Received: 8 October 2022 | Accepted: 3 November 2022

DOI: 10.1002/pamm.202200208

Aggregated type handling in CoDiPack

Max Sagebaum[1,*] and **Nicolas R. Gauger**[1]

[1] Chair for Scientific Computing, TU Kaiserslautern, Bldg/Geb 34, Paul-Ehrlich-Strasse, 67663 Kaiserslautern, Germany

The development of algorithmic differentiation (AD) tools focuses mostly on handling floating point types in the target language. Taping optimizations in these tools mostly focus on specific operations like matrix vector products. Aggregated types like std::complex are usually handled by specifying the AD type as a template argument. This approach provides exact results, but prevents the use of expression templates. If AD tools are extended and specialized such that aggregated types can be added to the expression framework, then this will result in reduced memory utilization and improve the timing for applications where aggregated types such as complex number or matrix vector operations are used. Such an integration requires a reformulation of the stored data per expression and a rework of the tape evaluation process. We will demonstrate the overheads on a synthetic benchmark and show the improvement when aggregated types are handled properly by the expression framework of the AD tool.

1 Introduction

The application of algorithmic differentiation (AD) to a computer program enables this program to evaluate the derivatives alongside the regular (primal) computation. With the operator overloading approach the computation type in the program is exchanged with the so called *active type* of the AD tool. In the reverse mode (back propagation) approach, data like internal identifiers and Jacobian entries are stored during the regular computation for each operation (e.g. *,+, sin, cos) on a tape (stack). Afterwards the data on the tape is interpreted in the reverse order for the computation of the derivatives.

The set of operations for the active type usually covers all the basic operations used in a computer program. Algorithms that are formed with these operations are recorded on the tape and the derivatives can be computed. The problem with this approach is that it is not optimal for all algorithms. In [1] Naumann et al. have demonstrated this recently for the Matrix-Matrix product multiplication and the solution of linear systems in Eigen. They extended the set of operations for the active type in dco/c++ [2] such that the Matrix-Matrix product and solution of a linear system are now known by dco/c++. For both operations this resulted in a reduced amount of memory on the tape and an improved runtime. In the case of the linear system solver the runtime improves from a complexity of n^3 to n^2. In the same way, the AD tool Adept [3] has been specialized for array and vector operations such that these are stored on the tape in an optimized way. The Stan math library [4] provides optimized routines for taping linear algebra operations, as well.

The vector and matrix operations handled in the above papers can be generalized to aggregated types which, by definition, are types that consist of floating point entries. Next to matrices and vectors, other examples for aggregated types are complex numbers, SIMD types, layers in neural networks etc.. The problem with the above approaches is that they usually do not integrate well with the expression template technique [3, 5] that modern AD tools use. Expression templates extend the set of operations defined for the active type by adding all combinations of operations. As an example the equation

$$w = \sqrt{u^2 + v^2} \tag{1}$$

is no longer recorded as four separate operations as shown in Listing 1. With expression templates, the Equation (1) is recorded as one operation with two arguments by the AD tool.

If, for example, complex numbers were used in Equation (1), then it would again break down into the four intermediate operations. The tape storage for these four operations with complex numbers would be 232 byte. If the statement in Equation (1) did not separate into intermediate operations and can be captured by the AD tool as one large expression then the tape storage would be 106 byte which is a reduction by about 50 % for the complex number case.

In general a reduction in memory by about 40 % can be expected if complex types are added to the expression framework of an AD tool. For other aggregated types the same or even bigger gains can be expected. Therefore, we want to analyze how aggregated types can be added to the existing expression template operator overloading taping strategies, namely the Jacobian taping approach [5] and the primal value taping approach [6]. The presented work is structured as follows. Section 2 will give a short introduction to AD, expression templates and the extension to aggregated types. Afterwards, Section 3 will discuss how the extended expressions can be stored on the common taping strategies for AD, which is followed in Section 4 by a discussion of the implementation. Finally, Section 5 will present performance measurements for a generic test case. All code in this paper is c++ code.

* Corresponding author: e-mail max.sagebaum@scicomp.uni-kl.de, phone +49 631 205 5638

```
t1 = pow(u, 2);
t2 = pow(v, 2);
t3 = t1 + t2;
w = sqrt(t3);
```

Listing 1: Seperation of the statement $w = \sqrt{u^2 + v^2}$ into intermediate operations.

