Efficiency of second-order differentiation schemes by algorithmic differentiation: Case Study

Thorsten Lajewski

Supervisor: Johannes Lotz
STCE, RWTH Aachen

May 6, 2013

1 Introduction

In numerical calculations derivatives are often needed. Algorithmic differentiation (**AD**) provides two different ways to compute the derivatives of a function up to machine accuracy. The forward and reverse mode. Since higher order derivatives can be understood as first order derivatives of an already calculated derivative it is obvious that using AD they can be gained by applying one mode on top of another. For the second order derivatives this leads to four different possibilities, the forward-over-forward, the forward-over-reverse, the reverse-over-forward and the reverse-over-reverse mode. While the reverse-over-reverse mode is not relevant in real applications (see [2] page 245) the other modes are.

While in theory the needed computation time and memory of the 2nd-order methods forward-over-reverse and reverse-over-forward should be the same, in reality the costs will vary depending on the used mode, the given problem and the implementation. In this paper the efficiency of this two different 2nd order methods of the AD-tool `dco/c++` applied to a simple heat flow problem will be examined.

2 Algorithmic Differentiation

In this section a very short introduction to algorithmic differentiation (AD) will be given, further more in depth information can be found in [2] and [1]. In mathematics the differentiation of a function can be calculated by applying simple rules like the product rule. In computer programs differentiating functions may not be as simple. This becomes clear when thinking about often used expressions like if , switch cases or loops. That is why in programs often the finite difference method (FD) is used. Equation 1 shows the central finite difference method applied to a general multidimensional function $f(\vec{x})$. The vector $\vec{e}_i$ denotes the unit vector, with a one int the i-th dimension and zero everywhere else.

$$\frac{df}{d\vec{e}_i}(\vec{x}) = \frac{f(\vec{x} + h\vec{e}_i) - f(\vec{x} + h\vec{e}_i)}{2h} \qquad (1)$$

This method has the disadvantage that it only calculates an approximation of the real derivative and the error depends on the choice of the parameter h, $|f(x)|$ and $|x|$. The finite difference method can be understood as the approximation of the tanget-linear model known from mathematics. To calculate the whole derivative the finite difference method has to be evaluated once for each input dimension. Because each evaluation consists of two calls of the function, the total cost for the calculation sums up to about twice the number of inputs times the cost of one function evaluation. AD is able to calculate the same derivative up to machine accuracy at the same cost like FD. When the function, which needs to be differentiated fulfills some conditions, the number of outputs is significantly higher than the number of inputs, AD can even reduce the calculation costs significantly further.

There are two different AD modes. The forward and the reverse mode. Both modes are supported by available AD tools. These tools use either operator-overloading or source to source compilers. Both AD modes will be explained in the next sections.

2.1 Single Assignment Code / Visualisation as a directed acyclic graph

A good way to understand, which variables need to be stored during the computation of the derivative, is to visualise the computation as a directed acyclic graph. The source code of a function can be expanded to a so called Single Assignment Code (SAC). This means, that every line of code only consists of one assignment and one mathematical operation. As Example Listing 1 illustrates the way a function is transformed into SAC code Listing 2. Source to source compiler use this single assignment code to generate AD code.

Listing 1: A simple example Function

```
void f(double& z, double x,
       double y){
    z = sin(x)+cos(x*y);
}
```

Listing 2: SAC code of the example

```
void f(double& z, double x,
       double y){
    int v1 = sin(x);
    int v2 = x*y;
    int v3 = cos(v2);
    z = v1 + v3;
}
```

The SAC code can also be visualised as directed acyclic graph. Every input, intermediate and output variable is a node. The The edges of the graph describe the dependencies of the variables. The values on the edges are then set as the partial derivative of the variable with respect to the variable it depends on. Figure 1 shows the graph of the example. The derivative of the result of the function with respect to the input variable can then be calculated by multiplying along all paths and summing the different paths up to the final result. Since the multiplication is commutative, it can be seen that it does not matter if the multiplication order is from the inputs to the outputs or the other way round. This observation is used by the reverse mode.

