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		

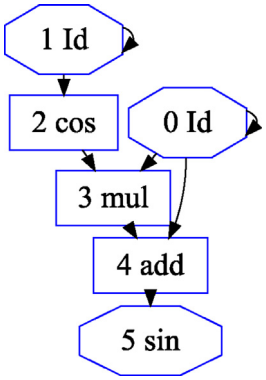

Fig. 1. This plot shows the computational graph of the function $f(x) = \sin(x_1 + \cos(x_2)x_1)$. The independent and dependent variables are depicted as octagons whereas intermediate variables are represented as rectangles.

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.

Table 2
UTP algorithms of binary scalar elementary functions.

$z=\phi(x,y)$	$d=0,\ldots,D$
$x+cy$	$z_{[d]}=x_{[d]}+cy_{[d]}$
$x\times y$	$z_{[d]}=\sum_{k=0}^{d}x_{[k]}y_{[d-k]}$
x/y	$z_{[d]}=\frac{1}{y_{[0]}}\left[x_{[d]}-\sum_{k=0}^{d-1}z_{[k]}y_{[d-k]}\right]$

Table 3
UTP algorithms for unary scalar elementary functions. We use $\tilde{x}_{[k]}:=kx_{[k]}$. The zeroth coefficient is computed as $y_{[0]}=\phi(x_{[0]})$.

$y=\phi(x)$	$d=1,\ldots,D$
$\ln(x)$	$\tilde{y}_{[d]}=\frac{1}{x_{[0]}}\left[\tilde{x}_{[d]}-\sum_{k=1}^{d-1}x_{[d-k]}\tilde{y}_{[k]}\right]$
$\exp(x)$	$\tilde{y}_{[d]}=\sum_{k=1}^{d}y_{[d-k]}\tilde{x}_{[k]}$
$\sqrt{x}$	$y_{[d]}=\frac{1}{2y_{[0]}}\left[x_{[d]}-\sum_{k=1}^{d-1}y_{[k]}y_{[d-k]}\right]$

```
from algopy import zeros, sin, cos, UTPM
D,P = 4,1
x = UTPM(zeros((D,P)))
x.data[1] = 1
y1 = sin(x)
y2 = sin(cos(1/(1+x)))
print(y1.data[:,0])
print(y2.data[:,0])
```

Listing 1. This example shows how AlgoPy can be used for UTP arithmetic.

```
[ 0.          1.          0.         -0.16666667]
[ 0.51439526  0.72160615 -1.13538994  1.46068226]
```

Listing 2. Output of Listing 1.

2.3. Forward mode in AlgoPy

In AlgoPy one can compute on general UTP operands

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

where $x_{[d]}$ is a tensor of arbitrary shape, e.g. a 3-tensor $x_{[d]}\in\mathbb{R}^{M\times N\times K}$. We call tuples of the form (M,N,K) the *shape* of the UTP $[x]_D$. The functionality is wrapped in the class `algopy.UTPM`. The Taylor coefficients are stored in the attribute `UTPM.data` as numpy ndarray. One could use an ndarray with a shape of the form (D,M,N,K). In other words, the first dimension is the number of coefficients, followed by the shape of the UTP. However, since it is often necessary to compute the same function with different inputs, we decided to use a data structure with a shape of the form (D,P,M,N,K) for P simultaneous evaluations. The class `algopy.UTPM` overloads the typical operators like $\pm$, $\times$, $\div$ as well as the methods `__getitem__` and `__setitem__`. It is also possible to create scalar UTPs, i.e., the numpy ndarray `UTPM.data` can have the shape (D,P).

To show an illustrative example from calculus, we demonstrate how AlgoPy can compute the series expansion of $y_1(t)=\sin(t)$ and $y_2(t)=\sin\left(\cos\left(\frac{1}{1+t}\right)\right)$. The code is shown in Listing 1.

One can check that the output is indeed correct (Listing 2).

```
from numpy import sin, cos

x1,x2 = 3.,7.

# Step 1: evaluation of the function
vm1 = x1                                  # independent
v0  = x2                                  # independent
# ------------------------------------------------------
v1 = cos(v0)                              # instruction 1
v2 = v1 * vm1                             # instruction 2
v3 = vm1 + v2                             # instruction 3
v4 = sin(v3)                              # instruction 4
# ------------------------------------------------------
y  = v4                                   # dependent

# Step 2: reverse mode
v4bar,v3bar,v2bar,v1bar,v0bar,vm1bar=1,0,0,0,0,0

v3bar += v4bar*cos(v3)                    # instruction 4
vm1bar+=v3bar; v2bar+=v3bar               # instruction 3
v1bar+=v2bar*vm1; vm1bar+=v2bar*v1        # instruction 2
v0bar-=v1bar*sin(v0)                      # instruction 1

print 'gradient =', [vm1bar, v0bar]
```

Listing 3. At first the function is evaluated by four operations, and then the gradient is evaluated in the reverse mode.

2.4. Reverse mode

Let $F:\mathbb{R}^N\to\mathbb{R}^M$, $x\mapsto y=F(x)$ be sufficiently smooth and define

$$\dot{y}=\frac{\partial F}{\partial x}\dot{x}\ . \tag{4}$$

The basic idea of the reverse mode is to regard the linear form

$$\bar{y}^T\dot{y}=\sum_{m=1}^{M}\bar{y}_m\dot{y}_m\ , \tag{5}$$

and then apply a pullback. A *pullback* is the action of going back one level of the functional dependence which yields, according to the rules of differential calculus, the relation

$$\bar{y}^T\dot{y}=\bar{y}^T\left.\frac{\partial F(z)}{\partial z}\right|_{z=x}\dot{x}=\bar{x}^T\dot{x}\ , \tag{6}$$

where $\dot{y}\in\mathbb{R}^M$, $\dot{x}\in\mathbb{R}^N$ and $\bar{x}^T=\bar{y}^T\left.\frac{\partial F(z)}{\partial z}\right|_{z=x}$. Since we require the function F to be a sequence of smooth elementary functions, one may apply the chain rule. In consequence it is just necessary to provide one pullback algorithm for each elementary function.

Example We consider again the example from Table 1 and apply the reverse mode by hand. The following calculation shows the successive pullbacks of the linear forms:

$$\bar{y}\dot{y}=\bar{y}\dot{v}_4=\bar{y}\left.\frac{\partial\phi_4(z)}{\partial z}\right|_{z=v_3}\dot{v}_3=\underbrace{\bar{y}\cos(v_3)}_{=\bar{v}_3}\dot{v}_3=\underbrace{\bar{v}_3}_{=\bar{v}_{-1}}\dot{v}_{-1}+\underbrace{\bar{v}_3}_{=\bar{v}_2}\dot{v}_2$$
$$=\underbrace{(\bar{v}_{-1}+\bar{v}_2v_1)}_{=\bar{v}_{-1}}\dot{v}_{-1}+\underbrace{\bar{v}_2v_{-1}}_{=\bar{v}_1}\dot{v}_1=\bar{v}_{-1}\dot{v}_{-1}+\underbrace{(-\bar{v}_1\sin(v_0))}_{=\bar{v}_0}\dot{v}_0$$