2 AD theory and expression templates

In this section we give a short introduction to AD. For a more complete overview, see [7, 8].

For applying AD, we assume that a program can be described as the function `void func( double const x[], double y[])` where `x` are the input values and `y` the output values. If `func` has a mathematical representation then it is defined as $F : \mathbb{R}^n \to \mathbb{R}^m$ with $y = F(x)$ but this is not necessary for applying AD. For AD it is assumed that `func` and therefore F can be broken down in a concatenation of elemental functions

$$\phi_i : \mathbb{R}^{d_i} \to \mathbb{R} \quad \text{with } w_i = \phi(v_i) \text{ and } i = 1 \dots N \, . \tag{2}$$

which can be operations like $*$, $+$, sin or exp. N can be in the order of 10^{12} for industrial applications. The reverse mode of AD applies the chain rule and the directional derivative on the chain of elemental functions ϕ_i and subsequently the discrete adjoint of the formulation is built. The reverse mode of AD is then defined by evaluating

$$\bar{v}_i \mathrel{+}= \frac{d\phi_i}{dv_i}^T (v_i)\bar{w}_i \quad \forall i = N \dots 1 \tag{3}$$

for all elemental functions. Please note, that Equation (3) is evaluated in reverse order from N to 1. $\bar{w}$ is the standard notation for adjoint variables in AD and is spoken as *bar w*. It is not the conjugate complex operators for complex numbers which we denote by $\circ^H$ in this paper. By evaluating all derivatives of ϕ_i in a reverse order, the reverse mode of AD computes the derivative for F such that

$$\bar{x} = \frac{dF}{dx}^T (x)\bar{y} \tag{4}$$

holds where $\bar{y}$ is the seeding for the reverse propagation of the derivatives and $\bar{x}$ is the resulting derivative. Thus $\frac{dF}{dx}^T$ is never setup explicitly. The reverse mode of AD is applied by first evaluating all ϕ_i and storing necessary information on a tape (stack) and afterwards the information is read in reverse order to evaluate Equation (3).

The regular theory for AD assumes that each elemental function has only one output argument. Since aggregated types are composed of multiple floating point values, operations on these objects are not covered by this definition. As done by Griewank and Walter in [7] the theory of AD can be extended such that elemental functions with an arbitrary number of output values are allowed. The definition of the elemental functions is then

$$\phi_i : \mathbb{R}^{d_i} \to \mathbb{R}^{p_i} \, . \tag{5}$$

AD can be applied to a code through source transformation [9] or operator overloading [2, 5]. In this paper we are focusing on operator overloading AD, where the AD tool provides a new computation type that is usually called the active type. This type is used instead of the regular computation type like `double` in the application which allows the AD tool to see all the executed operations and perform the necessary forward or reverse operations. The operator overloading approach can be extended by the expression template technique. It was first introduced by Aubert [10] to AD and constructs larger elemental operators by combining the binary (e.g. `*`, `+`) and unary (e.g. `sin`, `exp`) operators of the language. Therefore the number of arguments d_i to an elemental operation can be larger than two. For a more comprehensive overview please see [3, 5].

3 Tape data layout

In this section we will analyze the data layout for a Jacobian taping strategy [5] and a primal value taping strategy [6].

3.1 Jacobian taping

The Jacobian taping strategy described in [6] which is also used by dco/c++ [2] and adept [3] computes the Jacobian of the elemental function from Equation (2) and stores it on the tape. The Jacobian is used to evaluate the reverse mode AD Equation (3) for the elemental function. Since the number of inputs d_i is quite small, the size of the Jacobian for each element function is manageable. As described in [5] the adjoint (e.g. $\bar{a}$) of a primal variable (e.g. a) is identified with an index, that is a lookup

into the adjoint vector. Therefore the storage for an elemental operation is

- 1 byte for d_i,
- 4 byte for the identifier of $\bar{w}$,
- $8 \cdot d_i$ bytes for the Jacobian $\frac{d\phi_i}{dv_i}$ and
- $4 \cdot d_i$ bytes for the identifiers of $\bar{v}_i \in \mathbb{R}^{d_i}$.

