Research Computing Coursework Assignment

Automattic differentiation with dual numbers

Laura Just Fung (lj441)

December 18, 2024

Word count: 1647

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 $\varepsilon \neq 0$.

One application of dual numbers is their use in automatic differentiation. Any real function $f(x)$ can be extended to be a function of a dual number $f(a+b\varepsilon)$ such that

$$f(a+b\varepsilon) = f(a) + bf'(a)\varepsilon, \tag{2}$$

as all terms involving ε^2 are trivially zero. When these functions are computed over dual numbers, the dual part of the answer automatically encodes the derivative of the function. This makes them especially useful in machine learning, where derivatives are frequently calculated but can be expensive to obtain.

2 Repository

2.1 Structure

The source code for this project was organised into the following folders:

- `dual_autodiff` contains the Python implementation of the `Dual` class in `dual.py`.
- `dual_autodiff_x/src` contains the Cythonized implementation of the `Dual` class in `dual.pyc`.
- `tests` contains the various `.py` test modules that can be used by `pytest` [1].
- `wheelhouse` contains the pure Python wheels.
- `dual_autodiff_x/wheelhouse` contains the Cythonized wheels.

The following is a representation of the overall directory structure:

```
dual_autodiff
├── report
│   ├── report.tex
│   ├── report.pdf
│   └── ...
├── dual_autodiff
│   ├── __init__.py
│   └── dual.py
├── tests
│   ├── test_dual.py
│   ├── test_dual_x.py
│   └── ...
├── docs
│   ├── index.rst
│   ├── conf.py
│   ├── ...
│   └── _build
│       ├── index.html
│       └── ...
├── dual_autodiff_x
│   ├── wheelhouse
│   │   └── ...
│   ├── src
│   │   └── dual.pyc
│   ├── __init__.py
│   ├── setup.py
│   └── pyproject.toml
├── wheelhouse
│   └── ...
├── __init__.py
├── pyproject.toml
├── setup.py
├── README.md
└── LICENSE
```

2.2 Project configuration

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

Listing 1: Contents of the `pyproject.toml` file.

```
[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 = ["."]
include = ["dual_autodiff"]  # Explicitly include the
   dual_autodiff package
exclude = ["tests*", "docs*", "_*", "build*", "*.egg-info
   ", "report*", "__pycache__"]


[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 initialisation in `dual.py`

```
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
# additionally, some inverse trigonometric, hyperbolic,
# and inverse hyperbolic functions

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()}")
print("")
print(f"cosh(x) = {x.cosh()}")
print(f"arccosh(x) = {x.arccosh()}")
print(f"sec(x) = {x.sec()}")
print(f"sech(x) = {x.sech()}")
print("")
z = df.Dual(0.5, 0.5)
print(f"z = {z}")
print(f"arcsech(z) = {z.arcsech()}")
```

```
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)

cosh(x) = Dual(real=3.7621956910836314, dual=-3.626860407847019)
arccosh(x) = Dual(real=1.3169578969248166, dual=0.5773502691896258)
sec(x) = Dual(real=-2.402997961722381, dual=-5.25064633769958)
sech(x) = Dual(real=0.2658022288340797, dual=-0.2562406794416764)

z = Dual(real=0.5, dual=0.5)
arcsech(z) = Dual(real=1.3169578969248166, dual=-1.1547005383792517)
```

3.4 Partial differentiation

An additional method was added to allow for native partial differentiation with the `Dual` class.

This was done by passing the function and its variables to the method, which acts on the variable to be partially differentiated by. All passed in variables are assumed to be dual numbers and those that are not the partially differentiating variable have their dual parts set to zero. This is so they have no effect on the derivative.

Then, the variables are passed through the given function and the derivative is returned.

An example of the method is shown below:

Cell 16

```
#partial derivatives with Dual numbers

#define variables
x = df.Dual(3, 1)
y = df.Dual(4, 2)
z = df.Dual(5, 3)

#define function
def f(x, y, z):
    return x.sin() + y*y + z.cos()

#for each variable, call the 'partial' method
#and pass in the function and all other variables
print("Partial derivative of x:", x.partial(f, x, y, z))
print("Partial derivative of y:", y.partial(f, x, y, z))
print("Partial derivative of z:", z.partial(f, x, y, z))
```

