AIAA AVIATION Forum
22-26 June 2015, Dallas, TX
16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference

10.2514/6.2015-3240

16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 22 – 26 June 2015, Dallas, Texas

# Development of a Consistent Discrete Adjoint Solver in an Evolving Aerodynamic Design Framework

Tim Albring*, Max Sagebaum† and Nicolas R. Gauger‡
*TU Kaiserslautern, Kaiserslautern, 67663, Germany*



**Typically the development of adjoint solvers for the use in aerodynamic design is challenging. In this paper we will give an update on the development of a discrete adjoint solver that enables the computation of consistent gradients within the open-source multi-physics framework SU2. Due to the use of advanced programming techniques like Expression Templates and the application of Algorithmic Differentiation we obtain an automatic adaption to modifications and extensions of the flow/state solver while maintaining robustness and efficiency.**

## I. Introduction

Efficient aerodynamic optimization methods typically rely on the adjoint methods that were introduced by Jameson[1,2] for aeronautical computational fluid dynamics. He applied the theory of optimal control to the flow equations to get a formulation of the gradient that requires only the solution of the flow equation and the adjoint equation, thereby making it independent of the number of design parameters. Although the continuous version, based on the continuous integral formulation of the flow equations, is quiet efficient in terms of memory consumption and run-time, it is difficult to extend the approach to handle additional equations like turbulence or transition models, for example. Furthermore, there is a slight inconsistency between the calculated gradient and the objective function because the method yields a discrete approximation of the gradient of the analytic objective function, rather than of the discretized objective function. As a result the optimization is likely to fail to converge further once we are near a local minimum. Nevertheless, this approach is used frequently nowadays for shape optimization problems. A possible remedy for the latter problem is the use of discrete adjoint methods[4,5,6] where the optimal control theory is applied to the discretized flow equations. Still, it suffers the same drawback, namely the difficult extension to complex flow models if the traditional approach based on the transposed of the state Jacobian is deployed and it may also lead to consistency problems if the numerical methods are not properly linearized. More on the advantages and disadvantages of either of the methods can be found in the paper of Giles and Pierce.[3]

In this work we will present the current status of the development of an discrete adjoint solver that has the potential to eliminate many drawbacks of past adjoint solvers, while adding new distinguished features. The implementation makes extensive use of advanced techniques of Algorithmic Differentation (AD). Although there were several approaches in the past that applied AD in the field of aerodynamics,[7,8,9] most of them suffered either from poor performance (using AD tools based on the Operator overloading approach) or from limited flexibility (source-code transformation). To overcome these issues we deploy modern C++ features like expression templates[10,11] to automatically generate a representation of the computational graph of each expression at compile-time.

*PhD. Candidate, Chair for Scientific Computing, TU Kaiserslautern, 67663 Kaiserslautern, Germany.
†PhD. Candidate, Chair for Scientific Computing, TU Kaiserslautern, 67663 Kaiserslautern, Germany.
‡Professor, Chair for Scientific Computing, TU Kaiserslautern, 67663 Kaiserslautern, Germany, AIAA Senior Member.

The prime characteristics and goals of this approach are the following:

- **Consistency:** The resulting gradients or sensitivities will be consistent in a discrete sense, meaning that they constitute as the exact derivatives based on the particular numerical methods used in the primal flow solver.
- **Robustness:** The adjoint solver automatically inherits all the features of the primal flow solver. Thus, as long as the primal solver has reached a certain level of convergence, the adjoint solver will also converge, without the need of tweaking any parameters.
- **Efficiency:** We aim for a highly efficient competitive adjoint solver. So far, the factor to get one sensitivity vector compared to one flow solution is at roughly 2.5, without any distinct optimizations. However, there are several techniques available to reduce this overhead in the future.
- **Flexibility:** Due to the use of Algorithmic Differentiation we gain a high level of flexibility that allows for the automatic adaption of the adjoint solver to modifications and extensions of the flow solver. Even the incorporation of turbulence or transition models or grid movement strategies like moving walls and pitching/plunging is possible with little effort as long as they are implemented and ready to be used in the flow solver. Furthermore, in case of aerodynamic design optimizations, it is possible to use any derived flow quantity as the objective function or constraint.

This properties allow essentially the application of this approach to any available flow solver as long as it is written in C or C++. However, it will prove its strength particularly in an constantly evolving multi-disciplinary framework. Therefore we implemented this solver in the SU$^2$ open-source code that comprises a complete self-contained optimization framework for aerodynamic design.[15,16] Initially developed at Stanford University it now exhibits collaborations from all over the world and receives regularly updates and improvements. Figure 1 shows the number of commits per week to the SU2 git repository in the space of time from 05/25/14 to 11/23/14 (i.e. half a year). A maximum of 40 commits and an average of roughly 15 commits clearly highlight the activity of the community. Although some of the commits are just bugfixes, many of them add new features like lately for example real gas support.

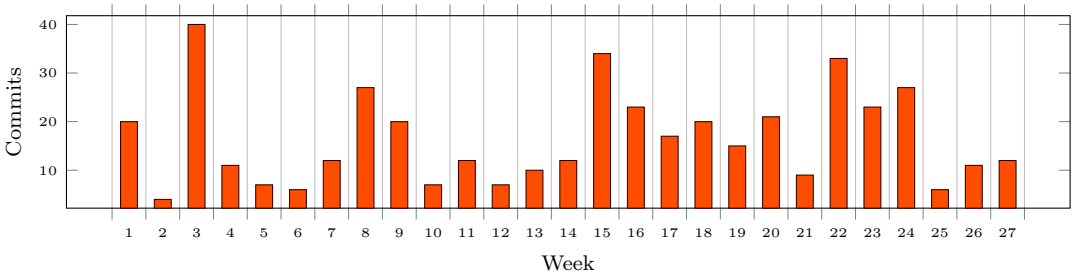



**Figure 1: Number of commits to the SU2 git repository per week from 05/25/14 to 11/23/14**

## II. The Aerodynamic Design Chain

In the following we will give a short overview of the optimization problem and the already implemented methods. Figure 2 show a simplified representation of the aerodynamic design chain. The components of the design vector $\alpha$ can for example be chosen as the amplitudes of Hicks-Henne functions[13] in 2D or as the control points of the Free-Form deformation method[14] for 3D problems. According to a movement of the surface based on the current values of the design variables, a mesh deformation routine using the Linear Elasticity method creates a new mesh $X$. The flow solver then evaluates the flow field $U$ and the objective function $J$. Using this setting, the optimization problem incorporating a steady state constraint can be written as

$$\min_{\alpha} \quad J(U(\alpha), X(\alpha)) \tag{1}$$

$$\text{subject to} \quad R(U(\alpha), X(\alpha)) = 0 \tag{2}$$

where $R(U)$ is the discretized residual. Note that $R(U)$ might not only include the flow residual but also other the residuals of other coupled models, like the turbulence model. Consequently, $U$ might also consists

of the variables of this additional equation. The discretization of the compressible RANS equations and the turbulence models is performed using the Finite-Volume method on a vertex-based median-dual grid. Several numerical fluxes like JST, Roe, AUSM etc. are implemented and flux and slope limiters enable second-order space integration. By using an implicit Euler scheme for the time integration we end up with the following linear system to be solved in each iteration $n$:

$$\left(D^n + \frac{\partial R(U^n)}{\partial U^n}\right) \Delta U^n = -R(U^n). \tag{3}$$