In the setting of aggregated types, we have to store the extended version of Equation (2) which is Equation (5). Therefore, the full Jacobian matrix would need to be stored on the tape. Here, it is no longer possible to apply memory optimizations that perform a runtime activity analysis. In addition, an overhead of approximately 6% would be introduced since also the number of outputs would need to be stored on the tape. Therefore, each assignment in Equation (5) is stored as a separate statement on the tape which does not require a change in the current data layout of the Jacobian taping approach.

With this approach, care has to be taken when a statement contains a self reference, e.g., `c *= a` which can be described as $c_n = c_c \cdot a$. $c_c \in \mathbb{C}$ is the current value of c and $c_n \in \mathbb{C}$ is the new value. With the proposed strategy we would first store $\Re(c_n) = \Re(c_c \cdot a)$ and afterwards $\Im(c_n) = \Im(c_c \cdot a)$. Since c_n and c_c have the same memory position the real part of c_c is already changed when the first equation is stored. Therefore, the result of $\Im(c_c * a)$ is changed. This can be overcome by first storing all data for the right hand side for both equations. Afterwards, the data for the left hand side can be updated and stored.

The same problems arise during the reverse interpretation of (5) when the expressions are stored as separate statements. The update of $\bar{c}_c$ would be used again as $\bar{c}_n$ in the second expression. It turned out, that a specialized handling of this connection during the reverse interpretation has the same overhead as insuring that $\bar{c}_c$ and $\bar{c}_n$ are distinct entries in the adjoint vector. Since the second approach is much simpler in terms of programming effort, the distinct entries approach is chosen.

3.2 Primal value taping

The primal value taping strategy described in [6] stores the primal values of the computation and stores a function pointer for the computation of the reverse mode AD elemental function equation (3). Here, the Jacobian is computed on the fly during the reverse interpretation of the tape. As in the Jacobian taping approach, identifiers for the bar values are used.

The advantage of the primal value taping approach is, that no additional data is required when switching to the extended form of the elemental function (Eq. 5). The only difference is, that there is not one output value but p_i many. For the current implementation we assume, that all aggregated types have a constant dimension which allows us to store p_i indirectly in the function pointer. Therefore, no overhead is generated when no aggregated types are used. But the data layout of the primal value tape needs to be changed, since it is now allowed to have multiple output values per statement.

The current implementation uses, next to others, one stack where the function pointer and the identifier as well as the overwritten primal value of the left hand side are stored in one entry. Here, the entries for the left hand side need to be moved into a separate stack to accommodate the aggregated types. This is only a minor change in the implementation, but unfortunately the additional number of arguments to the function pointer decreased the performance of the reverse evaluation by about 20%. The arguments to the function pointers could no longer be stored in the registers according to the x86-64 ABI [11] and the generated push and pop assembler instructions cause the overhead. This observation leads to a reformulation of the primal value tape implementation. The existing five stacks with specific type information have been discarded and replaced with two stacks that just store binary data. The first stack stores now the data that each statement has and the second stack stores all other data which can vary from statement to statement.

The data layout of the stacks is not optimal from a software engineering standpoint. Data entries of different sizes are stored which changes the alignment of the data for each statement. Nevertheless, the performance results in Section 5 show that the new data layout improves the reverse interpretation time by about 10% even if no complex numbers are used.

4 Implementation

The implementation of aggregated type handling in CoDiPack[1] assumes that an element $d \in D$ of an arbitrary type D can be interpreted as an element $r \in \mathbb{R}^n$ with $r = \mathrm{proj}(d)$ where the projection operator $\mathrm{proj} : D \to \mathbb{R}^n$ is usually the identity. d is considered an aggregated entity that is defined by a vector of real values. The implementation in CoDiPack implements the type `AggregatedActiveType` to represent such types in the expression framework. This type can be used to implement the specific handling of aggregated types such as complex numbers. A short description of the basic ingredients in the implementation follows.

