C1 Report

Ngoc Anh Doan

December 2024

Introduction

In this coursework, the aim is to develop a Python package, `dual_autodiff`, that performs automatic differentiation using dual numbers. The coursework involves constructing a robust software package that can compute derivatives efficiently in forward-mode, a fundamental approach for various applications such as deep learning and optimization. By leveraging the properties of dual numbers, the package will implement essential operations, mathematical functions, and derivative computations. Additionally, this coursework emphasizes best practices in software development, including project packaging, testing, documentation, and optimizing performance through Cython.

Through this coursework, the theoretical underpinnings of dual numbers will be explored, demonstrated their practical implementation, and analyzed performance differences between Python and Cython-based solutions. The final deliverable include a comprehensive repository featuring a working package, tests, documentation, and a comparison of numerical and analytical derivative methods.

Declaration of AI Assistance

In this coursework, I use ChatGPT, a language model developed by OpenAI, as a supportive tool for specific tasks. ChatGPT was employed to:

1. Debug Code: Identify and resolve issues in code implementations.

2. LaTex help: Provide suggestions for improving better display and use of LaTex.

The use of ChatGPT was restricted to enhance understanding, solving programming-related challenges, and improving the clarity of implementation. All decisions regarding the final design, implementation, and analysis were independently verified and made by me. ChatGPT's assistance was supplementary and did not generate or replace original work or intellectual contributions in this coursework.

All the files were run on Macbook Air M1 2020, Python version 3.9.

1 Create project repository structure and configuration (task 1+2)

1.1 Directory Structure

The project directory was carefully structured to ensure simplicity, usability, and clarity. A Python package named `dual_autodiff` was created, adhering to software engineering best practices, including the organization of source code, tests, and documentation.

The directory structure is as follows:

```
dual_autodiff/
├── dual_autodiff/
│   ├── __init__.py
│   ├── dual.py
│   └── tests/
├── docs/
│   ├── conf.py
│   ├── index.rst
│   └── ...
├── pyproject.toml
├── README.md
├── .gitignore
└── LICENSE
```

The rigorous organization of the project guarantee ease of use, but also easier management and extension of the package. A test folder is used, ensure all functionalities are thoroughly verified. The report elaborates `test` and `doc` in a deeper level in the next tasks.

1.2 Key Files and Their Purposes

This section details the files contained in the repository and what their uses are.

- `dual_autodiff/`: Contains the source code to make the package.
  - `dual.py`: Implements the `Dual` class and the operations for dual numbers.
  - `__init__.py`: Allows importing `dual_autodiff` as a package.
  - `tests/`: Contains all unit tests.
- `docs/`: Stores the project's documentation.
- `pyproject.toml`:
  - Replaces `setup.py` for project configuration in this project.
  - Includes metadata (author, version, dependencies) and build instructions.
- `README.md`: Provides an overview of the project, installation instructions, and usage examples.

- `.gitignore`: Ensures unnecessary or sensitive files (e.g., `.pyc`, `__pycache__`) are not included in version control.
- `LICENSE`: Specifies the open-source license for the project.

1.3 Project Configuration with `pyproject.toml`

The `pyproject.toml` file was created adhering to Python's best practices [1]. It standardizes project metadata, dependencies, and build system settings, replacing older tools like `setup.py` for many tasks. It has the following contents:

```
[build-system]
requires = ["setuptools>=61.0", "wheel", "setuptools_scm", "cython"]
build-backend = "setuptools.build_meta"

[project]
name = "dual_autodiff"
version = "0.1.0"
description = "Automatic differentiation using dual numbers."
readme = "README.md"
authors = [
    { name = "Ngoc Anh Doan", email = "nad60@cam.ac.uk" }
]
license = { file = "LICENSE" }
keywords = ["automatic-differentiation", "dual-numbers", "forward-mode"]
classifiers = [
    "Programming Language :: Python :: 3",
    "Operating System :: OS Independent",
    "License :: OSI Approved :: MIT License"
]
dependencies = [
    "numpy"
]
[tool.setuptools.packages.find]
where = ["dual_autodiff/"]
```

The `pyproject.toml` file serves as the central configuration for the project, contains several key sections:

The `[build-system]` section specifies the build system requirements, including tools like `setuptools`, `wheel`, `setuptools_scm` (for versioning), and `cython` (for performance optimization and later package cythonization). The `build-backend` is set to `"setuptools.build_meta"`, instructing tools like `pip` on how to build the package.

