# The Efficiency of geophysical adjoint codes generated by automatic differentiation tools

**A. V. Vlasenko, A. Köhl, and D. Stammer**

Center für Erdsystemforschung und Nachhaltigkeit, Universität Hamburg, Germany
Corresponding author: A. V. Vlasenko (andrey.vlasenko@uni-hamburg.de)

## Abstract

The accuracy of numerical models that describe complex physical or chemical processes depends on the choice of model parameters. Estimating an optimal set of parameters by optimization algorithms requires knowledge of the sensitivity of the process of interest to model parameters. Typically the sensitivity computation involves differentiation of the model, which can be performed by applying algorithmic differentiation (AD) tools to the underlying numerical code. However, existing AD tools differ substantially in design, legibility and computational efficiency. In this study we show that, for geophysical data assimilation problems of varying complexity, the performance of adjoint codes generated by the existing AD tools (i) Open_AD, (ii) Tapenade, (iii) NAGWare and (iv) Transformation of Algorithms in Fortran (TAF) can be vastly different. Based on simple test problems, we evaluate the efficiency of each AD tool with respect to computational speed, accuracy of the adjoint, the efficiency of memory usage, and the capability of each AD tool to handle modern FORTRAN 90-95 elements such as structures and pointers, which are new elements that either combine groups of variables or provide aliases to memory addresses, respectively. We show that, while operator overloading tools are the only ones suitable for modern codes written in object-oriented programming languages, their computational efficiency lags behind source transformation by orders of magnitude, rendering the application of these modern tools to practical assimilation problems prohibitive. In contrast, the application of source transformation tools appears to be the most efficient choice, allowing handling even large geophysical data assimilation problems. However, they can only be applied to numerical models written in earlier generations of programming languages. Our study indicates that applying existing AD tools to realistic geophysical problems faces limitations that urgently need to be solved to allow the continuous use of AD tools for solving geophysical problems on modern computer architectures.

## 1. Introduction.

To date, numerical modelling is a widespread and generally accepted approach for solving complex mathematical equations of physical, biological, or chemical processes in climate and earth system sciences. However, the accuracy of respective solutions fundamentally depends on the choice of – typically uncertain - model parameters. One therefore is usually faced with the following two questions: (i) How sensitive are model solutions to the detailed choice of model parameters, and (ii) what is the optimal set of these parameters required to minimize the difference between a simulated process to a given set of observations. One elegant way to answer these questions involves the computation of derivatives of the corresponding numerical model with respect to its parameters or state variables [1], [2],[3],[4],[5].

Generally, model derivatives can be generated in three different ways, the simplest of which is an estimation of approximate derivatives by applying a finite difference method. Being simple, however, this method is always plagued by approximation errors. The second way is to derive the model derivatives manually. Such differentiation leads to exact derivatives but is labor-intensive and therefore impractical for large numerical models. The third option is to use an algorithmic differentiation (AD) tool, which almost automatically provides the exact derivatives of any complex function represented by a numerical code with only little extra effort by the user.

For simplicity we introduce two definitions commonly used in AD. We consider a numerical model as mathematical operator that acts on state variables $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ and returns an output $\boldsymbol{y} = (y_1, y_2, \dots, y_n)$ (e.g. state variables at the end of computation). Let $\mathbf{J}$ be the Jacobian matrix of $\boldsymbol{y}$ with respect to $\boldsymbol{x}$ and $\boldsymbol{s}$ be an n-dimensional column vector, with only $s_i$ element equal to one and the rest to zero. The differentiation of $\mathbf{y}$ with respect to $x_i$ produced by the AD tool can be presented as a multiplication of $\boldsymbol{s}$ by $\mathbf{J}$, where $\mathbf{J}\boldsymbol{s}$ gives the derivative of $\boldsymbol{y}$ with respect to $x_i$. The corresponding computer code, generated by the AD tool, is called the tangent linear model and the derivatives referred to as tangents. The multiplication of $\boldsymbol{s}^{\boldsymbol{T}}\mathbf{J}$ (superscript T means transpose) provides the derivative of $y_i$ with respect to $\boldsymbol{x}$. The corresponding computer code, generated by the AD tool, is called the adjoint model and the derivatives referred as adjoints. These two ways of differentiation of a numerical model performed by the AD tool are referred as tangent linear and adjoint modes[1], respectively. A

[1]The terms adjoint and tangent linear **model<u>s</u>** should not be confused with terms adjoint and tangent linear **mode<u>s</u>**.

further field of AD tools is the computation of higher order derivatives. In particular, the second derivative, the Hessian matrix, or approximations to it, are used in parameter optimization algorithms such as the Gauss-Newton algorithm, and the posterior evaluation of the uncertainties of the estimated parameters requires the inverse of the Hessian.

Today, AD is a well-established field of applied mathematics, formulated initially in the middle of the previous century. In detail, AD techniques are based on the mathematical formalisms of differentiation represented by a set of simple, well-known mathematical operations. They are designed for the numerical differentiation of mathematical functions of any complexity represented by computer codes [6],[7]. The resultant derivatives are exact up to the numerical precision, i.e., no truncation is used so that no approximation errors appear in the output code.

