Team 01 - Milestone 1

Team Members: Sijia (Nancy) Li, Qingyang (Catherine) Ni, Yuqing (Lily) Pan, Hongyi (Chris) Sun, Jiashu Xu

1.0 Introduction

Differentiation is the operation of finding derivatives of a given function with applications in many disciplines. In classical mechanics, time derivatives are particularly useful in describing physical processes. For example, the first derivative of an object's change in position (displacement) with respect to time is the velocity. The second derivative of an object's displacement is the acceleration. Another famous example is Euler equations, a system of conservation laws (i.e., conservation of mass, momentum, and energy) given by three partial differential equations. In optimization, differentiation can be used to find mimina and maxima of functions. For a single variable, critical or stationary points are points in which the first derivative is zero. In higher dimensions, critical points are points in which the gradient is zero. Another useful technique is the second derivative test, whereby the second derivative of a single variable function or the eigenvalues of the Hessian matrix (a scalar-function's second partial derivatives) in higher dimensions can be used to test for local minimum and local maximum. In deep learning, optimization is used to train neural networks by finding a set of optimal weights that minimizes the loss function.

In particular, there are three techniques for computing derivatives: symbolic differentiation, numerical differentiation, and automatic differentiation. Symbolic differentiation finds derivatives by applying various differentiation rules (e.g., sum rule, constant rule, power rule) to break complex functions into simpler expressions. However, simplification of a complex function or a large number of functions can lead to an exponentially large number of expressions to evaluate. This means that although symbolic differentiation will yield accurate results, it can be too slow and costly to perform such computations in real-life. Numerical differentiation estimates the derivatives from known values of the functions (e.g., finite difference approximation method). However, this technique can lead to inaccurate results due to floating point errors. Automatic differentiation (or algorithmic differentiation) solves these problems. Specifically, automatic differentiation breaks down the original functions into elementary operations (e.g., addition, subtraction, multiplication, division, exponentiation, and trigonometric functions) and then evaluate the chain rule step by step. This technique is able to compute derivatives efficiently (with lower cost than symbolic differentiation) at machine precision (more accurate than numerical differentiation). Since derivatives are ubiquitous in many real-world applications, it is important to have a useful tool to perform differentiation. Natually, the best choice among the three aforementioned methods for computing derivatives is automatic diffferentiation, and `AutoDiff` is a Python package that implements this method.

2.0 Background

2.1 The Chain Rule

The chain rule helps differentiate composition functions. Applying the chain rule, the derivative of a function $f(g(x))$ with respect to a single independent variable x is given by:

$$\frac{df}{dx} = \frac{\partial f}{\partial g}\frac{dg}{dx}$$

For example, given $f(g(x)) = cos(2x)$ and $g(x) = 2x$, then $\frac{\partial f}{\partial g} = -sin(g)$, $\frac{dg}{dx} = 2$, and $\frac{df}{dx} = -2sin(2x)$. For functions with two or more variables, we are interested in the gradient of a function $f(\mathbf{x})$ with respect to all independent variables $\mathbf{x}$, $\nabla_x f$. Applying the chain rule, this is given by:

$$\nabla_{\mathbf{x}} f = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \nabla g_i(\mathbf{x})$$

where g_i's are functions with m variables, $i = 1, 2, \ldots, n$. For example, given $f(g_1(\mathbf{x}), g_2(\mathbf{x})) = sin(g_1) - 2g_2$, and $g_1(\mathbf{x}) = x_1 + 2x_2, g_2(\mathbf{x}) = 3x_1^2 x_2$, then $\nabla g_1(\mathbf{x}) = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \nabla g_2(\mathbf{x}) = \begin{bmatrix} 6x_1x_2 \\ 3x_1^2 \end{bmatrix}$, and $\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial g_1} \nabla g_1(\mathbf{x}) + \frac{\partial f}{\partial g_2} \nabla g_2(\mathbf{x}) = cos(x_1 + 2x_2) \begin{bmatrix} 1 \\ 2 \end{bmatrix} - 2(3x_1^2 x_2) \begin{bmatrix} 6x_1x_2 \\ 3x_1^2 \end{bmatrix}$. This is an example with $m = 2$ and $n = 2$.

