Fast Reverse-Mode Automatic Differentiation using Expression Templates in C++

ROBIN J. HOGAN, University of Reading

Gradient-based optimization problems are encountered in many fields, but the associated task of differentiating large computer algorithms can be formidable. The operator-overloading approach to performing reverse-mode automatic differentiation is the most convenient for the user but current implementations are typically 10-35 times slower than the original algorithm. In this paper a fast new operator-overloading method is presented that uses the *expression template* programming technique in C++ to provide a compile-time representation of each mathematical expression as a computational graph that can be efficiently traversed in either direction. Benchmarking with four different numerical algorithms shows this approach to be 2.6–9 times faster than current operator-overloading libraries, and 1.3–7.7 times more efficient in memory usage. It is typically less than 4 times the computational cost of the original algorithm, although poorer performance is found for all libraries in the case of simple loops containing no mathematical functions. An implementation is freely available in the *Adept* C++ software library.

Categories and Subject Descriptors: G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation—*Automatic differentiation*; G.1.6 [**Numerical Analysis**]: Optimization—*Gradient measures*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Adjoint code, quasi-Newton, template metaprogramming, Jacobian matrix

ACM Reference Format:
Robin J. Hogan. 2014. Fast reverse-mode automatic differentiation using expression templates in C++. ACM Trans. Math. Softw. 40, 4, Article 26 (June 2014), 16 pages.
DOI: http://dx.doi.org/10.1145/2560359

1. INTRODUCTION

Many problems in the physical sciences amount to the optimization problem of finding the vector $\mathbf{x}$ of n real-valued state variables that minimizes a real-valued function $J(\mathbf{x})$. When n is large and J is a well-behaved differentiable function, the quasi-Newton method (e.g., Liu and Nocedal [1989]), conjugate gradient or other gradient method is typically used. These methods make repeated estimates of $\mathbf{x}$, at which they require not only the value of J but also the vector of gradients $\partial J/\partial \mathbf{x}$ in order to know in which direction to search next. Numerical gradients, found by perturbing each element of $\mathbf{x}$ a small amount and recalculating J, are subject to round-off error as well as being extremely inefficient for large n. On the other hand, coding the *adjoint model* to compute $\partial J/\partial \mathbf{x}$ can be a formidable task; not only must each line of code be differentiated, but also the gradients must be passed back through the adjoint code ("reverse mode" differentiation) requiring intermediate states to be stored. For large codes, this presents a memory problem and check-pointing may be needed to recompute intermediate values.

This work benefited from ESA Grant 4000104528/11/NL/CT and NERC Grant NE/H003894/1.
Author's address: Department of Meteorology, University of Reading, Earley Gate, PO Box 243, Reading, RG6 6BB, United Kingdom; email: r.j.hogan@reading.ac.uk.

DOI: http://dx.doi.org/10.1145/2560359

Hand-written adjoint codes are generally the most efficient, typically involving 2–4 times the computational cost of the original algorithm, but they are also very time consuming to write and debug. This has motivated the development of *automatic differentiation* for generating tangent-linear and adjoint codes (i.e., forward- and reverse-mode, respectively) automatically from the original algorithm code [Griewank and Walther 2008]. Two main approaches exist. The first employs a *source-to-source compiler* (e.g., Giering and Kaminski [1998] and Voßbeck et al. [2008]), to analyze the original source and produce the source code for the equivalent adjoint model that may then be compiled in the usual way. While the resulting executable can have efficiencies approaching hand-coding, there are restrictions on the language features that may be used in the original code; C++ templates are not supported by any current tool and many tools have incomplete support for other C++ and Fortran-2003 features such as object orientation with inheritance. The second, more recent approach employs *operator overloading* (available in many modern languages) whereby those variables for which a gradient is needed are declared as a new "active" type and the standard arithmetic operators and mathematical functions are overloaded such that when applied to this new type they automatically perform the additional operations behind the scenes to ensure the gradients are properly computed. Provided all necessary operators are defined, this approach is compatible with all available language features and is much more convenient for the user. Forward-mode automatic differentiation is straightforward to implement in this way, with the new active type being composed of not only the value of the variable but also its derivative with respect to one or more independent input variables [Griewank and Walther 2008]. The reverse mode is much more difficult to implement since in the forward pass of the algorithm it is necessary to create a record in memory of each statement, and then traverse this in reverse order. Examples are ADOL-C [Griewank et al. 1996], CppAD [Bell 2007], Sacado [Gay 2005] and FADBAD [Bendtsen and Stauning 1996], but the best reported performance is a factor of 10 slower than the original algorithm [Gay 2005], while Willkomm and Bücker [2007] reported factors between 23 and more than 100.

This article demonstrates that reverse-mode automatic differentiation using operator overloading can be implemented in C++ such that it is typically only 2.7–4 times slower than the original algorithm, and therefore comparable to hand-coding. This is achieved by the use of *expression templates,* a technique introduced by Veldhuizen [1995] originally to enable C++ statements involving operations on arrays to be compiled to avoid the generation of temporary arrays. Expression templates were used by Aubert et al. [2001] to accelerate an operator-overloading implementation of forward-mode automatic differentiation, and since incorporated into Sacado [Bartlett et al. 2006]. They have also been used to accelerate the differentiation of individual expressions via "expression-level reverse mode" within an overall forward-mode automatic differentiation framework [Phipps and Pawlowski 2012]. The present paper is novel in its application of the technique to full reverse-mode, and also its efficient storage of the differential information that contributes significantly to the performance.

A C++ library "Adept" (Automatic Differentiation using Expression Templates) has been written by the author to demonstrate the technique.[1] Section 2 introduces the user interface, along with a simple set of operations to illustrate the technique throughout the paper. Section 3 outlines the way that the gradient information is stored in memory. Section 4 describes how expression templates are used to create this gradient information efficiently from C++ statements, and Section 5 describes how this information is parsed to compute adjoints and Jacobian matrices. Then, in Section 6, the speed of the new technique is compared to hand-written adjoint codes and to existing

[1]"Adept" is available from `http://www.met.reading.ac.uk/clouds/adept/` under a free-software license.

operator-overloading implementations of reverse-mode automatic differentiation, for four different algorithms. The speed of computing the Jacobian is also compared.

