C1: Research Computing Coursework Assignment

Fayzan Mahmood

December 2024

Wordcount: 2964

Introduction

This report details my implementation of a Python package that performs automatic differentiation using dual numbers. A dual number is of the form $a + b\epsilon$, where a and b are real numbers and ϵ is such that $\epsilon^2 = 0$ but $\epsilon \neq 0$. The values a and b are referred to as the 'real' and 'dual' parts of a dual number respectively. In this project, I implement a Python class that extends mathematical operations and some common mathematical functions to dual numbers.

The report will outline the various stages of development of the package. In section 1, I will briefly discuss the initial repository structure. In section 2, I will detail the implementation of the `Dual` class, how to install the package and how to utilise it to compute derivatives, including an example comparing to the corresponding analytical and numerical results. In section 3, I will discuss my test suite implementation for the package and in section 4, I will cover the Sphinx documentation. Finally in section 5, I will attempt to Cythonize the package, compare the performance of the Python and Cython versions and build and use Cython wheels.

1 Initial Repository Structure

```
.
├── tests/
├── docs/
├── dual_autodiff/
│   └── __init__.py
├── pyproject.toml
├── .gitignore
└── README.md
```

My project repository was structured following good practices and initially contained the following files and directories:

- `pyproject.toml` file: To configure the package.
- `dual_autodiff/` directory: This is where the source code of the package will be located. It must contain an `__init__.py` file to create a package out of the source code.
- `docs/` directory: Where the documentation will be placed.
- `tests/` directory: Where the test files will be placed to test the package.
- `README.md` file: This provides an overview of the package and instructions on how to install it.
- `.gitignore` file: To ignore unnecessary files so they do not appear in the repository.

1.1 Writing Project Configuration

My pyproject.toml file for this project is shown below:

```
[build-system]
requires = ["setuptools", "wheel", "build"]  # Build requirements
build-backend = "setuptools.build_meta"

[project]
name = "dual_autodiff"
version = "0.1.0"
description = "A Python package for performing automatic differentiation using dual numbers."
readme = "README.md"
requires-python = ">=3.9"
authors = [
    { name = "Fayzan", email = "fm565@cam.ac.uk" },
]
keywords = ["automatic differentiation" , "forward-mode" , "deep neural networks"]
classifiers = [
    "Development Status :: 4 - Beta",
    "Intended Audience :: Developers",
    "Programming Language :: Python :: 3",
    "Topic :: Software Development :: Libraries"
]

# Runtime dependencies
dependencies = [
    "numpy"
]


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

The file contains important information to configure the package. Under `build-system`, there are a list of of build requirements to build the python package and the build-backend.

The `[project]` section details other information about the project, including the name; version; a short description of the project; where to find the readme file; the minimum Python version required for this package; the author(s) of the package; and some keywords and classifiers which may help users find the package if uploaded to PyPI.

One of the core features of the package is the implementation of mathematical functions (e.g. `sin, exp`) on dual numbers, which all depend on NumPy. This has therefore been included in the package dependencies.

The final two lines tell the configuration file to find the package in the same directory level that this file is in.

2 Creating the `dual_autodiff` Package

2.1 Implementing the `Dual` Class

The package directory contains two files, `dual.py` and `autodiff_tools.py`, that define a `Dual` class and some member functions that implement dual numbers, define various operations on them and several mathematical functions.

The class constructor defines a dual number given a real part and dual part. These are assigned as attributes to an instance of the Dual class and can be called from within a Python session, or alternatively using the corresponding member functions which return either the dual part or real part.

Most of the functions in `dual.py` are 'magic methods' (defined by the double underline '__') which overload pre-existing operators to allow them to be used for dual numbers. An important method to implement was the `__repr__` method which defines how dual numbers are displayed in a Python session or notebook. This is to ensure the following output:

```
>>> from dual_autodiff import Dual
>>> x=Dual(2,1)
>>> x
Dual(real=2, dual=1)
```

The other methods extend arithmetic operators, operate-and-assign operators, comparison operators and unary operators (e.g. returning the inverse or negative of a dual number) so that they apply to dual numbers. The details of each method is explained in the package documentation. Considerations were taken to ensure that the arithmetic operators apply to two numbers of type 'Dual' as well as for real numbers (of type 'int' and 'float') to be combined with dual numbers on either the left-hand side or right-hand side. An example of this can be seen in the tutorial notebook.

The `autodiff_tools.py` file imports the `Dual` class from `dual.py` and defines functions that implement $\sin(x), \cos(x), \tan(x), \ln(x)$ and $\exp(x)$ on dual numbers. Writing these functions in a separate file is a matter of preference - I felt it is better to have these functions separate from the magic commands as further functions can be added to this file without looking through the main file. These functions are then imported into `dual.py` and assigned as member functions to the `Dual` class.

As explained in the question paper, for any function f applied to a dual number,

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

which follows from Taylor's theorem where the higher order terms vanish since $\epsilon^2 = 0$. This is used to implement the functions in `autodiff_tools`, for example,

$$\sin(a + b\epsilon) = \sin(a) + b\cos(a)\epsilon$$

so that

```
>>> x = Dual(2, 1)
>>> x.sin()
Dual(real=0.9092974268256817, dual=-0.4161468365471424)
```

where $\sin(2) = 0.9092...$ and $1 * \cos(2) = -0.4161....$

Docstrings have been provided for the `Dual` class and its member functions and structured according to best practices (see section 4). Where it is necessary, exception handling has been implemented to deal with errors e.g. division-by-zero when inverting a dual number (see section 3).

2.2 Installing the Package

I created a virtual environment in which to install the package. To do this, I ran the following commands in the root directory:

```
python -m venv venvs/c1_cwk
source venvs/c1_cwk/Scripts/activate
```

Then I run the following command to install the `dual_autodiff` package:

```
pip install -e .
```

I can then import the package from anywhere by opening a python session and writing

