RWTH Aachen

Department of Computer Science
Technical Report

Adjoints for Time-Dependent Optimal Control

Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-19

RWTH Aachen · Department of Computer Science · December 2007

Adjoints for Time-Dependent Optimal Control

Jan Riehme[1], Andrea Walther[2], Jörg Stiller[3], and Uwe Naumann[4]

[1] Department of Computer Science, University of Hertfordshire, UK, riehme@stce.rwth-aachen.de
[2] Department of Mathematics, Technische Universität Dresden, Germany, andrea.walther@tu-dresden.de
[3] Department of Mechanical Engineering, Technische Universität Dresden, Germany, joerg.stiller@tu-dresden.de
[4] Department of Computer Science, RWTH Aachen University, Germany, naumann@stce.rwth-aachen.de

Abstract. The use of discrete adjoints in the context of a hard time-dependent optimal control problem is considered. Gradients required for the steepest descent method are computed by code that is generated automatically by the differentiation-enabled NAGWare Fortran compiler. Single time steps are taped using an overloading approach. The entire evolution is reversed based on an efficient checkpointing schedule that is computed by `revolve`. The feasibility of nonlinear optimization based on discrete adjoints is supported by our numerical results.

1 Background

Controlling and optimizing flow processes is a matter of increasing importance that includes a wide range of applications, such as drag minimization, transition control, noise reduction, and the enhancement of mixing or combustion processes [3, 11]. The intention of the present work is to demonstrate the suitability of an approach to optimal control of transient flow problems based on automatic differentiation (AD) [5]. As an example we consider the impulsive flow of a compressible viscous fluid between two parallel walls. The objective is to determine a time dependent cooling rate that compensates the heat release caused by internal friction and thus leads to a nearly constant temperature distribution.

The flow is governed by the Navier-Stokes equations (see e.g. [17])

$$\partial_t \begin{bmatrix} \rho \\ \rho v \\ \rho e \end{bmatrix} = -\nabla \cdot \begin{bmatrix} \rho v \\ \rho v v + \nabla p - \nabla \cdot \tau \\ (\rho e + p) v - \nabla \cdot (v \cdot \tau + \lambda \nabla T \end{bmatrix} + \begin{bmatrix} 0 \\ f \\ v \cdot f + q \end{bmatrix}$$

with

$$\tau = \eta \left(\nabla v + (\nabla v)^{\mathrm{T}} \right) - \frac{2}{3} \eta \mathbf{I} \nabla \cdot v.$$

or in short

$$\partial_t u = \mathscr{F}(u) \tag{1}$$

where ρ is the density, v velocity, T temperature, $e = c_v T + \frac{1}{2} v^2$ total energy, $p = \rho R T$ pressure, R gas constant, $c_v = R/(\gamma - 1)$ specific heat at constant volume, f body force, q heat source, and u represents the state vector. The fluid is confined by two isothermal walls located at $y = \pm a$ and driven by a constant body force $f = f e_x$. The asymptotic solution for the case $q = 0$ is given by $u_\infty = u(\rho_\infty, v_\infty, T_\infty)$ where ρ_∞ is a constant,

$$v_\infty = \frac{f}{2\eta}(a^2 - y^2) e_x, \quad T_\infty = \frac{f^2}{12 \eta \lambda}(a^4 - y^4) + T_{\mathrm{w}},$$

and T_w is the wall temperature. We remark that, alternatively, $T_\infty = T_\mathrm{w}$ can be achieved by choosing $q = q_\infty := -\eta v_\infty'^2$. In the following we assume that the heat source is given by

$$q(c) = cq_\infty \tag{2}$$

where c is a time-dependent control parameter.

