AIAA AVIATION Forum
13-17 June 2016, Washington, D.C.
17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference
10.2514/6.2016-3518

17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 13 – 17 June 2015, Washington, D.C.

Efficient Aerodynamic Design using the Discrete Adjoint Method in SU2

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

I. Introduction

Almost 30 years have passed since Jameson[1] discussed the success and challenges in CFD and listed optimization and design as one the directions for future research. Unfortunately, computational methods for aerodynamic analysis are even today far from being incorporated in automatic design procedures. Although adjoint methods[2,3,4] have greatly decreased the computational effort and impressive results using the continuous[5] and discrete[8] adjoint approaches where published during the last few years, there are still open issues regarding the robustness, (discrete) consistency and generality for complex problems in turbulent flows. Typically, one has to make a compromise between efficiency and the latter properties while choosing an appropriate approach for a given problem. In general there exist different approaches to construct and solve the discrete adjoint system of equations. Most of them require the exact linearization of the flow residual, which is in contrast to the flow solver itself, where some approximation is most of the time sufficient to yield convergence. Depending on the complexity of the numerical methods the linearization by hand is time-consuming and error-prone. Furthermore it lacks the capability of adapting to changes in the flow solver. One way to circumvent this problems is the use of Algorithmic Differentiation (AD) applied to parts of the flow solver[6,7] to construct the Jacobian. Although it reduces the error-proneness, it still requires the manual application of AD to all subroutines involved in the computation of the residual. Even if we have the exact Jacobian, it is typically ill-conditioned so that applying a Krylov-method can be inefficient. This is often visible when including turbulence models. Here, *Duality-Preserving* methods can be useful as they guarantee to have the same convergence rate as the flow solver. These methods were originally suggested by Korivi and Newman,[9,10] albeit called *Incremental Iterative Form*. Until today they are only used by a manageable amount of people.[11,12,13]

In this paper we want to show that by exploitation of the fixed-point structure of the flow solver it is possible to derive a duality-preserving iteration to solve the adjoint system. All occurring gradients can be constructed by applying AD to the top-level routine of the flow solver, thereby eliminating the manual construction of the exact Jacobi matrix. Furthermore, we apply advanced AD techniques like expression templates[24,25] and local preaccumulation[14] to automatically generate a representation of the computational graph of each expression at compile-time. This results in competitive performance while still maintaining flexibility. Due to the use of AD the extension to new turbulence models, transitions models, fluid models or objective functions is straightforward. To contribute to the open-source idea, we therefore tightly integrated the discrete adjoint solver along with the AD features into the open-source framework SU2,[16,17] in order for the community to explore new and interesting optimization problems. For this reason we also dedicate a section to give a short overview on the implementation.

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

II. Aerodynamic Design Chain

In the following we will give a short overview of the optimization problem and the discretization of the flow equations in SU2. Figure 1 show a simplified representation of the aerodynamic design chain. The components of the design vector α can for example be chosen as the amplitudes of Hicks-Henne functions[26] in 2D or as the control points of the Free-Form deformation method[27] 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 solver then evaluates the state variable 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 residuals of other coupled models. Consequently, U might also consist of the variables of this additional equations, i.e in case of the RANS equations plus a turbulence model we have

$$U := \begin{pmatrix} U_f \\ U_t \end{pmatrix}, \quad R(U) = R(U_f, U_t) := \begin{pmatrix} R_f(U_f, U_t) \\ R_t(U_f, U_t) \end{pmatrix} \tag{3}$$

The discretization of the compressible RANS equations and the turbulence models is performed in SU2 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:

$$\left(D^n_{(\cdot)} + \frac{\partial R_{(\cdot)}(U^n_f, U^n_t)}{\partial U^n_{(\cdot)}} \right) \Delta U^n_{(\cdot)} = -R_{(\cdot)}(U^n_f, U^n_t). \tag{4}$$

where $(\cdot)$ indicates that it can be either the discretization of the flow or the turbulent equation. Here, $R_{(\cdot)}(U^n_f, U^n_t)$ is the residual resulting from the space integration, $\Delta U^n_{(\cdot)} := U^{n+1}_{(\cdot)} - U^n_{(\cdot)}$ and

$$(D^n)_{ij} := \frac{|\Omega_i|}{\Delta t^n_i} \delta_{ij}, \quad \left(\frac{\partial R_{(\cdot)}(U^n_f, U^n_t)}{\partial U^n_{(\cdot)}} \right)_{ij} := \frac{\partial R_{(\cdot),i}(U^n_f, U^n_t)}{\partial U^n_{(\cdot),j}}. \tag{5}$$

Ω_i represents the volume of the cell i and Δt^n_i 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[28] is used, then equation (4) is additionally discretized and solved on consecutively coarser meshes in each iteration to find a correction to be applied to U^{n+1}.