Here, $R(U^n)$ is the residual resulting from the space integration, $\Delta U_i^n := U_i^{n+1} - U_i^n$ and

$$(D^n)_{ij} := \frac{|\Omega_i|}{\Delta t_i^n}\delta_{ij}, \quad \left(\frac{\partial R(U^n)}{\partial U^n}\right)_{ij} := \frac{\partial R_i(U^n)}{\partial U_j^n}. \tag{4}$$

$\Omega_i$ represents the volume of the cell $i$ and $\Delta t_i^n$ is the (pseudo) time-step that may be different in each cell due to the local time-stepping technique. If non-linear multi-grid acceleration[17] is used, then equation (3) is additionally discretized and solved on consecutively coarser meshes in each iteration to find a correction to be applied to $U^{n+1}$.

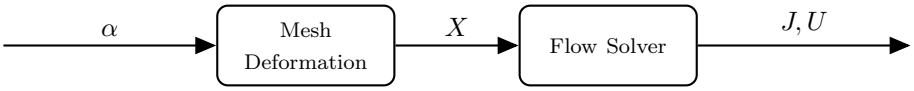



**Figure 2: Simplified representation of the aerodynamic design chain.**

## III. The Discrete Adjoint Solver

An important fact is that if $D^n$ would be zero ($\Delta t_i^n \to \infty$), then (3) would be identical to a step of Newton's method to solve the non-linear system $R(U) = 0$. However, using the implicit Euler discretization we naturally end up with a damped Newton method for solving $R(U) = 0$. Hence, if convergence is achieved, the resulting solution $U^*$ only depends on the right-hand side of (3), that is, the residual $R(U)$. The left-hand side can therefore be any reasonable approximation to the flow Jacobian $\frac{\partial R}{\partial U}$. This can be made clear by transforming the flow equation (3) into a fixed point equation $U = G(U)$, such that feasible solutions can be computed from the iteration

$$U^{n+1} = G(U^n) := U^n - P(U^n)R(U^n) \tag{5}$$

with the preconditioner $P$ defined as

$$P(U) := \left(D + \frac{\partial \tilde{R}(U)}{\partial U}\right)^{-1}. \tag{6}$$

Note, for simplicity the multi-grid method is neglected in this representation of $P$ and $G$. The tilde indicates that this might be an approximation to the exact Jacobian. It is naturally to assume that $G$ is stationary only at feasible points, i.e.

$$R(U^*) = 0 \Leftrightarrow U^* = G(U^*). \tag{7}$$

By the Banach fixed-point theorem, recurrence (5) converges, if $G$ is contractive, i.e. if $\|\frac{\partial G}{\partial U}\| < 1$ in a suitable matrix norm. In advanced CFD codes, like SU$^2$, there are several approximations often seen to reduce the complexity:

- Use of first order approximations of the implicit terms, even though a higher order spatial discretization is applied on the right-hand side.
- Consistent linearized treatment of the boundary conditions is typically neglected.
- Only approximate solutions of the linear system (3) are obtained.

If traditional discrete adjoint methods are used,[3] however, these approximations are not valid anymore, since they result in a linear system involving the exact flow Jacobian $\frac{\partial R}{\partial U}$ to be solved for the adjoint variables. To circumvent this problem, Korivi et al.[12] proposed a method for solving the adjoint system that resembles the iterative flow solver and permits the use of the same approximative Jacobian. For the derivation of the proposed discrete adjoint solver, we adopt this approach and combine it with the efficient evaluation of the occurring gradients using AD.

Since the computational mesh is subject to change, we consider now all functions additionally depended on $X$. To formally handle the surface and mesh deformation, we add it as a constraint to the original optimization problem (1) - (2) by using the equation $M(\alpha) = X$. A similar way of dealing with the mesh sensitivities was originally proposed by Nielsen and Park.[19] However, in the present case we do not make any assumptions on the structure of $M$, except that is differentiable. Then the optimization problem finally takes the form

$$\begin{aligned} \min_{\alpha} \quad & J(U(\alpha), X(\alpha)) && (8)\\ \text{subject to} \quad & U(\alpha) = G(U(\alpha), X(\alpha)) && (9)\\ & X(\alpha) = M(\alpha). && (10) \end{aligned}$$

We can define the Lagrangian associated to this problem as

$$\begin{aligned} L(\alpha, U, X, \bar{U}, \bar{X}) &= J(U, X) + (G(U, X) - U)^T \bar{U} + (M(\alpha) - X)^T \bar{X} && (11)\\ &= N(U, \bar{U}, X) - U^T \bar{U} + (M(\alpha) - X)^T \bar{X} && (12) \end{aligned}$$

where $N$ is the shifted Lagrangian

$$N(U, \bar{U}, X) := J(U, X) + G^T(U, X)\bar{U}. \tag{13}$$

If we differentiate $L$ with respect to $\alpha$ using the chain rule, we can choose the adjoint variables $\bar{X}$ and $\bar{U}$ in such a way, that the terms $\frac{\partial U}{\partial \alpha}$ and $\frac{\partial X}{\partial \alpha}$ can be eliminated. This leads to the following equations for $\bar{U}$ and $\bar{X}$:

$$\begin{aligned} \bar{U} &= \frac{\partial}{\partial U} N(U, \bar{U}, X) = \frac{\partial}{\partial U} J^T(U, X) + \frac{\partial}{\partial U} G^T(U, X)\bar{U} && (14)\\ \bar{X} &= \frac{\partial}{\partial X} N(U, \bar{U}, X) = \frac{\partial}{\partial X} J^T(U, X) + \frac{\partial}{\partial X} G^T(U, X)\bar{U} && (15) \end{aligned}$$

Finally, the derivative of the Lagrangian, that is, the total derivative of $J$, reduces to

$$\frac{\mathrm{d}L}{\mathrm{d}\alpha}^T = \frac{\mathrm{d}J}{\mathrm{d}\alpha}^T = \frac{\mathrm{d}}{\mathrm{d}\alpha} M^T(\alpha)\bar{X}. \tag{16}$$

Equation (14) is a fixed-point equation in $\bar{U}$ and can be solved in the style of the flow solver using the iteration

$$\bar{U}^{n+1} = \frac{\partial}{\partial U} N(U^*, \bar{U}^n, X) \tag{17}$$

once we have found a numerical solution $U = U^*$ of equation (5). Since $G$ is a contractive function if the flow solver has reached a certain level of convergence (i.e. $\|\frac{\partial G}{\partial U}\| < 1$ in some suitable matrix norm), also $\frac{\partial N}{\partial U}$ will be contractive since

$$\left\| \frac{\partial}{\partial \bar{U}} \left( \frac{\partial N^T}{\partial U} \right) \right\| = \left\| \frac{\partial G^T}{\partial U} \right\| = \left\| \frac{\partial G}{\partial U} \right\| < 1. \tag{18}$$

Thus, it directly inherits the convergence properties of the flow solver. Up to now the derivation of the discrete adjoint solver was rather abstract as we did not specify yet how to compute the necessary gradients. In turns out that the sensitivity equation (15) and equation (17) can efficiently be evaluated using Algorithmic Differentiation (see the next section) applied to the underlying routines in the program that compute $G$. Thereby we can consider $G$ as a black-box, thus making the development and implementation of the discrete adjoint solver independent of the particular implementation of $G$.

