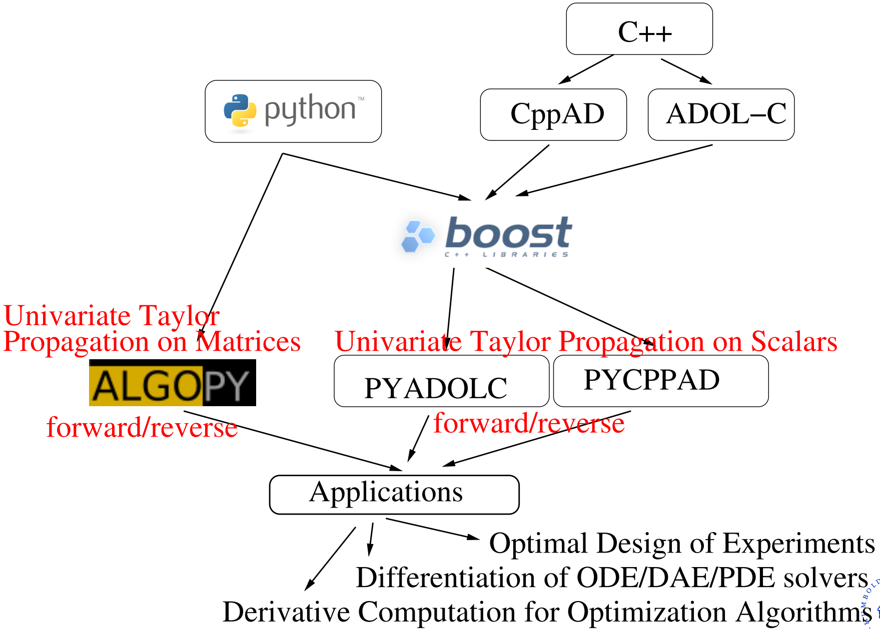
Algorithmic Differentiation in Python

Working with PYADOLC, PYCPPAD and ALGOPY from a User's Perspective

Sebastian F. Walter

Humboldt Universität zu Berlin

Juli, 17'th 2009


C++
python
CppAD
ADOL–C
boost
C++ LIBRARIES
Univariate Taylor
Propagation on Matrices
Univariate Taylor Propagation on Scalars
ALGOPY
PYADOLC
PYCPPAD
forward/reverse
forward/reverse
Applications
Optimal Design of Experiments
Differentiation of ODE/DAE/PDE solvers
Derivative Computation for Optimization Algorithms

Available Software for AD in Python to my Knowledge

- Differentiation Module in the ScientificPython package,[1]
  - forward mode, uses lambda functions
- **PYADOLC**, wrappper of ADOL-C [2]
  - arbitrary order vector forward/reverse taylor propagation
  - convenience function for hessian, jacobians, ...
  - sparse Hessian and Jacobian support by matrix compression
  - Very good Numpy support (array operations, slicing, ...)
  - Pythonic feel
- **PYCPPAD**, wrapper of CppAD (collaboration with Brad Bell),[3]
  - second order vector forward/reverse
  - convenience functions for hessian, jacobian, ...
  - AFAIK: same speed as CppAD
- **ALGOPY**[4]
  - pure Python, higher order vector forward/reverse on scalars and matrices

[1] http://dirac.cnrs-orleans.fr/plone/software/scientificpython
[2] http://www.github.com/b45ch1/pyadolc
[3] http://www.github.com/b45ch1/pycppad
[4] http://www.github.com/b45ch1/algopy

Example 1: Gradient of Toy Function with PYADOLC

- **if possible: Run Live Example...**
- **Simple Example: Gradient**

```python
import numpy
from adolc import *

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

x = numpy.array([1.*n +1. for n in range(3)])
ax = adouble(x)

trace_on(1)
independent(ax)
ay = f(ax)
dependent(ay)
trace_off()

print gradient(1,x)
```

