Milestone 1

Michael Scott , Royce Yap, Yiwen Wang, Dimitris Vamvourellis

INTRODUCTION

Describe the problem the software solves and why it's important to solve that problem.

Derivatives are ubiquitous in many fields such as engineering design optimization, fluid dynamics and machine learning. There are in general three ways to calculate the derivatives: automatic differentiation, numeric differentiation, and symbolic differentiation. Automatic Differentiation (AD) brings a family of techniques that can calculate the partial derivatives of any function at any point efficiently and accurately. Unlike numeric differentiation, AD does not have the problem of floating point precision errors, since it calculates the derivative of a simple function, and keeps track of these derivatives, and there is no need of step sizes. Compared to symbolic differentiation, AD is not as memory intense, and can be much faster in terms of the calculation. Therefore, AD is an important way to calculate derivatives in practice.

The software that we design calculates the derivatives given the user's input using the forward mode of automatic differentiation, and provides the user with an easy way to solve their optimization problem using derivatives.

BACKGROUND

Describe (briefly) the mathematical background and concepts as you see fit.

At the core of Automatic Differentiation is the principle that functions implemented as computer code can be broken down into elementary functions, ranging from arithmetic operations (e.g. addition, subtraction etc.) and other functions (e.g. power, exponential, sin etc.). Hence, any differentiable function can be interpreted as a composition of different functions.

E.g., for a function, $f = sin^2(e^x)$, it can be rewritten as

$$f = \phi_1(\phi_2(\phi_3(x)))$$

where

$$\phi_1(z) = z^2, \phi_2(y) = sin(y) \text{ and } \phi_3(x) = e^x$$

In the forward mode, the chain rule can then be applied successively to each elementary component function to obtain the derivative of the function. Using the same example above, let c be a real number:

$$f'(c) = \phi_3'(\phi_2(\phi_1(c))) \cdot \phi_2'(\phi_1(c)) \cdot \phi_1'(c)$$

Based on the example above, the derivative, $f'(c)$, can be evaluated based on the following function-derivative pairs at each stage of computing the function[1] :

$$(\phi_1(c), \phi_1'(c))$$

$$(\phi_2(\phi_1(c)), (\phi_2'(\phi_1(c)) \cdot \phi_1'(c)))$$

$$(\phi_3(\phi_2(\phi_1(c))), \phi_3'(\phi_2(\phi_1(c)) \cdot \phi_2'(\phi_1(c)) \cdot \phi_1'(c))$$

[1] *Reference: Hoffmann, P.H.W.: A Hitchhiker's Guide to Automatic Differentiation*

At each stage of the function, the derivative of the function with respect to its argument is calculated by multiplying the derivative from the previous pair by the derivative of the function of the same pair. The exact values of the pairs are calculated at each stage, and is then used for the following pair. Working through the steps in the calculation, the derivative, $f'(c)$ is then the second entry of the final tuple.

Effectively, the forward mode computes the Jacobian-vector product, J_p. This decomposition can be represented via a computational graph structure of calculations, requiring initial values to be set for x_1, and x_1':

$$x_1 \xrightarrow{\phi_3(x)} x_2 \xrightarrow{\phi_2(x)} x_3 \xrightarrow{\phi_1(x)} y$$

While the above illustrates the forward mode of AD (the focus of our package), AD also has a reverse mode. Without using chain rule, it first does a forward pass to store the partial derivatives, before undertaking a reverse pass, which starts from the final function to be differentiated, y. After fixing the derivative of the final function, it then computes the derivative of each component function with respect to its parent function recursively (using chain rule) until the derivative of the function with respect to the basic-level argument (e.g. x_1) can be calculated.

HOW TO USE *AUTODIFF*?

How do you envision that a user will interact with your package? What should they import? How can they instantiate AD objects?

The user will be able to install the package in the standard way using `pip`. We will also provide a `.yml` file which will be used by the user to create the appropriate environment and ensure that dependencies like `numpy` have been installed. Then, to use the package, the user will only need to import our package which will implicitly import any other packages used by *AutoDiff*.