# IV. Algorithmic Differentiation

*Algorithmic Differentiation* or also called *Automatic Differentiation* is a frequently used method to calculate the derivative of a function by means of the transformation of the underlying program which calculates the numerical values of this function. As distinguished from symbolic differentiation an explicit expression for the derivative is never formed. An advantage over FD is that no truncation errors are present, thus the numerical value can be determined up to machine accuracy. Typically AD is introduced based on the observation that every code is merely a sequence of elementary functions that depend on one or two variables. Although this assumption leads to an intuitive approach for the implementation it is rather inefficient as we will see in subsection IV.C. Therefore we will introduce the concept of Expression Templates to present a more efficient way of implementing the reverse mode as well as the forward mode of AD.

Let us suppose we have a function $f \in C^1 : U \subset \mathbb{R}^n \to \mathbb{R}^m$ and the mathematical expression

$$y = f(x), \tag{19}$$

that is, $x \in U, y \in \mathbb{R}^m$. We assume that the numerical evaluation of $f$ can be represented as a sequence of $l$ elementary functions $\varphi_i \in \mathbb{R}$ and intermediate values $v_i \in \mathbb{R}$ like it is shown in table 1. The precedence relation $j \prec i$ means that $i$ depends directly on $j$. Furthermore we have $u_i = (v_j)_{j \prec i} \in \mathbb{R}^{n_i}$, i.e. the vector $u_i$ is the concatenation of the $v_j$ on which $\varphi_i$ depends. The $\varphi_i$ can be thought of as operations occurring in statements of a computer program that evaluates $f$. Typically we have $n_i = 1$ (unary operations like for example square root, i.e. $\varphi_i(v_k) = \sqrt{v_k}$, $k \prec i$) or $n_i = 2$ (binary operations like summation or multiplication, i.e. $\varphi_i(v_k, v_l) = v_k \cdot v_l$, $k, l \prec i$).

| | | | |
|---|---|---|---|
| $v_{i-n}$ | $= x_i,$ | $i =$ | $1 \dots n$ |
| $v_i$ | $= \varphi_i(v_j)_{j \prec i},$ | $i =$ | $1 \dots l$ |
| $y_{m-i}$ | $= v_{l-i},$ | $i =$ | $m-1 \dots 0$ |

**Table 1: Evaluation procedure for a function $f$.**

## IV.A. Forward Mode

By applying the chain rule to the elementary functions in table 1 we can augment each statement with its corresponding derivative, which gives the forward mode of AD shown in table 2. Consequently we can get any matrix-vector product involving the Jacobian of $f$ and an arbitrary vector $\dot{x}$, i.e. $\dot{y} = \frac{\mathrm{d}f(x)}{\mathrm{d}x}\dot{x}$, by evaluating table 2.

| | | | |
|---|---|---|---|
| $[v_{i-n}, \dot{v}_{i-n}]$ | $= [x_i, \dot{x}_i]$ | $i =$ | $1 \dots n$ |
| $[v_i, \dot{v}_i]$ | $= [\varphi_i(v_j)_{j \prec i}, \sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(u_i) \dot{v}_j]$ | $i =$ | $1 \dots l$ |
| $[y_{m-i}, \dot{y}_{m-i}]$ | $= [v_{l-i}, \dot{v}_{l-i}]$ | $i =$ | $m-1 \dots 0$ |

**Table 2: Forward propagation of gradients.**

## IV.B. Reverse Mode

For the forward mode the derivatives are propagated in the same direction as the corresponding elemental function values, i.e. we ask how a infinitesimal change in the input values propagates through the evaluation trace and affects the output. But we could also ask the other way round: How sensitive are the output values to a change in the input values? The latter is the basis for the reverse propagation. There are many ways in which the reverse mode can be introduced, but in general it can be thought of as the backward application of the chain rule. For a detailed introduction we refer to relevant literature.[20]

Based on the representation of $f$ as a sequence of elementary functions, we can also write it as the composition

$$y = Q_m \Phi_l(\Phi_{l-1}(\dots(\Phi_1(P_n^T x)))) \tag{20}$$

where the state transformation $\Phi_i$ sets $v_i$ to $\varphi_i(v_j)_{j\prec i}$ and keeps all other components $v_j$ for $i \neq j$ unchanged. $P_n \in \mathbb{R}^{n\times(n+l)}$ and $Q_m \in \mathbb{R}^{m\times(n+l)}$ are the matrices that project an $(n+l)$-vector onto its first $n$ and last $m$ components, respectively. By using $A_i := \nabla\Phi_i$ and application of the chain rule for differentiation we get

$$\dot{y} = Q_m A_l A_{l-1} \dots A_2 A_1 P_n^T \dot{x}. \tag{21}$$

Thus, the Jacobian of $f$ can be written as

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = Q_m A_l A_{l-1} \dots A_2 A_1 P_n^T \tag{22}$$

By transposing the product we obtain the adjoint relation

$$\bar{x} = P_n A_1^T A_2^T \dots A_{l-1}^T A_l^T Q_m^T \bar{y} = \left(\frac{\mathrm{d}f}{\mathrm{d}x}\right)^T \bar{y} \tag{23}$$

and the identity

$$\bar{y}\dot{y} = \bar{x}\dot{x}. \tag{24}$$

If one looks into detail at some intermediate matrix multiplication in (23), it can be seen, that it is possible to write the adjoint relation as an evaluation procedure similar to the forward evaluation. All matrix-vector products are calculated for $i = l, l-1, \dots, 1$ thus we have to go backward, or reverse, through the sequence of elementary functions in table 1. Since the intermediate values $v_i$ are needed they have to be computed first. Again, we can then get the matrix-vector product involving the transposed of the Jacobian and an arbitrary vector $\bar{y}$, i.e. equation (23), by evaluating table 3.

| | | | |
|---|---|---|---|
| $v_{i-n}$ | $= x_i,$ | $i =$ | $1 \dots n$ |
| $v_i$ | $= \varphi_i(v_j)_{j\prec i},$ | $i =$ | $1 \dots l$ |
| $y_{m-i}$ | $= v_{l-i},$ | $i =$ | $m-1 \dots 0$ |
| $\bar{v}_{l-i}$ | $= \bar{y}_{m-i}$ | $i =$ | $0 \dots m-1$ |
| $\bar{v}_j$ | $= \bar{v}_j + \bar{v}_i \frac{\partial}{\partial v_j}\varphi_i(u_i), j \prec i,$ | $i =$ | $l \dots 1$ |
| $\bar{x}_i$ | $= \bar{v}_{i-n},$ | $i =$ | $n \dots 1$ |