The `[project]` section includes essential metadata about the package. This encompasses the `name` (a unique identifier for PyPI), the `version` (initial release version), a short `description` of the package's purpose, and the `readme`, which points to the `README.md` file for long-form project descriptions. Additionally, the `authors` field lists the author's name and contact email, while the `license` field specifies the license file, such as the MIT license. The `keywords` field improves searchability on PyPI, and the `classifiers` categorize the project for better visibility and indexing.

The `[dependencies]` section lists the required dependencies to use the package, such as `numpy` in this case.

Finally, the `[tool.setuptools.packages.find]` section configures `setuptools` to locate the Python package files. It ensures that all relevant files within the `dual_autodiff/` directory are included in the final distribution. This structure ensures clarity, maintainability, and compatibility with modern Python packaging tools.

2 Dual Package (task 3+4)

2.1 Setting Up Dual

Definition of Dual Numbers

A dual number [2] is any number which can be expressed as:

$$x = a + b\epsilon$$

where:

- a: The real part of the dual number.
- b: The dual part, representing derivative information.
- ϵ: An infinitesimal number satisfying $\epsilon^2 = 0$.

This formulation enables dual numbers to represent both function values and their derivatives, making them particularly useful for automatic differentiation.

Implementation

The `Dual` class is initialised with two components: `real` and `dual`. The class is designed to handle both the value of a function and its derivative simultaneously.

```python
class Dual:
    __slots__ = ['real', 'dual']

    def __init__(self, real: float, dual: float = 0.0):
        self.real = real
        self.dual = dual

#Example:
x = Dual(5, 18)
print(x.real, x.dual)

# and then we can get the expected output:
5, 18
```

The `__slots__` attribute optimises memory usage and improves performance by explicitly declaring the class `Dual` can only have two attributes `real` and `dual`. [3]

2.2 Mathematical Operations

2.2.1 Arithmetic Operations

The basic arithmetic operations on dual numbers $x = a+b\epsilon$ and $y = c+d\epsilon$ can be defined as follow:

- **Addition**: $(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$
- **Subtraction**: $(a + b\epsilon) - (c + d\epsilon) = (a - c) + (b - d)\epsilon$
- **Multiplication**: $(a + b\epsilon) \cdot (c + d\epsilon) = (a \cdot c) + (a \cdot d + b \cdot c)\epsilon$
- **Division**:
$$\frac{(a + b\epsilon)}{(c + d\epsilon)} = \frac{a}{c} + \frac{b \cdot c - a \cdot d}{c^2}\epsilon \quad (\text{for } c \neq 0).$$

We can have a look at an example of implementing division:

```python
def __truediv__(self, other):
    if isinstance(other, Dual):
        if other.real == 0:
            raise ZeroDivisionError("Division by zero.")
        new_real = self.real / other.real
        new_dual = (self.dual*other.real - self.real*other.dual)/(other.real**2)
        return Dual(new_real, new_dual)
    if other == 0:
            raise ZeroDivisionError("Division by zero.")
    return Dual(self.real/other, self.dual/other)

# Usage
x = Dual(6, 2)
y = Dual(3, 1)
z = x / y
print(z.real, z.dual)  # Outputs: real part and dual part

# Expected Output:
2.0, 0.0
```

2.2.2 Trigonometric, Exponential and Derivatives Functions

Dual numbers inherently encode derivative information in their dual part. For a function $f(x)$, evaluated at $x = a$, we have:
$$f(a + b\epsilon) = f(a) + f'(a)b\epsilon$$

Thus, if we let b=1, we can easily deduce that the dual part is the derivative $f'(a)$. This makes calculating trigonometric and exponential value of dual numbers much easier as follows:

- **Sine**: $\sin(a + b\epsilon) = \sin(a) + b\cos(a)\epsilon$
- **Cosine**: $\cos(a + b\epsilon) = \cos(a) - b\sin(a)\epsilon$
- **Logarithm**: $\log(a + b\epsilon) = \log(a) + \frac{b}{a}\epsilon$
- **Exponential**: $\exp(a + b\epsilon) = \exp(a) + b\exp(a)\epsilon$

