RWTH Aachen

Department of Computer Science
Technical Report

Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

Max Sagebaum and Nicolas R. Gauger and Uwe Naumann and Johannes Lotz and Klaus Leppkes

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2013-04

RWTH Aachen · Department of Computer Science · February 2013

Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

Max Sagebaum and Nicolas R. Gauger and Uwe Naumann and Johannes Lotz and Klaus Leppkes

Email: sagebaum@mathcces.rwth-aachen.de

Abstract. Algorithmic differentiation (AD) is a mathematical concept which evolved over the last decades to a very robust and well understood tool for computation of derivatives. It can be applied to mathematical algorithms, codes for numerical simulation, and whenever derivatives are needed. In this paper we report on the algorithmic differentiation of the discontinuous Galerkin solver *padge*, a large and complex code written in C++ with underlying external libraries. The reports on successful application of AD to large scale codes are rare in literature and up to now this is not state of the art. Most of the codes, which are differentiated nowadays, are written in C or Fortran. The *padge* code was differentiated with the operator overloading tool *dco/c++* in forward as well as reverse mode. The differentiated code is validated and runs in the expected time margins of AD.

1 Introduction

In this paper we report on the algorithmic differentiation of the discontinuous Galerkin solver *padge* from German Aerospace Center (DLR), an extensive code written in C++ with underlying external libraries. The *padge* code solves the compressible Reynolds-averaged Navier-Stokes equations (RANS). The code relies on seven different libraries, including the finite element package *deal.II*, *PETSc*, a library for solving linear and nonlinear problems, *MPI* for parallel computation, and the AD library *Sacado*.

We undertake the algorithmic differentiation of *padge* due to the need of derivatives for optimization strategies like one-shot [1]. The derivatives with respect to the state (e.g. the flow field) and the design parameters (e.g the wing shape) can be calculated with the algorithmic differentiated version of *padge*. The differentiated version of *padge* is created in a black box manner. The operator overloading tool *dco/c++* is used to activate all the variables of *padge* and the underlying libraries for the algorithmic differentiation. Then the activated variables can be used to calculated the derivatives with *padge*.

It turned out that it makes no sense to differentiate all the underlying libraries. For instance the Newton solves, performed with *PETSc*, need not to be differentiated in a black box manner. As the discrete adjoint of the Newton solver turns out to be infeasible in terms of memory usage, we decided to embed a continuous adjoint implementation in the algorithmic differentiation process. The continuous adjoint is included such that the normal *padge* code is not affected. Also the libraries for input and output operations such as NetCDF and OpenCascade and the libraries for mesh operations like METIS do not need to be differentiated as they are not involved in the computation of the solution.

The paper is organized as follows: First we give a short introduction to AD, which contains the forward as well as the reverse mode. Then, the AD tool *dco* is introduced. In a next chapter we describe how *padge* has been differentiated by *dco*. Furthermore, we discuss the special treatments in the differentiation process of the libraries, included in *padge*. In a last chapter we present the validation of the differentiated code and give numbers for its performance.

2 Algorithmic Differentiation

Algorithmic differentiation bases on the theory for the differentiation of a call sequence. A good foundation of algorithmic differentiation (AD) is given by the books of Naumann et. al [2] and Griewank et. al [3]. The efforts to extend the theory is ongoing, see e.g. [4], [5], [6]. There are two classes of AD tools, namely source transformation and operator overloading.

With source transformation the source code is parsed and a new code is generated, which is extended to calculate the derivatives alongside the normal calculations. This process can be applied to the whole program or to a single function. The functions in the generated code have the same layout as the corresponding functions in the original code. The difference is that for each input and output variable a new input or output variable is added. The new variables contain the information about the derivatives. An example for source transformation tools is *Tapenade* [7]. *Tapenade* was at first written as an AD tool for Fortran and recently extended to handle C code [8].

Examples for operator overloading tools are *ADOL-C* [9] and *dco* [10]. With operator overloading the calculation type in the application or in a function is exchanged to an active type, which is provided by the AD tool. The type overloads the arithmetic operators and the basic mathematical functions. During the execution of the program the AD tool stores information about the structure of the activated code or computes directly the derivative information. To apply AD on a code fragment or to make the code fragment active, each AD tool has its of own set functions and strategy.

For brevity we will just state the definitions for the *Forward* and *Reverse* Mode of AD. For a more detailed introduction see the book of Naumann et al.[2].

