Optimization Methods and Software

ISSN: 1055-6788 (Print) 1029-4937 (Online) Journal homepage: http://www.tandfonline.com/loi/goms20

Expression templates for primal value taping in the reverse mode of algorithmic differentiation

M. Sagebaum, T. Albring & N. R. Gauger

To cite this article: M. Sagebaum, T. Albring & N. R. Gauger (2018): Expression templates for primal value taping in the reverse mode of algorithmic differentiation, Optimization Methods and Software, DOI: 10.1080/10556788.2018.1471140

To link to this article: https://doi.org/10.1080/10556788.2018.1471140

OPTIMIZATION METHODS & SOFTWARE
https://doi.org/10.1080/10556788.2018.1471140

Taylor & Francis
Taylor & Francis Group

Expression templates for primal value taping in the reverse mode of algorithmic differentiation

M. Sagebaum, T. Albring and N. R. Gauger

Chair for Scientific Computing, Technische Universität Kaiserslautern, Kaiserslautern, Germany

ABSTRACT

The reverse mode of Algorithmic Differentiation (AD) can be implemented in several ways. The major choices are primal value taping vs. Jacobian taping, managed indices vs. unmanaged indices and operator level taping vs. statement level taping. Most of the current AD tools have implemented only one of the eight possible choices, and the data management of the implementation adds another complexity hierarchy. The focus in this paper is the implementation of primal value taping on a statement level. Statement level taping removes the need to create intermediate values on the AD tape which results in reduced memory compared to operator level taping. The implementation will be done for managed and unmanaged indices in the AD tool CoDiPack. Primal value taping with statement level taping has not yet been implemented in any other AD tool, thus we will analyse the properties of the taping approaches and highlight the important details for an efficient implementation. Furthermore, all existing taping approaches in CoDiPack will be compared with the new primal value taping approach. The comparison have been conducted on a simple toy problem and a fully featured computational fluid dynamics solver in the multi-physics suite SU2.

ARTICLE HISTORY
Received 28 February 2017
Accepted 19 April 2018

KEYWORDS
C++; memory analysis; implementation strategies; algorithmic differentiation; automatic differentiation; AD by overloading

AMS SUBJECT CLASSIFICATIONS
65Y20; 68N19; 68N30; 68Q25

1. Introduction

Algorithmic Differentiation (AD) describes how any computer program can be automatically differentiated. The program is broken down into elemental functions like $+$,$*$, sin, etc. On the combined elemental functions the chain rule as well as the directional derivative is applied which yields the AD forward mode differentiation of the program. The adjoint of the AD forward mode yields the AD reverse mode. For a full derivation of the theory see [12,15].

Since the reverse AD mode needs to iterate over all operations in reversed order, the tool needs to store some information for each operation. One possible choice is to implement the reverse AD mode via operator overloading in C++. These AD tools need to split a combined expression like

$$w = \sin((a + b) \cdot (c - d)) \tag{1}$$

CONTACT M. Sagebaum max.sagebaum@scicomp.uni-kl.de

into several sub-expressions

$$t_1 = a + b; \quad t_2 = c - d; \quad t_3 = t_1 \cdot t_2; \quad w = \sin(t_3). \tag{2}$$

They need to store the information for three binary operations and one unary operation. If expression templates [17] are used, the above statement can be handled as one large expression with four arguments. Depending on the memory per operation and argument, that is required by the AD tool, the gain in memory and efficiency can be significant [13].

AD tools using operator overloading in C++ and having implemented the reverse mode, can be divided into two major groups. The first group consists of the ones that store the local Jacobian of each expression during taping. We refer to this strategy as the *Jacobian* approach. Examples are Sacado [16], Adept [13] and CoDiPack [2,8]. Hogan showed in [13] that this method can be efficiently implemented by means of expression templates .

The second group of tools are the ones that store the input and output values of each expression. We call this method the *primal value* approach. FADBAD [5], CPPAD [4] and ADOL-C [18] are some representatives of this group. The implementations for this method range from traditional tape structures (stack management (ADOL-C)) to graph representations (tree structures (FADBAD)), but to our knowledge there is no tool available that uses expression templates for the primal value approach. Consequently, there is currently no possibility for a fair comparison between both approaches. The aim of this work is to provide a unified AD framework, based on expression templates, that enables the comparison of different taping methods regarding run time and memory requirements. Furthermore, we will highlight the strengths and weaknesses of either of the methods.

First an introduction to AD and for the expression template technique is provided. After some short general notions about an operator overloading implementation, the primal value taping approach with expression templates is discussed and analysed. The presented analysis is used to implement the primal value taping approach in CoDiPack [8], which then provides implementations for primal value taping and Jacobian taping. Finally, the four different methods namely, Jacobian taping, Jacobian taping with an index reuse, primal value taping and primal value taping with an index reuse, of CoDiPack are compared on a simple test example and on the CFD solver in the multi-physics package in SU2 [10]. All code examples in this paper are C++ code.

2. Algorithmic differentiation

We assume each program can be viewed as a function with the definition *void func(const double x[], double y[])*. Some input variables x are used to compute the output variables y. In most cases, the input variables x and the output variables y will not be arrays but collections of several variables. Nevertheless, *func* can always be described as a mathematical function $f : \mathbb{R}^n \to \mathbb{R}^m$ with $y = f(x)$. Each operation executed by *func* can be described as an elementary operation $\Phi : V \to V$ like $+, *$, sin, cos etc. $V := \mathbb{R}^n \times \mathbb{R}^k \times \mathbb{R}^m$ is the state space of the evaluation of *func*. It consists of the input values $x \in \mathbb{R}^n$, the intermediate values $z \in \mathbb{R}^k$ and the output values $y \in \mathbb{R}^m$. f is a combination of elementary operations Φ_i which yields the interpretation of f as a concatenation of elementary operations, that is

$$f(x) = P_y \circ \Phi_{k+m} \circ \Phi_{k+m-1} \circ \cdots \circ \Phi_2 \circ \Phi_1 \circ I_x(x). \tag{3}$$

$I_x : \mathbb{R}^n \to V$ and $P_y : V \to \mathbb{R}^m$ are projections onto and from the state space V, respectively. They are defined as $I_x(x) = (x, 0, 0)^{\mathrm{T}}$ and $P_y((x, z, y)^{\mathrm{T}}) = y$. $\Phi_i : V \to V \quad \forall i = 1 \ldots (k + m)$ represent one update in the state space and each Φ_i corresponds and corresponds to one operation in *func*. For example, if the ith operation in *func* is $c = a \cdot b$ then $\Phi_i(v) := (v_1, \ldots, v_{i+n-1}, v_a \cdot v_b, 0, \ldots, 0)$ with v_a and v_b being the corresponding entries in the vector $v \in V$ for the variables a and b.

The function *func* loses all structural information like function calls, loops, etc. in the mathematical representation and represents only one evaluation of *func*. This means that e.g. loops are unrolled and the number of operations k can vary. For real world applications k can be in the range of 10^{10}. The SU2 example from Section 8.2 has already 10^9 operations which is a relatively small test case.

The theory of AD [12,15] is applied on (3) and yields the formulation

$$\dot{y} = \frac{\mathrm{d}f}{\mathrm{d}x}\dot{x}, \tag{4}$$

for the directional derivative which is also known as the forward AD mode. $\dot{x} \in \mathbb{R}^n$ is the direction of the input vectors in which the directional derivative is evaluated. $\dot{y} \in \mathbb{R}^m$ is then the directional derivative of f in the direction $\dot{x}$. The theory also states that (4) holds for every elementary operation. For $c = w(a, b) = a \cdot b$ this would mean $\dot{c} = \mathrm{d}w/\mathrm{d}(a, b) \cdot (\dot{a}, \dot{b})^{\mathrm{T}}$.

