Journal of Computational Science 4 (2013) 334–344

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Algorithmic differentiation in Python with AlgoPy

Sebastian F. Walter*, Lutz Lehmann

Institut für Mathematik, Fakultät Math. Nat. II, Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany

ARTICLE INFO

Article history:
Received 3 February 2011
Received in revised form
14 September 2011
Accepted 3 October 2011
Available online 17 November 2011

Keywords:
Automatic differentiation
Cholesky decomposition
Hierarchical approach
Higher-order derivatives
Numerical linear algebra
NumPy
Taylor arithmetic

ABSTRACT

Many programs for scientific computing in Python are based on NumPy and therefore make heavy use of numerical linear algebra (NLA) functions, vectorized operations, slicing and broadcasting. AlgoPy provides the means to compute derivatives of arbitrary order and Taylor approximations of such programs. The approach is based on a combination of univariate Taylor polynomial arithmetic and matrix calculus in the (combined) forward/reverse mode of Algorithmic Differentiation (AD). In contrast to existing AD tools, vectorized operations and NLA functions are not considered to be a sequence of scalar elementary functions. Instead, dedicated algorithms for the matrix product, matrix inverse and the Cholesky, QR, and symmetric eigenvalue decomposition are implemented in AlgoPy. We discuss the reasons for this alternative approach and explain the underlying idea. Examples illustrate how AlgoPy can be used from a user's point of view.

1. Introduction

In scientific computing, mathematical functions are described by computer programs. *Algorithmic* (aka *Automatic*) *Differentiation* (AD) can be used to obtain polynomial approximations and derivative tensors of such functions in an efficient and numerically stable way. It is also suitable for programs with thousands of lines of code and is not to be confused with symbolic or numerical differentiation. The website http://www.autodiff.org provides an overview of current and past research. See also the standard references [16,15,3]. The most important features of AD are:

1. The computed derivatives have a finite-precision error on par with the nominal function evaluation.
2. The number of operations $\mathrm{OPS}(\{f, \nabla f\})$ to evaluate both the function $f : \mathbb{R}^N \to \mathbb{R}$ and its gradient ∇f is less than $\omega \cdot \mathrm{OPS}(f)$, where $\omega \in [3, 4]$. The gradient is thus at most four times more expensive than the function itself, no matter what N is [16]. Please note that this is a theoretical result: the actually observed ratio on a computer is generally worse.
3. It is possible to write programs in such a way that an AD tool can be applied with no or only small changes to the code.

Python is a popular programming language for scientific computing [34,33]. It has a clear syntax, a large standard library and there exist many packages useful for scientific computing. The de facto standard for array and matrix manipulations is provided by the package NumPy [26] and thus many scientific programs in Python make use of it. In consequence, concepts such as broadcasting, slicing, element-wise operations (ufuncs) and numerical linear algebra functions (NLA) are used on a regular basis.

The tool AlgoPy provides the possibility to compute high-order univariate Taylor polynomial approximations and derivatives (gradient and Hessian) of such programs. It is implemented in pure Python and has only NumPy and SciPy [20] as dependencies. Official releases can be obtained from [36].

The purpose of this paper is to serve as a reference for AlgoPy and to popularize its unique ideas. The target audience are potential users of AlgoPy as well as developers of other AD tools.

- We briefly discuss the forward and reverse mode of AD in Section 2. It is not the goal to provide a tutorial, but to explain how the theory is related to the implementation. We deem this discussion important for two reasons:
  1 It equips the user with the necessary know-how to extend the functionality of AlgoPy.
  2 To debug code it is necessary to understand what is happening behind the scenes.
- AlgoPy offers dedicated support for several numerical linear algebra functions such as the Cholesky, QR and real symmetric eigenvalue decomposition. This is, to the authors' best

* Corresponding author.
E-mail addresses: sebastian.walter@gmail.com (S.F. Walter), llehmann@mathematik.hu-berlin.de (L. Lehmann).

1877-7503/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.jocs.2011.10.007

Table 1
The three-part form of the function $f(x) = \sin(x_1 + \cos(x_2)x_1)$.

Independent	v_{-1}	=	x_1	=	3
Independent	v_0	=	x_2	=	7
	v_1	=	$\phi_1(v_0)$	=	$\cos(v_0)$
	v_2	=	$\phi_2(v_1, v_{-1})$	=	$v_1 v_{-1}$
	v_3	=	$\phi_3(v_{-1}, v_2)$	=	$v_{-1} + v_2$
	v_4	=	$\phi_4(v_3)$	=	$\sin(v_3)$
Dependent	y	=	v_4		

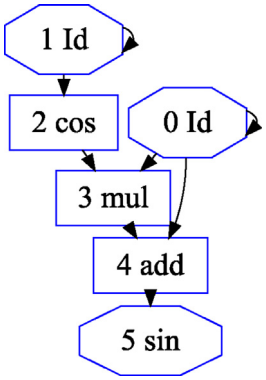

Fig. 1. This plot shows the computational graph of the function $f(x) = \sin(x_1 + \cos(x_2)x_1)$. The independent and dependent variables are depicted as octagons whereas intermediate variables are represented as rectangles.

knowledge, a unique feature of AlgoPy. In Section 3 we give a concise description of the approach.
- Finally, we compare in Section 4 the runtime of several related tools on some simple test examples to allow a potential user to decide whether the current state of AlgoPy is efficient enough for the task at hand.