Definition 1. Forward AD Mode: *For the function $y = F(x)$, given by the call sequence $F(x) = (f_N \circ f_{N-1} \circ ... \circ f_1 \circ f_0)(x)$, the* tangent-linear projection

$$\dot{y} = \frac{dF}{dx}\dot{x}$$

is computed by evaluating in a forward loop for $i = 0, \ldots, (N-1)$

$$v_{i+1} = f(v_i) \qquad \text{and} \qquad \dot{v}_{i+1} = \frac{df}{dv_i}(v_i)\dot{v}_i$$

simultaneously, with $f_i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_{i+1}}, v_i \mapsto v_{i+1}$. Furthermore, we identify $x \equiv v_0$, $\dot{x} \equiv \dot{v}_0$ for the input variables and $y \equiv v_{N+1}$, $\dot{y} \equiv \dot{v}_{N+1}$ for the output variables.

We want to emphasize that the Jacobi vector product $\frac{dF}{dx}\dot{x}$ in Definition 1 is not computed by building the jacobi $\frac{dF}{dx}$ and then multiplying it with $\dot{x}$. The computation is matrix free and calculated alongside the normal evaluation.

Definition 2. *Reverse AD Mode: For the function $y = F(x)$ given by the call sequence $F(x) = (f_N \circ f_{N-1} \circ ... \circ f_1 \circ f_0)(x)$, the* adjoint projection

$$\bar{x} = \frac{dF}{dx}^T \bar{y}$$

is computed by evaluating

$$\begin{aligned} v_{i+1} &= f(v_i) && \text{in a forward loop from } i = 0, \ldots, N \text{ and then evaluating} \\ \bar{v}_i &= \left[\frac{df}{dv_i}(v_i)\right]^T \bar{v}_{i+1} && \text{in a reverse loop from } i = N, \ldots, 0 \ , \end{aligned}$$

with $f_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_{i+1}}, v_i \mapsto v_{i+1}$. Furthermore, we identify $x \equiv v_0$, $\bar{y} \equiv \bar{v}_{N+1}$ for the input variables and $y \equiv v_N$, $\bar{x} \equiv \bar{v}_0$ the output variables.

We want to emphasize again that the Jacobi vector product in Definition 2 is not computed by building the Jacobi $\frac{dF}{dx}$, transpose the Jacobi and then multiplying it with $\bar{y}$. The product is calculated in a matrix free way during the reverse sweep of the reverse AD mode.

The f_i in the Definitions 1 and 2 are normally identified with the elemental functions $+, -, *, /, pow, sqrt, sin, cos, \ldots$ but f_i can also be a more complex function like linear or nonlinear solver. The identification is done in such a way that each f_i performs one elemental operation and sets the result to one intermediate variable. The other intermediate variables are unchanged. An AD tool for operator overloading will overload operators like $+$ and $*$ and functions like *sin* and *pow* to implement the results from Definitions 1 and 2. This enables the program to calculate the derivatives.

In Table 1 a few selected elemental functions are shown. The first three functions are not very difficult to derive but they illustrate the basic concept of AD very well. The fourth function describes the solution of a linear system. It turns out that the solver for the original system can be used to calculate the forward and reverse results.

Table 1. Forward and Reverse AD Mode for some elemental operations

Operation	$u = w + v$	$u = w * v$	$u = sin(v)$	$u = W^{-1}v$
Forward AD Mode	$\dot{u} = \dot{w} + \dot{v}$	$\dot{u} = v * \dot{w} + w * \dot{v}$	$\dot{u} = cos(v) * \dot{v}$	$\dot{u} = W^{-1}(\dot{v} - \dot{W}u)$
Reverse AD Mode	$\bar{v}$ += $\bar{u}$ $\bar{w}$ += $\bar{u}$ $\bar{u} = 0$	$\bar{v}$ += $w * \bar{u}$ $\bar{w}$ += $v * \bar{u}$ $\bar{u} = 0$	$\bar{v}$ += $cos(v) * \bar{u}$ $\bar{u} = 0$	$s = W^{-1^T}\bar{u}$ $\bar{W}$ += $-u \cdot s^T$ $\bar{v}$ += s $\bar{u} = 0$

3 The AD tool dco

dco/c++ (*d*erivative *c*ode by *o*verloading in c++) [10] is developed from the STCE (Software and Tools for Computational Engineering) group at RWTH Aachen. *dco/c++* make use of template expressions and other c++ features mainly to generate efficient derivative code, e.g. by statement-level pre accumulation.