By now more than 42 different AD packages exist, which were developed for different applications and programming languages (see www.autodiff.org for details). They use different strategies for computing derivatives [8]. Among them source code transformation (SCT) and operator overloading (OO) are the most common strategies [9]. SCT AD tools replace the original source code by a new source code, usually in the same programming language that includes the statements for computing user specified derivatives. In contrast, the OO AD tools generally leave the original source code as it is and the sequence of operations in the original source code for the function to be differentiated remains unchanged. However, it requires a change in basic data types for numbers and vectors for supporting differentiation, and establishes a special polymorphism[2] for mathematical operators called overloading. The change of data type is done for all differentiating and differentiated variables (which may be done automatically by the AD tool) in such a way, that any variable of a changed type holds both its actual value and its derivative. Overloading reintroduces the original mathematical operators in the source code by splitting each operator's action. An action of any overloaded elementary mathematical operator depends on whether it is applied to the value of the variable or to its derivative. For an actual value of the variable, the operator remains the same as it was before the overloading, while it operates according to the rules of differentiation when being applied to the derivatives of the variable.

Respective tools have been applied to a variety of problems in many research areas [10],

[2] Polymorphism is the ability of a function to be applied to different types of variables. The result of application depends on the types of these variables.

[11],[12],[13],[14],[15],[16],[17] .But, although all these AD tools simplify the computation of the derivatives, the computational efficiency of these derivatives strongly depends on the internal structure of the AD tool that produced it. In particular, existing AD tools differ substantially in the design, readability and computational efficiency of resulting numerical codes. Moreover, the AD tools also differ by their ability to operate with various high level programming languages. This difference in structure and design of different AD tools has a great effect on the performance of the adjoints that they produce. Applying these theories to real geophysical problems remains a huge challenge given the size, complexity, and often the non-linear nature of those problems. Understanding the efficiency of existing adjoint codes is therefore a prerequisite to applying them to real climate science problems. Rather than developing new algorithms, the goal of this paper is to analyze the practicality and efficiency of existing OO and STC based AD tools for geophysical problems. We focus our study on adjoint modes, since the computation of sensitivities in climate modeling is mainly associated with the execution of adjoint models. Based on simple test problems, we evaluate the efficiency of each AD tool with respect to computational speed, accuracy of the generated adjoint, the efficiency of memory usage, and the capability of each AD tool to handle modern FORTRAN 90-95 elements such as structures and pointers.

In detail, we compare adjoint codes generated by Transformation of Algorithms in Fortran (TAF) [18] and Open_AD [19] and TAPENADE [20] as the AD tools representing SCT approach. As for the OO based AD tool, we choose NAGWare [21]. All these AD tools have the tangent linear and adjoint mode for differentiation. We note that the same derivatives can be obtained with either an adjoint or a tangent linear mode; which mode is more appropriate is a question of efficiency.

Here, we focus on the performance of the adjoint mode of differentiation, because most of difficulties related to memory usage efficiency and execution runtime are typically associated with this mode [22]. Our case studies are specifically intended to identify strengths and weaknesses of the OO and SCT procedures and provide the corresponding benchmarks for their efficiency in climate applications. The ability to operate with different programming language features is also compared, and their effect on the computational speed performance (CSP) is investigated. By doing so, we expect to give recommendations to model developers guiding their choice of the proper AD procedures.

The remaining paper is organized as follows. A brief summary of the methodology for testing

the compilers is given in the second section. The test-bed codes on which the efficiency of the compilers is investigated is given in the third section. Section 4 gives a short theoretical overview of the differentiation machinery of the AD tools. Tests on the compiler's speed performance, accuracy and the efficiency in memory usage are described in fifth and six sections respectively. Section 7 describes the ability of the compilers to handle pointers and user defined structures. The conclusion is given in Section 8.

## 2. Test of the AD tools

Our inter-comparison is based on the latest versions of TAF (versions 2.3.5-2.8.4), Open_AD (versions S440,S469, S493), Tapenade (version 3.10) and NAGWare (version 5.1) AD packages. For a quantitative comparison of all three AD tools, we have prepared several sample codes to test computational speed, memory usage efficiency (MUE), accuracy, and compatibility with different programming features such as pointers, structures, and user defined data types. In order to test how the efficiency of computations of adjoints depends on the hardware, the tests were carried out on different platforms. All executables were subsequently run on three different computer platforms (Intel CoreDuo E8500 2Gb DDR3, AMD Sempron AM2 2Gb DDR2 and Intel Core I5-2500 8Gb DDR3), and the corresponding computational times were used to evaluate the performance. Although the SCT AD tools allow utilizing any FORTRAN compiler, the NAGWare AD tool is embedded in the NAG FORTRAN compiler, and consequently compilation has to be done with this. To avoid differences in execution time related to the usage of different compilers, we chose NAG for all three AD tools. The optimization options of the compiler were the same.

Throughout the remaining paper, the distinction of variables will be used that is common to AD: the input variables with respect to which the differentiation initially should be performed are referred to as independent variables. The output dependent variables are those whose derivatives are required. All other variables in the source code are divided into two classes. A variable is referred to as active and belongs to an active type if it depends on at least one independent variable and influences at least one dependent variable. All other variables are considered as passive and belong to the passive type.