The forward mode AD can be visualised by adding an auxiliary variable t to the graph. The input variables of the func-

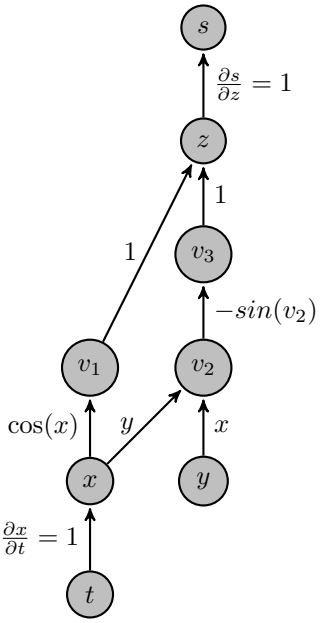

Figure 1: Graph of the example of Listing 2, variable t for tagential expansion, variable s for adjoint expansion

tion depend on this auxiliary variable. The derivative value on the edge defines the direction of differentiation. In the reverse mode another auxiliary variable s is added to the graph. In this mode this auxiliary variable depends on the output parameters and the value also defines the direction of the derivation.

2.2 Forward Mode

The forward mode AD of a function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m, \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$ is defined as

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) \tag{2}$$

$$\mathbf{y}^{(1)} = \nabla\mathbf{F} \cdot \mathbf{x}^{(\mathbf{1})}, \mathbf{y}^{(1)} \in \mathbb{R}^m, \mathbf{x}^{(1)} \in \mathbb{R}^n \tag{3}$$

The derivative of the function is calculated by inserting $\mathbf{x}^{(1)} = \mathbf{e}_i$ as directional derivative, where $\mathbf{e}_i$ is the i-th unit vector in $\mathbb{R}^n$. Therefore, n evaluations of the function are needed to calculate the complete Jacobian matrix. The total costs to calculate the derivative in forward mode AD are comparable to the amount needed for a finite difference method.

2.3 Reverse Mode

The reverse mode AD of a function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m, \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$ is defined as

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) \tag{4}$$

$$\mathbf{x}_{(1)} = \nabla\mathbf{F}^T \cdot \mathbf{y}_{(1)}, \mathbf{x}_{(1)} \in \mathbb{R}^n, \mathbf{y}_{(1)} \in \mathbb{R}^m \tag{5}$$

The derivative of the function is calculated by inserting $\mathbf{y}_{(1)} = \mathbf{e}_j$, where $\mathbf{e}_j$ is the j-th unit vector in $\mathbb{R}^m$. Therefore, m evaluations of the function are needed to calculate the whole Jacobian matrix.

When using the reverse mode values needed to calculate the derivatives are not available in the right order, see figure 1. That's why a implementation of the reverse mode consists of two phases. First in the forward run, the original function is run with the only modification that all intermediate values are stored in a datastructure called tape. This tape can be understood as the directed acyclic tree of the program. In the following reverse run the values inside the tape are used calculate the derivatives. The reverse mode is only efficient if the number of outputs is significantly higher than the number of inputs, because of the overhead work needed to record the tape.

2.4 2nd order adjoints

The second order derivatives can be calculated by applying one of the AD modes

to another AD mode. Since there are two different modes of differentiating a function this leads to four different 2nd-order modes. All four will be introduced in the following section. In the sections after that only the two modes Forward-over-reverse and Reverse-over-forward will be discussed, because the forward-over-forward mode can not be applied to the problem, because the derivative of the program is calculated using a adoint mode. The reverse-over-reverse mode will not be compared, because of the dataflow needs to be reversed twice during the computation. Therefore this method has a very bad performance and is of no use in real problems. Nevertheless, in the next section the mathematical calculations needed for all four possible second order AD modes will be shown, a detailed description can be found in [2].

