CS 207 Final Project: Milestone 2

Group 19: Chen Shi, Stephen Slater, Yue Sun

November 2018

1 Introduction

Differentiation, i.e. finding derivatives, has long been one of the key operations in computation related to modern science and engineering. In optimization and numerical differential equations, finding the extrema will require differentiation. There are many important applications of automatic differentiation in optimization, machine learning, and numerical methods (e.g., time integration, root-finding). This software library will use the concept of automatic differentiation to solve differentiation problems in scientific computing.

2 How to Use *DeriveAlive*

2.1 How to install

The user can either download from GitHub or use use `pip` to install the package. Installation details are listed in Section 4.4. After installation, the user will need to import our package (see pseudocode below). Implicitly, this will import other dependencies (such as `numpy`), since we include those dependencies as imports in our module. Then, the user will define an input of type `Var` in our module. After this, the user can define a function in terms of this `Var`. Essentially, the user will give the initial input x and then apply f to x and store the new value and derivative with respect to x inside f. At each step of the evaluation, the program will process nodes in the implicit computation graph in order, propagating values and derivatives. The final output yield another `Var` containing $f(x)$ and $f'(x)$.

2.2 Basic demo

For example, consider the case $f(x) = \sin(x)+5\cdot tan(x/2)$. If the user wants to evaluate $f(x)$ at $x = a$, where $a = \pi/2$, the user will instantiate a `Var` object as `da.Var(np.pi/2)`. Then, the user will give the initial input a and set $y = f(a)$, which stores $f(a)$ and $f'(a)$ as attributes inside the `Var` y. This functionality will propagate throughout the graph with more variables in a recursive structure, where each evaluation trace creates a new `Var`. See the code below for a demonstration. Note that $\sin(\pi/2) = 1.0$ and $\tan(\pi/4) = 1.0$, and their evaluated derivatives $\cos(\pi/2) = 0$ and $\frac{1}{\cos^2(\pi/4)} = 2$. We assign $y = f(a)$ by explicitly applying the `Var`-specific operations of f to a. The closed form of the derivative is $f'(x) = \cos(x) + 5 * \frac{1}{\cos^2(x/2)} * \frac{1}{2}$.

```
# Install at command line using installation instructions in Section 4.4
pip install DeriveAlive
python
>>> import DeriveAlive.DeriveAlive as da
>>> import numpy as np

# Expect value of 6.0, derivative of 5.0
>>> x = da.Var([np.pi / 2])
>>> y = x.sin() + 5 * (x / 2).tan()
>>> print (y.val, y.der)
[6.0] [5.0]
```

3 Background

The chain rule, gradient (Jacobian), computational graph, elementary functions and several numerical methods serve as the mathematical cornerstone for this software. The mathematical concepts here come from CS 207 Lectures 9 and 10 on Autodifferentiation.

3.1 The Chain Rule

The chain rule is critical to AD, since the derivative of the function with respect to the input is dependent upon the derivative of each trace in the evaluation with respect to the input.

If we have $h(u(x))$ then the derivative of h with respect to x is:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u} \cdot \frac{\partial u}{\partial x} \tag{1}$$

If we have another argument $h(u, v)$ where u and v are both functions of x, then the derivative of $h(x)$ with respect to x is:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u} \cdot \frac{\partial u}{\partial x} + \frac{\partial h}{\partial v} \cdot \frac{\partial v}{\partial x} \tag{2}$$

3.2 Gradient and Jacobian

If we have $x \in \mathbb{R}^m$ and function $h\left(u\left(x\right), v\left(x\right)\right)$, we want to calculate the gradient of h with respect to x:

$$\nabla_x h = \frac{\partial h}{\partial u} \nabla_x u + \frac{\partial h}{\partial v} \nabla_x v \tag{3}$$

In the case where we have a function $h(x) : \mathbb{R}^m \to \mathbb{R}^n$, we write the Jacobian matrix as follows, allowing us to store the gradient of each output with respect to each input.

$$\mathbf{J} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots & \frac{\partial h_1}{\partial x_m} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots & \frac{\partial h_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial x_1} & \frac{\partial h_n}{\partial x_2} & \cdots & \frac{\partial h_n}{\partial x_m} \end{bmatrix}$$

In general, if we have a function $g\left(y\left(x\right)\right)$ where $y \in \mathbb{R}^n$ and $x \in \mathbb{R}^m$. Then g is a function of possibly n other functions, each of which can be a function of m variables. The gradient of g is now given by

$$\nabla_x g = \sum_{i=1}^{n} \frac{\partial g}{\partial y_i} \nabla_x y_i \left(x\right). \tag{4}$$