```
>>> import dual_autodiff as df
dual_autodiff package version: 0.1.0
```

and the version number is displayed to confirm the package has been imported successfully.

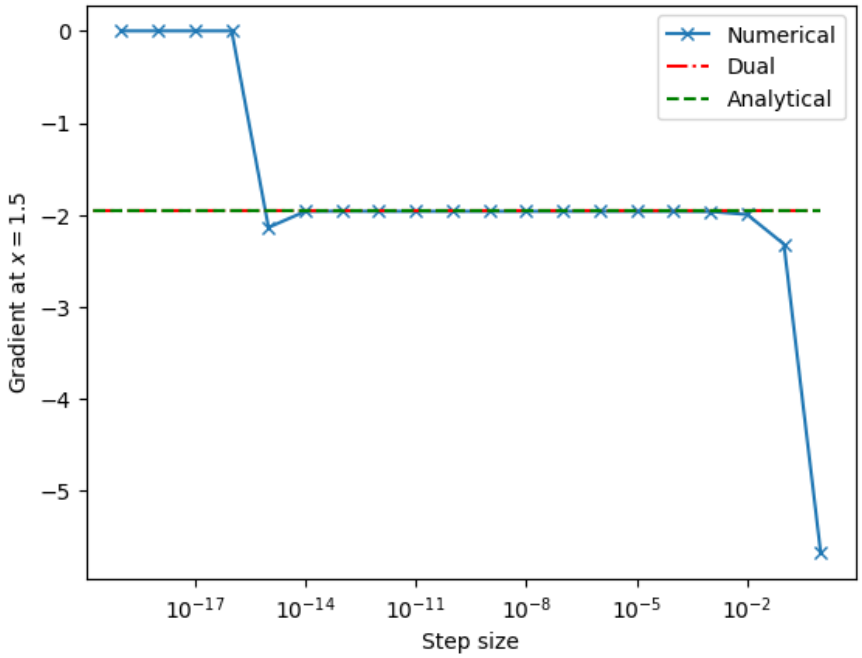

Figure 1: Derivative of $f(x) = \log(\sin(x)) + x^2\cos(x)$ at $x = 1.5$.

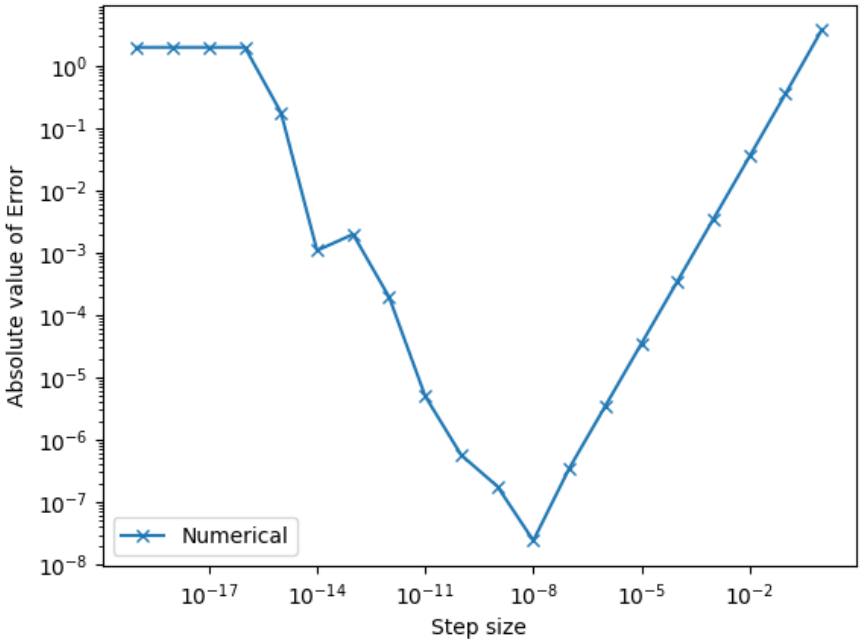

Figure 2: Absolute value of error in the numerical derivative of $f(x) = \log(\sin(x)) + x^2\cos(x)$ at $x = 1.5$.

2.3 Computing Derivatives using the Package

The code for this section, including the code to make the plots, can be found in the tutorial notebook `dual_autodiff.ipynb`, which is located in the `docs/Tutorial` directory in the GitLab repository.

As explained earlier, the derivative of any function at $x = a$ can be computed by evaluating the function at $a+\epsilon$ and taking the dual part. Here we evaluate the gradient for the function $f(x) = \log(\sin(x))+x^2\cos(x)$ at

$x = 1.5$. Computing the derivative using dual numbers and comparing it to the analytical gradient evaluated at the same point, we find that the two results agree up to machine precision (that is, to 16 decimal places). This shows that the dual number method is very accurate, which is highly useful for performing automatic differentiation. This also indicates that the `Dual` class has been implemented correctly.

For further comparison, I compute the numerical derivative of the function using a decreasing step size. The result is plotted in Figure 1. This shows that the numerical approximation is quite poor for a step size larger than around 10^{-3} (because the higher order terms in the Taylor expansion are not yet small enough) or smaller than around 10^{-15} (due to rounding error). From Figure 2, we also see that at the optimal step size, around 10^{-8}, the error is still of magnitude 10^{-7} which is much larger than machine precision.

In summary, the numerical derivative is limited by the step size we choose (which is not the case when using dual numbers) and is far less accurate than either the analytical or dual derivative. By contrast, the dual derivative agrees with the analytical one up to machine precision.

3 Implementing a Test Suite

The test suite was implemented using `pytest`. The test files are located in the `tests` directory in the GitLab repository and contains two test files corresponding to the two modules in the `dual_autodiff` directory. I also had to include a blank `__init__.py` file to remove an error in importing the package when running the tests.

The tests that are implemented *aim* to consider all possible cases in which each method could be used, including invalid uses in particular cases. Tests functions were implemented for every corresponding member function in the `Dual` class. Test functions of a similar nature were grouped into classes, to help readability.