2.2 Elementary Functions

An elementary function is a combination of elementary operations, and constant, algebraic, exponential, and logarithmic functions as well as their inverses, e.g., $2x, e^x, sin(x), x + 5$. We can decompose a function into smaller parts (elementary functions) in which their symbolic derivatives can be easily computed. The table below shows some examples of elementary functions with their respective derivatives.

Elementary Function	Derivative of the Function
c	0
ax	a
x^2	$2x$
e^x	e^x
$ln(x)$	$\frac{1}{x}$
$sin(x)$	$cos(x)$
$cos(x)$	$-sin(x)$
$tan(x)$	$sec^2(x)$
cf	cf'
x^n	nx^{n-1}
$f + g$	$f' + g'$
$f - g$	$f' - g'$
fg	$fg' + f'g$
f/g	$\frac{f'g - g'f}{g^2}$

2.3 Forward Mode

2.3.1 Forward Primal Trace

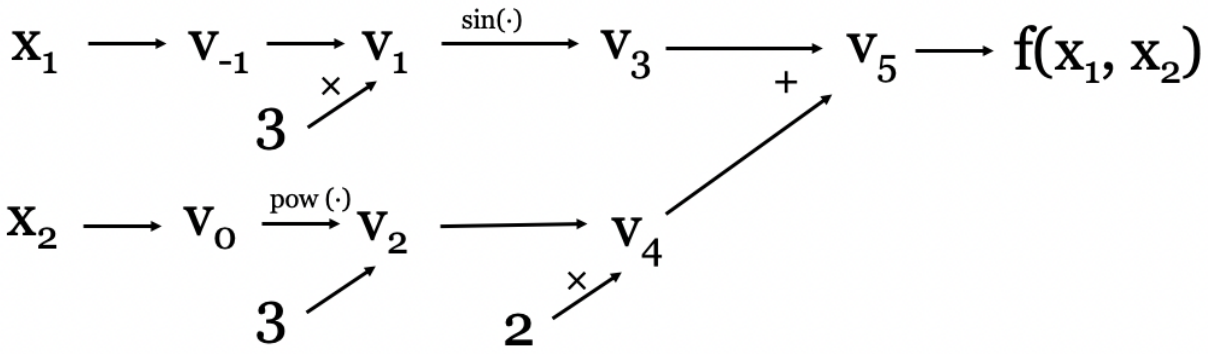
Take the function $f(x_1, x_2) = sin(3x_1) + 2(x_2)^3$ as an example. We develop the forward primal trace by finding intermediate results v_j in which j represents an elementary operation. This is achieved by working from the inside out. Given an arbitrary point (x_1, x_2), we can evaluate the intermediate results at the point. The following table show the forward primal trace of the function $f(x_1, x_2) = sin(3x_1) + 2(x_2)^3$ evaluated at the point $(\frac{\pi}{6}, 2)$.

Intermediate	Elementary Operation	Numerical Value
$v_{-1} = x_1$	$\frac{\pi}{6}$	0.52359877559

Intermediate	Elementary Operation	Numerical Value
$v_0 = x_2$	2	2
v_1	$3v_{-1}$	1.57079632679
v_2	v_0^3	8
v_3	$sin(v_1)$	1
v_4	$2v_2$	16
v_5	$v_3 + v_4$	17

2.3.2 Computational (Forward) Graph

The computational graph is a way of visualizing the partial ordering of elementary operations with each node representing an intermediate result. The computational graph for the aforementioned function $f(x_1, x_2) = sin(3x_1) + 2(x_2)^3$ is shown below.



2.3.4 Forward Tangent Trace

Along with the forward primal trace, we also develop the forward tangent trace simultaneously by computing the directional derivative for each intermediate variable $D_p v_j$. By definition, the directional derivative is given by the following equation.

$$D_p v_j = (\nabla v_j)^T p = \sum_{i=1}^{m} \frac{\partial v_j}{\partial x_i} p_i$$

In this definition, p is a m-dimensional seed vector in which m is the number of independent variables. We can specify the derivative of interest using this seed vector. For example, if the goal is to find $\frac{\partial v_j}{\partial x_5}$, then the element p_5 will be one and the remaining elements in the p vector will be zero. The seed vector can be freely selected by the user. Generalizing this, the forward mode automatic differentiation is in essence computing $\nabla f \cdot p$ for a scalar function $f(x)$ and $J \cdot p$ for a vector function $f(x)$ (J is the Jacobian).