Note that all of those trigonometric functions assume radian measurement. For example, we can look at the function of sine:

```python
def sin(self):
    return Dual(math.sin(self.real), self.dual * math.cos(self.real))

# Usage
x = Dual(2, 1)
result = x.sin()
print(result.real, result.dual)  # Outputs: real part and dual part

#Expected Output:
0.9092974268256817, -0.4161468365471424
```

2.3 Installing and importing the package

As can be seen from 1.1, within `dual_autodiff/` there are two `.py` files. One is `dual.py`, which is the file we created in the previous section. The other one is `__init__.py`. This Python file is crucial for initializing the `dual_autodiff` package and imports the `Dual` class.

```python
from .dual import Dual
```

Following standard practices, after navigating to the project folder, the package was installed in editable mode using:

```
pip install -e .
```

This command needs to be executed in the same working directory as the file `pyproject.toml`. Now, we can import the package with:

```python
import dual_autodiff as df
```

everywhere using the same environment where it is installed. We can also test if the package works by checking some simple checks:

```
 python -c "import dual_autodiff as df; print(df.Dual(5, 18))"
```

to have the same output as the example above 2.1.

3 Differentiating a function (task 5)

The first example function tested is given by:

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

whose derivative is given by:

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

Next, we define a Python function that takes a `Dual` number `X` and returns $f(X)$:

```python
from dual_autodiff.dual import Dual

def f(X):
    sinX = X.sin()           # sin(a) + b*cos(a)*
    log_sinX = sinX.log()    # log(sin(a)) + (cos(a)/sin(a))*b*
    x2 = X * X               # a   + 2ab
    x2cosX = x2 * X.cos()    # x cos(a), derivative by product rule
    return log_sinX + x2cosX
```

To get the derivative at $x = 1.5$, create a `Dual` number with a dual part of 1:

```python
x = Dual(1.5, 1.0)
fx = f(x)
print("f(1.5) =", fx.real)
print("f'(1.5) =", fx.dual)  # This is the derivative at x=1.5
```

And we use `python` to obtain analytical derivative: Compute the analytical derivative separately:

```python
a = 1.5
analytical_derivative = (
    (math.cos(a)/math.sin(a)) + 2*a*math.cos(a) - a*a*math.sin(a))
print("Analytical f'(1.5) =", analytical_derivative)
```

Lastly, we use numerical derivatives. We take 8 steps, `hs = [1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10]` and plot it in the same graph with analytical derivative.

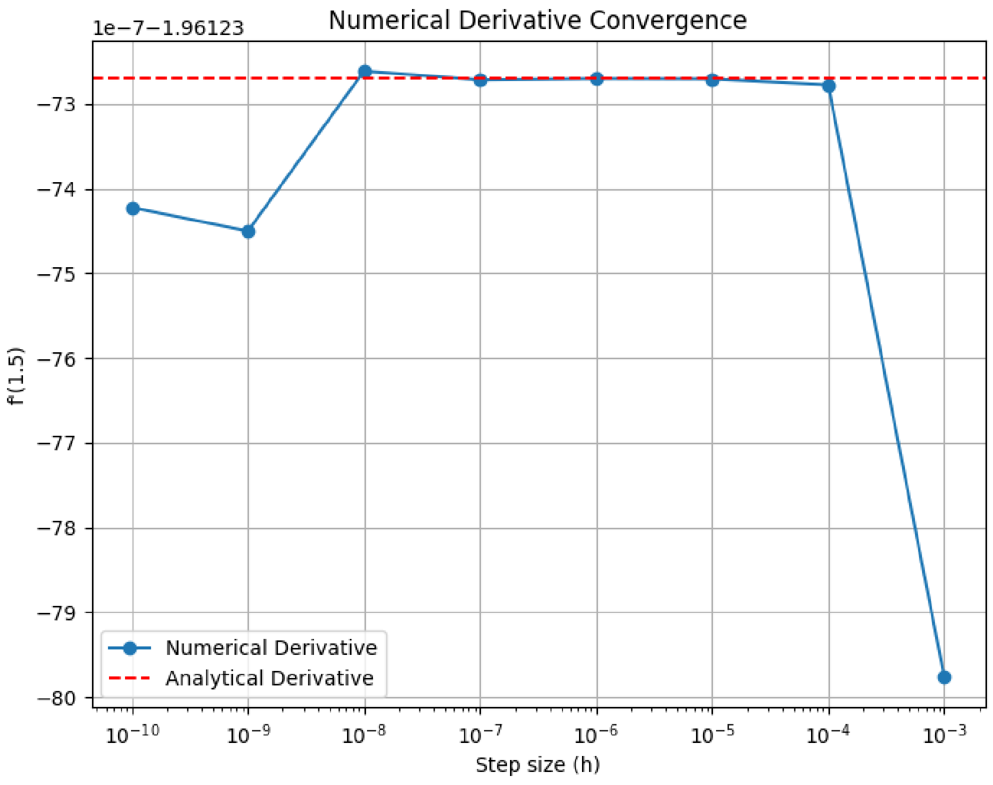
Compare `fx.dual` with `analytical_derivative` and **numerical derivative**. They are very close.

```
Using Dual Numbers:

f(1.5) = 0.15665054756073515
f'(1.5) = -1.9612372705533612  - this is the derivative obtained by Dual

Analytical Derivative:
f'(1.5) = -1.9612372705533612
```

```
Numerical Derivatives:
h = 1e-03, numerical derivative = -1.9612379763359749
h = 1e-04, numerical derivative = -1.9612372776108
h = 1e-05, numerical derivative = -1.9612372706409584
h = 1e-06, numerical derivative = -1.961237270337035
```