The model problem is discretized in the truncated domain $\Omega = (0,l) \times (-a,a)$ using a discontinuous Galerkin method in space and a TVD Runge-Kutta method of order 3 in time (see [2]). The time integration is performed on the interval $[0,t_e]$ with an a-priori fixed step size $h \in I\!R$ resulting in $n = t_e/h$ time steps. The control is distributed over the whole time interval. For our discretization, it can be represented by a finite dimensional vector $c \in I\!R^{n+1}$, where the ith component of c acts only on the time step that transfers the state $u_i \in I\!R^m$ at time t_i to the state $u_{i+1} \in I\!R^m$ at time t_{i+1} for $i = 0,\ldots,n-1$. Hence, the development of the complete system for a given initial value u_0 is computed by a Runge-Kutta integration of the following form

```
do i = 1, n
    call TVDRK(u, t, h, c)
end do
```

Here, the state vector u contains the system state at time t before the call of TVDRK(...) and the system state at time $t+h$ after the call of TVDRK(...) has been completed. The variable t is updated correspondingly.

2 Optimal Control

The mere simulation of physical systems described in Sect. 1 forms even nowadays an active research area. However, we want to go one step further in optimizing the transition from an initial state u_0 to a steady state $\tilde{u}$ at time t_e. Because of physical reasons, we aim at keeping the temperature in the whole domain close to the wall temperature, i.e., the regularized objective function becomes

$$J(u,c) = \int_0^{t_e} \int_\Omega |T(u,c) - T_\mathrm{w}|^2 dx dt + \mu \int_0^{t_e} c^2 dt \tag{3}$$

with a small penalty factor $\mu \in I\!R$. Throughout the paper, we assume that equation (3) admits a unique solution $u(c)$ for every control c. Hence, we can derive a reduced cost function

$$\hat{J}(c) = \int_0^{t_e} \int_\Omega |T(u(c),c) - T_\mathrm{w}|^2 dx dt + \mu \int_0^{t_e} c^2 dt \tag{4}$$

depending only on the control c. For our discretization of the model problem, the evaluation of the objective $\hat{J}(c)$ can be incorporated easily in the time integration:

```
obj = 0
do i = 1, n
    call TVDRK(u, t, h, c, o)
    obj = obj + o + mu * c(i) * c(i)
end do
```

Here, the time step routine TVDRK(...) computes in addition to the state transition the integral of the temperature difference at the time t, i.e. an approximation of the inner integral in (4). After the call, the contributions of o and c are added to the objective value approximating the outer integration in (4).

We want to apply a calculus-based optimization algorithm for computing an optimal control c such that (4) is minimized. For this purpose, we need at least the gradient $\partial \hat{J}(c)/\partial c$. Obviously, one could derive the continuous adjoint partial differential equation belonging to (1) together with an appropriate discretization approach for the gradient calculation. However, we want to exploit the already existing code for the state equation as much as possible by applying AD to compute the corresponding discrete gradient information for our discretization of the state equation.

3 Automatic Differentiation

Over the last few decades, extensive research activities have led to a thorough understanding and analysis of the basic modes of AD. The theoretical complexity results obtained here are typically based on the operation count of the considered vector-valued function. Using the forward mode of AD, one Jacobian-vector product can be calculated with an average operation count of no more than five times the operation count of the pure function evaluation [5] [5]. Similarly, one vector-Jacobian product, e.g. the gradient of a scalar-valued component function, can be obtained using the reverse mode in its basic form at a cost of no more than five times the operation count of the pure function evaluation [5]. It is important to note that the latter bound is completely independent of the number of input variables. Hence, AD provides a very efficient way to compute exact adjoint values which form a very important ingredient for solving optimal control problems of the kind considered in this paper.

The AD-enabled NAGWare Fortran Compiler AD-enabled research prototypes of the NAGWare Fortran compiler are developed as part of the CompAD project[6] by the University of Hertfordshire and RWTH Aachen University. The compiler provides forward [16] and reverse modes [15] by operator overloading as well as by source transformation [13] – the latter for a limited but constantly growing subset of the Fortran language standard. Second-order adjoints can be computed by overloading the adjoint code in forward mode as described in [14] or by generating a tangent-linear version of the compiler-generated adjoint code in assembler format [4].