The following table shows the forward primal trace and the forward tangent trace evaluated for the seed vectors $p_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $p_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

Forward Primal Trace	Forward Tangent Trace	Pass $p^{(j=1)} = [1, 0]^T$	Pass $p^{(j=2)} = [0, 1]^T$
$v_{-1} = x_1 = \frac{\pi}{6}$	$D_p v_{-1} = p_1$	$D_p v_{-1} = 1$	$D_p v_{-1} = 0$
$v_0 = x_2 = 2$	$D_p v_0 = p_2$	$D_p v_0 = 0$	$D_p v_0 = 1$
$v_1 = 3v_{-1} = \frac{\pi}{2}$	$D_p v_1 = 3D_p v_{-1}$	$D_p v_1 = 3$	$D_p v_1 = 0$
$v_2 = v_0^3 = 8$	$D_p v_2 = 3v_0^2 D_p v_0$	$D_p v_2 = 0$	$D_p v_2 = 12$
$v_3 = sin(v_1) = 1$	$D_p v_3 = cos(v_1) D_p v_1$	$D_p v_3 = 0$	$D_p v_3 = 0$
$v_4 = 2v_2 = 16$	$D_p v_4 = 2D_p v_2$	$D_p v_4 = 0$	$D_p v_4 = 24$
$v_5 = v_3 + v_4 = 17$	$D_p v_5 = D_p v_3 + D_p v_4$	$D_p v_5 = 0$	$D_p v_5 = 24$

2.3.5 Dual Numbers

By definition, a dual number is given by the equation $z = a + b\epsilon$, where $a, b \in \mathbb{R}$ and ϵ is a very small number not equal to zero such that $\epsilon^2 = 0$. a is the real part and b is the dual part. This structure is very helpful in encoding the primal trace and tangent trace in forward mode automatic differentiation. The primal trace can be encoded by the real part and the tangent trace can be encoded by the dual part, hence the equation $z_j = v_j + D_p v_j \epsilon$.

Going back to the example, we can compute the last intermediate state using dual numbers. The last intermediate state is z_5 which is equal to $z_3 + z_4$, where z_3, z_4, and z_5 are dual numbers. Let $z_3 = a_3 + b_3\epsilon$ and $z_4 = a_4 + b_4\epsilon$, then $z_5 = (a_3 + b_3\epsilon) + (a_4 + b_4\epsilon) = (a_3 + a_4) + (b_3 + b_4)\epsilon$, where $a_4 + b_4\epsilon$ is the real part and $b_3 + b_4$ is the dual part.

2.4 Reverse Mode

The reverse mode in automatic differentiation is a two-pass process for recovering the partial derivatives $\frac{\partial f_i}{\partial v_{j-m}}$, where f_i's are output functions and v_{j-m} are the intermediate variables. These partial derivatives are called the adjoints. In other words, $\bar{v}_{j-m} = \frac{\partial f_i}{\partial v_{j-m}}$ and $\bar{v}_{j-m}$ is the adjoint of v_{j-m}. The forward pass in the reverse mode will compute the the change in child node v_j with respect to v_j's parent node(s) v_i, denoted as $\frac{\partial v_j}{\partial v_i}$. After that, the reverse pass will build up the chain rule using the formula $\bar{v}_i = \bar{v}_i + \frac{\partial v_f}{\partial v_j}\frac{\partial v_j}{\partial v_i} = \bar{v}_i + \bar{v}_j\frac{\partial v_j}{\partial v_i}$, with $\bar{v}_i$ initialized to zero for all i and node j is a child of node i. The last intermediate state is always equal to 1 as the last nodes has no children. The following table illustrates the reverse mode computation for the same example in Section 2.3: $f(x_1, x_2) = sin(3x_1) + 2(x_2)^3$ evaluated at the point $(\frac{\pi}{6}, 2)$.

