High-Performance Derivative Computations using CoDiPack

MAX SAGEBAUM, TIM ALBRING, and NICOLAS R. GAUGER, TU Kaiserslautern

There are several AD tools available that all implement different strategies for the reverse mode of AD. The most common strategies are primal value taping (implemented e.g. by ADOL-C) and Jacobian taping (implemented e.g. by Adept and dco/c++). Particulary for Jacobian taping, recent advances using expression templates make it very attractive for large scale software. However, the current implementations are either closed source or miss essential features and flexibility. Therefore, we present the new AD tool CoDiPack (Code Differentiation Package) in this paper. It is specifically designed for minimal memory consumption and optimal runtime, such that it can be used for the differentiation of large scale software. An essential part of the design of CoDiPack is the modular layout and the recursive data structures which not only allow the efficient implementation of the Jacobian taping approach but will also enable other approaches like the primal value taping or new research ideas. We will finally present the performance values of CoDiPack on a generic PDE example and on the SU2 code.

CCS Concepts: • **Mathematics of computing** → **Automatic differentiation**; *Mathematical software performance*; • **Software and its engineering** → *Software design engineering*; *Software design tradeoffs*;

Additional Key Words and Phrases: Algorithmic differentiation, expression templates, recursive data structures, efficient implementation, maintainable implementation

ACM Reference format:
Max Sagebaum, Tim Albring, and Nicolas R. Gauger. 2019. High-Performance Derivative Computations using CoDiPack. *ACM Trans. Math. Softw.* 45, 4, Article 38 (December 2019), 26 pages.
https://doi.org/10.1145/3356900

1 INTRODUCTION

Algorithmic Differentiation (AD) describes the mathematical theory of how a computer program can be differentiated. A basic introduction to AD will be given in the second section of this article. However, we already state here the fundamental result of the reverse mode of AD, namely, that for each statement $y = w(x)$ with $w : \mathbb{R}^n \to \mathbb{R}^m$ in the software, the corresponding adjoint statement

$$\bar{x} \mathrel{+}= \left(\frac{dw}{dx}\right)^T \bar{y}; \quad \bar{y} = 0 \tag{1}$$

Funding was received within the project "Enabling Performance Engineering in Hesse and Rhineland-Palatinate" from the 'Deutsche Forschungsgemeinschaft' (DFG; grant number 320898076).
Authors' addresses: M. Sagebaum, T. Albring, and N. R. Gauger, TU Kaiserslautern, Bldg 34, Paul-Ehrlich-Strasse, 67663 Kaiserslautern, Germany; emails: {max.sagebaum, tim.albring, nicolas.gauger}@scicomp.uni-kl.de.

0098-3500/2019/12-ART38 $15.00
https://doi.org/10.1145/3356900

Table 1. Overview of AD Tools for C++ (nonexhaustive list)

Tool	Taping Method(s)	Implementation	License	Other Features
CoDiPack	Jacobian, Primal(*)	OO/ET	Open source (GPL3)	HO, VM, TL-FM
ADOL-C	Primal	OO	Open source (EPL/GPL2)	HO, VM, TL-FM, TB-FM
Adept	Jacobian	OO/ET	Open source (Apache2.0)	TB-FM
dco/c++	Jacobian	OO/ET	proprietary	HO, VM, TL-FM, TB-FM
CppAD	Jacobian	OO	Open source (EPL/GPL2)	HO, TL-FM, TB-FM
Sacado	Jacobian	OO/(ET, only FM)	Open source (LGPL2)	HO, VM, TL-FM, TB-FM

OO: Operator Overloading, ET: Expression Templates.
HO: Higher-Order, VM: Vector-Mode, TL-FM: Tapeless Forward Mode, TB-FM: Tape-Based Forward-Mode.
(*)as of version 1.3, not covered in this work.

must be evaluated. When the software is written in Fortran, source-code transformation [Griewank and Walther 2008] is normally the method of choice to automate the generation of code to compute (1). The application of source-code transformation to C++ code is limited owing to the complex language structure. Instead, Operator Overloading (OO) has been proven to be more appropriate. All available tools that implement AD based on the OO approach have to deal with the problem of storing the data and evaluating Equation (1) in a number of different ways depending on the programming language, field of application, or just the personal preference. However, they all have in common that, since the flow of data is reversed, it is required to either store the values of x or to store the Jacobian $\frac{dw}{dx}$ itself during the evaluation. This corresponds to what we refer to as the *primal value taping* and the *Jacobian taping* methods, respectively.

The most well-known representative implementing the primal value taping method is *ADOL-C* [Walther and Griewank 2009]. It is released under the Eclipse Public License (EPL) and GPL2 license and is widely used in science for small and medium-sized applications. Its shortcomings are the runtime and memory overhead when applied to large-scale problems. Still, its level of maturity and robustness makes it an excellent aid for validation and verification when developing new OO AD tools.

The *Jacobian taping* method is quite popular, especially in combination with expression templates (ETs) [Aubert et al. 2001; Veldhuizen 1995] to increase runtime efficiency. Two examples of tools that make use of this approach are *Adept* [Hogan 2014] and *dco/c++* [Leppkes et al. 2016]. *Adept* is released as open-source and offers high performance and a relatively low memory footprint. However, it lacks important features such as higher-order differentiation, a vector-mode, or a tapeless forward mode implementation. Although *dco/c++* offers all of these features, we think that its proprietary license makes it unattractive for individuals and organizations in science as well as industry to use it in their in-house or open-source software. Other tools, such as CppAD [Bell and Burke 2008] and Sacado [Phipps et al. 2008], also support the features mentioned above, but they do not use ETs for the evaluation of the Jacobians in reverse mode. However, for the latter, there is an experimental forward-mode implementation using ETs available [Phipps and Pawlowski 2012]. In our opinion, all tools have in common that their structure is quite inflexible so that they are not easily extensible. These were the initial reasons for starting the development of the *Code Differentiation Package* (CoDiPack). Table 1 shows a comparison of taping methods, implementation, license, and additional features of the tools mentioned above.

The main focus of *CoDiPack* is enabling gradient computations for high-performance computing (HPC) and industrial-grade software while still maintaining its open-source philosophy. To date, publications where *CoDiPack* is applied include a number of different applications in engineering research [Hück et al. 2018; Nørgaard et al. 2017; Sanchez et al. 2018; Schwalbach et al. 2018] and

Table 2. Evaluation Procedure for a Function f

v_i	$= x_i,$	$i =$	$1 \dots n$
v_{i+n}	$= \varphi_i(u_i),$	$i =$	$1 \dots l$
y_i	$= v_{n+l-i+1},$	$i =$	$1 \dots m$

industry [Economon 2018; Luers et al. 2018] as well as performance comparisons with other tools [Hück et al. 2017; Lubkoll 2017].

The initial design of the expression structure in CoDiPack is based on the one from *Adept* [Hogan 2014]. This yields a very similar calling convention and naming of basic methods, but the implementation and generation of the expression class is handled differently. Nevertheless, we want to thank R. Hogan for providing *Adept* as open-source software, which gave us the initial ideas for CoDiPack.

This article serves the purpose of highlighting in more detail the structure of the implementation and showing how it affects the resulting performance. First, an introduction to AD is given in Section 2, which is followed by an explanation of ETs in Section 3. The modular approach of the CoDiPack implementation is described in Section 4. Here, the basic idea for the efficient application of AD through OO is described and the design choices for the efficient and extensible memory structures are explained. The performance is investigated by applying *CoDiPack* to a generic PDE example and to the open-source computational fluid dynamics suite SU2 [Economon et al. 2015] in Section 6.

2 ALGORITHMIC DIFFERENTIATION

We assume that the part in the software for which the derivatives are sought can be described as a function with the header *void func(const double x[], double y[])*. In most cases, the input variables x and the output variables y will be not straight arrays but instead collections of several variables. These variables can also be members of structures that are in a local or global scope. In an extreme case, the header for *func* will be just *void func(void)*, but internally *func* accesses millions of input and output variables from the global scope. Nevertheless, *func* can always be described as the mathematical function $f : \mathbb{R}^n \to \mathbb{R}^m$ with

$$y = f(x). \tag{2}$$

We assume that the numerical evaluation of f can be represented as a sequence of l statements $\varphi_i : \mathbb{R}^{n_i} \mapsto \mathbb{R}$. Each of these statements represents a local evaluation procedure with an arbitrary complex right-hand side, for example,

$$v_4 = \varphi_4(v_1, v_2, v_3) = -10 \times v_2 \times \exp(v_3) + \ln(v_1) - 3 \times 10^7 \times v_3 \times (v_2 - 1) \times \sqrt{v_1}. \tag{3}$$

It is then possible to write any numerical evaluation in an arbitrary computer program using the general procedure shown in Table 2. We have that $u_i := (v_j)_{j \prec i} \in \mathbb{R}^{n_i}$, where the precedence relation $j \prec i$ means that variable v_i depends directly on variable v_j. That is, the vector u_i is the concatenation of the v_j on which φ_i depends. It is important to note that the mathematical representation of f and the implementation *func* is not a one-to-one relation. f represents all statements that are evaluated during one call of *func*, which implies that, e.g., loops are unrolled in f. For large applications, the number of statements l can become quite large and can vary for different configurations.