## 3. Test bed codes

Test of the computational speed and accuracy were performed using three different codes of increasing complexity in the following, called **linear model, regularized Lorenz model (RLM),** and **Shallow Water model (SWM) code.** The idea of such an approach involving codes with different levels of complexity is straightforward. An optimal adjoints in sense of runtime and memory usage should be easier to generate from codes, representing trivial linear mathematical problems, which exclude the need for recomputations. Such trivial mathematical problem is represented by the linear model example and optimal adjoints with the best CSP for OpenAD, NAGWare, Tapenade and TAF were achieved. The obtained level of CSP is used later as a benchmark (the upper bound, in fact) for the comparison of the CSPs obtained for the more complex codes, like the shallow water model (SW code), or the Lorenz model with relaxation (Lorenz code) [23]. With the Lorenz and SW codes, one can reveal how the complexity of these codes, i.e. the amount of inter-commuting active variables in a single expression, and non-linear and implicit dependencies between these variables presented in the code, affects the performance of the discussed AD tools.

**i) The linear model.**

This example represents a simple data assimilation problem with the following linear constraints:

$$\min_{x_0} \left|\left| x - x_{obs} \right|\right|_2^2$$

such that

$x(t) = \mathrm{a} \cdot \sin(t), \quad x(0) = x_0.$

In this problem, it is required to find an optimal initial state $x_0$ for computation of oscillations of a pendulum with the given amplitude $a$. In the optimum solution, the difference between the computations $x$ and observations $x_{obs}$ is minimal. Here, we used as optimization procedure for data assimilation procedure the gradient descent method in which the gradients are calculated by means of the AD routines. The code describing this problem is nearly 50 lines long. It contains only one driver program in which the variables are set and one external subroutine where the model equations and cost are computed.

**ii) The regularized Lorenz model code.**

The RLM code represents a simplified version of a data assimilation method with relaxation [23], applied to the Lorenz 1963 model [24]. The corresponding system of equations reads:

$$\min_{\sigma,\rho.\beta,p}\{\|x - x_{obs}\|_2^2 + \|y - y_{obs}\|_2^2 + \|z - z_{obs}\|_2^2 + p^2\}$$

such that

$$\frac{dx}{dt} = \sigma(x - y) + p(x - x_{obs})$$
$$\frac{dy}{dt} = x(\rho - z) - y + p(y - y_{obs})$$
$$\frac{dz}{dt} = xy - \beta z \ + p(z - z_{obs})$$

Here $x_{obs}, y_{obs}, z_{obs}$, are the observations of $x, y, z$ measured on a time period$[-T, 0]$. The goal of this variational problem is to find a set of parameters of the Lorenz (1963) system that optimally fit the model trajectory to given observations. The original method is based on an iterative 4D-var data assimilation algorithm, where the model equations are relaxed by a nudging term depending on the nudging parameter p. The usage of nudging and relaxation suggested by [23] stabilizes the chaotic behavior of the Lorenz (1963) model, which in turn allows performing assimilation for extensive periods of time. Note that the cost function is minimized with respect to the Lorenz (1963) parameters including the nudging parameter $p$. As before, we used the standard gradient descent method for solving this problem. With decreasing model-data differences, the impact of nudging on the model simulation reduces. Due to the models nonlinearity, its adjoints depend on the specific model's trajectory, which adds complexity.

**iii) The SWM code.**

The basis for the large code is the data assimilation problem discussed by [25]. It is constructed to mimic a simplified model of the Antarctic Circumpolar Current using the Shallow Water approximation. In that model a wind driven barotropic flow in a rectangular zonal channel over a central meridional Gaussian sill is considered. The sill extends across the channel and the overflowing current is connected to a perturbation of the sea level in that place. The shape and size of this perturbation depends on the sill height depending on the latitudinal direction.

For a stationary case, the goal of this exercise is to estimate the height of the sill,$h_g$, from observations of the sea level by minimizing the difference of the simulated sea surface height to its observations. The mathematical formulation of that data assimilation problem is as follows:

$$\min_{h_d} \frac{C}{2} \|h_{obs} - h_d\|_2^2$$

In order to find the stationary solution, the following time dependent Shallow Water equations are integrated in time until the solution becomes stationary:

$$\begin{cases} \dfrac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{g}\dfrac{\partial \boldsymbol{u}}{\partial x} - \boldsymbol{f}\boldsymbol{u} + r\dfrac{\boldsymbol{u}}{h} = \dfrac{\tau}{h}, \quad h = h_b + h_d \\ \dfrac{\partial h}{\partial t} + \nabla(\boldsymbol{u}h) = 0 \\ \boldsymbol{u_{t=0}} = \boldsymbol{u_0}, h_{t=0} = h_o \end{cases}$$

Here $h$ is the total depth including the sea surface height $h_d$; $h_b$ is the bathymetry with respect to the sea surface zero value depending on sill's height $h_g$, which is a function of latitude only; $h_{obs}$ is the observed sea surface height; $\boldsymbol{u} = (u, v)$ are longitudinal and latitudinal fluid velocity components respectively; $\tau$ is the wind stress, which is a known function of time and space; $f = f_0 + \beta y$ is the Coriolis parameter; r is the parameter of bottom friction, $t \in [0:T]$ is time and $\boldsymbol{x} = (x, y)$ is the coordinate vector.

