AD-Tools from a user’s perspective

Johannes Willkomm, Martin Bücke

RWTH Aachen
Institute for Scientific Computing (SC) http://www.sc.rwth-aachen.de/

Sixth European Workshop on Automatic Differentiation
November 15 and November 16, 2007
INRIA Sophia-Antipolis, France
Introduction

AD with C++

A regression test

A real life use case
Here: Operator-Overloading (OO)  
Show AD as experienced by a novice user  
Test four different AD tools

- ADOL–C
- CppAD
- FADBAD
- Sacado
The potential AD user

The potential AD user:

- Is a domain expert
- Is not a programming or CS expert
- Uses programming language as a tool to achieve ends
- Does not want to change the program code too much

For AD by operator overloading this means the user

- may understand how to change the data type to the AD tool’s active type
- may understand this requires some work
- may even know template programming (C++) to achieve this
- will possibly not understand the full implications
- will expect it to work
How to do C++ AD

C++ templates are a great help for overloading AD tools. The basic usage pattern:

- Make the relevant functions and classes templates
- Change `double` to a type parameter, `T`
- Use `f<double>` where before used `f`
- Use `f<adouble>` for differentiation with ADOL–C, etc.

Advantages:

- One code is used for normal and active version
- Quick to implement given an existing code
- Can easily use both the normal and active versions together
Four overloading AD tools

- ADOL–C 1.10
- CppAD 20071001
- FADBAD 2.1
- Sacado from Trilinos 8.0.2

Two basic variants exist

- With a tape: Make the overloaded operators trace the execution, compute derivative later by reading trace forwards or backwards.
- Tapeless: Compute forward derivatives on the fly, stored in the active object. This is only applicable with few input variables.

Some tools can also write the tape to disk, others simply create a linked pointer structure in memory during the reverse mode.
ADOL–C 1.10.2 [Griewank, 2006].

- Forward and reverse mode
- Scalar and vector functions
- First and higher order derivatives
- Active type is class adouble
- New tapeless active type adtl :: adouble
- Can write the tape to disk
- Open source, Common Public License 1.0
CppAD 20071001 [Bell, 2003].

- Forward and reverse mode
- Scalar and vector functions
- First and higher order derivatives
- Active type is template CppAD::AD<\texttt{class } T> 
- Can use std :: complex<\texttt{double}> for T
- Recursive usage possible, e.g. AD<AD<\texttt{double}>>
- Open source, dual licensed, GPL or Common Public License

Version 1.0
FADBAD

FADBAD 2.1 [Stauning, Bendtsen, 1996].

- Forward and reverse mode
- Active types are templates fadbad::F<\texttt{class T}> (forward dynamic), fadbad::F<\texttt{class T, int N}> (forward static), and fadbad::B<\texttt{class T}> (reverse).
- No utility functions, only the derivative information at individual active variables is available
- Higher order derivatives by recursive usage, e.g. F<B<\texttt{double}>> (forward on reverse mode) or fadbad::T<\texttt{class T}> for Taylor series.
- Extensible to new types and functions by implementing spezialization of a ”traits” template
- License: Proprietary, free for non-commercial use.
Sacado from Trilinos 8.0.2 [Gay, Phipps, 2007].

- Forward and reverse mode
- Scalar and vector functions
- Active types are templates Sacado::Fad::DFad<\texttt{class T}> (forward dynamic), Sacado::Fad::DFad<\texttt{class T, int N}> (forward static), and Sacado::Rad::ADvar<\texttt{class T}> (reverse).
- Second order derivatives by different active type.
- Open source, GPL licensed
A regression test for expressions

Test all possible expressions for correctness:

- regular result correct?
- all partial derivatives correct?

All the basic functions and operators are unary or binary. Generate all $a_n$ trees of height $n$ of unary and binary operators with $b$ binary, $u$ unary and $z$ ”0-ary” nodes:

$$a_n = b \cdot a_{n-1}^2 + u \cdot a_{n-1} + z$$

Test only done for $n = 2$.
Expression trees generated in XML with XSLT, testing C++ code generated from these with XSLT.
Application to other languages should not be difficult.
Unary operators:

- +, -, !, sin, cos, exp, log, ceil, floor, ...