Support for operator overloading is provided through automatic type changes. All *active*[7] [10] program variables are redeclared as `compad_type` by the compiler. Runtime support modules are included. Various further transformations are required to ensure semantic correctness of the resulting code. See [14] for details.

[5] The computational graph of the original function contains one vertex for every *elemental* function (arithmetic operations and intrinsic functions) with at most two incoming edges (labeled with the local partial derivatives). Assuming that elemental functions are evaluated at unit cost, that local partial derivatives are evaluated at unit cost, and that the propagation of the directional derivatives is performed at unit costs per edge, the computational cost factor adds up to five.

[6] `wiki.stce.rwth-aachen.de/bin/view/Projects/CompAD/WebHome`

[7] Currently, all floating-point variables are considered to be active – thus forming a conservative overestimation of the set of active variables. A computational overhead depending on the size of the overestimation is introduced. Static activity analysis [10] should be used to reduce the number of activated variables. This capability is currently being added to the compiler.

In the given context the reverse mode is implemented as an interpretation of a variant of the computational graph (also referred to as the *tape*) that is built by overloading the elemental functions appropriately. This solution is inspired by the approach taken in ADOL-C [6]. The result of each elemental function is associated with a unique tape entry. All tape entries are indexed. They store opcode[8], value, adjoint value, and indices of the corresponding arguments. The independent variables are registered with the tape through a special subroutine call. The tape entries of dependent variables can be accessed via the respective index stored in `compad_type`. Their adjoint values need to be *seeded* by the user. Knowing the opcode of each elemental function and the values of its arguments, local partial derivatives can be computed and used subsequently in the reverse propagation of adjoints through the tape. By the end of the interpretive reverse tape traversal the adjoints of the independent variables can be *harvested* by the user through accessing the corresponding tape entries.

Use of revolve We are interested in the optimization of an evolutionary process running for at least $n = 5000$ time steps, each evaluating the same computational kernel `TVDRK(u, t, h, c)` (see Sect. 1). Because reverse propagation of a time step requires the state of the system at the end of that time step, adjoining the complete time evolution needs the computational graph of the complete system. The adjoint propagation through the complete system implies the inversion of the order of the time steps. If we assume that for a specific time step i with $1 \leq i \leq n$ the adjoint propagation through all subsequent time steps $n, n-1, \ldots, i+1$ is already done, only the tape of time step i is required to propagate the adjoints through that time step. Thus the tapes of the time steps are required in opposite order, one at a time only.

Various *checkpointing* strategies have been developed to overcome the drawbacks of the two most obvious techniques: *STORE ALL* stores the complete tape at once avoiding any reevaluation of time steps, whereas *RECOMPUTE ALL* evaluates $n*(n-1)/2$ times the computational kernel `TVDRK` from the program's inputs. Checkpointing strategies use a small number of memory units (checkpoints) to store states of the system at distinct time steps. The computational complexity will be reduced dramatically in comparison to *RECOMPUTE ALL* by starting the recomputation of other required states from the checkpoints (see Fig. 2).

A simple checkpointing is called windowing in the PDE-related literature (see [1]). Here, the checkpointing strategy is based on a uniform distribution of checkpoints. However, for a fixed number of checkpoints there exists an upper bound on the number of time steps whose adjoint can be calculated. More advanced checkpointing strategies, as e.g., the binary checkpointing approach [12], had been proposed in the literature. However, only the binomial checkpointing strategy yields a provable optimal, i.e. minimal, amount of recomputations ([8],[7]). A detailed comparison of different checkpointing strategies can be found in [19]. The binomial checkpointing approach is implemented in the C++ package `revolve` [7].