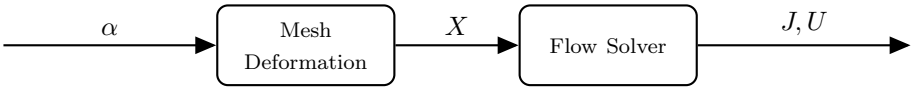


Figure 1: Simplified representation of the aerodynamic design chain.

III. Discrete Adjoint Solver

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 (4), that is, the residual $R(U)$. The left-hand side can therefore be any reasonable approximation to the Jacobian $\frac{\partial R}{\partial U}$. This can be made clear by transforming the solution process of the coupled equations $R(U) = 0$ into a fixed point equation, such that feasible flow and turbulent solutions can be computed from the iteration

$$U^{n+1} = U^n - P^{-1}(U^n) R(U^n) =: G(U^n). \tag{6}$$

Due to the loose coupling of the flow and turbulent equations the preconditioner P can simply be written as

$$P(U) := \begin{pmatrix} D + \frac{\partial \tilde{R}_f(U_f,U_t)}{\partial U_f} & 0 \\ 0 & D + \frac{\partial \tilde{R}_t(U_f,U_t)}{\partial U_t} \end{pmatrix} \approx \frac{\partial R(U)}{\partial U}. \tag{7}$$

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 partial Jacobian $\frac{\partial R_{(\cdot)}}{\partial U_{(\cdot)}}$. It is natural to assume that G is stationary only at feasible points, i.e.

$$R(U^*) = 0 \Leftrightarrow U^* = G(U^*). \tag{8}$$

By the Banach fixed-point theorem, recurrence (6) 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 in the preconditioner P 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 (4) are obtained.
- The coupling of the flow and turbulent equations is neglected (represented by the zero off-diagonal elements in (7)).

If traditional discrete adjoint methods are used,[4] these approximations are not valid anymore, since they result in a linear system involving the exact Jacobian $\frac{\partial R}{\partial U}$ to be solved for the adjoint variables. The explicit construction of $\frac{\partial R}{\partial U}$, however, is in general a formidable task. To circumvent this problem, Korivi et al.[10] 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.[30] 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

$$\min_{\alpha} \quad J(U(\alpha), X(\alpha)) \tag{9}$$

$$\text{subject to} \quad U(\alpha) = G(U(\alpha), X(\alpha)) \tag{10}$$

$$X(\alpha) = M(\alpha). \tag{11}$$

We can define the Lagrangian associated to this problem as

$$L(\alpha, U, X, \bar{U}, \bar{X}) = J(U, X) + [G(U, X) - U]^T \bar{U} + [M(\alpha) - X]^T \bar{X} \tag{12}$$

$$= N(U, \bar{U}, X) - U^T \bar{U} + [M(\alpha) - X]^T \bar{X} \tag{13}$$

where N is the shifted Lagrangian

$$N(U, \bar{U}, X) := J(U, X) + G^T(U, X)\bar{U}. \tag{14}$$

If we differentiate L with respect to α 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 $\bar{X}$:

$$\bar{U} = \frac{\partial}{\partial U} N(U, \bar{U}, X) = \frac{\partial}{\partial U} J^T(U, X) + \frac{\partial}{\partial U} G^T(U, X)\bar{U} \tag{15}$$

$$\bar{X} = \frac{\partial}{\partial X} N(U, \bar{U}, X) = \frac{\partial}{\partial X} J^T(U, X) + \frac{\partial}{\partial X} G^T(U, X)\bar{U} \tag{16}$$

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

$$\frac{\mathrm{d}L}{\mathrm{d}\alpha}^T = \frac{\mathrm{d}J}{\mathrm{d}\alpha}^T = \frac{\mathrm{d}}{\mathrm{d}\alpha} M^T(\alpha)\bar{X}. \tag{17}$$