It can be seen that the **numerical derivative** converge at all the steps between `1e-8` and `1e-5` but diverge at very small step sizes or very large step sizes. This means that **numerical derivative** only gives us stability and accuracy in a specific range of step sizes hence that it is very sensitive to step-size selection. The sensitivity comes from the fact that at very small h, rounding errors dominate due to floating-point precision limitations. Conversely, at large h, truncation error increases because the approximation becomes less accurate.

This does not apply to dual numbers as it does not depend on step size. The derivative is computed exactly using algebraic properties. A dual number $x = a + b\epsilon$ automatically propagates both the value of $f(a)$ and the derivative $f'(a)$ when passed through a function $f(x)$. Hence it eliminates all approximation errors that happen during numerical differentiation. Dual numbers also compute the derivative in a **single pass** through the function, which is more efficient.

4 Building the test suite (task 6)

Many errors can arise when running any Python package. To make sure that the code is error-free and robust, the best way is to write a test suite to check every functionality of the package every time we update its distribution. The way we develop the test suite is, firstly, to check whether all of the codes we implement in `Dual` give us the expected answers, and also think of any possible edge cases cases.

In the package file `dual_autodiff`, we have one `__init__.py` file and one module (`.py` files). `dual.py` is the main `Dual` class where we implement the core functionality like mathematical operations. We also add an extended definitions for the `Dual` class, where we define functions to simplify and accelerate computations for more complex functions such as *tanh*.

Accordingly, in the test file, we want to a file corresponding to these modules called `autodiff_tools.py`. This file provides all test cases for the `dual.py` module, focusing on special cases.

The first thing we want to test is whether the function is correct. We want to have examples and use `assert` to check whether the package result in the answer that we expected.

Next, there are several types of errors, however, because of the nature of the module, we may only encounter `ZeroDivisionError`, `TypeError`, `ValueError` and `AttributeError`. Since we want accuracy in the package, there is no Exception handling.

`TypeError` occurs when an operation or function is applied to an object of inappropriate type. For example,

```
    def test_initialization_with_non_numeric_real():
    with pytest.raises(TypeError):
        Dual("two", 1)
```

`ZeroDivisionError` happens when attempting to divide by zero. For example,

```
    def test_division():
    x = Dual(6, 2)

    # Division by Dual with zero real part (raises error)
    with pytest.raises(ZeroDivisionError):
        x / Dual(0, 0)
```

Last but not least, `AttributeError` raised when an attribute referenced is unavailable, i.e, not defined in the package. An example of it is

```
    def test_attribute_error_on_nonexistent_attribute():
    x = Dual(2, 1)
    with pytest.raises(AttributeError):
        value = x.non_existent_attribute
```

With more than 20 tests conducted, the `dual.py` file is verified and implements all necessary mathematical concepts and the package can be used confidently.

5 Documentation with Sphinx (task 7)

The objective of this task was to develop comprehensive and user-friendly documentation for the `dual_autodiff` Python package using Sphinx [4]. The documentation is intended to assist users in understanding and using the package effectively by providing clear explanations, detailed examples, and interactive tutorials.