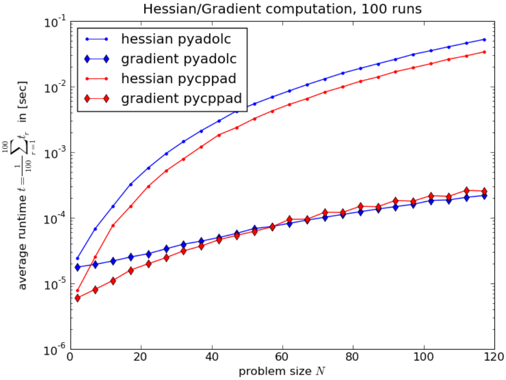
Example 2: Toy Problem with PYADOLC and PYCPPAD

```python
# PYCPPAD
x      = numpy.zeros(N, dtype=float)
ax    = pycppad.independent(x)
atmp = []
for n in range(N):
    atmp.append(numpy.sin( numpy.sum(ax[:n])))
ay    = numpy.array( [ ax[0] * numpy.sin( numpy.sum(atmp)) ] )
f    = pycppad.adfun(ax, ay)
x    = numpy.random.rand(N)
w    = numpy.array( [ 1.] )
H    = f.hessian(x, w)

# PYADOLC
x      = numpy.zeros(N, dtype=float)
adolc.trace_on(0)
ax = adolc.adouble(x)
adolc.independent(ax)
atmp = []
for n in range(N):
    atmp.append(numpy.sin( numpy.sum(ax[:n])))
ay    = numpy.array( [ ax[0] * numpy.sin( numpy.sum(atmp)) ] )
adolc.dependent(ay)
adolc.trace_off()
H = adolc.hessian(0,x)
```

Example 2: Performance, PYCPPAD vs PYADOLC

code at ./pyadolc/tests/comparison_pycppad_pyadolc/compare_pycppad_pyadolc.py

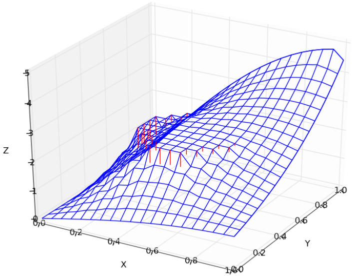


-
- factor two in Hessian computation: 1) ADOLC can't do vector forward followed by a reverse run 2) more cache misses because of nonlocality
- PYADOLC better for implementing univariate Taylor propagation forward/reverse and more Pythonic User Interface

Example 3: Minimal Surface Problem with PYADOLC

- Minimal Surface Problem:

$$
\begin{aligned}
u : S \subset [0,1] \times [0,1] \quad &\rightarrow \quad \boldsymbol{R} \qquad u \in C^1(S) \\
O(u) \quad &= \quad \int_0^1 \int_0^1 \sqrt{1 + \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2} \mathrm{d}x\mathrm{d}y \\
&\approx \quad \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} O_{ij}(u) \\
O_{ij}(u) \quad &:= \quad h^2 \left[1 + \frac{(u_{i+1,j+1} - u_{i,j})^2 + (u_{i,j+1} - u_{i+1,j})^2}{4} \right]
\end{aligned}
$$



Nonlinear Program with Inequality Box Constraints:

$$
\begin{aligned}
\boldsymbol{R}^{m \times m} \ni u_* \quad &= \quad \mathrm{argmin}_u O(u) \\
\text{s.t.} \quad 0 \quad &\leq \quad u_{ij} \quad \forall (i,j) \in \text{Cylinder set}
\end{aligned}
$$

Example 3: Code, Numpy Slicing and Broadcasting works

```
import numpy
from adolc import *
def O_tilde(u):
        M = numpy.shape(u)[0]
        h = 1./(M-1)
        return M**2*h**2 +
          numpy.sum(0.25*( (u[1:,1:] - u[0:-1,0:-1])**2
            + (u[1:,0:-1] - u[0:-1, 1:])**2))
M = 5
h = 1./M
u = numpy.zeros((M,M),dtype=float)
u[0,:]=  [numpy.sin(numpy.pi*j*h/2.) for j in range(M)]
u[-1,:] = [ numpy.exp(numpy.pi/2) * numpy.sin(numpy.pi * j * h / 2.)
u[:,0]= 0
u[:,-1]= [ numpy.exp(i*h*numpy.pi/2.) for i in range(M)]
trace_on(1)
au = adouble(u)
independent(au)
ay = O_tilde(au)
dependent(ay)
trace_off()
ru = numpy.ravel(u)
rg = gradient(1,ru)
g = numpy.reshape(rg, numpy.shape(u))
```

