ECCOMAS Congress 2016
VII European Congress on Computational Methods in Applied Sciences and Engineering
M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. Plevris (eds.)
Crete Island, Greece, 5–10 June 2016

A MIXED OPERATOR OVERLOADING AND SOURCE TRANSFORMATION APPROACH FOR ADJOINT CFD COMPUTATION

Zahrasadat Dastouri[1], **Sinan M. Gezgin**[1], **Uwe Naumann**[1]

[1]LuFG Informatik 12
Software and Tools for Computational Engineering (STCE)
RWTH Aachen University, DE 52056 Aachen
(dastouri,gezgin)@stce.rwth-aachen.de

Keywords: Algorithmic Differentiation, Computational Fluid Dynamics, Operator Overloading, Source Transformation

Abstract. *Adjoint based calculation of sensitivities pertaining to a Computational Fluid Dynamics (CFD) Solver has proven to be a vital tool in tackling large scale problems often found in industrial applications. There are two basic approaches for applying Algorithmic Differentiation (AD) to a CFD solver, namely, Operator Overloading and Source Transformation. Unfortunately, in both cases, it is still necessary to invest a significant amount of manual coding in order to get to an application that performs acceptably in terms of memory consumption and runtime. In this paper reverse mode of AD has been applied to an unstructured pressure-based steady Navier-Stokes solver. We explore the feasibility of combining both kinds of AD approaches to show where and how the advantages of each method can be exploited in order to reach a suitable compromise between performance, simplicity and efficiency. Additionally, we propose a methodology to automate the implementation of an adjoint software in order to minimize work the developer must carry out to produce the desired derivative. We investigate the effectiveness of this approach for relevant flow test cases.*

1 INTRODUCTION

Algorithmic Differentiation (AD) [1, 2] employs the rules of differential calculus in an algorithmic manner to determine accurate derivatives of a function defined by computer programs. For a given implementation of the flow model $F : \mathbb{R}^n \to \mathbb{R}^m$ over a computational grid, the computer program is developed to simulate the functional dependence of one or more objectives $\mathbf{y} \in \mathbb{R}^m$ on a potentially large number of input variables $\mathbf{x} \in \mathbb{R}^n$ by simulation of:

$$F : \mathbb{R}^n \to \mathbb{R}^m, \quad \mathbf{y} = F(\mathbf{x}) \tag{1}$$

AD enables us to compute the corresponding derivatives $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ in forward (forward mode) or backward (reverse mode). For a given implementation of the primal function in Equation 1, the function $F_{(1)} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n \times \mathbb{R}^m$, defined as

$$(\mathbf{y}, \mathbf{x}_{(1)}) = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) \equiv \left(F(\mathbf{x}), \left(\frac{d\mathbf{y}}{d\mathbf{x}} \right)^T \cdot \mathbf{y}_{(1)} \right) \tag{2}$$

is referred to as the *adjoint* model of F. The adjoint model implementation yields the objective $\mathbf{y}$ and the product $\mathbf{x}_{(1)} \equiv \nabla F(\mathbf{x})^T \cdot \mathbf{y}_{(1)}$ of its transposed jacobian at the current point $\mathbf{x} \in \mathbb{R}^n$.

There are two main methods for implementing AD: by source code transformation (S-T) or by use of derived data types and operator overloading (O-O). In O-O AD the code segments and arguments of the primal code are stored inside a memory structure called tape during the forward run of the primal. In reverse mode the stored values on the tape are interpreted to get the resulting adjoints , while the S-T approach parses the code at compile time and generates the actual derivative code.

Prior to this paper [3], [4], we have discussed the implementation of the AD tool `dco/fortran`[1] *(Derivative Code by Overloading in Fortran)* to an unstructured pressure-based steady Navier-Stokes solver. The solver is an incompressible flow solver with cell-centered storage and face-based residual assembly. It works on unstructured meshes with collocated variables and uses the SIMPLE [5] pressure correction algorithm in a pseudo time stepping scheme towards a steady solution. It is written in Fortran 90-95 (7,000 lines) as a test-bed for developing adjoint Navier-Stokes fields [6]. The case study that is used for flow model simulation and sensitivity studies is the S-bend channel flow case which is a simplified vehicle climatisation duct that is presented in [7]. We have addressed proper solution algorithms adapted to the code for the improvement of efficiency of the adjoint code by optimizing the checkpointing scheme for the iterative solver and development of symbolically differentiated of linear solver. In addition, we investigated the benefit of the reverse accumulation technique [8] for the fixed point iterative construction in the primal code.

In this paper we combine the flexibility and robustness of operator overloading with the efficiency of source transformation by coupling `dco/fortran` and `TAPENADE` [9]. The latter is used for the derivation of computationally expensive kernels. Our emphasis is to automate the implementation of an adjoint software to decrease the development time of the differentiated code. The overall design is presented in Sections 3 and 4 illustrating the transformation procedure. Numerical results demonstrate the computational efficiency in terms of memory

[1] developed at the institute a*Software and Tools for Computational Engineering* at RWTH Aachen University implementing AD by overloading in Fortran [1]

consumption and speed while preserving the flexibility of an O-O approach applied to a CFD code.