2. USER INTERFACE

In this section, we illustrate how simply automatic differentiation may be invoked from a user perspective. The user need have no knowledge of C++ templates, let alone expression templates, to use the library. Consider, the following algorithm to calculate one dependent variable y from two independent variables x_0 and x_1, indicated by the sequence of statements in the left-hand column:

$$y \leftarrow 4; \qquad \delta y \leftarrow 0; \tag{1}$$

$$s \leftarrow 2x_0 + 3x_1^2; \qquad \delta s \leftarrow 2\delta x_0 + 6x_1\delta x_1; \tag{2}$$

$$y \leftarrow y\sin(s); \qquad \delta y \leftarrow \sin(s)\delta y + y\cos(s)\delta s. \tag{3}$$

Note that "←" indicates the assignment operator. Obviously, this algorithm could be contracted into a single statement, but it has been contrived for didactic purposes, with the three statements representing cases that must treated carefully: a constant on the right-hand side, different variables on the left- and right-hand sides, and the same variable on the left- and right-hand sides. The equivalent differentiated statements are shown in the right-hand column, where formally δa for any variable a is defined as a function of the n independent variables as $\delta a = \sum_{i=0}^{n-1}(\partial a/\partial x_i)\delta x_i$. The three expressions (1)–(3) will be used to illustrate concepts throughout this article. The equivalent adjoint algorithm consists of the following sequence of statements where δ^* is defined such that the adjoint $\delta^* a = \partial J/\partial a$:

$$\delta^* s \leftarrow \delta^* s + y\cos(s)\delta^* y; \tag{4}$$

$$\delta^* y \leftarrow \sin(s)\delta^* y; \tag{5}$$

$$\delta^* x_0 \leftarrow \delta^* x_0 + 2\delta^* s; \tag{6}$$

$$\delta^* x_1 \leftarrow \delta^* x_1 + 6x_1\delta^* s; \tag{7}$$

$$\delta^* s \leftarrow 0; \tag{8}$$

$$\delta^* y \leftarrow 0. \tag{9}$$

This sequence is derived following the well-known rules of adjoint coding (e.g., Giering and Kaminski [1998]) that will not be repeated here. It can be seen that these expressions are in reverse order to those of the original algorithm: (4)–(5) correspond to (3), (6)–(8) correspond to (2), and (9) corresponds to (1).

The original algorithm could be implemented in C or C++ trivially as follows.

```
double algorithm(const double x[2]) {
  double y = 4.0;
  double s = 2.0*x[0] + 3.0*x[1]*x[1];
  y *= sin(s);
  return y;
}
```

From a user perspective, using Adept to create an equivalent adjoint function is very similar to using ADOL-C, CppAD or Sacado: we define all active variables as of a special type `adouble`. Since all variables in this algorithm are active, this would involve a simple search-and-replace of `double` with `adouble` in the code above. The code to compute the adjoint could then look like this.

```
double algorithm_ad(const double x_val[2], // Input values
                    double* Y_ad,          // Input-output adjoint
                    double x_ad[2]) {      // Output adjoint
```

```
  using namespace adept;                    // Import Stack and adouble from adept
  Stack stack;                              // Where differential information is
                                               stored
  adouble x[2] = {x_val[0], x_val[1]};      // Initialize adouble inputs
  stack.new_recording();                    // Start recording derivatives
  adouble Y = algorithm(x);                 // Version overloaded for adouble args
  Y.set_gradient(*Y_ad);                    // Load the input-output adjoint
  stack.reverse();                          // Run the adjoint algorithm
  x_ad[0] = x[0].get_gradient();            // Extract the output adjoint for x[0]
  x_ad[1] = x[1].get_gradient();            //   ...and x[1]
  *Y_ad   = Y.get_gradient();               // Input-output adjoint has changed too
  return Y.value();                         // Return result of simple computation
}
```

More commonly the `Stack` object would be declared once in a higher level part of the program, enabling memory allocated in the first call to `algorithm_ad` to be reused in subsequent calls, avoiding the overhead of allocating memory each time.

3. MEMORY STRUCTURES

In this section, we consider how the information necessary to compute the adjoint may be stored efficiently, which is as important for the speed of the algorithm as the use of expression templates. It is not possible to store the gradients within each `adouble` object because many such objects will be local to individual functions and loops, and so become undefined when the function exits or when a loop finishes a cycle and therefore no longer available when the adjoint needs to be computed. Instead, each `adouble` object stores an index into the array of gradients that will be created when needed for the adjoint computation. When an `adouble` object is created, its constructor requests the next free index from the currently active `Stack` object, accessed via a global pointer that is defined as thread-local in multi-threaded applications.[2] When an `adouble` object's destructor is called it informs the `Stack` object so that the index can be used again. By default, indices are stored as 4-byte unsigned integers, sufficient to index 32 GiB of double-precision numbers.

The storage of the gradient information is best illustrated by considering the differentiated statements in the right-hand column of (1)–(3). In this case, we have three *statements,* each of which has an *expression* on the right-hand side composed of zero or more *operations*. This information is stored in a "statement stack" and an "operation stack", and Figure 1 illustrates what would be stored as a consequence of the `algorithm` and `algorithm_ad` function listings in Section 2 (with `double` replaced by `adouble` in `algorithm`). The order in which `adouble` objects are created leads to the variables `x[0]`, `x[1]`, `Y`, `y` and `s` being assigned indices 0–4, respectively. Equation (1), $\delta y \leftarrow 0$, is represented by the 0th entry in the statement stack: δy is represented by the index 3, and the zero on the right-hand side of the statement is indicated by the fact that the indices to the first operation of the 0th and 1st statements are the same. Equation (2) represented by the next entry in the statement stack, results in three operations on the right-hand side represented by entries 0–2 in the operation stack. The reason that there are three rather than two entries as in (2) is that the code treats the two occurrences of `x[1]` in the term `3*x[1]*x[1]` independently, and so produces two entries with a multiplier of $3x_1$ rather than one of $6x_1$. Unfortunately, it seems impossible to detect whether a

[2]Using a global but thread-local pointer to the current storage object is more efficient than in some operator-overloading implementations of automatic differentiation where each active variable keeps its own pointer or reference to the current storage object, which is then passed from one variable to another in each mathematical operation. It is also thread-safe, unlike others that simply store the information in ordinary global variables.

