milestone1

Introduction

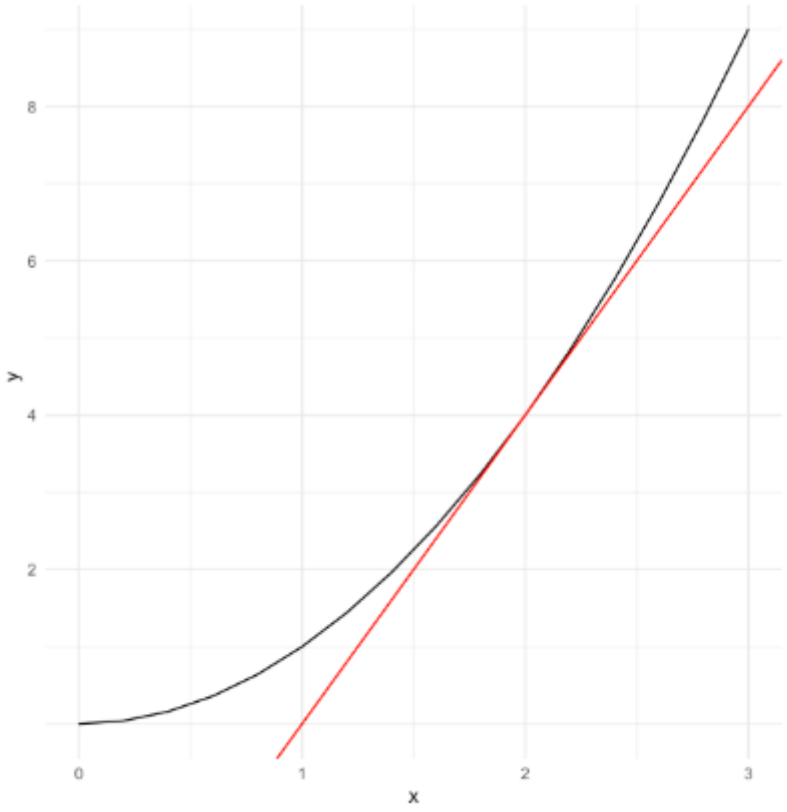
In its simplest form, we define the derivative of a function as its rate of change. We should familiarize ourselves with an example of a function, the notation for a derivative, and the graphical representation of rate of change.

Say we are incredibly lucky in our investments and our return, $f(x)$, is modeled by the function $f(x) = x^2$. Here, x can represent the dollar amount we invest. We would like to measure the rate at which our return, $f(x)$, changes with respect to a change in our investment dollar amount, x. The notation for such is represented mathematically as:

$$\frac{df}{dx} = lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x}$$

For example, we may model the relationship between investment (x) and returns ($f(x)$) using a function $f : \mathbb{R} \to \mathbb{R}$, by $f(x) = x^2$.

The derivative is modeled as $2 \cdot x$ and we can interpret it as the instantaneous rate of change - slope - as seen in the illustration below.



While the example above serves as a toy example to familiarize ourselves with the topic of differentiation and its graphical interpretation, the power of the derivative is not to be understated. Its origins date back to Isaac Newton and applications in physics and movement; however, it has since grown with applications in various different branches such as statistics, biology, finance, computer science, and many more fields.

There are three popular methods to calculate derivatives:

1. **Numerical**
2. **Symbolic**
3. **Automatic**

Numerical differentiation is the most basic and general introduction to calculating derivatives. In numerical differentiation we rely on the definition of the derivative, where we measure the amount of change in our function with a very small change in our input (x+h).

$$\frac{\delta f(x)}{\delta x} = lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

However, numerical differentiation can have issues with round off errors that lead to not achieving machine precision and can struggle with computational time when many dependent variables exist.

Symbolic differentiation attempts to manipulate formulas to create new formulas rather than performing numerical calculations. In doing so, we can in essence memorize derivatives of functions. However, symbolic interpretation is challenging to implement in computer programs and can be inefficient coding.

Automatic differentiation focuses on certain core elements: the chain rule, elementary functions and, to a lesser extent, dual numbers. The benefits of automatic differentiation are that it does not suffer from the same round off errors that numerical differentiation is susceptible to and does not suffer from the overly expensive, inefficient methods of symbolic differentiation. For these reasons, automatic differentiation is ubiquitous in tasks requiring quick differentiation, such as optimization in machine learning.