Forward Pass		Reverse Pass
Intermediate	Partial Derivative	Adjoint
$v_{-1} = x_1 = \frac{\pi}{6}$		$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1\frac{\partial v_1}{\partial v_{-1}} = 0$

Forward Pass		Reverse Pass
$v_0 = x_2 = 2$		$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} = 24$
$v_1 = 3v_{-1} = \frac{\pi}{2}$	$\frac{\partial v_1}{\partial v_{-1}} = 3$	$\bar{v}_1 = \bar{v}_1 + \bar{v}_3 \frac{\partial v_3}{\partial v_1} = 0$
$v_2 = v_0^3 = 8$	$\frac{\partial v_2}{\partial v_0} = 3v_0^2 = 12$	$\bar{v}_2 = \bar{v}_2 + \bar{v}_4 \frac{\partial v_4}{\partial v_2} = 2$
$v_3 = sin(v_1) = 1$	$\frac{\partial v_3}{\partial v_1} = cos(v_1) = 0$	$\bar{v}_3 = \bar{v}_3 + \bar{v}_5 \frac{\partial v_5}{\partial v_4} = 1$
$v_4 = 2v_2 = 16$	$\frac{\partial v_4}{\partial v_2} = 2$	$\bar{v}_4 = \bar{v}_4 + \frac{\partial f}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_4 + \bar{v}_5 \frac{\partial v_5}{\partial v_4} = 0 + 1 = 1$
$v_5 = v_3 + v_4 = 17$	$\frac{\partial v_5}{\partial v_3} = 1; \frac{\partial v_5}{\partial v_4} = 1$	$\bar{v}_5 = \frac{\partial f}{\partial v_5}\frac{\partial v_5}{\partial v_5} = 1$

3.0 How to Use the `AutoDiff` Package

3.1 Installation

The user can install `AutoDiff` using the following command:

```
pip install AutoDiff
```

3.2 Usage

The following code snippet demonstrates how users will interact with our package.:

3.2.1 Example 1: One Independent Variable

```
>>> import AutoDiff as ad
>>> import numpy as np
>>> # Create a node x with value 5.
>>> x = ad.Node(5)
>>> # Create a function y = exp(cos(x)+2).
>>> y = np.exp(np.cos(x)+2)
>>> # Create a forward mode instance.
>>> fwd = ad.Forward(y)
>>> # Evaluate the value of y at x=5
>>> fwd.evaluate()
9.81255
>>> # Calculate dy/dx = -exp(cos(x))+2sin(x) for x = 5.
>>> fwd.grad()
-2.68892
>>> # Create a reverse mode instance.
>>> rev = ad.Reverse(y)
>>> # Evaluate the value of y at x=5
>>> rev.evaluate()
9.81255
>>> # Calculate dy/dx = -exp(cos(x))+2sin(x) for x = 5.
```

```
>>> rev.grad()
-2.68892
```

3.2.2 Example 2: Two Independent Variables

```
>>> import AutoDiff as ad
>>> import numpy as np
>>> # Create two nodes x1 with value 5 and x2 with value 3.
>>> x1 = ad.Node(5)
>>> x2 = ad.Node(3)
>>> # Create a function y = exp(cos(x1)+2sin(x2)).
>>> y = np.exp(np.cos(x1)+2*np.cos(x2))
>>> # Create a forward mode instance.
>>> fwd = ad.Forward(y)
>>> # Evaluate the value of y at x1 = 5, x2 = 3
>>> fwd.evaluate()
1.761036
>>> # Calculate dy/dx at x1 = 5, x2 = 3
>>> fwd.grad()
[1.688700, -3.486825]
>>> # Create a reverse mode instance.
>>> rev = ad.Reverse(y)
>>> # Evaluate the value of y at x1 = 5, x2 = 3
>>> rev.evaluate()
1.761036
>>> # Calculate dy/dx at x1 = 5, x2 = 3
>>> rev.grad()
[1.688700, -3.486825]
```

4.0 Software Organization

This section briefly discusses our plan on organizing the Python package `AutoDiff`. The directory structure is shown below, following the recommended python package structure.

```
. # root dir
├── docs # development documentations
│   ├── milestone1.ipynb
│   └── ...
├── reference # autogenerate docs by sphinx
│   ├── Makefile
│   └── source
│       └── ...
├── examples
│   └── example1.ipynb # some examples of how to use this library
├── LICENSE
├── README.md
├── setup.cfg
├── setup.py
├── test
│   ├── test_node.py # test for node module
│   ├── test_forward.py # test for forward module
│   └── test_reverse.py # test for reverse module
└── AutoDiff # our library source code
    ├── __init__.py
    ├── node.py # node module
    ├── forward.py # forward module
    └── reverse.py # reverse module
```

The functionalities of each aforementioned directory/file is shown below.

- `docs`: This directory contains the major deliverables from each milestone.