If the number of time steps performed during the integration of the state equation is known a-priori, one can compute (optimal) binomial checkpointing schedules in advance to achieve for a given number of checkpoints an optimal, i.e. minimal, runtime increase [7]. This procedure is referred to as offline checkpointing and implemented in the C++ package `revolve` [7]. We use a C-wrapper wrap_revolve(..., int mode, ...) to call the relevant library routine from within the Fortran 90 code. Therein an instance

[8] A unique number representing the type of the elemental function (e.g., addition, sine, ...)

t of class Revolve is created whose member t $\rightarrow$ revolve(..., mode, ...) returns the integer mode that is used for steering the reversal of the time-stepping loop. Its value is a function of the total number of loop iterations performed and the number of checkpoints used in the reversal scheme. The following five modes are possible.

1. mode == TAKESNAPSHOT: A checkpoint is stored allowing for numerically correct out-of-context (no evaluations of prior loop iterations) evaluation of the remaining loop iterations.
2. mode == RESTORESNAPSHOT: A previously stored checkpoint is restored to restart the computation from the current time step.
3. mode == ADVANCE: Run a given number of time steps (this number is computed by wrap_revolve(...) alongside with mode) from the last restored checkpoint.
4. mode == FIRSTTURN: Compute the adjoint of the last time step, that is, generate a tape for the last time step and call the tape interpreter after initializing the adjoint of the objective (our sole dependent variable) with one.
5. mode == TURN: Compute the adjoint of a single (not the last one) time step, similarly to the previous case. The correct communication of the adjoints computed by interpretation of the tape of the following time step ($tape_{i+1}$) into that of the current one ($tape_i$) needs to be taken care of.

The two special modes TERMINATE and ERROR indicate success or failure of the adjoint computation.

More specifically, our time step consists of a single call to the time-integration routine $\mathsf{TVDRK}(\mathsf{u},\mathsf{t},\mathsf{h},\mathsf{c},\mathsf{o})$ followed by incrementing the objective $obj = \hat{J}(c) \in \mathbb{R}$ by $o \in \mathbb{R}$ and the appropriate weighted component of $c \in \mathbb{R}^n$ as described in Sect. 2. Further arguments of the called subroutine are the state vector $u \in R^{4\times m}$, where $m = 72$ is the number of grid points, the current time $t \in \mathbb{R}$, the size of a single time step $h \in \mathbb{R}$, and the control vector $c \in \mathbb{R}^n$ with the following i/o pattern

$$\mathsf{TVDRK}(\underset{\downarrow}{\overset{\downarrow}{\mathsf{u}}}, \underset{\downarrow}{\overset{\downarrow}{\mathsf{t}}}, \overset{\downarrow}{\mathsf{h}}, \overset{\downarrow}{\mathsf{c}}, \underset{\downarrow}{\mathsf{o}}) \quad .$$

Overset down-arrows mark inputs. Outputs are marked by underset down-arrows. Any single checkpoint consists of u, t, and obj. The corresponding adjoints of u, t and obj need to be communicated from $tape_{i+1}$ to $tape_i$.

4 Numerical Results. Conclusion. Outlook

From a theoretical point of view, the optimization problem is easily solved just by setting the control to unity. In practice, however, the situation is not trivial when starting the iteration process with zero control. Because the initial contribution to the objective is always zero (or very small) implied by the initial conditions and the explicit time integration, it is difficult (if possible) for a gradient-based method to adjust the control parameter correctly. As a consequence a (temperature) perturbation develops, which results in an unavoidable increase in the objective until a new equilibrium is established.

For our numerical tests, we computed for 5000 time steps in advance the velocity and the temperature for the control q equal to one. We refer to this setting as the unperturbed situation. For the optimization task, we considered the following perturbed situation: We took the velocity and temperature obtained for $q(t) \equiv 1$ as initial state but set the current control equal to zero. That is, our initial value for the control is $q_0(t) \equiv 0$.