Background

Automatic differentiation builds off of two fundamental and relatively easy to understand concepts: elementary functions and the chain rule.

Elementary Functions

First, we can begin by providing an example of identifying elementary functions within a function. Consider the function:

$$f(x_1, x_2) = exp(sin(x_1^2 + x_2^2) + 2 * cos(\frac{x_1}{x_2}))$$

In the function above we can identify several functions that would be considered elementary functions: multiplication, division, sin(), cos(), exponentiation, powers. Automatic differentiation breaks about functions such as f(x) into the components of its elementary functions to act on intermediate steps in order to solve for its derivative. A more comprehensive list of elementary functions is included below:

Category	Elementary Functions
Arithmetic	multiplication, addition, subtraction, division
Powers and Roots	x^2, $y^{1/2}$
Trigonometric	sine, cosine, tangent, secant, cosecant, cotangent
Logarithmic	$\log(x)$
Exponential	$\exp(x)$

Chain Rule

Utilizing the above elementary functions, automatic differentiation applies the ever important chain rule to the elementary functions in order to solve the derivative of more complex functions. As a quick recap of the chain rule, let us define the following function:

$$f(x) = exp(4x)$$

We can replace $4x$ with $u(x)$. This will allow us to do the following differentiation to get our desired derivative of $f(x)$ with respect to x.

$$\frac{df}{dx} = \frac{df}{du} \cdot \frac{du}{dx} = exp(u) \cdot 4 = 4 \cdot exp(4x)$$

Computational Graph (Forward Mode)

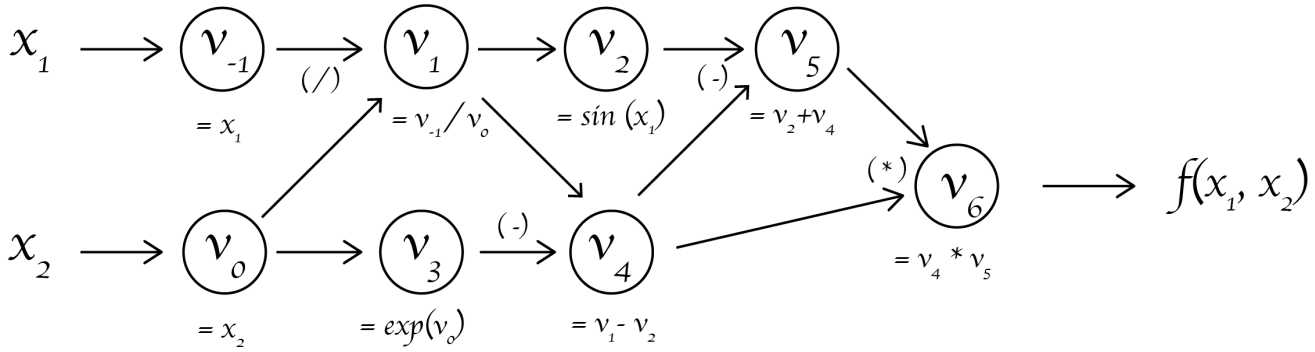
A computational graph allows us to see the ordered sequence of elementary functions, how we break down a more complex function from inside to outside (in forward mode), and how we can calculate intermediate steps to arrive at our final derivative result.

In the computational graph below, we can see that we begin with the inputs to the function, independent variables denoted by subscripts -1 and 0 (these generally take values < 1). Additionally, we build on these with intermediate variables from $v_0, v_1, \ldots$. The intermediate variables parallel the elementary functions applied at each step until we arrive at the full complex model from the inside out (again in forward mode). We can follow the computational graph's arrows to see how the elementary functions are applied until we reach our desired result of differentiation.

Let us examine the utility of a computational graph with a complex function such as:

$$f(x_1, x_2) = [sin(\frac{x_1}{x_2} + \frac{x_1}{x_2}) - e^{x_2}] \cdot [\frac{x1}{x2} - e^{x_2}]$$

We can see that elementary functions we will need are exp(), sin(), addition, subtraction, multiplication, and division. Additionally, we will need to create intermediate steps that build on the independent variables x_1 and x_2 in order to create all parts of the complex model. By following the arrows of the graph, we can see how we can begin at the independent variables and arrive back at the full complex function f(x).