- `reference` : We will use sphinx for automated documentation generation due to its popularity. Such flexible automated tool will read content from the docstring and create documentations automatically.
- `examples` : This directory provides examples to show users how to use the package.
- `LICENSE` : This is the file specifying the license for our python package.
- `README.md` : This is the readme for the project repository.
- `setup.cfg` : This file contains the setup configuration and will be used by setuptools.
- `setup.py` : This is used for defining and publishing our package. It contains the version number, dependencies, upstream GitHub URL, and etc. The functionality is described in here.
- `test` : This is the directory that contains the pytest modules. They are unit tests. We plan to separate these tests into modules. We plan to use CodeCov to show the test coverage for our code.
- `AutoDiff` : This is the folder for the actual source code. We will follow the recommended structure of python module shown in the python documentation.

To clearly visualize the directory structure, we did not include files such as `.gitignore` , `.github` for github action, `.codecov.yml` for CodeDev, and `requirements.txt` for listing dependencies in local development.

For distribution, we will package the code by the standard setuptools and then distribute our software using Python Package Index (PyPi). In this way, future users can download the package by simply using the following command.

```
pip install AutoDiff
```

GitHub action will be used to conduct CI with pytest. Each time a commit is pushed to GitHub, a pytest test will be initiated. Moreover, the traditional Git workflow will be followed. The main branch will be used for stable release of the package and all development will be conducted in separate branches. We will only merge to main branch if the code in development branch passes the test. Finally, CodeCov will be used to show the test coverage of the package.

5.0 Implementation

The `AutoDiff` package contains three classes in the order of implementation: `Node` , `Forward` , and `Reverse` . Our implementation need to depend on other libraries such as NumPy and Matplotlib.

Before computing the gradient of the input function, the input function is wrapped into a graph structure that stores the partial ordering of the intermediate results v_j (Section 2.3). To realize the graph structure, we need to implement the class `Node` , where each node contains a reference of its parents. Since we have constructed the core data structure `Node` of the input function, we can implement the forward mode of automatic differentiation in the class `Forward` to compute the Jacobian of the input function. In addition, we plan to implement the class `Reverse` to realize the reverse mode of automatic differentiation as the extension funtionality. The following subsections are more detailed introduction of the classes with class attributes and methods.

5.1 `class Node`

The class `Node` turns the function into the core data structure ---- graph. Attributes of `Node` includes `self.val` , `self.der` , `self.parent` , `self.op` . The class `Node` incorporate the structure of dual numbers (Section 2.3.5), where `self.val` + `self.der` ε is assemble to the dual number's structure $a + b\varepsilon$. In this class, we further overload the elementary operators and useful numpy functions. We also plan to implement a method plot_graph to visualize the input function.

```
import numpy as np
import Matplotlib as plt
class Node:
  '''
  This class implements the element of the function graph.
  '''
  __init__(self, value, derivative=0):
    self.val = value
    self.der = derivative
    self.parent = []
    self.op = []

  update_node(self, parent, op):
    self.parent = parent
    self.op = op
    return self

  '''
  Overloading elementary operators in numpy
  '''
  __neg__(self):
    value = -self.val
    derivative = -1 * self.der
    return Node(value, derivative).update_node([self], ['-'])

  __add__(self, other: Node):
    value = self.val + other.val
    derivative = self.der + other.der
    return Node(value, derivative).update_node([self, other], ['+'])

  __mul__(self, other: Node):
    value = self.val * other.val
    derivative = self.der * other.val + other.der * self.val
    return Node(value, derivative).update_node([self, other], ['*'])

  sqrt(self):
    value = np.sqrt(self.val)
    derivative = 0.5/np.sqrt(self.val) * self.der
    return Node(value, derivative).update_node([self], ['sqrt()'])

  log(self, base=e):
    value = np.log(self.val)
    derivative = 1 / self.val * self.der
    return Node(value, derivative).update_node([self, other], ['log()'])

  exp(self):
    value = np.exp(self.val)
    derivative = np.exp(self.val) * self.der
    return Node(value, derivative).update_node([self], ['exp()'])

  sin(self):
    value = np.sin(self.val)
    derivative = np.cos(self.val) * self.der
    return Node(value, derivative).update_node([self], ['sin()'])

  '''
  some other elementory operators that needs to be overloaded
  '''
  __radd__(self, other)
  __sub__(self, other: Node)
  __rsub__(self, other)
  __truediv__(self, other: Node)
  __pow__(self, other)
```

```
  cos(self)
  tan(self)
  __matmul__(self, other: Node)
  ...

  update_graph(self, p):
    '''
    self is the root of the function
    p is the seed vector
    update_graph loops to the leaf nodes, update p and reconstruct the graph

    '''
    return self

  plot_graph(self):
    '''
    self is the root of the function
    plot_graph loops to the leaf nodes to create a visualization
    '''
```