**Table 3: Reverse propagation of gradients.**

### IV.C. Naive Implementation

A naive way of implementing a forward mode tool in C++ is to simply create a new datatype that stores $x$ and $\dot{x}$. An exemplary implementation is shown in listing 1. Using the facility to overload operators we can then compute the value $\dot{y}$ along with the computation of $y$ by replacing all floating point variables involved in the evaluation of $f$ with this new datatype. To see the drawbacks of this naive approach consider the case where the computer program that evaluates $f(x)$ consists of just one expression, for example $y = f(x_1, x_2, x_3) = \cos(x_1)x_2$ (listing 2).

```
1  class adfloat{
2    float val;
3    float dot;
4  }
```

**Listing 1: Class adfloat for forward propagation**

```
1  adfloat y, x_1, x_2;
2
3  // ... Initialization of x_1, x_2
4  y = cos(x_1) * x_2;
```

**Listing 2: Expression to evaluate $f$**

We assume that we have implemented the overloaded operators for forward propagation of gradients for the multiplication and the cosine:

```
adfloat operator*(adfloat a, adfloat b){
  adfloat tmp;
  tmp.val = a.val + b.val;
  tmp.dot = b.val * a.dot + a.val * b.dot;
  return tmp;
}
```

**Listing 3: Overloaded multiplication operator for forward propagation**

```
adfloat cos(adfloat a){
  adfloat tmp;
  tmp.val = cos(a.val);
  tmp.dot = -sin(a.val)*a.dot;
  return tmp;
}
```

**Listing 4: Overloaded cos operator for forward propagation**

To compute $\dot{y}$ we can then simply initialize the `x_1.val` and `x_2.val` fields with appropriate values and evaluate the expression in listing 2. The implementation of a simple reverse mode tool is similar. But instead of storing the associated $\bar{\cdot}$ values in the datatype itself only an index that identifies this value is stored. Furthermore we have a stack `operation_stack` that stores the kind of operation and a stack `index_stack` that stores the indices of the arguments of the operations. Then the implementation of the overloaded operators for the reverse propagation of gradients would be as follows:

```
adfloat operator*(adfloat a, adfloat b){
  adfloat tmp;
  indexcount += 1;
  tmp.index = indexcount;
  operation_stack.push(MULT);
  index_stack.push(a.index);
  index_stack.push(b.index);
  index_stack.push(tmp.index);
  tmp.val = a.val*b.val;
  return tmp;
}
```

**Listing 5: Overloaded multiplication operator for reverse propagation**

```
adfloat cos(adfloat a){
  adfloat tmp;
  indexcount += 1;
  tmp.index = indexcount;
  operation_stack.push(COS);
  index_stack.push(a.index);
  index_stack.push(tmp.index);
  tmp.val = cos(a.val);

  return tmp;
}
```

**Listing 6: Overloaded cos operator for reverse propagation**

`MULT` and `COS` are integer variables to identify the operations. Using appropriate values for the $\bar{\cdot}$ entries of $x_1$ and $x_2$ in a second step we can then traverse through the recorded stack of operations and accumulate the $\bar{\cdot}$ values according to table 3.

Although this approach is very intuitive and very flexible, the performance of this implementation in comparison with hand-crafted C code (and thus with the source code transformation approach) is bad due to the creation of the temporary variables `tmp` in listings 3, 4, 5 and 6. Furthermore the increased memory footprint of each operation might be interfering with the cache locality. The performance loss is even worse if we have several operations in one expression due to the "greedy" evaluation where for each single operation a separate temporary `adfloat` variable is created. For example in listing 2 the compiler will create a temporary variable storing the result of `cos(x_1)` before calling the multiplication operator. This temporary variable essentially corresponds to the intermediate values $v_i$ in the evaluation procedure in table 1. Due to this drawbacks this approach is unsuitable for the use in large scale aerodynamic design.

### IV.D. Expression Templates

On the mathematical level we can also apply the chain rule to the whole expression. We essentially make the $\varphi_i$ more general, such that they may now depend on an arbitrary number of inputs and other functions (or expressions). However, writing down a closed mathematical form for arbitrary expressions can be very involved. Instead we consider the example from listing 2. We can write $f(x_1, x_2)$ as the composition of a function $h$ and $g$ such that

$$y = f(x_1, x_2) = h(g(x_1), x_2) = \cos(x_1)x_2.$$

Application of the forward mode yields

$$\dot{y} = \frac{\partial h}{\partial g}\frac{\partial g}{\partial x_1}\dot{x}_1 + \frac{\partial h}{\partial x_2}\dot{x}_2 = -\sin(x_1)x_2\dot{x}_1 + \cos(x_1)\dot{x}_2, \tag{25}$$

and application of the reverse mode gives

$$\bar{x}_1 \quad += \quad \frac{\partial h}{\partial g}\frac{\partial g}{\partial x_1}\bar{y}, \tag{26}$$

$$\bar{x}_2 \quad += \quad \frac{\partial h}{\partial x_2}\bar{y}. \tag{27}$$

Figure 3(a) shows the evaluation of $f$ as a computational graph. The Expression Template[10,11] approach is to create a compile-time parse tree of the whole expression using C++ meta-programming techniques to remove the creation of the costly temporary objects entirely and to delay the execution of the expression until it is assigned to its target. Therefore the overloaded operators no longer return the (computationally expensive) result of an expression, but a small temporary object that acts as a placeholder for this particular expression. Using this objects we can build an internal representation of each expression like shown in figure 3(b) to directly compute $\dot{y}$ (equation (25)) or to store the necessary information to compute $\bar{x}_1$ and $\bar{x}_2$ (equations (26)-(27)). Below you find an exemplary implementation of the multiplication expression:

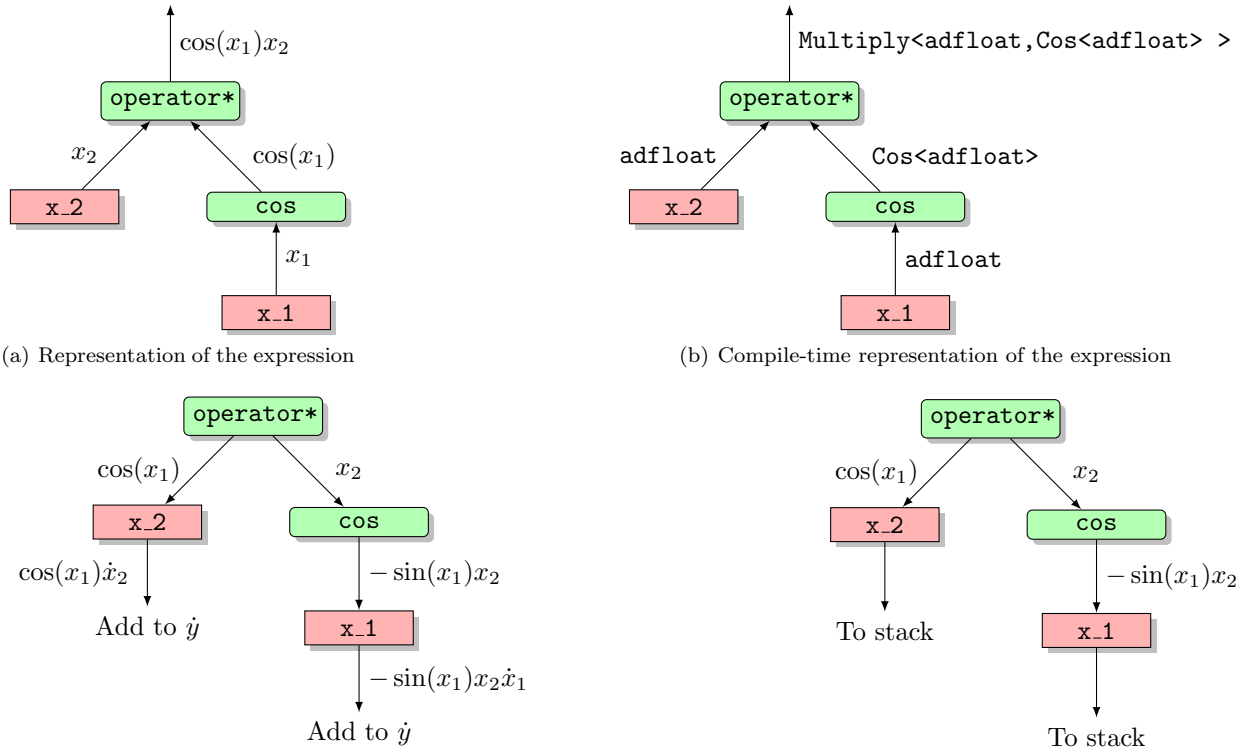
```
1  template< typename A, typename B >
2  class Multiply
3  {
4  public:
5    Multiply (const A& a, const B& b) : a_( a ), b_( b ), value(a.value*b.value)) {}
6
7  // Forward mode version
8  void calc_gradient(double &gradient, const double& multiplier){
9    a_.calc_gradient(gradient, b_.value*multiplier);
10   b_.calc_gradient(gradient, a_.value*multiplier);
11 }
12 // Reverse mode version
13 void calc_gradient(int &index, const double& multiplier){
14   a_.calc_gradient(index, b_.value*multiplier);
15   b_.calc_gradient(index, a_.value*multiplier);
16 }
17   float value;
18 private:
19   const A& a_;
20   const B& b_;
21 };
22
23 template< typename A, typename B>
24 Multiply<A,B> operator*(const A& a, const B& b){
25   return Multiply<A,B>(a,b);
26 }
```

Instead of calculating the result of the multiplication of two `adfloat` variables, the multiplication operator now returns an object of type `Multiply<A,B>`, where A and B are the types of the left- and right-hand side operands, respectively. The only requirements the multiplication operator poses on A and B are the existence of a `calc_gradient` routine and a variable `value`. The Multiply class has two additional data members, which are references-to-const to the two operands of the multiplication operation. Therefore this object is cheap to create and copy in comparison to the `adfloat` variable. Note that all of the operators will essentially look similar to this Multiply class except that the implementation of the `calc_gradient` routine differs. Once we then encounter an assignment operator in the code we start handing over the current gradient value (forward mode) or the index (reverse mode) of the left-hand side value to the expression object (here `Multiply<adfloat, Cos<adfloat> >`) on the right-hand side using the `calc_gradient` routine. This expression then sends its partial derivative to its arguments as it is shown in figure 3(c) for the forward mode and in figure 3(d) for the reverse mode. Each expression itself then multiplies the received derivative value with its partial derivative and sends it further down such that we have the partial derivatives of the right-hand side with respect to its inputs when we reach the `adfloat` variable. There we will either add the derivatives to the gradient of the left-hand side for the forward mode, or we push the values to a stack to accumulate the gradients as in equations (26)-(27) in a second step. An important fact is that the compiler can inline almost all calls of the `calc_gradient` routines such that the performance will be similar to plain C-code.

The overloaded datatype and the stack handling is implemented in the open-source differentiation tool CoDiPack developed at the Chair for Scientific Computing at TU Kaiserslautern. The differentiated MPI routines for the reverse mode of AD are provided by the AdjointMPI library.[21]

## V. Implementation in SU2

SU2 is an open-source, integrated analysis and design tool for solving multi-disciplinary problems governed by partial differential equations (PDEs) on general, unstructured meshes written in C++. Due its object-oriented structure the application of source-code transformation to generate the needed derivatives in section III is challenging, except for small enclosed parts of the code. However, even in this case we have to manually modify the discrete adjoint solver whenever there is a modification or extension of the flow solver in order




(a) Representation of the expression

(b) Compile-time representation of the expression

(c) Propagation of gradients in forward mode

(d) Propagation of gradients in reverse mode

**Figure 3: Computational graphs for the example expression $y = \cos(x_1)x_2$.**

to maintain its consistency. On the contrary using the Operator Overloading approach in combination with Expression Templates detailed in section IV.D there are no modifications necessary in the adjoint solver since the generation of the derivatives is implicitly performed during the evaluation of the expressions in the code.

The implementation can be eased by the fact that we can get all necessary derivatives during one evaluation of the flow solver iteration $G$ defined in equation (5). Therefore consider the extended iteration

$$\begin{pmatrix} V \\ W \end{pmatrix} = \begin{pmatrix} G(U,X) \\ J(U,X) \end{pmatrix} \tag{28}$$

that represents one iteration of the flow solver and the immediate evaluation of the objective function at an arbitrary flow field $U$. If we apply the reverse mode of AD to equation (28) we get

$$\begin{pmatrix} \bar{U} \\ \bar{X} \end{pmatrix} = \begin{pmatrix} \frac{\partial G(U,X)}{\partial U} & \frac{\partial G(U,X)}{\partial X} \\ \frac{\partial J(U,X)}{\partial U} & \frac{\partial J(U,X)}{\partial X} \end{pmatrix}^T \begin{pmatrix} \bar{V} \\ \bar{W} \end{pmatrix} = \begin{pmatrix} \frac{\partial G(U,X)}{\partial U}^T & \frac{\partial J(U,X)}{\partial U}^T \\ \frac{\partial G(U,X)}{\partial X}^T & \frac{\partial J(U,X)}{\partial X}^T \end{pmatrix} \begin{pmatrix} \bar{V} \\ \bar{W} \end{pmatrix} \tag{29}$$

Equation (29) means that if we set the $\bar{\cdot}$ values of the output of $G$ and $J$ to $\bar{V}$ and $\bar{W}$, respectively, we will find the values for $\bar{U}$ and $\bar{X}$ in the $\bar{\cdot}$ fields of the inputs of $G$ and $J$ after we have evaluated table 3. Note that the choice of $\bar{V}$ and $\bar{W}$ is arbitrary at this point. If we choose $\bar{W} \equiv 1$ we get

$$\begin{pmatrix} \bar{U} \\ \bar{X} \end{pmatrix} = \begin{pmatrix} \frac{\partial G(U,X)}{\partial U}^T & \frac{\partial J(U,X)}{\partial U}^T \\ \frac{\partial G(U,X)}{\partial X}^T & \frac{\partial J(U,X)}{\partial X}^T \end{pmatrix} \begin{pmatrix} \bar{V} \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{\partial G(U,X)}{\partial U}^T \bar{V} + \frac{\partial J(U,X)}{\partial U}^T \\ \frac{\partial G(U,X)}{\partial X}^T \bar{V} + \frac{\partial J(U,X)}{\partial X}^T \end{pmatrix} \tag{30}$$