Binary operators:

- +, -, *, /, pow, atan2, min, max, ==, !=, <, =, ...

"0-ary" elements:

- One double constant, two variables \( a \) and \( b \).

Omit invalid cases: assignment to non-L-value.

Omit duplicate cases:

- only test \( \text{pow}(a,a) \) and \( \text{pow}(a, b) \), but not \( \text{pow}(b, a) \)
- only test \( \text{pow}(0.12,a) \) and \( \text{pow}(a, 0.12) \), but not \( \text{pow}(b, 0.12) \) or \( \text{pow}(0.12, b) \)

Test all first order partial derivatives in forward and reverse mode.
Results

Results are positive, but some weaknesses revealed.

- Range of supported operators varies. Sacado seems to implement the most operators of the four.
- Three computational bugs found in two of the tools. One interesting case: $a \ast= b$ works OK, but reflexive case $a \ast= a$ does not.
- Operator signatures not always correct, e.g. `operator =` returning `void`, which does not allow chaining $x = y = z = 0$.
- No support for POSIX math functions like `expm1`, `log1p`, `j0`
Use case DROPS

DROPS: 3D Finite Element software package [Groß et al., 2005].

- Used for simulation of levitating droplets.
- Here: apply AD to 3D time-dependent heat conduction problem to solve inverse heat conduction problem with conjugate gradients method
- Requires very long gradients (length 3 million) of a scalar objective function $J$ measuring the error
- Analytical adjoint solution available

Additionally differentiated with ADOL–C for comparison.
The code could easily be rewritten into one large template class $P$. Use $P<\text{adouble}>$ with ADOL–C, ..., to differentiate the whole program at once. The default constructor of all four active (reverse) types must be modified first, to initialize value to zero, allocate heap values... Very easy to fix in all cases: Remove default constructor, make one by adding a default argument to the one that takes a base type $T$ as argument: $\text{AD}<T>::\text{AD}(T \ \text{const} \ &\text{initial} = T())$ Successful, all four tools return correct gradient. Interesting: ADOL–C and Sacado return exactly the same gradient.
Black box approach: Results

Spatial resolution: $2 \times 12 \times 24$, with 15, 30 or 45 time steps: gradient lengths 5200, 10075 or 14950.

<table>
<thead>
<tr>
<th>$n_t$</th>
<th>$t_{\nabla}/t_J$</th>
<th>ADOL-C</th>
<th>CppAD</th>
<th>FADBAD</th>
<th>Sacado</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td>26.97</td>
<td>23.22</td>
<td>92.14</td>
<td>52.09</td>
</tr>
<tr>
<td></td>
<td>Mem/MB</td>
<td>13 + 335</td>
<td>361</td>
<td>1726</td>
<td>2953</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>29.25</td>
<td>25.62</td>
<td>91.11</td>
<td>59.77</td>
</tr>
<tr>
<td></td>
<td>Mem/MB</td>
<td>13 + 711</td>
<td>825</td>
<td>3535</td>
<td>5997</td>
</tr>
<tr>
<td>45</td>
<td></td>
<td>34.71</td>
<td>30.17</td>
<td>106.39</td>
<td>68.55</td>
</tr>
<tr>
<td></td>
<td>Mem/MB</td>
<td>17 + 1126</td>
<td>1381</td>
<td>5376</td>
<td>9089</td>
</tr>
</tbody>
</table>

ADOL–C: Memory is in RAM + on disk, all other only RAM. Timings are *wallclock* times.
Advanced solution

Two enhancements are necessary:

- Checkpointing over time steps using `revolve`
- Differentiating iterative solver of underlying $Ax = b$ systems
  by hand accelerates and ensures correctness of reverse solution
  $Ab' = x'$.