Choosing $\bar{y}=1$ one obtains $\bar{v}_{-1}\equiv\partial f/\partial x_1$ and $\bar{v}_0\equiv\partial f/\partial x_2$. Listing 3 shows a Python code where the sequence of instructions of the successive pullbacks are shown. From this simple example one can already make several fundamental observations:

```
from algopy import UTPM, zeros, sin, cos

x1 = UTPM([[3.],[1.]])
x2 = UTPM([[7.],[0.]])

# Step 1: evaluation of the function
vm1 = x1                                # independent
v0  = x2                                # independent
# ------------------------------------------------
v1 = cos(v0)                            # instruction 1
v2 = v1 * vm1                           # instruction 2
v3 = vm1 + v2                           # instruction 3
v4 = sin(v3)                            # instruction 4
# ------------------------------------------------
y  = v4                                 # dependent

# Step 2: reverse mode
v4bar = zeros((),dtype=y)
v3bar = zeros((),dtype=y)
v2bar = zeros((),dtype=y)
v1bar  = zeros((),dtype=y)
v0bar = zeros((),dtype=y)
vm1bar = zeros((),dtype=y)
v4bar.data[0,0] = 1.
# instructions 4,3,2 and 1
UTPM.pb_sin(v4bar, v3, v4, (v3bar,))
UTPM.pb_add(v3bar, vm1, v2, v3, (vm1bar, v2bar))
UTPM.pb_mul(v2bar, v1, vm1, v2, (v1bar, vm1bar))
UTPM.pb_cos(v1bar, v0, v1, (v0bar,))

forward_dfdx1= y.data[1,0]
reverse_df = [vm1bar.data[0,0], v0bar.data[0,0]]
hess_vec   = [vm1bar.data[1,0], v0bar.data[1,0]]
```

Listing 4. This Python code shows how the directional derivative $\nabla f(3,7)\cdot\begin{pmatrix}1\\0\end{pmatrix}$, the gradient $\nabla f(3,7)$ and the Hessian $\times$ vector product $\nabla^2 f(3,7)\cdot\begin{pmatrix}1\\0\end{pmatrix}$ can be evaluated using UTP arithmetic and the reverse mode.

1. One reverts the direction of the function evaluation to compute the gradient. Hence, this is called *reverse mode*.
2. Each of the four instructions to evaluate the function requires less than $\omega - 1$ operations during the reverse mode. To compute both the function and the gradient thus requires less than 4ω operations.
3. The intermediate values v_0, v_1, v_3, v_4 are required during reverse mode. I.e., the gradient can be computed cheaply in terms of arithmetic operations, but it is expensive in terms of memory.

It is also possible to compute higher-order derivatives by combining UTP arithmetic and the reverse mode of AD. The basic idea is that one can evaluate the program from Listing 3 in UTP arithmetic. One can see in Listing 4 that the `UTPM.pb_xyz` functions mirror the computation in the forward mode. The xyz is a wildcard for `sin`, `cos`, `add` and `mul`. These functions implement the relations defined by (6), i.e., compute $\bar{x}$ given x, y and $\bar{y}$. The function `algopy.zeros((), dtype=y)` is a generalization of `numpy.zeros` and returns a zero instance of the same type as `y`. I.e., new `UTPM` instances `vibar.data.shape=(2,1)` (`i`=,1,2,3,4) are created. Evaluating `# Step 1` yields the directional derivative $\nabla f(3,7)\cdot(1,0)^T$ and `# Step 2: reverse mode` computes the gradient $\nabla f(3,7)$ and the Hessian$\times$vector product $\nabla^2 f(3,7)\cdot(1,0)^T$.

2.5. Reverse mode in AlgoPy

Since it is cumbersome to revert large computer programs by hand, one typically automates this process as much as possible (hence the name "automatic" differentiation). In AlgoPy one

```
from algopy import Function, CGraph, sin, cos

cg = CGraph()
x1 = Function(3.)
x2 = Function(7.)
y = sin(x1 + cos(x2) * x1)
cg.trace_off()
cg.independentFunctionList = [x1,x2]
cg.dependentFunctionList = [y]
cg.plot('example_trace.svg', method='neato')
```

Listing 5. This example shows how the computation is traced in AlgoPy and saved in a computational graph. The graph is depicted in Fig. 1.

records the sequence of instructions in a computational graph. This process is called *tracing*. When the gradient (or the Hessian$\times$vector product) is requested, the graph is walked in reverse direction and the same instructions as shown in Listing 4 are performed. The tracing is most conveniently explained on the simple example shown in Listing 5: the variable of interest is wrapped in an `algopy.Function` instance. All the standard operators ($\pm$, $\times$, $\div$) and functions (exp, sin, ...) are overloaded and build, as a side effect, the computational graph shown in Fig. 1. For further information on implementation aspects we refer to [8,10].

2.6. Easy to use drivers

Let $f:\mathbb{R}^N\to\mathbb{R}$ and $g:\mathbb{R}^N\to\mathbb{R}^M$ be two sufficiently smooth functions. Then the following derivatives may be desired:

- gradient $\nabla f(x)\in\mathbb{R}^N$
- Jacobian $\times$ vector $\frac{\partial g}{\partial x}(x)v\in\mathbb{R}^M$
- vector $\times$ Jacobian $w^T\frac{\partial g}{\partial x}(x)\in\mathbb{R}^N$
- Hessian $\times$ vector $\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x}(x)v\right)\in\mathbb{R}^N$
- vector$\times$Hessian$\times$vector $\frac{\partial}{\partial x}\left(w^T\frac{\partial g}{\partial x}(x)v\right)\in\mathbb{R}^N$

where $v\in\mathbb{R}^N$ and $w\in\mathbb{R}^M$. Listing 6 demonstrates how AlgoPy can be used to compute these derivatives.

3. Numerical linear algebra functions

Now that the basics have been explained, we can discuss the more advanced features of AlgoPy. Traditionally, AD tools support just the scalar elementary functions. More elaborate operations such as slicing and algorithms such as the Cholesky and QR decomposition can, after all, be represented as a sequence of scalar elementary instructions. In that view, any AD tool supports NLA functions. However, there are a couple of issues that complicate matters:

1. Existing reference algorithms often contain non-differentiable statements. For instance, many algorithms shown in [13] contain checks whether some element is zero (or nearly so) to avoid redundant computations. It is therefore no surprise that reference implementations such as LAPACK [2] contain the same checks. Hence, to be on the safe side, one would need to modify and implement algorithms anew. This violates the idea of code reuse.
2. Since many NLA functions have an $\mathcal{O}(N^3)$ complexity, the computational graph is of the same order. In consequence memory of order $\mathcal{O}(N^3)$ is necessary. In practice one therefore quickly runs out of available memory.
3. Operator overloading tools require an interpreter that operates on the computational graph. This interpretation comes with an

```
import algopy; from algopy import UTPM
import numpy

def f(x):
    return x[0]*x[1]*x[2] + 7*x[1]

def g(x):
    out = algopy.zeros(3, dtype=x)
    out[0] = 2*x[0]**2
    out[1] = 7*x[0]*x[1]
    out[2] = 23*x[0] + x[2]
    return out

x = numpy.array([1,2,3],dtype=float)
v = numpy.array([1,1,1],dtype=float)
w = numpy.array([4,5,6],dtype=float)

# forward mode gradient
y=UTPM.extract_jacobian(f(UTPM.init_jacobian(x)))
# forward mode Jacobian
y=UTPM.extract_jacobian(g(UTPM.init_jacobian(x)))
# forward mode Jacobian-vector
y=UTPM.extract_jac_vec(g(UTPM.init_jac_vec(x, v)))

# trace f
cg = algopy.CGraph()
fx = algopy.Function(x)
fy = f(fx)
cg.trace_off()
cg.independentFunctionList = [fx]
cg.dependentFunctionList = [fy]

# trace g
cg2 = algopy.CGraph()
fx = algopy.Function(x)
fy = g(fx)
cg2.trace_off()
cg2.independentFunctionList = [fx]
cg2.dependentFunctionList = [fy]

# reverse mode gradient
result = cg.gradient(x)
# forward/reverse mode Hessian
result = cg.hessian(x)
# forward/reverse mode Hessian-vector
result = cg.hess_vec(x,v)
# forward mode Jacobian-vector
result = cg2.jac_vec(x,v)
# reverse mode vector-Jacobian
result = cg2.vec_jac(w,x)
# reverse mode Jacobian
result = cg2.jacobian(x)
# forward/reverse mode vector-Hessian-vector
result = cg2.vec_hess_vec(w,x,v)
```