The right-hand side is nothing but the gradient of the shifted Lagrangian $N$ defined in equation (13) evaluated at $(U, \bar{V}, X)$. This means we can easily perform iteration (17) by storing the computational graph (i.e.

the information to accumulate the $\bar{\cdot}$ values in table 3) of $G(U^*, X)$ once at $n = 0$ using the Expression Template approach. Starting with some initial $\bar{U}^0$ in each subsequent iteration we can then evaluate the stored information with $\bar{V}$ set to the previous adjoint solution $\bar{U}^n$ and $\bar{W} \equiv 1$ in equation (29) to compute $\frac{\partial N}{\partial U}(U^*, \bar{U}^n, X)$. The iteration is then performed until $\|\bar{U}^n\|$ is sufficiently small. The mesh sensitivity equation (16) is then solved by applying the reverse mode of AD to the mesh deformation routine.

This approach yields a very flexible implementation as we only need access to the input/output variables of the solver (or rather of the iteration function $G$). Based on this it is in principle possible to use any available iterative coupled solver in SU2 as a constraint for the optimization. Furthermore a lot of the already present input/output routines, data structures and optimization methods can be reused for the discrete adjoint solver, such that it will integrate smoothly into the existing framework.

# VI. Results

## VI.A. Inviscid 3D

The implementation of the general framework for the discrete adjoint is still ongoing, such that we will present some results of the current stage of development using the Euler equations including some remarks on the direction of current and future research. The test case features the drag reduction of the ONERA M6 airfoil in transonic, inviscid flow. The free-stream Mach number is set to $Ma_\infty = 0.8395$ with an angle of attack of 3.06°. The computational domain is discretized using an unstructured grid with 582,752 elements, with the surface consisting of 19,894 triangles. In order to model an extension, the airfoil is attached to a symmetry boundary. The convective flux is discretized using the JST scheme.

We measured the performance of the discrete adjoint solver using different numbers of processors and nodes. Each of the nodes features 16 Intel Xeon E5-2670 processors. The communication is performed using the OpenMPI library. Figure 4 shows the measured speed up and the parallel efficiency. The average time for one iteration of the flow and the discrete adjoint solver was taken as the basis. For 2 and 4 processors the discrete adjoint solver exhibits superlinear behaviour (parallel efficiency $> 100\%$) compared to the flow solver. Why this is the case would be just speculation at the moment and needs more attention in the future. If more processors are used the discrete adjoint solver matches the values of flow solver, meaning that it inherits its parallel performance. Whether this behavior holds also for other cases will be the subject of related projects. Since 32 processors offer a reasonable speed up and efficiency in both cases, this number was used in the remainder of this section.

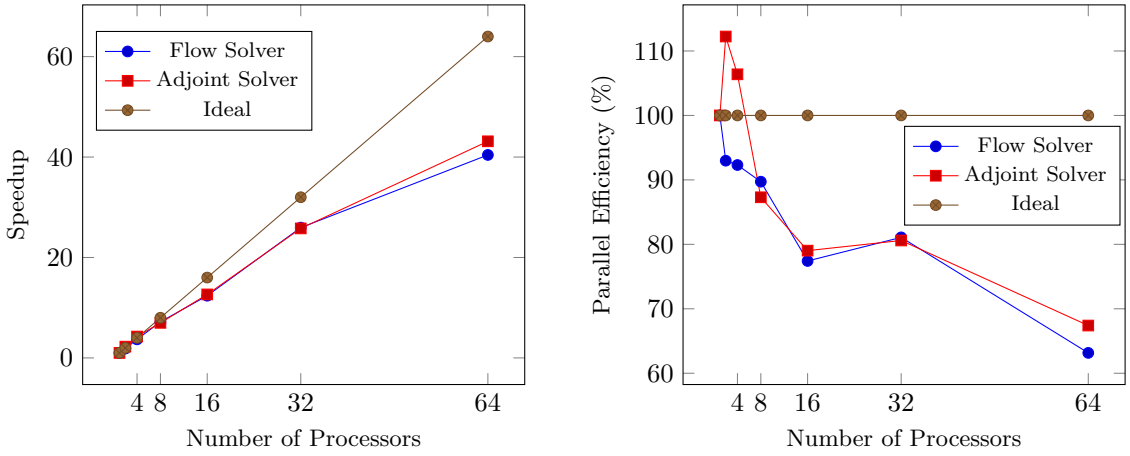
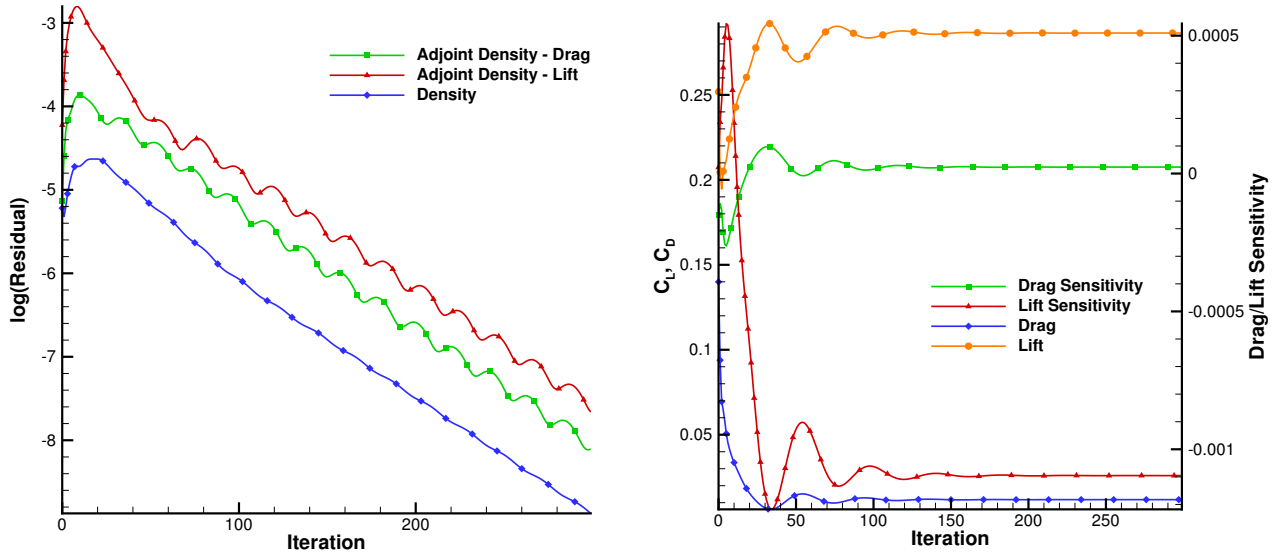



**Figure 4: Speedup (left) and parallel efficiency (right) using the OneraM6 test case.**

Figure 5(a) shows the root-mean square residual of the flow and adjoint solver for the lift and drag objective function. In section III we have seen that the convergence rate of the discrete adjoint solver is in theory equal or at least similar to the convergence rate of the flow solver. That this is indeed the case is verified here. To achieve the same reduction in the residual of the flow and adjoint we need therefore the same number of iterations. However, further studies are necessary to investigate the behavior of the adjoint solver when the flow residual is still quite high.