Equation (15) is a fixed-point equation in $\bar{U}$ and can be solved in the style of the flow solver using the iteration

$$\bar{U}^{n+1} = \frac{\partial}{\partial U} N(U^*, \bar{U}^n, X) \tag{18}$$

once we have found a numerical solution $U = U^*$ of equation (6). 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

$$\left\| \frac{\partial}{\partial \bar{U}} \left[\frac{\partial N}{\partial U}\right]^T \right\| = \left\| \left[\frac{\partial G}{\partial U}\right]^T \right\| = \left\| \frac{\partial G}{\partial U} \right\| < 1. \tag{19}$$

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. But as shown in Albring et al.[18] it turns out that the sensitivity equation (16) and equation (18) can easily be evaluated using Algorithmic Differentiation applied to the underlying routines in the program that compute G.

III.A. Simplified Iteration

The simplicity of the above mentioned approach relies upon the fact that we have made no assumptions regarding the structure of G. While applying AD in a black-box fashion certainly is appealing, it is also rather inefficient. Especially the application of the preconditioner $P(U, X)$ on the residual $R(U, X)$ in equation (6), which requires the (approximate) solution of the linear system (4) in each iteration of the flow solver, would need a lot of resources. Since the efficiency of this adjoint approach directly depends on the number of active variables, that is, variables that are involved in the computation of L and with dependency of X or U, considering some parts of the code as passive can greatly reduce the memory requirements and the runtime of the adjoint solver.

For the derivative of G with respect to U holds

$$\frac{\partial G}{\partial U} = I - P^{-1}\frac{\partial R}{\partial U} - \left[\frac{\partial P}{\partial U}\right]^{-1} R, \tag{20}$$

where we have omitted the arguments to ease the notation. If $\left[\frac{\partial P}{\partial U}\right]^{-1}$ is uniformly bounded, which is the case if P is continuously differentiable, the last term will disappear as $R(U, X)$ converges to zero. Hence, we may drop the last term and instead use the simplified version

$$\bar{U}^{n+1} = \frac{\partial J}{\partial U^*} + \left(I - \left[\frac{\partial R}{\partial U^*}\right]^T P^{-T}\right)\bar{U}^n \tag{21}$$

instead of iteration (18). The same holds for the derivative with respect to X and the evaluation of $\bar{X}$. This explicit separation of the preconditioner P from G is only possible if no multi-grid method is used. Still, it can easily be shown that this approach is also possible if G is more complex. See for example the book of Griewank.[14]

III.B. Relation to other Discrete Adjoint approaches

A form more often seen of iteration (21) can be derived by multiplying it with P^{-T} and introducing $\Lambda^n := P^{-T}\bar{U}^n$:

$$\Lambda^{n+1} = P^{-T}\frac{\partial J}{\partial U^*} + \left(I - P^{-T}\left[\frac{\partial R}{\partial U^*}\right]^T\right)\Lambda^n \tag{22}$$

In the literature this iteration is often referred to as *Duality-Preserving Iteration*[11,12,13] because it can be constructed by requiring that the duality condition[4]

$$\frac{\partial J}{\partial U^*}\frac{\partial U^*}{\partial X} = -\Lambda\left[\frac{\partial R}{\partial X}\right]^T \tag{23}$$

is preserved for each iterate Λ^n of the adjoint solver. Note, however, that condition (23) is derived using $R(U,X)=0$ instead of $G(U,X)=U$ in the problem formulation (9)-(11). In that sense, neither iteration (21) nor iteration (22) are duality-preserving when considering the original problem formulation. However, what this means in practice is still part of ongoing research. Another interesting fact is that if we would have used Newton's method (i.e. $P=\left[\frac{\partial R}{\partial U^*}\right]^{-1}$), iteration (22) reduces to the solution of the linear system

$$\left[\frac{\partial R}{\partial U^*}\right]^T \Lambda = \frac{\partial J}{\partial U^*}. \tag{24}$$

IV. Implementation

In this chapter we give a short summary of the AD features currently available in SU2. In general these methods can be used to compute arbitrary derivatives throughout the code, we will however focus on their use in the implementation of the discrete adjoint solver derived in Section III.