The ***`AggregatedTypeTraits`*** class defines the necessary operations on the aggregated types that are required by CoDiPack for aggregated type handling. The first operation is the vector access operator $\cdot[\cdot] : D \subseteq \mathbb{R}^n \times \mathbb{N} \to \mathbb{R}$ with $w = d[i] := d_i$. The second operation is the array construction $C : \mathbb{R}^n \to D \subseteq \mathbb{R}^n$ with $d = C(v_1, \ldots, v_n)$. For both operations also the reverse mode AD operation based on Equation 3 is required. That is $\bar{d}$ += $\frac{d \cdot [\cdot]}{dd}^T \bar{w}$ for the array access and $\bar{v}_i$ += $\frac{dC}{dv_i}^T \bar{d}$ for the array construction. A base class `AggregatedTypeTraitsBase` exists which implements these operations for the identity embedding.

[1] https://www.scicomp.uni-kl.de/software/codi/

The **`AggregatedActiveType`** class uses the defined functionality in the `AggregatedTypeTraits` to implement the common operations for all aggregated types. This is the copy constructor and the copy assignment operator. In addition it declares the array for the data storage and implements the interface such that the aggregated type fits into the template expression framework of CoDiPack.

The **`std::complex<ActiveType>`** specialization of the `std::complex` type extends from `AggregatedActiveType` and therefore adds the complex numbers of the standard C++ library to the template expression framework of CoDiPack.

The handling of aggregated types in general is completed by specialization of the `store` methods in the tape implementations such that assignments of expressions to aggregated types are recorded on the tapes. For the complex number handling the required operations for the Coupled Burgers' equation are implemented for the expression template framework.

5 Performance results for the Coupled Burgers' equation

The coupled Burgers' equation is an established test case for the performance comparison of CoDiPack implementations and is described in [5] and [6]. For simplicity, we want to use the same test case for the performance evaluations in this paper. Since the original test case is only formulated for real valued numbers, a few changes for complex numbers have to be made. One consequence is that the obtained solution is not exact. Nevertheless, we accept this error since we are only interested in performance values and memory consumption. For completeness, we recapitulate the problem formulation here.

The coupled Burgers' equation [12–14]

$$u_t + uu_x + vu_y = \frac{1}{R}(u_{xx} + u_{yy}), \tag{6}$$

$$v_t + uv_x + vv_y = \frac{1}{R}(v_{xx} + v_{yy}) \tag{7}$$

is discretized with a central finite difference scheme because we are using complex numbers. The initial and boundary conditions are taken from the exact solution given in [12] and are shifted into the complex plain by adding i. The modified boundary conditions and initial solution are

$$u(x, y, t) = \frac{x + y - 2xt}{1 - 2t^2} + i \quad (x, y, t) \in D \times \mathbb{R}, \tag{8}$$

$$v(x, y, t) = \frac{x - y - 2yt}{1 - 2t^2} + i \quad (x, y, t) \in D \times \mathbb{R}\,. \tag{9}$$

The computational domain D is the unit square $D = [0, 1] \times [0, 1] \subset \mathbb{C} \times \mathbb{C}$. As far as the differentiation is concerned, we choose the initial solution of the time stepping scheme as input parameters, and as output parameter we take the norm of the final solution.

The node for the test case consists of two Intel Xeon 6126 CPUs with a total of 24 cores and 384 GB of main memory. The computational grid contains 601×601 elements and is solved for 16 time iterations. gcc version 9 is used as the compiler. We remark that similar results are obtained with the Intel and clang compiler as well as on nodes with Epyc and Haswell CPUs. All timing values are averaged over 20 evaluations.

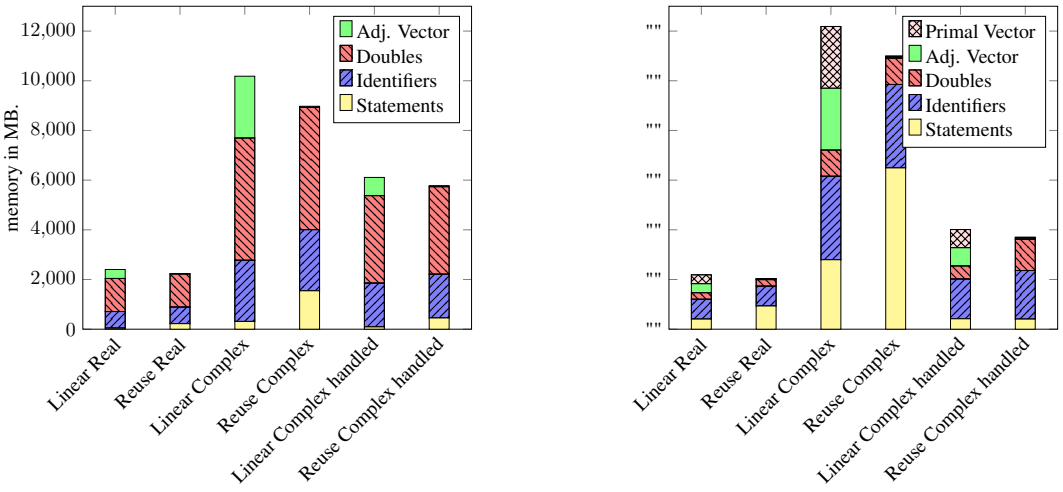
The load layout runs the same process on each of the 24 cores, which simulates a use case where the full memory bandwidth of the socket is used. This test setup is evaluated with four different configurations:

- *Real*: Baseline for performance comparisons where no complex numbers types are used.
- *Complex*: Introduction of complex numbers but no special handling is performed. This is the baseline for the complex number comparisons.
- *Real handled*: The same as the *Real* case but with the new CoDiPack version that supports complex numbers.
- *Complex handled*: The same as the *Complex* case but with the new CoDiPack version that supports complex numbers.

These four configurations are checked against the four major CoDiPack types which are *Linear Jacobian* (`codi::RealReverse`), *Reuse Jacobian* (`codi::RealReverseIndex`), *Linear Primal* (`codi::RealReversePrimal`) and *Reuse Primal* (`codi::RealReversePrimalIndex`).

5.1 Memory comparison

The memory comparison in Figure 1 shows the data for the Jacobian taping approach on the left hand side and the values for the primal value taping approach on the right hand side. The memory is increased for the Jacobian taping approach by a factor of 4.25 when switching from real to complex numbers, the factor for the primal value taping approach is 5.5. Due to runtime tape optimizations for the Jacobian approach, the factor is in general lower for this approach.



(a) Jacobian taping

(b) Primal value taping

Fig. 1: Memory consumption for the real and complex numbers in different tapes.

For the Jacobian taping approach the handling of the complex numbers reduces the memory by approximately 40%. The comparison with the *Real* case yields a memory increase by a factor of 2.6. This is in the expected range of 2.0 to 4.0 since we write two statements for each assignment and each argument has two values which are written two times. For the primal value taping approach the handling provides a memory reduction by approximately 66%. In comparison to the *Real* case the memory factor is only 1.85 which is in the expected range of 1.0 to 2.0. Here, each assignment writes only one statement and each argument has two values.

The reduction of 40% memory for the Jacobian taping approach and 66% for the primal value taping approach show how important it can be to add complex numbers or other types to the expression template framework of AD tools.

5.2 Time comparison

A runtime comparison (Figure 2) between the original and the enhanced version of CoDiPack supporting aggregated types show that the performance is nearly the same. Only in the reverse interpretation for the reuse primal case an improvement by about 15 % is seen which comes from the new data layout. A similar effect can be seen in all primal value taping cases when only one core of the node is used, but this case is usually not interesting for industrial applications.

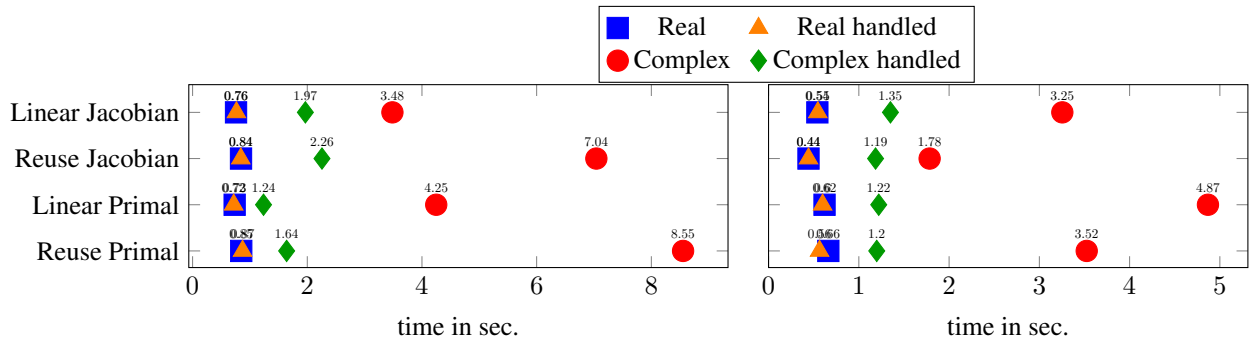
For the complex numbers we can see in the recording results that if they are not handled then the recording time is increased from 4.5 to up to 9.8 with respect to the real cases. The larger overhead in the reuse cases comes from the handling of the intermediate variables. The creation and deletion of the indices for the intermediates causes the timing degradation. If the complex numbers are handled through the expression templates of the AD tool then we have a factor of 2.7 and 1.76 for the linear Jacobian and primal taping approaches respectively. Again, the factors for the reuse index management are a little bit larger and for both cases they are similar to the memory factors.