Evaluation Trace (Forward Mode)

The evaluation trace allows us to utilize the components of our computational graph to aid us in solving the function value at a specific point and the partial derivatives with respect to each independent variable. The latter is done by utilizing seed vectors, p, which indicate the input variable to calculate the partial derivative of for the function. We will require one pass for each of the independent variables (x_1, x_2). In the first pass, shown below, we set the seed vector $p = [1\ 0]$.

Forward Primal Trace	Forward Tangent Trace ($p = [1\ 0]$)
$v_{-1} = x_1 = 1.5$	$D_p v_{-1} = 1$
$v_0 = x_2 = 0.5$	$D_p v_0 = 0$
$v_1 = \frac{v_{-1}}{v_0} = 3$	$D_p v_1 = \frac{(v_0 D_p v_{-1} - v_{-1} D_p v_0}{v_o^2} = 2$
$v_2 = sin(v_1) = 0.141$	$D_p v_2 = cos(v_1) \cdot D_p v_1 = -1.98$
$v_3 = exp(v_0) = 1.649$	$D_p v_3 = v_3 \cdot D_p v_0 = 0$
$v_4 = v_1 - v_3 = 1.351$	$D_p v_4 = D_p v_1 - D_p v_3 = 2$
$v_5 = v_2 + v_4 = 1.492$	$D_p v_5 = D_p v_2 + D_p v_4 = 0.02$
$v_6 = v_5 \cdot v_4 = 2.017$	$D_p v_5 \cdot v_4 - D_p v_4 \cdot v_5 = 3.012$

We point out that the left column gives us the result of our function $f(x_1 = 1.5, x_2 = 0.5)$. Meanwhile, the right column gives us the results of the partial derivative with respect to x_1. We would require another pass with $p = [0\ 1]$ in order to solve for the partial derivative with respect to x_2.

How to Use AutoDiff

The package will include a module for an `AutoDiff` class that utilizes the core data structure, the `DualNumber` objects. The user will interact with the `AutoDiff` module, without needing to interact with the `DualNumber` class. As such, user should import the `AutoDiff` module and the elementary functions for dual numbers. The user will initialize an `AutoDiff` object with a list of lambda functions representing a vector function $\mathbf{f}$. The user can then evaluate either a directional derivative, gradient, or Jacobian. and an associated `value` at which to evalauate.For example:

```
from autodiff import AutoDiff
from autodiff.utils import *

f1 = lambda x, y : x**2 + 2*y
f2 = lambda x, y : sin(x) + 3*y
f = [f1, f2]
ad = AutoDiff(f)
value = {"x": 2, "y" : 5}
jacobian = ad.get_jacobian(value) # [[4, 2], [cos(2), 3]]

# to get the partial derivatives w.r.t. x evaluated at given point (Both are
```

```
# to get the partial derivatives w.r.t. x evaluated at given point (both are t
derivative = ad.get_derivative(value, "x") # 4
derivative = ad.get_derivative(value, [1, 0]) # 4
# to get the directional derivative at the seed  vector p=[-2, 1]
derivative = ad.get_derivative(value, np.array([-2, 1])) # 2
# gradient vector evaluate at given point
gradient  = ad.get_gradient(value) # [4, 2]
```

Software Organization

- What will the directory structure look like?

  We plan to set up our package directory structure as the following:

```
team14/
    ├── src/
    │   └── autodiff/
    │       ├── __init__.py
    │       ├── auto_diff.py
    │       ├── utils/
    │       │   ├── __init__.py
    │       │   ├── dual_numbers.py
    │       │   ├── auto_diff_math.py
    │       │   └── helpers.py
    │   └── examples/
    │       ├── __init__.py
    │       ├── example_1.py
    │       └── ...
    ├── tests/
    ├── docs/
    │   └── milestone1
    ├── LICENSE
    └── README.md
```

- What modules do you plan on including? What is their basic functionality?
  - Modules for the AutoDiff package:
    - auto_diff.py: This module is the interface of the package. Users will initiate an AutoDiff object to carry out any necessary calculations.
    - dual_numbers.py: The DualNumber class is defined in this module. Although users do not need to directly interact with the DualNumber objects, the AutoDiff objects carry out function calculations and differentiation using DualNumber objects.