This data assimilation problem is solved numerically in the following setup. The channel is centered at approximately 55°S, it has solid boundaries in the north and the south and idealized bottom topography. The motion is considered on a β-plane. The horizontal and vertical extents of the model domain are 4000 and 1600 kilometers, respectively. The initial guess for the height of the crest is set to 400 m above the average depth of 4000 m. Its half-width of 200 km corresponds to an average slope of 1 m in 1000 m. The numerical implementation of this problem is given at http://www.mcs.anl.gov/OpenAD the OpenAD official website.

## 4. Theoretical basis

The computational time required for executing an adjoint code prepared by an AD tool directly depends on the method used by that tool for interpretation and differentiation of the

original code. To understand why adjoint codes produced by different AD tools require different computational time, one needs to consider the basics of algorithmic differentiation. The general idea underlying an AD tool is the construction of a computational flow graph representing a target mathematical function, and the application of the chain rule to this graph.

For simplicity, consider a source code consisting only of a single program aimed at computing a mathematical function, which should be differentiated with respect to some of its arguments. Let this program be without subroutine calls, "if" branches and loops. In such a case, the function can be presented by a sequence of elementary mathematical operations. This sequence in turn may be presented in the form of a directional computational graph with independent variables as input, and the dependent variable (the function) as output. If the active variables in the graph are described as edges, and immediate arithmetic interrelations between these variables are located in nodes[3], the computational graph can be considered as a path from independent to dependent variables. Note that any elementary mathematical operation at maximum involves only two variables. Hence the construction of such graph requires that not more than two edges are input at each node. For some mathematical expressions (e.g. $a \cdot \sin(b)$), the direct representation as a set of node-connected pairs of vertices is not possible. For resolving this issue, auxiliary edges and nodes are introduced. They represent auxiliary variables and operations between them, which split the original expression in a set of subexpressions with at maximum two variables in each subexpression. For instance, for $c = a \cdot \sin b$, auxiliary edge $b_1 = \sin b$ with node sin() are introduced and the original expression simplifies to $c = b_1 \cdot a$. The number of these auxiliary edges is chosen in such a way that each node has only two entering edges. In case of the SCT AD tools, these auxiliary edges are presented by corresponding auxiliary variables in the differentiated code. Computational graphs significantly simplify the implementation of the chain rule: it reduces to an accumulation (summation) of derivatives obtained by differentiating each vertex with respect to a pair of its immediately preceding vertices. Depending on the mode, the accumulation of the derivatives starts from the entrance of the graph (tangent linear mode) towards its exit or from its exit (adjoint mode) towards its entry.

We note that splitting composite expressions with the aid of auxiliary variables may not be unique. For instance, the expression $y = e^{x+z}$ may be split in two ways; 1) $a = x + z$, $y = e^{a}$ with a single auxiliary variable $a$ or 2) $a = e^{x}$, $b = e^{z}$, $y = a \cdot b$, where two

[3] In a real case with loops, if-else braches etc. a computational flow graph should be considered.

auxiliary variables are used. The first case corresponds to a two edged graph, while the second case corresponds to the three edged graph. This example shows that the same composite mathematical expression may be presented with different computational graphs. The choice of the graph with shortest path minimizes the amount of applications of the chain rule and hence results in higher efficiency in computing derivatives.

The scheme described above is an oversimplification of real algorithms underlying the AD tools. However, it shows that constructing a computational graph with an un-optimized length results in suboptimal memory usage efficiency (MUE) and computational speed performance (CSP) efficiency. Finding a graph with the optimal path is still a research question and different AD tools employ different strategies.

The implementation of the chain rule requires that, for the places where the active variables depend non-linearly on other variables, the actual values of those variables are known. In the case of tangent linear mode, this requirement is easy to fulfill, since the differentiation starts from the entry of the graph and the accumulating derivatives and all required variable's values are estimated at the same time during the forward sweep of the source code. In the case of adjoint mode, this problem is more complex, since the differentiation starts from the exit of the graph and hence the estimation of the derivatives requires the values of all active variables, beginning from the end of the graph.

The values of active variables may be obtained during the adjoin run by implementing either the "store-all" or the "recompute-all" strategy. The "recompute-all" strategy implies that all active variables, which are overwritten during the forward sweep, are recomputed during the adjoint sweep. This strategy is utilized by default in NAGWare, OpenAD and TAF. The rational behind a "store-all" strategy is to store all values that are overwritten by an assignment during the forward sweep just before the assignment and then restore them before the adjoint of this assignment in the reverse sweep. The array where these variables are stored is called "tape". In order to minimize the set of saved variables in a tape, the store-all strategy is performed together with the "to be recorded" (TBR) analysis. It is a static data flow analysis, which determines the variables that are not used in derivative computation and then excludes them from taping [26]. The store-all strategy together with TBR analysis is realized in Tapenade [20],[27]. Note that the reverse mode implemented with the store-all strategy has additional memory cost proportional to the execution time of the forward sweep; the reverse

mode implemented with recompute-all strategy has no additional memory cost, however, its execution time is proportional to the square of the execution time of the forward sweep [28].

Although the recomputed-all strategy is set in TAF, OpenAD and NAGWare by default, these tools however, allow the initiation of the tape by the user [18],[19],[21]. To do so, the user explicitly specifies the names of the active variables and the place in the source code where they should be recorded. The AD tool has only to exclude these variables from recomputation in the adjoint sweep. All variables recomputed in forward sweep that are not included in tape but required for adjoint computations will be recomputed in the adjoint sweep.