2. Relating theory and implementation

The purpose of this section is to summarize the forward and reverse mode of AD and describe how these two concepts are implemented in AlgoPy.

2.1. Computational model

Let $F : \mathbb{R}^N \to \mathbb{R}^M$, $x \mapsto y = F(x)$ be the function of interest. We require that it can be written as a finite sequence of differentiable instructions. Traditionally, the instructions include $\pm, \times, \div, \sqrt{\cdot}$, $\exp(\cdot)$ and the (hyperbolic) trigonometric functions as well as their inverse functions. We refer to them as *scalar elementary functions* to distinguish them from vector and matrix operations as they will be discussed in Section 3. The quantity x is called the *independent variable* and y is the *dependent variable*. In the important special case $M = 1$ we use f instead of F.

As an example, consider the function $f : \mathbb{R}^2 \to \mathbb{R}$, $x \mapsto y = f(x) = \sin(x_1 + \cos(x_2)x_1)$. The sequence of operations to evaluate $f(3, 7)$ is shown in Table 1. This representation is called *three-part form*. Each intermediate variable is computed by an elementary function call $v_l = \phi_l(v_{i \prec l})$, where $v_{i \prec l}$ is the tuple of input arguments of ϕ_l. For a more detailed discussion of the computational model and its relation to algorithmic differentiation see Griewank [14]. Alternatively one can represent the computational sequence also as computational graph. This is shown in Fig. 1.

2.2. Forward mode

We follow the approach of ADOL-C [16] and use univariate Taylor polynomial (UTP) arithmetic to evaluate derivatives in the forward mode of AD. One can find a comprehensive introduction of the approach in [25].

Let $\epsilon > 0$ and $F : \mathbb{R}^N \to \mathbb{R}^M$ be sufficiently smooth. Define the smooth curve $y(t) = F(x(t))$ for a given smooth curve $x : (-\epsilon, \epsilon) \in \mathbb{R}^N$. One can compute the first directional derivative of F by setting $x(t) = x_{[0]} + x_{[1]}t$ and computing the first-order Taylor approximation

$$\begin{aligned} y_{[0]} + y_{[1]}t &= F(x_{[0]} + x_{[1]}t) + \mathcal{O}(t^2) \\ &= F(x_{[0]}) + \frac{\mathrm{d}F}{\mathrm{d}t}(x(t))\Big|_{t=0} t \ . \end{aligned}$$

The use of the chain rule yields the directional derivative

$$\frac{\mathrm{d}F}{\mathrm{d}t}(x(t))\Big|_{t=0} = \frac{\partial F}{\partial x}(x_{[0]}) \cdot x_{[1]} \ .$$

E.g., by choosing $x_{[1]}$ to be the ith Cartesian basis vector e_i one obtains $\frac{\partial F}{\partial x_i}(x)\Big|_{x=x_{[0]}}$ as result.

The important point to notice is that the desired directional derivative does not depend on t. To generalize the idea to higher-order derivatives, one extends functions $F : \mathbb{R}^N \to \mathbb{R}^M$, $x \mapsto y = F(x)$, to functions $E_D(F) : \mathbb{R}^N[T]/(T^D) \to \mathbb{R}^M[T]/(T^D)$, $[y]_D = E_D(F)([x]_D)$. We denote representatives of the polynomial factor ring $\mathbb{R}^N[T]/(T^D)$ as

$$[x]_D := [x_{[1]}, \ldots, x_{[D-1]}] := \sum_{d=0}^{D-1} x_{[d]} T^d \ , \tag{1}$$

where $x_{[d]} \in \mathbb{R}^N$ is called a *Taylor coefficient*. The quantity T is an indeterminate, i.e., a formal variable. It plays a similar role for the polynomials as $i := \sqrt{-1}$ for the complex numbers. We make a distinction between t and T to stress that t is regarded as real variable whereas T is an indeterminate. The *extended function* $E_D(F)$ is defined by its action

$$[y]_D = \sum_{d=0}^{D-1} y_{[d]} T^d = E_D(F)([x]_D) \tag{2}$$

$$= \sum_{d=0}^{D-1} \frac{1}{d!} \frac{\mathrm{d}^d}{\mathrm{d}t^d} F\left(\sum_{d=0}^{D-1} x_{[d]} t^d \right) \Bigg|_{t=0} T^d \ . \tag{2}$$

The fundamental result of the forward mode is that the operator E_D is a function composition preserving homomorphism. Explicitly, for any sufficiently differentiable composite function $F(x) = (H \circ G)(x) = H(G(x))$ it holds that

$$E_D(H \circ G) = E_D(H) \circ E_D(G) \ . \tag{3}$$

Since any function satisfying the assumptions from Section 2.1 can be written as such a composite function (c.f. [14]), it completely suffices to provide algorithms for $[y]_D = E_D(\phi)([x]_D)$, where

$$\phi \in \{\pm, \times, \div, \sin, \exp, \ldots\} \ .$$

Tables 2 and 3 show an incomplete list of the most important algorithms [16,24]. They are shown here to stress their similarity to the algorithms shown in Section 3.