Table 3. Tangent Interpretation (Forward Mode of AD)

$\dot{v}_i$	$= \dot{x}_i,$	$i =$	$1 \dots n$
$\dot{v}_{i+n}$	$= \sum_{j<i} \frac{\partial}{\partial v_j} \varphi_i(u_i) \dot{v}_j,$	$i =$	$1 \dots l$
$\dot{y}_i$	$= \dot{v}_{n+l-i+1},$	$i =$	$1 \dots m$

Table 4. Adjoint Interpretation (Reverse Mode of AD)

$\bar{v}_j$	$= 0$	$j = 1 \dots n + l - m$
$\bar{v}_{n+l-i+1}$	$= \bar{y}_i$	$i = m \dots 1$
$\bar{v}_j$	$= \bar{v}_j + \bar{v}_{i+n} \cdot \frac{\partial}{\partial v_j} \varphi_i(u_i),\ \bar{v}_{i+n} = 0$	$j < i, i = l \dots 1$
$\bar{x}_i$	$= \bar{v}_i$	$i = n \dots 1$

The representation of f can also be written as the composition

$$f(x) = Q_m \circ \Phi_l \circ \Phi_{l-1} \circ \cdots \circ \Phi_2 \circ \Phi_1 \circ P_n^T(x), \tag{4}$$

where the state transformation $\Phi_i : V \mapsto V$ sets v_{i+n} to $\varphi_i(v_j)_{j<i}$ and keeps all other components v_j for $i + n \neq j$ unchanged. $V := \mathbb{R}^n \times \mathbb{R}^l$ is the state space of the evaluation of *func*. It contains the input values x, the intermediate values and the output values y as the last m values of the intermediate values. $P_n \in \mathbb{R}^{n \times (n+l)}$ and $Q_m \in \mathbb{R}^{m \times (n+l)}$ are the matrices that project an $(n + l)$-vector onto its first n and last m components, respectively. By applying the chain rule for differentiation, we get that

$$\dot{y} = Q_m A_l A_{l-1} \dots A_2 A_1 P_n^T \dot{x}, \tag{5}$$

where $A_i := \nabla \Phi_i$. Using the analytic representation of A_i [Griewank and Walther 2008], it is possible to write the tangent relation in Equation (5) as the evaluation procedure shown in Table 3. The matrix vector products are calculated in the same order as for the evaluation procedure in Table 2 and they can be computed alongside the primal evaluation. It is then possible to compute the matrix-vector product of the Jacobian and an arbitrary direction $\dot{x}$, i.e., Equation (5), by evaluating the tangent interpretation.

The tangent relation in Equation (5) also gives the alternative representation of the Jacobian of f, which can be written as

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = Q_m A_l A_{l-1} \dots A_2 A_1 P_n^T. \tag{6}$$

By transposing the product, we obtain the adjoint relation

$$\bar{x} = P_n A_1^T A_2^T \dots A_{l-1}^T A_l^T Q_m^T \bar{y} = \left(\frac{\mathrm{d}f}{\mathrm{d}x} \right)^T \bar{y}. \tag{7}$$

We can again use the analytic representation of A_i [Griewank and Walther 2008] to write the adjoint relation as the evaluation procedure shown in Table 4. All matrix-vector products are calculated for $i = l, l-1, \dots, 1$; thus, we have to go in reverse through the sequence of statements in Table 2. We can then get the matrix-vector product involving the transposed Jacobian and an arbitrary seed vector $\bar{y}$, i.e., Equation (7), by evaluating this adjoint interpretation.

```
1  template<typename Impl>
2  struct Expression {
3    Impl& cast() { return static_cast<Impl&>(*this);}
4    double value() { return this->cast().value();}
5  };
6
7  template<typename A, typename B>
8  struct MULT : public Expression<MULT<A, B> > {
9    const A& a;
10   const B& b;
11
12   MULT(const A& a, const B& b) : a(a), b(b) {}
13   double value() { return a.value() * b.value();}
14 }
15
16 template<typename A, typename B>
17 MULT<A,B> operator * (const Expression<A>& a, const Expression<B>& b) {
18   return MULT<A,B>(a, b);
19 }
```

Fig. 1. Expression template example.

3 EXPRESSION TEMPLATES

One big issue for an efficient OO AD tool implementation is that C++ supports the overloading of only unary and binary operators. A regular implementation that follows the layout $Real \circ Real \rightarrow Real$, where *Real* is the AD type, would split the statement

$$w = ((a + b) * (c - d))^2 \tag{8}$$

into the intermediate statements

$$\begin{aligned} t_1 &= a + b \\ t_2 &= c - d \\ t_3 &= t_1 * t_2 \\ w &= t_3^2. \end{aligned}$$

The Jacobian taping approach stores for every statement the Jacobian $\frac{\partial}{\partial v_j}\varphi_i(u_i)$ from Table 4 in order to evaluate the reverse procedure. For the procedure with the intermediate statements, 4 Jacobians with in total 7 entries and 4 statements must be stored for the evaluation. However, the original statement in Equation (8) needs only 1 Jacobian with 4 entries and 1 statement. Therefore, it would be much more efficient to treat the whole statement at once. In order to do that, we change the layout of the operator to $Expr_A \circ Expr_B \rightarrow Expr_{A \circ B}$, where *Expr* is some class that stores information about the particular operation. This information can then be used to compute the partial derivatives of the whole statement by individually calling the functions that compute the partial derivatives of each single expression involved. However, if a regular inheritance scheme is used, virtual function calls would be required for every statement of the program, which would drastically increase execution time.

The ET technique [Aubert et al. 2001; Veldhuizen 1995] avoids the use of virtual functions and uses static polymorphism as well as the curiously recurring template pattern for the inheritance. The basic idea is to define a template class that has the extending class as a template argument. Figure 1 shows this as an example. The base class provides a cast method in line 3 to return a reference to the extending class. Every method in the base class uses this method to call the implementation of the extending class, which removes all virtual function calls since the inheritance relation is resolved at compile-time. In line 8, the structure *MULT* is implemented and the

```
1  // general recursive implementation
2  void calcGradient(const double& multiplier) {
3    double dw_da = derivativeA(a.value(), b.value(), this->value()) * multiplier;
4    double dw_db = derivativeB(a.value(), b.value(), this->value()) * multiplier;
5
6    a.calcGradient(dw_da);
7    b.calcGradient(dw_db);
8  }
9
10 // terminating implementation in variables
11 void calcGradient(const double& multiplier) {
12   globalTape.pushJacobi(multiplier, this->index);
13 }
```

Fig. 2. Implementation for the Jacobian computation in an expression template.

Expression interface is used as a base class. The layout change mentioned above is seen in the overload of the multiplication operator in line 17. Two general expressions are provided to the operator and the specialized *MULT* expression is returned. Assuming that the type for the variable is called *Real*, then the statement (8) is represented by the structure

$$\text{SQUARE<MULT<ADD<Real, Real>, SUB<Real, Real>>>}. \tag{9}$$

If an implementation similar to the *MULT* class also exists for the *SQUARE*, *ADD*, and *SUB* structures, then a call to the *value()* routine using a reference of type *Expression* evaluates and returns its value.

For a Jacobian taping approach, we follow the ideas and implementation of Hogan [Hogan 2014]. Since it is now necessary to compute the derivatives of the statement, the *Expression* interface from Figure 1 needs to be extended to provide this information. The implementation will be the AD reverse mode of the *value* method in the *MULT* expression example. The *value* method can be viewed as the algorithm

$$\begin{aligned} a &= A(p) \\ b &= B(q) \\ w &= a * b, \end{aligned} \tag{10}$$

where A and B represent the expressions of the two arguments from the multiplication. $p \in \mathbb{R}^s$ and $q \in \mathbb{R}^t$ represent the variables that are used in the expressions. If reverse AD is applied to this algorithm, then the resulting algorithm is

$$\begin{aligned} \bar{a} &= b * \bar{w} \\ \bar{b} &= a * \bar{w} \\ \bar{q} &= \frac{dB}{dq} * \bar{b} \\ \bar{p} &= \frac{dA}{dp} * \bar{a}. \end{aligned} \tag{11}$$

The reverse algorithm states that the Jacobian values with respect to the arguments need to be computed; then, the adjoint values are applied recursively to the arguments. Therefore, we extend the *Expression* interface with a new method *calcGradient* that implements this algorithm. Figure 2 shows a general implementation for an arbitrary binary operator as well as the termination of the recursion. The two methods, *derivativeA* and *derivativeB*, calculate the derivatives with respect to the first and second argument, i.e., they evaluate the first two lines in algorithm (11). For the last two lines, it is now necessary to do recursive calls of *calcGradient* on the arguments of the expression. The recursion terminates at the variables of the expression, where the *multiplier* argument of

```
1   double add = a.value() + b.value();
2   double sub = c.value() - d.value();
3   double mul = add * sub;
4   double w = pow(mul, 2.0);
5
6   double w_b = multiplier; // = 1.0
7   double mul_b = 2 * mul * w_b;
8   double sub_b = add * mul_b;
9   double add_b = sub * mul_b;
10
11  c.calcGradient(sub_b);
12  d.calcGradient(-sub_b);
13  a.calcGradient(add_b);
14  b.calcGradient(add_b);
```

Fig. 3. The code that a compiler should generate when *calcGradient* is called on statement (8).

the method will contain the derivative of the whole expression with respect to this argument. The function in line 11 of Figure 2 shows the implementation for the variables. Here, the information is pushed to the global tape. For the example statement (8), a call of *calcGradient(1.0)* on the expression template will evaluate the code in Figure 3. This should also be the code that is generated by the compiler after everything is inlined.

The remaining calls to *calcGradient* are then implemented such that they access the tape and store the data. This will be covered in the next section.

