Automatic Differentiation using Dual Numbers Package

Lewis McConkey
Cambridge University
Word count: 1872

February 28, 2025

Abstract

The focus of this report is on implementing forward-mode automatic differentiation using dual numbers. The dual number implementation is encapsulated in a Python package called sub_dual_autodiff, which supports a variety of mathematical operations such as addition, multiplication, and transcendental functions (e.g., sine, cos, log). After the creation of the package and implementing a test suite we write project documentation with Sphinx and then cythonize the package. We determine that cythonizing our package results in better performance than the pure python version before finally finishing on creating and uploading wheels for our cythonized package.

1 Introduction

In this report we go through creating a sub_dual_autodiff package for automatic differentiation using dual numbers. We go through how to create the package, how to implement the dual numbers and how to turn the code into a package. We also compare different differentiation methods to dual numbers to see which has the best performance. We implement a test suite covering all the operations we can use in our package and then write project documentation with Sphinx. We cythonize the package and call our new package sub_dual_autodiff and use this to compare to the pure python version (sub_dual_autodiff). We finish on creating wheels (for our cythonized package) and uploading them to the GitLab project repository.

2 Create Project Repository

We open the terminal and type mkdir dual_autodiff to create our main folder. We then change the directory to this by entering cd dual_autodiff. We now create another folder inside our main folder and call it sub_dual_autodiff (again by entering mkdir sub_dual_autodiff) and we add files to this subfolder by entering touch dual_autodiff/core.py, touch dual_autodiff/__init__.py and touch dual_autodiff/version.py. We add a folder called tests using mkdir sub_auto_diff/tests and add in the file called test__init__.py with the touch command (touch sub_auto_diff/tests/<file_name> to add the file into the right subfolder). Finally we add (using touch again) the pyproject.toml file, the README.md file, the LICENSE file and the .gitignore file outside the subfolders and just in the main folder by just typing touch <file_name>. The full file structure from initially setting it up is shown below:

```
dual_autodiff/
├── sub_dual_autodiff/
│   ├── __init__.py
│   ├── core.py
│   ├── version.py
│   └── tests/
│       └── __init__.py
├── pyproject.toml
├── README.md
├── LICENSE
└── .gitignore
```

3 Project Configuration

Next we write the file contents for the pyproject.toml file. It starts with the [build-system] section that specifies the tools required for building the project which are setuptools, wheel, and setuptools_scm. [The project] section contains the name of the package, the version derived from Git tags (using setuptools_scm) and essential project details such as description, author information, and licensing. This section also contains the dependencies numpy and pytest as required packages and indicates compatibility with Python 3 and compliance with open-source licensing. We add the [project.urls] section to add URLs for the source code and issue tracker to improve accessibility and collaboration. We also add the [tool.setuptools_scm] section which automatically manages the projects versioning and writes it to the given file. Lastly we add [tool.setuptools.packages.find] which instructs setuptools to look for the package in the current directory.

4 Implement Dual Numbers

We create the dual.py file in the sub_dual_autodiff folder and write our class that defines the dual numbers. It takes in two parameters (the real part and the dual part) and allows correct representation and mathematical operations including multiplication, addition etc. It also defines sin, cos, tan, log and exp. We also add from .dual import Dual and __all__ = ["Dual"] to the __init__.py file to import the Dual class for easier usage and to define the public API of the package.

5 Code into Package

Our pyproject.toml is already setup correctly for turning our code into a package from our initial project configuration step. Inside our package folder in the terminal we enter git init, git add ., git commit -m "Initial commit" and git tag 0.0.0beta0 for the versioning to work with setuptools_scm. We then install the package in development mode with pip install -e. We test in a jupyter notebook (R1_code) by writing from sub_dual_autodiff import Dual and then typing $x = \text{Dual}(10, 4)$, $y = \text{Dual}(5, 17)$ and $\text{print}(x + y)$ which gives us 15 21 (the expected result). We could also test out any other part of our package (sin, addition, log) etc. to check it gives us what we expect and so we have seen that it has been successful and we can now use this installed package we have created in python.

6 Comparing Derivatives

For this task we use a Jupyter notebook which is called R1_code (In the dual_autodiff folder). We now differentiate the given function to give our analytical derivative:

$$\frac{cosx}{sinx} - x^2 sinx + 2xcosx$$

Entering $x = 1.5$ into this analytical derivative gives -1.9612372705533614. Comparing this to the derivative we get from using dual numbers and the package we created we get a difference of 2e-15 (See the R1_code file to view the exact number we get from the dual numbers method). Hence we observe that the two methods give almost identical answers which is what we expect but there is also some very slight fluctuation. Calculating the numerical derivative (with step size 1e-5) we get -1.9612723090589588 which is a much bigger fluctuation away from the other two methods but still not so much (0.00004... away). To observe further we look at the numerical derivative error against the step size. We plot this:

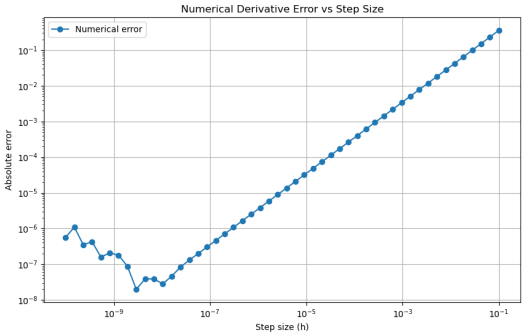

Figure 1: Numerical derivative error vs step size

From this we see that if we increase the step size past 10^{-8} then the absolute error continues to increase through higher step sizes. We quickly see that the numerical derivative method performs poorly for large step sizes and approximates the derivative worse than the dual numbers and analytical derivative especially for higher step sizes. The error from a step size of 1e-5 was shown to be 0.00004... which can be seen on the plot and even for step sizes around 1e-8 we still only get errors around 1e-8 whereas with the dual numbers we got the error of 2e-15 which is still so much lower than the error for the numerical derivative method. Overall we conclude that the dual numbers and analytical derivative methods perform way better than the numerical method especially for large step sizes but even for small step sizes it still approximates the derivative and calculates the value (when given an x value) way better.

7 Test Suite

We first install pytest using pip install pytest. We now add the files tests_dual.py, tests_operations.py and tests_functions.py and write our tests in these files. The tests_dual.py file contains tests for the basics for dual numbers in terms of the representations of the dual numbers (real and dual part and checking these are represented correctly and this part of the Dual class works as required). The tests_operations.py contains the tests for the mathematical operations used in the Dual class (addition, subtraction, multiplication and division) and checks these are correct. The tests_functions.py contains the tests for the essential functions (sin, log, cos, tan and exp) and checks these are correct. We now change the directory to the dual_autodiff folder, write pip install -e to install the package in editable mode and finally run the test suite using pytest -s tests/*. We expect to run and see all 11 tests passed.

8 Sphinx

We start by installing sphinx by entering pip install sphinx sphinx_rtd_theme nbsphinx into the terminal and setting up the docs folder with sphinx-quickstart docs (making sure we are in the dual_autodiff folder). Now we change directory to the docs folder and enter make clean and then make html to build the documentation. We add some changes to the conf.py file like adding extensions for example and add changes to the index.rst file to include the docstrings from our code (see the conf.py and index.rst files for the full implementation of these files). We enter sphinx-apidoc -o source ../sub_dual_autodiff into the terminal to generate the .rst files with the docstrings from our code and add the dual_autodiff.ipynb file in the source folder (we update the changes in the index.rst file). We finally add the instructions to demonstrate how to use the package and also add docstrings to the file dual_autodiff.ipynb (view this file to see what we added)

9 Cythonize the Package

We create a separate directory called dual_autodiff_x and add in a pyproject.toml file aswell as a setup.py file. We then create a new folder in this directory called src and add a new folder in that folder called sub_dual_autodiff_x (This is where our .pyx files will be). We can copy

and paste our .py files from our sub_dual_autodiff folder into the sub_dual_autodiff_x folder we just created. We then change the .py files into .pyx by entering into the terminal mv dual.py dual.pyx for the dual file and doing this with each .py file for the rest of them (making sure we are in the right directory when we do this). Next we modify the pyproject.toml and setup.py files with the relevant code (view these files to see the changes). We finally build the package by entering python setup.py build_ext –inplace into the terminal. We can then see that this works when we enter pip install -e inside the project folder to install the package.

10 Pure Python vs Cython

In the dual_autodiff.ipynb file (view the file to see the code that compares the two packages) we compare the performance of the pure python and the Cythonized version. We create a function to compare the packages performance and choose 1000 trials to loop through for both of them. Here is the bar-plot we get to compare the execution time:

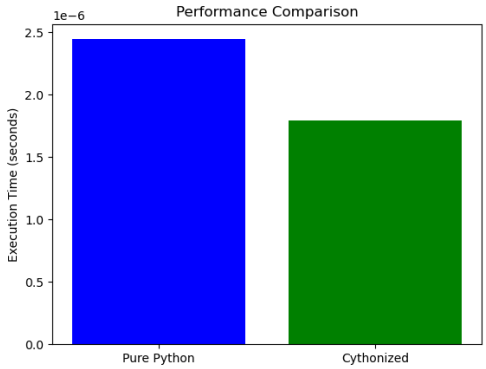

Figure 2: Pure Python vs Cythonized version

This shows we get 0.0000024 second execution time for the pure python version and 0.0000018 for the Cythonized version. This is because Cythonized code should perform better for computationally heavy functions like loops.

11 Wheels for Linux

We first install cibuildwheel using pip install cibuildwheel. We build the first wheel using CIBW_BUILD="cp310-manylinux_x86_64" CIBW_ARCHS="x86_64" cibuildwheel –platform linux and build the second in the same way but change cp310... to cp311... We use the code given in the question to check that one of the wheels doesn't contain the source code which can be seen in the wheel_contents folder.

12 Uploading Wheels

We finish this report by uploading our wheels to the gitlab repository. First we clone the GitLab repository using git clone <url-of-the-repo> and then create a directory for the wheels using mkdir wheels. We then copy the .whl files into this repository using cp wheelhouse/*.whl wheels/. We now stage the files and commit these changes using git add wheels/ and git commit -m "Added wheels for dual_autodiff_x" respectively. Finally we push these changes using git push origin main and verify this has worked using the code given in the question and this is successful.