Differences Between ADOL-C and PYADOLC

- Python: **can't** overload the = operator
- Python: **garbage collector**
- ADOL-C differentiates between unnamed variables (adub) and named variables (adouble)
- **Test Function**:

```
for n in range(N):
    ay = ay * ay
```

- in C++: `ay * ay` does create adub object, assign to adouble, go out of scope. Thus, memory address may be reused

- PYADOLC

register	0	1	2	3	4	5	6
operation							
	1.	0.	0.	0.	0.	0.	0.
assign \$0 → \$1	1.	1.	0.	0.	0.	0.	0.
mul \$1 \$1 → \$2	1.	1.	1.	0.	0.	0.	0.
mul \$2 \$2 → \$3	1.	1.	1.	1.	0.	0.	0.
mul \$3 \$3 → \$4	1.	1.	1.	1.	1.	0.	0.

- ADOL-C

register	0	1	2	3	4	5	6
operation							
	1.	0.	0.	0.	0.	0.	0.
assign \$0 → \$1	1.	1.	0.	0.	0.	0.	0.
mul \$1 \$1 → \$1	1.	1.	0.	0.	0.	0.	0.
mul \$1 \$1 → \$1	1.	1.	0.	0.	0.	0.	0.
mul \$1 \$1 → \$1	1.	1.	0.	0.	0.	0.	0.

Tape of ADOL-C: tape_11.tex

generated by pyadolc/tests/tape_equivalence_PyADOLC_ADOLC/adolc.exey

Only two named variables using: loc 0 and loc 1

code	op	loc	loc	loc	loc	double	double	value	value	
33	start of tape									
39	take stock op			2	0		$6.908924e-310$			
1	assign ind				0		$1.000000e+00$			
3	assign a			0	1					1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
15	mult a a		1	1	1				$1.000000e+00$	1.000000
2	assign dep				1					
0	death not			0	2					1.000000
32	end of tape									

Tape Generated by PYADOLC, tape_9.tex

generated by pyadolc/tests/tape_equivalence_PyADOLC_ADOLC/pyadolc.py

21 named variables

code	op	loc	loc	loc	loc	double	double	value	value	
33	start of tape									
39	take stock op			2	0		0.000000e + 00			
1	assign ind				1		1.000000e + 00			
15	mult a a		1	1	2				1.000000e + 00	1.000000e
15	mult a a		2	2	3				1.000000e + 00	1.000000e
15	mult a a		3	3	4				1.000000e + 00	1.000000e
15	mult a a		4	4	5				1.000000e + 00	1.000000e
15	mult a a		5	5	6				1.000000e + 00	1.000000e
15	mult a a		6	6	7				1.000000e + 00	1.000000e
15	mult a a		7	7	8				1.000000e + 00	1.000000e
15	mult a a		8	8	9				1.000000e + 00	1.000000e
15	mult a a		9	9	10				1.000000e + 00	1.000000e
15	mult a a		10	10	11				1.000000e + 00	1.000000e
15	mult a a		11	11	12				1.000000e + 00	1.000000e
15	mult a a		12	12	13				1.000000e + 00	1.000000e
15	mult a a		13	13	14				1.000000e + 00	1.000000e
15	mult a a		14	14	15				1.000000e + 00	1.000000e
15	mult a a		15	15	16				1.000000e + 00	1.000000e
15	mult a a		16	16	17				1.000000e + 00	1.000000e
15	mult a a		17	17	18				1.000000e + 00	1.000000e
15	mult a a		18	18	19				1.000000e + 00	1.000000e
15	mult a a		19	19	20				1.000000e + 00	1.000000e
15	mult a a		20	20	21				1.000000e + 00	1.000000e
2	assign dep				21					
0	death not			0	21					1.000000e
32	end of tape									

Workaround for PYADOLC: use $\ll=$ operator instead of $=$

```python
from adolc import *
import numpy as npy
trace_on(10)
ax = adouble(1.)
independent(ax)
ay = ax
for i in range(N):
    ay <<= ay * ay

dependent(ay)
trace_off()

tape_to_latex(10,npy.array([x]),npy.array([0]))
```

- $\ll=$ in Python calls operator_eq_adub in C++:

  `badouble& (badouble::*operator_eq_adub) ( const adub& ) = &badouble::operator=`

Resulting Tape: tape_10.tex

generated by pyadolc/tests/tape_equivalence_PyADOLC_ADOLC/pyadolc.py

code	op	loc	loc	loc	loc	double	double	value	value	
33	start of tape									
39	take stock op			1	0		$0.000000e + 00$			
40	assign d one				1					
1	assign ind				1		$1.000000e + 00$			
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
15	mult a a		1	1	1				$1.000000e + 00$	$1.000000e$
2	assign dep				1					
0	death not			0	21					$1.000000e$
32	end of tape									

Quick Performance Comparison:

- **ADOL-C**

```
speelpenning:
Adolc function taping: ........ elapsed time: 0.000058
Adolc function evaluation: 0.000000 elapsed time: 0.000013
gradient evaluation: ........ elapsed time: 0.000028

matrix vector multiplication:
Adolc function taping: ........ elapsed time: 0.001564
Adolc function evaluation: 1.051874 elapsed time: 0.000209
jacobian evaluation: ........ elapsed time: 0.009419
```

- **PYADOLC**

```
speelpenning:
PyADOLC function taping: ........ elapsed time: 0.000535
Adolc function evaluation: 0.000000 elapsed time: 0.000031
gradient evaluation: ........ elapsed time: 0.000035

matrix vector multiplication:
PyADOLC function taping: ........ elapsed time: 0.036444
Adolc function evaluation: 1.051874 elapsed time: 0.000294
jacobian evaluation: ........ elapsed time: 0.015260
```

- Approximately: $1 \leq \frac{\text{Runtime(PYADOLC)}}{\text{Runtime(ADOL-C)}} \leq 2$

Univariate Taylor Propagation on Matrices UTPM

Differentiate obj. fun. operating on matrices by forward/reverse UTP:

$$\begin{aligned} q_* &= \operatorname{argmin}_q \Phi(C(q)) \\ C &= \begin{pmatrix} I & 0 \end{pmatrix} \begin{pmatrix} J_1^T J_1 & J_2^T \\ J_2 & 0 \end{pmatrix}^{-1} \begin{pmatrix} J_1^T J_1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} J_1^T J_1 & J_2^T \\ J_2 & 0 \end{pmatrix}^{-T} \begin{pmatrix} I \\ 0 \end{pmatrix} \end{aligned}$$

Regard matrices $\mathbb{M}$ as **elementary datatypes**.

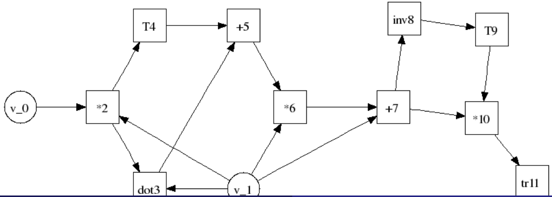
$$F : \boldsymbol{R}^{N_X \times M_X} \to \boldsymbol{R}^{N_Y \times M_Y} \Longrightarrow F : \mathbb{M}_{N_X, M_X} \to \mathbb{M}_{N_Y, M_Y}$$

Transformation: UTPS $\Leftrightarrow$ UTPM

$$\begin{pmatrix} \sum_{d=0}^D X_d^{11} t^d & \dots & \sum_{d=0}^D X_d^{1M} t^d \\ \vdots & \ddots & \vdots \\ \sum_{d=0}^D X_d^{N1} t^d & \dots & \sum_{d=0}^D X_d^{NM} t^d \end{pmatrix} = \sum_{d=0}^D \begin{pmatrix} X_d^{11} & \dots & X_d^{1M} \\ \vdots & \ddots & \vdots \\ X_d^{N1} & \dots & X_d^{NM} \end{pmatrix} t^D$$