These are the steps that we need to make:

Sphinx Setup

The `docs` directory contains a basic Sphinx configuration that was generated using:

```
sphinx-quickstart
```

This initial setup created essential files such as `conf.py`, `index.rst`, and a `Makefile`.

Integration with Code and Notebooks

Docstrings: The documentation is built directly from Python docstrings. We follow Google-style docstring conventions, the codebase's function and class descriptions are automatically pulled into the documentation.

Tutorial Notebook: A Jupyter notebook named `TutorialNotebook.ipynb` is included in the `docs` folder. This notebook provides a hands-on tutorial, guiding new users through the functionalities of the `dual_autodiff` package. It includes:

- **Introduction:** Introduce the concept of Dual numbers and why the package is needed in the modern world.
- **Explanations:** Clear textual explanations accompany the code to ensure that users understand what the library does and how to apply it to their own problems.
- **Code Snippets:** Step-by-step examples on how to instantiate dual numbers and apply forwardmode differentiation.
- **Examples:** Demonstrations of how to perform automatic differentiation on various mathematical functions.

Building the Documentation

To generate the HTML documentation, simply run:

```
cd docs
make html
```

This command processes the `index.rst`, imports docstrings from the `dual_autodiff` codebase. We also added two `.rst` files, one is `tutorials.rst` to read the Jupyter notebook and the other `api_reference.rst` to read the API. The resulting HTML files are placed in `docs/_build/html`, ready to be opened in a browser.

Outcome

The final documentation site provides:

- A clean index page that introduces `dual_autodiff`.
- API documentation automatically generated from the codebase's docstrings, ensuring accuracy and up-to-date information.
- A tutorial notebook that illustrates practical use cases, helping users quickly grasp the core functionalities of the package.

By integrating docstrings with Sphinx and providing a user-friendly notebook tutorial, the `dual_autodiff` documentation ensures that both new and experienced users can understand the project's purpose, capabilities, and intended usage patterns.

6 Cython and performance comparison (task 8+9)

6.1 Cythonize pure Python

The core code consists of these following files:

```
│ pyproject.toml
└─ dual_autodiff/
   ├─ __init__.py
   └─ autodiff_tools.py
```

To cythonize the package `dual_autodiff`, we follow the steps as stated in the lecture notes:

1. Move the python files into a `/dual_autodiff_x` folder.
2. Rename the `.py` files into `.pyx` files.
3. Modify the `pyproject.toml` file and add `setup.py` file.

Then from the inside the folder we can install `dual_autodiff_x` by using the line `python setup.py build_ext --inplace`. The hardest part of the task is modifying the `setup.py` file because we want to know the package structure and repository well to create the correct setup.

6.2 Performance Comparision

We want to analyse the difference in performance of the Dual number class using **pure Python version** (`dual_autodiff.dual`) and the `Cythonized version` (`dual_autodiff.dual.dual`). From here we can make up our mind of which version to choose when it comes to optimisation and computational efficiency.

6.2.1 Steps

1. Implementations:
   - Pure Python version of the Dual class is implemented using standard Python constructs. It includes methods for basic arithmetic operations, leveraging Python's dynamic typing and interpreted nature.
   - Cythonized version translates the Dual class into Cython with optional, C-inspired syntax extensions that yields performance comparable to C.
   - Both of the implementations were tested within the same file `dual_autodiff.ipynb` to ensure fairness and on MacBook Air with Apple M1 chip running Python 3.9.
2. Performance measurement: generating 2 test functions to perform arithmetic operations (addition and multiplication). The `timeit` module was used to measure the execution time

of both functions in many numbers of operations `n`: 1,000; 5,000; 10,000; 50,000; 100,000; 500,000. Each test was run multiple times to obtain average execution times, ensuring statistical reliability.