The user will be able to specify the variables of the function or functions that he/she wants to differentiate with respect to, by instantiating instances of class `ad_var`. Then, the user will be able to specify a function (scalar or vector functions) of these variables to differentiate which is an `ad_var` object itself. If the function is scalar, by calling `get_ders` on the given function, a numpy array which corresponds to the gradient vector evaluated at the given point will be returned. Respectively `get_val` will return the value of the function at the given point.

If the user provides a vector function, then `get_ders` will return the entire Jacobian matrix evaluated at the given point.

```
>>> import AutoDiff as ad
>>> x = ad.ad_var(1.0, 'x')
>>> y = ad.ad_var(3.0, 'y')
>>> z = ad.ad_var(2.0, 'z')
>>> f = x*(5+x)*y + z
>>> print(f.get_ders())
>>> [21.0, 6.0, 1.0]
>>> print(f.get_ders('x'))
>>> 21.0
>>> print(f.get_val())
>>> 20.0
```

SOFTWARE ORGANIZATION

Discuss how you plan on organizing your software package.

Directory Structure:

Our intended directory structure is as follows:

```
cs207FinalProject
├── README.md
├── tests
├── docs
│   └── milestone1.pdf
└── src
    └── AutoDiff.py
```

Modules:

The primary module will be a single `AutoDiff.py` file. Contained within will be the definition for an `AutoDiff.ad_var` class. Instances of this class, through interaction with other `ad_var` objects, will be able to compute the value of a function as well as the value of that function's derivative with respect to any input variable. At present, we envision that this module will be powerful enough to handle forward differentiation of any function comprised of the following "elementary" functions:

- Fundamental arithmetic operators (addition, subtraction, multiplication, and division).
- Logarithm (of any base).
- Negation.
- Exponentiation (e^x for `AutoDiff.ad_var` instance `x`).
- Power and root functions (x^n for some real n).
- Trigonometric functions ($\sin(x)$, $\cos(x)$, $\tan(x)$).
- Inverse trigonometric functions ($\arcsin(x)$, $\arccos(x)$, $\arctan(x)$).

Depending on our eventual choice for the "additional" feature of this project, or future design decisions, there may be additional modules added in the future that supplement or subdivide the functionality of `AutoDiff.py`.

Module Functionality:

Each instance of the `AutoDiff.ad_var` class represents the definition of a set of variables at a particular evaluation point. Through manipulations of these instances (either through fundamental arithmetic operations or built-in methods representing additional elementary functions described earlier), a user has the capability of representing any continuous differentiable function, be it scalar or vector. The associated function value and derivative(s) of any `ad_var` instance may be retrieved through the `get_val()` and `get_ders(wrt_var)` where `wrt_var` is an optional argument specifying the variable whose partial derivative the user desires (if the function is multivariate). It not specified, the entire contents of the derivatives attribute is returned. Note that the user does not have the ability to manually set function and derivative values outside of instance initialization, to prevent them from doing the following:

```
x = AutoDiff.ad_var(1.0, 'x')
x.val = 2.0 # AutoDiff.ad_var instance no longer reflects evaluation point
```

Instead, the function value and derivatives attributes will be made pseudoprivate (`_val` and `_ders` respectively).

Testing and Coverage:
Testing files will be contained within the test/directory of the package. Continuous integration and code coverage will be managed by TravisCI and CodeCov respectively.

Distribution and Packaging:
At present, we intend to distribute the package via PyPI. There will be no additional packaging framework included; we believe the scope of this project can be contained within a relatively simple directory structure with few functional python files and should not require additional overhead for users to install and use.

IMPLEMENTATION
Discuss how you plan on implementing the forward mode of automatic differentiation.

Core Data Structures

- `numpy` arrays: 1-D `numpy` arrays will be used to keep the gradient vectors as the entire trace is evaluated. Numpy provides vectorized operations which will make the overloading of elementary functions much more efficient for multivariate functions. If a vector function is provided, 2-D `numpy` arrays will be used to hold the Jacobian matrix.
- dictionaries: once the derivatives have been evaluated, the package will also provide the functionality to create a dictionary which will hold the derivative for each function with respect to each variable. In this way, the user will be able to access the derivatives in a user-friendly way by referencing the given names of the variables and functions that he/she previously instantiated instead of accessing them by standard slicing practices.