IV.A. Advanced AD Techniques in SU2

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,[18] as we have to store information for each single operation. Another approach is to apply AD on the statement-level. Here we only need to store information for each statement, independent of the number of operations involved. Expression Templates are then used to efficiently compute the partial derivatives of the statements.

Statement-Level Reverse

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

$$y = f(x), \tag{25}$$

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 single-assignment statements with a scalar $v_i \in \mathbb{R}$ on the left-hand side. Each of this statements represents a local evaluation procedure with an arbitrary complex right-hand side $\varphi_i \in \mathbb{R}$, for example

$$v_4 = \varphi_4(v_1, v_2, v_3) = -10 * v_2 * \exp(v_3) + \ln(v_1) - 3 * 10^7 * v_3 * (v_2 - 1) * \sqrt{v_1} \tag{26}$$

It is then possible to write any numerical evaluation in an arbitrary computer program using the general procedure 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 φ_i depends.

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: Statement-Level evaluation procedure for a function f.

Based on the representation of f as a sequence of single-assignment statements, we can also write it as the composition

$$y = Q_m \Phi_l(\Phi_{l-1}(\dots(\Phi_1(P_n^T x)))) \tag{27}$$

where the state transformation Φ_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

$$\dot{y} = Q_m A_l A_{l-1} \dots A_2 A_1 P_n^T \dot{x}. \tag{28}$$

Thus, the Jacobian of f can be written as

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = Q_m A_l A_{l-1} \dots A_2 A_1 P_n^T \tag{29}$$

By transposing the product we obtain the adjoint relation

$$\bar{x} = P_n A_1^T A_2^T \dots A_{l-1}^T A_l^T Q_m^T \bar{y} = \left(\frac{\mathrm{d}f}{\mathrm{d}x}\right)^T \bar{y} \tag{30}$$

and the identity

$$\bar{y}\dot{y} = \bar{x}\dot{x}. \tag{31}$$

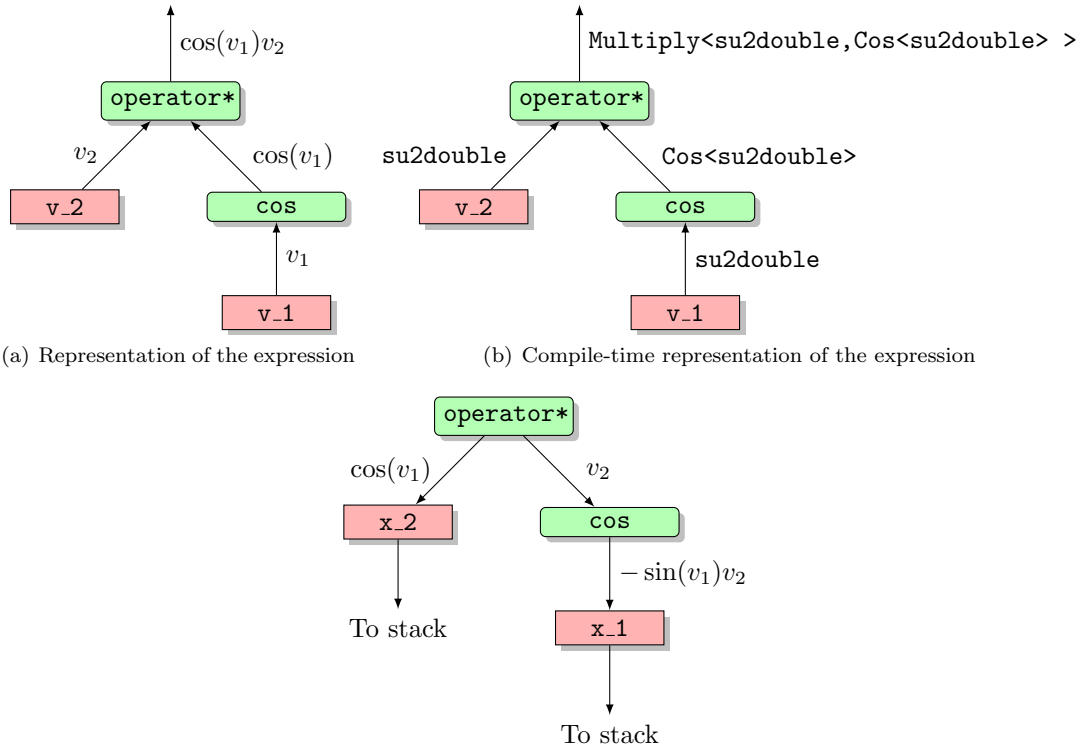
If one looks into detail at some intermediate matrix multiplication in (30), it can be seen, that it is possible to write the adjoint relation as an evaluation procedure. All matrix-vector products are calculated for $i = l, l-1, \dots, 1$ thus we have to go backward, or reverse, through the sequence 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 seed vector $\bar{y}$, i.e. equation (30), by evaluating Table 2.

