F. Wermelinger
Office: Pierce 211

Pair-Programming 7

Forward mode in higher dimensions, Jacobian, Implementation

Issued:	October 14, 2022
Due:	October 28, 2022 11:59pm

In this pair-programming session you will continue work on forward mode AD in higher dimensions and consider implementation approaches.

You should work on the exercises in groups of 3 to 4 students via a `tmate` session. Your team members can submit the same file. Please indicate your names in a header in the files. See the tutorials on the class website for an example pair-programming workflow.[1] Do not forget to commit and push your work when you are done. Ensure that you are on your *default branch* for this and not, possibly, on your homework branch.

Exercise 1: Forward Mode, Jacobian and Seed Vectors

Deliverables:

1. You can submit this exercise in different forms. Either an image file `exercise_1.png` of your handwritten work, write it up in a LaTeX document and submit `exercise_1.pdf` or write a Markdown formatted Jupyter notebook `exercise_1.ipynb`.

Consider the vector function from the last lab

$$f(x_1, x_2) = \begin{bmatrix} x_1^2 + x_2^2 \\ e^{x_1+x_2} \end{bmatrix}. \tag{1}$$

The function in this example is a mapping $f : \mathbb{R}^2 \mapsto \mathbb{R}^2$. That is, it takes a two-dimensional input ($x \in \mathbb{R}^2$) and its output is also two-dimensional ($f(x) \in \mathbb{R}^2$). The Jacobian J of $f(x)$ is a matrix with all the first-order partial derivatives of $f(x)$. We have seen in the lecture that this corresponds to taking the gradient $J(x) = \nabla f(x)$ with individual elements given by $J_{ij} = \frac{\partial f_i}{\partial x_j}$.

a) Compute the Jacobian *analytically* and evaluate it at the point $x = [1, 1]^\intercal$.

[1] https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-pp

b) In the lecture a *directional derivative* has been introduced which is a generalized derivative that allows to compute the derivative in an *arbitrary* direction implied by the seed vector p in the space spanned by the input coordinates x. In forward mode automatic differentiation the result in the forward pass is therefore not the Jacobian directly but the *projection* of J in direction p (the seed vector), where p is a *parameter* that is chosen depending on the problem at hand. This projection is given by the *inner product* (dot product) Jp, or in index notation $J_{ij}p_j$ where summation is applied over index j. To compute the *full* Jacobian with forward mode AD you therefore need as many passes as there are input coordinates x. As you have seen in the lecture, all partial derivatives in coordinate direction x_1 are obtained by setting the seed vector to $p = [1, 0]^\intercal$ and similarly in direction x_2 with $p = [0, 1]^\intercal$. These are special cases since they only take into account one *principal coordinate*. In general, you can compute *any linear combination* that involves all of the bases.

Compute the analytical form for the inner product Jp and evaluate it at the point $x = [1, 1]^\intercal$ using the directions given below.

i) $p = [1, 1]^\intercal$

ii) $p = [1, -2]^\intercal$

Exercise 2: Forward Mode Implementation in Python

Deliverables:

1. `exercise_2.py`

In this task we are going to consider two implementation approaches for forward mode AD using the following test function

$$f(x_1, x_2) = x_1^2 + x_2^2 \tag{2}$$

Evaluate the function and its directional derivative at the point $x = [1, 1]^\intercal$ given the following seed vectors

$$p = [1, 0]^\intercal, \qquad p = [0, 1]^\intercal, \qquad p = [1, 1]^\intercal.$$

The goal of this exercise is for you to start thinking about how you would go about implementing some elemental functions required for your final project.

a) Implement the forward mode computation using a Python *decorator* that can be used to wrap a function `f`. The decorator should be called `@derivative(Dpf)` and takes one argument `Dpf` which is a function object that corresponds to the directional derivative of the wrapped function `f`. The closure returned by the decorator should take two arguments, one for the point of evaluation x and another for the seed vector p. Note that the function object `Dpf` *takes the same arguments as the closure*. The closure should return the function value and the value of the directional derivative in a list where the first element is the function value and the second element corresponds to the value of the directional derivative. Does the use of such a decorator correspond to an *automatic differentiation* algorithm?

Note that the decorator you are implementing here is an advanced type of decorator that allows for arguments. Such a decorator can be implemented as a *functor* which is a class that defines the `__call__` dunder method. This special method will be the *outer function* call when the decorator executes and it should return the *closure*. You could start with something like this:

```python
class derivative:
    """Functor for function decoration adding derivatives
    """

    def __init__(self, Dpf=None):
        # save state obtained from @decorator(arg) argument
        self.Dpf = Dpf

    def __call__(self, f):
        """Outer function of closure takes wrapped function f as an
        argument.
        """
        pass
```

b) Implement the forward mode computation using a simple dual number class with supported operator overloading for the `__mul__`, `__add__` special methods to perform multiplication and addition with dual numbers. Use your dual number class to evaluate the function in equation 2. You will need two dual numbers, say z_1 and z_2, for this and the function should return a new dual number. Compare the real and dual part of the returned dual number with your results from task 2a. Did you ever compute the derivative *manually*?