Another very helpful feature is the possibility to integrate external functions easily. An external function is a code block, which is not to be differentiated by *dco*. The derivative of the external function is provided by the user himself via an interface, next to the code parts differentiated by *dco*. This interface can be used to differentiate a routine or library, for which *dco* should not be applied (e.g. linear system or iterative solvers), but where a derivative function is available or the derivative could be easily implemented by hand.

In the reminder of the paper we will give examples on how *dco* is used in the differentiation progress. The derivative types programmed with *dco* are the *dco::t1s::type* for the tangent projection of the *Forward AD Mode* and the *dco::a1s::type* for the adjoint projection of the *Reverse AD Mode*. For the evaluation of second order derivatives the type *dco::t2s_a1s::type* is available. The second order derivatives are computed by a tangent linear projection on the adjoint projection.

4 The differentiation of Padge

The *padge* code [11], developed by DLR Braunschweig, is a discontinuous Galerkin solver, which solves the compressible Reynolds-averaged Navier-Stokes equations (RANS). The code is written in C++ with underlying external libraries. It solves on structured grids, which can be hp-refined. The solvers for the flow solution are multigrid or Newton type. The internal structure of *padge* separates different areas of the solution process into different packages, which are then combined to form the solution process. Some of the packages depend on special external libraries, other external libraries are used throughout the *padge* code. There are seven different libraries:

- *NetCDF* (Network Common Data Form) [12] is a library for loading and storing data. The data is stored in a machine independent format, which is stored such that each file describes its own structure.
- *METIS* [13] provides functions and algorithms for the decomposition of graphs and finite element meshes.
- *OpenCascade* is a software package, which provides methods for the handling of CAD files.
- *Sacado* [14] is a part of the Trilinos project. Sacado is another AD tool for the algorithmic differentiation of C++ codes. The forward mode of Sacado is only used at three locations inside *padge*.
- *deal.II* (Differential Equations Analysis Library) [15] is a library for the implementation of finite element methods.
- *PETSc* (Portable, Extensible Toolkit for Scientific Computation) [16] is a extensive library for solving linear and nonlinear problems. It defines its own structures and solver routines.
- *MPI* (Message Parsing Interface) [17] is a standard for the parallelisation of programs for multiple machines. MPI takes care of the communication of the different processes and the distribution of the tasks.

4.1 Concept of Differentiation

With respect to AD the libraries, which are used by *padge*, can be classified into two categories. The first category contains with *NetCDF*, *METIS* and *OpenCascade* all packages, which are used for data handling and IO operations. In Figure 1 these packages are placed on the left hand side. All libraries in this category can be seen as non dependent with respect to AD. The consequence is that the libraries don't need to be differentiated with AD. Only at the interface to *padge* a data conversion to the active type is needed.

On the right hand side in Figure 1, the second category is placed. The libraries *Sacado*, *deal.II*, *PETSc* and *MPI* contribute all to the calculation of the solution. *padge* uses *deal.II* as the core for the computation. Therefore, *deal.II* has to be differentiated. Furthermore, *Sacado* needs also to be differentiated by *dco*, which yields consequently second order derivatives. Finally, *PETSc* is handled by the external function interface of *dco*. We decided to disable *MPI* for the differentiation of *padge* to reduce initial workload.

Consequently, the grey box in Figure 1 contains all parts of code and libraries to be differentiated by AD. The strategy for the differentiation is to implement a black box differentiation of *deal.II*, *sacado* and *padge* first. During this black box differentiation critical parts of code are identified for the application of more advanced AD techniques and special treatments. The differentiated code of each library is to be verified after each black box differentiation. This ensures that dependent libraries are not affected by errors during further differentiation.

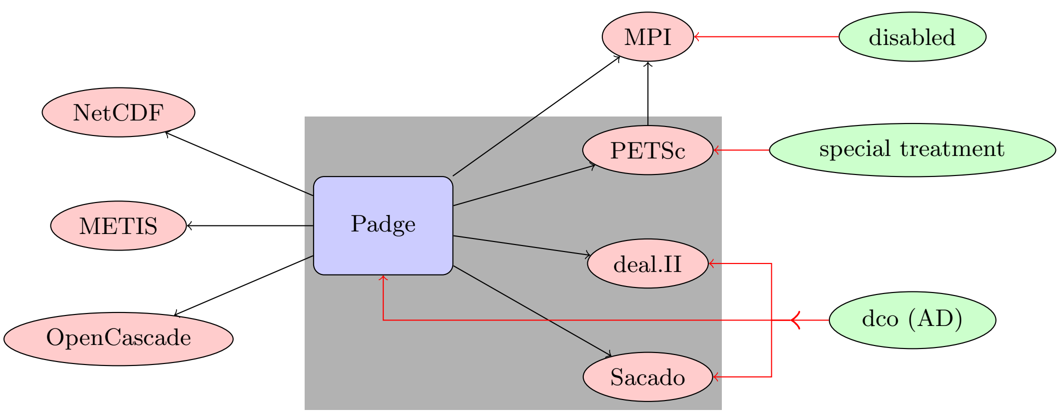

Fig. 1. Dependencies for padge and AD approach overview.