The adjoint of (4) yields the reverse mode of AD that evaluates

$$\bar{x} += \frac{\mathrm{d}f}{\mathrm{d}x}^{\mathrm{T}} \bar{y}; \quad \bar{y} = 0. \tag{5}$$

$\bar{y} \in \mathbb{R}^m$ is the vector for the adjoint direction and $\bar{x}$ is the result of the adjoint formulation. If $\bar{x}$ is initially zero and f is the only function that is called, then (5) simplifies to

$$\bar{x} += \frac{\mathrm{d}f}{\mathrm{d}x}^{\mathrm{T}} \bar{y}, \tag{6}$$

which is referred to as the reverse AD equation that is computed by the AD tool. It is very important that the Jacobian $\mathrm{d}f/\mathrm{d}x$ in Equation (6) is never set up explicitly. Most of the time it is also not possible to derive the representation of $\mathrm{d}f/\mathrm{d}x$ in an analytical way for a fully featured code base. The theory of AD states that if Equation (5) is evaluated for every elementary operation Φ_i in reverse order, that is for $i = (k + m) \ldots 1$, the end result will be the correct derivative, described by Equation (6).

Since, Equation (5) needs to be evaluated for every operation but in reverse order, the AD tools implement the reverse AD mode by storing certain information for each operation as it is evaluated in *func* during the primal evaluation. After the primal evaluation is finished, the AD tools go through the recorded information in reversed order for the reverse evaluation.

In this paper an arbitrary statement with $l \in \mathbb{N}$ arguments is denoted by

$$w = \phi(v), \tag{7}$$

where $v \in \mathbb{R}^l$ and $w \in \mathbb{R}$. If Equation (5) is applied on Equation (7) the AD tool needs to evaluate

$$\bar{v} += \frac{\mathrm{d}\phi}{\mathrm{d}v}(v)^{\mathrm{T}} \bar{w}; \quad \bar{w} = 0 \tag{8}$$

for every operation that is evaluated in *func*. In the following sections of the paper it is explained how Equation (8) is generated with expression templates and which kind of data is required by the primal value taping approach for Equation (8).

2.1. Primal value taping and Jacobian taping

The difference between primal value taping and Jacobian taping is the handling of the elementary operator in Equation (8). The gradient of ϕ is required for the update of the bar variables which can be computed and stored directly so that it may be retrieved and used during the reverse interpretation. This approach is the Jacobian taping technique. The data required for the statement in Equation (1), are the four derivatives $\partial w/\partial a$, $\partial w/\partial b$, $\partial w/\partial c$ and $\partial w/\partial d$ as well as the number of arguments, which in this case is four. The advantage is that the primal values do not need to be stored and that during the reverse interpretation only a fused multiply-accumulate operation is evaluated.

On the other hand, it is possible to store the arguments v and corresponding operations. This information is then used during the reverse interpretation to compute the gradient of ϕ on the fly. The required data for Equation (1) is the value of w and a function pointer to evaluate the gradient during the reverse sweep. Section 5.1 describes why w is stored instead of v. The advantage of the primal value taping approach is that less memory per elementary operation is required.

For a primal value taping approach that does not use expression templates the required data would be the primal values of t_1, t_2, t_3 and w as well as the identifiers for the addition, subtraction, multiplication and power operations from Equation (2). Especially if large statements are used the advantage of handling the statement as one large operation becomes quite clear.

The next section will introduce expression templates and after that the implementation of primal value taping with expression templates is discussed.

3. Expression templates

C++ only allows the overloading of single operators and not the overloading of whole statements. Therefore, for a regular reverse mode implementation the statement

$$w = \sin((a + b) \cdot (c - d)) \tag{9}$$

is split into the sub-expressions

$$t_1 = a + b; \quad t_2 = c - d; \quad t_3 = t_1 \cdot t_2; \quad w = \sin(t_3).$$

Hence, the AD tool would need to store the information for three binary operations and one unary operation. This approach views an operator as the function

$$\texttt{Type} \circ \texttt{Type} \rightarrow \texttt{Type} \tag{10}$$

where *Type* is the AD type that overloads the operator.

The layout of the operators can now be changed, such that the type of the arguments is not fixed and the type of the result depends on the arguments. The notation for the operator

is then

$$\texttt{Expr}_A \circ \texttt{Expr}_B \rightarrow \texttt{Expr}_{A \circ B} \tag{11}$$

where *Expr*, which stands for expression, is a general type that stores information about the operations. Subscripts like A and B indicate the information stored in the expressions. The expression resulting from the operator will contain the information about both arguments and the operation, which is indicated by the subscript $A \circ B$.

With this approach multiple operators can be concatenated together and create one large expression for the full statement. For example, the generated structure for Equation (9) would be

$$\texttt{SIN} < \texttt{MULT} < \texttt{ADD} < \texttt{Type}, \texttt{Type} >, \texttt{SUB} < \texttt{Type}, \texttt{Type} >>> . \tag{12}$$

The advantage of this large expression is that no intermediate variables are generated and it is also possible to access the information about the whole statement. An AD tool can use this information to store the required data for the tape interpretation more efficiently. In this case it would need to store the information for one operation with four arguments.

It is now necessary to implement some methods for the structure in Equation (12) such that the primal value can be computed or the AD tool can access the AD specific information. For this we will use the curiously recurring template pattern [7,9] in C++. This pattern uses the template mechanism of C++ to give the base class access to the implementing class. The advantage of using this pattern is that the compiler can inline all functions. The implementation would also be possible with virtual functions but then the evaluation time of the program would be much slower since the compiler cannot inline any longer. How expression templates can be implemented with curiously recurring template pattern is now demonstrated for the calculation of the primal value.

Later in this paper, this implementation will be extended for the primal value taping of the reverse mode. For an implementation of the forward mode with expression templates see [3].

The base class for the pattern is the *Expression* class:

```
1 template<typename A>
2 class Expression {
3     A& cast() { return static_cast<A&>(*this);}
4     double getValue() { return this->cast().getValue();}
5 };
```

The template argument defines the implementing class of the interface and is used to cast instances of the interface to the implementing class (line 3). All other methods like the *getValue* method are then called via the *cast* method on the implementing class.

A possible implementation for the multiplication would then look like:

```
template<typename A, typename B>
class MULT : public Expression<MULT<A, B > > {
    const A& a;
    const B& b;

    MULT(const A& a, const B& b) : a(a), b(b) {}
    double getValue() { return a.getValue() * b.getValue();}
}
```

The template argument to the expression interface is the class itself. Otherwise the *getValue* implementation is straightforward.

It is now necessary to overload the multiplication operator such that the multiplication expression is generated:

```
template<typename A, typename B>
MULT<A, B> operator * (const Expression<A>& a, const Expression<B>& b) {
    return MULT<A,B> (a.cast(), b.cast());
}
```

This implementation shows clearly the paradigm shift in the implementation of the overloaded operators. Instead of returning the result of the operation, a general structure is returned.

In order to capture all possible cases, a complete set of operators needs to be defined for the expression templates. This includes the binary operators $+$, $-$, $/$ and the unary functions sin, exp, etc. The next step is now to extend the functionality of the expression interface such that AD types can use it to compute derivatives.

4. General AD implementation

From the theory in Section 2 we have learned that for each primal variable a an adjoint variable $\bar{a}$ is introduced. If AD is introduced through operator overloading, the calculation type, e.g. *double*, is replaced with an appropriate AD type. The AD type stores the primal value but cannot be used to store the adjoint value as we now show. Consider the following listing:

```
1    {
2        Type a = ...
3        ... = ... a ...
4    }
```

Here, a runs out of scope in line 4 and is deleted. The adjoint variable is therefore not accessible any longer and needs to be stored in a different location for the reverse evaluation. This is usually a structure, called 'the tape', and contains all the data for the reverse evaluation. We assume that the tape has some basic functionality to store arbitrary data in an optimal way.