Listing 6. This listing shows how AlgoPy can be used to compute derivatives of the functions f and g.

overhead and it is therefore desirable to keep the computational graph as small as possible.

4. Many Gaussian elimination type algorithms use pivoting and it is therefore often mandatory to rebuild the computational graph when input values change.

To circumvent such problems one can take a *hierarchical approach* to AD as suggested for instance by Bischof [6] or in [9]. I.e., one treats certain mathematical functions as atomic and derives dedicated algorithms for UTP arithmetic and the reverse mode. More to the point, let a sufficiently smooth function $F : \mathbb{R}^N \to \mathbb{R}^M, x \mapsto y = F(x)$ be given. In the hierarchical approach one tries to find an algorithm to evaluate $[y]_D = E_D(F)([x]_D)$ as well as an algorithm to compute $\bar{x}$,

```
import numpy
def f(A,b):
    v1 = A*b                          # broadcasting
    v2 = v1[:2,:2]                    # slicing
    v3 = numpy.dot(v2.T,v2)           # nla
    v4 = numpy.linalg.cholesky(v3)    # nla
    v5 = numpy.diag(v4)
    v6 = numpy.prod(v5)
    v7 = v6**2
    return v7

A = numpy.random.random((3,3))
b = numpy.random.random(3)
y = f(A,b)
```

Listing 7. A NumPy based program using slicing, broadcasting and NLA functions.

given $\bar{y}$ and x such that $\bar{y}^T\dot{y} = \bar{y}^T\frac{\partial F}{\partial x}\dot{x} = \bar{x}^T\dot{x}$ holds. As long as these relations are satisfied, this approach doesn't make any assumptions on the underlying algorithm.

The arguments and function values of NLA functions are matrices instead of column vectors. The canonical inner product for matrices $A, B \in \mathbb{R}^{M\times N}$ is

$$(A, B) = \mathrm{tr}(A^T B) = \sum_{n=1}^{N}\sum_{m=1}^{M} A_{mn}B_{mn} .$$

Thus, for functions $F : X \mapsto Y$, where X and Y are some matrices, the relation (6) generalizes to

$$tr(\bar{Y}^T\dot{Y}) = tr(\bar{X}^T\dot{X}) \quad . \tag{7}$$

One can find a nice discussion of matrix derivative results in the forward and reverse mode in [11,12].

3.1. Illustrative Example 1

Consider the program from Listing 7. It shows a contrived Python program where several features of NumPy are used: the element-wise multiplication between two arrays `A` and `b` (which are of different shapes) requires broadcasting. In a next step, a slice `v1[:2,:2]` is created, followed by several operations to compute the determinant of `v3`.

In Listing 8 one can see how AlgoPy can be used to compute its gradient in the reverse mode. The function evaluation gets traced and stored in the object `cg`. The corresponding computational graph is depicted in Fig. 2. Note that there is no node for `algopy.prod`. Instead, the required operations to evaluate the product are represented as nodes. This is because there doesn't exist a simple symbolic identity suitable for the reverse mode. In fact, the product is one of the motivating examples for AD and is also known as *Speelpenning's example* [31].

It is also possible to use UTP arithmetic to compute the Jacobian of the function. This is demonstrated in Listing 9. The function `UTPM.init_jacobian` takes care of initializing `Ab.data` whereas `UTPM.extract_jacobian` constructs the Jacobian from `y.data`. One finds that `Ab.data.shape= (2, 12, 3, 4)`, i.e., `UTPM.init_jacobian` initializes a `UTPM` instance with $P = 12$ directions and $D = 2$ coefficients of the shape (3, 4).

3.2. Illustrative Example 2

Listing 10 shows a Python program where the intermediate states of ten explicit time integration steps are stored in a two-dimensional array. Subsequently, this array is used as input for numerical linear algebra functions, including the QR decomposition and the real symmetric eigenvalue decomposition. The goal

```
import algopy
def f(A,b):
    v1 = A*b                          # broadcasting
    v2 = v1[:2,:2]                    # slicing
    v3 = algopy.dot(v2.T,v2)          # nla
    v4 = algopy.cholesky(v3)          # nla
    v5 = algopy.diag(v4)
    v6 = algopy.prod(v5)
    v7 = v6**2
    return v7

cg = algopy.CGraph()
A = algopy.Function(numpy.random.random((3,3)))
b = algopy.Function(numpy.random.random(3))
y = f(A,b)
cg.trace_off()
cg.independentFunctionList = [A,b]
cg.dependentFunctionList = [y]
cg.plot('nla_functions.svg',method='dot')

A = numpy.random.random((3,3))
b = numpy.random.random(3)
result = cg.gradient([A,b])
```

Listing 8. The function `f(A,b)` is traced and the resulting computational graph is used to evaluate the gradient.

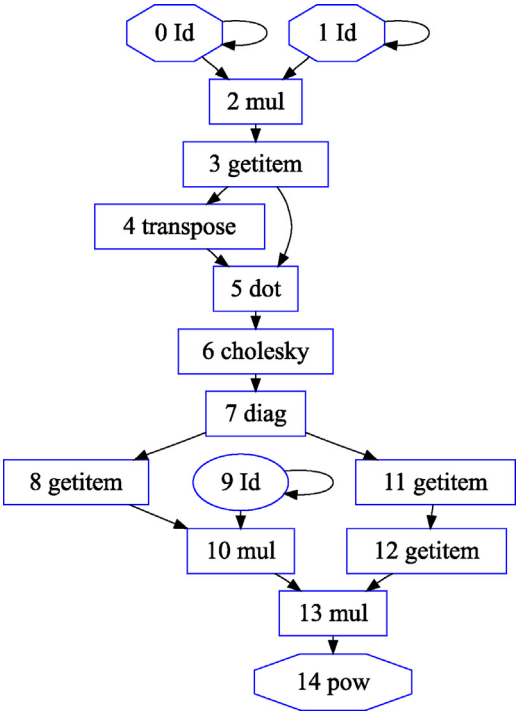

Fig. 2. The computational graph as generated by Listing 8.

of the example is to compute the first $D=30$ Taylor coefficients of the function $f(x(t))$, where $x(t) = (3,2)^T + (1,2)^T t \in \mathbb{R}^2$ is implemented as `eval_x` and $f : \mathbb{R}^2 \to \mathbb{R}$ as `eval_f`. In Fig. 3 one can see the function values of $f(x(t))$ sampled at points $t \in [-1.5, 1.5]$ and polynomial approximations of varying orders. E.g., for the 4th order approximation only the first 5 coefficients are used in a call to `numpy.polyval`.