4.2 Differentiation of deal.II and padge

The first challenge to differentiate *deal.II* and *padge* is the absence of definitions for a general calculation type. Large parts of code in *deal.II* and *padge* are not templated and typedefs aren't used. Consequently, we introduce the typedef *BASE_TYPE* as calculation type. Each 'double' is replaced by the type *BASE_TYPE*. *BASE_TYPE* is now used to switch between the previous calculation type 'double' and the active types from *dco*. The current code for switching between the different types is displayed in Figure 2. Within *deal.II* there is the

```
//#define BT_DOUBLE        // normal calculation type
//#define BT_DCO_T1S       // forward mode of dco
//#define BT_DCO_A1S       // reverse mode of dco
#define BT_DCO_T2S_A1S     // second order derivative mode of dco

#if defined(BT_DOUBLE)
// nothing here
#elif defined(BT_DCO_T1S) || defined(BT_DCO_A1S) ||
    defined(BT_DCO_T2S_A1S)
#define BT_DCO
#else
#error Define one base type with BT_*
#endif

// include dco header if needed
#ifdef BT_DCO
#include "../../../dco/include/dco.hpp"
#endif

#if defined(BT_DOUBLE)
  typedef double BASE_TYPE;
#elif defined(BT_DCO_T1S)
  typedef dco::t1s::type BASE_TYPE;
#elif defined(BT_DCO_A1S)
  typedef dco::a1s::type BASE_TYPE;
#elif defined(BT_DCO_T2S_A1S)
  typedef dco::t2s_a1s::type BASE_TYPE;
#else
#error Missing typedef for a BT_*
#endif
```

Fig. 2. The structure for switching between the different base types.

possibility to call the linear algebra package with mixed precision types. In the following example the matrix is stored in double precision and multiplied by a float vector:

```
template void FullMatrix<double>::vmult<float>(Vector<float>&,
    const Vector<float>&, bool) const;
```

Inside these methods a conversion between the different precisions would be needed. Therefore, the active types for the different precisions should be provided by AD, but this is usually not available for none of the tools. The solution was to disabled the methods for the mixed precisions and remove the usage of this feature in *padge*.

deal.II and *padge* were developed with no templates in mind, and with the assumption that the template type would always be a machine type like float or double. Therefore, the programming rules for template code were not applied. This leads to compiler errors for code lines with implicit conversion, and hard-coded double or float values. The example illustrated in Figure 3 reports the error "ambiguous overload for 'operator='". The compiler has the options to use the operator with the argument BASE_TYPE or IdentityMatrix. The solution is to use a proper template coding convention, which is the cast to template type.

```
SparseMatrix<BASE_TYPE> system_matrix;
...
system_matrix = 0.0;    // error
system_matrix = BASE_TYPE(0.0);   // no error
```

Fig. 3. Example for hardcoded values that lead to compiler errors. For this example: ambiguous overload for 'operator='.

4.3 Sacado

In sacado the calculation type for the derivatives is a template. The implementation of sacado is done in a way that additional information about the template type is needed. Consequently, we have to provide this information for the *dco* type. This is done through an adapted macro from sacado. The new macro creates the information sacado needs to know about the *dco* types. With some other minor changes to sacado, all complier errors could be prevented. The result is the differentiation of an AD tool with another AD tool.

4.4 PETSc

Matrix and Vector classes The parallelisation in *padge* is implemented through the vector and matrix classes of *PETSc*. Therefore, these classes are used throughout the whole *padge* code, next to some MPI calls to start the parallelisation. As we choose to disable MPI in the current project, the challenge is that the matrix and vector classes of *PETSc* are not usable with a custom calculation type. The solution is the exchange of the *PETSc* vectors and matrices with wrapper classes, which provide the same functionality as the *PETSc* classes and are usable with a custom type.

The wrapper classes are derived from the standard *deal.II* vector and matrix. During the differentiation of *deal.II* these two classes were prepared for the usage with the *dco* types and thus meet the requirement that the wrapper should work with custom types. The interface of the standard *deal.II* classes and the *PETSc* classes, which are used in *padge*, are nearly the same.