For the comparison of the run-time we also compare one iteration of the flow solver with one iteration of




(a) History of the RMS residuals

(b) History of $C_D$, $C_L$ and the corresponding sensitivities.

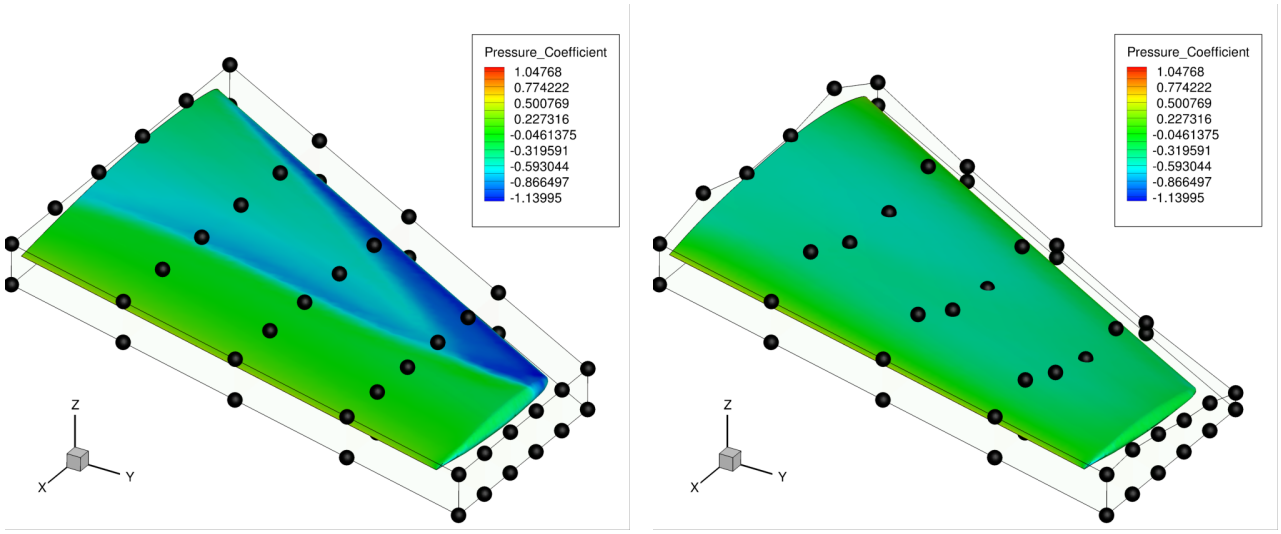
**Figure 5: Convergence history for the OneraM6 test case.**

the discrete adjoint solver averaged over 100 iterations. With the current implementation we get a factor of

$$\frac{t_{adjoint}}{t_{flow}} \approx 2.3. \tag{31}$$

There are several techniques available to further improve the performance, including the explicit (hand) differentiation of critical routines like matrix vector products etc. Note that due to the fact that we store the computational graph the memory consumption will raise by approximately a factor of 10-12. However, we noticed that often memory does not pose as a problem as modern computing clusters typically have a memory per core ratio of about 4GB (the present adjoint computation used 13 GB in total, thus 13 GB /32 = 0.4 GB per core). Since the load (the number of elements) is evenly distributed among the processes, using the current implementation and the Euler solver each process could handle a maximum number of elements in the order of $\mathcal{O}(10^5)$ (if we assume a linear growth in memory requirement). Since this is just an assumption we will conduct large scale computations very soon to verify this.

The baseline wing has a strong shock structure appearing on the upper surface of the airfoil that extends from the leading edge and is visible in the contour plot of the pressure distribution in figure 6(a) which makes it an attractive test case for the drag reduction. The FFD control points are distributed as also shown in figure 6(a). Note, for the optimization only a subset of the control points on the upper surface is used and only a movement in the $z$-direction is allowed. To be more precise, the last row of control points near the trailing edge is fixed, resulting in a total of 25 design variables. To validate the solver we compare in figure 7 the drag sensitivity vector with finite differences and the continuous adjoint method available in SU2. The discrete adjoint and the continuous adjoint method match quite well. The finite difference approximation shows discrepancies at the points that are near the trailing edge of the wing. This might due to the discontinuity (Kutta condition) that exists at the trailing edge that leads to a non-differentiability of the objective function. For the discrete and continuous adjoint this problem is circumvented by neglecting the contribution of the first few cells near the trailing edge to the final sensitivity vector. In 2D this is implicitly avoided by choosing Hicks-Henne functions for the parameterization of the design space since they are zero at the trailing edge. For this case we get perfect agreement with finite differences.[24] The lift is constraint to remain larger than or equal to the baseline value of $C_L = 0.2864$. Figure 8 shows the development of the drag and lift coefficient during the optimization. After 11 design cycles the process has reached a state where significant improvements are not noticeable anymore resulting in a reduction of 35% compared to the initial value. The lift has not changed and remained essentially at the constraint initial value. The optimization successfully removed the shock, leaving a continuous pressure distribution over the surface (see figure 6(b)).