Nevertheless, the AD tool requires some means to identify a variable during the reverse sweep. One efficient notion is already introduced by the splitting of f into elementary operators. Each elementary operation Φ_i in Equation (8) updates the $n+i$th component in the state space. This index can be used to identify each adjoint variable. The implementation is done such that each variable e.g. c is associated with an index $c.index$. During the primal evaluation the index of c is populated by increasing a global index counter and assigning the

new index to *c.index*. The adjoint value of *c* can then be accessed through a global adjoint vector via *adjoint[c.index]*. The concept is called linear indexing, as for each operation in the program the index is increased.

The AD type, that we will use in this paper, is called *RReal* (reverse real) and contains the primal value and the associated index. The structure is defined as:

```
struct RReal : public Expression<RReal> {
    double value; // primal value
    int index; // identifier for the reverse sweep

    double getValue() { return value};
};
```

It implements the expression interface and can therefore be used to generate expressions. The implementation of reverse AD with expression templates can now be discussed.

5. Primal value taping with expression templates

In the previous sections the reverse AD equation (8) for elementary operations was introduced. The equation is

$$\bar{v}_i \mathrel{+}= \frac{\partial \phi}{\partial v_i}(v)^{\mathrm{T}} \bar{w}; \quad i = 1 \dots l \tag{13}$$

which comes from the elementary operator $w = \phi(v)$ with $v \in \mathbb{R}^l$ and $w \in \mathbb{R}$. Whenever an assignment in a code is performed, the elementary operator is generated by expression templates, as introduced in Section 3, and now all necessary information needs to be stored, such that Equation (13) can be evaluated during the reverse interpretation of the tape.

For the primal value taping approach the required data from Equation (13) will be the indices associated with v in order to access the primal as well as the adjoint value, the primal value of w and some means to evaluate the gradient of ϕ during the reverse sweep. In our case this will be a function pointer.

The assign operator is overloaded for the *RReal* class and calls the *record* method on a global tape:

```
\\ RReal class
    static Tape globalTape;

    template<typename RHSExpr>
    RReal& operator=(RHSExpr& rhs) {
        globalTape.record(*this, rhs);
   }
```

The *record* method will acquire all information for the primal value taping via the expression templates. This process is discussed step by step in the following subsections

5.1. Storing of primal values

The primal value taping approach needs to store the value of v in order to evaluate Equation (13). If this were done for the following four statements

```
a = ...
b = a * a;
c = a + b;
d = a - b;
```

the value of a appears four times on the right-hand side (rhs) in these statements and would need to be stored four times. It would be more appropriate to store the value of a when it is assigned as a left-hand side value (lhs) and then use this information during the reverse sweep. Because AD assigns a unique index to each variable (as we assumed in Section 4), this index can be used to store the primal value. The tape class is extended by a global primal value vector and the record method updates this primal value vector:

```
\\ Tape class
    double* primalValues;

    template<typename RHSExpr>
    void record(RReal& lhs, RHSExpr& rhs) {
        lhs.index = nextIndex();
        lhs.value = rhs.getValue();
        primalValues[lhs.index] = lhs.value;
    }
```

Since the primal value is stored for every expression, 8 bytes are now required per expression.

5.2. Storing of the arguments

The primal values of the arguments are now available through the primal value vector in the tape. In order to know which values are used in the expression, the indices of the values need to be stored. The indices are used to access the primal values and to access the adjoint values $\bar{v}_i$ from Equation (13). This requires 4 bytes per argument. If the primal value vector in the tape were not available, 12 bytes would be required.

The indices of the arguments are stored in the rhs expression and need to be extracted from the expression. That is, for each argument the index must be stored, which is a per value action on the expression. Since it is necessary to create several actions, that perform an operation for each argument, we implement a more general function with function objects [14].

The function object, that stores the indices, is implemented in the tape class as a static function *storeIndices*:

```
\\ Tape class
    static void storeIndices(const RReal& value, Tape& tape) {
         tape.pushIndex(value.index);
    }
```

The arguments are the current argument value from the expression and the tape that called the per value action on the expression. The method *pushIndex* will store the indices in memory that can be sequentially accessed.

In order to implement the per value action, the expression interface is extended with a *perValueAction* method. The example implementation for the multiplication is:

```
\\ MULT<A,B> class
    template< typename Tape, typename FuncObj>
    void perValueAction(Tape& tape, FuncObj& func) {
        a.perValueAction(tape, func);
        b.perValueAction(tape, func);
    }
```

The recursive nature of the method will propagate the call to all arguments of the whole expression, which are the *RReal* objects. Here, the function object needs to be called:

```
\\ RReal class
    template< typename Tape, typename FuncObj>
    void perValueAction(Tape& tape, FuncObj& func) {
        func(*this, tape); // call the function object for each argument
    }
```

Now an arbitrary per value action can be evaluated on the expression and the record method can be extended such that it calls the per value action for storing the indices on the expression:

```
\\ Tape class
    template<typename RHSExpr>
    void record(RReal& lhs, RHSExpr& rhs) {
        rhs.perValueAction(*this, &Tape::storeIndices);

        lhs.index = nextIndex();
        lhs.value = rhs.getValue();
        primalValues[lhs.index] = lhs.value;
    }
```

This will store the indices for the expression before the primal value of the lhs is stored. It is important to note that the order of the arguments is fixed by the recursive calls in the *perValueAction* method. As long as the same recursive calls are used, the order of the arguments will stay the same. With this assumption, the arguments of the expression can be numbered from 0 to $l-1$. It is also important that the indices are stored in sequence, without any interleave.

5.3. Storing of the expression

The information for the primal values and the arguments is now available. The next required data is the information how the Jacobian of ϕ can be computed with the available data.

Before the chosen approach is presented, it is necessary to evaluate the drawbacks of other approaches. It is possible to store the generated expression directly on the tape, when no references are used in the expression template implementation. This would also store the indices and primal values of the arguments, but it would add extra memory due to the padding bytes of the structure. Since the type of the expression cannot be stored, additional bytes are required for the virtual function table of the interface. A direct storage of the expression is therefore not advisable, because it increases the required memory.

The chosen approach uses static methods in the expressions to provide the functionality for the computation of the Jacobian of ϕ. In order to do that, it is necessary to have a

function that can compute the primal value in the static context. Both implementations, for the reverse and the primal evaluation, are based on the same principle and therefore we will start by introducing the computation of the primal value.

The implementation is done in a recursive fashion and it is required that it has the same order as in the *perValueAction* method. Otherwise, the method only needs to get the primal values of the arguments and perform the operation. This is shown as an example for the multiplication:

```
\\ MULT<A,B> class
    template< size_t indexOffset>
    static double getValue(int* indices, double* primalValues) {
        double a = A::getValue<indexOffset> (indices, primalValues);
        double b = B::getValue<indexOffset + A::NumberOfArguments> (indices,
            primalValues);

        return a*b;
    }
```

The interesting notion in this implementation is the *indexOffset* template parameter. It assumes that the index of the first argument for the whole expression is stored at *indices[0]*, the second at *indices[1]* etc. It needs to be computed correctly in the recursive calls, such that the arguments for the whole expression get the correct offset. For the first argument of the multiplication the offset does not need to be changed. For the second argument the offset is changed by the number of arguments in the sub-graph of the first one. This ensures that each termination point gets a different index, which is increased one by one. The termination is now implemented in the datatype *RReal*. It is just a lookup in the index vector:

```
\\ RReal class
    template< size_t indexOffset>
    static double getValue(int* indices, double* primalValues) {
        return primalValues[indices[indexOffset]];
    }
```

The index calculation can be evaluated by the compiler at compile time, therefore no index offset calculations are performed during run time. The next step is to compute the gradient.