2 REVIEW OF AD TOOLS

Algorithmic Differentiation is generally implemented using one of two strategies: source code transformation or operator overloading. Here we present a brief review of both tools.

2.1 Operator overloading

Using AD by operator overloading to evaluate directional derivatives of an existing program is comparatively easy. The main idea is to replace all relevant floating point data types within a given program with corresponding enhanced types of the particular operator overloading tool. The enhanced data types then keep track of any *tangent*- or *adjoint*-information of the specific variables declared as floating point data types throughout the forward execution of the program. This is accomplished by defining special operators for all intrinsic functions like +, -, sin(), exp(), etc. , hence the name *operator overloading*. In case of a *tangent-linear* model derivative information is acquired alongside the forward execution of the program. Using the *adjoint* model this is not the case. When an intrinsic function is called during the forward execution of the program, the required information to compute the adjoint information of all involved active(i.e. tracked) variables does not yet exists. To show that, consider the assignment $y = x_1 \cdot x_2$. The adjoints for both inputs are $x_{(1),1} += y_{(1)} \cdot x_2$ and $x_{(1),2} += y_{(1)} \cdot x_1$ respectively. Unfortunately, the value for $y_{(1)}$ is not yet available when the assignment is executed. Therefore, the derivative information cannot be computed at that point in time. A two step approach to solve this problem first stores all required information for every operation during the forward execution of the program. The second step is running through the operations in reverse order and calculating adjoint values of its inputs (x_1, x_2) from adjoint values of its outputs $y_{(1)}$ which are now available. Storing of such information is usually realized by creating log entries on a stack also referred to as tape. In case of `dco/fortran`, an entry in such a tape stores a code for the type of operation, virtual addresses of operands x_1 and x_2 as well as floating point values for its value y and adjoint $y_{(1)}$. This method automatically fulfils requirements of *data-flow* and *control-flow-reversal* for an adjoint model without any extra effort by the tool or the developer. Understandably, such an approach may require a very large amount of available memory in order to store the entire tape for programs of even small to moderate complexity. Its big advantage is its ease of implementation. There are different O-O AD tools for Fortran programs such as ADF [10], ADOL-F [11], AUTO_DERIV [12], etc.

2.2 Source transformation

AD by source transformation analyses the source code of a particular function and produces a new source code which, when executed, calculates the original function value and its directional derivative. The source code for a function is replaced by an automatically generated source code that includes statements for calculating the derivatives interleaved with the original instructions. The advantage is that the resulting code can be compiled into an efficient program due to compile time optimizations having a greater effect. This is a valid approach for all programming languages. The drawback is that this is an enormous transformation that usually cannot be done by hand. Tools are needed to perform this transformation correctly and rapidly. For a *tangent-linear* model, the transformation is straight forward because the derivative statements are in the same order as the original statements. The adjoint model requires control flow reversal which

turns it into a challenging problem for S-T tool developers. Tapenade [9] engine is just one such Automatic Differentiation tool that uses source transformation.Other examples of S-T tools that provide automatic differentiation for Fortran programs are ADG [13], ADFOR [14], TAF [15], OPENAD [16].
The relative advantages and dis- advantages of using the two different tools are summarized in Table 1. The + sign indicates that the corresponding AD tool possesses the desired characteristic.

AD tool	O−O	S−T
Ease of implementation	+	−
Changing the original source code	+	−
Memory consumption	−	+
Compile time optimizations	−	+
Compatibility of numerical gradients with the discrete PDE	+	+
Flexibility to handle arbitrary functions	+	−
Robustness	+	−

Table 1: Comparison operator overloading and source transformation AD tools

3 MIXED APPROACH: SOURCE TRANSFORMATION VIA OPERATOR OVERLOADING TOOL

We apply S-T via O-O tool using a feature of `dco/fortran` called External Function. This interface provide adjoints of specific functions of a program to `dco/fortran` or `TAPENADE`. It is typically used to provide analytically derived adjoints for computationally intense kernels. Instead of recording the operations of such a kernel during the forward run a function pointer is registered on tape. It points to a function capable of providing the adjoint information of the kernel. Basically, if the adjoint model of a function f is known, this information can be used during the interpretation in order to remove the necessity to record f during the forward execution of the program. Depending on the complexity of f, this may have a drastic impact on memory consumption. The following small example shows how this feature is used in order to mix different approaches to adjoint computation.
Consider a function $\mathbf{z} = g(\mathbf{x}, p)$ with $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{R}^m$ which at any point during its execution calls $\mathbf{y} = f(\mathbf{x})$ (1). Let $f_{(1)}$, described in (2), be its known first order adjoint model and let P be a program that computes the adjoint $\mathbf{z}_{(1)}$ of g using `dco/fortran`. Including $f_{(1)}$ into P takes two function implementations. First, a function $f\_wrap(\mathbf{x})$ that hides the process described in figure 1 denoted by *X-Forward* from the rest of P. Second, a function f_ext which includes the steps *X-Reverse* and calls the desired implementation of $f_{(1)}$ in between steps 3 and 4.