Statement Stack:	Index to LHS gradient	Index to first operation
0	3 (δy)	0
1	4 (δs)	0
2	3 (δy)	3
3	2 (δY)	5
	⋮	⋮

Operation Stack:	Multiplier	Index to RHS gradient
0	2.0	0 (δx_0)
1	$3.0x_1$	1 (δx_1)
2	$3.0x_1$	1 (δx_1)
3	$\sin(s)$	3 (δy)
4	$y\cos(s)$	4 (δs)
5	1.0	3 (δY)
	⋮	⋮

Fig. 1. Illustration of the contents of the statement and operation stacks (stored within a `Stack` object) for the algorithm given by (1)–(3). Note that all variables would normally be stored as unsigned 4-byte integers, except for the multiplier in the operation stack which would be of type `double`.

variable appears twice on the right-hand side of a statement at compile time, and while it would be possible to parse the stack at run-time to combine operations such as 1 and 2, this would be more computationally costly than simply performing one extra operation. Equation (3) leads to entries 3 and 4 in the operation stack. Finally, `y` is returned from the `algorithm` function and copied into `Y`, leading to entry 3 in the statement stack and entry 5 in the operation stack, which represents the simple assignment $\delta Y \leftarrow \delta y$. Section 5 describes how the adjoint is calculated from this information.

4. IMPLEMENTATION USING EXPRESSION TEMPLATES

It is now demonstrated how expression templates may be used to build the stacks shown in Figure 1. Expression templates were originally proposed as a means to speed up mathematical expressions involving operations on arrays [Veldhuizen 1995] and this technique is now widely used in C++ vector/matrix libraries, including the GNU implementation of the Standard Template Library (STL) "`valarray`" class. The idea is that mathematical operators are overloaded not to return the result of the operation immediately, but to return an object derived from a base "expression" class whose template parameters describe the type of expression. The key capability this adds for the purposes of this article is that information can be propagated not only from the most nested part of the expression outwards, as in the case of ordinary operator overloading, but also from the outermost expression in to the most nested. Moreover, the compiler can inline the function calls that mediate information between expressions and subexpressions resulting in very efficient compiled code.

To illustrate how this works, we consider the line `y *= sin(s)` from the simple algorithm in Section 2. The "`*=`" operator is implemented as if the line had been written `y=y*sin(s)`, which is a case that must be treated carefully since it involves `y` appearing on both sides of the assignment. Figure 2(a) depicts a computational graph of this expression, where the arrows indicate the conventional flow of information from the most nested subexpression outwards eventually to be returned to the variable on the left-hand side of the statement. Expression templates are implemented by overloading

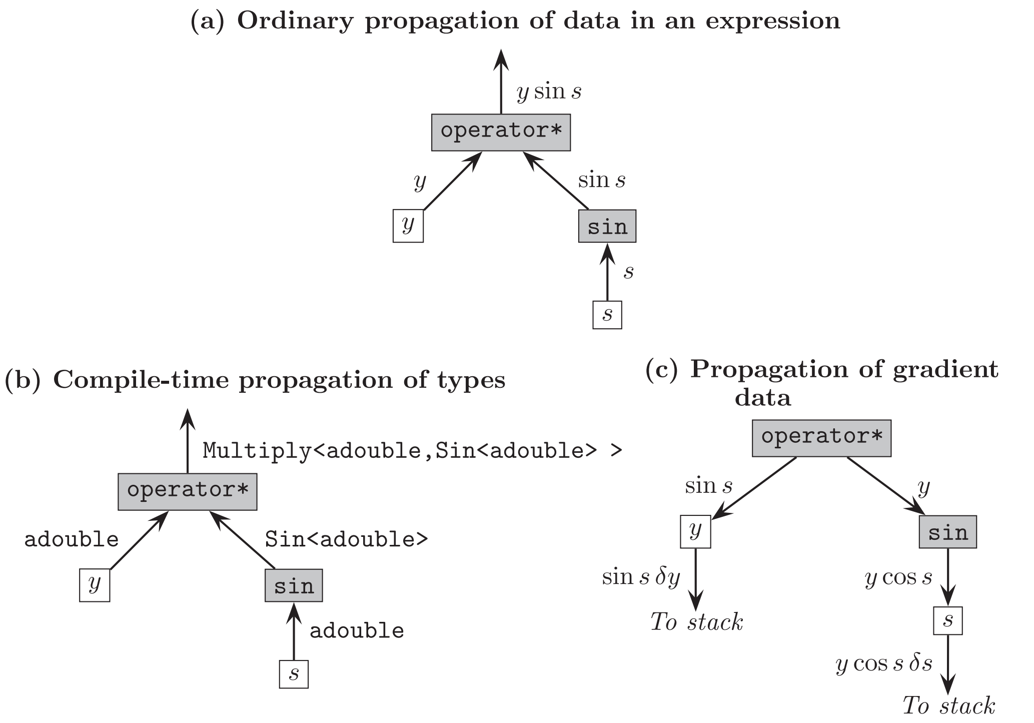

Fig. 2. Computational graphs representing the propagation of data or types for the expression `y*sin(s)`: (a) ordinary propagation of data from the most nested subexpression up to the main expression; (b) compile-time propagation of types in the Adept library resulting in an expression template representing the full expression; (c) propagation of gradient data when the `calc_gradient` member function of the `Multiply<adouble,Sin<adouble> >` object is called.

the operator "*" and the function `sin` such that they take `Expression` objects as arguments and return `Multiply` and `Sin` objects, respectively, with template arguments representing the type of the arguments to this operator and function. Figure 2(b) indicates the types that are passed up the chain and which are known at compile time. Thus the expression on the right-hand side of the statement resolves to an object of type `Multiply<adouble,Sin<adouble> >`. The classes `adouble`, `Multiply` and `Sin` are all derived from a common base class `Expression`, which is what allows them to be used as arguments to arithmetical operators and mathematical functions. This behavior is implemented using the concept of *static polymorphism,* where an object of one type can masquerade as another, but in a way that is known at compile time, thus avoiding the extra overhead of virtual function look-up.