Linear system solver The next challenge in *padge* is the differentiation of the linear and nonlinear system solvers. As they are implemented with the functionality provided by *PETSc*. As *PETSc* is not easily differentiable with AD tools the linear and nonlinear system solvers have to be treated separately. The first step for the derivation of these routines was a refactoring of these classes, to provided a more convenient way to use the external function interface of *dco*. The solution of a linear system is now done in the method *solve_system*. It is used by the refactored nonlinear solver. For the differentiation of the linear system solver, only *solve_system* has to be overloaded and differentiated by hand.

The hand differentiated version of a linear system solver is presented in Table 1. The fourth row of the table contains the procedures for the forward and reverse mode of the linear system solve for the system $Wu = v$.

For the forward *dco* type dco::t1s::type the implementation can be summarized into three steps: Extract the normal and derivative value from the rhs vector and the matrix, solve the linear systems for the calculation and combine the results to the output vector. In Figure 4 the whole implementation of the method for the forward type is shown. The lines 3 to 6 perform the initialization of the *PETSc* matrices and vectors. They also initialize the *PETSc* structure with the values from *padge*. The normal linear system $Wu = v$ is solved with the call in line 10. The calculation for the forward AD mode $W\dot{u} = (\dot{v} - \dot{W}u)$ is performed in the lines 12 to 14. The third step is done in line 16, which writes the results to the output vector.

```
1  static inline void petsc_solve(KSP ksp, dco::DcoPetscMatrix &W,
      dco::DcoPetscVector &v, dco::DcoPetscVector &u) {
2    PETScWrappers::MPI::SparseMatrix_SEQBAIJ W_values, W_dot; // init
       petsc matrix
3    PETScWrappers::MPI::Vector v_values, v_dot, u_values , u_dot,
       temp ;  // init petsc vectors
4
5    dcoMatrixToPetscMatrix(W, W_values, W_dot);
6    dcoVectorToPetscVector(v, v_values, v_dot);
7
8    // set and solve the system
9    KSPSetOperators(ksp,W_values ,W_values,SAME_NONZERO_PATTERN);
10   KSPSolve(ksp,v_values, u_values); //solve Wu=v
11
12   MatMult(W_dot, u_values, temp); // temp = W_dot * u
13   VecAXPY(v_dot, -1.0, temp); // v_dot = v_dot - temp;
14   KSPSolve(ksp,v_dot, u_dot); //solve Wu_dot=v_dot - W_dot * u
15
16   petscVectorToDCOVector(u_values, u_dot, u);   // create the
       solution vector
17 }
```

Fig. 4. Hand differentiation for the solution to a linear system for the *dco* forward mode.

The implementation of the reverse mode has to be split into two methods, one for the normal evaluation and one for the reverse evaluation. *petsc_solve* for the reverse type of *dco* is shown in Figure 5. The middle part of the method solves the normal linear system. In the lines 2 to 3 and 19 the information for the reverse evaluation is generated. Line 20 registers the function *petsc_solve_adjoint_a1s* for the reverse call on the *dco* tape. The implementation of *petsc_solve_adjoint_a1s* is presented in Figure 6. In the method the values for the reverse evaluation are read and then the linear system $W^T s = \bar{u}$ for the reverse mode is evaluated. The update for the matrix W and the right hand side vector v conclude the adjoint method.

5 Validation, Performance & Statistics

The validation of the gradients, which are calculated with the differentiated version of *Padge*, is done against finite differences. It is known by theory [3], that the gradients obtained by AD are of machine accuracy. With finite differences of first order, the approximation is just of order $\mathcal{O}(h)$ accurate, but the step size cannot be chosen arbitrarily small due to cancellation errors. On the other hand, if h is chosen too large, the truncation error grows. Consequently, the validation plots

```
static inline void petsc_solve(KSP ksp, dco::DcoPetscMatrix &W, dco::DcoPetscVector &v, dco::DcoPetscVector &u) {
  dco::a1s::external_function_data_helper *cp=new dco::a1s::external_function_data_helper();
  cp->write_to_checkpoint(W);  cp->write_to_checkpoint(v); cp->write_to_checkpoint(ksp);

  PETScWrappers::MPI::SparseMatrix_SEQBAIJ W_values; // init petsc matrix
  PETScWrappers::MPI::Vector v_values, u_values; // init petsc vectors

  dcoMatrixToPetscMatrix(W, W_values);
  dcoVectorToPetscVector(v, v_values);

  // set and solve the system
  KSPSetOperators(ksp,W_values ,W_values,SAME_NONZERO_PATTERN);
  KSPSolve(ksp,v_values, u_values); //solve Wu=v

  // create the solution vector and setup everything for the reverse sweep
  petscVectorToDCOVector(u_values, u);
  makeActive(u);

  cp->write_to_checkpoint(u);
  dco::a1s::global_tape->register_external_function(&petsc_solve_adjoint_a1s, cp);
}
```