3.3. Algorithms for hierarchical AD

We now come to the discussion how hierarchical algorithms for Taylor polynomial arithmetic and the reverse mode can be derived, i.e., *without* augmenting existing algorithms. This idea is is by no means new: first-order derivatives of a large variety of NLA functions have been thoroughly treated in the literature [23,30,17,29]. However, these references do not put their results in an algorithmic context and rather provide symbolic identities. Giles collected existing results and formulated them in a consistent way using the standard AD notation [12]. Much less literature exists on Taylor polynomial arithmetic applied to NLA functions [35]. So far,

```
def f(A,b):
    v1 = A*b                          # broadcasting
    v2 = v1[:2,:2]                    # slicing
    v3 = algopy.dot(v2.T,v2)          # nla
    v4 = algopy.cholesky(v3)          # nla
    v5 = algopy.diag(v4)
    v6 = algopy.prod(v5)
    v7 = v6**2
    return v7

Ab = numpy.random.random((3,4))
Ab = algopy.UTPM.init_jacobian(Ab)
A = Ab[:,:3]; b = Ab[:,3]
y = f(A,b)
result = algopy.UTPM.extract_jacobian(y)
```

Listing 9. UTP propagation.

```
import algopy, numpy
from algopy import zeros, dot, qr, eigh
Nts = 10
def eval_f(x):
    tmp = zeros((Nts,x.size),dtype=x)
    tmp[0] = x
    for nts in range(1,Nts):
        x[0] -= 0.001*x[0]*x[1]
        x[1] += 0.001*x[0]*x[1]
        tmp[nts] = x

    Q,R = qr(tmp)
    l = eigh(dot(R.T,R))[0]
    return l[0]/l[1]

def eval_x(t):
    return numpy.array([3,2])+t*numpy.array([1,2])

D,P = 30,1
t = algopy.UTPM(numpy.zeros((D,P)))
t.data[1,0] = 1.
x = eval_x(t)
y = eval_f(x)
```

Listing 10. Python code with a loop akin to an explicit time integration and calls to numerical linear algebra functions. The program is evaluated in univariate Taylor polynomial arithmetic.

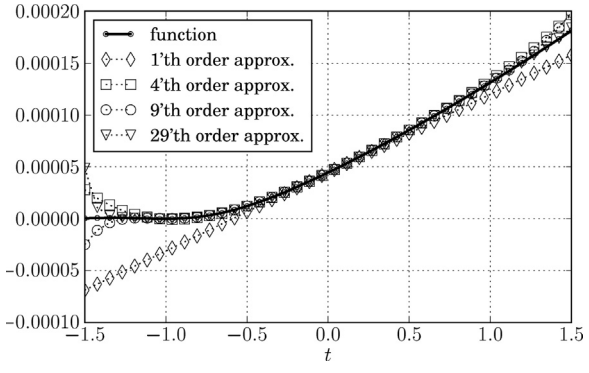

Fig. 3. Taylor series expansion of Listing 10.

apparently only Phipps [27] has described univariate Taylor polynomial (UTP) arithmetic applied to the NLA functions `dot, solve, inv` in the combined forward/reverse mode for use in a Taylor series integrator for differential algebraic equations.

The matrix multiplication $Z = XY$, $X \in \mathbb{R}^{N\times K}$, $Y \in \mathbb{R}^{K\times M}$ can be evaluated in UTP arithmetic using the recurrence

$$\sum_{d=0}^{D-1} Z_{[d]} T^d = \sum_{d=0}^{D-1}\sum_{k=0}^{d} X_{[d-k]} Y_{[k]} + \mathcal{O}(T^D) \ .$$

We define $\underline{\underline{D}}$ to mean equality in the first D coefficients to avoid the $\mathcal{O}(T^D)$ terms. The pullback has to satisfy

$$\mathrm{tr}(\overline{Z}^T \dot{Z}) = \mathrm{tr}(\overline{X}^T \dot{X}) + \mathrm{tr}(\overline{Y}^T \dot{Y})$$

and thus, according to the rules of matrix calculus, one obtains

$$\overline{X} = \overline{Z} Y^T \ \textit{and} \ \overline{Y} = X^T \overline{Z} \ . \tag{8}$$

The transpose $B = A^T$ of a matrix $A \in \mathbb{R}^{M\times N}$ corresponds to $[B]_D = [A^T_{[0]}, \ldots, A^T_{[D-1]}]$ and one finds for the reverse mode the relation

$$\mathrm{tr}(\overline{B}^T \dot{B}) = \mathrm{tr}(\overline{B}^T \dot{A}^T) = \mathrm{tr}(\overline{B} \dot{A}).$$

The inverse $[B]_D$ of the matrix-valued UTP $[A]_D = \sum_{d=0}^{D-1} A_{[d]} T^D$ is defined by

$$1 \overset{D}{=} [A]_D [B]_D \overset{D}{=} \sum_{d=0}^{D-1}\sum_{k=0}^{d} A_{[d-k]} B_{[k]} T^d. \tag{9}$$

Once $B_{[0]} = A^{-1}_{[0]}$ is known, one can successively compute the coefficients $B_{[d]}$ ($d = 1, 2, \ldots$), i.e., $B_{[1]} = -B_{[0]} A_{[1]} B_{[0]}$, etc. To derive the pullback formula it is necessary to find $\overline{A}$ given $\overline{B}, A, B$ such that $\mathrm{tr}(\overline{B}^T \dot{B}) = \mathrm{tr}(\overline{A}^T \dot{A})$ holds. Differentiation of the defining equation yields $0 = \dot{A}B + A\dot{B}$ and hence, using the cyclic invariance of the trace, one calculates $\mathrm{tr}(\overline{B}^T \dot{B}) = \mathrm{tr}(-\overline{B}^T B \dot{A} B) = \mathrm{tr}(-B\overline{B}^T B \dot{A})$ and thus

$$\overline{A} = -B^T \overline{B} B^T \ . \tag{10}$$

As one can see, both $[B]_D$ and $\overline{A}$ can be evaluated by application of NLA functions. Analogously to the scalar elementary functions, the pullback algorithm of `Y = inv(X)` is provided by `Xbar = UTPM.pb_inv(Ybar, X, Y, (Xbar,))`.