4 DESIGN AND LAYOUT OF CODIPACK

Since the basic expression layout and implementation is based on Adept [Hogan 2014], the expression interface of CoDiPack is very similar to the one of Adept. The actual data required per statement is also similar and will "only" save 1 byte per statement. Although this might seem to be a modest saving, it actually reflects a 15% to 5% reduction in tape memory for statements with 1 to 4 arguments, respectively. The first major difference comes from the implementation of the data stores. CoDiPack introduces here a general recursive structure such that multiple data streams can be handled easily and additional data streams can be added in further developments. A second important difference arises in the implementation of the expressions; the super macro technique [Beal 2004] in CoDiPack allows for an improved debugging of the ET code. The implementation also avoids other kinds of preprocessor directives to improve readability. Other improvements with respect to Adept are a more template-oriented implementation and the strict separation of the tape implementations and floating point replacement type.

As it is stated in the introduction, one of the main goals of CoDiPack is to be as fast and memory efficient as possible in order to be able to use it in HPC environments. Therefore, which data is stored needs to be carefully considered, along with how storage will be done.

The data requirements can be analyzed with the help of the reverse AD equation for a statement, as shown in Table 4:

$$\bar{v}_j = \bar{v}_j + \bar{v}_{i+n} \cdot \frac{\partial}{\partial v_j} \varphi_i(u_i),\ \bar{v}_{i+n} = 0 \quad j < i, i = l \ldots 1. \tag{12}$$

In order to evaluate the *Jacobian taping* approach for this equation, the required data consists of the Jacobian $\frac{d\varphi}{dv}$ and the means to access the adjoint variables $\bar{u}_i$ and $\bar{v}_{i+n}$.

The adjoint variables could be directly stored in the AD type; however, this is not a feasible solution since this information would be deleted when they run out of scope. It is common practice to use the identification that is provided by the AD theory. When the function f is separated into the elemental operations φ_i, every operation gets an index i that runs from one to the maximum

number of operations l. Therefore, each variable can be identified with the index of the elemental operation. This index is implemented as a global counter, which is incremented each time a statement is stored in the AD tool. Then, the current value of the counter is used as the index for the value on the left-hand side of the assignment.

As a result, the memory for the data of one elemental operation with k input values is k double values for the Jacobian, k indices for the arguments and one index for the output value. In addition, 1 byte is needed to store the number of arguments. This assumes that a statement has no more than 255 arguments, which is a reasonable assumption for normal code. The total memory requirement for each statement is then $k * 8 + (k + 1) * 4 + 1 = 12 * k + 5$ bytes, but this is not yet optimal.

The indexing scheme that we use increases the index of the left-hand side by 1 for each statement, and the index is stored directly for the reverse evaluation. During the reverse evaluation, all statements are evaluated in the exact same order but just reversed. The index of the left-hand side can be computed by decrementing it one by one; therefore, it is no longer necessary to store it. This reduces the memory requirement by 4 bytes, which is then $k * 8 + k * 4 + 1 = 12 * k + 1$ for each statement. We call this indexing scheme "linear indexing," which is also used by dco/c++ [Leppkes et al. 2016].

In addition to the data that we have to store for each statement, we add to CoDiPack the capability to evaluate user-defined functions. These are functions defined by the user that are evaluated during the reverse evaluation of the tape. They are required to add, for example, reverse MPI calls to the tape evaluation or to add improved algorithms for the differentiation of linear systems of equations. The data for user-defined functions is all data required by the user to evaluate the function. In addition, the position where the function should be evaluated is required. These two data items will be stored on the tape together with the information for the statements.

The next sections will introduce the efficient computation and storing of the required data for the Jacobian taping.

4.1 Design and Implementation of the Expression Templates

The computation of the Jacobian for the elemental operation is described by the ET implementation. The CoDiPack interface for them is shown in Figure 4. The *cast* method and the *getValue* method are the same as in Figure 1, but an additional template parameter for the class is introduced. It removes the constraint that only double types can be used in the computation such that, for example, higher-order derivatives can be computed. The *calcGradient* method is also extended by a template parameter that defines some user data that can be used in the computations. Furthermore, two versions of *calcGradient* are defined, a two-argument version and a version with one argument that assumes that the multiplier is equal to 1.0. This addition to the interface is made to give the compiler as much information as possible and to prevent unnecessary multiplications by 1.0.

The implementation of the interface for the unary operations is very straightforward and directly uses the code from Figures 1 and 2. For each operation, only the function *derivativeA* is different; therefore, a general template file *unaryExpression.tpp* is written and included in a super macro fashion. The template file expects that the macros *NAME*, *FUNCTION*, and *PRIMAL_FUNCTION* are defined. They provide the names of the structure, operator, and a function that calls the operator. From the name of the structure, the file derives the name *gradNAME* for the function that computes the derivative. In order to make it more convenient to use the template file multiple times, the three macros are undefined at the end of the file. The implementation of an unary operation is shown in Figure 5 for the sine.

For binary operators, the same principle can be used, but it requires more predefined functions. *binaryExpressions.tpp* expects the same predefined macros but also expects the functions

```
1  template<typename R, class A>
2  struct Expression {
3
4    static const bool storeAsReference;
5    typedef typename TypeTraits<R>::PassiveReal PassiveReal;
6
7    inline const A& cast() const {
8      return static_cast<const A&>(*this);
9    }
10
11   template<typename Data>
12   inline void calcGradient(Data& data) const {
13     cast().calcGradient(data);
14   }
15
16   template<typename Data>
17   inline void calcGradient(Data& data, const R& multiplier) const {
18     cast().calcGradient(data, multiplier);
19   }
20
21   inline const R getValue() const {
22     return cast().getValue();
23   }
24
25 private:
26   Expression& operator=(const Expression&) = delete;
27 };
```

Fig. 4. The interface definition for the expression templates in CoDiPack.

```
1  template<typename R> inline R gradSin(const R& a, const R& result) {
2    return cos(a);
3  }
4  using std::sin;
5  #define NAME Sin
6  #define FUNCTION sin
7  #define PRIMAL_FUNCTION sin
8  #include "unaryExpression.tpp"
```

Fig. 5. Expression implementation for the sinus function with the *unaryExpression.tpp* template file.

derv(11|11M|10|10M|01|01M)_NAME to be defined. They cover all possible cases of how a binary operator can be called: with a constant as the first argument, with a constant as the second argument, and for all cases where the multiplier is 1.0 or different. This requires more effort to implement a new binary operator but gives the developer all possible options for optimizations. An example implementation for the multiplication is shown in Figure 6. Macro definitions are avoided in the implementation as much as possible in order to ease debugging. With the chosen implementation, most of the code has a specific line and is not defined in a multiline macro.

4.2 Design of the Tape and Calculation Type

The missing ingredients are now the implementation of the calculation types that the users can employ in their software and the tapes that store the Jacobian data.

The calculation type is called *ActiveReal*, which represents the L-values in a program. This type has to implement the *Expression* interface; therefore, it acts as a termination point for the ETs. The question is now how much logic the *ActiveReal* structure should contain. If it contains some logic that is special for a specific tape implementation, then an extra *ActiveReal* implementation is needed for each tape, which would make it difficult to add new tapes. We opted for removing all logic from the *ActiveReal* implementation. It stores only the primal values for the computation and

```
1  template<typename Data, typename R, typename A, typename B>
2  inline void derv11_Multiply(Data& data, const A& a, const B& b, const R&
       result) {
3    a.calcGradient(data, b.getValue());
4    b.calcGradient(data, a.getValue());
5  }
6  template<typename Data, typename R, typename A, typename B>
7  inline void derv11M_Multiply(Data& data, const A& a, const B& b, const R&
       result, const R& multiplier) {
8    a.calcGradient(data, b.getValue() * multiplier);
9    b.calcGradient(data, a.getValue() * multiplier);
10 }
11 template<typename Data, typename R, typename A>
12 inline void derv10_Multiply(Data& data, const A& a, const typename
       TypeTraits<R>::PassiveReal& b, const R& result) {
13   a.calcGradient(data, b);
14 }
15 template<typename Data, typename R, typename A>
16 inline void derv10M_Multiply(Data& data, const A& a, const typename
       TypeTraits<R>::PassiveReal& b, const R& result, const R& multiplier) {
17   a.calcGradient(data, b * multiplier);
18 }
19 template<typename Data, typename R, typename B>
20 inline void derv01_Multiply(Data& data, const typename
       TypeTraits<R>::PassiveReal& a, const B& b, const R& result) {
21   b.calcGradient(data, a);
22 }
23 template<typename Data, typename R, typename B>
24 inline void derv01M_Multiply(Data& data, const typename
       TypeTraits<R>::PassiveReal& a, const B& b, const R& result, const R&
       multiplier) {
25   b.calcGradient(data, a * multiplier);
26 }
27 CODI_OPERATOR_HELPER(Multiply, *)
28 #define NAME Multiply
29 #define FUNCTION operator *
30 #define PRIMAL_FUNCTION primal_Multiply
31 #include "binaryExpression.tpp"
```

Fig. 6. Expression implementation for the multiplication with the *binaryExpression.tpp* template file.

the specific data for the tape. All function calls are forwarded to the tape interface. Because of this choice, no extra interface needs to be defined for *ActiveReal*. In addition, it defines the common convenience functions to retrieve the data.