Fig. 5. Hand differentiation for the solution of a linear system for the dco reverse mode. Function for the evaluation.

```
void petsc_solve_adjoint_a1s(dco::a1s::tape &tape, const dco::a1s::tape::interpretation_settings &settings DCO_UNUSED, dco::a1s::tape::external_function_data *userdata)
{
  dco::a1s::external_function_data_helper *cp=static_cast<dco::a1s::external_function_data_helper*>(userdata);

  dco::DcoPetscMatrix W;
  dco::DcoPetscVector v, u;

  KSP          ksp;        /* linear solver context */

  cp->read_from_checkpoint(W);    cp->read_from_checkpoint(ksp);
  cp->read_from_checkpoint(u);    cp->read_from_checkpoint(v);


  PETScWrappers::MPI::SparseMatrix_SEQBAIJ W_values; // init petsc matrix
  PETScWrappers::MPI::Vector s, u_bar; // init petsc vector

  dcoMatrixToPetscMatrixTranspose(W, W_values); // W_values contains the transposed matrix
  dcoVectorToPetscVector_(u, u_bar);

  // solve the adjoint system
  KSPSetOperators(ksp,W_values  ,W_values,SAME_NONZERO_PATTERN);
  KSPSolve(ksp,u_bar, s);   //solve W^T x=b

  updateAdjointDcoVector(v, s);
  updateAdjointDcoMatrix(W, u, s);
}
```

Fig. 6. Hand differentiation for the solution of a linear system for the dco reverse mode. Function for the reverse sweep.

should show reductions in the approximation error for decrease in step size, until cancelation occurs and the error increases again. This leads to a typical V-shaped curve.

The problem setup for the validation is a NACA 0012 airfoil with the following settings:
$Ma = 0.5$ $\alpha = 2°$ $Re = 5000$ DOF $y = 25600$ (1600 Cells) DOF $u = 40$

The discretized Navier Stokes Equations is solved with a Newton Solver or Backward Euler Solver

$$\begin{aligned} y_{k+1} &= y_k - J(y_k, u)^{-1} R(y_k, u), \\ C_d(y*, u) &\text{ with} \end{aligned}$$

- $y_k \in \mathbb{R}^n$ is the state vector of the k-th iterate
- $y*$ converged solution from the update y_k
- $u \in \mathbb{R}^m$ vector for the design parameters
- $R(y, u)$ residual of the discretized Navier-Stokes equations
- $J(y, u) \approx \frac{\partial R(y,u)}{\partial y}$ approximation of the Jacobi of the residual, as it is coded in padge as Newton or Backward Euler
- $C_d(y, u)$ its the cost functional: drag coefficient of the airfoil

The symbol $\nabla_{dco}(\cdot)$ is used for the gradients produced by *dco*, and $\nabla_{FD}(\cdot)$ for the gradients calculated by finite differences.

Figure 7 illustrates the differences between gradients obtained by AD (in forward as well as reverse mode) and finite differences, for the residuals and Jacobians, differentiated with respect to design as well as state variables (see Def. of $\nabla f_1, \nabla f_2, \nabla f_3$ and ∇f_4 in Figure 7). $\nabla f_5 := \frac{dC_d(y^*(u),u)}{du}$ defines the differentiation of the drag cost functional $C_d(y, u)$ with respect to the vector of design variables u. The function C_d depends directly and indirectly on the design u. The indirect dependency on u is introduced by the converged flow solution y^*, which depends on the design u. The derivative of C_d with respect to u is

$$\frac{dC_d(y^*(u), u)}{du} = \frac{\partial C_d}{\partial y}(y^*(u), u)\frac{\partial y^*}{\partial u}(u) + \frac{\partial C_d}{\partial u}(y^*(u), u) \quad .$$

The flow solution y^* is attained by the Newton solver in *padge*. Through the differentiation of *padge* the newton solver is also differentiated. The term $\frac{\partial y^*}{\partial u}(u)$ can thus be calculated with the differentiated Newton solver. The evaluation of ∇f_5 is therefore done in two step. First the solution y^* is obtained through the Newton solver and then the function $C_d(y^*, u)$ is evaluated. Through AD the the derivative $\frac{dC_d(y^*(u),u)}{du}$ is calculated in the background.