For example, to test my overloading of the addition operator, I checked that addition worked in the following cases:

- Two dual numbers where the real and dual parts are a mixture of positive and negative integers and floats
- Check that adding `Dual(0,0)` does not change the answer
- Adding a dual number and its conjugate (i.e. the conjugate of $a + b\epsilon$ is $a - b\epsilon$) gives the correct result $(2a + 0\epsilon)$.
- Addition of a real number to a dual on the right
- Addition of a real number to a dual on the left
- Check that the sum of two real numbers equals a dual number with zero dual part.

The above checks were carried out using the `assert` statement. Other tests involved checking the correct output was displayed, such as when printing an instance of the `Dual` class, but also in the cases where an error is expected due to an invalid-use case. To do this, I used the pytest functionality `capsys` to capture the output and an assert statement to check the correct output is shown. Further, in other tests where floating-point precision affects the result, the use of `pytest.approx` was useful in comparing to the expected outcome to a relevent precision.

We can then run the test suite with the following command in the root directory:

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

and all of my tests are successful.

4 Documentation

The documentation for this package was created using Sphinx. The docstrings from the `Dual` class were implemented by linking to the core package modules in the `*.rst` files and including the `sphinx.ext.autodoc`

extension in the `conf.py` file, which configures the documentation. The tutorial notebook was also included similarly using the `nbsphinx` extension.

The docstrings for the `Dual` class and its member functions have been formatted as shown in the C1 Research Computing course notes [1] which follows best practices. The purpose of these docstrings are to help the user understand the code I have written. I have given a short description of the main class and its attributes (i.e. dual, real) along with their types. For the methods, this includes a short description of the function, a list of parameters and their types, and the output. I have also included some examples in the docstrings for the functions in `autodiff_tools`.

The tutorial notebook showcases some of the various features implemented in the package including examples of the overloaded operators and mathematical functions. There is also an example which shows how to differentiate the function in section 2.3 using the `Dual` class and the same discussion, including the graphs from this section, is included here. There is also a section comparing the performance of the Python and Cythonized package, as will be seen in section 5.

Instructions on how to build the documentation are given in the `README.md` file in the GitLab repository.

5 Cythonizing the Package

5.1 Creating `dual_autodiff_x`