The interfaces for the tapes are separated into one that is called from the *ActiveReal* and an interface for the user interaction. The interaction with *ActiveReals* requires functions for the handling of the tape-specific data and functions that trigger the storing of the statements. A statement is stored with the function *store*. Here, a tape implementation can access the data from the expression and perform the necessary actions for AD. In general, the store is called in every method, which assigns a value (e.g., constructors, *operator* =) and is implemented in the *ActiveReal* type. The overloaded method for the assign operator just calls *store* on the tape:

```
1  template<class R>
2  ActiveReal<Tape>& operator=(const Expression<R>& rhs){
3    globalTape.store(*this, rhs.cast());
4    return *this;
5  }
```

The design principle behind CoDiPack is that the *store* method calls the *calcGradient* method of the expression. This causes a recursive call of the *calcGradient* method until the *ActiveReal* types are reached. As discussed above, we did not want to add any logic to the *ActiveReal* implementation;

```
1  template<typename R>
2    class ForwardEvaluation : public TapeInterface<R, R>{
3    public:
4
5      typedef R GradientData;
6
7      template<typename Rhs>
8      inline void store(R& value, GradientData& lhsTangent, const Rhs& rhs) {
9        R gradient = R();
10       rhs.template calcGradient<R>(gradient);
11       lhsTangent  = gradient;
12       value = rhs.getValue();
13     }
14
15     template<typename Data>
16     inline void pushJacobi(Data& lhsTangent, const R& jacobi, const R& value,
           const GradientData& curTangent) {
17       ENABLE_CHECK(OptIgnoreInvalidJacobies, isfinite(jacobi)) {
18         lhsTangent += jacobi * curTangent;
19       }
20     }
21
22     ...
23   }
```

Fig. 7. Tape implementation for the AD forward mode.

therefore, the call of *calcGradient* is forwarded to the *pushJacobi* method on the tape. The call hierarchy is then:

Active Real		Tape		Expression		Active Real		Tape
operator =	→	store	→	calcGradient	→	calcGradient	→	pushJacobi

which shows that the tape calls itself through the *calcGradient* method. As a result, in the *pushJacobi* method, the tape implementation can gather information about the Jacobian of the expression.

The user interface is more involved and is required only for the reverse mode. The most important functions are *registerInput*, *registerOutput*, *evaluate*, *setActive*, and *setPassive*. They are needed in order to declare what the inputs and outputs of the area are that is taped. *evaluate* performs the evaluation of Equation (12) for all statements in the area that are marked with *setActive* and *setPassive*.

4.3 Implementation of the Tapes

The implementation of the (tapeless) forward AD mode described in Table 3 becomes now very simple. The full logic is shown in Figure 7, with a few abbreviations. All assignment operators will call the *store* method in line 8 from the tape interface. This method initializes a zero gradient and calls the *calcGradient* method on the expression in order to compute the gradient for the whole statement. The recursive nature of the *calcGradient* implementation will cause a call of *pushJacobi* for each argument of the statement, which contains the gradient of the statement with respect to this variable. According to the forward mode in Table 3, we only need to multiply this gradient value by the dot value, and then add it to the dot value of the left-hand side, which is implemented in line 18 of Figure 7. Because no external resources are used in the method, the compiler is able to optimize the code as aggressively as possible, which should yield optimal performance results.

The real challenge is the efficient implementation of the reverse AD mode. We identified three different data items at the beginning of the section that we need to store:

```
1
2  typedef int StatementIndex;
3
4  struct ArgumentsPos {
5    int chunkPos;
6    int dataPos;
7    StatementIndex nested;
8  };
9
10 struct NumArgumentsPos {
11   int chunkPos;
12   int dataPos;
13   ArgumentsPos nested;
14 };
15
16 struct ExternFuncPos {
17   int chunkPos;
18   int dataPos;
19   NumArgumentsPos nested;
20 };
```

Fig. 8. Example of the "generated" structure for the tape position in CoDiPack.

—number of arguments for each statement,
—the Jacobian entry for each argument,
—and the index for each argument.

These data entries are represented in three different data streams. The last two, the Jacobian entry and index for each argument, are written per argument. The first one, the number of arguments, is written once per statement. This yields two different running indices that need to be managed in the tape structure. In addition, we add two extra data items for user-defined functions and the position where the user-defined function needs to be evaluated. These two items yield a third running index for data management. Each data item is now written to a separate data stream where the streams with the same running index are handled together. The management of five different data streams with three different running indices is already significantly involved; we anticipate that other tape developments will require additional data streams with other running indices. Therefore, we implemented a generalized solution in *chunkVector.hpp* that can handle an arbitrary set of data streams with different running indices.

The implementation in *chunkVector.hpp* is a recursive data structure, which is called *chunk vector*. It has a template argument for a child vector and uses the child vector's position type to define its own position. Therefore, every time the position of the parent is queried, the position of the child is also returned. The example position for the three different running indices is shown in Figure 8, which also includes the terminating position from the index manager. If the position of the parent is set (e.g., *reset*), then the same method is called on the child with the child's position. An example of such a call structure is shown in Figure 9. This has the advantage that all data streams can be handled as one data object. All operations are applied only to the root vector and recursively evaluated on all child vectors. If a new data stream needs to be added, then only a new vector must be inserted in the recursive structure and no additional logic needs to be implemented. The data streams are now stored in chunks of data. Each chunk can have multiple entries, such as the *double* for the Jacobian and the corresponding index, which makes it possible to handle data streams with the same running index in one chunk vector. Because no array of structures is required by this implementation, the memory layout will be optimal for caching, and no dead memory is generated owing to padding bytes. This will yield optimal performance on HPC clusters that are optimized for continuous memory access.

```
\\ Class Tape:
  void reset(ExternFuncPos pos) {
    externalFunctionVector.reset(pos);
  }
\\ Class ChunkVector:
  void reset(Position pos) {
    performLocalReset(pos.chunkPos, pos.dataPos);

    nested.reset(pos.nested);
  }
```

Fig. 9. Implementation of the recursive reset in the chunk vectors.

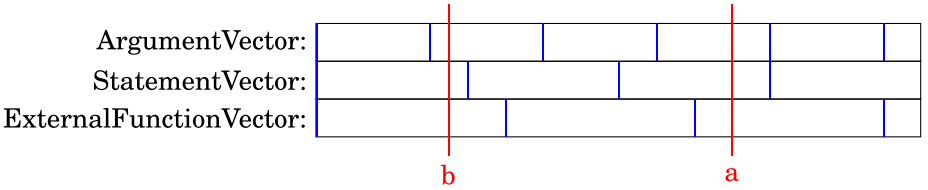


Fig. 10. Example for the data layout of the Jacobian tape implementation. The blue bars show the chunk boundaries and the marks *a* and *b* a possible region for the interpretation.

With these implementations, the five arrays can be easily defined. The user functions and the positions for them are always accessed at the same time; therefore, they have the same running index. The chunk is instantiated as *Chunk2<ExtFuncData, ChildPosition>*, where *ExtFuncData* is the data for the user-defined functions and the *ChildPosition* is the position type of the child vector. The Jacobian entries and indices for the arguments also have the same running index. Therefore, they can be represented by the chunk *Chunk2<double, int>*, which creates separate vectors for the Jacobian values and indices. The fifth array is the number of arguments for each statement. It can be represented by the chunk *Chunk1<uint8_t>*. With these three chunk definitions, the corresponding three chunk vectors *User Function Vector*, *Argument Vector*, and *Statement Vector* are created and linked together as

$$\text{User Function Vector} \rightarrow \text{Statement Vector} \rightarrow \text{Argument Vector} \rightarrow \text{linear index},$$

with the linear index as the terminator of the dependency chain. The linear index defines a counter only for the current statement. Therefore, it has no associated array. The definition in CoDiPack for the vectors is now:

```

    typedef Chunk2< double, int> JacobiChunk;
    typedef ChunkVector<JacobiChunk, IndexHandler> ArgumentVector;

    typedef Chunk1<uint8_t> StatementChunk;
    typedef ChunkVector<StatementChunk, ArgumentVector> StatementVector;

    typedef Chunk2<ExternalFunction,typename StatementVector::Position>
        ExternalFunctionChunk;
    typedef ChunkVector<ExternalFunctionChunk, StatementVector>
        ExternalFunctionVector;
```

The data dependency is now illustrated in Figure 10. It illustrates the data chunks, vector implementation, and recursive nature. Each chunk boundary is illustrated by a blue bar. The bottom lane contains the user function data and has the slowest running index, whereas the midlane contains the statement data and has a faster running index. Finally, the top lane contains the data for each

```
1  void evaluateInt(NumArgumentsPos start, NumArgumentsPos end) {
2    auto evalFunc = [this] (
3        // from linear index manager
4        const int& startAdjPos, const int& endAdjPos,
5        // from statement vector
6        int& stmtPos, const int& endStmtPos, uint8_t* &statements,
7        // from argument vector
8        int& dataPos, const int& endDataPos, double* &jacobies, int* &indices
9    ) {
10     size_t adjPos = startAdjPos;
11
12     while(adjPos > endAdjPos) {
13       const double adj = adjoints[adjPos];
14       adjointData[adjPos] = 0.0;
15       adjPos -= 1;
16       stmtPos -= 1;
17
18       for(uint8_t curVar = 0; curVar < statements[stmtPos]; curVar += 1) {
19         dataPos -= 1;
20         adjoints[indices[dataPos]] += adj * jacobies[dataPos];
21       }
22     }
23   };
24
25   statementVector.evaluateReverse(start, end, evalFunc);
26 }
```

Fig. 11. Implementation of the recursive reset in the chunk vectors.

argument that is written most often and has the fastest running index. The linear index handler contains no data; it is used only for position calculations.