1-Forward The state of every non-constant variable which is needed to evaluate f has to be saved. In addition, independent inputs for f are registered using the checkpointing features of `dco/fortran` in order to have the correct variable addresses available later in step 4.

2-Forward The output of f is saved and the implemented function $f_{(1)}$ is registered as the external function that is later called by the interpretation process to evaluate adjoints $\mathbf{x}_{(1)}$.

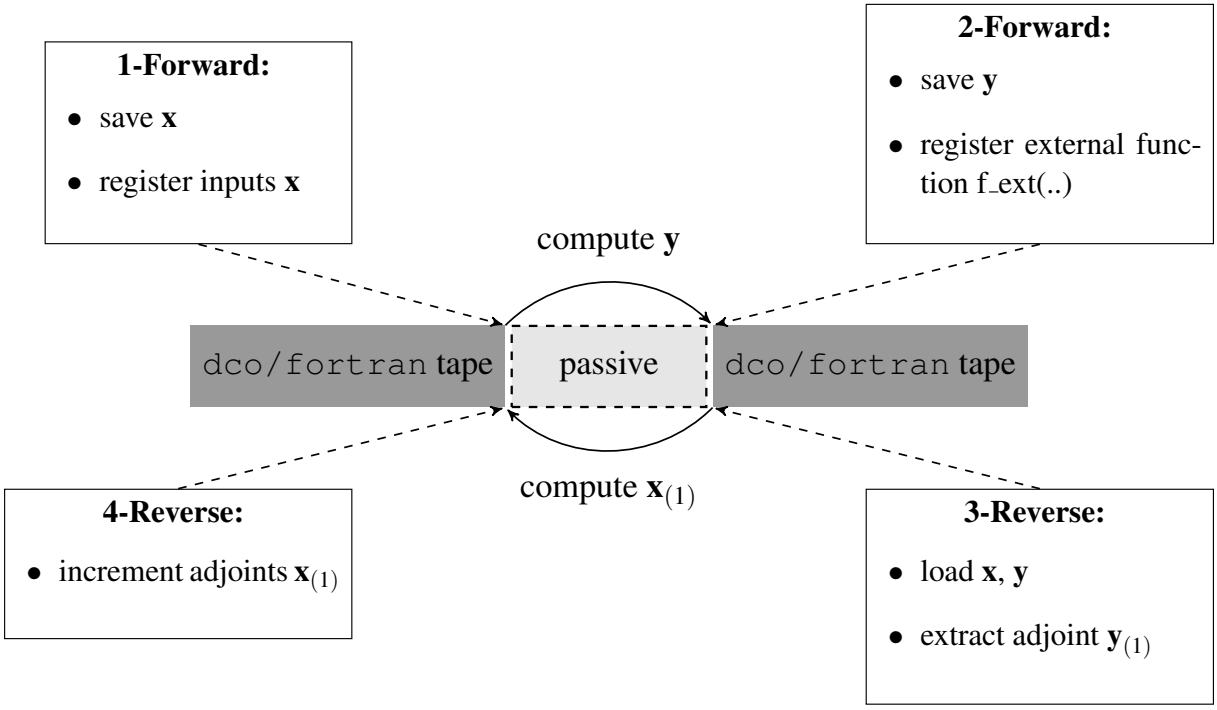

Figure 1: External Function Concept

3-Reverse The stored data is loaded and the adjoint $y_{(1)}$ is read from tape.

4-Reverse The computed adjoint information has to be added back into the tape at their corresponding positions. This is done by incrementing adjoint values of all registered variables in reverse order.

4 AUTOMATION METHODOLOGY

The idea of the automation process described here assumes that there already exists an overloaded program derived by using `dco/fortran`. This section lists steps that need to be taken in order to produce a combined (`dco/fortran` and `TAPENADE`) solution. The overview of the method is presented in figure 2. In addition, a basic approach for automatic source code generation of wrapper- and external functions is presented in 4.2.1 and 4.2.2 respectively. This section uses some abbreviations for the different versions of a function:

f	:	The original *overloaded* function of the black-box `dco/fortran` version
$f\_passive$	:	The original function of the original program
f_b	:	The adjoint function derived by `TAPENADE`
f_wrap	:	The wrapper function corresponding to f
f_ext	:	The function called by the interpreter to compute $f_{(1)}$

Table 2: Function abbreviations

4.1 Kernel identification

The first and vital step is to choose the correct functions for extraction. Using a profiling tool like gprof[17] or intel's vtune[18] is more than sufficient to provide a list of computationally