2.4.1 Forward over Forward Mode

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) \tag{6}$$

$$\mathbf{y}^{(2)} =< \nabla\mathbf{F}(\mathbf{x}), \mathbf{x}^{(2)} > \tag{7}$$

$$\mathbf{y}^{(1)} =< \nabla\mathbf{F}(\mathbf{x}), \mathbf{x}^{(1)} > \tag{8}$$

$$\begin{aligned}\mathbf{y}^{(1,2)} = &< \nabla\mathbf{F}(\mathbf{x}), \mathbf{x}^{(1,2)} > + \\ &< \nabla^2\mathbf{F}(\mathbf{x}), \mathbf{x}^{(1)}, \mathbf{x}^{(2)} >\end{aligned} \tag{9}$$

2.4.2 Reverse over Reverse Mode

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) \tag{10}$$

$$\mathbf{x}_{(1)} =< \mathbf{y}_{(1)}, \nabla\mathbf{F}(\mathbf{x}) > \tag{11}$$

$$\mathbf{x}_{(2)} = \mathbf{x}_{(2)} + < \mathbf{x}_{(1,2)}, \mathbf{y}_1, \nabla^2\mathbf{F}(\mathbf{x}) > \tag{12}$$

$$\mathbf{y}_{(1,2)} = \mathbf{y}_{(1,2)} + < \mathbf{x}_{(1,2)}, \nabla\mathbf{F}(\mathbf{x}) > \tag{13}$$

2.4.3 Forward over Reverse Mode

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) \tag{14}$$

$$\mathbf{y}^{(2)} =< \nabla\mathbf{F}(\mathbf{x}), \mathbf{x}^{(2)} > \tag{15}$$

$$\mathbf{x}_{(1)} =< \mathbf{y}_{(1)}, \nabla\mathbf{F}(\mathbf{x}) > \tag{16}$$

$$\begin{aligned}\mathbf{x}^{(2)}_{(1)} = &< \mathbf{y}^{(2)}_{(1)}, \nabla\mathbf{F}(\mathbf{x}) > + \\ &< \mathbf{y}_{(1)}, \nabla^2\mathbf{F}(\mathbf{x}), \mathbf{x}^{(2)} >\end{aligned} \tag{17}$$

2.4.4 Reverse over Forward Mode

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) \tag{18}$$

$$\mathbf{y}^{(1)} =< \nabla\mathbf{F}(\mathbf{x}), \mathbf{x}^{(1)} > \tag{19}$$

$$\begin{aligned}\mathbf{x}_{(2)} = &< \mathbf{y}_{(2)}, \nabla\mathbf{F}(\mathbf{x}) > + \\ &< \mathbf{y}^{(1)}_{(2)}, \nabla^2\mathbf{F}(\mathbf{x}), \mathbf{x}^{(1)} >\end{aligned} \tag{20}$$

$$\mathbf{x}^{(1)}_{(2)} =< \mathbf{y}^{(1)}_{(2)}, \nabla\mathbf{F}(\mathbf{x}) > \tag{21}$$

2.4.5 Comparison

The forward-over-forward mode is straight forward and does not need any extra memory. If an adjoint solution is needed one of the other three modes has to be chosen. Because the reverse-over-reverse mode needs to reverse the dataflow twice and is inefficient. The forward-over-reverse mode seem to be very similar to the reverse-over-forward mode in terms of efficiency. The number of operations is similar and both methods only reverse the dataflow once. But in detail there are some slight differences. Because the reverse-over-forward mode applies the reverse mode after the first order code is calculated by the forward mode, not only the intermediate values but also the intermediate values of the first order derivative need to be stored inside the tape, which

leads to a slightly larger memory consumption. But in the examined implementation the reverse-over-forward mode is the one, which is implicitly used, because the program gets differentiated using a first order reverse method. Some extra work is needed to use the forward-over-reverse mode instead. It will be shown in the next sections that this overhead does not outweigh the advantages of the forward-over-reverse mode.