Class Implementation

- We will have a general `ad_var` class to represent the variables that are used in the Automatic Differentiation process. Each instance should be initialized with a scalar value of that variable to be evaluated on when calculating both the function and derivative values later, and a variable argument indicating which variable it is, e.g. `ad_var`(1,'x') will initialize an `ad_var` instance with variable name 'x' and value 1.
- We will also use the try-except method to catch unexpected input types, for example, if the user initializes the variable value of the `ad_var` instance with a value of type string, which is not a valid input type.
- As discussed in the Modules section, dunder methods such as "add" and "mul", and other elementary functions will be implemented under this class. More method information below in the *Methods for Autodiff class* section.

Name attributes

- `val`: 1-D array of floats, indicating the value of `ad_var` evaluated at the given point
- `ders`: 2-D array of floats, representing the Jacobian matrix of `ad_var` the vector function evaluated at the given point
- `val` and `ders` attributes will be made pseudoprivate to prevent users from manually setting function and derivative values outside of instance initialization

Methods for Autodiff class

1. `__init__()`:
   - Sets self.val as an array of 'val'
   - Sets self.ders as a 2D array of 'ders' attributes
2. `__eq__(self)`:
   - Returns True if self.val == other.val and self.ders == other.ders, returns False otherwise
3. `__str__(self)`:
   - Returns a string including information on self.val and self.ders of the given AutoDiff instance
4. `get_val(self)`:
   - Returns the value of the attribute self.val
5. `get_ders(self, wrt_var=None)`:
   - If `wrt_var = None`, returns the value of the attribute self.ders.
   - Otherwise, return the value of attribute `self.ders` with respect to `wrt_var` which is an optional argument.
6. `__add__(self, other)` and `__radd__(self, other)`:
   - Other can be a float, int or AutoDiff object
   - Returns a new AutoDiff instance when calculating self + other or other + self
7. `__sub__(self, other)` and `__rsub__(self, other)`:
   - Other can be a float, int or AutoDiff object
   - Returns a new AutoDiff instance when calculating self - other (sub) or other - self (rsub).
8. `__mul__(self, other)` and `__rmul__(self, other)`:
   - Other can be a float, int or AutoDiff object
   - Returns a new AutoDiff instance when calculating self * other or other * self
9. `__div__(self, other)` and `__rdiv__(self, other)`:
   - Other can be a float, int or AutoDiff object
   - Returns a new AutoDiff instance when calculating self / other or other / self
10. `__abs__(self)`:
    - Returns a new AutoDiff instance when calculating abs(self)
11. `__pow__(self, other)`:
    - Other can be a float, int or AutoDiff object
    - Returns a new AutoDiff instance when calculating self ** other
12. `exp(self)`:

- Returns a new AutoDiff instance with new.val = numpy.exp(self.val),
  new.ders = numpy.exp(self.val) * self.ders

13. `log(self, logbase = e)`:
    - Logbase can be a float or int
    - Returns a new AutoDiff instance with new.val = numpy.log(self.val) / numpy.log(logbase),
      new.ders = self.ders / self.val / numpy.log(logbase)
14. `sin(self)`:
    - Returns a new AutoDiff instance with new.val = numpy.sin(self.val),
      new.ders = numpy.cos(self.val)*self.ders
15. `cos(self)`:
    - Returns a new AutoDiff instance with new.val = numpy.cos(self.val),
      new.ders = -numpy.sin(self.val)*self.ders
16. `tan(self)`:
    - Returns a new AutoDiff instance with new.val = numpy.tan(self.val),
      new.ders = 1/(numpy.cos(self.val)**2)*self.ders
17. `arcsin(self)`:
    - Returns a new AutoDiff instance with new.val = numpy.arcsin(self.val),
      new.ders = self.ders/numpy.sqrt(1-self.val**2)
18. `arccos(self)`:
    - Returns a new AutoDiff instance with new.val = numpy.arccos(self.val),
      new.ders = -self.ders/numpy.sqrt(1-self.val**2)
19. `arctan(self)`:
    - Returns a new AutoDiff instance with new.val = numpy.arctan(self.val),
      new.ders = self.ders/(self.val**2 + 1)

External dependencies

- `numpy` for implementation of the elementary functions (e.g. sin, sqrt, log and exp), by overloading `numpy` implementations for these functions
- `doctest` and `pytest` for testing
- TravisCI and CodeCov used to manage continuous integration and code coverage