3.4. Cholesky decomposition

As another, more complicated example, we consider the Cholesky decomposition of a symmetric matrix positive definite matrix A. It is defined by

$$0 = G = A - L L^T \tag{11a}$$

$$0 = H = P_R \circ L \ , \tag{11b}$$

where $A, L \in \mathbb{R}^{M\times M}$. The element-wise product $P_R \circ L$ selects all elements above the diagonal of L and sets the other elements to zero. The functional dependence of the defining equations is denoted

$$L = \mathit{cholesky}(A) \ . \tag{12}$$

To compute $[L]_D = E_D(\mathrm{cholesky})([A]_D)$ one needs to solve

$$0 = [G]_D \overset{D}{=} [A]_D - [L]_D [L]_D^T \tag{13a}$$

$$0 = [H]_D \overset{D}{=} P_R \circ [L]_D \ . \tag{13b}$$

Now assume that the coefficient matrices of $[L]_D$ for some $D \geq 1$ were already obtained and compute the next $1 \leq E \leq D$ coefficients by performing a first order Taylor expansion of the system (13) for the extended Taylor polynomial $[L]_{D+E} = [L]_D + [\Delta L]_E T^D$. This is then solved for the yet unknown $[\Delta L]_E$. The resulting algorithm for the special case $E = 1$ is shown in Algorithm 1. $P_L \circ B$ and $P_D \circ B$ select the strictly lower triangular and diagonal part of B by setting all remaining elements of B to zero.

For the pullback we need to find $\overline{A}$ such that, given $\overline{L}$ and $A = LL^T$, the equation $\mathrm{tr}(\overline{L}^T \dot{L}) = \mathrm{tr}(\overline{A}^T \dot{A})$ is satisfied. One obtains

$$\begin{aligned} &\overline{A} = L^{-T} \overline{K} L^{-1} \ \textit{where} \\ &\overline{K} = \frac{1}{2}\left((P_L + P_D) \circ (L^T \overline{L}) + P_R \circ (\overline{L}^T L)\right). \end{aligned} \tag{14}$$

Other factorizations such as the QR and real symmetric eigenvalue decomposition lead to similar algorithms.

```
import numpy, algopy
def dot(x,y):
    N,K = x.shape
    K,M = y.shape
    z = numpy.zeros((N,M), dtype=algopy.UTPM)
    for n in range(N):
        for m in range(M):
            for k in range(K):
                z[n,m] += x[n,k]*y[k,m]
    return z

def eval_f(x,y):
    return numpy.trace(dot(x,y))

N = 2
cg = algopy.CGraph()
x = [algopy.Function(1.) for n in range(N**2)]
x = numpy.array(x).reshape((N,N))
y = [algopy.Function(2.) for n in range(N**2)]
y = numpy.array(y).reshape((N,N))
z = eval_f(x,y)
cg.trace_off()
cg.independentFunctionList = list(x.ravel())+\
                             list(y.ravel())
cg.dependentFunctionList = [z]
cg.plot('dot_traditional.svg')
```

Listing 11. Naive Python implementation of the matrix product. All operations to compute $\mathrm{tr}(XY)$ are traced and stored in the `CGraph` instance `cg`.

Algorithm 1. Algorithm to evaluate the Cholesky decomposition in UTP arithmetic.

input: $[A]_D = [A_{[0]}, \ldots, A_{[D-1]}]$, where $A_{[d]} \in \mathbb{R}^{M\times M}$ symmetric $\mathrm{rank}(A_{[0]}) = M$.

output: $[L]_D = [L_{[0]}, \ldots, L_{[D-1]}]$ lower triangular, where $L_{[d]} \in \mathbb{R}^{M\times M}$

$L_{[0]} = \mathrm{cholesky}(A_{[0]})$, $H_{[0]} = L_{[0]}^{-1}$

for $d = 1$ **to** $D-1$ **do**

$$\begin{aligned} \Delta G &= A_{[d]} - \sum_{k=1}^{d-1} L_{[d-k]} L_{[k]}^T \\ K &= H_{[0]} \Delta G H_{[0]}^T \\ L_{[d]} &= L_{[0]} \left(P_L \circ K + \frac{1}{2} P_D \circ K\right) \end{aligned}$$

end

3.5. Comparison of hierarchical and scalar AD

We have argued in the introduction of this section that the hierarchical approach can have distinct advantages over the scalar elementary AD. In Listing 11 one can find a Python code for a naïve implementation of the matrix multiplication of two matrices $X, Y \in \mathbb{R}^{N\times N}$. In a second step, the trace of the resulting matrix is evaluated. The corresponding computational graph (for $N = 2$) is depicted in Fig. 4.

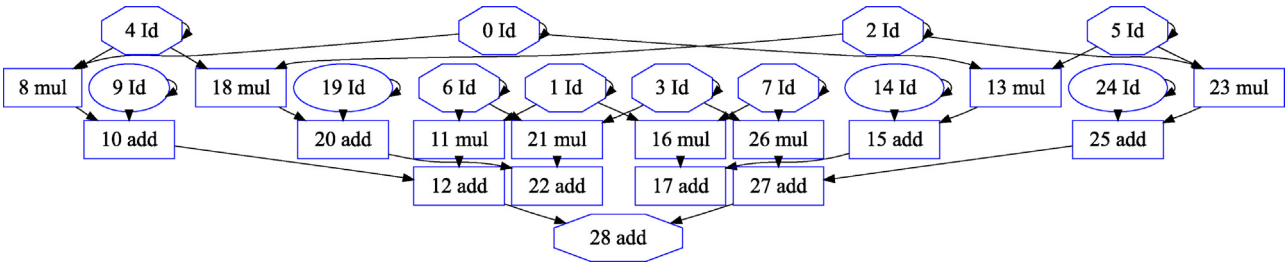

Fig. 4. The computational graph of the matrix product followed by taking the trace of the result.

Using the functions `algopy.dot` and `algopy.trace` would result in a computational graph with only four nodes: two "Id" nodes, one "dot" node and one "trace" node. Hence, instead of a graph of size $\mathcal{O}(N^3)$ one obtains a graph with just $\mathcal{O}(1)$ nodes. Working with large graphs is relatively slow in AlgoPy since the implementation of the reverse mode uses a simple Python for loop. For $N \gg 100$ the graph would also require a lot of memory. It is also much faster to use `algopy.dot` since it internally calls `numpy.dot`.

The pullback (8) of the matrix multiplication $Z = XY$, X, $Y \in \mathbb{R}^{N\times N}$ requires only the inputs X and Y. I.e., to apply the reverse mode only memory of order $\mathcal{O}(N^2)$ instead of $\mathcal{O}(N^3)$ is necessary.

On the other hand, in Fig. 4 one can see that there are several redundant computations (e.g., node "22 add"). I.e., one could optimize this graph. That means there is a trade-off between both approaches.

4. Comparison to other tools

There are several other Python tools that can be used to evaluate derivatives in Python. To allow a potential user to decide whether AlgoPy is suitable for the task at hand, we show here a rudimentary runtime analysis on two benchmark problems. The authors are aware of the following tools in Python: *pycppad* [4], *PyAdolc* [37], *uncertainties* [22], *numdifftools* [7], *funcdesigner* [21], *ScientificPython* [18], *upy* [1], *Theano* [5] and *SymPy* [32]. We decided to restrict the comparison to AlgoPy, numdifftools, PyAdolc and Theano. The reasoning is as follows: we wanted to include at least one tool that uses finite differences (numdifftools), one existing AD tool that is very popular (PyAdolc resp. ADOL-C) and one tool that uses symbolic differentiation in combination with code generation techniques (Theano).

All the tests were performed on a machine with an Intel(R) Core(TM)2 Duo CPU T7300 @2.00 GHz and 2,048,624 kB of RAM. The operating system is Linux version 2.6.32-26-generic with standard compiler gcc version 4.4.3. The versions are as follows: AlgoPy 0.3.1, Theano (2010-12-16), Numdifftools 0.3.1, PyAdolc (2010-12-16), Python 2.6.5, NumPy 1.3.0 and SciPy 0.7.0.