To avoid recomputation in all test codes, the same tapes for all active variables were generated. In the case of the SWM test code, the generation of continuous tape for the entire adjoint sweep was not possible and the same checkpointing scheme for all AD tools was established, which is as follows: The SWM code is represented by a set of subroutines executed in a time stepping loop, where the active variables are overwritten at each iteration. This loop was split into an outer and an inner loop with subroutines embedded in the inner loop. The total number of operations remains the same. When the computations in the inner loop are finished, all values of the active variables are recorded in a special checkpoint. Hence the amount of checkpoints equals to the amount of the outer loop steps. The tape is initiated only in the inner loop during the reverse sweep and is reset when it is finished. Thus, departing from the nearest checkpoint, the values of active variables are recomputed in the inner loop once again and stored in the tape. After that, the corresponding values of derivatives are computed and accumulated. Then the tape is reset and the recomputations in the inner loop start from the next checkpoint. The size of the inner loop is chosen in such a way that the generated tape almost completely fills the available memory.

## 5. Testing the computational speed and accuracy

The tests for all AD tools were performed as follows: At first, the adjoint source codes were generated from test source codes with aid of TAF, Tapenade and OpenAD. Then the executables of the adjoint executables were obtained by compiling them with NAG compiler. The adjoint executables for NAGWare were obtained directly by application of this AD tool to all test source codes. Since the actual runtime of the executables strongly depends on the platform, the tests were subsequently carried out on three different computer platforms (Intel CoreDuo E8500 2Gb DDR3, AMD Sempron AM2 2Gb DDR2 and Intel Core I5-2500 8Gb

DDR3), and the corresponding computational times were used to evaluate the performance. These results are presented in Table 1. In the table, the CSP ratio is obtained as $ADJ_{AD}/FORW$, where $FORW$ is the runtime of the initial source code. This ratio shows how much slower the execution of adjoint computations are in comparison to the execution of the corresponding forward computation, thus bigger ratio numbers correspond to worse CSP. The invariance of these ratios to the choice of the executing platforms is found in all experiments.

Note that among the SCT AD tools, Open_AD has the worst CSP value for all 3 test codes, which differs from CSP of TAF and Tapenade typically at least by a factor of two. The analysis of the adjoint source codes generated by TAF, Tapenade, and Open_AD revealed that Open_AD introduces excessive auxiliary variables into the adjoint code which do not influence the dependent variables or simply alias other variables in the computations. The idle manipulations with these variables and their allocation in random access memory (RAM) require additional computational time. The amount of idle operations produced by Open_AD was at least two times bigger than the amount of idle operations produced by TAF and for all test codes.

Since the adjoint codes generated by Tapenade and TAF from the linear test code have a similar pattern, they should have similar CSP. However, this is not confirmed in the experiments. The difference in CSP accounts for the fact that TAF can use both static (constant shape arrays) and dynamic tapes (flexible shape arrays), while Tapenade uses only dynamic tapes. Note that the usage of dynamic tape results in a small reduction of CSP, since its permanent reallocation during the run requires additional computational time. For better CSP in the tests, static tapes were set in TAF.

The inspection of adjoints of RLM code revealed that although $x, y, z$ were declared for taping, TAF still introduces a single recomputation for each of these variables inside the main loop of the adjoint code, although TAF did not report any recomputation warnings. During each loop-iteration, all these variables are loaded from the tape but then recomputed in the next step. Such recomputations are equivalent to one excessive forward sweep. Note that, if continuous taping is applied, the run time of adjoint executable should consist only of single forward and reverse sweeps. Such run times have adjoint executables of the linear test code, generated with the aid of TAF and Tapenade. However, due to extra recomputations, the runtime of the adjoint code generated by TAF is increased by the run time of a single forward sweep. Because of this, TAF's CSP obtained for the RLM test code are 1.6 times worse than

RCSP and CSP obtained for the linear model. The inspection of the adjoint of RLM code, produced by Tapenade revealed that it did not initiate excessive recomputations and its CSP values remain the same as for the linear model.

In case of the SWM code, the same checkpointing scheme was applied for all AD tools. The tests revealed that CSP for TAF and Tapenade obtained for the SWM code differ much less than the same ratios for linear and RLM codes. The inspection of the adjoint code did not reveal excessive recomputation, while in case of OpenAD most of operations fall on excessive variables.

Tests with the NAGWare tool showed that its adjoints are the slowest of all three test-codes. Comparing the CPS for all tools (see Table 1) shows that TAF and Tapenade generated codes are 20-30 times faster than NAGWare, while OpenAD has a 5.3 – 9.23 times better CSP than NAGWare.

Along with the evaluation of the computational speed performance, the similarity between the values of the resultant derivatives obtained by means of NAGWare, TAF, Tapenade and OpenAD were checked on all three test-codes. All derivatives were computed with double precision. The value for gradients obtained with double precision accuracy coincides for all three AD tools in case of the linear model, with only differences in the last digit. Similar results were obtained for NAGWare and OpenAD tools for the RLM code, while TAF produces the difference in the 7-th digit. Finally, for all three AD tools, the gradients differ in the 7-th digit in the case of SWM code.

