Automatic differentiation with dynamic typing

Romain Brette
Ecole Normale Supérieure, Paris

I describe the implementation of automatic differentiation with dynamic typing in the Python language.

Categories and Subject Descriptors: ... [...]: ...

General Terms: Algorithms

Additional Key Words and Phrases: Automatic differentiation, Python, dynamic typing

1. INTRODUCTION

The simplest implementation of automatic differentiation for object-oriented languages is forward accumulation by operator overloading. It consists in augmenting standard operations (additions, multiplications, etc.) by the corresponding operations on the derivative. One defines a class of objects holding both the value of the function and its derivative, and defines operations as class methods. For example, additions and multiplications of two such objects are defined as $(x, x') + (y, y') = (x + y, y + y')$ and $(x, x') * (y, y') = (x * y, x' * y + x * y')$, a constant x_0 is defined as $(x_0, 0)$ and the identity function $x \mapsto x$ evaluated at x_0 is defined as $(x_0, 1)$. Thus, the derivative of $x \mapsto 2 * (x + 3)$ at x_0 can be calculated by forward accumulation by the expression $2 * ((x_0, 1) + 3)$, which evaluates to $(2 * x_0 + 3, 2)$. In this article, I show how dynamic typing makes this technique simple and flexible, and I describe such an implementation of automatic differentiation in Python. The Python module is released under the terms of the BSD licence.

In dynamically typed languages such as Python, the type of arguments is not specified in the definition of functions. Instead, it is determined at evaluation time. For example, the following function:

```
f = lambda x,y:x+y
```

can be evaluated with any arguments that have an addition method. Thus, to implement forward differentiation in a dynamically typed language, it is sufficient to define a class that implements all standard operations, without specifying the type of the values, and to evaluate the given function with an object of that class. In the Python implementation I will describe in this article, the derivative of $x \mapsto 3x^2$ at $x = 2$ is simply calculated as follows:

```
> differentiate(lambda x:3*x*x+2,x=2)
12
```

The function `differentiate` evaluates the function passed as an argument at the point `Differentiable(2,1)`, which is an object of class `Differentiable` holding the value $x = 2$ and the derivative $x' = 1$. Thanks to dynamic typing, the same function can be applied to complex values:

```
> differentiate(lambda x:3*x*x+2,x=1+2j)
(6+12j)
```

and, in fact, to any object which defines the necessary operations. For example, we can apply it to values with physical units (using e.g. the Pyquant module):

```
> differentiate(lambda x:x*x,x=2*meter)
4.0 m
```

and the result has the correct units. Dynamic typing for automatic differentiation becomes more interesting when one realizes that `Differentiable` objects can also hold values which are themselves `Differentiable` objects. It follows that differentiation operations can be nested, giving higher order derivatives:

```
> differentiate(lambda x:differentiate(lambda x:3*x*x+2,x=x),x=2)
6
```

Here the first `differentiate` function defines the object `Differentiable(2,1)`, and the second (nested) one defines the object `Differentiable(Differentiable(2,1),1)`. To simplify the expressions, I introduced the optional argument `order`:

```
> differentiate(lambda x:3*x*x+2,x=2,order=2)
6
```

Partial derivatives can be obtained in the same way with nested `Differentiable` objects, for example:

```
> differentiate(lambda x,y:7*x*y+2*y+1,x=(1,2),order=(1,1))
7
```

where $\partial^2 f/\partial x \partial y$ is calculated. In section 0**??**, I describe a more efficient implementation of partial derivatives. In section 0**??**, I show how to use these ideas to implement differential operators such as gradients and Hessians in an efficient way.

2. OPERATOR OVERLOADING AND FIRST ORDER DERIVATIVES

In order to calculate multiple partial derivatives and gradients in a single pass, a `Differentiable` object holds a value and a dictionary of values of partial derivatives. A constant is initialized with an empty dictionary and the variable x is initialized with dictionary `{x:1.0}`. That dictionary is a sparse implementation of the gradient. All operations are then defined as expected. I briefly describe three examples: addition, multiplication and cosine.

When two `Differentiable` objects are added, a new object is created whose dictionary is the sum of the dictionaries of those objects, seen as sparse vectors. When two `Differentiable` objects x and y are multiplied, a new object is created whose

values in the dictionary for key k is `x.val*y.diff['k']+x.diff['k']*y.val`. To apply a unary function, we just use the chain rule. For example, for the cosine function, the values in the dictionary is `-sin(x.val)*x.diff['k']`.

In this way, one can calculate a gradient in just one pass of the function, by calling the function with `Differentiable` objects as arguments.

3. HIGHER ORDER DERIVATIVES

Higher order derivatives of single variable functions can be readily calculated by recursion, as mentioned in the introduction. For higher order partial derivatives, we call the function with higher order `Differentiable` objects, which are nested `Differentiable` objects with n levels of recursion, where n is the total order of the partial derivative. Operations on those objects calculate all partial derivatives of order n. Then one just need to extract the relevant one from the result.

To calculate the Hessian of a function, one calls the function with second order `Differentiable` objects. Then if the result is `y`, the partial derivative with respect to variables i and j is `y.diff[i].diff[j]`. The same trick can be used to calculate Taylor series.

REFERENCES

...

Received Month Year; revised Month Year; accepted Month Year