4.1. Example 1: minimal surface problem

It is advantageous to use the reverse mode of AD to compute the gradient of functions with many inputs. Consider the cost function

$$\mathbf{u} \mapsto \int_0^1 \int_0^1 \sqrt{1 + \left(\frac{\partial \mathbf{u}}{\partial x}\right)^2 + \left(\frac{\partial \mathbf{u}}{\partial y}\right)^2}\, dx dy$$

```
from numpy import random
def O_tilde(u):
    return numpy.sum(((u[1:,1:] - u[0:-1,0:-1])**2
           + (u[1:,0:-1] - u[0:-1, 1:])**2))/4 + 1
```

Listing 12. Python code to evaluate the gradient and Hessian–vector product on the minimal surface problem.

for $\mathbf{u} \in C^1([0,1]^2)$. Discretizing the function $\mathbf{u}$ and the partial derivatives on a grid by using finite differences yields

$$\mathbb{R}^{M\times M} \ni u \mapsto \tilde{O}(u) = 1 + \sum_{i=0}^{M-2}\sum_{j=0}^{M-2} O_{ij}(u) \ ,$$

where the components are computed by

$$\tilde{O}_{ij}(u) = \frac{(u_{i+1,j+1} - u_{i,j})^2 + (u_{i,j+1} - u_{i+1,j})^2}{4} \ .$$

Let the cost function be used as objective function of the constrained optimization problem

$$\min_{u\in\mathbb{R}^{M\times M}} \quad \tilde{O}(u) \ ,$$

with prescribed values on the boundary and additional box constraints inside the square:

$$\begin{aligned} &u(x,0) = 0, && u(0,y) = \sin\left(\frac{\pi}{2}y\right), \\ &u(x,1) = \exp\left(\frac{\pi}{2}x\right), && u(1,y) = \exp\left(\frac{\pi}{2}\right)\sin\left(\frac{\pi}{2}y\right); \\ &u(x,y) \geq \frac{5}{2} \quad \text{for} \quad (x,y) \in B\left(\left(\frac{1}{2},\frac{1}{2}\right),\frac{1}{4}\right). \end{aligned}$$

The optimization with a gradient-based optimizer requires at least the gradient $\nabla\tilde{O}(u) \in \mathbb{R}^{M\times M}$. Listing 12 shows a typical implementation of the cost function.

Fig. 5.

As a first benchmark we compute the gradient and Hessian–vector product of the minimal surface cost function. Of highest importance is that the computations be correct. And indeed one can observe in Fig. 6 that AlgoPy, PyAdolc and Theano differ only at a level close to the machine precision $\approx 10^{-16}$. Fig. 7 shows the time for the preprocessing (construction of the computational graph, graph optimization, code generation, etc.) and the time of the function evaluation. Fig. 8 shows that numdifftools and the forward mode of AlgoPy require more and more time to compute the derivatives when M is increased whereas AlgoPy (reverse mode), Theano and PyAdolc compute the gradient in a time which is within a constant multiple of the time to compute the function itself. The observed factor for AlgoPy is not quite the theoretical $\omega \in [3,4]$ but between 30 and 40.

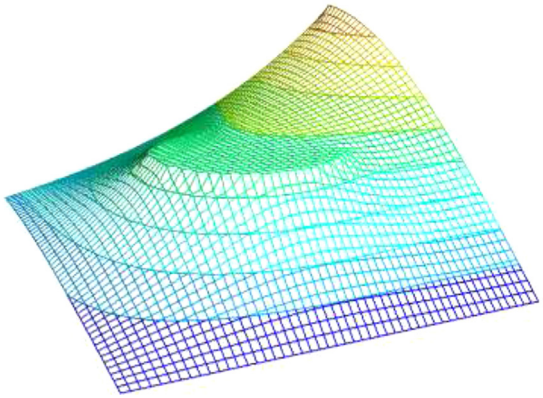

Fig. 5. Solution of the minimal surface optimization problem with cylindrical box constraints for $M = 50$.

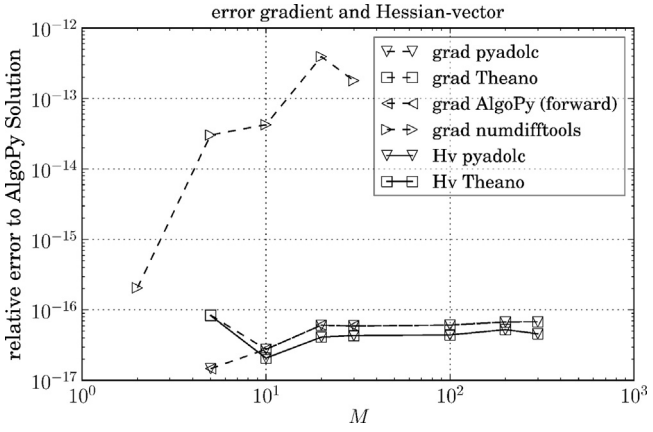

Fig. 6. The plots show the relative error $\|f_{ref} - f\|/\|f_{ref}\|$ between the gradient resp. Hessian × vector (Hv) product computed by AlgoPy and the other tools of the minimal surface problem.

4.2. Example 2: ODE fitting

As an example where directional derivatives are required, consider the parameter dependent differential equation

$$\dot{y}(t) = F(y(t), p), \tag{15}$$

with $y : [0, t_f] \to \mathbb{R}^n$ a differentiable function and $p \in \mathbb{R}^d$ the parameter vector. Now suppose that a number of position measurements (t_k, y_k) of the development of a physical system are given. The task

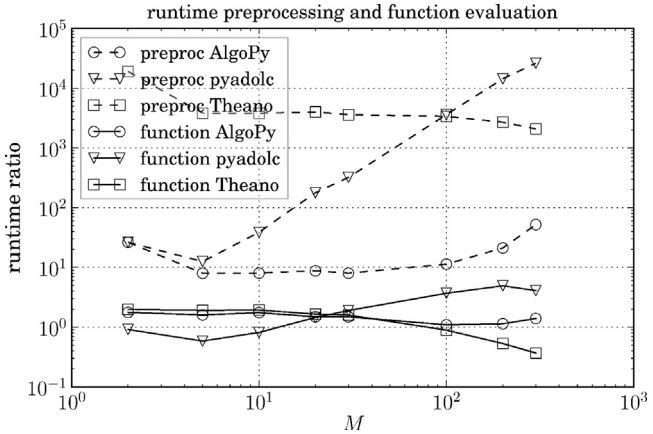

Fig. 7. Time for the preprocessing and the function evaluation relative to the time of the Python function evaluation (minimal surface problem).