The computation for the gradient can be derived with AD. Let

$$a = A(o)$$
$$b = B(p)$$
$$c = a \cdot b$$

be the computational algorithm for the *getValue* method, where A and B represent the sub-graphs of the arguments from the multiplication. o and p are the inputs to these sub-graphs respectively. The algorithm can be differentiated with reverse AD, which yields

$$\bar{a} \mathrel{+}= b \cdot \bar{c} \tag{14a}$$

$$\bar{b} \mathrel{+}= a \cdot \bar{c} \tag{14b}$$

$$\bar{c} = 0 \tag{14c}$$

$$\bar{p} \mathrel{+}= \frac{\mathrm{d}B}{\mathrm{d}p}\bar{b} \tag{14d}$$

$$\bar{o} \mathrel{+}= \frac{\mathrm{d}A}{\mathrm{d}o}\bar{a}. \tag{14e}$$

The algorithm shows, that with a given seeding of $\bar{c}$, the gradient of the whole expression can be computed recursively. If A or B are *RReals* and therefore arguments of the full expression, the updates in Equations (14d) and (14e) become the updates to the corresponding adjoint variable of the expression. The implementation of Equation (14) for the multiplication is now straightforward:

```
1  \\ MULT<A,B> class
2      template<size_t indexOffset>
3      static double computeGrad(double seeding, int* indices, double*
           primalValues, double* adjointValues) {
4          double a = A::getValue<indexOffset> (indices, primalValues);
5          double b = B::getValue<indexOffset + A::NumberOfArguments> (indices,
               primalValues);
6
7          double a_b = b * seeding; // d phi/ d v_1 = b
8          double b_b = a * seeding; // d phi/ d v_2 = a
9          A::computeGrad<indexOffset> (a_b, indices, primalValues,
               adjointValues);
10         B::computeGrad<indexOffset + A::NumberOfArguments> (b_b, indices,
               primalValues, adjointValues);
11     }
12
13 // RReal class
14     template<size_t indexOffset>
15     static double computeGrad(double seeding, int* indices, double*
           primalValues, double* adjointValues) {
16         adjointValues[indices[indexOffset]] += seeding;
17     }
```

Lines 7 and 8 represent Equation (14a) and (14b). Equations (14d) and (14e) are implemented in lines 9 and 10. The function *computeGrad* can be used to compute the gradient $\mathrm{d}\phi/\mathrm{d}v$ and perform the update $\bar{v}_i \mathrel{+}= \partial\phi/\partial v_i^T \cdot \bar{w}$ for all $i = 1 \ldots l$. In the case of expression templates l can be arbitrarily large and not only one or two as is the case when operator overloading is used. The *computeGrad* method can be stored on the tape which uses 8 bytes. The *record* method looks now as follows:

```
1  \\ Tape class
2     template<typename RHSExpr>
3      void record(RReal& lhs, RHSExpr& rhs) {
4          rhs.perValueAction(*this, &Tape::storeIndices);
5
6          pushFunction(&RHSExpr::computeGrad<0>);
7
8          lhs.index = nextIndex();
9          lhs.value = rhs.getValue();
10         primalValues[lhs.index] = lhs.value;
11     }
```

The current implementation only stores the method for the reverse interpretation. If also a primal or forward interpretation is required additional pointers could be stored. For multiple pointers it would then be more efficient to gather these in a static object and store a pointer to the static object.

5.4. Implementing the reverse sweep

All the information is now available to implement the reverse sweep. Since we assume a linear indexing scheme (see Section 4), there is a start and end position for which the tape needs to be evaluated, which results in a simple loop:

```
\\ Tape class
    void interpret(int start, int end, double* primalValues, double*
        adjointValues) {
         for(int curPos = end; end > start; --curPos) {
             RevFunc* revFunc = popFunction();
             int* indices = popIndices(numberOfArguments);

             double w_b = adjointValues[curPos];
             // evaluate equation (13) for the current statement
             revFunc(w_b, indices, primalValues, adjointValues);
         }
   }
```

For every statement the information is retrieved and then the reverse evaluation function is called. The only missing information is *numberOfArguments*, which is required to pop the correct number of indices and corresponds to l from Equation (13). This number could also be stored on the tape but would require 1 to 4 bytes for each statement. It is also not necessary because the information is static and tightly coupled to the function pointer of *revFunc* which is implemented by *computeGrad* from the expression interface. This means that the function pointer can be used to get l without storing additional data on the tape. In order to do this, a wrapper is written for *computeGrad*:

```
\\ Tape class
    template<typename RHSExpr>
    static void evalRevFunc(Tape& tape, double seed, double* primalValues,
        double* adjointValues) {

       int* indices = tape.popIndices(RHSExpr::NumberOfArguments);
       RHSExpr::computeGrad<0>(seed, indices, primalValues, adjointValues);
  }
```

The number of arguments is taken from the *RHSExpr* template argument in order to pop the correct amount of indices and to call the *computeGrad* method. If *evalRevFunc* is stored instead of *computeGrad*, the number of arguments l does not need to be stored on the tape. The call in the record method then looks like

```
    pushRevFunc(&Tape::evalRevFunc<RHSExpr>);
```

and the interpret loop can be adjusted to:

```
    RevFunc* revFunc = popRevFunc();
    double w_b = adjointValues[curPos];
    revFunc(*this, w_b, primalValues, adjointValues);
```

The wrapper method could also be implemented in the expression interface, but this would introduce tape specific logic to the expressions which should be avoided.

This concludes the implementation of the expression templates for primal value taping. Per statement the number of bytes stored is 8 bytes for the primal value, 8 bytes for the function pointer and $l \cdot 4$ bytes for the indices of the arguments, which is in total $16 + l \cdot 4$ bytes for each expression.

5.5. Further considerations

5.5.1. Constant value treatment

The primal value implementation is now able to treat all statements that only use variables. Expressions like

$$w = a \cdot b, \tag{15}$$

where a and b are of the type *RReal* are no problem. If the expression is changed to

$$w = 4.0 \cdot b \quad \text{or} \quad w = c \cdot b, \tag{16}$$

where c is of the type *double*, the behaviour is not well defined. If a constructor existed, that can construct a *RReal* object from a *double*, this construction would write an additional statement before the statement for w is stored. However, this requires at least 16 bytes and increases the number of indices, which increases the size of the adjoint vector. Therefore, the constant value 4.0 needs to be handled directly in the expression templates for a more efficient solution.

The first step is to specialize the expression templates for the case that an expression and a double is used. The *MULT* expression from Section 3 is reimplemented for the case where the first argument is a *double*:

```
template<typename B>
class MULT_ConstantActive : public Expression<MULT_ConstantActive<B > > {
    const double a;
    const B& b;

    MULT(const double a, const B& b) : a(a), b(b) {}
    double getValue() { return a * b.getValue();}
}
```

The specialization of the operator is then:

```
template<typename B>
MULT_ConstantActive<B> operator * (const double a, const Expression<B>& b) {
    return MULT_ConstantActive<B> (a, b.cast());
}
```

The implementation is the same if the second argument is constant and the first one is an active value, as well as for other binary expressions.

Through the expressions the constant values are now available to the AD tool. The next step is to store the *double* values. In order to do this, the expressions are extended by a constant value action, similar to the *perValueAction* method. The code for the multiplication expression calls the function object for the passive value and performs a recursive call on the second argument:

```
1 \\ MULT_ConstantActive class
2     template< typename Tape, typename FuncObj>
3     void perConstantAction(Tape& tape, FuncObj& func) {
4         func(a, tape);
5         b.perConstantAction(tape, func);
6     }
```

The implementation in the *RReal* class is empty. The tape can now implement the store constant action as

```
\\ Tape class
    void static storeConstant(const double value, Tape& tape) {
        tape.pushConstant(value);
    }
```

and we extend the record method to store the constant data:

```
\\ Tape class
    template<typename RHSExpr>
    void record(RReal& lhs, RHSExpr& rhs) {
        rhs.perValueAction(*this, &Tape::storeIndices);
        rhs.perConstantAction(*this, &Tape::storeConstant);

        pushRevFunc(&Tape::evalRevFunc<RHSExpr>);

        lhs.index = nextIndex();
        lhs.value = rhs.getValue();
        primalValues[lhs.index] = lhs.value;
  }
```

The same principle as for the indices applies to the data of the constant values. They are stored sequentially without any interleave between the values. Therefore, the same compile time indexing technique can be used as in the computation of the gradients.