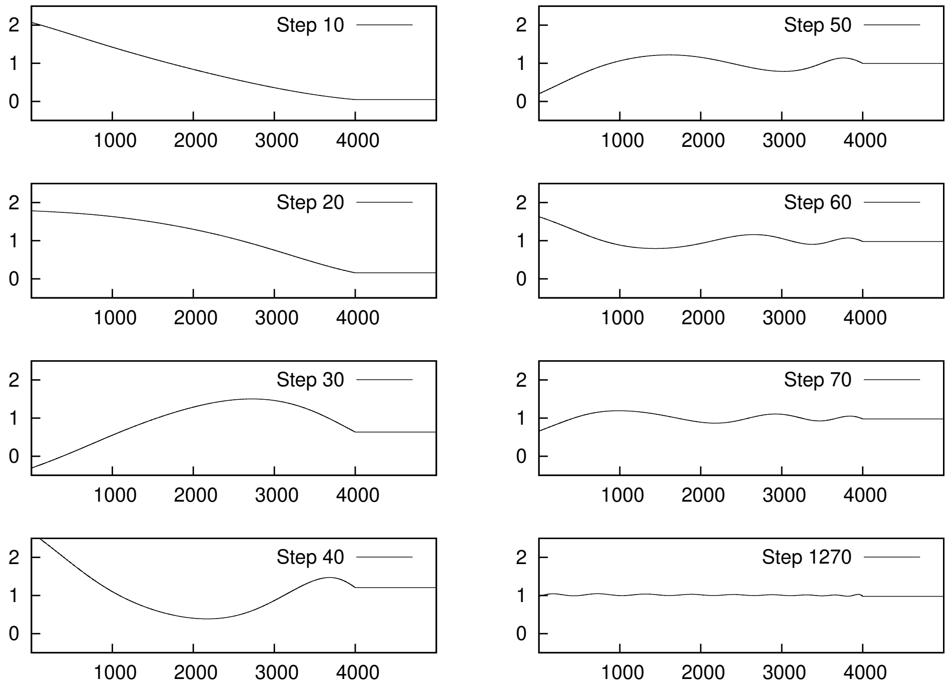

Fig. 1. Development of the values of the control vector: Substantial changes can be observed during the first 100 optimization steps. A good approximation to the asymptotically expected solution $(1,\ldots,1)^T$ is obtained after 1270 steps. Only the first 80% of the control vector are modified.

state space dimension	288	time steps	5000
independent variables	5000	dependent variables	1
size of a tape entry	36 Byte	tape size per time step	12 MB
		STORE ALL would need	60 GB
variables in checkpoint	288 + 1	size of a checkpoint	2.3 KB
number of checkpoints	400	memory for checkpoints	920 KB
Recomputations per optimization step:			
`revolve`	9.598	*RECOMPUTE ALL*	12.497.500
Line-search, function evaluations:			
total	$\approx$ 15600	average per iteration	12.3
minimum	4	maximum	16
Line-search, step length:			
average	$3.33 \cdot 10^{-3}$		
minimum	$3.05 \cdot 10^{-5}$	maximum	0.125

Fig. 2. Test Case Characteristics. The state vector U consists of 288 elements. A checkpoint consists of the state U and the value of the objective function.