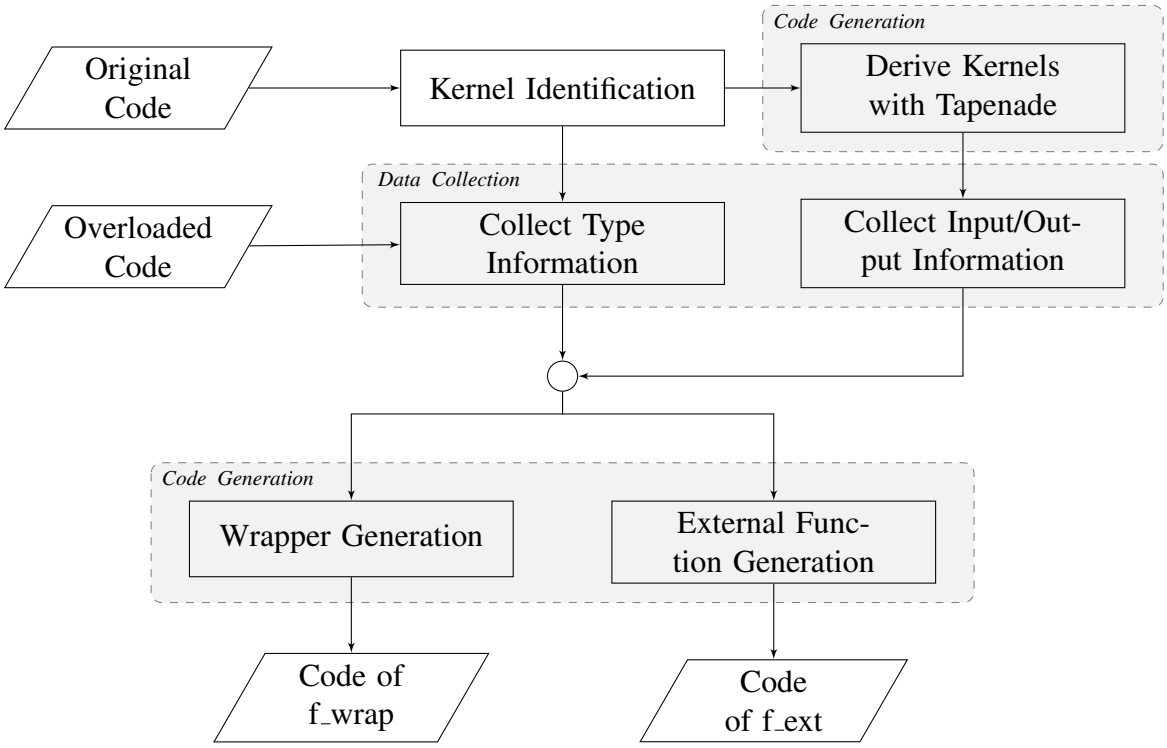

Figure 2: Automation of Source Code Generation

expensive functions. Another important information is how often those have been called during one execution of the original program. The reason for collecting that particular information is that using the external function feature involves storing/copying of data. This overhead may negate any benefit when the computational effort per function call is too small. Thus, the optimal candidates for extraction are functions with a significant self time and minimal number of calls. In addition, it is beneficial if such a function is low in the hierarchical calling order so as to avoid passing the entire program on to `TAPENADE`. Once such functions are identified they can be passed on to `TAPENADE` in order to generate the corresponding overloaded versions.

4.2 Code generation

This sections describes how the development of wrapper- and external functions can be automated. The focus is on source code generation. Not considered is the automated integration of newly created functions into a build system due to the variety in which such a system can be realized.

4.2.1 Wrapper generation

A Wrapper function *f_wrap* of an arbitrary function f has a fixed structure outlined in Algorithm 1. Due to this fixed structure, it is possible to define the complete source code of *f_wrap* using only a few additional informations about the arguments of f. Once all required information is collected a Python preprocessor could generate wrappers, register routines and replace the code. The source code is determined knowing the following information about each variable appearing in the argument list of the function f:

1. Input/Output/Parameter

2. Array Dimension(s)

3. Active/Basic/Derived Type

An active type is a specific derived type, namely `dco_a1s_type`. Knowing the input/output status of an active variable defines whether it needs to be registered as such. It also determines when its value has to be saved. Inputs and parameter values are saved before calling f whereas outputs naturally have to be saved after their evaluation. Array dimensions determine the loop structures encasing calls of functions like *register_input*. All basic types are classified as parameters by default and only their current values need to be stored. A special case occurs when processing derived types. This is due to the fact that a derived type may hold all kinds (input/output/parameter) of variables inside. Consequently, this information needs to be collected for all members of a derived type independently. In addition, a derived type may also have a member which is a derived type itself. A program/script capable of processing arbitrary types therefore has to have a part handling nested derived types. One way to realize such a program is to split it in two parts. The first part collects all required information and fills a data structure similar to Table 4 for each argument and each basic/active member variable of any derived types appearing in the argument list of f. Then, for each step of Algorithm 1 it is checked for each variable if code has to be added to the current section corresponding to the current variable. This can be done by adding references of all *var* structures into three lists, *inputs*, *outputs* and *parameters*, depending on their `var.intent` value.