## 6 Testing the efficiency of memory usage

Due to the taping process, running the adjoint models in most cases requires more computer memory for data storage than for the execution of the forward codes. The amount of this data-storage related memory depends on the AD tool used for the construction of the adjoint model. In this test, the memory requirements of adjoint executables differentiated by TAF, NAGWare, Tapenade and OpenAD were tested. The results of the measurements are given in Table 2. There, the values in each column represent the relative MUE for a given AD tool obtained for all three test codes. Each relative MUE is the ratio between the amounts of memory required by the same adjoint codes obtained by the OpenAD and the specific AD tool. The choice of OpenAD's MUE as a reference one is made because of it worse

performance in all tests. Hence, big numbers in the table corresponds to better MUE. The table demonstrates that Tapenade is the most efficient in memory usage. NAGWare is less efficient, but still it is better than TAF and OpenAD.

The advantage in memory usage of Tapenade may be explained by two factors. Firstly, it is related to the altering of the tape sizes during the reverse sweep, which in turn allows minimizing the used memory. This was achieved by using stacks for taping: the used values in the reverse sweep are removed from the stack during the reverse sweep and hence release the occupied memory. In contrast, TAF uses regular arrays for dynamic or static tapes. The dynamic tape in TAF grows during the forward sweep, but once it is finished it remains the same during the reverse sweep. Secondly, the advantage in memory usage of Tapenade in case of RLM may be connected to its efficient TBR analysis. Note that efficient TBR analysis excludes recomputations of the taped variables. By contrast, TAF and OpenAD still initiate a single recomputation of these variables in the adjoint code of RLM (see section above). Thus, in the adjoint code generated by these tools the same variable is stored twice: in tape and in a representative array that is used for recomputation. As the result, duplex saving of the same variable increase memory costs.

## 7 Handling structures and pointers.

The ability of Tapenade, TAF, OpenAD and NAGWare tools in their recognition and correct interpretation of pointers and structures is investigated in this section. For these tests the data assimilation computations on the basis of linear, RLM and SWM scripts discussed in the second section were repeated with a small modification. In each test active variables were substituted with pointers and structures, and after that the same was done for passive variables. The adjoint codes and corresponding executables were generated for each modified source code. The values of derivatives obtained from these executables were compared with the derivatives obtained in the experiments described in previous section where no pointes and structures were used. If the derivatives obtained with and without pointers and structures differed, the corresponding adjoint codes were investigated and compared, and the reasons that caused difference were figured out. The corresponding tests are discussed in the following paragraphs with results summarized in the Table 3.

### 7.1 Pointers on active values.

The idea of the test is to check the ability of the AD tools in recognizing and interpreting pointers on active values, without discussing the sense of their practical usage. It was investigated how the substitution of an active variable with a corresponding pointer affects the correctness of gradient computations. The tests revealed that for the linear model, except Tapenade, all three AD tools were able to interpret pointers correctly, even when the active variables were not used. Tapenade ignored the pointer, considering it as a passive variable, and generated a wrong adjoint. However, for the RLM and SWM codes, TAF and OpenAD failed when using only a single pointer. This pointer was not recognized as an active variable and did not appear as part of the expressions in the adjoint codes. At the same time, the NAGWare passed the tests for all codes, while Tapenade failed the test completely. Therefore, we conclude that in general Tapenade, TAF and OpenAD is likely to fail differentiating the code correctly if it contains pointers on active variables, a fact that has significant consequences for model developers. The conclusion of this test is summarized in the first row of Table 3.

### 7.2 Alias pointers.

By alias pointers, we mean a case when two or more pointers are pointing to the same variable. Similar to the previous paragraph, we test here only the ability of AD tools in recognizing and interpreting alias pointers. As before, all three linear, RLM and lSWM codes were used for tests. At least two pointers that point on the same active variable were set in these codes. It was found that Tapenade failed all tests, but the other three AD tools delivered correct adjoints when they were applied to the linear model. However, the adjoints produced by TAF and OpenAD calculated incorrect gradients for RLM and SWM codes. The results of this test are summarized in the second row in Table 3.

### 7.3 Pointers on passive values.

In the cases when pointers were set only on passive variables, all four AD tools deliver correct adjoints with differences only in the 7-th digit. Therefore, we conclude that Tapenade NAGWare, OpenAD and TAF passed the test successfully. The results of this test are listed in row 4 of Table 3.

### 7.4 User-defined types (structures).

Structures, or user-defined types, are a quite common feature in modern models such as the ICOsahedral Nonhydrostaticl( ICON) model [29]. The idea of using structures is to combine groups of variables into mathematically logical units, e.g., state vectors, and then operate with these units as variables. The correctness of the adjoints to such models produced by an AD tool depends on the capability of the AD tool to interpret respective structures. To test this capability for Tapenade, TAF, NAGWare and OpenAD, the three problems from the second section were employed again. In each of these problems, all active variables were combined into a single structure. After such substitution, the variables cannot be accessed directly in the code; instead one has to address them as a part of a structure. Therefore, the active variables in the equations of all three codes were substituted with the corresponding elements of the structure. Subsequently, we generated adjoints for the three test codes without using structures and compared the results.