(a) Baseline design (b) Optimized design

**Figure 6: Pressure contours and distribution of the FFD control points.**

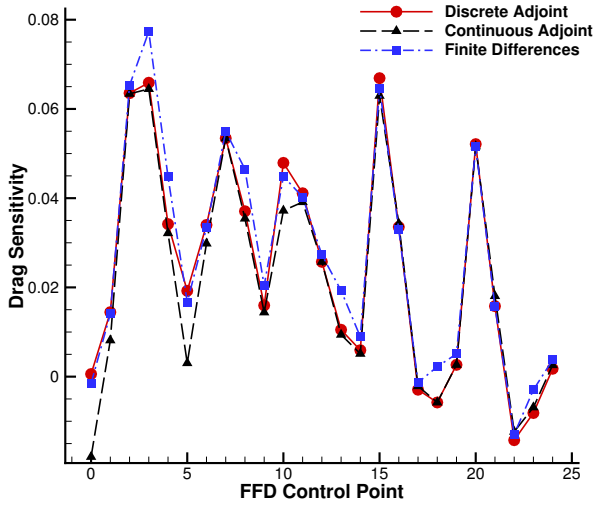



**Figure 7: Drag sensitivity comparison with finite differences and the continuous adjoint method.**

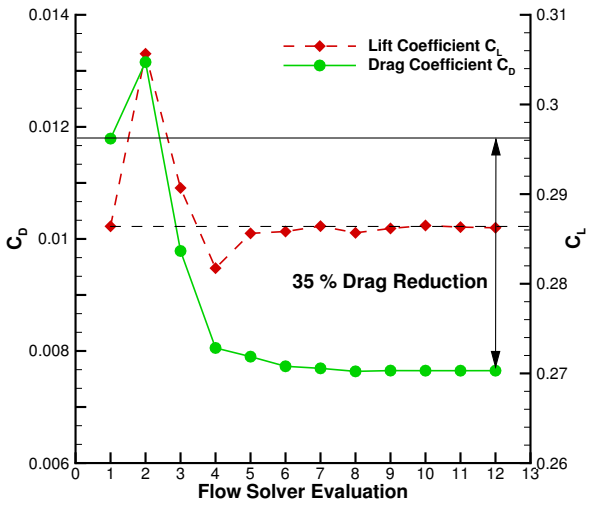



**Figure 8: Lift and drag coefficients during the optimization.**

## VII. Conclusion and Outlook

In this work we presented the status of the development of a discrete adjoint solver based on Algorithmic Differentiation. While we only showed results for a steady Euler case, the solver was recently also applied for the consistent computation of gradients in turbulent as well as in unsteady flows.[24,25] The presented approach offers the flexibility to easily incorporate other additional models that are available in the design suite (like FSI, reaction terms, mesh movement and many more in the future). Once the development of the general framework has been completed it will maintain its consistency automatically even in a constantly evolving environment like SU2. While the first results and the performance look promising there are still aspects that need further attention. We have seen that for the presented test case the parallel performance of the adjoint solver is inherited from the flow solver. However, whether this holds in general is currently investigated. Furthermore, the convergence rate of the adjoint solver depends on the contraction factor of the flow solver iteration. If this factor is sufficiently small, the adjoint solver must converge as we have shown. Still, the notion of "sufficiently small" needs to be quantified somehow for general applications. Finally, we need to apply the solver to additional test cases to further verify the robustness and efficiency.

The base version of the framework, including adjoint solver for Euler, Navier-Stokes and (U)RANS equations, will be made available to the public and will be included in the next major release of SU2.

## References


[1]Anthony Jameson. 'Aerodynamic design via control theory.' J. Sci. Comput., 3:233–26-, 1988.

[2]Anthony Jameson and J. Reuther. 'Control theory based airfoil design using the Euler equations.' AIAA Paper 94-4272-CP, 1994.

[3]Michael B. Giles and Niles A. Pierce. 'An introduction to the adjoint approach to design'. Flow, Turbulence and Combustion, 65(3-4):393-415, 2000.

[4]Eric J. Nielsen, 'Aerodynamic design optimization on unstructured meshes using the Navier-Stokes equations'. AIAA Paper 98-4809, 1998.

[5]Jonathan Elliott and Jaime Peraire. 'Practical 3D aerodynamic design and optimization using unstructured meshes'. AIAA J.35 (9) 14791485, 1997.

[6]Jonathan Elliott and Jaime Peraire. 'Aerodynamic design using unstructured meshes'. AIAA Paper 96-1941, 1996.

[7]Bijan Mohammadi. 'Optimal shape design, reverse mode of automatic differentiation and turbulence'. AIAA Paper 97-0099, 1997.

[8]Paul Hovland, Bijan Mohammadi and Christian Bischof. 'Automatic Differentiation and Navier-Stokes Computations'. Computation Methods for Optimal Design and Control, Birkhauser, Basel, pp. 265–284, 1998.

[9]Nicolas R. Gauger, Andrea Walther, Carsten Moldenhauer and Markus Widhalm. 'Automatic Differentiation of an Entire Design Chain for Aerodynamic Shape Optimization', Notes on Numerical Fluid Mechanics and Multidisciplinary Design, Vol. 96, pp. 454-461, 2007.

[10]Robin J. Hogan, 'Fast reverse-mode automatic differentiation using expression templates in C++'. ACM Trans. Math. Softw., 40, 26:1-26:16, 2014.

[11]Jochen Härdtlein. 'Moderne Expression Templates Programmierung - Weiterentwickelte Techniken und deren Einsatz zur Lösung partieller Differentialgleichungen', PhD thesis, University of Erlangen-Nuremberg, 2007.

[12]Vamshi M. Korivi and Arthur C. Taylor. 'An incremental strategy for calculating consistent discrete CFD sensitivity derivatives', National Aeronautics and Space Administration, Langley Research Center, 1992.

[13]Raymond M. Hicks and Preston A. Henne, 'Wing Design by Numerical Optimization'. Journal of Aircraft, 15:407–412, 1978.

[14]Thomas W. Sederberg and Scott R. Parry. 'Free-form deformation of solid geometric models'. SIGGRAPH Comput. Graph. 20, 151-160, 1986.

[15]Francisco Palacios, Juan Alonso, Karthikeyan Duraisamy et al. 'Stanford University Unstructured (SU2): An Open-Source Integrated Computational Environment for Multi-Physics Simulation and Design.' In 51st AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition. Aerospace Sciences Meetings. American Institute of Aeronautics and Astronautics, 2013.

[16]Francisco Palacios, Thomas Economon, Aniket Aranake, Sean R. Copeland, Amrita K. Lonkar, Juan Alonso et al. 'Stanford University Unstructured (SU2): Open-source analysis and design technology for turbulent flows.' AIAA Paper 2014-0243, AIAA Science and Technology Forum and Exposition 2014: 52nd Aerospace Sciences Meeting, National Harbor, MD, January 13-17, 2014.

[17]Alfio Borzi. 'Introduction to multigrid methods'. 2005.

[18]Anil Nemili, Emre Özkaya, Nicolas R. Gauger, Felix Kramer, Angelo Carnarius and Frank Thiele. 'Discrete Adjoint based Sensitivity Analysis for Optimal Flow Control of a 3D High-Lift Configuration'. AIAA 2013-2585, 2013.

[19]Eric J. Nielsen and Michael A. Park. 'Using An Adjoint Approach to Eliminate Mesh Sensitivities in Computational Design.', AIAA Journal, Vol. 40, No. 5, 2006.

[20]Andreas Griewank and Andrea Walther. 'Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.', Other Titles in Applied Mathematics. SIAM, 978–0–898716–59–7., 2008.

[21]Michel Schanen, Uwe Naumann, Laurent Hascoët and Jean Utke. 'Interpretative Adjoints for Numerical Simulation Codes using MPI', Procedia Computer Science, Volume 1, Issue 1, Pages 1825-1833, 2010.

[22]P.H. Cook, M.A. McDonald and M.C.P. Firmin, "Aerofoil RAE 2822 - Pressure Distributions, and Boundary Layer and Wake Measurements," Experimental Data Base for Computer Program Assessment, AGARD Report AR 138, 1979

[23]Dieter Kraft 'A software package for sequential quadratic programming.' Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace Center Institute for Flight Mechanics, Koln, Germany, 1988.

[24]Tim Albring, Beckett Y. Zhou, Nicolas R. Gauger, Max Sagebaum. 'An Aerodynamic Design Framework based on Algorithmic Differentiation', ERCOFTAC Bulletin, 102 pp. 10-16, 2015.

[25]Beckett Y. Zhou, Tim Albring, Nicolas R. Gauger, Thomas D. Economon, Francisco Palacios, Juan J. Alonso. 'A Discrete Adjoint Framework for Unsteady Aerodynamic and Aeroacoustic Optimization'. AIAA Aviation Forum 2015, Dallas, 2015.