Why is this design useful? Well it enables the operation stack in Figure 1 to be populated efficiently by propagating gradient information in the other direction. Essentially, we wish to build up the differential statement by applying the chain rule of differentiation to a statement, and letting each object that represents an elemental function contribute a derivative to the chain. Thus, an object representing a unary function $f(a)$ (e.g., `Sin` in this example) would implement a member function `calc_gradient` that can take a real number w representing the numerical value of the chain of derivatives up to that point, multiply it by $\partial f/\partial a$ and pass the result to the object representing its argument. In this example, the `Sin` object is passed the value of y and then passes $y \cos s$ to the `adouble` object representing s. Likewise, objects representing binary operators and functions $f(a, b)$ (e.g., the "*" operator) would implement a `calc_gradient` member function that passes $w\partial f/\partial a$ to a and $w\partial f/\partial b$ to b. The `calc_gradient` function of an `adouble` object behaves differently: it simply adds an entry to the operation stack consisting of multiplier w and its own index to its gradient. It can be seen from Figure 2(c) that this results in the correct information being added to the operation stack in the case of `y=y*sin(s)`.

The implementation is now described. The base `Expression` class is defined as follows.

```
template<class A>
struct Expression {
  const A& cast() const { return static_cast<const A&>(*this); }
  double value() const { return cast().value(); }
  void calc_gradient(Stack& stack) const { cast().calc_gradient(stack); }
  void calc_gradient(Stack& stack, const double& multiplier) const
    { cast().calc_gradient(stack, multiplier); }
}
```

It can be seen that `Expression` takes one template parameter `A`, and that all its member functions simply use a `static_cast` to convert the `Expression` object into type `A` and then call the equivalent function for that type. The meaning of each member function will be described later. The derived class `adouble` is then defined as follows.

```
struct adouble : public Expression<adouble> {
  enum { n_active_vars = 1 };
  // ...constructors and other member functions here...
  template<class R>
  adouble& operator=(const Expression<R>& rhs)
    { val_ = rhs.value();                    // Compute value of the expression
      _stack->push_lhs(gradient_offset_);    // Add an entry to statement stack
      _stack->check_space(R::n_active_vars); // Ensure is space for operations
      rhs.calc_gradient(*_stack);            // Add gradient data to op. stack
      return *this; }
  double value() const { return val_; }
  void calc_gradient(Stack& stack) const { stack.push_rhs(1.0, gradient_offset_); }
  void calc_gradient(Stack& stack, const double& multiplier) const
    { stack.push_rhs(multiplier, gradient_offset_); }
 private:
  double val_;
  const unsigned int gradient_offset_;
}
```

This is an example of the "Curiously Recurring Template Pattern": `adouble` inherits from `Expression<adouble>`. This facilitates the required behavior: when an `adouble` object is used in an expression requiring `Expression` arguments, the inheritance relationship enables it to be interpreted as an `Expression<adouble>` object, and the behavior of each member function in the definition of `Expression` to defer to its template parameter (in this case `adouble`) means that it behaves as an `adouble` object. Note that in practice `adouble` could be implemented as `active<double>`, that is, there could be a generic active type that could be specialized for underlying floating-point variables including `float` and `complex<double>`. This would mean that `Expression` would need to take another template argument and promotion rules would be required to ensure that an `active<double>` object multiplied by an `active<float>` results in an expression in terms of `double` objects. Since this extra layer of complexity detracts from the main discussion, we have assumed all active variables have an underlying `double` type in this article.

The `Multiply` and `Sin` classes are implemented as follows.

```
template <class A>
struct Sin : public Expression<Sin<A> > {
  enum { n_active_vars = A::n_active_vars };
  Sin(const Expression<A>& a) : a_(a.cast()), result_(sin(a_.value())) { }
  double value() const { return result_; }
  void calc_gradient(Stack& stack) const {a_.calc_gradient(stack, cos(a_.value()));}
  void calc_gradient(Stack& stack, const double& multiplier) const
    { a_.calc_gradient(stack, cos(a_.value())*multiplier); }
 private:
  const A& a_;     // A reference to the argument of the sin function
  double result_;  // The numerical value of sin(a)
};

template <class L, class R>
struct Multiply : public Expression<Multiply<L,R> > {
  enum { n_active_vars = L::n_active_vars + R::n_active_vars };
  Multiply(const Expression<L>& l, const Expression<R>& r)
    : l_(l.cast()), r_(r.cast()) { }
  double value() const { return l_.value() * r_.value(); }
  void calc_gradient(Stack& stack) const
    { l_.calc_gradient(stack, r_.value());
      r_.calc_gradient(stack, l_.value()); }
  void calc_gradient(Stack& stack, const double& multiplier) const
    { l_.calc_gradient(stack, r_.value()*multiplier);
      r_.calc_gradient(stack, l_.value()*multiplier); }
 private:
  const L& l_;     // A reference to the left argument
  const R& r_;     // A reference to the right argument
};
```

Note that these two classes take template parameters that define the types of their arguments. They also store references to their arguments in order they can propagate information to their arguments as shown in Figure 2(c). The `sin` function and "*" operator are overloaded to return these two classes.

```
template <class A> inline
Sin<A> sin(const Expression<A>& a) { return Sin<A>(a); }

template <class A, class B> inline
Multiply<A,B> operator*(const Expression<A>& a, const Expression<B>& b) {
  return Multiply<A,B>(a, b);
}
```

We are now in a position to outline what happens when `y=y*sin(s)` is invoked. First, the overloaded `sin` function returns an object of type `Sin<adouble>`, whose constructor is passed a constant reference to the `adouble` object `s`, which it stores. It also stores the numerical value of sin(s) in `result_` to avoid it being recalculated if its member function `value` is called more than once; this optimization was also used by Phipps and Pawlowski [2012] for forward-mode automatic differentiation. Next, the "*" operator is invoked, which returns a `Multiply` object; this time constant references to both arguments are stored within `Multiply`. Ultimately, the right-hand side of the statement resolves to a `Multiply<adouble,Sin<adouble> >` object. The `operator=` member function of `y` is then invoked on the expression (labeled `rhs` in the definition of `adouble` above). After obtaining the actual value of the expression via `rhs.value()`, it passes to the currently active `Stack` object (accessed via a thread-local but global pointer `_stack`) the index to the location of $\delta^* y$ (stored as `gradient_offset_`). The stack is informed (via the `Stack::check_space` function) how many operations will appear on the right-hand side of the differential statement, in order that enough space can be allocated on the operation stack. This is simply the number of `adouble` objects that appear on the right-hand side of the statement and is determined at compile time when the `n_active_vars` enumeration is computed for each sub-part of the expression.