```
Partial derivative of x: -0.9899924966004454
Partial derivative of y: 8.0
Partial derivative of z: 0.9589242746631385
```

3.5 Collections

The package was extended to allow for lists of `Dual` objects to also be operated on. Every operation and method implemented in the `Dual` class is also implemented in the `Collections` class.

An example of this can be seen below:

Cell 17

```
#collections of Dual numbers

#create a 'Collection' of dual numbers. Only lists can be taken.
vars = df.Collection([y, z])

#methods from the Dual class can then be used on Collections.
arccsch_vars = vars.arccsch()
print("Inverse hyperbolic cosecant:")
for i in range(len(arccsch_vars)):
    print(f"acsch({vars[i]}) = {arccsch_vars[i]}")
print("")
#or methods from NumPy
sin_vars = np.sin(vars)
for i in range(len(sin_vars)):
    print(f"sin({vars[i]}) = {sin_vars[i]}")
```

```
Inverse hyperbolic cosecant:
acsch(Dual(real=4, dual=2)) = Dual(real=0.24746646154726346, dual=-0.12126781251816648)
acsch(Dual(real=5, dual=3)) = Dual(real=0.19869011034924142, dual=-0.11766968108291041)

sin(Dual(real=4, dual=2)) = Dual(real=-0.7568024953079282, dual=-1.3072872417272239)
sin(Dual(real=5, dual=3)) = Dual(real=-0.9589242746631385, dual=0.8509865563896788)
```

Cell 18

```
#can perform operations with Collections
print(f"vars + vars = {vars+vars}")
print(f"vars - vars = {vars-vars}")
print(f"vars * vars = {vars*vars}")
print(f"vars / vars = {vars/vars}")
```

```
vars + vars = Collection([Dual(real=8, dual=4), Dual(real=10, dual=6)])
vars - vars = Collection([Dual(real=0, dual=0), Dual(real=0, dual=0)])
vars * vars = Collection([Dual(real=16, dual=16), Dual(real=25, dual=30)])
vars / vars = Collection([Dual(real=1.0, dual=0.0), Dual(real=1.0, dual=0.0)])
```

Cell 19

```
#can perform operations with Collections and Dual numbers
print(f"vars + {x} = {vars+x}")
print(f"vars - {x} = {vars-x}")
print(f"vars * {x} = {vars*x}")
print(f"vars / {x} = {vars/x}")
```

```
vars + Dual(real=3, dual=1) = Collection([Dual(real=7, dual=3), Dual(real=8, dual=4)])
vars - Dual(real=3, dual=1) = Collection([Dual(real=1, dual=1), Dual(real=2, dual=2)])
vars * Dual(real=3, dual=1) = Collection([Dual(real=12, dual=10), Dual(real=15, dual=14)])
vars / Dual(real=3, dual=1) = Collection([Dual(real=1.3333333333333333, dual=2.0), Dual(real=1.6666666666666667, dual=4.0)])
```

4 Local package

The Python package is installable with `pip install -e .` from the root project folder `dual_autodiff`. The package is importable by running in Python:

```
import dual_autodiff as df
```

5 Differentiation

Consider the function

$$f(x) = \log \sin x + x^2 \cos x \tag{3}$$

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

$$\frac{df}{dx} = \frac{\cos x}{\sin x} + 2x\cos x - x^2 \sin x \tag{4}$$

and various numerical methods including the central [2]

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

forward [2]

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

backward [2]

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

and five-point [3] equations

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

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.

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 [4] 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:

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

$$D\left(\frac{h}{2}\right) = \frac{f(x+\frac{h}{2}) - f(x-\frac{h}{2})}{h}, \tag{10}$$

$$\frac{d}{dx}f(x) \approx \frac{4D(\frac{h}{2}) - D(h)}{3}. \tag{11}$$

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.

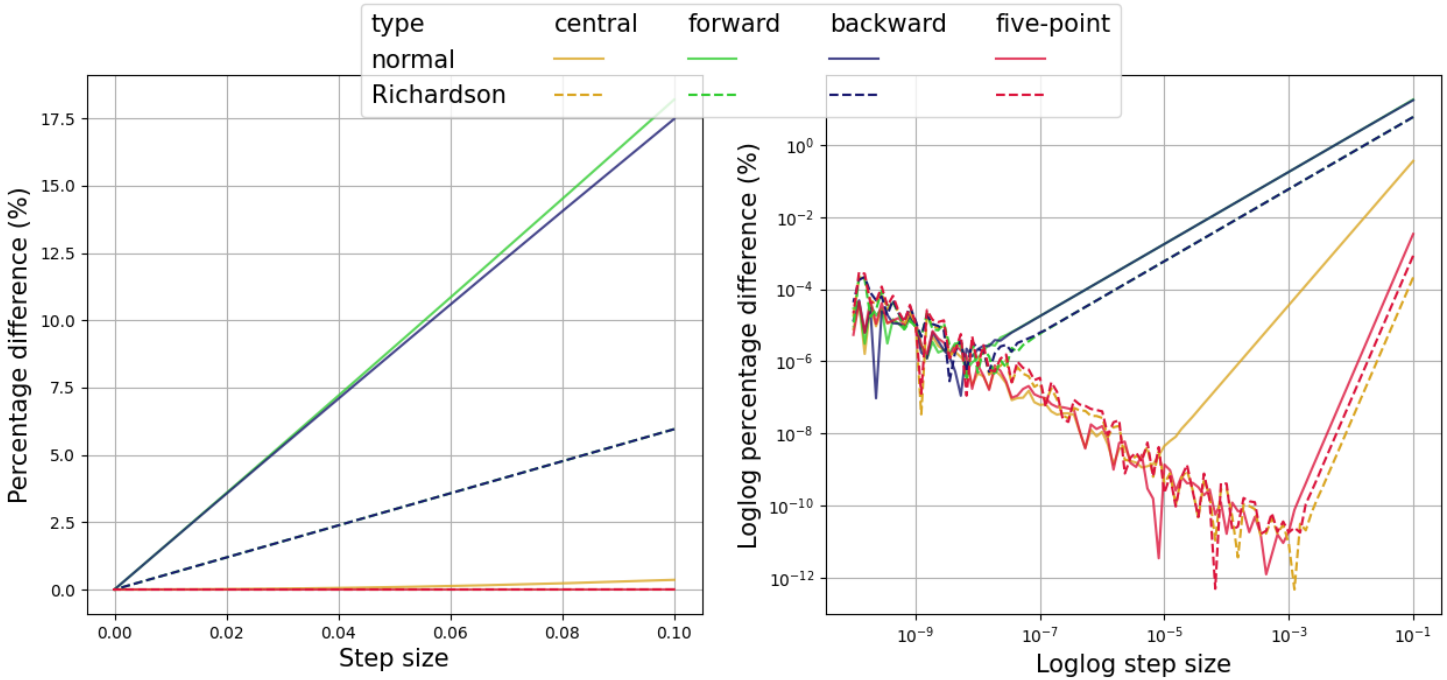

Figure 1: Percentage difference between numerical and `Dual` results for the differentiation of Eq. 3 at $x = 1.5$ as a function of step-size h.

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. Tests were also made for the `Collection` class, ensuring that left and right-sided operations would work on this class as well as the same unary functions implemented for the `Dual` class. Additional tests were also made for the Cythonized `Dual` and `Collection` classes.

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

CELL 21

```
#running pytest

!pytest tests
```

```
============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/ljf1/dis/dual_autodiff
configfile: pyproject.toml
plugins: anyio-4.7.0
collected 220 items

tests/test_collection.py ..........................................  [ 21%]
.........                                                           [ 25%]
tests/test_collection_x.py ........................................ [ 45%]
...........                                                         [ 50%]
tests/test_dual.py ................................................ [ 75%]
.                                                                   [ 75%]
tests/test_dual_x.py .............................................. [ 98%]
...                                                                 [100%]

============================== 220 passed in 0.73s ==============================
```

7 Project documentation with `Sphinx`

The project documentation for the `dual_autodiff` package was done using docstrings in the `dual.py` file and compiled with `Sphinx` [5]. The `index.rst` and `conf.py` files can be found in `docs` whilst the compiled `index.html` file can be found in `build`.

To use `Sphinx`, `cd` to the `docs` folder and run:

```
make html
```

Please note that this requires `Sphinx` which can be pip installed using the `requirements.txt` file:

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

8 Cythonization

Cythonization [6] 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: The contents of the `setup.py` file.

```
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

The effects of Cythonization can be seen in Fig. 2 and Fig. 3. In terms of memory usage, the Cythonized `dual_autodiff_x` is much more efficient, while the pure Python `dual_autodiff` takes up a lot more space.