3 Test Case

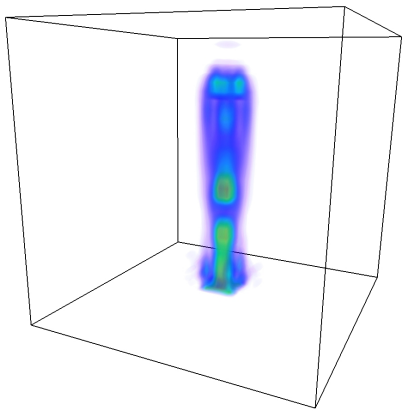

Figure 2: Solution of the given Problem after $t = 23.4sec$

The program evaluated as test case, is a small flow solver, which calculate the air flow over a hot square. The program uses the Boussinesq approximation, which means that the incompressible Navier-Stokes equations are solve for the x- and y- directions and the differences in the density are only considered in the z- direction (the direction of the gravity force $\vec{g}$).

For the time discretisation an implicit euler method is used. The space is discretised using a predictor-corrector scheme. For the prediction equation the pressure is assumed constant and the equations are discretised using a finite difference method. This leads to a nonlinear equation system. This system is solved using the newton method (Listing 3). Inside each Newton iteration a matrix-free implementation of the GMRES solver is used to solve the linear equation system.

Listing 3: Newton's algorithm

```
in f(x), f'(x), eps, x0;
    x = x0;
    currentEps = norm(f(x));
    while(currentEps > eps)
    {
        solve : f'(x)*s = -f(x);
        x = x + s;
        currentEps = norm(f(x));
    }
out x
```

The correction equation is solved using a SOR-Solver.

For example if this simulation program should be used to calculate the temperature of the plate, when the flow field is measured at a certain time, an inverse problem needs to be solved.

To calculate the starting temperature the function

$$J = || \begin{pmatrix} \mathbf{u}^{t^{\text{end}}} \\ \mathbf{T}^{t^{\text{end}}} \end{pmatrix} - \begin{pmatrix} \mathbf{u}_m \\ \mathbf{T}_m \end{pmatrix} ||_2 \qquad (22)$$

has to be minimized. This can be achieved by applying an iteration method like the steepest descent algorithm.

$$T_{i+1}^{t^{\text{begin}}} = T_i^{t^{\text{begin}}} - \nabla_{T^{t^{\text{begin}}}} J \qquad (23)$$

∇J has to be calculated by differentiating the whole simulation program.