This information is sufficient to add entry 2 to the statement stack depicted in Figure 1. The final step is to add the gradient terms to the operation stack, which is achieved by calling `rhs.calc_gradient`. Each object then calls the `calc_gradient` function of its stored arguments in order to propagate the gradient information through the entire tree, as shown in Figure 2(c). For efficiency, each class has two `calc_gradient` functions, one with a multiplier and the other without (corresponding to the multiplier being assumed to be one). In some expressions this avoids a redundant multiplication by one. When the `calc_gradient` function of an `adouble` object is called, it instead simply pushes the differential expression on to the operator stack resulting in entries 3 and 4 in Figure 1.

Thus, the efficient memory layout described in Section 3 can be built at the time the statements are parsed. While the reader may be struck by the large number of function calls that are necessary to perform a simple calculation in order to place the entries on the relevant stacks, it should be stressed that the compiler is able to inline all of these function calls leaving little more than the underlying computations. For a fully functional library that can differentiate any code, it is necessary to overload all arithmetic operators, comparison operators and mathematical functions. This has been implemented in the Adept library.

5. ADJOINT AND JACOBIAN COMPUTATION

We now describe how the adjoint calculations proceed given a particular statement and operation stack illustrated in Figure 1. The first time the `Stack::set_gradient` function is called (see the listing for the `algorithm_ad` function in section 2), a real-valued vector is allocated of just the right size to store the gradients. One or more calls to `set_gradient` then set the gradients of the dependent variables. The `Stack::reverse` function then performs the adjoint computation by looping in reverse through the statement stack and executing the adjoint of each differential statement.

A general differential statement may be written as

$$\delta y \leftarrow \sum_{i=0}^{k-1} w_i \delta x_i, \tag{10}$$

where in the terminology of Figure 1, δy is the left-hand side gradient, k is the number of terms on the right-hand side (which may be zero), w_i is a multiplier equal to the partial derivative $\partial y / \partial x_i$, and δx_i is an right-hand side gradient. The recipe for forming the equivalent adjoint statements is usually described in the literature in terms of taking the transpose of the local Jacobian matrix. However, it turns out that the equivalent adjoint statements may be expressed by a very simple algorithm given by the following pseudocode:

```
$a \leftarrow \delta^* y$,
$\delta^* y \leftarrow 0$,
If $a \neq 0$ then for $i$ between 0 and $k-1$ do:
    $\delta^* x_i \leftarrow \delta^* x_i + w_i a$.                (11)
```

This is applied to all the statements in the stack in reverse order, and may be coded in a few lines of C or C++. The use of a temporary variable a was proposed by Griewank and Walther [2008] in their Table 4.6, and ensures the correct behavior in the three contrasting cases exemplified by (1)–(3). Equation (1) has no active variables on the right-hand side of the statement, so the statement stack in Figure 1 has the same number for elements 0 and 1 of the list of indices to the operations, and hence $k = 0$. This leads to the loop in (11) not being executed so the corresponding gradient is correctly set to zero as in (9). Equation (2) has different active variables on the left- and right-hand sides, so the gradient corresponding to the left-hand side of the statement is not changed from zero by any action within the loop and so (8) is correctly applied. Finally, (3) has the same variable on both sides of the statement, which means that the gradient $\delta^* y$ is the same as one of the $\delta^* x_i$ gradients in (11). Therefore, it is set to zero before the loop but is then modified within the loop, resulting in (5). The test to see whether a is nonzero removes many unnecessary multiplications and is found to speed up the reverse pass by 20–50%.

We may also use the stack information to compute the full $m \times n$ Jacobian matrix, that is, the rate of change of each of the m dependent variables $\mathbf{y}$ with respect to each of the n independent variables $\mathbf{x}$, sometimes written as $\partial\mathbf{y}/\partial\mathbf{x}$. If $n > m$, then this is fastest achieved by m executions of (11), each time setting the input values of $\partial J/\partial\mathbf{y}$ to zero except for a single value that is set to 1. Each iteration results in a row of the Jacobian matrix. If $n < m$ then the Jacobian may be calculated by n executions of the forward "tangent linear" code, which is simply a direct implementation of (10) for each statement in the stack. Each iteration then yields a column of the Jacobian matrix. In practice, to facilitate compiler optimizations such as loop unrolling and vectorization instruction sets (such as SSE2 on Intel hardware), "strips" of several rows or columns of the Jacobian are computed at once. The more common approach to computing "$n < m$" Jacobians would be using a forward-mode automatic differentiation library that replaces each intermediate scalar with an object containing not only the value of the scalar but also a vector containing the derivatives of the scalar with respect to each of the independent variables. This approach avoids the memory expense of storing the stack needed for reverse mode, but on the other hand much more memory is needed to store working variables for large n. It will be shown in the next section that this is typically no more efficient than the use of a stack as described in this article.

6. BENCHMARKING

In this section, we compare the speed of reverse-mode automatic differentiation as implemented in the Adept library with the speed of the original algorithm, a hand-coded adjoint algorithm, and the reverse-mode automatic differentiation of the ADOL-C, CppAD and Sacado libraries [Griewank et al. 1996; Bell 2007; Gay 2005]. We further compare the speed of computing Jacobian matrices both in forward and reverse mode. Memory usage and compile times are also compared.

6.1. Description of Test Algorithms

The four algorithms we use for testing consist of two simple cases for which the code will be shown here, and two more complex real-world algorithms. The simple algorithms each solve the 1D linear advection equation $\partial q/\partial t = -u\partial q/\partial x$ in a periodic domain, where q is the quantity being advected, t is time, x is the spatial coordinate and u is the velocity, which is constant with x. A uniform grid of 100 points is used, which includes two dummy points to deal with the periodic boundary conditions. The first algorithm uses the linear scheme of Lax and Wendroff [1960] as follows.

```
#define NX 100
void lax_wendroff(int nt, double c, const adouble q_init[NX], adouble q[NX]) {
  adouble flux[NX-1];                          // Fluxes between boxes
  for (int i=0; i<NX; i++) q[i] = q_init[i]; // Initialize q
  for (int j=0; j<nt; j++) {                 // Main loop in time
    for (int i=0; i<NX-1; i++) flux[i] = 0.5*c*(q[i]+q[i+1]+c*(q[i]-q[i+1]));
    for (int i=1; i<NX-1; i++) q[i] += flux[i-1]-flux[i];
    q[0] = q[NX-2]; q[NX-1] = q[1];          // Treat boundary conditions
  }
}
```

where nt is the number of timesteps to run, c is the Courant number (i.e., the velocity expressed as the fraction of a spatial step traversed in a timestep), q_init is the initial distribution of q (the 100 independent variables) and q is the final distribution that is output (the 100 dependent variables).