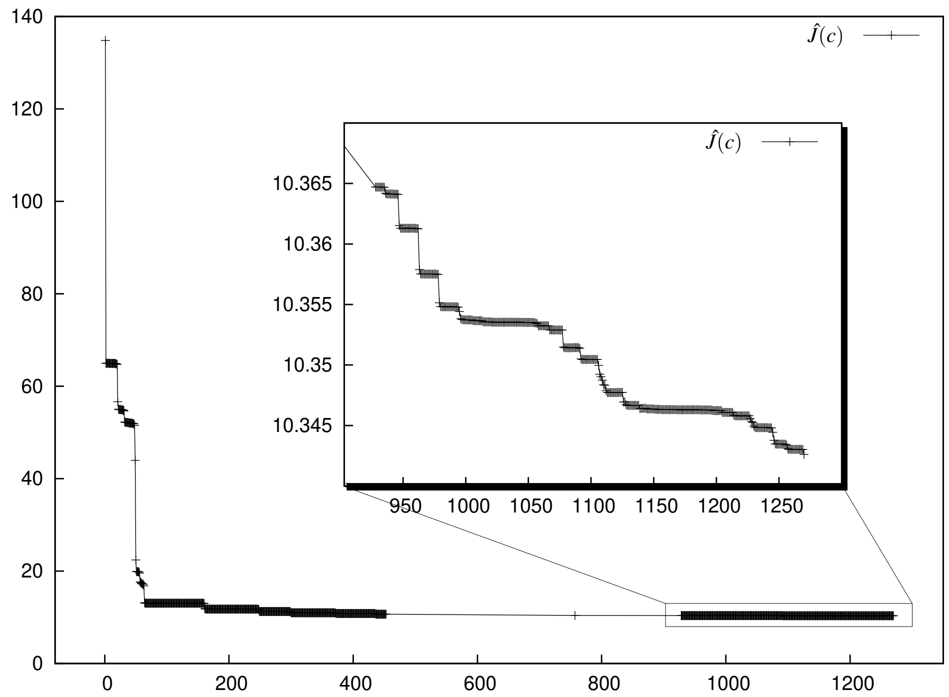

Fig. 3. Development of the value of the objective function over 1270 optimization steps. **Outer figure**: The most substantial advances are made during the first 100 optimization steps. The gap in the logged data between step 450 and 930 is caused by a technical problem (disc space quota exceeded). The computation itself ran uninterrupted with logging resumed after 930 optimization steps. Gradual improvement can be observed throughout the (logged) optimization process. **Inner figure**: During the last 300 out of 1270 steps a continuous (but rather small) improvement of the value of the objective function can be observed.

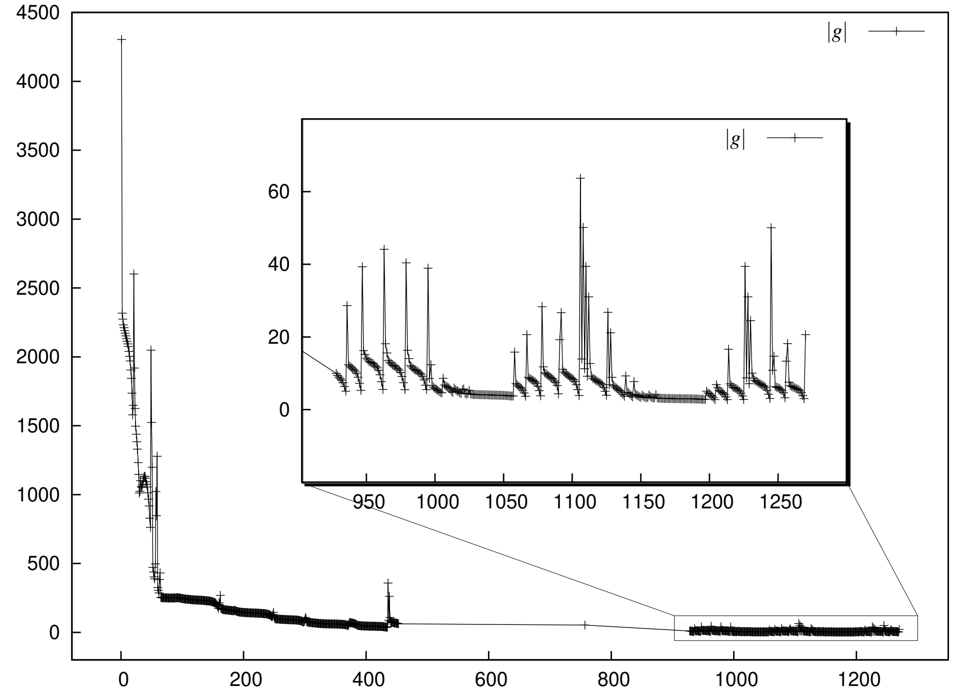

Fig. 4. Development of the value of the L_2-norm of the gradient over 1270 optimization steps **Outer figure**: The value is reduced significantly from over 4000 down to less than 2.75. **Inner figure**: Over the last 300 out of 1270 optimization steps we observe significant changes in the value of the norm of the gradient even at this later stage of the optimization process. Nevertheless, a reduction of the objective is obtained.

