual}")
```

```
x.real = 2, x.dual = 1
```

And it produces the following output when printed:

Cell 03

```
# printing dual number
print(x)
```

```
Dual(real=2, dual=1)
```

3.2 Binary operations

Binary operations include: addition, subtraction, multiplication, division, and exponentiation. The `Dual` class allows for these operations to performed over dual numbers as well as real numbers when overloaded. An example is as follows:

Cell 04

```
# basic binary operations

x = df.Dual(2, 1)
y = df.Dual(3, 2)

print(f"x + y = {x + y}")
print(f"x - y = {x - y}")
print(f"x * y = {x * y}")
print(f"x / y = {x / y}")
print(f"x**y = {x**y}")
```

```
x + y = Dual(real=5, dual=3)
x - y = Dual(real=-1, dual=-1)
x * y = Dual(real=6, dual=7)
x / y = Dual(real=0.6666666666666666, dual=-0.25)
x**y = Dual(real=8, dual=35.090354888959126)
```

3.3 Unary operations

The unary operations implemented in the Dual class include trigonometric functions (sine, cosine, tangent), inverse trigonometric functions, hyperbolic functions, some inverse hyperbolic functions, the natural logarithm function, and the exponential function. An sample of the functions implemented is shown below:

Cell 05

```
# trigonometric operations as well as the natural logarithm and exponential

print(f"sin(x) = {x.sin()}")
print(f"cos(x) = {x.cos()}")
print(f"tan(x) = {x.tan()}")
print(f"log(x) = {x.log()}")
print(f"exp(x) = {x.exp()}")
```

```
sin(x) = Dual(real=0.9092974268256817, dual=-0.4161468365471424)
cos(x) = Dual(real=-0.4161468365471424, dual=-0.9092974268256817)
tan(x) = Dual(real=-2.185039863261519, dual=5.774399204041917)
log(x) = Dual(real=0.6931471805599453, dual=0.5)
exp(x) = Dual(real=7.38905609893065, dual=7.38905609893065)
```

3.4 Collections

4 Local package

The Python package is installable with `pip install -e .` from the root project folder `dual_autodiff`. The package is importable with `import dual_autodf as df`.

5 Differentiation

Consider the function

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

where $x = 1.5$. The derivative can be easily calculated using `dual_autodiff`:

Cell 06

```
# automatic differentiation

#initialise x = 1.5, with x.dual = 1 to allow for differentiation
x = df.Dual(1.5, 1)

function = x.sin().log() + x**2 * x.cos() #f(x) = log(sin(x)) + x^2 * cos(x)

print(f"log(sin(x)) + x^2 cos(x) = {function.real}")
print(f"d/dx(log(sin(x)) + x^2 cos(x)) = {function.dual}")
```