Algorithm 1: wrapper function generation

Input :
- module dependency information of the overloaded adjoint function $f_{(1)}$
- three lists of var structs: *inputs*, *outputs*, *parameters*

Output:
- source code of function f_wrap

Algorithm:
generate function head
foreach *var* ***in*** *inputs* **do**
| generate block *register input*
end
foreach *var* ***in*** *inputs & parameters* **do**
| generate block *checkpoint*
end
generate call of original function $\mathbf{y} = f\_passive(\mathbf{x}, \mathbf{p})$
foreach *var* ***in*** *outputs* **do**
| generate block *checkpoint*
end
foreach *var* ***in*** *outputs* **do**
| generate block *register output*
end
generate closure statements

4.2.2 External function generation

Like the concept of the wrapper function, the external function also follows a specific sequence of tasks that need to be executed in a fixed order. The sequence is presented in algorithm 2. The information required to generate external function code is equal to that of a wrapper function when making one simplification. Every array is assumed to be *allocatable*, meaning that all data structures are allocated when loading the checkpointed data in *f_ext*. Not having this assumption just means adding a allocatable flag to the *Var* structure and distinguishing the two cases during code generation.

Algorithm 2: external function generation

Input :
- module dependency information of *f_passive*
- three lists of var structs: *inputs*, *outputs*, *parameters*

Output:
- source code of function f_ext

Algorithm:
generate function head
foreach *var* ***in*** *inputs & parameters* **do**
| generate block *read checkpoint*
end
foreach *var* **do**
| generate block *allocate adjoint*
end
foreach *var* ***in*** *outputs* **do**
| generate block *get_input_adjoint*
end
generate call to f_b
foreach *var* ***in*** *inputs* **do**
| generate block *increment_adjoint*
end
generate closure statements

4.3 Data collection

The collection of necessary information can be done in many ways. The method proposed here consists of parsing the output of `gfortran -fdump-parse-tree` and the result of the data flow analysis of `TAPENADE`. The latter is necessary for the identification of in- and outputs of the function where as the former is used to fill data type and shape information. The `TAPENADE` data flow analysis can be found on top of the source code of the derived function. For example, let a function `f_passive(x,y)` implement $y = \sum_{i=1}^{n} x_i^2$. The corresponding analysis of `TAPENADE` produces the following comments on top of the derived function `f_b(x,xb,y,yb)`:

```
!  Differentiation of f_passive in reverse (adjoint) mode (with
   options noISIZE):
```

State	meaning of $var_{(1)}$ for f_b	meaning of *var* for f	code blocks for f_wrap
:zero	output, $var_{(1)} := 0$	parameter	save_in
:incr	input, $var_{(1)}$ incremented	input	rgstr_in, save_in
:in-zero	input, $var_{(1)} := 0$ at end	output	rgstr_out, save_out
:in-out	input & output	input & output	rgstr_in, rgstr_out, save_in
:out	output, $var_{(1)} := 0$ at start	input & output	rgstr_in, save_in

Table 3: Intent states of active variables for f_b

```
!   gradient     of useful results: x y
!   with respect to varying inputs: x y
!   RW status of diff variables: x:incr y:in-zero
```

The input/output status of an argument can be extracted from the *RW status of diff variables* line. Arguments appearing here can have the following states listed in Table 3. Arguments of *f* not appearing in this line are parameters and only need to be stored to ensure correct execution of `f_b`. If any variable appearing in that line is a member of a derived type the corresponding part denoting the *RW status* looks like

```
*(<var>.<member>):<state>.
```

A convenient solution to parse nested types is to construct each `var.name` like the string inside the round brackets. This way, it is easy to produce the Fortran source code accessing said member by just exchanging every "." with a "%".

5 NUMERICAL RESULTS

Numerical results are described for the S-bend test case in three dimensions for different problem sizes ranging from 47k up to 500k cells. Sensitivity analysis results of the adjoint code using operator overloading have been presented in earlier work [4] to calculate the sensitivity of the pressure loss objective function with respect to surface boundary nodes of the S-bend. These results have been obtained for the whole iteration process for the steady solver up to the convergence point of the flow solver. Applying only operator overloading zo the flow solver leads to significant memory usage presented in Figure 3 (-◇- on tape). The performance has been improved by symbolically differentiating the linear solver. However, this is only applicable to problems where the function to be differentiated is simple enough. Nevertheless, it is a very good baseline for performance measurements.

Here we apply the alternative approach presented in this paper by applying source transformation for repetitive parts of the solver where the operator overloading interconnect the upper level. Numerical results are verified and the cost is compared for most expensive repetitive part of the CFD solver.