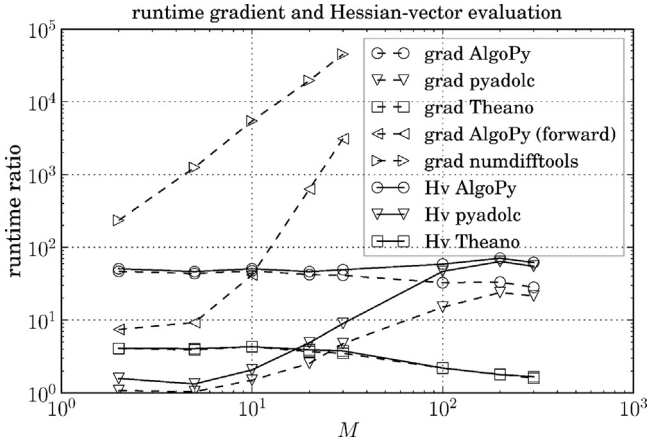

Fig. 8. The plot shows the time required to compute the gradient and Hessian–vector on the minimal surface problem relative to the Python function evaluation.

is to find a trajectory of the ODE that is close to the measurements by tuning the parameter vector p (and the initial point $y_0 = y(0)$). We pose this as a least squares problem, among all trajectories $y(t; y_0, p)$ find one that minimizes

$$I(y_0, p) = \sum_{k=1}^{N} \|y(t_k; y_0, p) - y_k\|_2^2.$$

Using a Gauss-Newton approach to this problem one needs to compute the Jacobian of the residual vector

$$G(y_0, p) = (y(t_k; y_0, p) - y_k)_{k=1}^{N}.$$

The case where d is much smaller than N this is most expediently done using the forward mode of automatic differentiation. As example ODE we use the van-der-Pol oscillator

$$\begin{aligned} 0 &= \ddot{x}(t) - p\,(1 - x(t)^2)\dot{x}(t) + x(t), \\ x(0) &= 3,\ \dot{x}(0) = 2 \end{aligned}$$

where the parameter p is to be estimated from noisy measurements of the original trajectory with the "true" value $p = 2.3$ (Listing 13).

```
import algopy; import numpy

def eval_f(p, x):
    """ computes ODE solution given initial
    conditions x and the parameter p"""
    tmp = algopy.zeros(Nts,dtype=p)
    y = algopy.zeros(x.shape, dtype=p)
    tmp[0] = y[0] = x[0]
    for nts in range(1,Nts):
        y[0] += 0.01*y[1]
        y[1] += 0.01*p[0]*(1-y[0]**2)*y[1] - y[0]
        tmp[nts] = y[0]
    return tmp

def eval_jac_f(p, x, meas):
    p  = algopy.UTPM.init_jacobian(p)
    y  = eval_f(p, x)
    return algopy.UTPM.extract_jacobian(y)

Nts = 60; x0 = numpy.array([3.,2.])
p1 = numpy.array([2.3])
y1 = eval_f(p1, x0)
```

Listing 13. Python code with an explicit integration of the ODE and computation of the derivative of the trajectory x wrt. the parameter p.

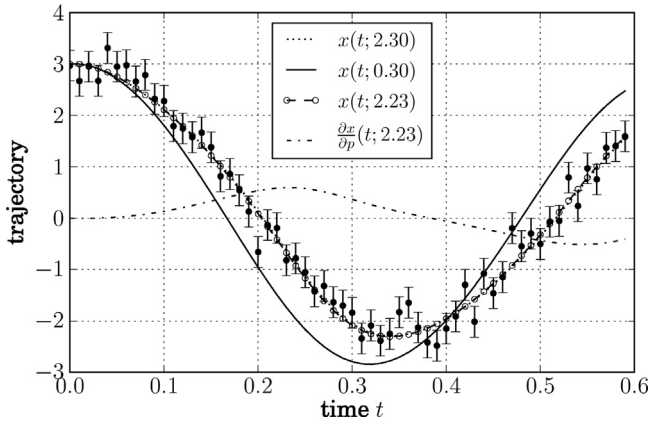

Fig. 9. Trajectories and sample points for the least squares fit.

```
Nts = 10
def eval_f(x):
    tmp = numpy.zeros((Nts,x.size),dtype=float)
    r   = numpy.array([-1.,1])
    tmp[0] = x
    for nts in range(1,Nts):
        x += r*0.001*x[0]*x[1]
        tmp[nts] = x
    return tmp
```

Listing 14. Python code of an explicit time-step algorithm.

In Fig. 9, one can see the original trajectory ($p = 2.3$) and simulated measurements where the outcomes of normal, independent and identically distributed random variables are added to the original trajectory. The resulting values are plotted as points with error bars denoting the standard deviation. The best least-squares fit was found for parameter $p = 2.37$. The trajectory for $p = 0.3$ is the initial value used for the least squares fit. Lastly, the sensitivity of the solution to changes in the parameter is plotted.

For the second benchmark problem we use the ODE fitting problem with a slightly modified code as shown in Listing 14. In Fig. 10 one can observe that many function evaluations have to be performed in order to amortize the time for the preprocessing. AlgoPy doesn't require a preprocessing and is therefore not shown in the graph. The runtimes to evaluate the Jacobian are depicted in Fig. 11. Numdifftools computed the Jacobian correctly within sensible errors but raised an exception for $Nts \geq 40$. In the current version of Theano there is no possibility to compute directional

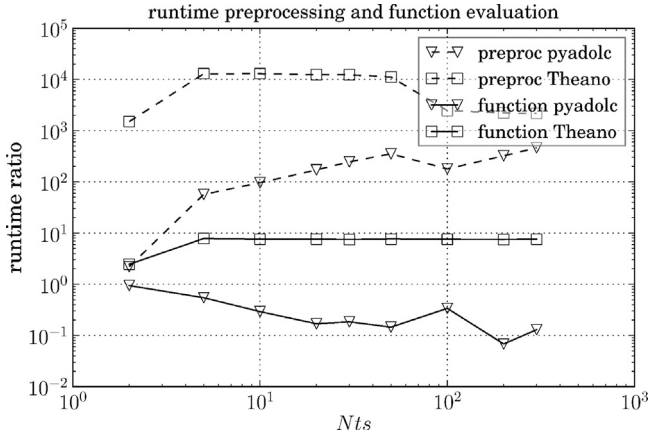

Fig. 10. Time for preprocessing and for the function evaluation relative to the runtime of the normal function evaluation (ODE fit problem).

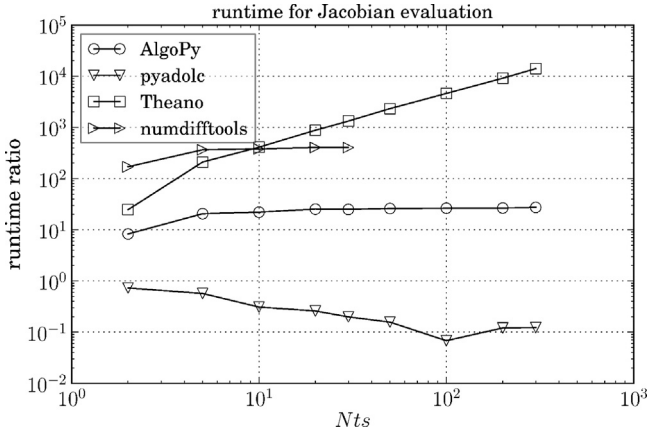

Fig. 11. Time for the Jacobian evaluation relative to the runtime of the normal function evaluation (ODE fit problem).

derivatives and thus the runtime increases approximately linearly with Nts. AlgoPy requires about 20–30 times more time to compute the Jacobian than to compute the function.

5. Conclusions and outlook

We have discussed how the well-known AD techniques (UTP arithmetic, reverse mode) are implemented in AlgoPy and have illustrated how these AD techniques generalize to hierarchical AD by application of matrix calculus rules. The differences between scalar AD and hierarchical AD have been highlighted and it has been demonstrated, at the example of the Cholesky decomposition, that it is possible to apply matrix calculus also to matrix factorizations.