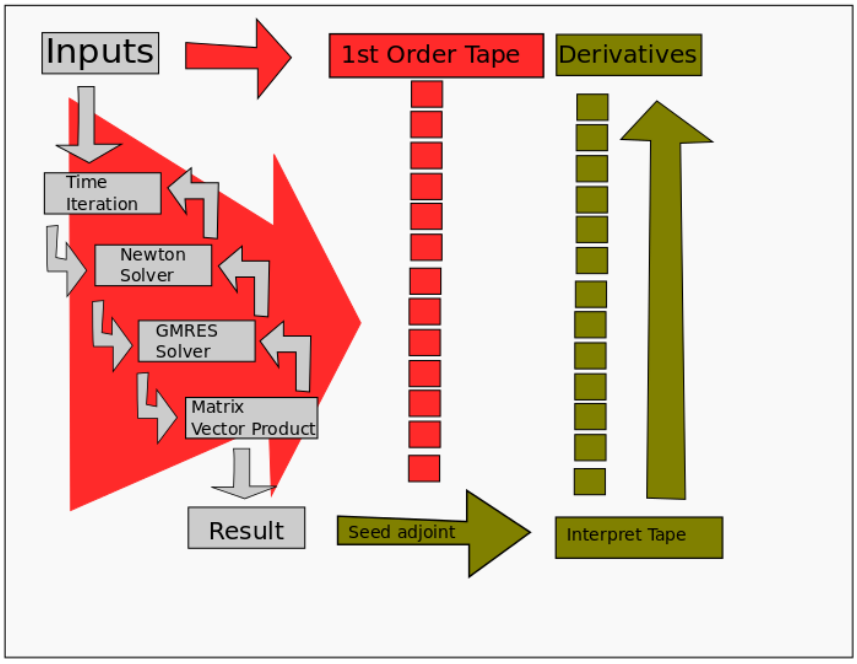

Figure 3: Original implementation of the program, second order derivatives are calculated internally by the reverse-over-forward mode

Due to the fact, that the given problem consists only of one output parameter. The norm J (see equation 22) and the number of inputs depends on the number of cells in the model, which will normally be very large, the program is differentiated using a tool which uses reverse mode AD. Because inside the Newton algorithm a Jacobian matrix is needed, at this point the AD tool calculates a second-order derivative. Since the original program used a tangent-linear implementation this second-order derivative will be calculated using the reverse-over-forward mode. The next sections will show if some advantage can be gained by using the forward-over-reverse mode to calculate the derivatives of this part of the program.

4 Implementation

The original flow solver is implemented in C++. As AD tool `dco/c++` is used, which is being developed at the STCE. A brief introduction to this tool will be given in the next section. The flow solver uses templates to provide the abillity to exchange the datatypes of all variables easily. In functions where special behaviour for certain datatypes is needed, specialisations of the templates are used. For details on the specialisation of templates refer to [4]. The original program was modifed as few as pos-

sible. The only part which was changed was the matrix-vector product, which is called inside the innermost loop of the GMRES-solver. Because the number of calls to this function is quite high, the changes will have a huge impact on the overall efficiency. The comparison of the original program and the modified one will be discussed in the next section.

4.1 `dco/c++`

The AD-tool `dco/c++` is being developed the STCE uses operator-overloading to implement the different AD modes. It provides special datatypes for each mode. dco::a1s:type is used for the adjoint mode and dco::t2s_a1s:type for the second order forward-over-reverse mode. All double inside the original program need to be replaced by this datatypes. Both modes need a tape which needs to be allocated globally. After this all variables have to be registered inside this tape. The program can then be run the same way as without the AD tool. The derivatives can be calculated by inserting values for the adjoint direction and interpreting the tape using the corresponding functions. For a more in-depth description of `dco/c++` see [3].

4.1.1 external functions

`dco/c++` provides a way to exclude some function from being recorded inside the tape. The adjoints for these excluded functions need to be calculate inside special functions, which are defined by the user. This functionality can be needed if parts of the program are library, which are only available compiled and therefore the datatypes cannot be changed. The external functions can also be used to implement improved calculations of the adjoint of a function by hand. A external function will also be used explicitly use the forward-over-reverse mode inside the calculation of the heat-flow problem. To use external functions the original function which should be excluded from recording must be modified. The inputs of the function must be copied into passive variables of type double using a special function of dco/c++, which tells the tape, that the adjoint of the variables will be provided separatly. Here passive variables means, that all operations on them will have no effect on the status of the tape. After this all values, which are needed to calculate the adjoint in the reverse run need to be copied into a special userdata datastructure provided by `dco/c++`. This datastructure and a pointer to the function, which calculates adjoints in the reverse run is stored inside the tape. After the function run the result of the function need to be copied into the active datatype again. Here again a special function of `dco/c++` is used to tell the tape that this value is a output of the function.

Inside the function, which has been registered during the forward run, first of all datavalues stored inside the userdata object need to be restored. Than the adjoints of the output of the original function can be read from the tape. After this the function calculates the adjoints and the adjoints need to be written back to the tape. It is important that the adjoint are written to the tape in the same order the active types are copied into the passive versions inside the forward run.