Var	
name	string
type	string
rank	integer
shape	$[dim_1, dim_2, \ldots, dim_{rank}]$
intent	{'in','out', 'in-out', 'parameter'}

Table 4: Members of Var struct

Table 5: Verification of results with finite difference

version	error
`dco/fortran`	$4.622\mathbf{63870374081}E-008$
`dco/fortran` +`TAPENADE`	$4.622\mathbf{73583679148}E-008$

5.1 Verification

We verify the sensitivity results for adjoint solver with finite difference for all cells as presented in Equation 3. The value of step size increment is determined by smallest relative error between the derivatives of adjoint code and finite difference, see [4]. The error comparison in Table 5 show the accuracy of applying `dco/fortran` and `TAPENADE` tools for generating derivatives.

$$err = \sum_{i=1}^{n} |\, sen_{fd}(i) - sen(i)_{version} \,| \tag{3}$$

5.2 Test environment

The test case is a S-bend ventilation duct for incompressible steady-state flow as presented in [4] . The objective function computes the loss of pressure at the outlet of the channel and the derived program adds computation of sensitivities of surface nodes with respect to the objective function (pressure loss at outlet). All programs are compiled and run on a machine with hardware specifications shown in Table 6.

CPU	Intel(R) Xeon(R) CPU E5-2630
Memory	DDR3 @ 1600 Mhz
OS	Debian GNU/Linux 8 (jessie)
Fortan Compiler	GNU Fortran (Debian 4.9.2-10) 4.9.2
C++ Compiler	gcc (Debian 4.9.2-10) 4.9.2
S-T tool	`TAPENADE` 3.9

Table 6: Machine Specifications

5.3 Overloaded program variants

The process described in section 4 aims to combine the overloaded `dco/fortran` program with kernels derived by `TAPENADE`. The kernel identification exposed the linear solver as, once again, being the function doing most of the computation. Thus, it is chosen to be replaced using the process presented in sections 4.2.1 and 4.2.2. Two program variants are compiled which differ in one option for `TAPENADE`. The option is *-nocheckpoint* which splits the recording of the variable and control flow stacks from the computation of the adjoint values into two separate functions `f_fwd` and `f_bwd` [9]. This allows a more detailed performance comparison. As a baseline for performance the symbolic differentiation of a linear solver is used and incorporated using the external function feature producing a third program variant.

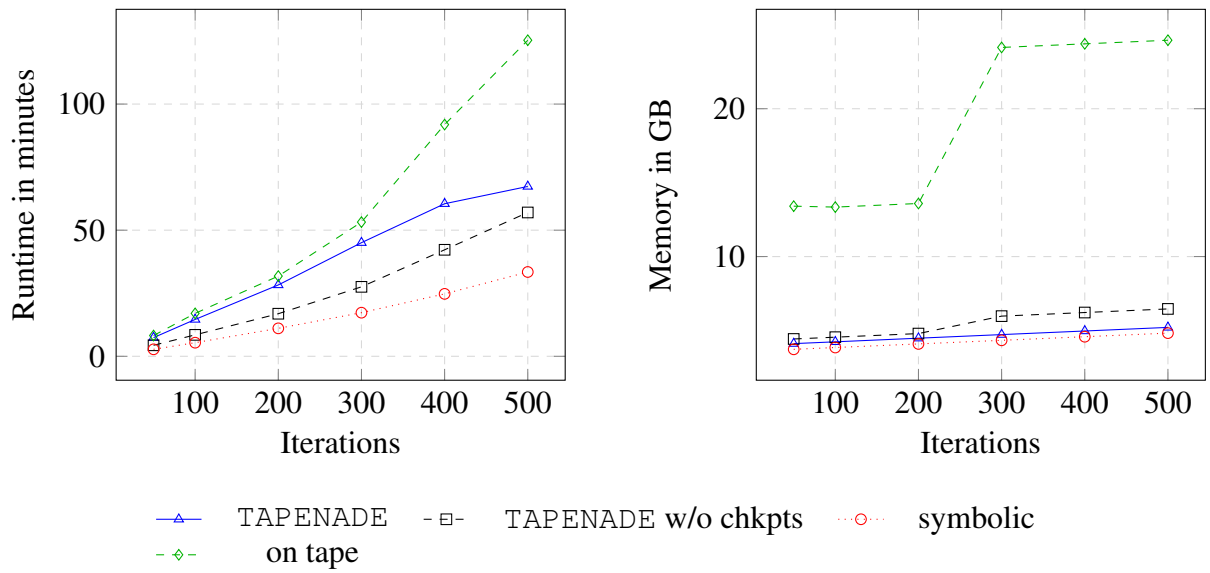

Figure 3: Runtime and Memory for ventilation duct with 47k elements

5.4 Performance

Figure 3,4 and 5 show performance comparison results for the above mentioned test case with 47k, 130k and 500k elements respectively. Runtime and peak memory requirement is measured for computation of sensitivities with a fixed iteration count for the forward computation. The blue solid line is the default variant combining `dco/fortran` and `TAPENADE`. Here, the function derived by `TAPENADE` reruns the forward evaluation of the linear solver during the interpretation of the `dco/fortran` tape. The black, dashed line represents the second combination where temporary data used by `TAPENADE` is acquired during the passive execution of the linear solver, i.e. in between steps 1-Forward and 2-Forward. This saves one linear equation solve during the interpretation of each outer iteration but adds to the peak memory requirement. The theoretical maximum memory requirement for this variant is