For each of the gradients, Figure 7 shows the expected V-shaped curves.

The runtime measurements, shown in Table 2, are performed for $R(y, u)$ and $J(y, u)$, with degrees of freedom $n = 14400$ for the state vector y, and $m = 40$ for the design vector u. The runtime factors with respect to the first line are represented in brackets. The factors are very close to the optimal theoretical values: 2 to 2.5 for the forward mode, and 3 to 4 for the reverse mode [3]. But this is only true for the differentiation with respect to the design u. The higher dimension of the state vector y introduces an additional factor of 2, which can be

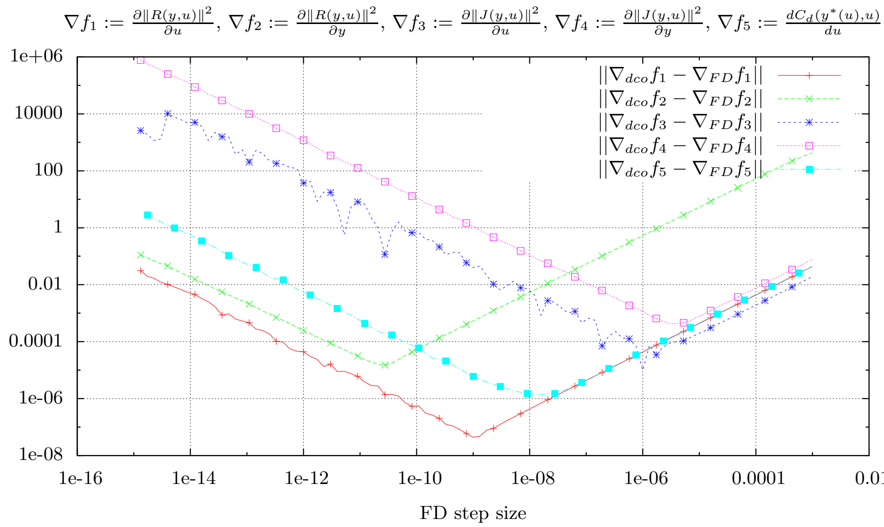

Fig. 7. Validation of *dco* gradients against finite differences approximation of the gradient.

Table 2. Runtimes(ms(factor)) for normal *padge* and differentiated *padge*

Configuration	$\frac{\partial \|R\|_2^2}{\partial u}$	$\frac{\partial \|R\|_2^2}{\partial y}$	$\frac{\partial \|J\|_F^2}{\partial u}$	$\frac{\partial \|J\|_F^2}{\partial y}$
normal	12.7(1.00)	12.7(1.00)	248.4(1.00)	248.4(1.00)
dco forward (one component)	23.7(1.87)	23.7(1.87)	493.7(1.99)	493.7(1.99)
dco reverse (full gradient)	44.7(3.50)	118.0(9.29)	823.0(3.31)	1869.6(7.53)

explained by the larger set of variables. The state y is used in more calculations than the design u and such more information is stored on the tape.

The statistics for the changed code lines are counted for *padge* as well as *deal.II*. The counting is done with the tool *cloc*, which is configured to ignore spaces and empty lines, but to include comment lines. The results from *cloc* are displayed in Table 3. The changes are relatively large because, we had to introduce the typedef for the basic calculation type.

Table 3. Changed code lines for padge and *deal.II*.

deal.II	Code lines + comments	Padge	Code lines + comments
Overall	219235	Overall	132586
Modified	8719 (4,0%)	Modified	8874 (6,7%)

6 Conclusion & Outlook

The successful differentiation of a complex C++ code with underlying libraries, supported by *dco*, has been demonstrated. The key AD techniques have been the introduction of *typedefs* and the use of external functions after refactoring the code.

As future projects we address the incorporation of MPI [18] as well as the differentiation of matrix and vector classes from *PETSc*. This will enable us to revert to the original vectors in *padge* and to remove our own wrapper classes.