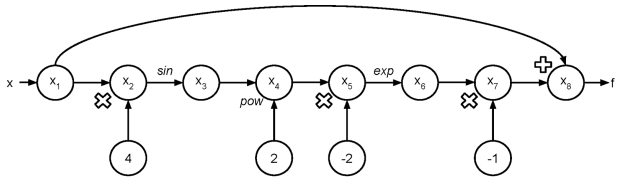
3.3 The Computational Graph

The computational graph lets us visualize what happens during the evaluation trace. The following example is based on Lectures 9 and 10. Consider the function:

$$f\left(x\right) = x - \exp\left(-2\sin^2\left(4x\right)\right)$$

If we want to evaluate f at the point x, we construct a graph where the input value is x and the output is y. Each input variable is a node, and each subsequent operation of the execution trace applies an operation to one or more previous nodes (and creates a node for constants when applicable).
As we execute $f(x)$ in the "forward mode", we can propagate not only the sequential evaluations of operations in the graph given previous nodes, but also the derivatives using the chain rule.

Figure 1: Sample computational graph for $f(x) = x - \exp\left(-2\sin^2(4x)\right)$.



3.4 Elementary functions

An elementary function is built up of a finite combination of constant functions, field operations $(+, -, \times, \div)$, algebraic, exponential, trigonometric, hyperbolic and logarithmic functions and their inverses under repeated compositions. Below is a table of some elementary functions and examples that we will include in our implementation.

Elementary Functions	Example
powers	x^2
roots	$\sqrt{x}$
exponentials	e^x
logarithms	$\log(x)$
trigonometrics	$\sin(x)$
inverse trigonometrics	$\arcsin(x)$
hyperbolics	$\sinh(x)$

4 Software Organization

4.1 Current directory structure

```
cs207-FinalProject/
                    README.md
                    LICENSE
                    DeriveAlive/
                                DeriveAlive.py
                    docs/
                         milestone1.pdf
                         milestone2.pdf
                    tests/
                          test_DeriveAlive.py
                    ...
```

4.2 Basic modules and their functionality

- `DeriveAlive`: This module contains our custom library for autodifferentiation. It includes functionality for a `Var` class that contains values and derivatives, as well as class-specific methods for the operations that our model implements (e.g., tangent, sine, power, exponentiation, addition, multiplication, and so on).

- `test_DeriveAlive`: This is a test suite for our module (explanation in the following section). It currently includes tests for scalar functions to ensure that the `DeriveAlive` module properly calculates values of scalar functions and gradients with respect to scalar inputs.

4.3 Where will your test suite live?

Our test suite is currently in a test file called `test_DeriveAlive.py` in its own `tests` folder. We use Travis CI for automatic testing for each push, and Coveralls for line coverage metrics. We have already set up these integrations, with badges included in the `README.md`. Users may run the test suite by navigating to the `tests/` folder and running the command `pytest test_DeriveAlive.py` from the command line (or `pytest tests` if the user is outside the `tests/` folder).

4.4 How can someone install your package?

We provide two ways for our package installation: GitHub and PyPI.

1. Installation from GitHub
   - Download the package from GitHub to your folder via these commands in the terminal:

     ```
     mkdir test_cs207
     cd test_cs207/
     git clone https://github.com/cs207-group19/cs207-FinalProject.git
     cd cs207-FinalProject/
     ```

   - Create a virtual environment and activate it

     ```
     # If you don't have virtualenv, install it
     sudo easy_install virtualenv
     # Create virtual environment
     virtualenv env
     # Activate your virtual environment
     source env/bin/activate
     ```

   - Install required packages and run module tests in `tests/`

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

   - Use DeriveAlive Python package (see demo in Section 2.2)

     ```
     python
     >>> import DeriveAlive.DeriveAlive as da
     >>> import numpy as np
     >>> x = da.Var([np.pi/2])
     >>> x
     Var([1.57079633], [1.])
     ...
     >>> quit()

     # deactivate virtual environment
     deactivate
     ```

2. Installation using PyPI

   We also utilized the Python Package Index (PyPI) for distributing our package. PyPI is the official third-party software repository for Python and primarily hosts Python packages in the form of archives called sdists (source distributions) or precompiled wheels. The url to the project is `https://pypi.org/project/DeriveAlive/`.