$$\text{peak mem} \approx tape + \sum_{i=1}^{n} stack_i \tag{4}$$

where *tape* represents the space needed to store the entire `dco/fortran` tape and $stack_i$ refers to the size of the temporary data of `TAPENADE` generated for calls of the linear solver during the ith outer iteration. The default variant is between two and three times slower than the symbolic variant for all three mesh configurations. As expected, the *-nocheckpoint* version(black, dashed line) is faster than the default variant but requires more space. In particular, for smaller test cases(47k) the peak memory depends more on the iteration count of the linear solver as shown in figure 3. Here, the number of iterations needed to meet a certain tolerance of the linear solve about triples after roughly 250 outer iterations. That is the reason for the sudden jump of the peak memory for the variant combining `dco/fortran` and `TAPENADE` without checkpoints. A similiar but much more pronounced, jump can be observed for the O-O only case. The configuration with 130k elements has a constant high number of iterations of the linear solver and thus does not show a similar jump.

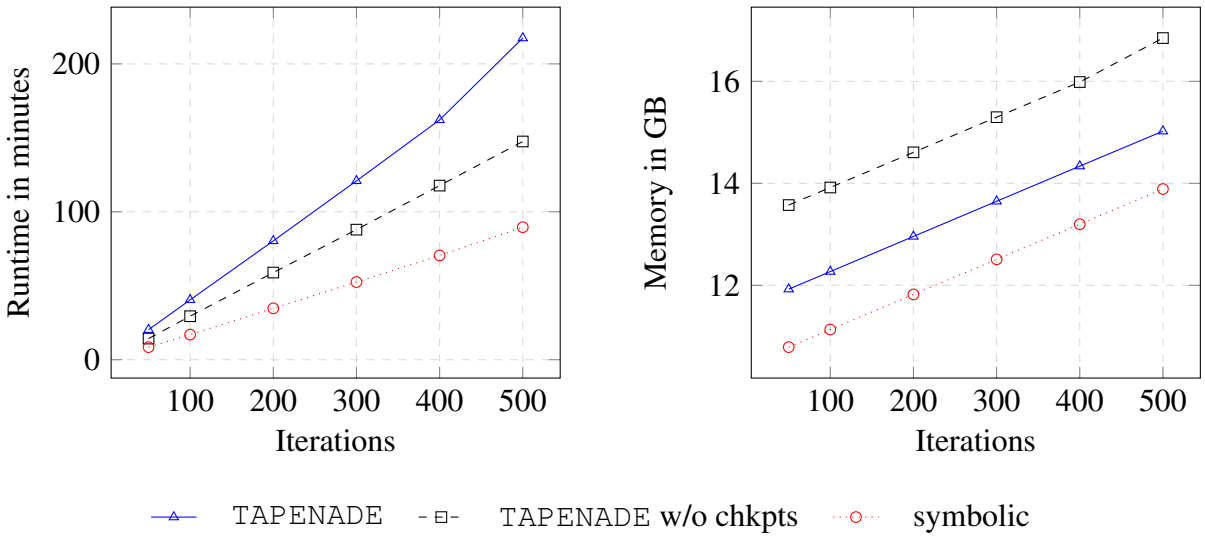

Figure 4: Runtime and Memory for ventilation duct with 130k elements

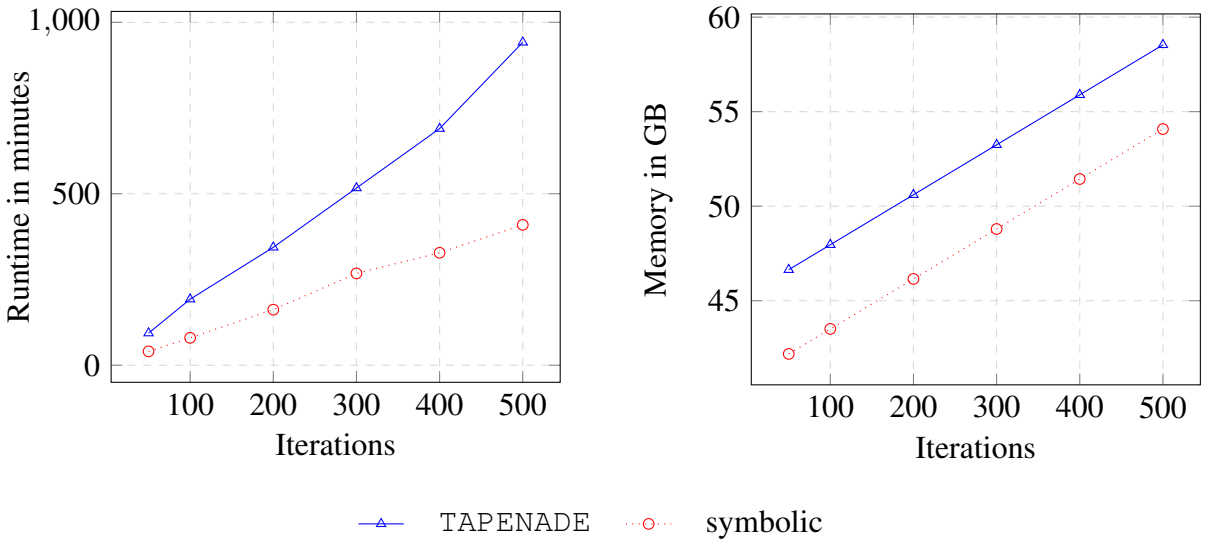

Figure 5: Runtime and Memory for ventilation duct with 500k elements

6 CONCLUSIONS

The method described in this paper employs a source transformation via operator overloading approach such that the resulting derivative code computes the sensitivity results of the given problem in CFD solver. The key aspect of this method is a combination of source-to-source transformation and operator overloading approaches. The numerical experiments are tested for expensive repetitive part of the solver demonstrating significant memory reduction of the resulting combined code compared to the overloaded adjoint version. In terms of memory usage, these results are competitive with symbolic differentiation of a given function. It could be used as a lower bound to decide whether developing a symbolic adjoint is viable. Moreover, the time needed to computer the derivatives speed up by applying the combined approach which is clearly superior for higher number of iteration. The overall design of a generic transformation methodology is presented that is applicable for complicated computer programs.

ACKNOWLEDGEMENT

The presented research is supported by the project **aboutFlow**, funded by the European Commission under grant no.FP7-PEOPLE-2012-ITN-317006.

References

[1] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. SIAM, Philadelphia, 2012.

[2] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, Jan 2000.