This test revealed that Tapenade failed the test completely, but the other three AD tools produced correct adjoints for the linear model, although only NAGWare was able to produce correct adjoints for RLM and SWM codes. On the other hand, if structures consisting only of passive values are introduced into the scripts, all four AD tools deliver correct adjoints (see the fifth and six rows in Table 3).

## 8. Conclusions

In this study we analyze the practicality of four existing OO and STC based AD tools for generating adjoint codes of geophysical models, and tested their efficiency with respect to computational speed and accuracy of the generated adjoint plus the efficiency of memory usage. In general terms, the best computational speed performance (CSP) is found for SCT AD tools. As discussed in Section 5, the CSP of such AD tools' adjoint code in some of our test cases is only 2.5 times slower than the original forward code. On the other hand, the SCT AD tools usually have limitations in supporting some elements of modern programming languages. In contrast, the operator overloading (OO) AD tools have none of those limitations intrinsic to SCT tools, allowing the generation of adjoints to modern climate model codes using all FORTRAN 90-95 coding. However, they have much degraded CSP (see Sec. 2 for details).

The finding of the present paper can be summarized as follows:

- The NAGWare showed the best compatibility with all types of coding elements (e.g., pointers and structures). For the more complex scripts that represent non-linear problems, Tapenade, TAF and OpenAD were not able to interpret pointers correctly; the same holds for structures containing active variables.
- The Tapenade and NAGWAre are the most efficient in terms of memory usage.
- However, NAGWare has the worst computational performance. The fastest performance is obtained by the TAF and Tapenade generated code, which is 20-30 times faster than NAGWare, while OpenAD showed a CSP being 5.3 – 9.23 times better than NAGWare.

From our results, any model developer interested in creating an adjoint from a forward code using the existing AD tools at present faces a serious dilemma: while it is almost mandatory to use SCT tools if one were to generate an efficient adjoint model, it restricts the model programming language to older (seemingly obsolete) programming standards. Having a new model developed in FORTRAN 95, however, requires using OO procedures, capable of handling these programming standards, for the generation of the adjoint. On the other hand, the generated adjoint model is too slow to be applicable for any state-of-the art climate-related problem, rendering modern models useless as source code for an adjoint model unless they were already written with this application in mind.
Our results therefore suggest that every-day cutting edge assimilation efforts in weather, ocean or climate applications for the foreseeable future will heavily rely in SCT tools and therefore will require to be written in previous generation FORTRAN. At the same time our results almost demand that substantial effort is being spent on bringing SCT tools to the next level where modern circulation models can be adjointed in an efficient way in support of climate research and operational applications alike.

## Acknowledgements

This work was funded in part through a Max Planck Society Fellowship to D. Stammer, the European Union 7th Framework Programme (FP7 2007-2013), under grant agreement n.308299 NACLIM (www.naclim.eu), and the CliSAP Excellence Cluster of the University of Hamburg, funded through the DFG.

## References

[1]. Bücker H.M., B. Lang, A. Rasch, C.H. Bischof. Computing sensitivities of the electrostatic potential by automatic differentiation**.** Computer Physics Communications Volume 147(1), 2002, Pp. 720–723.

[2]. Saltelli A., M. Ratto, S. Tarantola, F. Campolongo, Sensitivity analysis practices: Strategies for model-based inference, Reliability Engineering and System Safety 91 (2006) 1109–1125.

[3] Wunsch, C.W. Discrete Inverse and State Estimation Problems. With Geophysical Fluid Applications, Cambridge Un. Press, 2006, Cambridge.

[4] Nichols, N. K. Mathematical concepts of data assimilation. In Data Assimilation: Making Sense of Observations. (Lahoz, W., Khattatov, B. and Menard, R. Eds.), Springer, 2010, p 13–39.

[5] Molkenthin C, Scherbaum F, Griewank A, Kuehn N, Stafford Pet al., 2014, A Study of the Sensitivity of Response Spectral Amplitudes on Seismological Parameters Using Algorithmic Differentiation, Bulletin of the Seismological Society of America, 2014, **104** pp: 2240-2252.

[6] Naumann U. The Art of Differentiating Computer Program. 2012, SIAM.

[7] Griewank, A., and A. Walther. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Ed., Siam, 2008.

[8] Gay D.M., Semiautomatic Differentiation for Efficient Gradient Computations. In Automatic Differentiation of Algorithms: Applications Theory and Implementations, (H. M. Bücker and G. Corliss and P. Hovland and U. Naumann and B. Norris Eds.), Springer, 2005, pp. 147--158.

[9] Bischof C. H., H. M. Bücker, Computing Derivatives of Computer Programs

Modern Methods and Algorithms of Quantum Chemistry: In Proceedings, Second Edition, NIC-Series, 2000, 3, pp. 315-327.

[10] Kaminski T., Giering R., Scholze M., Rayner P., Knorr W. An Example of an Automatic Differentiation-Based Modelling System. In Computational Science—ICCSA 2003, Part II, LNCS, 2003, **2668**, p. 95—104.

[11] Marotzke, J., Giering, R., Zhang, K. Q., Stammer, D., Hill, C. and Lee, T. Construction of the adjoint MIT ocean general circulation model and application to atlantic heat transport variability, J. Geophys. Res., 1999, **104**, C12:29, 529–547.