4.2 call-order

The modified program, which calculates the needed second order derivative using the forward-over-reverse mode uses an external function. Inside the matrix-vector-product function a flag can be used to either call the original function, which results in the use of the reverse-or-forward mode (see figure 3), or use a external function which uses the forward-over-reverse mode of `dco/c++`.

If the external function is used inside the forward run the forward mode code will run using passive datatypes only. Inside the userdata object the whole FlowField containing the vectors $u, v, w, T, u_{old}, v_{old}, w_{old}, T_{old}$ and some other scalar variables is stored. The vectors with the subscript old are the values of the previous iteration and needed for the correct calculation of the function value. In addition to the FlowField the vector of the matrix vector product is stored inside the datastructure.

In the reverse run for each variable of the FlowField one variable of the type dco::t2s_a1s::type is created and the values of the Flowfield are copied into this second order datatype. After these variables are registered inside the second order tape, so that the second order tape is build, when evaluating the residual function. The values of the vector are needed for seeding the tangent linear direction. Because seeding is a crucial part this will be explained in extra section. After the tangent linear direction is seeded, the residual function is evaluated. After this the adjoint direction has to be seeded and the tape interpreted. Finally, the second order derivatives need to be placed into the first order tape. Because it is not this obvious, which derivatives need to be placed inside the first order tape this will also be explained in an extra section.

This implementation of the external function leads to a program flow, which can be seen in figure 4. In the forward run the first order tape is generated with gaps (yellow squares). When the first order tape is interpreted, whenever such a gap is encountered, the corresponding external function gets called, this function generates a second order tape and interprets it. The results of this are used to fill the gap in the first order tape. After this the first order tape is interpreted again till the next gap is encountered or the interpretation is finished.

4.3 Seeding

In order to calculate the second-order derivatives, two directions have to be seeded. Before the recording of the tape, the tangent-linear direction needs to be seeded and before interpreting the tape the adjoint direction. The tangent linear direction needs to be seeded for the flow field variables u, v, w, T. The tangent linear direction $x^{(2)}$ is given by the vector v, which was input of the original matrix-vector-product function. In the reverse run it can be read from the checkpoint. In a matrix vector product of a Jacobian Matrix and a vector, the vector determines the directional derivative, which means that not the whole Jacobian needs to be calculated, but only the function needs to be differentiated in direction of the vector.

The adjoint direction can be read from the tape, and needs to be seeded into to $y_{(1)}$ of result vector of the matrix-vector-product function. The outputs of the original function become inputs of the reverse function.

In case of the matrix-vector-product $y^{(2)}_{(1)}$

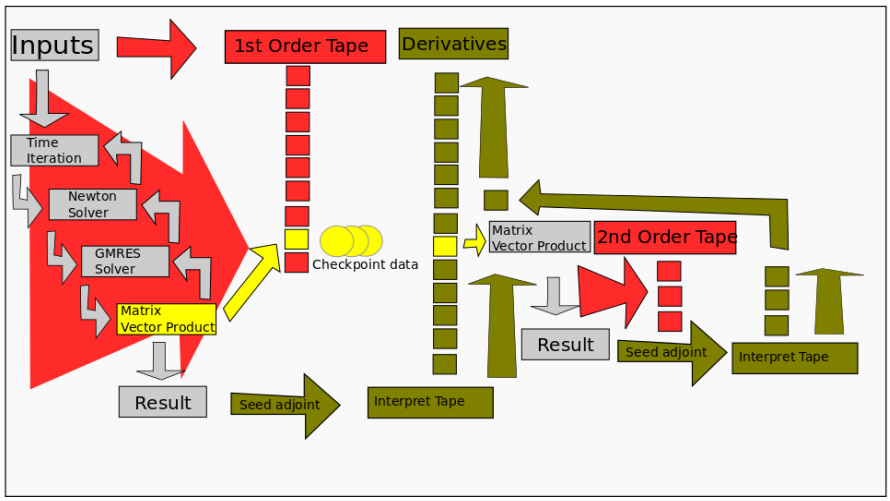

Figure 4: Program run using a external function to calculate the needed second order derivatives using forward-over-reverse mode

is always zero, equation 17 can therefore be simplified to equation 24.