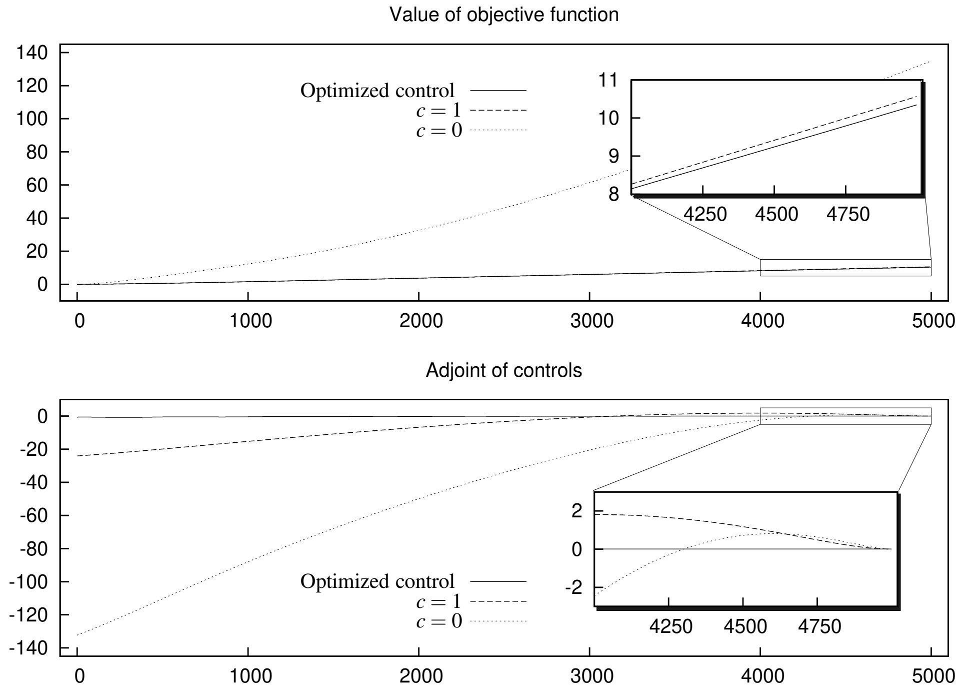

Fig. 5. Development of the value of the objective function (upper figure) and gradient (lower figure) over 5000 time steps.

Hence, we start the optimization with a severely disturbed system yielding the objective value 134 and a comparatively high norm of the gradient $\|\nabla q\| \approx 4303$. Refer to Fig. 2 for further characteristics of this test case.

As the optimization algorithm we apply a simple steepest descent method with backtracking as line search to determine the step size. Because of the chosen discretization the last components of the control have either no or only very small influence on the objective. Therefore, we perform the optimization only for the first 80 % of the considered time interval as illustrated in Fig. 1. This approach can be interpreted as steering the process over a certain time horizon with a second time interval where the system can converge to a certain state. Using this simplification, the objective could be reduced to 10.346, i.e., a value less than 10.560, which was obtained for the unperturbed situation. The development of the value of the objective is shown graphically in Fig. 3. Because we are interested in the overall performance of the optimization, we do not enforce a strong termination criterion. As can be seen also from the inner figures of Fig. 3 and Fig.4, the development of the objective and the norm of the gradient show a typical behavior as expected for a simple steepest descent method. After 1270 gradient steps, the optimization yields a recovery strategy for the perturbed system. Caused by the severe disturbance of the system, the norm of the gradient in the last optimizations steps is considerably reduced compared to the starting point, but not equal to or very close to zero. See Fig. 4 for illustration.