With this setup of the data structure, it is now possible to define the interpretation loop for the reverse mode of AD. The example in Figure 11 provides the full implementation after the external function data has been handled. Lines 2 to 23 define the lambda expression for the evaluation that is then called in line 25. The *evaluateReverse* method loads the data for the current vector and calls the same method on the child vector with the positions for the child vector. The loaded data is appended to the list of arguments containing the current index to the data and the index up to which it is safe to access the data. As the last element in the sequence, the index manager calls the lambda expression with all of the loaded data and ranges. In lines 10 to 23, the reverse evaluation defined by Equation (12) is performed.

If the example from Figure 10 is now interpreted from position a to b, then 6 chunk boundaries are crossed. At these points, the interpretation is stopped, the next chunk is loaded, and then the interpretation is continued, which results in 6 calls of the lambda expression with 6 different ranges. Because of the chunk vector design, the chunk management is fully automatic.

There are now several implementations of the reverse AD mode. The most general one is the *RealReverse* type, which uses a configuration of the chunk vectors that allocates new memory on the fly. Because of the required bounds checking, this implementation requires at least two if statements that are evaluated when an operation is stored on the tape.

There is also the *RealReverseUnchecked* type, which is configured such that no bound checks are performed. This has the advantage that no if statements are required when an operation is stored on the tape and should yield faster code. The disadvantage of this type is that the user has to allocate the required memory in advance.

For both types, it holds that they are compatible with C-like memory operations (e.g. memcpy). The implementation of the types is done such that the primal computation type and the derivative type can be freely configured by template parameters. This enables the creation of higher-order

derivative types by nesting of the existing CoDiPack types. A second-order forward-over-reverse type can be created with *RealReverseGen<RealForward>*. The *Gen* suffix represents the type that can be freely configured. A vector mode is as easily created by using a vector type for the derivative type. An example of a vector mode with 4 directions would be *RealReverseGen<double, Direction<double, 4> >*. The *Direction* structure is a minimal implementation of such a vector type in CoDiPack. In general, the derivative type needs to implement only a scalar multiplication and an addition operation.

4.4 Type Configuration

The *RealReverse* and *RealReverseUnchecked* types can be configured via several different parameters. It is common to introduce the logic for such parameters by preprocessor macros. A good example is the check in CoDiPack for a zero Jacobian entry. If the Jacobian is zero, then its value should not be pushed on the tape. If this functionality needs to be disabled, the usual implementation would be:

```
#if CODI_ENABLE_JACOBI_ZERO_CHECK
  if(jacobi != 0.0) {
#endif
    tape.pushJacobi(jacobi);
#if CODI_ENABLE_JACOBI_ZERO_CHECK
  }
#endif
```

The preprocessor macros make the code very hard to understand and the indentation of the code between the two macros is not clear. The same effect can be achieved if a constant Boolean is added to the if statement. The new code is then:

```
static const bool CODI_EnableJacobiZeroCheck = true;
...
if(!CODI_EnableJacobiZeroCheck || jacobi != 0.0) {
  tape.pushJacobi(jacobi);
}
```

The variable *CODI_EnableJacobiZeroCheck* can force the *if* to be true. If *CODI_ EnableJacobiZeroCheck* is true, then the first condition of the if is false and the second condition is checked. If *CODI_EnableJacobiZeroCheck* is false, then the first condition of the if is always true and the second condition does not need to be checked. The compiler can optimize this code such that the if is completely eliminated or the constant true is never evaluated.

The readability of the code has improved but the if statement is more complex to understand. With the introduction of a macro, the readability can be further improved:

```
static const bool CODI_EnableJacobiZeroCheck = true;
#define ENABLE_CHECK(option, condition) if(!(option) || (condition))
...
ENABLE_CHECK(CODI_EnableJacobiZeroCheck, jacobi != 0.0) {
  tape.pushJacobi(jacobi);
}
```

It is now clear that the check is enabled only if the option is true. The advantage of the approach is that it reads like a normal *if* and the code structure is still intact. If several of these checks are required in a code block, then the macro approach with *#if* and *#endif* becomes unreadable in most cases.

A short overview of all parameters that can be configured in this way is given in Section 7 of this article.

Table 5. Overview of Available CoDiPack Types

CoDiPack Type	AD method	Memory per statement	Properties
RealForward	Tapeless AD Forward Mode	-	C compatible
RealReverse	AD Reverse mode, Jacobian Taping, linear indexing	$1 + 12 \times k$	C compatible
RealReverseIndex	AD Reverse mode, Jacobian Taping, index reuse	$5 + 12 \times k$	-
RealReversePrimal	AD Reverse mode, Primal Value Taping, linear indexing	$19 + 4 \times k$	C compatible
RealReversePrimalIndex	AD Reverse mode, Primal Value Taping, index reuse	$23 + 4 \times k$	-

k describes the number of arguments in one statement. *Unchecked* variants of the types have the same properties.

4.5 Index Reuse

At the start of Section 4, the linear indexing model was introduced. This model allows the types of the AD tool to be compatible with C-like memory operations, but it produces large adjoint vectors where most of the memory is used just once. Several things have to be considered for an implementation that reuses indices.

The *ActiveReal* types now require a destructor in order to release the index. Therefore, an index manager is implemented that stores the freed indices in a list and tracks the maximum amount of live indices. A best practice for index handling has yet to be determined (e.g., use of a sorted list, store ranges of free indices). The current implementation simply uses a stack for index management in order to be as fast as possible. If, for example, a sorting logic is used in index handling, then it impacts the evaluation of each statement, which has a huge performance impact on the runtime.

Furthermore, because the indices are reused, every *ActiveReal* needs a distinct index. If two *ActiveReals* would have the same index, it would be released twice and can then be given to two different variables that would use the same memory location in the reverse evaluation. With this in mind, the *ActiveReals* with an index reuse technique can no longer support C-like memory operations.

The advantage of index reuse is the reduced size of the adjoint vector. Instead of having the size of the total amount of variables, it shrinks to the size of the maximum amount of concurrent variables. Because of the reduced size, the reverse evaluation is usually faster.

The CoDiPack types with an index reuse have an *Index* suffix in their type names. This results in two new types: *RealReverseIndexUnchecked* and *RealReverseIndex*.

5 IMPLEMENTATION SUMMARY

The implementation of CoDiPack has evolved over the past several years and multiple features have been added. As this article is an overview of the basic implementation principles, not all features will be discussed in detail. Table 5 lists all basic types that are currently available in CoDiPack. The *RealForward*, *RealReverse* and *RealReverseIndex* types are discussed in this article. For a discussion of how the *RealReversePrimal* types are implemented, the reader is referred to Sagebaum et al. [2018]. Owing to the code layout described in this article, it is simple to add new tape implementations to CoDiPack. The primal value tapes are one example where several general classes in CoDiPack could be reused for the implementation. As CoDiPack is a header-only library, no special system-dependent code is necessary. The only requirement of CoDiPack is a C++11 compliant compiler.

All types are implemented as template types such that the computation type (e.g., double) and the derivative type can be freely chosen by the user. This enables higher-order derivatives via the nesting of multiple CoDiPack types or vector mode evaluations by choosing a vector type for the derivative type. A full list of the major features is shown in Table 6. How these features are used

Table 6. Overview of Available Helpers and Features for Reverse CoDiPack Types

Feature	Tutorial	Description
Higher-order derivatives	7, 7.1, 7.2	Nested CoDiPack types can be used to compute higher-order derivatives, e.g. *RealReverseGen<RealForward>>*.
Higher-order helper	7, 7.1, 7.2	Interface for an easy access to the higher-order derivatives.
Vector mode	6	Managed via the definition of a custom evaluation direction, e.g., *RealReverseVec<4>*.
Vector mode helper	A4	Decoupling of the tape evaluation from the tape recording.
OpenMP support for reverse evaluation	A4.1	-
External function interface	A1	Add user-defined function to the tape evaluation process.
MPI support	-	Managed via external functions and the MeDiPack [Sagebaum 2017] library.
External function helper	A1	User-friendly way to add user-defined function to the tape evaluation process.
Preaccumulation helper	A2	Improved storing for medium-sized functions.
Manual storing of statements	A3	Add derivatives for functions with a known derivative.

can be found in the documentation of CoDiPack. The tutorial column describes which features are described in the corresponding CoDiPack tutorial.

It should be noted that the MPI support for CoDiPack is provided via the MeDiPack library [Sagebaum 2017]. MeDiPack was developed, like CoDiPack, by the Scientific Computing group at TU Kaiserslautern. Currently, 80% of the MPI standards are supported, including asynchronous communication and user-defined types. MeDiPack provides generalized interfaces so that other AD tools can interface with MeDiPack, one example being ADOL-C.

6 TESTS

The two test cases from Sagebaum et al. [2018] are also considered here. The first is a simple unit square solution of the coupled Burgers equation. The second test case is a solution of the LM1021 supersonic aircraft with SU2. Both test cases compute the derivatives with CoDiPack.

6.1 Coupled Burgers Equation

The coupled Burgers equation [Bahadır 2003; Biazar and Aminikhah 2009; Zhu et al. 2010] is chosen as a lightweight test that can be used to do a rapid evaluation of how some changes in CoDiPack affect performance.