[3] Z. Dastouri, J. Lotz, and U. Naumann. Development of a discrete adjoint CFD code using Algorithmic Differentiation by Operator Overloading. In *OPT-i: An International Conference on Engineering and Applied Sciences Optimization*, Athens, 2014. National Technical University of Athens.

[4] Z. Dastouri and U. Naumann. Improving efficiency of a discrete adjoint CFD code for design optimization problems. In *EUROGEN: Proceeding of International Conference on Evolutionary and Deterministic Methods for Design, Optimization and Control with Applications to Industrial and Societal Problems*, Glasgow, UK, 2015.

[5] S.V. Patankar and D.B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal for Heat Mass Transfer*, 15(10):1787–1806, 1972.

[6] D. Jones, F. Christakopoulos, and J-D. Mller. Preparation and assembly of adjoint CFD codes. *Computers and Fluids*, 46(1):282286, July 2011.

[7] C. Othmer and T. Grahs. Approaches to fluid dynamic optimization in the car development process. In R. Schilling et. al., editor, *Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*, Munich, 2005. FLM.

[8] B. Christianson. Reverse Accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326, 1994.

[9] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013.

[10] C. W. Straka. Adf95: Tool for automatic differentiation of a fortran code designed for large numbers of independent variables. *Computer Physics Communications*, 168(2):123–139, 2005.

[11] D. Shiriaev, A. Griewank, and J. Utke. A user guide to ADOL–F: Automatic Differentiation of Fortran codes. *Tech. Report*, (IOKOMO–04–1995), 1996.

[12] S. Stamatiadis, R. Prosmiti, and S. C. Farantos. AUTO_DERIV: Tool for Automatic Differentiation of a FORTRAN code. *Computer Physics Communications*, 127(2&3):343–355, 2000.

[13] C. Qiang, Z. Linbo, and W. Bin. Model adjointization and its cost. *Science in China, Series F–Information Sciences*, 47(5):587–611, 2004.

[14] C. Bischof, A. Carle, G. Corliss, A Griewank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[15] R. Giering and T Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24(4):437–474, 1998.

[16] U. Naumann, U. Utke, C. Wunsch, C. Hill, P. Heimbach, M. Fagan, N. Tallent, and M. Strout. Adjoint code by source transformation with OpenAD/F. 2006. Available online at `http://proceedings.fyper.com/eccomascfd2006/documents/35.pdf`.

[17] S. Graham, P. Kessler, and M. Mckusick. Gprof: A call graph execution profile. In *SIGPLAN '82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction,*, number ISBN:0-89791-074-5, pages 120–126, ACM New York, NY, USA, 1982.

[18] J. Reinders. Vtune performance analyzer essentials measurement and tuning techniques for software developers. In *Intel Press,*, Hillsboro, 2005.