The third step is to extend the static *getValue* and *computeGrad* methods with the constant value information. Because the derivatives with respect to the constant values do not need to be computed, the implementation is only shown for the *getValue* method:

```
\\ MULT<A,B> class
    template< size_t indexOffset, size_t constantOffset>
    static double getValue(int* indices, double* primalValues, double*
        constantValues) {
          double a = constantValues[constantOffset];
          double b = B::getValue<indexOffset, constantOffset + 1> (indices,
              primalValues, constantValues);

          return a*b;
    }
```

This provides all the necessary information to handle constant values in an expression, which requires 8 bytes for each constant value. If the constant would be converted to a *RReal* and no special handling is done, 28 bytes would be required. This saves 20 bytes for each constant value.

5.5.2. Passive value treatment

In contrast to constant values, passive value are *RReal* values that do not depend on the input values. If only a small part of the code is passive, then a special handling of these passive values is not necessary. On the other hand, if a legacy code is differentiated with AD, then larger code parts might be passive and nothing should be recorded for these code parts. A Jacobian taping approach can just ignore these values. However, in a primal value taping approach, the primal values of the passive arguments need to be available in the expressions.

The implementation uses the zero index to indicate passive *RReal* values. The problem with this approach is seen during the reverse interpretation. If an expression has the form

$$w = a \cdot b \cdot v, \tag{17}$$

where a, b and v are *RReal* objects with a and b as passive objects (i.e. $a.index = 0$, $b.index = 0$, $v.index \neq 0$), then the storing of the indices would create the sequence 0, 0, v_i with $v_i \in \mathbb{N}$. These indices are used to access the primal values through the global primal value vector. If now the values of a and b are different, the position 0 in the vector cannot store both values. It is therefore necessary to generate temporary indices to store the primal values of the passive arguments. The generation of an index would generate a statement and would use at least 28 bytes, which should be avoided. It is also not possible to use compile time constants because a zero index can only be identified during runtime and not during compile time. Therefore, additional data needs to be stored in order to handle the passive values.

The implemented solution aims at providing a very memory efficient way for the handling of passive data. A certain number of indices are reserved which represent the passive data entries. Each passive *RReal* is temporarily assigned to one of these indices and the primal value of the *RReal* is written to the tape and not stored in the global primal value vector. In the reverse sweep the primal value is then read again and restored at the corresponding position. This will increase the memory of all expressions and not only of the ones that contain passive values. Because of this, the memory for the passive variable count is chosen as small as possible which is one byte on modern computer architectures. This limits the maximum number of arguments for an expression to 256. CoDiPack will throw a compile time error if statements with more arguments are detected.

The first step for the implementation is to count the number of passive values in the expression. The tape can implement the per value action *countPassives*:

```
\\ Tape class
    void static countPassives(const RReal& value, int& passiveCount) {
        if(value.index == 0) {
            passiveCount += 1;
        }
    }
```

This will count the number of passive values and can be used to ignore the whole expression if all arguments are passive. The *storeIndices* per value action is now changed such that the passive indices are exchanged with the reserved ones:

```
\\ Tape class
    void static storeIndices(const RReal& value, Tape& tape) {
        int index = value.index;
        if(0 == index) {
            tape.curPassiveIndex += 1;
            index = tape.curPassiveIndex;
        }
        tape.pushIndex(index);
    }
```

The implementation assumes that the first 256 indices are reserved and that the member *curPassiveIndex* is used to count how many passive indices have been used in the current expression. It is set to zero each time an expression is stored.

The storing of the primal values for the passive arguments is very simple and implemented as a function object:

```
\\ Tape class
    void static storePassives(const RReal& value, Tape& tape) {
       if(0 == value.index) {
           tape.pushPassive(value.value);
        }
    }
```

The *record* method is now extended by the appropriate calls:

```
\\ Tape class
    template<typename RHSExpr>
    void record(RReal& lhs, RHSExpr& rhs) {
        int passiveVariableCount = 0;
        rhs.perValueAction(passiveVariableCount, &Tape::countPassives);

        if(passiveVariableCount != RHSExpr::NumberOfArguments) {
            this->curPassiveIndex = 0;

            rhs.perValueAction(*this, &Tape::storeIndices);
            rhs.perConstantAction(*this, &Tape::storeConstant);
            rhs.perValueAction(*this, &Tape::storePassives);

            pushPassiveValueCount(passiveVariableCount);
            pushRevFunc(&Tape::evalRevFunc<RHSExpr>);

            lhs.index = nextIndex();
       } else {
           lhs.index = 0;
       }

       lhs.value = rhs.getValue();
       primalValues[lhs.index] = lhs.value;
  }
```

In line 5 the number of passive arguments is counted and used in line 7 to avoid the taping of the whole expression when it does not depend on the input values. *storeIndices* in line 10 has now a different logic which also replaces the passive indices. In lines 12 and 14 the passive values and the number of passive values are stored.

The reverse interpretation has to restore the passive values. This is a simple loop that sets the first n values of the global primal value vector:

```
\\ Tape class
    template<typename RHSExpr>
    static void evalRevFunc(Tape& tape, double seed, double* primalValues,
        double* adjointValues) {

        int passiveValueCount = tape.popPassiveValueCount();
        double* passiveValues = tape.popPassive(passiveValueCount);
        for(int i = 0; i < passiveValueCount; ++i) {
            primalValues[i + 1] = passiveValues[i];
        }

        int* constants = tape.popConstant(RHSExpr::NumberOfConstants);
        int* indices = tape.popIndices(RHSExpr::NumberOfArguments);
        RHSExpr::computeGrad<0, 0>(seed, indices, constants, primalValues,
            adjointValues);
  }
```

The rest of the method has changed only due to the constant value treatment.

This implementation requires 8 bytes per passive value and 1 byte per expression, which saves 19 bytes compared to an implementation that uses temporary indices. Nevertheless, the memory per expression is increased which might lead to an increased use of memory if the application has only a few passive values.

6. Primal value taping and index reuse

Until now we assumed that a linear indexing scheme is used. That is, each index is only given once to a variable and after that the index is never used for another variable. This causes the adjoint vector to be quite large. For example the loop

```
1    RReal w = 0.0;
2    for(int i = 0; i < n; ++i) {
3         RReal t = a[i] * a[i];
4         w += t;
5    }
```

will create an index for each t which is used only once inside the loop and never again. If the index management is changed from a linear one to a scheme that reuses indices, a few additional things need to be considered for the primal value taping.

The index reuse affects the global primal value vector. If in the above example t gets the same index in each iteration, the primal value will be different each time. Since, the primal value on the same index can now be changed several times, the meaning of the global primal value vector changes. For the linear indexing it represented the primal values of all intermediate variables in the program, now it just represents the primal values that are currently active. It is therefore necessary to restore the old values of t during the reverse interpretation so that the primal value vector contains the correct values. This can be done by storing the overwritten values of the primal value vector such that they can be used to undo the changes that are made during the reverse evaluation. In addition to the storing of the old primal value, it is also necessary to store the index of the lhs value. For the linear indexing the lhs index is increased one by one, but now the index can be arbitrary and can no longer be computed.