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 the basic arithmetic operations: addition, subtraction, multiplication, and division. While speedup occurs for all functions timed, as seen in Fig. 4, the trigonometric functions show less of an improvement, as do the exponentiation, logarithmic, and exponential functions.

10 Wheels

`cibuildwheel` [7] 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 the `dual_autodiff_x` package, 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.

For this project, `tox` was used to build the wheels in the correct `Python` environment. The `MANIFEST.in` file was used to control which files were included and excluded within the wheels. Compiled Python files, `.pyc` files, were specifically excluded.

To install the wheels, download them from `dual_autodiff_x/wheelhouse`. Then run

```
pip install dual_autodiff_x/wheelhouse/dual_autodiff_x
    -0.0.1-cp310-cp310-manylinux_2_17_x86_64.
    manylinux2014_x86_64.whl
```

or

```
pip install dual_autodiff_x/wheelhouse/dual_autodiff_x
    -0.0.1-cp311-cp311-manylinux_2_17_x86_64.
    manylinux2014_x86_64.whl
```

depending on whether Python 3.10 or Python 3.11 is installed.

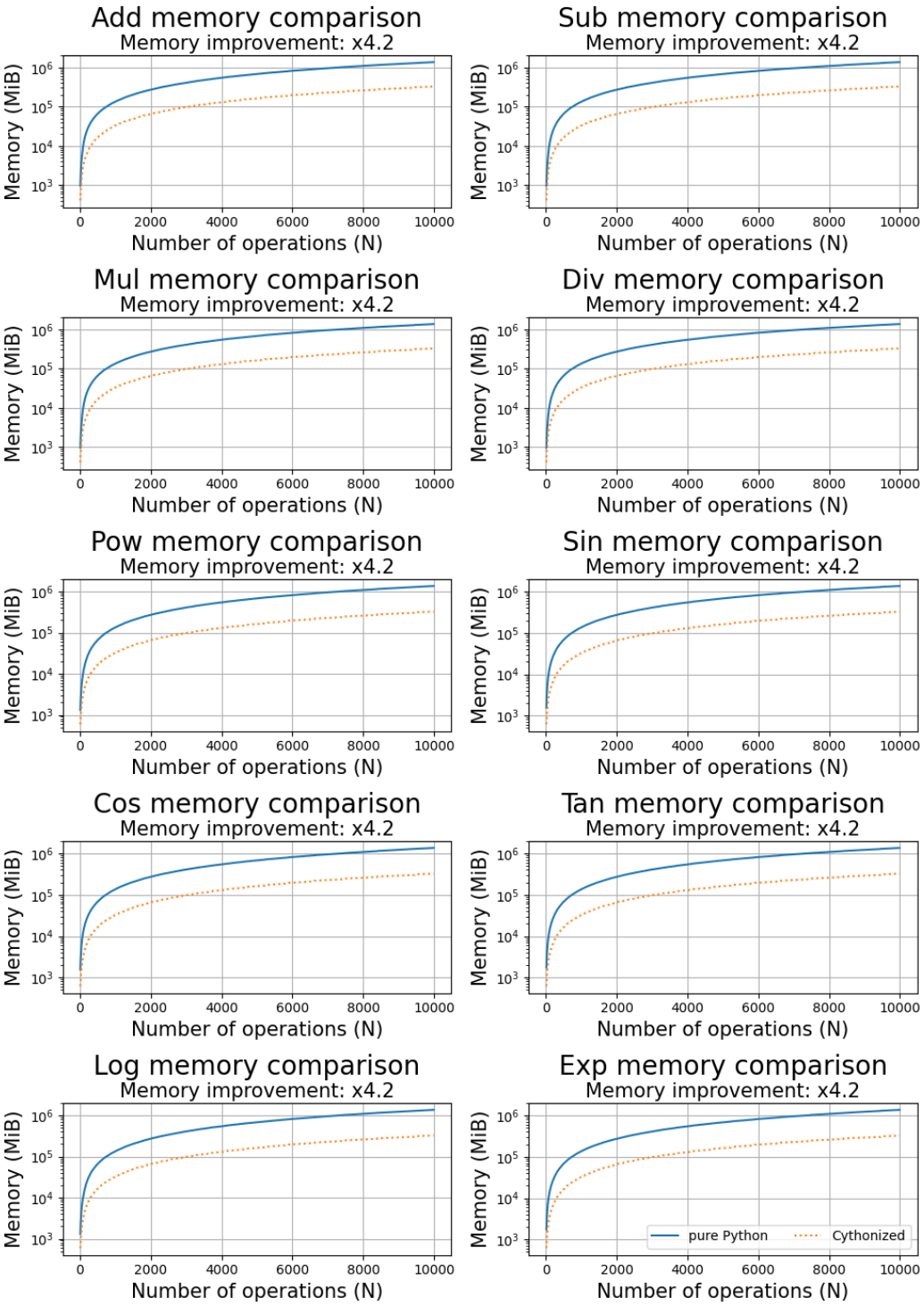

Figure 2: Performance of the Cythonized and pure Python versions of the `dual_autodiff` functions in terms of memory usage.