ALGOPY builds internally a Computational Graph

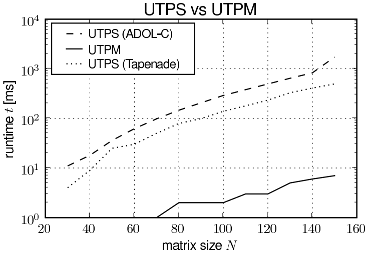
```
X = 2 * numpy.random.rand(2,2,2,2); Y = 2 * numpy.random.rand(2,2,2,2
AX = Mtc(X)
AY = Mtc(Y)
cg = CGraph()
FX = Function(AX)
FY = Function(AY)
FX = FX*FY
FX = FX.dot(FY) + FX.transpose()
FX = FY + FX * FY
FY = FX.inv()
FY = FY.transpose()
FZ = FX * FY
FTR = FZ.trace()
cg.independentFunctionList = [FX, FY]
cg.dependentFunctionList = [FTR]
cg.plot(filename = 'trash/computational_graph_circo.png', method = 'c
```

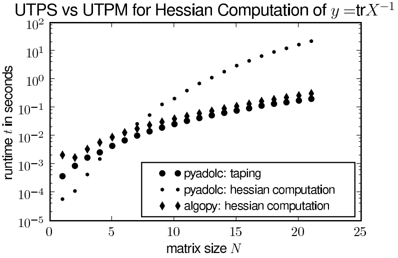


AD on Matrix Valued Functions: UTPM vs UTPS

source available at http://github.com/b45ch1/hpsc_hanoi_2009_walter

- As far as I know: ADMC++ also uses UTPM (operator overloading in Matlab/ execution in C++).[5]
- Comparison: UTPS vs UTPM





Test Function: $f : \boldsymbol{R}^{N\times N} \to \boldsymbol{R}\,, \qquad X \mapsto \mathrm{tr}(X^{-1})$

[5] `http://sourceforge.net/projects/admcpp`

Implementation 1: Operator Overloading in Python

Code snippet from algopy/algopy.py

```python
class Mtc:
    def __init__(self, X):
        """ INPUT:  shape(X) = (D,P,N,M)
            D: Degree of the Matrix Polynomial
            P: Number of Forward Directions
            N: Number of rows of the matrix
            M: Number of cols of the matrix """
        if ndim(X) == 4:    self.TC = asarray(X)
        else: raise NotImplementedError

    def __mul__(self,rhs):
        retval = Mtc(zeros(shape(self.TC)))
        (D,P,N,M) = shape(retval.TC)
        for d in range(D):
            retval.TC[d,:,:,:] = sum(
                self.TC[:d+1,:,:,:] * rhs.TC[d::-1,:,:,:], axis=0)
        return retval

X = Mtc( zeros((D,P,M,N))
Y = Mtc( zeros((D,P,N,M))
Z = X * Y
Z = X.__mul__(Y) # equivalent
```

Implementation 2: Using Boost:Python

```
adub *adub_add_badouble_badouble(const badouble &lhs, const badouble &
void hov_forward(short tape_tag, int M, int N, int D, int P,
                 bpn::array &bpn_x, bpn::array &bpn_V, bpn::array &bp
    double* x = (double*) nu::data(bpn_x);
    ...
    hov_forward(tape_tag, M, N, D, P, x, V, y, W);
}
BOOST_PYTHON_MODULE(_adolc){
import_array();
bpn::array::set_module_and_type("numpy", "ndarray");
def("trace_on",trace_on_default_argument);
def("trace_off",trace_off_default_argument);
def("gradient",        &c_wrapped_gradient);
def("hessian",         &c_wrapped_hessian);
def("jacobian",        &c_wrapped_jacobian);
def("hov_forward",     &hov_forward);
class_<badouble>("badouble", init<const badouble &>())
    .def("__add__", adub_add_badouble_badouble, return_value_policy<m
    .def("__mul__", adub_mul_badouble_badouble, return_value_policy<m
}
```

Summary: The current state

- All examples here are part of the examples resp. unit test of PYADOLC and ALGOPY
- AD tools in Python are sufficiently mature to do serious prototyping
  1. PYADOLC and PYCPPAD transform code to low level register machine language (without jump statements)
  2. ALGOPY high level description of algorithms
- Execution speed is comparative to pure C++ ADOL-C or CppAD

Outlook: Where to go from here

- wrap the Checkpointing functionality of ADOL-C
- Fix the problem of ADOL-C calling `exit()` on errors (this quits python too...)
- improve memory management of ALGOPY
- add missing linear algebra routines (LU, QR, LDU, eig(A))
- add sparse matrix support in ALGOPY