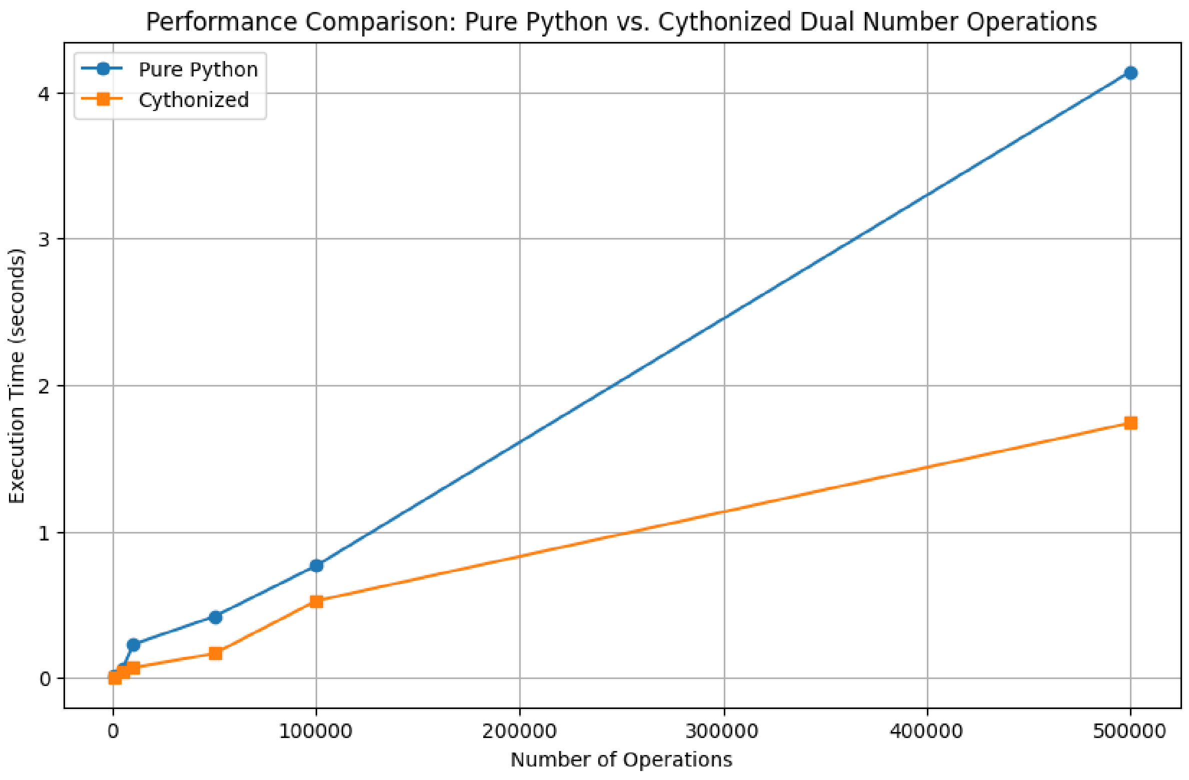
3. Data visualisation : execution time was plotted using `matplotlib` to visually compare performance.

6.2.2 Results

This table and plot below summarise the execution time for both implementations.

Table 1: Execution Times and Speedup Factors

Number of Operations (n)	**Pure Python Time (s)**	**Cythonized Time (s)**	**Speedup Factor**
1,000	0.0115	0.0046	2.47× faster
5,000	0.0591	0.0415	1.42× faster
10,000	0.2274	0.0687	3.31× faster
50,000	0.4178	0.1646	2.54× faster
100,000	0.7642	0.5241	1.46× faster
500,000	4.1356	1.7390	2.38× faster



Across all tested operation counts, the Cythonized implementation outperformed the pure Python version, achieving speedup factors ranging from 1.42× to 3.31× faster. In simple operations like addition or a small number of multiplications, Python's overhead is not as significant relative to the actual computation. Hence, the speedup from Cython is smaller. For computationally expensive operations (e.g., large loops or repeated calculations), Cython has significant speedups.

The significant performance gains observed in the Cythonized implementation can be attributed to several factors:

- Cython allows for better management of how memory is used, helping the CPU access data more quickly and reducing delays.
- By turning Python code into C, Cython removes the slow step of interpreting code, making the program run faster, especially in loops that perform many calculations. [5]

7 Linux Wheel Creation with `cibuildwheel` (task 10)

Create Wheels

To generate Linux wheels for `dual_autodiff_x`, we used `cibuildwheel` [6], which employs Docker to create portable and compliant binary distributions. The objective was to build wheels for the following environments:

- Python 3.10 on `manylinux_x86_64` (i.e., `cp310-manylinux_x86_64`)
- Python 3.11 on `manylinux_x86_64` (i.e., `cp311-manylinux_x86_64`)