The equations

$$u_t + uu_x + vu_y = \frac{1}{R}(u_{xx} + u_{yy}) \tag{13}$$

$$v_t + uv_x + vv_y = \frac{1}{R}(v_{xx} + v_{yy}) \tag{14}$$

are discretized with an upwind finite difference scheme. The initial conditions are:

$$u(x, y, 0) = x + y \quad (x, y) \in D \tag{15}$$

$$v(x, y, 0) = x - y \quad (x, y) \in D \tag{16}$$

and the exact solution is [Biazar and Aminikhah 2009]

$$u(x, y, t) = \frac{x + y - 2xt}{1 - 2t^2} \quad (x, y, t) \in D \times \mathbb{R} \tag{17}$$

$$v(x, y, t) = \frac{x + y - 2xt}{1 - 2t^2} \quad (x, y, t) \in D \times \mathbb{R}. \tag{18}$$

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. As far as the differentiation is concerned, we choose as the input parameters the initial solution of the time-stepping scheme. As the output parameter, we take the norm of the final solution.

For the implementation of the program, all methods are written in such a way that they can be inlined and the requested memory is allocated and initialized before the time measurement starts. The required memory for the tape is computed beforehand and then allocated. This yields very stable time measurements, which are run on one node of the Elwetritsch cluster at the TU Kaiserslautern, which consists of two Intel E5-2640v3 CPUs with a total of 16 cores and 256 GB of memory. The general setup calculates the Burgers equations on a 601×601 grid with 32 iterations and the time measurement is repeated 10 times. For the time measurements, two different configurations are tested:

- The *multi* test configuration runs the same process on each of the 16 cores. This setup simulates a use case where the full node is used for computation and every core uses the memory bandwidth of the socket.
- The *single* test configuration runs just one process on the whole node. This eliminates the memory bandwidth limitations and provides a better view on the computational performance.

Both test configurations will be evaluated with the default CoDiPack settings and then with special configurations options in order to see how these options effect the performance of CoDiPack.

6.2 SU2

An important application where derivatives are currently frequently needed is numerical optimization. When constraints are defined using partial differential equations (PDEs), this usually requires efficient adjoint methods. In that case, AD facilitates the development of those 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* [Economon et al. 2015]. The latter is a collection of tools for the analysis and optimization of internal and external aerodynamic problems using a finite-volume method. The differentiated code was used to generate a flexible and robust discrete adjoint solver for the Reynolds-averaged Navier-Stokes equations [Albring et al. 2015], optionally coupled with the Ffowcs-Williams-Hawkings equation [Zhou et al. 2016] for aeroacoustic optimizations. The adjoint solver is based on the fixed-point formulation of the underlying solver for the discretized state equation [Christianson 1994]. That is, we assume that feasible solutions U^* of the state equation

$$U = G(U, X) \tag{19}$$

are computed by the iteration $U^{n+1} = G(U^n, X)$ for $n \to N^*$. X represents the design variables and $G(U)$ some (pseudo) time-stepping scheme, such as the explicit or implicit Euler method. By applying the first-order necessary conditions on the optimization problem to minimize some scalar

objective function $J(U, X)$ with the constraint that the state Equation (19) is fulfilled, we end up with the following fixed-point equation for the adjoint state $\bar{U}$:

$$\bar{U} = \left[\frac{\partial J(U^*, X)}{\partial U}\right]^T + \left[\frac{\partial G(U^*, X)}{\partial U}\right]^T \bar{U}. \tag{20}$$

This equation can be solved using the iterative scheme

$$\bar{U}^{n+1} = \left[\frac{\partial J(U^*, X)}{\partial U}\right]^T + \left[\frac{\partial G(U^*, X)}{\partial U}\right]^T \bar{U}^n, \quad \text{for} \quad n \to \bar{N}^*. \tag{21}$$

This scheme can be easily constructed by applying AD to the code that computes J and G. It is important to note that since we need the right-hand side of Equation (21) repeatedly at the fixed-point U^*, we need to record the tape only once at the last fixed-point iteration. Hence, the time required for the interpretation determines the overall runtime. In contrast to other implementations of adjoint solvers, where AD is applied to only a certain part of the code, we replaced all occurrences of the default computation type with the AD-type provided by CoDiPack. Although this leads to a small overhead in runtime (one if statement per expression), the maintainability and on-the-fly differentiation of new features typically outweigh this disadvantage.

As the test case, we consider the inviscid flow over the LM1021 supersonic aircraft. The computational mesh consists of 5,730,841 interior elements and the aircraft is discretized using 214,586 boundary elements. For the spatial integration, we use a central scheme with artificial dissipation in combination with an explicit Euler method for the pseudo-time stepping.

7 RESULTS

The results of both test configurations explained in Section 6 are now discussed.

7.1 Coupled Burgers Equation

The first analysis discusses the general timings of the implementation and the comparison between the Intel and GCC compiler. The recording times for the coupled Burgers equation are always averaged over all processors and runs and are shown in Figure 12 for the GCC compiler and Figure 13 for the Intel compiler.

The results are shown for the single and multi configuration. For each compiler and configuration, the four different available tapes are checked and compared with Adept and ADOL-C [Walther and Griewank 2009]. The tested CoDiPack types are *RealReverseUnchecked*, *RealReverseIndexUnchecked*, *RealReverse* and *RealReverseIndex*. The forward mode of CoDiPack is also compared with the Sacado forward AD type that uses ETs [Phipps and Pawlowski 2012].

As we can see, there are some discrepancies between the single and the multi variant of the Burgers test case. The single variant runs approximately 60% faster than the multi variant because of the memory bandwidth limitation of the computing node. In the single test case, only one process uses the full bandwidth, while in the multi test case, 16 processes share the memory bandwidth. The performance values for the Intel compiler are nearly the same as for the GCC compiler. The single variant is slightly faster than the GCC version.

It can also be seen that, for the single test case, the required recording time increases with the complexity of the tape implementation. The *RealReverseUnchecked* tape is the simplest one and only a small increase is seen for the *RealReverseIndexUnchecked* version that also uses an index handler. The jump from the *RealReverseUnchecked* type to the *RealReverse* type, which allocates memory on the fly and performs bound checking, is around 9% computation time. Interestingly, the index reuse is faster than the linear indexing in that case, probably due to caching effects. In

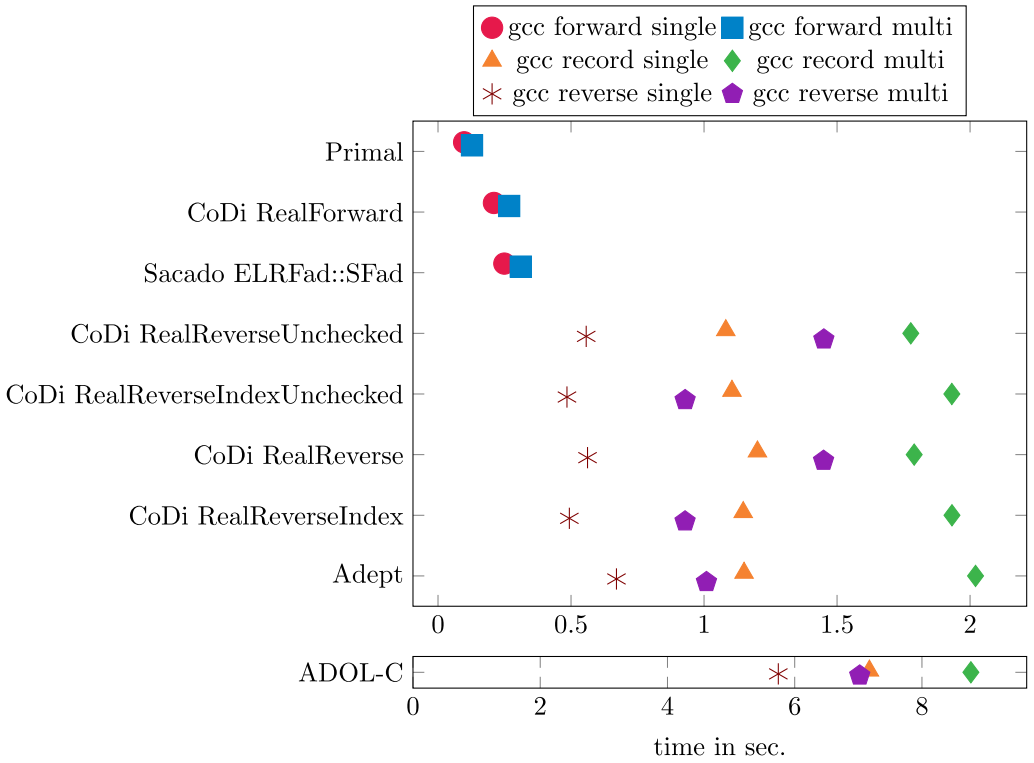


Fig. 12. Evaluation times for CoDiPack and other selected tools for the coupled Burgers equation test case for the GCC compiler.