Both require the piecewise differentiation of the code.
This turned out to be problematic.
Advanced solution: One easy problem

FADBAD has no function for reverse differentiation with a seed vector. This can be solved with a wrapper function as follows: To evaluate $w \nabla f$ with a (long) weight vector $w$ and a (huge) Jacobian $\nabla f$ efficiently, make FADBAD compute the gradient of the scalar function

$$\tilde{f} = \sum_i w_i f_i.$$

But I could test this with the real code: FADBAD appears to require all intermediate variables to be destroyed, or to differentiate them all. The first needs major rewrites, the second is slow and also difficult to implement.
ADOL–C has the LIFO requirement, which is not met by the code. Memory usage and tapes get ever larger, the differentiation ever slower. Solved by creating a slightly modified ADOL–C, removing the LIFO requirement.

CppAD wants all dependent and independent variables in one vector each. Possible with additional copying in wrapper functions. Sacado crashes for an unknown reason.

Results for Checkpointing with LIFO-modified ADOL–C:

<table>
<thead>
<tr>
<th>$n_t$</th>
<th>$t_\nabla/t_J$</th>
<th>Mem (MB)</th>
<th>$t_\nabla/t_J$</th>
<th>Mem (MB)</th>
<th>$t_\nabla/t_J$</th>
<th>Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>5.2</td>
<td>25</td>
<td>4.63</td>
<td>26</td>
<td>5.17</td>
<td>26</td>
</tr>
<tr>
<td>30</td>
<td>25</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ADOL–C’s LIFO requirement: example program

```cpp
#include <adolc/adolc.h>
#include <valarray>
#include <list>

int main() {
    std::list<adouble> li;
    li.push_back(0);
    for (size_t i = 0; i < 100000; ++i) {
        std::valarray<adouble> ve(100000);
    }
}
```

This program exits because of a lack of memory. Note you would not expect this if data type was regular `double`. 

Johannes Willkomm, Martin B"ucker

AD-Tools from a user’s perspective
ADOL–C’s LIFO requirement: solutions

The LIFO-requirement is documented in the ADOL–C manual. What are the solutions?

1. Make your code adhere to the LIFO-requirement
2. Remove the LIFO requirement from ADOL–C, using a modified Pool-Allocator [Stroustrup, 2000]

Solution 1: How to spot the non-LIFO instruction(s)? One suffices to break the LIFO-ness of the program.
Solution 2: What are the costs?
**ADOL–C’s LIFO requirement: our solution**

**Table:** Properties of ADOL–C’s current and proposed memory manager and of the Pool-Allocator for managing $N$ entries.

<table>
<thead>
<tr>
<th></th>
<th>ADOL–C</th>
<th>Pool-Allocator</th>
<th>proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>$N + c$</td>
<td>$N + c$</td>
<td>$1.5N + c$</td>
</tr>
<tr>
<td>Time alloc</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Time free</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>store-array saved</td>
<td>part</td>
<td>–</td>
<td>whole</td>
</tr>
<tr>
<td>LIFO</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>support alloc(n)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
ADOL–C’s LIFO requirement: our solution

- Memory usage $1.5N$ could possibly be reduced to $N$ if destruction of adoubles was recorded.
- New technique does not support allocating multiple entries at once → classes avector and amatrix are not supportable.
- New technique does not work like a stack → it needs $O(N)$ time to find out which entries in store array are in use. We currently simply save everything, used or unused.
Conclusion

Attempt to present experiences with C++ OO Tools from a novice, non-C++-expert user’s perspective.

- Four C++ AD-Tools presented
- Only few weaknesses in differentiating individual expressions
- Problems with application to large existing code
- Black-Box differentiation of large code requires double and respective active type to be fully equivalent semantically (Default-Constructor)
- Checkpointing without rewriting too much of the code still only working with our modified ADOL–C
Bjarne Stroustrup
The C++ Programming Language
2000, 4. revised edition, Addison-Wesley, Boston

Andreas Griewank, David Juedes and Jean Utke,
ADOL–C, A Package for the Automatic Differentiation of
Algorithms Written in C/C++,

Bradley M. Bell,
CppAD, A Package for C++ Algorithmic Differentiation,
2003, http://www.coin-or.org/CppAD/


Andreas Griewank and Andrea Walther,
Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation,
2000, ACM Transactions on Mathematical Software Vol. 26