        - auto_diff_math.py: The overload functions for auto_diff
        - helpers.py: Any other utility functions that do not conceptually belong in the AutoDiff class or the DualNumber class.
    - Third-party modules:
        - NumPy: used for mathematical operations in automatic differentiation.
        - Math: for mathematical constants like π and e.
- Where will your test suite live?
    - As indicated above, the test suite will be in the `tests/` directory, separated from the source files.
- How will you distribute your package (e.g. PyPI with PEP517/518 or simply setuptools)?
    - PyPI with PEP517.
- Other considerations?
    - If the operations included in the `dual_numbers` module prove to be too extensive for a single file we will consider changing it into a directory and separating the dual number related operations in different modules

Implementation

Overview

The package will implement Automatic Differentiation by appropriately translating variables into **dual numbers**, and then evaluating expressions containing dual numbers using the built-in order of operations defined within Python, while potentially improving the efficiency by utilizing a compuational graph to store and reuse nodes for computed values. Crucially, when we perform (binary or unary) operations in evaluating these expressions, we will do so **using only** elemental operations which we explicitly define ourselves via "operation overloading" on `DualNumber`s (an object which we define), and which obey the characteristics of dual numbers. The resulting expression will itself be a dual number, the **real** part of which represents the evaluation of the function at the provided input, and the **dual** part of which represents the derivative of the functions evaluated at the provided inputs.

Classes

Class 1: `DualNumber`

- This class will be used inside the `AutoDiff` class; it is the foundation upon which our implementation is built.
- This class defines a `DualNumber` object which has two attributes `real` and `dual`
  - If not specified, the `dual` part of a `DualNumber` will default to 1
- We need to be able to perform elementary operations on `DualNumber`s in such a way that adheres to the behavior of dual numbers, as defined above.
- For example, for $z_1 = a_1 + b_1\epsilon$ and $z_2 = a_2 b_2 \epsilon$, we want that:
  - $z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)\epsilon$
  - $z_1 z_2 = (a_1 a_2) + (a_1 b_2 + b_1 a_2)\epsilon$
- In order to do this we will perform "operation overloading" on dunder methods, and define, for example:

```
class DualNumber:
    def __init__(self, real, dual=1):
        self.real=real
        self.dual=1

    def __add__(self, other):
        assert isinstance(other, (DualNumber, int, float)), "Invalid input"

        if isinstance(other, (int, float)):
            other = DualNumber(other, 0)

        return DualNumber(self.real+other.real, self.dual+other.dual)

    # the rest will follow a similar template as self.__add__()
    def __mul__(self, other):
        pass #TODO

    def __radd__(self, other):
        pass #TODO

    def __rmul__(self, other):
        pass #TODO

    def __pow__(self, other):
        pass #TODO

    # and more
```

- These methods will be carefully constructed to handle cases of, say, adding a DualNumber to a scalar (no matter the order in which they are passed).