The second algorithm solves the same problem but using the nonlinear scheme of Toon et al. [1988], which was designed to treat concentrations of a tracer that may vary by orders of magnitude across the domain and treats q as varying exponentially between gridpoints. It is implemented simply by replacing the sixth line of the previous listing with the following.

```
    for (int i=0; i<NX-1; i++) flux[i] = (exp(c*log(q[i]/q[i+1]))-1.0)
                                         * q[i]*q[i+1] / (q[i]-q[i+1]);
```

While sharing the same structure, these two algorithms are very different in terms of performance because of the presence of transcendental functions only in the second case; this is discussed in detail in Section 6.2.

The third and fourth algorithms are real-world radiative transfer models that simulate the measurements made by a satellite-borne atmospheric lidar instrument for probing clouds.[3] Both models predict the profile of apparent backscatter (the dependent variables) at $N = 50$ equally spaced height intervals that would be measured given an input profile described by $7N$ active independent variables that describe the properties of the atmosphere by seven values at each of the N height intervals. The third algorithm uses the Photon Variance-Covariance (PVC) method of Hogan [2008] and computes the contribution to the measured signal from light that has been scattered multiple times, but only by a small angle, between the lidar transmitter and receiver. It uses only five of the seven independent variables at each range interval and integrates four coupled ordinary differential equations forward in space. It has a computational cost proportional to N. The fourth algorithm uses the Time-Dependent Two Stream (TDTS) method of Hogan and Battaglia [2008] and computes the contribution to the measured signal from light that has been scattered multiple times by any angle. It uses only four of the seven independent variables at each range gate and integrates four coupled partial differential equations forward in time for $2N$ timesteps. Its computational cost is proportional to N^2. Both of these last two algorithms have been coded in C using C-style arrays, and can be adapted to Adept, ADOL-C, CppAD and Sacado, and compiled in C++ with virtually no code modifications beyond relabeling most of the double objects as adouble.[4]

[3]The radiative transfer models form part of the *Multiscatter* package freely available from http://www.met.reading.ac.uk/clouds/multiscatter.

[4]The *Multiscatter* package makes extensive use of variable-length arrays, defined in the C99 standard but not part of C++; therefore, we rely on the GNU compiler extension to permit variable-length arrays in C++. It would be straightforward to use std::vector<adouble> instead.

Table I. Speed Comparison for Reverse-Mode Differentiation Applied to the Advection Schemes of Lax and Wendroff [1960] (**LW**) and Toon et al. [1988] (**Toon**) using 2000 Timesteps, and to the Radiative Transfer Models based on the Photon Variance-Covariance (**PVC**) and Time-Dependent Two Stream (**TDTS**) Methods for Profiles of Atmospheric Variables at $N = 50$ Height Intervals, Taken from Version 1.2.10 of the *Multiscatter* Package

Algorithm	LW	Toon	PVC	TDTS
Time of original algorithm	0.60 ms	13.1 ms	0.013 ms	0.85 ms
Relative cost of computing the adjoint				
Hand coded	**2.1** (1.0 + 1.1)	**2.3** (1.0 + 1.3)	**3.0** (1.0 + 2.0)	**3.5** (1.0 + 2.5)
Adept (thread unsafe)	**32** (21 + 11)	**2.7** (2.1 + 0.6)	**3.7** (2.8 + 0.9)	**3.8** (2.6 + 1.2)
Adept (thread safe)	**33** (22 + 11)	**2.8** (2.2 + 0.6)	**3.8** (2.9 + 0.9)	**3.9** (2.7 + 1.2)
ADOL-C	**106** (81 + 25)	**9.2** (7.2 + 2.0)	**25** (18 + 7)	**20** (15 + 5)
CppAD	**214** (120 + 45 + 49)	**16** (8 + 4 + 4)	**29** (15 + 7 + 7)	**34** (17 + 8 + 9)
Sacado::Rad	**238** (125 + 113)	**15** (8 + 7)	**10** (7 + 3)	**30** (16 + 14)

The tests were performed on a 2.5-GHz 64-bit Intel E5200 Pentium with a 2-MB cache running Linux 2.6.27 (openSUSE 11.1). The versions used were Adept 1.0, ADOL-C 2.3.0, CppAD 20120101.3 and Sacado::Rad (the reverse-mode version of Sacado) from Trilinos 11.0.3. All algorithms were compiled using the 64-bit GNU C++ compiler version 4.3.2, with SSE2 instructions enabled by default, and optimization arguments `-O3 -DNDEBUG`, the latter which eliminates error checking for CppAD and approximately halves its execution time. All algorithms were run at least a thousand times with memory reuse to minimize the overhead of memory allocation, where possible, although the tapes and stacks were recomputed each time. ADOL-C was configured to ensure that all data were kept in memory rather than temporary files being written to disk. "Relative cost" is the execution time divided by the time to run the original algorithm. In parentheses these times are split into the time of the forward pass (first number), the time of the reverse pass (last number), and in the case of CppAD, the time to generate the `ADFun` object ready for the reverse pass (middle number).

6.2. Results

The results of the comparisons in terms of computational cost divided by the cost of the original algorithm are shown in Table I. All algorithms have been repeated at least a thousand times, with available options to maximize memory reuse enabled to eliminate time spent dynamically allocating memory. However, all tapes and stacks have been recomputed each time to best mimic real-world applications where different independent input variables may lead to different paths of execution. Considering the two advection algorithms first (Lax and Wendroff 1960; Toon et al. 1988), run for 2000 timesteps, we see that, in both cases, Adept is 3.3–7 times faster than the other three libraries. The thread-safe version of Adept is a few percent more computationally expensive due to the global variable that stores a pointer to the currently active `Stack` object being declared "thread-local", which leads to an extra level of indirection whenever it is accessed. However, the most striking aspect is how much poorer all the libraries appear to perform for the Lax-Wendroff scheme. What actually happens is that the loop to compute the `flux` vector in the original Lax-Wendroff scheme can be aggressively optimized by the compiler. When transcendental functions are added to this line to convert it to the Toon et al. scheme, they not only incur their own cost, but optimization is impeded, and the algorithm is slowed by more than a factor of 20. The use of automatic differentiation libraries impedes much of this optimization, and the Toon et al. scheme is only 50–90% slower than Lax-Wendroff. These two schemes can be thought of as representing extreme cases, with one having more use of transcendental functions per ordinary arithmetic operation than most real-world applications, and the other having none at all.