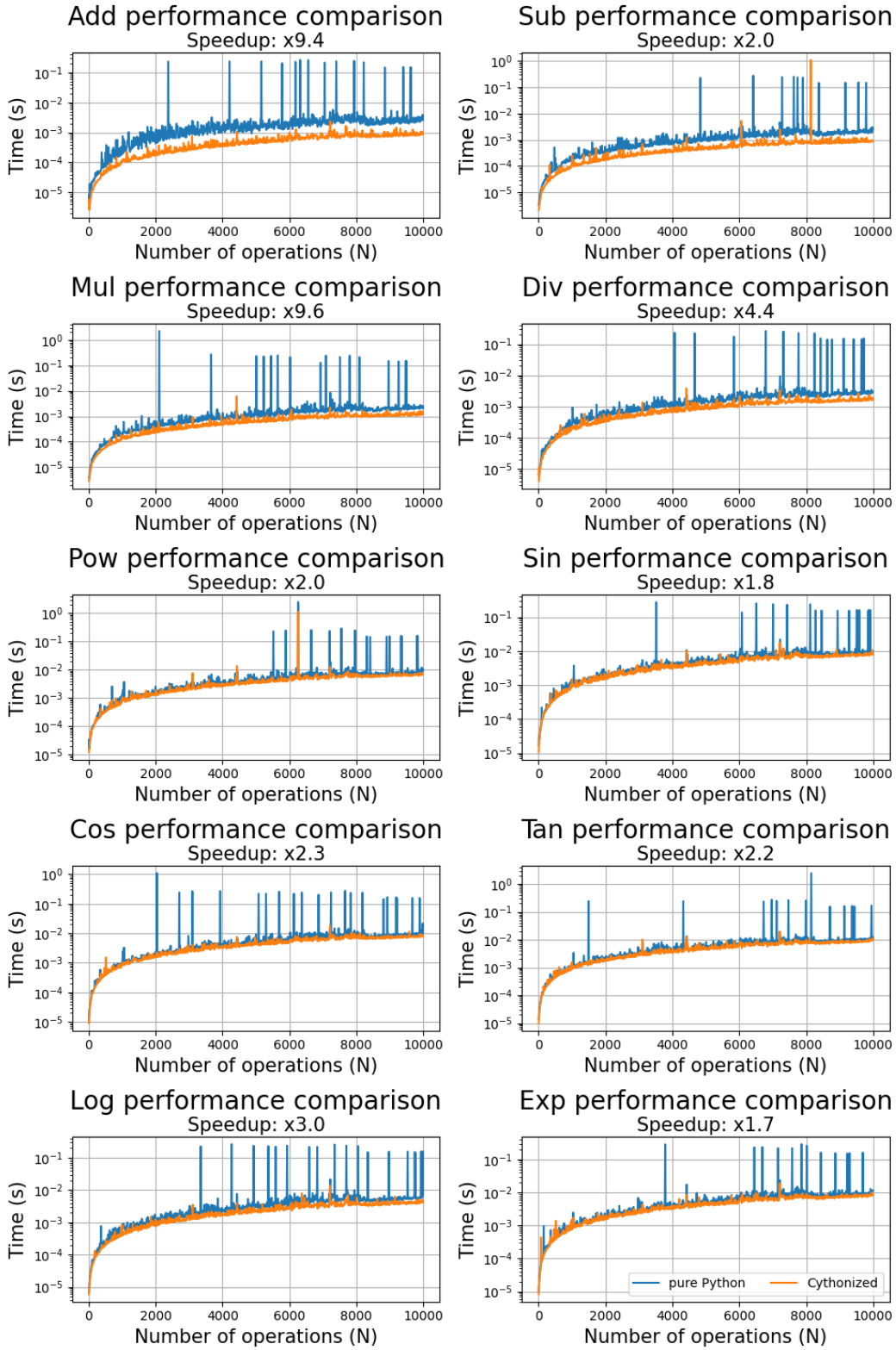

Figure 3: Performance of the Cythonized and pure Python versions of the `dual_autodiff` functions in terms of time taken.