The changes to the record method are quite minimal. Before the new value is set into the primal value vector, the old value is pushed to the tape and afterwards the index of the lhs is pushed as well:

```
1  \\ Tape class, record function
2          lhs.value = rhs.getValue();
3          pushDouble(primalValues[lhs.index]);
4          pushIndex(lhs.index);
5          primalValues[lhs.index] = lhs.value;
```

This increases the memory per statement by 12 bytes, but the general memory per statement has increased only by 4 bytes, because for the linear index management 8 bytes were counted for the primal value vector. Since the primal value vector is no longer coupled to the statements, these 8 bytes can no longer be counted to the statements and need to be counted separately. However, the size of the global primal value vector is now significantly smaller. In the reverse evaluation function, the changes are also very minimal. The index for the lhs is restored first, followed by the old double value:

```
\\ Tape class, interpret function
       int curPos = popIndex();
       primalValues[curPos] = popDouble();
```

This will modify the primal value vector such that the correct state is maintained during the reverse interpretation of the tape.

The modification of the primal value vector has some side effects that can be seen if the tape is evaluated multiple times. After the first reverse interpretation, the primal value vector has the state from the beginning of the recording. If a second reverse interpretation is performed, the primal value vector has the wrong state. Therefore, the primal value vector cannot be modified directly during the reverse interpretation and needs to be copied. The reverse interpretation is then performed on the copied vector. Because of the copy, the size of the primal value vector is counted twice in the analysis of the memory.

6.1. Type property changes for index reuse

The discussion of the index reuse implementation is very brief and does not show the full complexity of the change. One property, which the index management enforces, is that each index needs to be unique in the application. If two *RReal* variables have the same index the derivative results will be wrong. Therefore the types that reuse indices are not compatible with c-like memory operations (e.g. memcpy). The implementation with linear indices on the other hand is compatible with c-like memory operations.

7. Implementation summary

The implementation of the primal value taping approach with expression templates is now complete. It can handle arbitrarily large expressions that can contain passive and constant values. All classes and structures are templated with respect to the floating point type. This allows for higher order derivatives by nesting different CoDiPack types. Table 1 summarizes the memory requirements and properties of the two types.

In order to give a better intuition about the stored data in the final implementation, it is displayed for the example statement:

```
// a.primal = 0.5 , a.index = 257
// b.primal = 0.25, b.index = 0
// c.primal = 10  , c.index = 1042
// d.primal = 1   , c.index = 12345
    w = 4.0 * sin(a + b) / (c - d);
```

The statement contains a constant value, a passive value (namely that of b) and three active values. The created expression template structure for the statement would be:

Table 1. Summary of the properties and required memory for the primal value types with expression templates.

Type	Memory in byte for one statement	Properties
Primal lin. indexing	$4 \cdot \text{nArgs} + 8 \cdot \text{nPas} + 8 \cdot \text{nCon} + 8 + 8 + 1$	behaves as POD type
Primal index reuse	$4 \cdot \text{nArgs} + 8 \cdot \text{nPas} + 8 \cdot \text{nCon} + 8 + 8 + 1 + 4$	no memcpy etc.

Note: Per statement values: *nArgs*: Number of arguments, *nPas*: Number of passive values, *nCon*: Number of constant values.

Table 2. Stored information for the statement w = 4.0 * sin(a + b)/ (c - d);.

Data	Value	Memory
Indices	257, 1, 1042, 12345	4 · 4 byte
Constant doubles	4.0	8 byte
Passive doubles	0.25	8 byte
Expression	Function pointer	8 byte
Number of passive values	1	1 byte
lhs value	0.30295	8 byte

```
MULT_PA<DIV<SIN<ADD<RReal, RReal>>, SUB<RReal, RReal>>>
```

and the function pointer for the statement is generated with:

```
evalRevFunc<MULT_PA<DIV<SIN<ADD<RReal, RReal>>, SUB<RReal, RReal>>>>
```

Table 2 lists the required data for the statement. In total 49 bytes are required. If no expression templates are used, the intermediate values would also be stored. This would require 98 bytes while assuming that instead of function pointers an enumeration of one byte is used to identify the operations.

8. Results

8.1. Coupled Burgers equations

The coupled Burgers equations [6]

$$u_t + uu_x + vu_y = \frac{1}{R}(u_{xx} + u_{yy}) \tag{18}$$

$$v_t + uv_x + vv_y = \frac{1}{R}(v_{xx} + v_{yy}) \tag{19}$$

are chosen as a first test case. They can be used to rapidly evaluate how changes in CoDiPack [8] affect the performance of various tape implementations.

The coupled burgers equations are discretized with an upwind finite difference scheme. The initial conditions are

$$u(x, y, 0) = x + y \quad (x, y) \in D \tag{20}$$

$$v(x, y, 0) = x - y \quad (x, y) \in D \tag{21}$$

and the exact solution is after [6],

$$u(x, y, t) = \frac{x + y - 2xt}{1 - 2t^2} \quad (x, y, t) \in D \times \mathbb{R} \tag{22}$$

$$v(x, y, t) = \frac{x + y - 2xt}{1 - 2t^2} \quad (x, y, t) \in D \times \mathbb{R}. \tag{23}$$

The computational domain D is the unit square $D = [0, 1] \times [0, 1] \subset \mathbb{R} \times \mathbb{R}$ and the boundary conditions are taken from the exact solution. For the test runs a grid size of

601 × 601 grid points and 32 discrete time steps are used. The computations are performed on the Elwetritsch cluster of the TU Kaiserslautern and the test case is evaluated on one node of the cluster which consists of two Intel E5-2640v3 processors. Two load cases are considered. For the first case, only one process is run. For the second, the sequential program is run on each of the 16 cores simultaneously. This simulates an environment where the memory bandwidth of the node is fully utilized. The two cases are called 'sequential' and 'bandwidth limited' in the further analysis. Smaller test cases with a grid size of 61 × 61 or 21 × 21 grid points have also been considered, but the results are qualitatively the same.

As a comparison the AD tools Adept [13] and ADOL-c [18] are run with the same configurations. For Adept we adhered to the 'tips for the best performance' section in the 'Adept User Guide' [1]. In the case of ADOL-c we configured the *.adolcrc* file such that no tapes are written to files. In terms of comparison Adept implements a Jacobian taping approach with an index reuse scheme that uses expression templates. ADOL-c implements a primal value taping approach also with an index reuse scheme. Here, no expression templates are used.

The tape memory for the coupled Burgers equation using all four taping approaches is shown in Table 3. When the primal value taping is used, the required memory can be reduced with respect to the Jacobian taping by 17% or 20% if the indices are reused.

Figure 1 shows the timing results for all tapes and test cases. The test case with one process is not memory bandwidth limited and therefore the evaluation time of the Jacobian tapes is faster then for the primal value taping. The time for the recording of the tapes with one process is nearly the same for all tapes. Only the primal value tape with the index reuse has a slightly higher recording time than the other tapes, which comes from the increased complexity of the evaluation.

Adept has nearly the same performance as the Jacobian tape with index reuse from CoDiPack. Both tools use the same expression template technique for the implementation which yields similar results. The memory consumption is slightly larger for Adept since the tool assumes an arbitrary number of arguments for each expression.

The comparison with ADOL-c cannot be done directly. ADOL-c does not use expression templates which requires the tool to store the data for all intermediate values. This explains the additional 3 Gb of data for the ADOL-c tape. If the memory bandwidth is the major factor for the time discrepancy with respect to the expression template AD tools, the evaluation time of ADOL-c should only have a factor of two. Why the factors are higher can partly be explained by the additional loss of structure in the tape. That is, the elemental operations in an expression template tape handle complete statements instead of binary and unary operators in an operator overloading tape. Therefore the compiler can apply more optimizations. In addition, the the number of random memory accesses is increased since all intermediate values are handled separately.

Table 3. Memory requirements of the the coupled Burgers equation for the primal value approach and the Jacobian approach.