Nevertheless, the numerical results show that AD-based optimization is feasible for such complicated optimization tasks. Future work will be dedicated to the usage of higher-order derivatives in the context of more sophisticated optimization algorithms. Here, one has to distinguish time-dependent problems as our model example and pseudo-time-dependent methods frequently used for example in aerodynamics. To this end, the currently used derivative calculation will be adapted for the usage in so-called SAND methods or one-shot approaches depending on the problem at hand [9]. Furthermore, our example will be adapted to more realistic scenarios as for example a plasma spraying problem.

The NAGWare compiler's capabilities to generate adjoint code (as opposed to changing the types of all floating-point variables and using operator overloading for the generation of a tape) will be enhanced to be able to handle the full code. All these measures in addition to the exploitation of internal structure of the problem [18] are expected to result in a considerably decreased overall runtime. We expect savings of at least a factor of 50 driving the current runtime of 2 weeks on a state-of-the-art PC down to a couple of hours.

References

1. Martin Berggren. Numerical solution of a flow-control problem: Vorticity reduction by dynamic boundary action. *SIAM J. Sci. Comput.*, 19(3):829–860, 1998.
2. B. Cockburn and C.-W. Shu. Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, 16:173–261, 2001.
3. S. S. Collis, R. D. Joslin, A. Seifert, and V. Theofilis. Issues in active flow control: theory, control, simulation and experiment. *Progress in Aerospace Sciences*, 40:237–289, 2004.
4. D. Gendler, U. Naumann, and B. Christianson. Automatic differentiation of assembler code. In *Proceedings of the IADIS International Conference on Applied Computing*, pages 431–436. IADIS, 2007.
5. A. Griewank. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*. SIAM, Apr. 2000.

6. A. Griewank, D. Juedes, and J. Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Soft.*, 22:131–167, 1996.
7. A. Griewank and A. Walther. Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Software*, 26:19–45, 2000.
8. Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.
9. Max D. Gunzburger. *Perspectives in Flow Control and Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.
10. L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.
11. J. Kim and T. R. Bewley. A linear systems approach to flow control. *Annual Review of Fluid Mechanics*, 39:383–417, 2007.
12. K. Kubota. A Fortran77 preprocessor for reverse mode automatic differentiation with recursive checkpointing. *Optimization Methods and Software*, 10:319 – 336, 1998.
13. M. Maier and U. Naumann. Intraprocedural adjoint code generated by the differentiation-enabled NAGWare Fortran compiler. In *Proceedings of 5th International Conference on Engineering Computational Technology (ECT 2006)*, pages 1–19. Civil-Comp Press, 2006.
14. U. Naumann, M. Maier, J. Riehme, and B. Christianson. Automatic first- and second-order adjoints for truncated Newton. In *Proceedings of the Workshop on Computer Aspects of Numerical Algorithms (CANA'07)*, Wisla, Poland, 2007. To appear.
15. U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In H. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, number 50 in Lecture Notes in Computational Science and Engineering, pages 159–170. Springer, 2005.
16. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, 2005.
17. J. H. Spurk. *Fluid Mechanics*. Springer, 2007.
18. P. Stumm, A. Walther, J. Riehme, and U. Naumann. Structure-exploiting automatic differentiation of finite element discretizations. Technical report, SPP1253-15-02, Technische Universität Dresden, 2007.
19. A. Walther and A. Griewank. Advantages of binomial checkpointing for memory-reduced adjoint calculations. In M. Feistauer, V. Dolejší, P. Knobloch, and K. Najzar, editors, *Numerical Mathematics and Advanced Applications, ENUMATH 2003, Prag*, pages 834–843. Springer, 2004.