```
dual_autodiff_x
├── pyproject.toml
├── setup.py
└── src
    └── dual_autodiff_x
        ├── __init__.py
        └── dual.pyx
```

I created a package directory for my Cython package `dual_autodiff_x` as shown above. Initially, I wanted to make separate folders for the pure Python package and Cython package, but I had issues with dynamic versioning (which I later removed) due to the location of the git tags when I moved the Python package into a subdirectory. Instead, I decided to leave my Python package structure as before and add the Cython package directory at the root level.

Within the `dual_autodiff_x` directory, I added a new `pyproject.toml` file indicating the location of the new Cython modules in the `src` folder and including `"Cython"` in the build requirements. The `setup.py` file follows closely the example provided in the C1 notes [1], which sets up the Cythonized modules/extensions correctly so that we can correctly build the package and wheels.

I copied my Python source code into a single `dual.pyx` file, as I found it simpler to run in the next section when adding static typing. I then ran the following command

```
1    python setup.py build_ext --inplace
```

and this created the `.c` and `.pyd` files as expected.

I was then able to install the Cythonized package with `pip install -e .` from inside the `dual_autodiff_x` directory successfully.

5.2 Comparing Performance

After creating the Cython package, I decided to include static typing in my source code to further optimise the code. The purpose of this is to declare the variable type so that it is known at compile time, removing the need for type checking during execution [1]. Changing the `def class Dual` to `cdef class Dual`, or including variable declarations such as `cdef public double real` help are examples of the static typing I

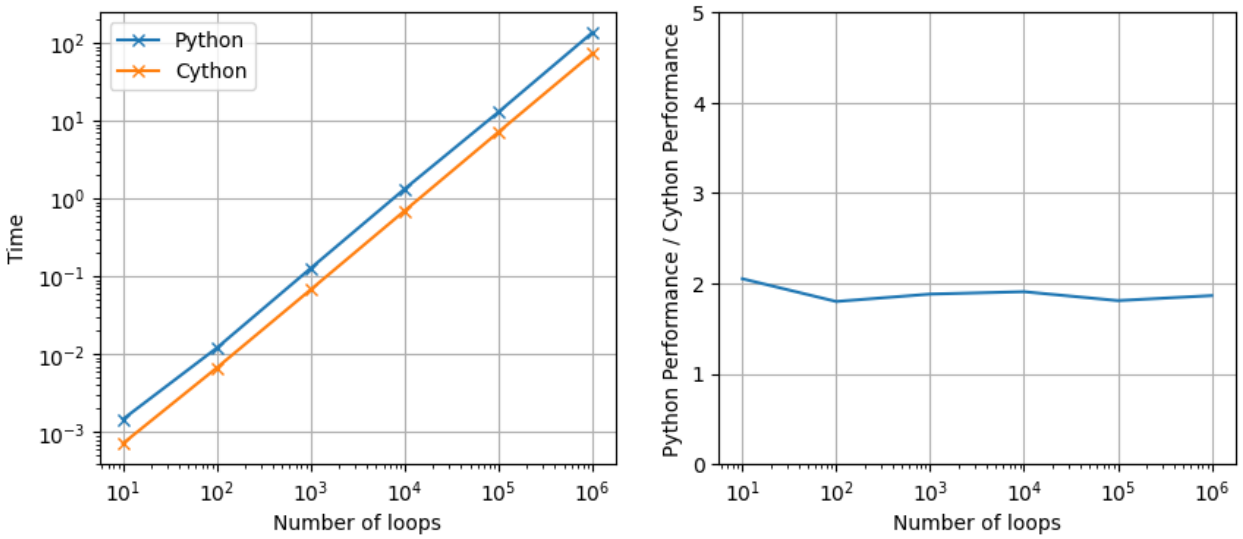

Figure 3: Comparing the performance of the pure Python and Cythonized packages.

included (public is included here as attributes in C are private by default). A consequence of this is that my Cython package can no longer apply the binary operators (e.g. +,-,*,/,+=,...) I defined in the Dual class to a Dual and a non-Dual type (for example, `Dual(2,1) + 2` is no longer permitted but `Dual(2,1) + Dual(2,0)` is). Since this would have required redefining many of the functions in my package, I decided to leave this extra functionality out since it is not strictly required for the package to work.

I compared the performance of both packages in the tutorial notebook by running a set of operations and functions implemented in the packages. I then computed the total time taken to run these tests for different numbers of iterations over 10 calls using the `timeit` library. The results of this performance comparison are shown in Figure 3. For example, for 100000 iterations I computed an execution time of 13.44 seconds for the Python package compared to 7.55 seconds for the Cython package. From the first graph, we observe that the Cythonized package is slightly faster across different numbers of samples, and is almost twice as fast as is deduced from the second graph. This is due to the fact that the Cython code is compiled in C++, which provides an initial speedup, and static typing which provides an additional speedup. Further optimisation may improve the difference in performance further (for example, using Cython's `annotate` feature to identify bottlenecks in the code [2]).

5.3 Linux Wheels

To build wheels for the Cythonized package I use `cibuildwheel` and run Docker on my Windows PC by installing Docker Desktop and opening it. I built Linux wheels for Python 3.11 using the following command given in C1 notes [1]:

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

and similarly for Python 3.10 where we change the Python version to `cp310`. The wheels are located in the `wheelhouse` directory in the root directory of my Cythonized package. Unzipping the wheels into the `wheel_contents` directory, I checked that `*.pyd, *.so` files and no `*.pyx` files were present. However, there are still `*.c` files present and `__init__.py` files present.

5.4 Using the Linux Wheels

I now install the package using the wheels I have just created. After uploading the wheels to GitLab and downloading them again, I attempted to install the package on Windows Subsystem for Linux (WSL) on

my PC. After creating a virtual environment in the Ubuntu terminal, I was able to install the Python 3.10 wheel successfully using `pip`. I was then able to run the package in a Python session and import the package in the tutorial notebook (after installing Jupyter with `pip install jupyter`). Note that, in order to use the tutorial notebook fully, we also need to install `numpy, matplotlib` and the `dual_autodiff` package as most of the examples are designed for the Python package, not the Cython one. In the GitLab repository, I have provided a conda environment which contains all the packages installed in my local virtual environment during this project, which is sufficient to run the notebook. I have also created the `cython_test` notebook to check that the package runs correctly without the need to set up the conda environment or install more packages.

I also installed the Python 3.11 wheel on CSD3 using the `scp` command to transfer the downloaded wheel file. The wheel installed successfully and I was able to use the package in a Python session.

Summary

In this project, I have implemented a package that performs automatic differentiation using dual numbers in both Python and Cython. Throughout this project, good software development practices have been demonstrated. I have created a well-structured package; provided a comprehensive and meaningful test suite; and outlined basic usage of the Python package with clear explanations, docstrings and examples in the documentation. I have also successfully Cythonized the package and created Cython wheels for Linux. Finally, I have evaluated the performance of the Cython package and shown that it provides a slight improvement due to static typing and compilation in C.

Appendix - Declaration of Use of Autogeneration Tools:

Generative tools were not used in the writing of the introduction, main body, or summary of this report. I have used generative tools to generate the references at the end of this document. I have modified these references from the output produced from the generative tool.

A declaration of the use of generative tools in writing the code in this project is given in the `README.md` file in the coursework GitLab repository.

References

[1] Boris Bolliet. Research computing documentation, 2024. `https://researchcomputing.readthedocs.io/en/latest/`.

[2] Neurohackweek. Cython tutorial: Annotations, 2024. `https://neurohackweek.github.io/cython-tutorial/03-annotations/`.