	Jacobian lin. ind.	Jacobian ind. reuse	Primal lin. ind.	Primal ind. reuse	Adept	ADOL-c
Memory in MB	4830	4496	4030	3853	4796	7119
Reduction (CoDiPack)	0 %	7 %	17 %	20 %	–	–
Reduction (other tools)	–	36 %	–	45 %	32 %	0 %

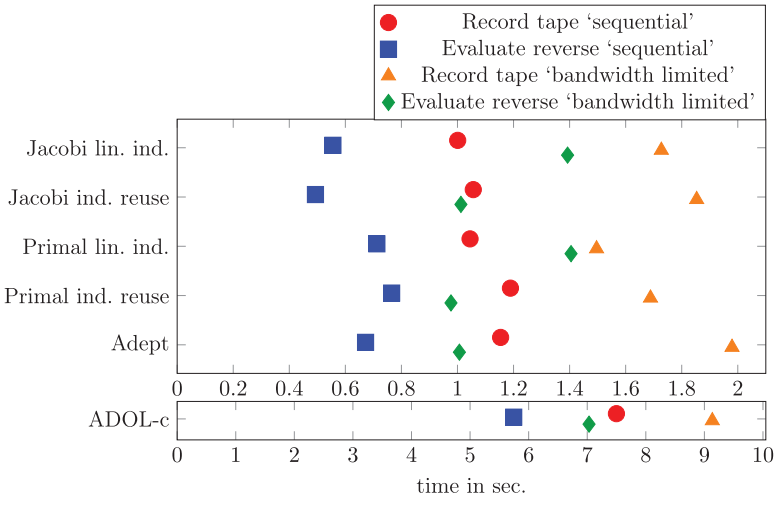

Figure 1. Comparison of the evaluation times of the coupled Burgers equation for the primal value approach and the Jacobian approach. Adept implements a Jacobian index reuse scheme. ADOL-c implements a primal index reuse scheme without expression templates. (Compiler options: `-O3 -DCODI_UseForcedInlines`).

If the results for the memory bandwidth limited case are analysed, the relations are reversed. The recording of the Jacobian tapes takes longer than the recording of the primal value tapes. Since all 16 processes use the memory bandwidth at the same time, it becomes the bottleneck and the reduced memory size of the primal value tapes causes the faster recording. For the tape evaluation, the times for Jacobian taping and primal value taping are the same. There is no difference, because the computation of the gradients in the reverse mode requires more complexity and therefore increases the time for evaluation. This cancels out any effect of the reduced memory for the primal value tapes.

It can be seen that the Jacobian taping and primal value taping approaches are in the same complexity region and no method has a distinct advantage over the other one. If a process is memory bound, which is usually the case in high performance computations, primal value tapes have an advantage over Jacobian tapes due to the reduced memory requirements.

8.2. SU2 with a proof of concept implementation

An important application, where derivatives are nowadays frequently needed, is numerical optimization. When constraints are only implicitly defined using partial differential equations (PDE), this usually requires efficient adjoint methods. In that case AD facilitates the development of these solvers in the discrete setting. This is especially valuable in computational fluid dynamics, where the analysis, i.e. the evaluation of the constraining PDE, is achieved by highly complex algorithms.

Recently, *CoDiPack* was applied by the authors to the open-source framework *SU2*. The latter is a collection of tools including the analysis and optimization of internal and external aerodynamic problems. The differentiated code was used to generate a flexible and robust discrete adjoint solver for the Reynolds Averaged Navier–Stokes equations [2], optionally

coupled with the Ffowcs–Williams–Hawkings equation [19] for aeroacoustic optimizations. The adjoint solver is based on the fixed-point formulation of the underlying solver for the discretized state equation [11]. That is, we assume that feasible solutions U^* of the state equation

$$U = G(U, X) \tag{24}$$

are computed by the iteration $U^{n+1} = G(U^n, X)$ for $n \to \infty$. X represents the design variables and $G(U)$ some (pseudo) time-stepping scheme like the explicit or implicit Euler method. By stating the first-order necessary optimality conditions to minimize some scalar objective function $J(U, X)$, with the constraint that the state equation (24) is fulfilled, we end up with the following equation for the adjoint state $\bar{U}$:

$$\bar{U} = \left[\frac{\partial J(U,X)}{\partial U}\right]^{\mathrm{T}} + \left[\frac{\partial G(U,X)}{\partial U}\right]^{\mathrm{T}} \bar{U}. \tag{25}$$

This equation can be solved using the iterative scheme

$$\bar{U}^{n+1} = \left[\frac{\partial J(U^*,X)}{\partial U}\right]^{\mathrm{T}} + \left[\frac{\partial G(U^*,X)}{\partial U}\right]^{\mathrm{T}} \bar{U}^n, \quad \text{for } n \to \infty. \tag{26}$$

Hence, we can easily construct the adjoint solver by simply applying AD to the code that computes J and G. In this paper, we want to generate the adjoint solver not only with the Jacobian tape, but also with the primal value tape. The current implementation is a proof of concept that enables SU2 to work with all 4 tapes. Therefore, all memory optimizations like preaccumulation and function checkpoints, are disabled and the MPI communication is not yet implemented for the primal value tapes. The computation of the test cases uses therefore an explicit scheme and only one process. The presented timing results are not representative for the usual performance of the adjoint solver in the SU2 code and are only used to compare the different taping strategies. It is also important to note, that since we need the rhs of Equation (26) repeatedly at the fixed state U^*, we only need to record the tape once at the beginning of the iteration. Subsequently, only the interpretation of the stored information is required. This is why for this application we are most interested in the time of the interpretation step. As a test case we consider the viscous laminar flow over the NACA0012 aerofoil. As free-stream values we consider a Mach number of 0.5, a angle of attack of 1.0° and a Reynolds number of 5000. The computational mesh consists of 152,068 elements. For the spatial integration we use a Roe upwind scheme in combination with an explicit Euler method for the pseudo-time stepping.

The results for the SU2 code show that the primal value taping can also be applied to large codes, but due to the structure of the code there are not yet any gains for primal value taping. Figure 2 shows timing results for the SU2 test case. The recording time for linear indexing schemes of the Jacobian taping and primal value taping approach is almost the same. In contrast to this, the times for the index reusing schemes are larger due to the increase in complexity of the index management. But these values are not that important because in SU2 the tape is usually recorded only once and then evaluated multiple times, which makes the evaluation times of the tape much more important.

It can be seen in Figure 2 that the evaluation times for the Jacobian tapes are nearly the same. The primal value tape with the linear indexing requires the same time as the Jacobian

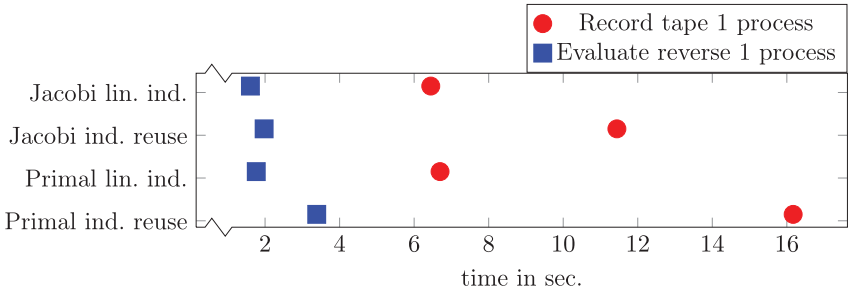

Figure 2. Comparison of the evaluation times of the SU2 test case for the primal value approach and the Jacobian approach. The values are taken from a proof of concept implementation that runs only single threaded and has no AD specific optimizations enabled, they do not reflect the usual performance of the discrete adjoint in SU2.