Class 2: `AutoDiff`

- Users instantiate an `AutoDiff` object with one parameter `f`, which will then become an attribute of the object `self.f`.
  - `f` is either a *function* or *list of functions* ($f : \mathbb{R}^n \to \mathbb{R}^m$) over which to evaluate derivatives.
  - Example:
    - `f1 = lambda x : x**2; f2 = lambda x, y : x + y`
    - `ad = AutoDiff([f1, f2])`
  - Upon initialization the function will also check for valid input. For example, it will check the input represents valid mathematical functions.
- The AutoDiff class will have an instance method call `get_jacobian` which takes an argument `value`, representing the point for evaluating the Jacobian matrix. The method performs forward mode AD by default (with the possibility of performing AD with reverse mode, provided the team decides to implement a reverse mode). This implementation allows automatic differentiation of functions of $\mathbb{R}^m \to \mathbb{R}^n$.
  - `value` is a *dictionary* (`str` : `float`), representing the value(s) at which the user seeks to evaluate the derivative.
  - The method checks valid correspondence in variable names between functions and value names.
  - For each partial derivative, $\frac{\partial f_1}{\partial x}$, x_i will be converted into a DualNumber object `(x_i, 1)` while other variables $x_j, j \neq i$ will be converted into `DualNumber` objects `(x_j, 0)`, such that the differentiation will be done with respect to x_i.
- The class will also have an instance method called `get_derivative` which takes a point `value` and either an explicit reference to a varible to differentiate with respect to (e.g., `"x"`) or a seed vector `p` and return the specified directional derivative of `self.f` at the point `value`.
  - `get_derivative` will operate on the functions `self` and return the specified derivative
- It will also have class methods called `get_gradient` which similarly operates on `self.f` and returns a gradient vector, provided these these methods are compatible with the AutoDiff instance attributes provided.
- Each of these functions will compute partial derivatives by evaluating expression involving dual number using elementary operations which we explicitly define on the `DualNumbers` class (discussed below).
- Below is skeleton code for the `AutoDiff` class which relies upon the DualNumber objects discussed above.

```
from utils import *

class AutoDiff:
```

```
    def __init__(self,f):
        self._func_check(f)
        self.f = f

        # store to improve efficiency for repeated function calls
        self.value = None
        self.seed = None
        self.derivative = None
        self.jacobian = None
        self.gradient = None

    def _func_check(self):
        '''check that vector function passed are valid math operations'''
        pass #TODO

    def _var_check(self):
        '''check that the variables correspond to those in the functions'''
        pass #TODO

    def get_derivative(self, value, df_wrt, seed_vector = None):
        if seed_vector !=None:
          # TODO: Calculate derivative using seed_vector
        else:
          # Calculate derivative w.r.t explicitly specified variable (Sketch)
          dual_var = { df_wrt : DualNumber(self.value[df_wrt]) }
          other_vars = {(k, self.value[k]) for k != df_wrt)}
          kwargs = {**dual_var, **other_vars}
          derivative = self.f(**kwargs).dual
          return derivative

    def get_gradient(self):
      pass #TODO

    def get_jacobian(self, value):
        if self.value == value and self.jacobian is not None:
            return self.jacobian

        self._var_check()
        jacobian = np.array()
        for i in len(value):
            for j in len(self.f):
                x = DualNumber(value[i], 1)
                # the i,j entry of the Jacobian matrix is the partial derivati
                jacobian[i,j] = self.get_derivative(value, np.put(np.zeros(len

        return jacobian
```

Module: `auto_diff_math.py`

- The `DualNumber` objects are useful for implementing the automatic differentiation algorithm. However, conceptually they do not only serve the purpose of carrying out automatic differentiation. Furthermore, we also envision that the `AutoDiff` objects only perform differentiation-related operations. As a result, we include another module the overloading functions that perform mathematical operations on `DualNumber` objects.
- We include these functions in a module which we import for use in the `AutoDiff` class defined above.
- These functions each follow the same structure: for a `DualNumber`, `a = DualNumber(real, dual)`, and a function `func`, if we pass `func(a)`, we will return another `DualNumber`, say `DualNumber(new_real, new_dual)` such that:
  - `new_real` is `func` applied to `real`
  - `new_dual` is the derivative of `func` applied to `real` *times* `dual`
- These functions will ultimately need to gracefully handle non-Dual numbers, by, for example, falling back to the standard implementation (e.g., `np.sin`) when passed a real number.
- By explicitly defining elemental operations in this way, we ensure that when evaluating expressions containing dual numbers, python will resolve to a final expression which is itself a dual number whose dual part represents the derivative of interest
- Moreover, to improve efficiency of forward mode, we may overload these functions in ways such that when they are called, they also add new nodes to a graph objects that stores the elementary operations carried out for an `AutoDiff` object.
- Here are some such functions.

```
import numpy as np
import math

def sin(DualNumber: x):
    return DualNumber(np.sin(x.real), np.cos(x.real)*x.dual)

def cos(DualNumber: x):
    return DualNumber(np.cos(x.real), -np.sin(x.real)*x.dual)

# ... and more
```

Other Comments

- For a naive implementation of forward mode, we will not need a graph class to resemble the computational graph since, as discussed above, we can avoid storing this information by simply casting certain variables to dual numbers and using python's built in "order of operations" to evaluate these expressions in the ways we define. However, we will still consider implementing and using a graph class to store nodes of already carried-out calculations to avoid repeating the same calculation and improve the efficiency of the algorithim. Furthermore, the graph class will be necessary if we proceed to implement reverse mode at a later stage in the project.

Licensing

We will use the *MIT License*, since we want our library to be open for anyone to modify it. This library will not provide all uses or versions of automatic differentiation available or the ones being still developed. Due to the limited timeframe of the project we only aim to provide the backbones for automatic differentiation and some of the basic algorithms in automatic differentiation. Thus we want anyone that wants to make changes to the library to fill their specific needs to be able to do so, either in a commercial setting or not.