– Create a virtual environment and activate it

```
# If you don't have virtualenv, install it
sudo easy_install virtualenv
# Create virtual environment
virtualenv env
# Activate your virtual environment
source env/bin/activate
```

– Install DeriveAlive using pip. In the terminal, type:

```
pip install DeriveAlive
```

– Run module tests before beginning.

```
# Navigate to https://pypi.org/project/DeriveAlive/#files
# Download tar.gz folder, unzip, and enter the folder
pytest tests
```

– Use DeriveAlive Python package (see demo in Section 2.2)

```
python
>>> import DeriveAlive.DeriveAlive as da
>>> import numpy as np
>>> x = da.Var([np.pi/2])
>>> x
Var([1.57079633], [1.])
...
>>> quit()

# deactivate virtual environment
deactivate
```

5 Implementation

We plan to implement the forward mode of autodifferentiation with the following choices:

- Variable domain: The variables are defined as real numbers, hence any calculations or results involving complex numbers will be excluded from the package.
- Type of user input: Regardless of the input type (e.g., a float or a list or a numpy array), the `Var` class will automatically convert the input into a numpy array. This will provide flexibility in the future for implementing vector to vector functions.
- Core data structures: The core data structures will be classes, lists and numpy arrays.
  - Classes will help us provide an API for differentiation and custom functions, including custom methods for our elementary functions.
  - Lists will help us maintain the collection of trace variables and output functions (in the case of multi-output models) from the computation graph in order. For example, if we have a function $f(x) : \mathbb{R}^1 \to \mathbb{R}^2$, then we store $f = [f1, f2]$, where we have defined $f1$ and $f2$ as functions of x, and we simply process the functions in order. Depending on the extensions we choose for the project, we may use lists to store the parents of each node in the graph.
  - Numpy arrays are the main data structure during the calculation. We store the list of derivatives as a numpy array so that we can apply entire functions to the array, rather than to each entry separately. Each trace `Var` has a numpy array of derivatives where the length of the array is the number of input variables in the function. In the vector-vector case, if we have a function

$f : \mathbb{R}^m \to \mathbb{R}^n$, we can process this as $f = [f_1, f_2, \ldots, f_n]$, where each f_i is a function $f_i : \mathbb{R}^m \to \mathbb{R}$. Our implementation can act as a wrapper over these functions, and we can evaluate each f_i independently, so long as we define f_i in terms of the m inputs. Currently, the module supports scalar to scalar functions, but we have expanded several parts of the implementation to include arrays so that providing vector to vector functions will be a smooth transition.

- Our implementation plan currently includes 1 class which accounts for trace variables and derivatives with respect to each input variable.
  - `Var` class. The class instance itself has two main attributes: the value and the evaluated derivatives with respect to each input. Within the class we redefine the elementary functions and basic algebraic functions, including both evaluation and derivation. Since our computation graph includes "trace" variables, this class will account for each variable. Similar to a dual number, this class structure will allow us easy access to necessary attributes of each variable, such as the trace evaluation and the evaluated derivative with respect to each input variable. This trace table would also be of possible help in future project extensions.
- Class attributes and methods:
  - Attributes in `Var`: `self.var`, `self.der`. To cover vector-to-vector cases, we implement our `self.var` and `self.der` as numpy arrays, in order to account for derivatives with respect to each input variable. Also the constructor checks whether the values and derivatives are integers, floats, or lists, and transforms them into numpy arrays automatically.
  - We have overloaded elementary mathematical operations such as addition, subtraction, multiplication, division, sine, pow, log, etc. that take in 1 `Var` type, or 2 Var types, or 1 Var and 1 constant, and return a new `Var` (i.e. the next "trace" variable). All other operations on constants will use the standard Python library. In each `Var`, we will store as attributes the value of the variable (which is calculated based on the current operation and previous trace variables) and the evaluated gradient of the variable with respect to each input variable.
  - Methods in `Var`:
    - `__init__`: initialize a `Var` class object, regardless of the user input, with values and derivatives stored as numpy arrays.
    - `__add__`: overload add function to handle addition of `Var` class objects and addition of Var and non-`Var` objects.
    - `__radd__`: preserve addition commutative property.
    - `__sub__`: overload subtraction function to handle subtraction of `Var` class objects and subtraction between Var and non-`Var` objects.
    - `__rsub__`: allow subtraction for $a -$ `Var` case where a is a float or an integer.
    - `__mul__`: overload multiplication function to handle multiplication of `Var` class objects and multiplication between Var and non-`Var` objects.
    - `__rmul__`: preserve multiplication commutative property.
    - `__truediv__`: overload division function to handle division of `Var` class objects over floats or integers.
    - `__rtruediv__`: allow division for $a \div$ `Var` case where a is a float or an integer.
    - `__neg__`: return negated `Var`.
    - `__abs__`: return the absolute value of `Var`.
    - `__eq__`: return `True` if two `Var` objects have the same value and derivative, `False` otherwise.
    - `__pow__`, `__rpow__`, `pow`: extend power functions to `Var` class objects.
    - `log`: extend logarithmic functions to `Var` class objects.
    - `exp`: extend exponential functions to `Var` class objects.
    - `sin`, `cos`, `tan`: extend trigonometric functions to `Var` class objects.
    - `arcsin`, `arccos`, `arctan`: extend inverse trigonometric functions to `Var` class objects.