References

1. N. Gauger, A. Griewank, A. Hamdi, C. Kratzenstein, E. Özkaya, T. Slawig, Automated extension of fixed point pde solvers for optimal design with bounded retardation, International Series of Numerical Mathematics 160 (2012) 99–122.
2. U. Naumann, The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation., Software, Environments, and Tools, SIAM, Philadelphia, PA, 2012.
3. A. Griewank, A. Walther, Evaluating Derivatives, second edition, SIAM, 2008.
4. A. Lyons, I. Safro, J. Utke, Randomized heuristics for exploiting jacobian scarcity, Optimization Methods and Software 27 (2) (2012) 311–322. arXiv:http://www.tandfonline.com/doi/pdf/10.1080/10556788.2011.577774, doi:10.1080/10556788.2011.577774.
URL `http://www.tandfonline.com/doi/abs/10.1080/10556788.2011.577774`
5. V. Mosenkis, U. Naumann, On optimality preserving eliminations for the minimum edge count and optimal jacobian accumulation problems in linearized dags, Optimization Methods and Software 27 (2012) 337–358.
6. M. Wagner, B.-J. Schaefer, A. Walther, On the efficient computation of high-order derivatives for implicitly defined functions, Computer Physics Communications 181 (2010) 756–764.
7. L. Hascoët, V. Pascual, Tapenade 2.1 user's guide, Technical Report 0300, INRIA (2004).
URL `http://www.inria.fr/rrrt/rt-0300.html`
8. V. Pascual, L. Hascoët, TAPENADE for C, in: Advances in Automatic Differentiation, Lecture Notes in Computational Science and Engineering, Springer, 2008, pp. 199–210, selected papers from AD2008 Bonn, August 2008.
9. A. Walther, Computing sparse hessians with automatic differentiation, ACM Trans. Math. Softw. 34 (1) (2008) 3:1–3:15. doi:10.1145/1322436.1322439.
URL `http://doi.acm.org/10.1145/1322436.1322439`
10. J. Lotz, K. Leppkes, U. Naumann, dco/c++ – efficient derivative code by overloading in c++, Tech. Rep. AIB-2011-05, RWTH Aachen (2011).
URL `http://aib.informatik.rwth-aachen.de/2011/2011-05.pdf`
11. R. Hartmann, J. Held, T. Leicht, F. Prill, Discontinuous Galerkin methods for computational aerodynamics – 3D adaptive flow simulation with the DLR PADGE code, Aerospace Science and Technology 14 (2010) 512–519. doi:DOI: 10.1016/j.ast.2010.04.002.
12. R. K. Rew, G. P. Davis, S. Emmerson, H. Davies, NetCDF User's Guide for C, An Interface for Data Access, version 3 Edition (April 1997).
13. G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs., SIAM Journal on Scientific Computing 20 (1) (1999) 359 – 392.
14. E. T. Phipps, R. A. Bartlett, D. M. Gay, R. J. Hoekstra, Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation, in: Advances in Automatic Differentiation, Springer, 2008, pp. 351–362.
15. W. Bangerth, R. Hartmann, G. Kanschat, deal.II – a general purpose object oriented finite element library, ACM Trans. Math. Softw. 33 (4) (2007) 24/1–24/27.
16. S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A. M. Bruaset, H. P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.
17. MPI Forum, MPI: A Message-Passing Interface Standard. Version 2.2, available at: `http://www.mpi-forum.org` (Dec. 2009) (September 4th 2009).
18. U. Naumann, L. Hascoët, C. Hill, P. Hovland, J. Riehme, J. Utke, A framework for proving correctness of adjoint message-passing programs, in: Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 316–321. doi:http://dx.doi.org/10.1007/978-3-540-87475-1_44.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

- 2010-01 * Fachgruppe Informatik: Jahresbericht 2010
- 2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
- 2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
- 2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
- 2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
- 2010-06 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
- 2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
- 2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
- 2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
- 2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
- 2010-11 Martin Zimmermann: Parametric LTL Games
- 2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
- 2010-13 Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems
- 2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
- 2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode
- 2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles
- 2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten
- 2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit
- 2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm
- 2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

2011-01 * Fachgruppe Informatik: Jahresbericht 2011

2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

2011-06 Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++

2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV

2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog

2011-09 Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing

2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations

2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains

2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems

2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP

2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games

2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM

2011-17 Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis

2011-18 Kamal Barakat: Introducing Timers to pi-Calculus

2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode

2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations

2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations

2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata

2012-01 Fachgruppe Informatik: Annual Report 2012

2012-02 Thomas Heer: Controlling Development Processes

2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

2012-04 Marcus Gelderie: Strategy Machines and their Complexity

2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting

2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data

2012-07 André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes

2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains

2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs

2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations

2012-16 Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets

2013-01 * Fachgruppe Informatik: Annual Report 2013

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

* These reports are only available as a printed version.

Please contact biblio@informatik.rwth-aachen.de to obtain copies.