We build the wheels by running `cibuildwheel` with the appropriate configuration (such as specifying the `CIBW_ARCHS` and `CIBW_MANYLINUX_X86_64_IMAGE` environment variables),

```
export CIBW_BUILD="cp310-manylinux_x86_64 cp311-manylinux_x86_64"
export CIBW_ARCHS=x86_64
cibuildwheel --platform linux
```

we obtained the following wheels in the `wheelhouse` directory:

```
dual_autodiff_x-0.0.1b2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
dual_autodiff_x-0.0.1b2-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
```

These wheels were successfully built, we make to make sure the build configuration and the correct Docker images and environment variables are set accordingly.

Verification of Wheel Contents

To confirm that the wheels do not contain the original `.pyx` source code files, but only the compiled `.so` (and `.pyd` on other platforms) files, the following steps were taken:

1. Extract the wheel's contents:

   ```
       unzip dual_autodiff_x-0.0.1b2-cp310-cp310-manylinux_2_17_x86_64.
       manylinux2014_x86_64.whl -d wheel_contents
       unzip dual_autodiff_x-0.0.1b2-cp311-cp311-manylinux_2_17_x86_64.
       manylinux2014_x86_64.whl -d wheel_contents
   ```

2. Inspect the `wheel_contents` directory to ensure it contains only the expected binary files.

The desired final state is to produce two wheels that include only the compiled binary artifacts. Adjusting the `cibuildwheel` configuration to target the `x86_64` architecture and verifying the file contents as described above ensures compliance with the specified requirements.

8 Uploading Wheels to GitLab and Verifying Installation (task 11)

After building the required Linux-compatible wheels for the `dual_autodiff_x` package, these wheels were uploaded to the project's GitLab repository. Storing pre-compiled wheels in a repository ensures ease of access and eliminates the need for source compilation on end-user machines.

8.1 Procedure

1. **Wheel Creation:** The wheels were generated using `cibuildwheel`, which runs in a Docker environment. This ensures that the final wheels comply with the `manylinux` specifications and are fully portable.
2. **Uploading to GitLab:** Once generated, the wheel files were uploaded directly to the GitLab project repository, under `dual_autodiff_x/wheelhouse` directory. This makes the wheels easily accessible to collaborators and end-users.
3. **Verification of Installation:** To confirm that the wheels work as intended, they were downloaded from the GitLab repository onto the Docker Desktop and installed using:

```
docker build --platform linux/amd64 -t my-python-app .
docker run -p 8888:8888 my-python-app
```

8.2 Testing the Installation

After installing the wheel, the `dual_autodiff_x` package was tested by running the provided examples in the tutorial notebook (i.e., `TutorialNotebook_x.ipynb`). The code executed successfully, confirming that:

- The wheels were properly built and require no additional compilation.
- Users can immediately utilize the package upon installation.
- The notebook examples ran smoothly, validating the integrity and functionality of the distributed wheels.

8.3 Outcome

By hosting the wheels in the GitLab repository, the distribution and installation process is significantly simplified. Users can simply download and install the pre-built wheels, thus avoiding common build-related complications. This approach ensures a more user-friendly and robust installation experience, allowing researchers and practitioners to focus on leveraging the `dual_autodiff_x` library's capabilities rather than dealing with installation hurdles.

References

[1] D. S. d. a. s. Brett Cannon brett at python.org, Nathaniel J. Smith njs at pobox.com, “Pep 518 – specifying minimum build system requirements for python projects.” https://peps.python.org/pep-0518/.

[2] Dual number Wikipedia Explanation, https://en.wikipedia.org/wiki/Dual$_n$*umber*.

[3] Unlock the Magic of Python with __slots__: Boost Performance Save Memory!, https://medium.com/@sompod123/unlock-the-magic-of-python-with-slots-boost-performance-save-memory-348400739072.

[4] Sphinx, https://www.sphinx-doc.org/en/master/.

[5] I. M. Wilbers, H. P. Langtangen, and Å. Ødegård, “Using cython to speed up numerical python programs,” *Proceedings of MekIT*, vol. 9, pp. 495–512, 2009.

[6] cibuildwheel package, https://cibuildwheel.pypa.io/en/stable/.

Word count : 2787