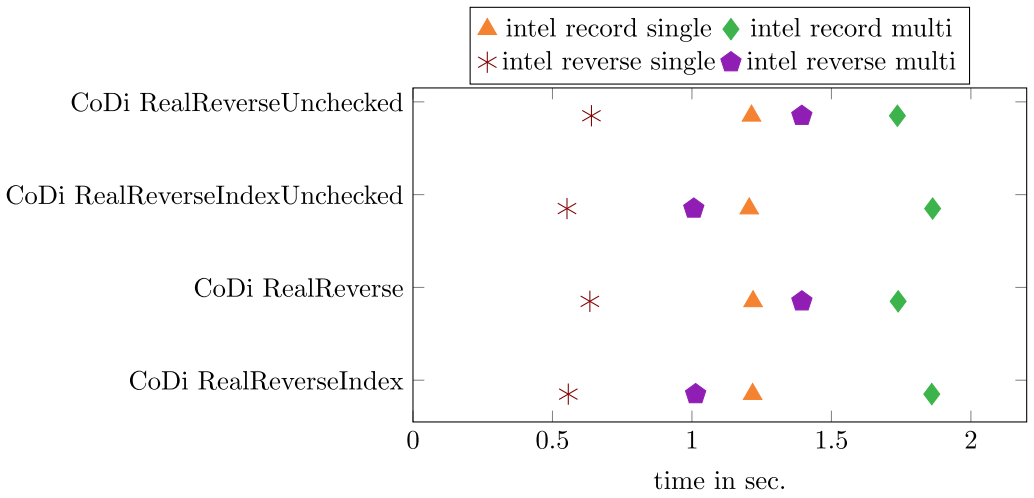


Fig. 13. Evaluation times for CoDiPack for the coupled Burgers equation test case for the Intel compiler.

the multi test case, all of these effects are no longer visible. The *RealReverseIndexUnchecked* and *RealReverseIndex* types require more taping time because they need to store additional data.

The interpretation time for the Burgers equation is also shown in Figure 12. Here, the difference between the *RealReverseUnchecked* and *RealReverseIndexUnchecked* types and the corresponding regular types *RealReverse* and *RealReverseIndex* is relatively small, because the code for both tape types is nearly the same. Only the index reuse types show a different timing. They require less memory in the reverse interpretation; therefore, they are faster in the interpretation. The Intel compiler is again slightly slower in the single test case, which is no longer seen in the multi case with the memory bandwidth limitation. Figure 12 also shows that the reduced memory of the index tapes has a large impact on the reverse interpretation time. Therefore, we can reach the conclusion that anything that saves memory can also improve the required time for the AD process.

The comparison of CoDiPack with Adept and ADOL-C shows that the performance with respect to Adept is quite similar. The implementation of Adept can be compared to the *RealReverseIndex* type from CoDiPack. For both configurations, CoDiPack has a slightly increased performance for the recording and reversal of the tape. CoDiPack requires 7% (250 MB) less memory than Adept, which comes from the assumption that an expression can have only 256 arguments.

The comparison with ADOL-C is not very meaningful, since ADOL-C implements a primal value taping method that only uses operator overloading and no expression templates. Therefore, the times for ADOL-C are roughly 5 times larger than the times for CoDiPack.

The comparison of the forward mode AD tools shows that the CoDiPack implementation is slightly better than the Sacado implementation. In terms of time factors, the CoDiPack implementation has a factor of 2.15 in relation to the primal evaluation. Sacado has a factor of 2.54, which makes CoDiPack for the single configuration 15% more efficient. In the multi configuration, the factors are nearly the same, which yields the same improvement of 15%.

As was already shown in Hogan [2014], the performance of AD not only depends on the structure of the code and its memory footprint but also on the number of transcendental functions. Usually, the compiler can aggressively optimize code that contains only nontranscendental functions, but the application of AD interferes in general with these optimizations. Hence, the more transcendental functions occur in the code, the better the relative performance of recording and interpretation. Since the algorithm for the Burgers equation represents an extreme case with no transcendental functions at all, we can expect relatively high numbers when comparing the runtime of the derivative computation with the runtime of the primal code (in fact, we get factors 17 and 26 for the single and multi configurations, respectively).

The next analysis considers the default block size for the chunk tapes. The data in Table 7 shows that the recording of the tape is not influenced by the size of the blocks. This is different when the evaluation of the tape is considered. Here, both test configurations show an overhead for very small blocks.

A small increase in time is also seen for blocks with more than 33 million entries.

In order to have an optimal block size for all cases, the default is set in CoDiPack to 2 million entries.

The implementation of CoDiPack contains several switches that can be used to fine-tune the tapes to the needs of the user. The data in Table 8 shows the switches separated into the ones that influence the recording of the tape and the ones that influence the interpretation of the tape:

- With the *Check expression arguments* switch enabled, CoDiPack performs checks if the arguments for the elemental functions are in the differentiable domain.
- *Ignore invalid Jacobian values* and *Ignore zero Jacobian values* prevent zero and invalid Jacobian values from being recorded on the tape.

Table 7. Block Size Time Comparison in Seconds for the Coupled Burgers Equation

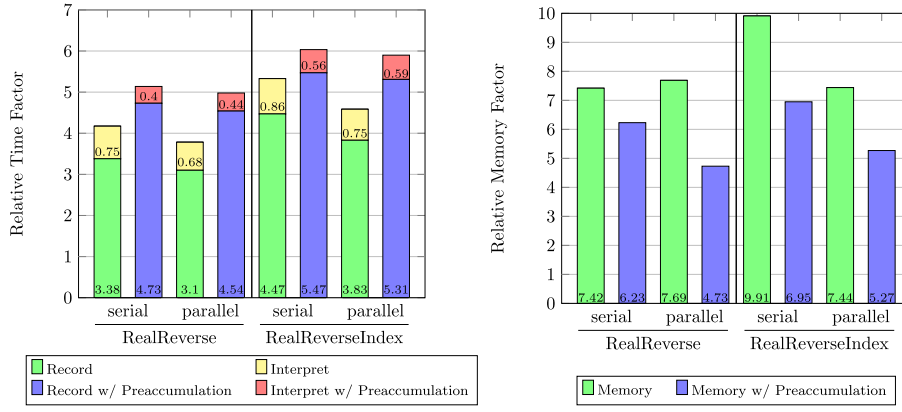
Block entries	Record multi	Record single	Interpret multi	Interpret single
1,024	2.15	1.30	1.89	0.98
2,048	2.15	1.29	1.81	0.85
4,096	2.16	1.28	1.77	0.79
8,192	2.17	1.27	1.75	0.74
16,384	2.16	1.27	1.75	0.73
32,768	2.18	1.28	1.74	0.71
131,072	2.18	1.28	1.74	0.71
262,144	2.18	1.29	1.74	0.72
524,288	2.16	1.28	1.75	0.70
1,048,576	2.16	1.29	1.75	0.72
2,097,152	2.16	1.28	1.75	0.70
4,194,304	2.16	1.28	1.75	0.70
8,388,608	2.16	1.28	1.75	0.70
16,777,216	2.16	1.28	1.75	0.70
33,554,432	2.16	1.28	1.75	0.71
67,108,864	2.16	1.28	1.75	0.72
134,217,728	2.17	1.28	1.75	0.73

Table 8. Configuration Switches Time Comparison in Seconds for the Coupled Burgers Equation

Record switch	Record multi	Record single	Interpret multi	Interpret single
All off	2.13	1.12	1.75	0.67
Check expression arguments	2.13	1.12	1.76	0.67
Ignore invalid Jacobian	2.15	1.15	1.74	0.67
Ignore zero Jacobian	2.14	1.15	1.76	0.71
Check tape activity	2.14	1.16	1.76	0.71
All on	2.15	1.23	1.76	0.75
Interpretation switch	Record multi	Record single	Interpret multi	Interpret single
All off	2.13	1.12	1.75	0.67
Skip zero adjoints	2.13	1.13	1.76	0.71
All on	2.15	1.23	1.76	0.75

— *Check tape activity* enables the test if the tape is currently active and, therefore, should record statements.
— The reverse switch *Skip zero adjoints* prevents CoDiPack from performing the update $\bar{a}$ += $c * 0.0$ in the reverse interpretation.

Because the test case is highly optimized, only the ignoring of zero Jacobian values changes the tape, but to a margin that can be ignored. The memory bandwidth limited case shows only a small increase in recording and evaluation time when all switches are enabled because most of the logic can be hidden in the memory latency. Therefore, for large cases, all checks can be enabled and the additional time will be minimal. The single case shows an increase of 10% in the computation time for the recording and evaluation of the tape. This shows that if CoDiPack is used on a very small portion of a code, where the tape size is very small, it can improve the performance when specific checks are disabled.



(a) Relative time values.

(b) Relative memory values

Fig. 14. Measurements of the SU2 CFD suite for the LM1021 aircraft. All values are give as relative measurements with respect to the unmodified primal code.

7.2 SU2

Figure 14 shows the relative time and memory factors for 100 iterations of Equation (21) with respect to one evaluation of Equation (19). Here, we also distinguish between the serial run and parallel run using a full compute node with 16 cores and the *RealReverse* and *RealReverseIndex* types. Owing to its structure, CoDiPack allows direct access to the Jacobian values stored on the tape. This enables the use of advanced AD techniques such as the preaccumulation of Jacobian values [Utke 2005], where parts of the tape are already interpreted during the recording. In SU2, this method is employed in certain parts of straight-line code to significantly reduce memory and interpretation time at the cost of increased recording time. We include this approach in the discussion since it represents the common setting when using the adjoint solver.