For the recording we see that the recorded memory influences the timing to a large degree. In addition a lot of intermediate variables put strain on the reuse index management strategy.

The results for the reversal time in Figure 2 show the same general trend as the results for the recording. Since the reuse index management strategy produces smaller adjoint vectors, the reverse evaluation time is faster than the time for the linear index management approaches. The factors for the unhandled complex numbers range from 4.0 to up to 7.9 with respect to the real cases and are reduced to 2.5 for the Jacobian and to 2.02 primal value taping approaches, when the complex numbers are handled for the expression templates.

6 Conclusion

In this paper we demonstrated first that the special handling of aggregated types in operator overloading expression template AD tools can have a large benefit in terms of reducing the stored data on the AD tape. The reduced memory will also speedup the runtime of the recording and reverse interpretation of the reverse AD mode. We then demonstrated how aggregated types can be integrated into existing taping strategies such that no overhead is generated when no aggregated types are used. The runtime and memory measurements with complex numbers as aggregated types show that an improvement up to a factor of 5 is seen for the runtime and that the overhead with respect to the real valued case is in the expected margins for complex numbers. This shows that a further investigation of integrating the full set of expressions for complex numbers and other aggregated types should be performed.



(a) Recording time **(b)** Reversal time

Fig. 2: Timings for the complex number handling in different tapes for the multi configuration.

Acknowledgements Open access funding enabled and organized by Projekt DEAL.

References

[1] P. Peltzer, J. Lotz, and U. Naumann, Eigen-AD: Algorithmic Differentiation of the Eigen Library, in: Computational Science – ICCS 2020, (Springer International Publishing, Cham, 2020), pp. 690–704.
[2] K. Leppkes, J. Lotz, and U. Naumann, Derivative code by overloading in C++ (dco/c++): Introduction and summary of features, Tech. Rep. AIB-2016-08, RWTH Aachen University, September 2016.
[3] R. Hogan, ACM Transactions on Mathematical Software (TOMS) **40**(4), 26 (2014).
[4] B. Carpenter, M. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt, arXiv preprint arXiv:1509.07164 (2015).
[5] M. Sagebaum, T. Albring, and N. Gauger, ACM Transactions on Mathematical Software (TOMS) **45**(4) (2019).
[6] M. Sagebaum, T. Albring, and N. Gauger, Optimization Methods and Software **33**(4-6), 1207–1231 (2018).
[7] A. Griewank and A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, second edition (SIAM, Philadelphia, PA, USA, 2008).
[8] U. Naumann, The art of differentiating computer programs: an introduction to algorithmic differentiation (Siam, 2012).
[9] L. Hascoët and V. Pascual, ACM Transactions on Mathematical Software **39**(3), 20:1–20:43 (2013).
[10] P. Aubert, N. Di Césaré, and O. Pironneau, Computing and Visualization in Science **3**(4), 197–208 (2001).
[11] H. Lu, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, AMD64 Architecture Processor Supplement (2018).
[12] J. Biazar and H. Aminikhah, Mathematical and Computer Modelling **49**(7), 1394–1400 (2009).
[13] A. Bahadır, Applied Mathematics and Computation **137**(1), 131–137 (2003).
[14] H. Zhu, H. Shu, and M. Ding, Computers & Mathematics with Applications **60**(3), 840–848 (2010).