```
log(sin(x)) + x^2 cos(x) = 0.15665054756073515
d/dx(log(sin(x)) + x^2 cos(x)) = -1.9612372705533612
```

Comparing this to the analytic result

$$\frac{df}{dx} = \frac{\cos x}{\sin x} + 2x\cos x - x^2 \sin x \tag{3}$$

and various numerical methods including the central

$$\frac{d}{dx} f(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \tag{4}$$

forward

$$\frac{d}{dx} f(x) \approx \frac{f(x+h) - f(x)}{2h}, \tag{5}$$

backward

$$\frac{d}{dx} f(x) \approx \frac{f(x) - f(x-h)}{2h}, \tag{6}$$

and five-point equations

$$\frac{d}{dx} f(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}, \tag{7}$$

where accuracy mostly depends on the step size h. For the following comparisons in Table 1, a step-size of $h = 1 \times 10^{-5}$ was used.

The Richardson extrapolation was also used to improve the accuracy of the numerical methods. It works by taking two central approximations of the differentiation with two different step sizes h and $\frac{h}{2}$ and using these to eliminate the $O(h^2)$ error term:

Method	Result (12 d.p.)	Percentage difference (%) (12 d.p.)
`Dual`	-1.961237270553	0.0
Analytic	-1.961237270553	0.0
Central	-1.961237270641	4.466×10^{-9}
Forward	-1.961272309059	$1.786551078000 \times 10^{-3}$
Backward	-1.961202232223	$1.786542145000 \times 10^{-3}$
Five-point	-1.96123727058	1.353×10^{-9}
Richardson central	-1.961237270561	4.1×10^{-10}
Richardson forward	-1.961248950033	$5.955158990000 \times 10^{-4}$
Richardson backward	-1.961225591090	$5.955150808000 \times 10^{-4}$
Richardson five-point	-1.961237270558	2.21×10^{-10}

Table 1: Results of each method and percentage difference of each from the `Dual` result.

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

$$D\left(\frac{h}{2}\right) = \frac{f(x+\frac{h}{2}) - f(x-\frac{h}{2})}{h}, \tag{9}$$

$$\frac{d}{dx}f(x) \approx \frac{4D(\frac{h}{2}) - D(h)}{3}. \tag{10}$$

Comparing the numerical methods to the `Dual` result, it can be seen that with increasing step size h, the percentage difference gets larger, as shown in Fig. 1.

6 Test suite

Test suites for each of the overloaded operations and functions were made in `test_dual.py`.

Tests were made for both left and right-sided operations, ensuring that the `Dual` class can operate on and be operated on by both real and Dual numbers. Tests were also made for the unary functions to ensure that the correct differentiation of each function was made.

Additional tests were also made for the Cythonised `Dual` class, to show that there is no decrease in functional performance.

Tests can be run using `python -m pytest tests` from within the root directory.

7 Project documentation with `Sphinx`

Project documentation was done using docstrings in the `dual.py` file and compiled with `Sphinx`. The documentation can be found in `docs`.

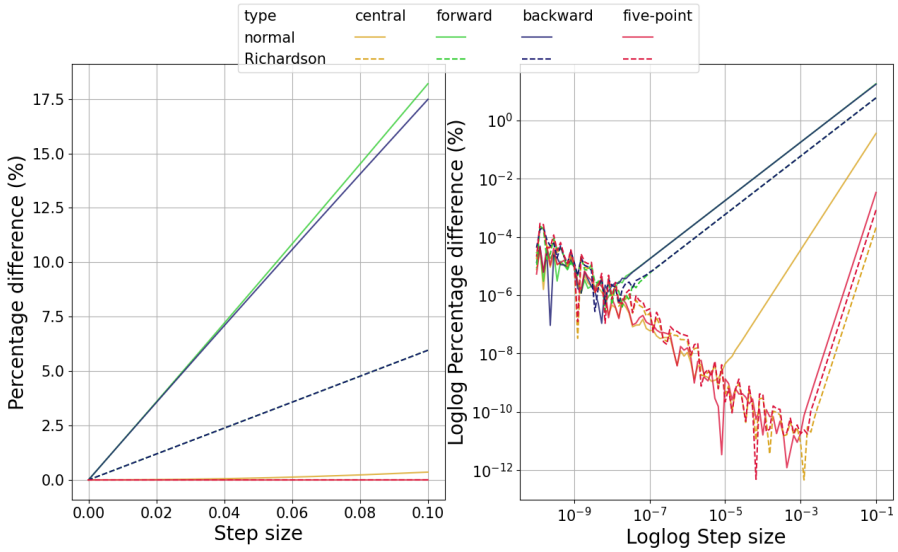

Figure 1: Percentage difference between numerical and `Dual` results for the differentiation of Eq. 2 at $x = 1.5$ as a function of step-size h.

8 Cythonization

Cythonization is when a Python file is compiled into optimised C/C++. This allows for fast program execution and the ability to directly call C libraries. It uses `Cython` to do this, which requires either a `.py` or `.pyx` file to Cythonize.

Within the file, it is helpful to add `cdef` before any definitions of classes or functions and to also publicly declare any variables that need to be accessed outside of the class through `cython.declare(cython.float, visibility="public")`.

Then, using a `setup.py` file, `Cython` can be called to cythonize the program. An example of this can be seen below:

Listing 3: setup.py

```
from setuptools import setup, find_packages, Extension
from Cython.Build import cythonize
from Cython.Distutils import build_ext
import numpy as np

ext_modules = [
    Extension(name="src.dual_autodiff_x.dual", sources=["
        src/dual_autodiff_x/dual.pyx"],
         include_dirs=[np.get_include()],
    )
]
```

```
setup(
    name="dual_autodiff_x",
    version="0.0.1",
    description="A Python package for automatic
       differentiation using dual numbers.",
    python_requires=">=3.10",
    packages=["dual_autodiff_x"],
    package_dir={"":"."},
    ext_modules=cythonize(ext_modules, compiler_directives
       ={'language_level':"3"}),
    zip_safe=False,
    package_data={"dual_autodiff_x": ["*.so"]},
    exclude_package_data={"dual_autodiff_x": ["*.pyx"]},
)
```

9 Comparison between Cythonized and pure Python programs

In terms of time taken, `dual_autodiff_x` and `dual_autodiff` tend to take about the same amount of time to execute operations. The main difference is that `dual_autodiff_x` tends to be a lot more stable, whilst `dual_autodiff` exhibits intermittent spikes of lag.

The operations in which `dual_autodiff_x` shows the greatest performance boost are INSERT TEXT HERE.

For both `pow` and `exp` operations tend to take a lot of time, which is to be expected.

10 Wheels

cibuildwheel is a package that uses Docker to create wheels specifically for different operating systems and architectures. Wheels are a way to distribute Python packages.

For this `dual_autodiff_x`, two wheels were created: one for Python 3.10 in a Linux x86 64 environment, and the other for Python 3.11 in the same environment.

These wheels can be found in `dual_autodiff_x/wheelhouse`.