The runtime comparison reveals that the current implementation of AlgoPy suffers speed-wise from a relatively large overhead. A reduction of this overhead could be a next step, possibly by code generation techniques similar to Theano.

It would also be a good idea to make all algorithms truly generic. Then one could trace the derivative evaluation. With such a feature it would be easy to compute nested derivatives of the form $\nabla g(\nabla f(x))$.

Many functions in large-scale optimization have sparse derivatives. One could add support for sparse derivatives, e.g., by propagation of sparsity patterns and the use graph coloring methods.

Acknowledgements

The authors wish to thank Gaël Varoquaux, Hans Petter Langtangen, Christophe Pradal and Andreas Griewank for advice on how to improve the readability of this work.

The plots were generated with Matplotlib [19] and Mayavi [28].

This research was partially supported by the Bundesministerium für Bildung und Forschung (BMBF) within the project NOVOEXP (Numerische Optimierungsverfahren für die Parameterschätzung und den Entwurf optimaler Experimente unter Berücksichtigung von Unsicherheiten für die Modellvalidierung verfahrenstechnischer Prozesse der Chemie und Biotechnologie) (03GRPAL3), Humboldt Universität zu Berlin.

References

[1] F. Romstedt, Python module: upy. http://github.com/friedrichromstedt/upy (2010).

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, 3rd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[3] M. Bartholomew-Biggs, S. Brown, B. Christianson, L. Dixon, Automatic differentiation of algorithms, J. Comput. Appl. Math. 124 (1–2) (2000) 171–190.
[4] B.M. Bell, S.F. Walter, PyCppAD, Python bindings to CppAD. https://github.com/b45ch1/pycppad, 2009.
[5] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, Y. Bengio, Theano: a CPU and GPU math expression compiler, in: Proceedings of the Python for Scientific Computing Conference (SciPy), June 2010, Oral.
[6] C.H. Bischof, M.R. Haghighat, Hierarchical approaches to automatic differentiation., in: Martin Berz, C. Bischof, G. Corliss, Andreas Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, PA, 1996, pp. 83–94.
[7] P.A. Brodtkorb, Numdifftools. http://code.google.com/p/numdifftools/, 2009.
[8] B. Christianson, Reverse accumulation of functions containing gradients. Tech. Report 278, Numerical Optimisation Centre, School of Information Sciences, University of Hertfordshire, Hatfield, UK, 1993.
[9] D.B. Christianson, A.J. Davies, L.C.W. Dixon, R. Roy, P. Van der Zee, Giving reverse differentiation a helping hand, Optim. Methods Softw. 8 (1997) 53–67.
[10] D.M. Gay, Automatic differentiation of nonlinear AMPL models, in: A. Griewank, G.F. Corliss (Eds.), Automatic Differentiation of Algorithms: Theory, Implementation, and Application, SIAM, Philadelphia, PA, 1991, pp. 61–73.
[11] M.B. Giles, An extended collection of matrix derivative results for forward and reverse mode automatic differentiation, Technical report, Oxford University Computing Laboratory, 2007. Report no 08/01.
[12] M.B. Giles, Collected matrix derivative results for forward and reverse mode algorithmic differentiation, in: C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008, pp. 35–44.
[13] G.H. Golub, C.F. Van Loan, Matrix Computations, 3rd ed., Johns Hopkins University Press, Baltimore, MD, USA, 1996.
[14] Andreas Griewank, A mathematical view of automatic differentiation, Acta Numer. 12 (2003) 321–398.
[15] A. Griewank, D. Juedes, Jean Utke, Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++, ACM Trans. Math. Softw. 22 (2) (1996) 131–167.
[16] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, in: Number 105 in Other Titles in Applied Mathematics, 2nd ed., SIAM, Philadelphia, PA, 2008.
[17] M.J.R. Healy, Matrices for Statistics, 2nd ed., Clarendon Press, Oxford, 2000.
[18] K. Hinsen, ScientificPython. http://dirac.cnrs-orleans.fr/plone/software/scientificpython/.
[19] J.D. Hunter, Matplotlib: A 2d graphics environment, Comput. Sci. Eng. 9 (3) (May–Jun 2007) 90–95.
[20] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org, 2001.
[21] D.L. Kroshko. Funcdesigner. http://openopt.org/FuncDesigner, 2010.
[22] E.O. Lebigot. Uncertainties. http://pypi.python.org/pypi/uncertainties, 2010.
[23] J.R. Magnus, H. Neudecker, Matrix Differential Calculus with Applications in Statistics and Econometrics, 2nd ed., John Wiley & Sons, 1999.
[24] R.D. Neidinger, Directions for computing truncated multivariate Taylor series, Math. Comput. 74 (249) (2005) 321–340.
[25] R.D. Neidinger, Introduction to automatic differentiation and MATLAB object-oriented programming, SIAM Rev. 52 (3) (2010) 545–563.
[26] E. Travis, Oliphant, Guide to NumPy, Trelgol Publishing, USA, 2006.
[27] E.T. Phipps, Taylor Series Integration of Differential-Algebraic Equations: Automatic Differentation as a Tool For Simulationg Rigid Body Mechanical Systems. PhD thesis, Cornell University, February 2003.
[28] P. Ramachandran, G. Varoquaux, Mayavi: 3D Visualization of Scientific Data, Comput. Sci. Eng. 13 (2) (2011) 40–51.
[29] J.R. Schott (Ed.), Matrix Analysis for Statistics, Wiley, New York, 1997.
[30] G.A.F. Seber, A Matrix Handbook for Statisticians, Wiley-Interscience, New York, NY, USA, 2007.
[31] B. Speelpenning, Compiling fast partial derivatives of functions given by algorithms. PhD thesis, Champaign, IL, USA, 1980. AAI8017989.
[32] SymPy Development Team. Sympy: Python library for symbolic mathematics. http://www.sympy.org, 2009.
[33] G. Varoquaux, S. van der Walt, K. Jarrod Millman (Eds.), Proceedings of the 8th Python in Science Conference, 2009, 8.
[34] G. Varoquaux, Travis Vaught, Jarrod Millman (Eds.), Proceedings of the 7th Python in Science Conference 8, 2008.
[35] William J. Vetter, Matrix calculus operations and Taylor expansions, SIAM Rev. 15 (2) (1973) 352–369.
[36] S.F. Walter, ALGOPY: algorithmic differentiation in Python, http://pypi.python.org/pypi/algopy, 2009.
[37] S.F. Walter, Pyadolc, Python bindings to ADOL-C, https://github.com/b45ch1/pyadolc, 2009.

Sebastian F. Walter was born in Germany in 1981. He received his diploma in physics from the ETH Zürich and continued his studies at the Humboldt-Universität zu Berlin, Institut für Mathematik, in the BMBF project NOVOEXP. His research interests include model-based optimum experimental design, algorithmic differentiation and scientific computing in general.

Lutz Lehmann was born 1972 in Germany. He received in 1999 the diploma in mathematics and in 2007 the PhD in wavelet analysis and computational algebraic geometry, both at Humboldt-Universität zu Berlin. Since 2009 he is a member of Matheon. His research interests include differential equations, Newton's method applied to singular situations and automatic differentiation.