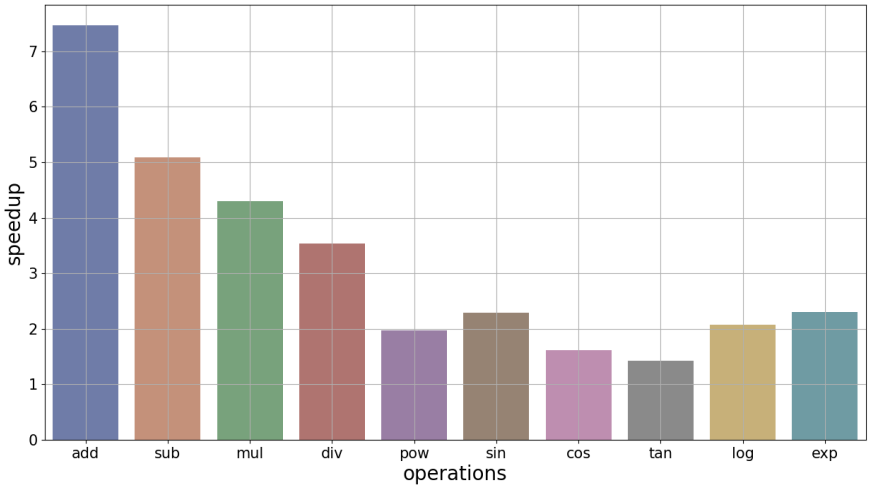

Figure 4: Averaged speedup of Cythonized version of `dual_autodiff` compared to the time taken by the pure Python version.

11 Docker

A Dockerfile was created such that the tutorial notebook `dual_autodiff.ipynb` can be run using both `dual_autodiff` and `dual_autodiff_x` packages that are installed from wheels.

This can be compiled by running:

```
docker build -t dual_autodiff-img .
```

And then to use the Docker image:

```
docker run -p 8888:8888 dual_autodiff-img
```

12 Summary

A package that implements a dual number class for automatic differentiation was implemented in Python 3. This package was extended such that partial differentiation is natively possible, collections of `Dual` objects can be acted on by class methods, and additional trigonometric functions were added. A testing suite was added to be used by `pytest`. The package was then Cythonized to take advantage of C/C++ libraries and to allow for fast program execution. The package was also built then into wheels specifically for Python 3.10 and 3.11 in a Linux environment and uploaded to Gitlab.

References

[1] Krekel H, Oliveira B, Pfannschmidt R, Bruynooghe F, Laugher B, Bruhin F. pytest x.y; 2004. Available from: `https://github.com/pytest-dev/pytest`.

[2] Kaw A. 2.03: Numerical Differentiation of Functions at Discrete Data Points;. Available from: `https://math.libretexts.org/Workbench/Numerical_Methods_with_Applications_(Kaw)/2%3A_Differentiation/2.03%3A_Numerical_Differentiation_of_Functions_at_Discrete_Data_Points`.

[3] Sauer T. Numerical analysis. 2nd ed. Boston: Pearson; 2012. OCLC: ocn725295545.

[4] Richardson LF. The Approximate Arithmetical Solution by Finite Differences of Physical Problems Involving Differential Equations, with an Application to the Stresses in a Masonry Dam. Philosophical Transactions of the Royal Society of London Series A. 1911 Jan;210:307-57.

[5] Brandl G. Sphinx documentation. URL http://sphinx-doc org/sphinx pdf. 2021.

[6] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K. Cython: The best of both worlds. Computing in Science & Engineering. 2011;13(2):31-9.

[7] Rickerby J, Jadoul Y, Darbois M, Schreiner H, Grzegorz B. cibuildwheel; 2017. Available from: `https://github.com/pypa/cibuildwheel`.

A Use of auto-generation tools

Auto-generation tools were used to help parse error messages throughout the project and to help format this LaTeX report.

Auto-generation tools were also used for code prototyping with the `__array_ufunc__` and `__array_function__` methods and code alteration when Cythonizing the package.

Auto-generation tools were not used elsewhere, for code generation, writing, or otherwise.