We consider next the real-world algorithms shown by the final two columns of Table I, for the case when $N = 50$ (described in Section 6.1). This time the results are more consistent: Adept is 3.7–3.9 times the computational expense of the original algorithm, and 10–25% more computationally expensive than the hand-written adjoint codes. The

Table II. Same as Table I but for the Memory Usage Associated with an Adjoint Calculation

Algorithm	LW	Toon	PVC	TDTS
Number of statements / variables stored by Adept				
Number of statements	398101	398101	1363	58721
Mean no. of active variables on RHS of a statement	3.5	4.5	2.6	4.3
Number of gradients to store	305	305	475	1598
Memory footprint (kB)				
Hand-coded adjoint minus original	0	1632	20	168
Adept	19796	24548	54	3348
ADOL-C	31102	46888	84	4504
CppAD	34398	44115	106	7308
Sacado::Rad	110912	168504	416	23904

The first three rows are statistics from Adept and indicate, respectively, the length of the statement stack, the ratio of the lengths of the operation and statement stacks, and the length of the vector holding the gradients in the reverse pass. The final five rows indicate the memory usage in kilobytes needed to store the information carried between the forward and reverse passes, and in the first two of these also the storage used to execute the reverse pass. For the hand-coded adjoint, this is largely the intermediate model states. For Adept, it is the memory associated with the first three rows of this table. For ADOL-C, it is the size of the files written when the user requests the tape not to be held in memory. For CppAD, it is the sum of the values returned from the `memory()` member function of the tape object and the `size_op_seq()` member function of the `ADFun` object. For Sacado, it is computed from the number of `ADmemblock` objects allocated.

existing automatic differentiation libraries are considerably slower: ADOL-C is 5–7 times more expensive than Adept, CppAD is 7–9 times more expensive and Sacado is 2.6–8 times more expensive. The fact that these algorithms show a performance more similar to Toon et al. than Lax-Wendroff is because they both contain some use of transcendental functions, or have a serial dependence between loops, which impedes aggressive optimization of the original algorithm by the compiler.

It is interesting to look at a more detailed breakdown of these timings for all four algorithms. The benefit of expression templates is revealed when we consider the forward pass (the first number in parentheses in Table I), for which Adept is 2.4–7 times faster than the others. However, there is also an indirect speed-up due to the fact that Adept stores the differential information in a very efficient form for the reverse pass (see Section 3). ADOL-C and CppAD, by contrast, store a complete description of the algorithm including a symbolic representation of every operator and mathematical function, resulting in the reverse pass being 2.3–12 times slower than in Adept (compare the last numbers in parentheses). Sacado is most similar to Adept in that only differential information is stored, yet the reverse pass is 3.3–12 times slower than in Adept. The reason that Adept can sometimes spend half as much time in the reverse pass as the hand-coded adjoint is because it does most of the extra differential computations in the forward pass, while the hand-coded adjoint simply stores intermediate values in the forward pass with all the extra computation in the reverse pass.

In terms of memory, Table II shows that Sacado uses 5.6–7.7 times as much as Adept in these examples, which appears to be due to Sacado storing 4 `doubles` and 9 pointers for every binary operation. CppAD uses close to twice as much as Adept due to CppAD's need to process the tape in the forward pass to produce one that is suitable for the reverse pass; each tape individually is about the same size as the memory footprint of Adept. ADOL-C only generates one tape and its memory footprint lies between those of CppAD and Adept. The relative difference in memory usage between Adept and the hand-coded adjoints is very variable. The linearity of the Lax-Wendroff algorithm means that in fact no intermediate values need to be stored in its hand-coded adjoint function. For the Toon and TDTS algorithms, Adept uses 15–20 times more memory

Table III. Same as Table I, but Showing the Relative Time to Compute the Jacobian Matrix using a Forward or a Reverse Pass

Algorithm	LW	Toon	PVC	TDTS
Size of Jacobian matrix ($m \times n$)	100×100	100×100	50×350	50×350
Relative cost of computing the Jacobian matrix in a forward pass				
Adept	309	18	133	129
ADOL-C	371	34	318	250
CppAD	3197	244	1793	1799
Sacado::ELRFad	166	20	134	157
Relative cost of computing the Jacobian matrix in a reverse pass				
Adept	360	21	20	19
ADOL-C	558	52	80	68
CppAD	4535	402	355	456
Sacado::Rad	10426	715	170	789

In the case of Adept, ADOL-C and CppAD, the same tape or stack is recorded, but then processed by different functions for the forward and reverse methods. In the case of Sacado, two completely different versions of the library are used. In the forward case we use Sacado::ELRFad (expression-level reverse mode for forward-mode automatic differentiation), which does not store a tape but rather replaces every intermediate scalar with an object containing the derivative of the scalar with respect to each of the independent variables. In the reverse case we use Sacado::Rad (reverse-mode automatic differentiation), which is the same as used for the adjoint calculations in Table I but with m calls to the reverse pass, where m is the number of dependent variables.

than hand coding. In these examples, the hand-written adjoints recompute certain variables in the reverse pass, which leads to a big saving in memory. For example, for the Toon algorithm, only the values of q at each timestep were stored for the reverse pass, but not the values of the fluxes between grid points. In the case of the PVC algorithm, Adept uses only 2.7 times the memory of the hand-coded adjoint, which in this case is due to the latter storing a much greater fraction of the intermediate variables.