First, there is no performance degradation noticeable for the parallel computations compared with the serial computations. This indicates that the latter is already bandwidth limited. In fact, the parallel case consistently shows even a slightly lower factor. Although the majority of operations in SU2 are multiplications and additions, in some parts of the code transcendental functions are also used, which intervene with compiler optimizations. Indeed, unlike the test case presented in the previous section, we get relative runtime factors between 3.1 to 4.47 for the recording and between 0.4 to 0.86 for the interpretation of one flow iteration without preaccumulation. Again, we emphasize that this represents a "black-box" differentiation that contains *all* routines to compute the flow update in Equation (19), even if they do not contribute to the final gradient and, therefore, can be considered as passive. Hence, little knowledge of the code is required in that case. The application of preaccumulation requires slightly more knowledge of the code to identify inputs/outputs of certain regions. Still, once identified, the modifications to apply the method are minor. Figure 14 also shows the effect of preaccumulation on runtime and memory. Although the time for taping increases by roughly 40%, the interpretation time reduces by almost 50%, which, in turn, reduces the overall runtime of the adjoint solver by nearly 50%. In addition, memory consumption is reduced by a large amount.

In contrast to the Burgers test case in Section 7.1, here, the nonlinear indexing scheme (represented by the *RealReverseIndex* type) does not offer a better performance compared with the *RealReverse* type with linear indexing. A detailed analysis showed that this is due to a high amount of "active copies" in SU2, i.e., copy/assignment operations with just one active type on the right-hand

side. For linear indexing, a simple copy including the index value is sufficient. For the nonlinear indexing type, a more complex treatment is necessary since each index needs to be unique in the application. Therefore, a statement has to be written for each copy operation.

8 CONCLUSION AND OUTLOOK

In this article, we demonstrated the efficient implementation of a novel AD tool that employs Jacobian taping with expression templates. Unlike other tools, the layout of CoDiPack is designed so that other taping approaches can be easily added to further enhance its capabilities while still maintaining an easy-to-use interface and high performance. Furthermore, the recursive layout of the data streams allow for a simple addition of new streams and several design choices, such as the avoidance of preprocessor macros, that make the code easy to understand. CoDiPack provides a useful basis that hopefully encourages people to join the research in implementation strategies for AD.

The performance of the ET approach heavily relies on the capability of the compiler to inline certain routines. Hence, to ensure that CoDiPack offers compiler-independent behavior, we first compared the performance between the Intel and GCC compilers. Here, it was shown that both compilers give similar performances, though the GCC compiler was slightly faster when just a single process is used. This difference diminishes, however, when occupying all available cores of a node because memory bandwidth becomes the limiting factor. Similar results were reported for the indexing and memory management schemes for both compilers. Using the same compiler, it was shown that there is only a moderate performance degradation during recording when the more complex chunk memory management implementation is used compared with simple management without any bound checking. The former offers a more user-friendly behavior without the need to know the tape size *a priori*. Furthermore, the types that use an index handler need less runtime for the interpretation owing to the reduced tape size.

CoDiPack yields a slightly increased performance with respect to Adept and Sacado. Since CoDiPack is inspired by Adept and ETs in general are the most efficient solution for an AD tool implementation, no significant performance gains could be expected.

The application to the CFD software SU2 certified CoDiPack's high performance even for the case of a "black-box" differentiation of complex program code and in parallel. High-level tuning using preaccumulation techniques can help to further decrease memory consumption.

Future research will be devoted to the improvement of performance using novel indexing and memory management schemes. We will also expand the research to the field of analysis of codes to provide guidelines on which CoDiPack type is optimal for what kind of code structure. Additionally, the user interface in CoDiPack is constantly extended in order to provide convenient routines to easily handle user-differentiated functions or preaccumulation.

ACKNOWLEDGMENTS

The authors would like to thank all referees for their thorough review and helpful comments.

REFERENCES

T. Albring, M. Sagebaum, and N. R. Gauger. 2015. Development of a consistent discrete adjoint solver in an evolving aerodynamic design framework. In *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. American Institute of Aeronautics and Astronautics, 1–14. DOI: https://doi.org/doi:10.2514/6.2015-3240

P. Aubert, N. Di Césaré, and O. Pironneau. 2001. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science* 3, 4, 197–208. DOI: https://doi.org/10.1007/s007910000048

A. Bahadır. 2003. A fully implicit finite-difference scheme for two-dimensional Burgers' equations. *Applied Mathematics and Computation* 137, 1, 131–137.

S. Beal. 2004. Supermacros: Powerful, Maintainable Preprocessor Macros in C++. Retrieved December 21, 2016 from http://wanderinghorse.net/computing/papers/supermacros_cpp.pdf.

B. M. Bell and J. V. Burke. 2008. Algorithmic differentiation of implicit functions and optimal values. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke (Eds.). Springer, 67–77. DOI: https://doi.org/10.1007/978-3-540-68942-3_7

J. Biazar and H. Aminikhah. 2009. Exact and numerical solutions for non-linear Burgers' equation by VIM. *Mathematical and Computer Modelling* 49, 7, 1394–1400.

B. Christianson. 1994. Reverse accumulation and attractive fixed points. *Optimization Methods and Software* (1994), 311–326. DOI: https://doi.org/10.1080/10556789408805572

T. D. Economon. 2018. Simulation and adjoint-based design for variable density incompressible flows with heat transfer. In *Multidisciplinary Analysis and Optimization Conference*. AIAA 2018-3111.

T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso. 2015. SU2: An open-source suite for multiphysics simulation and design. *AIAA Journal* 54, 3, 828–846.

A. Griewank and A. Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (2nd ed.). SIAM, Philadelphia, PA.

R. Hogan. 2014. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software* 40, 4, 26.

A. Hück, S. Kreutzer, D. Messig, A. Scholtissek, C. Bischof, and C. Hasse. 2018. Application of algorithmic differentiation for exact Jacobians to the universal laminar flame solver. In *International Conference on Computational Science*. Springer, 480–486.

A. Hück, C. Bischof, M. Sagebaum, N. R. Gauger, B. Jurgelucks, E. Larour, and G. Perez. 2017. A usability case study of algorithmic differentiation tools on the ISSM ice sheet model. *Optimization Methods and Software* 33, 4–6 (2018), 844–867. http://dx.doi.org/10.1080/10556788.2017.1396602

K. Leppkes, J. Lotz, and U. Naumann. 2016. *Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features*. Technical Report AIB-2016-08. RWTH Aachen University. Retrieved October 22, 2019 from http://aib.informatik.rwth-aachen.de/2016/2016-08.pdf.

L. Lubkoll. 2017. FunG — Invariant-based modeling. *Archive of Numerical Software* 5, 1, 169–191. DOI: https://doi.org/10.11588/ans.2017.1.27477

M. Luers, M. Sagebaum, S. Mann, J. Backhaus, D. Grossmann, and N. R. Gauger. 2018. Adjoint-based volumetric shape optimization of turbine blades. In *2018 Multidisciplinary Analysis and Optimization Conference*. AIAA 2018-3638.

S. A. Nørgaard, M. Sagebaum, N. R. Gauger, and B. S. Lazarov. 2017. Applications of automatic differentiation in topology optimization. *Structural and Multidisciplinary Optimization* 56, 5 (2017), 1135–1146. http://dx.doi.org/10.1007/s00158-017-1708-2

E. Phipps and R. Pawlowski. 2012. Efficient expression templates for operator overloading-based automatic differentiation. In *Recent Advances in Algorithmic Differentiation*, Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther (Eds.). Lecture Notes in Computational Science and Engineering, Vol. 87. Springer, Berlin, 309–319. DOI: https://doi.org/10.1007/978-3-642-30023-3_28

E. T. Phipps, R. A. Bartlett, D. M. Gay, and R. J. Hoekstra. 2008. Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke (Eds.). Springer, 351–362. DOI: https://doi.org/10.1007/978-3-540-68942-3_31

M. Sagebaum. 2017. MeDiPack: Message Differentiation Package. Retrieved October 22, 2019 from https://www.scicomp.uni-kl.de/software/medi.

M. Sagebaum, T. Albring, and N. R. Gauger. 2018. Expression templates for primal value taping in the reverse mode of algorithmic differentiation. *Optimization Methods and Software* 33, 4–6, 1207–1231. https://doi.org/10.1080/10556788.2018.1471140

R. Sanchez, T. Albring, R. Palacios, N. R. Gauger, T. D. Economon, and J. J. Alonso. 2018. Coupled adjoint-based sensitivities in large-displacement fluid-structure interaction using algorithmic differentiation. *Internatational Journal for Numerical Methods in Engineering* 113, 7, 1081–1107. http://onlinelibrary.wiley.com/doi/10.1002/nme.5700/full.

M. Schwalbach, L. Müller, T. Verstraete, and N. R. Gauger. 2018. CAD-based adjoint multidisciplinary optimization of a radial turbine under structural constraints. *Conference Proceedings of GPPS 2018*. GPPS, 1–8.

J. Utke. 2005. Flattening basic blocks. In *Automatic Differentiation: Applications, Theory, and Implementations*, H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris (Eds.). Lecture Notes in Computational Science and Engineering, Vol. 50. Springer, 121–133. DOI: https://doi.org/10.1007/3-540-28438-9_11

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

A. Walther and A. Griewank. 2009. Getting started with ADOL-C. In *Combinatorial Scientific Computing*. Chapman-Hall CRC Computational Science, 181–202.

B. Y. Zhou, T. Albring, N. R. Gauger, C. R. Ilario da Silva, T. D. Economon, and J. J. Alonso. 2016. An efficient unsteady aerodynamic and aeroacoustic design framework using discrete adjoint. *AIAA*, 1–18.

H. Zhu, H. Shu, and M. Ding. 2010. Numerical solutions of two-dimensional Burgers' equations by discrete Adomian decomposition method. *Computers & Mathematics with Applications* 60, 3, 840–848.

Received October 2017; revised July 2019; accepted July 2019