    * `sinh`, `cosh`, `tanh`: extend hyperbolic functions to `Var` class objects.

- External dependencies:
  - `NumPy` - This provides an API for a large collection of high-level mathematical operations. In addition, it provides support for large, multi-dimensional arrays and matrices.
  - `doctest` - This module searches for pieces of text that look like interactive Python sessions (typically within the documentation of a function), and then executes those sessions to verify that they work exactly as shown.
  - `pytest` - This is an alternative, more Pythonic way of writing tests, making it easy to write small tests, yet scales to support complex functional testing. We plan to use this for a comprehensive test suite.
  - `setuptools` - This package allows us to create a package out of our project for easy distribution. See more information on packaging instructions here: `https://packaging.python.org/tutorials/packaging-projects/`.
  - Test suites: Travis CI, Coveralls

- Elementary functions
  - Our explanation of our elementary functions is included in the "Class attributes and methods" section above. For the elementary functions, we defined our own custom methods within the `Var` class so that we can calculate, for example, the $\sin(x)$ of a variable x using a package such as `numpy`, and also store the proper gradient $(\cos(x)dx)$ to propagate the gradients forward. For example, consider a scalar function where `self.val` contains the current evaluation trace and `self.der` is a numpy array of the derivative of the current trace with respect to the input. When we apply sin, we propagate as follows:

```
def sin(self):
    val = np.sin(self.val)
    der = np.cos(self.val) * self.der
    return Var(val, der)
```

    The structure of each elementary function is that it calculates the new value (based on the operation) and the new derivative, and then returns a new `Var` with the updated arguments.

6 Future

6.1 Possible software changes

Currently, the software can handle scalar-to-scalar functions. In the future, we will expand the module to handle vector-to-vector functions (and scalar-to-vector and vector-to-scalar), and also be able to trace the Jacobian at each step. In the present state of the project, we have not stored the derivative with respect to multiple input variables, since there is just one input variable in the scalar-to-scalar case.

6.2 Primary challenges

- Write a trace table that is growing along with running time of the module.
- The current structure cannot track partial derivatives with respect to different input variables, which we plan to do in the form of a numpy array. For example, if the function has two input variables with values a and b, the ideal set up is:

```
>>> x1 = Var(a)
>>> x1
```

```
Var(a, [1, 0])
>>> x2 = Var(b)
>>> x2
Var(b, [0, 1])
>>> x3 = x1 + x2
>>> x3
Var(a + b, [1, 1])
```

6.3 Additional features

This package will have two possible additional features (at least one of which we will implement):

- Write an application that uses *DeriveAlive* to implement optimization methods, like different forms of Newton's methods for optimization.
- Reverse mode, in which case we will store the Jacobian at each step.

6.4 Basic use case

We demonstrate a possible use case: Newton root finding for $y = x^2 - 1$ with *DeriveAlive*.

```
## Install at command line as in Section 4.4
pip install DeriveAlive
python
>>> import DeriveAlive.DeriveAlive as da
>>> import numpy as np

# Initial guess: root at 0.5
# Expect root at 1.0
>>> x0 = da.Var([0.5])
>>> f = x0 ** 2 - 1

# Newton root finding method
>>> error = 1
>>> while error > 0.000001:
      x1 = x0 - (f.val / f.der)
      error = da.abs(x1 - x0) / da.abs(x0)
      x0 = x1

# Expect root x = 1.0
>>> print (x1.val)
1.0
```

7 Sources

- CS 207 Lectures 9 and 10 (Autodifferentiation)
- Elementary functions: `https://en.wikipedia.org/wiki/Elementary_function`
- Package distribution: `https://packaging.python.org/tutorials/packaging-projects/`