$$\mathbf{x}_{(1)}^{(2)} =< \mathbf{y}_{(1)}, \nabla^2 \mathbf{F}(\mathbf{x}), \mathbf{x}^{(2)} > \tag{24}$$

4.4 Writing the correct adjoints into the tape

In the last step of the implementation of the forward-over-reverse mode external functions, the resulting second order and first order derivatives need to be placed inside the first order tape, which is used for the rest of the program. As a quick reminder, the inputs of the original function, where all flow field variables and the vector of the matrix-vector-product. The function is part of the differentiation of the Newton Solver. This means since in the original program the flow field variables contributed to the calculation of the Jacobian, their derivatives need to be the second-order values calculated inside the second-order tape. Therefore $x_{(1)}^{(2)}$ (where x consists of all values for u, v, w, T) needs to be placed inside $x_{(1)}$ of the first order tape. For the input vector of the matrix-vector-product the values of $x_{(1)}$ of the second order tape need to be seeded inside $x_{(1)}$ of the first order tape.

5 Results

In this last section the results of the new implementation of the external forward-over-reverse function will be compared to the original program which internal used the reverse-over-forward mode. Because in the external function the whole matrix-vector-product function is run again and a lot of memory needs to be allocated it can be expected, that the runtime of the new implementation has become worse. On the other hand the first order tape should become much smaller because the original matrix-

vector-product is not recorded. Also this memory advantage of the first order tape is reduced by the overhead which is needed to store the whole flow field each time the function is called and the size of the second order tape needed for the forward-over-reverse mode calculation. To be able to compare both implementations both are tested for the same input parameters. The solver calculates 15 timesteps with $dT = 0.05$ seconds. The abort condition of the Newton solver is set to $eps_{Newton} = 1e{-}7$, the one of the GMRES solver to $eps_{GMRES} = 1e - 8$ and the one of the SOR Solver to $eps_{SOR} = 1e - 6$. The number of grid points will be increased to show the effect of different numbers of variables. Grid sizes tested, started from 4 grid points in each room dimension up to 11, which led to 81 up to 3700 variables.

5.1 Memory Consumption

The memory consumption of the old implementation in comparison to the forward-over-reverse implementation is no big surprise. While the tape size of both implementations grows drastically when increasing the number of grid points, the implementation of the external forward-over-reverse mode requires always only 20% of the original implementation. While in the original implementation only the first-order tape contributes to the memory consumption, in the new implementation on top of the first order tape, the memory required to store all checkpoints, the memory overhead of the external function and the second-order tape has to be added. The checkpoints consists of all variables, which are needed to set up the residual function call for the second-order tape, in fact for each external function call the whole flow field has to be saved once. On top of the checkpoints inside each external function call a small 2nd order tape is recorded and removed. But altogether these memory requirements compared to the first-order tape are very small.

For the biggest tested problem, the first order tape was 1126.44 MB large, the memory needed to store the 296 needed checkpoint is only 40.17 MB, the overhead of the external function 0.3 MB and the second-order tape is 4.566 MB. Therefore, this memory overhead results only in 3.8% of the overall memory. The comparison of the total memory required can be found in figure 5.