Next, we consider the speed of the calculation of the full Jacobian matrix, and the results are shown in Table III. Both the forward and reverse approaches to computing the Jacobian matrix are considered, as outlined in Section 5. Consider first the reverse approach, which is generally the most efficient in the case of the PVC and TDTS algorithms that have seven times as many independent variables as dependent variables. In this case, a forward pass is used to store a stack or tape, followed by m reverse passes each with a different seed vector (where m is the number of dependent variables). CppAD and Sacado::Rad follow this approach exactly, leading to a cost very close to m times that of the reverse pass of the adjoint computation. ADOL-C and Adept treat multiple seed vectors at once, leading to code that is more easily optimized by the compiler and a much faster Jacobian calculation. Adept is substantially faster for Jacobian calculations using a reverse pass than the other libraries. As with the adjoint calculations, the Jacobian calculation for the Lax-Wendroff algorithm has a high relative cost because the original algorithm could be aggressively optimized by the compiler. Now consider the forward approach for computing the Jacobian matrix. Adept, ADOL-C and CppAD take the same stack or tape and then perform n forward passes each with a different seed vector (where n is the number of independent variables). The time taken is then a little less than n/m the cost of using a reverse approach. Sacado::ELRFad does not store a tape, but for every intermediate variable in the algorithm it carries forward a vector of derivatives of that variable with respect to each of the n input variables. This turns out to be marginally slower than Adept for the Toon, PVC and TDTS algorithms, but appreciably faster for the Lax-Wendroff algorithm. Note that for these particular examples the Jacobian is relatively dense and we have neither attempted to use a coloring approach to exploit sparsity in the Jacobian (e.g., Gebremedhin et al. [2005]), nor attempted to speed-up the reverse pass by eliminating vertices from the computational graph (e.g., Naumann [2004]).

Finally we consider the compile times of the various automatic differentiation libraries. When applied to source files in the *Multiscatter* package, the relative compile times (i.e. divided by the time to compile the original code) for Adept, ADOL-C, CppAD, Sacado::Rad and Sacado::ELRFad are 8.6, 3.5, 12, 14, and 22, respectively. This is for the test system described in the caption of Table I. Thus, even though it uses expression templates, the compile time of Adept is competitive to other libraries.

7. CONCLUSIONS

This article has demonstrated how the operator-overloading approach to first-order reverse-mode automatic differentiation can be significantly accelerated using the technique of expression templates in C++, which enables the differential statements to be computed and stored very efficiently. In addition to fast adjoint calculations, the efficient storage of differential information allows fast calculation of the full Jacobian matrix. A freely available software library, “Adept”, has been released with these features.

In the four algorithms tested in this article, Adept is found to be 2.6–9 times faster than three other state-of-the-art operator-overloading automatic-differentiation libraries. Moreover, in three of the four algorithms it is only 2.7–3.9 times the cost of the original algorithm. In the algorithm containing no transcendental functions, aggressive optimization of the original algorithm by the compiler leads to a significantly worse relative performance by all automatic-differentiation libraries tested, including Adept. Nonetheless, for many real-world algorithms, this paper has demonstrated that the operator-overloading approach to automatic differentiation holds out the promise of having an efficiency approaching a source-to-source compiler, while being more convenient for the user and placing far fewer restrictions on the range of language features (such as templates) that can be used in the code to be differentiated.

It is hoped that this speed-up will make automatic differentiation a more obvious choice in the development of large codes for optimization problems. However, the library described in this article applies only to expressions involving individual real numbers and further development is needed to fully support active complex numbers, as well as arrays of active variables using array containers that support operations on entire arrays. An additional future area of work will be to investigate whether automatic computation of the Hessian matrix can be sped up by the use of expression templates; this would require second-order differential information to be stored during the forward pass.

REFERENCES

P. Aubert, N. Di Césaré, and O. Pironneau. 2001. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Comput. Vis. Sci.* 3, 197–208.

R. Bartlett, D. Gay, and E. Phipps. 2006. Automatic differentiation of C++ codes for large-scale scientific computing. In *Proceedings of the International Conference on Computational Science (ICCS'06)*. Lecture Notes in Computer Science, vol. 3994, 525–532.

B. Bell. 2007. CppAD: A package for C++ algorithmic differentiation. `http://www.coin-or.org/CppAD`.

C. Bendtsen and O. Stauning. 1996. FADBAD, a flexible C++ package for automatic differentiation.Tech. Rep. IMM-REP-1996-17, Technical University of Denmark.

D. M. Gay. 2005. Semiautomatic differentiation for efficient gradient computations. In *Automatic Differentiation: Applications, Theory, and Implementations*, H. M. Bücker, G. F. Corliss, P. Hovland, U. Naumann, and B. Norris, Eds., Springer, 147–158.

A. H. Gebremedhin, F. Manne, and A. Pothen. 2005. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Rev.* 47, 629–705.

R. Giering and T. Kaminski. 1998. Recipes for adjoint code construction. *ACM Trans. Math. Softw.* 24, 437–474.

A. Griewank and A. Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* 2nd Ed. SIAM.

A. Griewank, D. Juedes, and J. Utke. 1996. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.* 22, 131–167.

R. J. Hogan. 2008. Fast lidar and radar multiple-scattering models – 1. Small-angle scattering using the photon variance-covariance method. *J. Atmos. Sci.* 65, 3621–3635.

R. J. Hogan and A. Battaglia. 2008. Fast lidar and radar multiple-scattering models – 2. Wide-angle scattering using the time-dependent two-stream approximation. *J. Atmos. Sci.* 65, 3636–3651.

P. D. Lax and B. Wendroff. 1960. Systems of conservation laws. *Commun. Pure Appl. Math.* 13, 217–237.

D. C. Liu and J. Nocedal. 1989. On the limited memory method for large scale optimization. *Math. Programming B* 45, 503–528.

U. Naumann. 2004. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Program.* 99, 399–421.

E. Phipps and R. Pawlowski. 2012. Efficient expression templates for operator overloading-based automatic differentiation. In *Recent Advances in Algorithmic Differentiation,* Lecture Notes in Computational Science and Engineering, vol. 87, Springer.

O. B. Toon, R. P. Turco, D. Westphal, R. Malone, and M. S. Liu. 1988. A multidimensional model for aerosols: Description of computational analogues. *J. Atmos. Sci.* 45, 2123–2143.

T. Veldhuizen. 1995. Expression templates. *C++ Report* 7, 26–31.

M. Voßbeck, R. Giering, and T. Kaminski. 2008. Development and first applications of TAC++. In *Advances in Automatic Differentiation,* Lecture Notes in Computational Science and Engineering, vol. 64, Springer, 187–197.

J. Willkomm and M. Bücker. 2007. AD tools from a user's perspective. *In Proceedings of the 6th European AD Workshop.* http://www.sc.rwth-aachen.de/willkomm/pdf/6th-euro-ad-willkomm.pdf.

Received September 2012; revised September 2013; accepted December 2013