5.2 `class Forward`

In the class `Forward`, we need to model arbitrary high-level function f. We treat a vector function $f: \mathbb{R}^m \mapsto \mathbb{R}^n$ as a list of scalar functions $f: \mathbb{R}^m \mapsto \mathbb{R}$. The `evaluate()` method evaluates the function at user-given values of x. The `grad()` method will compute the full Jacobian by looping through the vector of scalar functions.

```
from node import Node
class Forward:
  '''
  Forward mode AD for vector functions of vectors
  '''
  __init__(self, f_lst):
    '''
    User should input a list of function root Nodes of length n as a vector
    function
    '''
    self.f_lst = f_lst

  evaluate(self):
    '''
    Evaluate f_lst(x)
    '''

  grad(self):
    '''
    Evaluate the full Jacobian in forward mode.
    For each scalar function f, use m passes with different seed vector p.
    Stack the gradient rows into the full Jacobian.
    '''
    m = len(x)
    n = len(self.f_lst)
    p_lst = list of the seed vector p
    f_list_grad = [] #
    for f in self.f_lst:
      # Evaluate gradient for each scalar funtion
      f_grad = []
      for p in p_lst:
        # Evaluate through the Graph f in the direction of p
        f.update_graph(p)
        grad_p = f.der
```

```
        f_grad.append(grad_p)
      f_list_grad.append(f_grad)
    return np.array(f_list_grad) # Jacobian
```

5.3 `class Reverse`

In the class `Reverse`, we perform the reverse mode automatic differentiation. The `evaluate()` method evaluates the function at user-given values of x. The `grad()` method will compute the full Jacobian by looping through the vector of scalar functions, notice that by our implementation of the `Node` class, we get the partial derivatives column in forward pass of the reverse mode computation table while initializing functions.

```
from node import Node
class Reverse:
  '''
  Reverse mode AD for vector functions of vectors
  '''
  __init__(self, f_lst):
    self.f_lst = f_lst

  evaluate(self):
    '''
    Evaluate f_lst(x)
    '''

  grad(self):
    '''
    Evaluate Jacobian in reverse mode.
    For each scalar function f, use a forward pass and a reverse pass.
    Stack the gradient rows into the full Jacobian.
    '''
    n = len(self.f_lst)
    f_list_grad = [] #
    for f in self.f_lst:
      # Evaluate gradient for each scalar funtion
      f_grad = gradient for f
      f_list_grad.append(f_grad)
    return np.array(f_list_grad)
```

6.0 License

We chose to use the MIT License. As described by the MIT license, it will allow users to reuse the code for commercial or private use, distribution and modification (essentially any purpose), as long as the users include the original copy of the MIT license in their distribution. The MIT license is a permissive license as it does not require the user to make their work publicly available as well. Automatic differentiation is a project that has been worked on by many users, i.e. the software is not very substantial, and the project can be very useful for anyone wants to differentiate their functions quickly. Since this is a course project, we do not want to profit from the software, but we want to use the software as a component of a broader service. Therefore we do not feel the need to limit the use of our software, and it is not necessary for us to use a copyleft license to force out user to make their project open source. Due to similar reasons, we do not care about patents.

7.0 Feedback

7.1 Milestone 1

Team,

I am incredibly impressed by the overall quality for this piece of work !

Introduction(2/2): This is beautifully written and concise explained.

Background (2/2): Excellent !

How to use: (3/3) This section was really nicely done!

Software Organization(2/2): Well Done ! One nit pick is that it is not advisable to put init.py into your test folder (see here: [https://docs.pytest.org/en/latest/explanation/goodpractices.html#tests-outside-application-code])

The reason behind this is that modules are contained within the package. Tests are usually written in a separate directory (which is not a package).

Implementation(4/4): This is beautifully implemented. I have nothing else to add ...

License(2/2): Well done !

Total: (15/15)

7.2 Milestone 1 Updates

Based on the feedback, the init.py file in the test folder is removed. Some minor formatting edits are also made (e.g., resolving some "Unknown character" issues that showed up on GitHub).