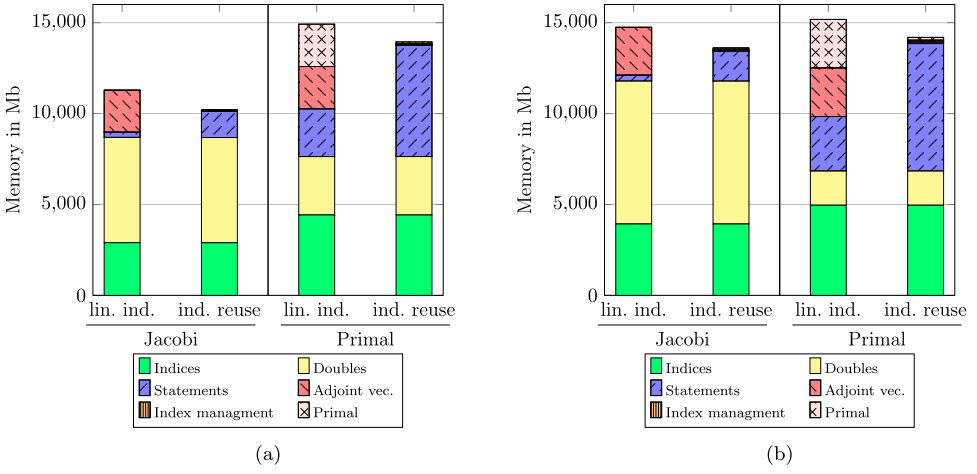

Figure 3. Comparison of the memory requirements of the SU2 test case for the primal value approach and the Jacobian approach. The values are taken from a proof of concept implementation that runs only single threaded and has no AD specific optimizations enabled, they do not reflect the usual performance of the discrete adjoint in SU2. (a) Tape for the state and (b) Tape for the state and the design.

tapes. This indicates that the process is already memory bound. Yet, the evaluation time for the primal value tape with index reuse is twice as large. One possible reason for this behaviour is the increased memory of the primal value tapes, which is probably also the reason why the evaluation time of the index management schemes is not lower as it is seen in the Burgers test case.

The memory of the different approaches is displayed in Figure 3(a). It can be seen that the Jacobian tapes require approximately 3 GB less memory than the primal value tapes. One of the main reasons for this is the larger statement size and the storing of more indices. All four tapes have recorded the same number of statements, but the memory per statement is different for each taping approach.

The difference in the number of stored indices comes from the different handling of passive values for the Jacobian tapes and the primal value tapes. The Jacobian tapes ignore passive values and do not need to store the associated index and the Jacobian value. For the

primal value tapes the situation is different, the expression handles require that all arguments for the expression are available. For passive arguments the associated index and value needs to be stored and can not be left out.

On the other hand, if fewer passive values are used in the code, the memory consumption of the primal value tapes is on the same level as the Jacobian taping. Figure 3(b) shows the tape memory, when the design variables X are also registered as an input in addition to the state variables U. Since the state and the design are now inputs, the number of passive variables decreases. If the memory of the Jacobian tapes is compared to the memory when only the state is taped, it has increased by 3 GB. This increase comes from additional variables that depend on the design. However, the number of statements is nearly the same. The memory for the primal value tapes increases only slightly. They already needed to record most of the data, that was previously not dependent on any input value. Between the Jacobian taping and the primal value taping there is still a discrepancy in the number of recorded indices, which indicates that there are some regions in the SU2 code, that are not dependent on the state or design variables.

9. Conclusion and outlook

The results presented in this paper, show that the extension of the primal value approach with expression templates can reduce the required amount of memory for the burgers test case. The evaluation speed for the reverse interpretation stays the same for this case in a memory bandwidth limited scenario. The recording time for the same setting is improved with respect to the Jacobian approach. For applications that are specifically written for AD and for which care has been taken to exclude passive calculations the same results are to be expected. For large codes, the situation is not clear. The prototype for the SU2 code shows, that if an application has a lot of passive computations, the primal value approach needs to store more data than the Jacobian approach. Nevertheless, the evaluation times for all tapes are nearly the same which shows that both methods have the same complexity. However, in order for the primal taping approach to be competitive with the Jacobian approach we must work on improving the data management and the implementation.

For example if the management of the passive values can be improved, the overall memory for the primal value taping will be reduced, which will result in improved evaluation and recording times. It is also important to review the current implementation and to check for methods to reduce the required memory per statement. The expression handles require 8 bytes for each statement, which can probably be reduced with improved data management and storing techniques.

Disclosure statement

No potential conflict of interest was reported by the authors.

References

[1] *Adept User Guide*. Available at http://www.met.reading.ac.uk/clouds/adept/documentation.html.

[2] T.A. Albring, M. Sagebaum, and N.R. Gauger, *Development of a consistent discrete adjoint solver in an evolving aerodynamic design framework*, 16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA Aviation, American Institute of Aeronautics and Astronautics, 2015. Available at http://dx.doi.org/10.2514/6.2015-3240.
[3] P. Aubert and N. Di Césaré, *Expression templates and forward mode automatic differentiation*, in *Automatic Differentiation of Algorithms: From Simulation to Optimization*, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, eds., Springer New York, New York, NY, 2002, pp. 311–315. Available at http://dx.doi.org/10.1007/978-1-4613-0075-5_37.
[4] B.M. Bell and J.V. Burke, *Algorithmic differentiation of implicit functions and optimal values*, in *Advances in Automatic Differentiation*, C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, and J. Utke, eds., Springer, Berlin, Heidelberg, 2008, pp. 67–77.
[5] C. Bendtsen and O. Stauning, *FADBAD, a flexible C++ package for automatic differentiation*, Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1996.
[6] J. Biazar and H. Aminikhah, *Exact and numerical solutions for non-linear Burger's equation by VIM*, Math. Comput. Model. 49 (2009), pp. 1394–1400.
[7] T. Budd, *Multiparadigm Programming in Leda*, Addison-Wesley, Reading, MA, 1995.
[8] *CoDiPack AD Tool*. Available at http://www.scicomp.uni-kl.de/software/codi.
[9] J.O. Coplien, *Curiously recurring template patterns*, C++ Report 7 (1995), pp. 24–27.
[10] T.D. Economon, F. Palacios, S.R. Copeland, T.W. Lukaczyk, and J.J. Alonso, *SU2: An open-source suite for multiphysics simulation and design*, AIAA J. 54 (2015), pp. 828–846.
[11] M.B. Giles, D.P. Ghate, and M.C. Duta, *Using automatic differentiation for adjoint CFD code development*, Post SAROD Workshop, 2005.
[12] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Frontiers in Appl. Math., vol. 19, SIAM, Philadelphia, 2000.
[13] R.J. Hogan, *Fast reverse-mode automatic differentiation using expression templates in C++*, ACM Trans. Math. Softw. 40 (2014), p. 26.
[14] D.R. Musser, G.J. Derge, and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 3rd ed., Addison-Wesley Professional, Boston, MA, USA, 2009.
[15] U. Naumann, *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation, Vol. 24*, Siam, Philadelphia, PA, USA, 2012.
[16] E.T. Phipps, R.A. Bartlett, D.M. Gay, and R.J. Hoekstra, *Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation*, in *Advances in Automatic Differentiation*, C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, and J. Utke, eds., Springer, Berlin, Heidelberg, 2008, pp. 351–362.
[17] T. Veldhuizen, *Expression templates*, C++ Report 7 (1995), pp. 26–31.
[18] A. Walther and A. Griewank, *Getting started with ADOL-C*, in *Combinatorial Scientific Computing*, U. Naumann and O. Schenk, eds., Chapman-Hall CRC Computational Science, London, UK, 2009, pp. 181–202.
[19] B. Zhou, T. Albring, N. Gauger, C.I. da Silva, T. Economon, and J.J. Alonso, *An efficient unsteady aerodynamic and aeroacoustic design framework using discrete adjoint*, 17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA 2016-3369, 2016.