5.2 Runtime

The resulting runtime of the solver is a little bit different than expected. For the small problem sizes $nx = 3, 4$ the results are as expected. The forward run of of the new implementation is shorter, because the call of the matrix-vector-product does not need to be recorded. In contrast the reverse run takes longer, because the matrix-vector-product has to be run forward and reverse and the resulting second-order tape has to be interpreted. The overall runtime of this new implementation is still faster than the original implementation. This can be explained by a property of the programming language c++. In c++ allocating memory takes a long time. The bigger the first order tape gets the more often new memory blocks have to be allocated, for small problems this takes even longer than the actual calculation time. For bigger problems not only the forward-run but also the reverse-run of the new implemented code becomes faster. This is a result of the massive difference in first

order tape size. For big problems the overhead of executing the external function isn't as big as interpreting the resulting first-order tape of the matrix-vector-product of the old function. The runtimes of the forward, the reverse and the complete program run compared the the original implementation can be found in figure 6.

5.3 Conclusion

The implementation of the external forward-over-reverse function has advantages in terms of memory consumption and runtime. The theoretical advantage of the forward-over-reverse mode is bigger than the disadvantage that it needs to be implemented as external function in this test case. Therefore, it is useful to invest the manual work to implement the discussed external function.

References

[1] A. Griewank, G. Corliss, S. for Industrial, and A. Mathematics. *Automatic differentiation of algorithms: theory, implementation, and application*. Society for Industrial and Applied Mathematics Philadelphia, PA, 1991.

[2] U. Naumann. *The Art of Differentiating Computer Programs*. SIAM, 2011.

[3] The Numerical Algorithms Group Ltd. (NAG), Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, United Kingdom. *dco/c++ User Guide*.

[4] D. Vandevoorde and N. Josuttis. *C++ templates: the Complete Guide*. Addison-Wesley Professional, 2003.

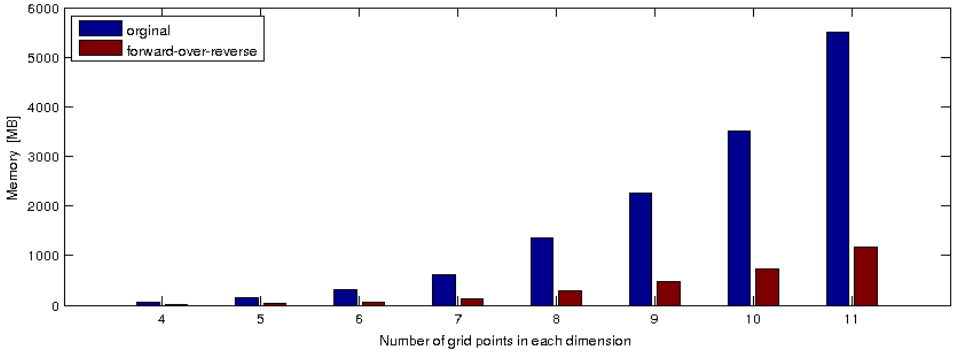

Figure 5: Absolute memory consumption of both implementations

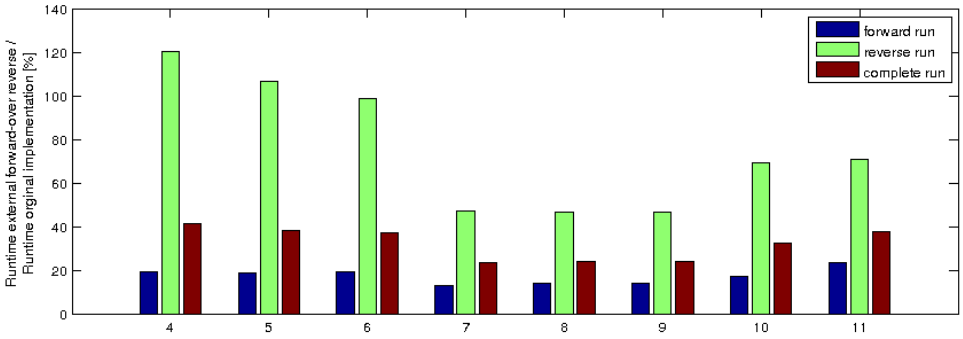

Figure 6: Runtime of the new implementation compared to the original one.