[12] Recent Advances in Algorithmic Differentiation In Lecture Notes in Computational Science and Engineering (Forth S., Hovland P., Phipps P., Utke J., Walther A. Eds.), Springer, 2012, 87, doi = 10.1007/978-3-642-30023-3.

[13] Advances in Automatic Differentiation. In LNCSE, (Bischof C.H., Bücker H.M., Hovland P., Naumann U., Utke J. Eds.), Springer, 2008, **64**.

[14] Automatic Differentiation: Applications, Theory, and Implementations. In LNCSE, ( Bücker H.M., Corliss G.F., Hovland P., Naumann U., Norris B. Eds.), Springer, 2005, 50.

[15] Automatic Differentiation of Algorithms: From Simulation to Optimization. In Computer and Information Science, (Corliss G.F., Faure C., Griewank A., Hascoët L., Naumann U. Eds.), Springer, 2002.

[16] Automatic Differentiation of Algorithms: Theory, Implementation, and Application (Griewank A., Corliss G.F. Eds.), SIAM, 1991.

[17] Sambridge A., Rickwood P., Rawlinson N. and. Sommacal S. Automatic differentiation in geophysical inverse problems, Geophys. J. Int.", 2007. 170(1), pp. 1—8.

[18] Giering, R., Kaminski T. and Slawig, T. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. Future Gener. Comp. Sy., 21(8), 2005, p. 1345–1355.

[19] Utke J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill C., and Wunsch, C. Open AD/F: A modular, open-source tool for automatic differentiation of Fortran codes. ACM T. Math. Software (TOMS), 34(4), 2008, p.18:1-18:36.

[20] Hascoët L., Pascual, V. The Tapenade Automatic Differentiation tool: Principles, Model and Specification. ACM Trans. Math. Soft., **39(3),** 2013.

[21] Naumann U., and Riehme, J.: A differentiation-enabled Fortran 95 compiler. ACM T. Math. Software (TOMS), **31(4)**, 2005, 458–474.

[22] Hogan, R. J. Fast reverse-mode automatic differentiation using expression templates in C++. ACM T. Math. Software (TOMS), 2014. available only online at http://www.met.reading.ac.uk/clouds/publications/adept.pdf.

[23] Abarbanel, H. D. I, Kostik, M. and Whartenby, W. Data assimilation with regularized nonlinear instabilities. Q. J. R. Meteorol. Soc., **136**, 2010, p. 769–783.

[24] Lorenz, E. N.. Deterministic nonperiodic flow. J. Atm. Sci., 20(2), 1963, p. 130–141.

[25] Losch, M., and Wunsch, C.: Bottom Topography as a Control Variable in an Ocean Model. J.Atm. and Ocean. Tech., 20, 2003, p. 1685-1696.

[26] Hascoët. Adjoints by Automatic Differentiation. Advanced Data Assimilation for Geosciences (Blayo, E. and Bocquet, M. and Cosme, E., Eds.), Oxford University Press, 2012, p362—364 .

[27] Hascoët L, Naumann U., Pascual V. TBR Analysis in Reverse-Mode Automatic Differentiation. Future Gener. Comp. Sy. 2005. **21(8)**, pp. 1401--1417.

[28] Hascoët, L., Automatic differentiation by program transformation. Technical report, INRIA, 2007. https://www-sop.inria.fr/tropics/papers/supportCoursDA.pdf

[29] Ripodas, P., Gassmann, A., Förstner, J., Majewski, D., Giorgetta, M., Korn, P., Kornblueh, L., Wan, H., Zängl, G., Bonaventura, L., and T. Heinze. Icosahedral Shallow Water Model (ICOSWM): results of shallow water test cases and sensitivity to model parameters. Geosci. Model Dev., **2**, 2009, p.231–251.

Table 1. Comparison of computational speed performance for TAF OpenAD and NAGWare.

| | | **OPEN_AD** | **NAGWare** | **TAF** | **TAPENADE** |
|---|---|---|---|---|---|
| **Linear model** | | | | | |
| | CSP | 8.69 | 84.03 | 2.5 | 3.28 |
| **RLM code** | | | | | |
| | CSP | 5.88 | 80.5 | 4 | 3.25 |
| **SWM code** | | | | | |
| | CSP | 21.27 | 112.3 | 5.95 | 7.2 |

Table 2. Memory Usage efficiency.

| | | **OPEN_AD** | **NAGWare** | **TAF** | **Tapenade** |
|---|---|---|---|---|---|
| **Linear model** | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| | MUE | 1 | 70 | 70 | 70 |
| **RLM code** | | | | | |
| | MUE | 1. | 9.4 | 7.08 | 7.5 |
| **SWM code** | | | | | |
| | MUE | 1. | 5.5 | 4.2 | 17.3 |

Table 3. Ability in handling FORTRAN 90-95 elements

| 1 | | **OPEN_AD** | **NAGWare** | **TAF** | **Tapenade** |
|---|---|---|---|---|---|
| **Pointers** | | | | | |
| 2 | Pointers on active values | No | Yes | No | No |
| 3 | Alias pointers | No | Yes | No | No |
| 4 | Pointers on passive values | Yes | Yes | Yes | Yes |
| **Structures** | | | | | |
| 5 | Structures containing active values | No | Yes | No | No |
| 6 | Structures containing only passive values | Yes | Yes | Yes | Yes |