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

Table 2: Statement-Level Adjoint Interpretation.

Expression Templates

Table 2 illustrates what information is necessary in order to correctly accumulate the adjoint vector $\bar{x}$. Aside from some reference to the input variables of each statement we need to compute the partial derivatives $\frac{\partial}{\partial v_j}\varphi_i(u_i)$. An efficient way is the use of Expression Templates[24,25] , a C++ meta-programming technique. Here 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 to directly compute and store $\frac{\partial}{\partial v_j}\varphi_i(u_i)$ during the evaluation of f. Figure 2(a) shows the computational graph of the expression $\varphi(v_1, v_2) = \cos(v_1)v_2$ and Figure 2(b) the corresponding compile-time representation as object with `su2double` being the new datatype used throughout the SU2 code. This object can then be traversed to compute and store the partial derivatives $\frac{\partial\varphi}{\partial v_1} = -\sin(v_1)v_2$ and $\frac{\partial\varphi}{\partial v_2} = \cos(v_1)$ as shown in figure 2(c) based on the derivatives of each involved unary or binary operation. If recording is enabled the traversal of the computational graph of each expression is started as soon as it occurs on the right-hand side in a statement during the evaluation of Table 1. Note that the partial derivatives $\frac{\partial}{\partial v_j}\varphi_i(u_i)$ are only stored if v_j depends on x. This kind of dependency or activity tracking is relatively easy to accomplish since every variable stores an index along with its value. A zero index represents passive variables while a non-zero index identifies active variables. This index will be non-zero only if the corresponding variable has been assigned an expression with at least one active variable as an argument.



(a) Representation of the expression

(b) Compile-time representation of the expression

(c) Propagation of gradients in reverse mode

Figure 2: Computational graphs for the statement $\varphi = \cos(v_1)v_2$.

Local Preaccumulation

To alleviate the high memory requirements we apply the so called *local preaccumulation*[14] method. Here we compute and store the local Jacobi matrices of certain enclosed code parts instead of storing each individual statement contribution. To illustrate the method consider the code shown in listing 1 which computes the volume of 2D elements of the mesh. Without using preaccumulation and if we assume that `nDim` equals 2 we have to store 12 partial derivative values for the 3 statements ($2*2$ in line 8, $2*2$ in line 9 and 4 in line 12).

```
1   AD::StartPreacc();
2   AD::SetPreaccIn(val_coord_Edge_CG, nDim);
3   AD::SetPreaccIn(val_coord_Elem_CG, nDim);
4   AD::SetPreaccIn(val_coord_Point, nDim);
5
6
7   for (iDim = 0; iDim < nDim; iDim++) {
8     vec_a[iDim] = val_coord_Elem_CG[iDim]-val_coord_Point[iDim];
9     vec_b[iDim] = val_coord_Edge_CG[iDim]-val_coord_Point[iDim];
10  }
11
12  Local_Volume = 0.5*fabs(vec_a[0]*vec_b[1]-vec_a[1]*vec_b[0]);
13
14  AD::SetPreaccOut(Local_Volume);
15  AD::EndPreacc();
```

Listing 1: Example for Local Preaccumulation

With preaccumulation we only store the derivatives of the variables flagged using `AD::SetPreaccIn` with respect to the variables flagged with `AD::SetPreaccOut`. For our example this results in 6 values, which means total saving of memory of 50%. Note that this exemplary code snippet is executed for every element in the mesh. Throughout the code this method is applied at several spots where a clear identification of inputs and outputs was possible.

Handling of the Linear Solver

It is a well known fact that the differentiation of linear solvers is unfeasible due to the high amount of statements. A more efficient way is the use of the analytic derivative of the solution process. Consider the application of an inverse matrix A^{-1} on some vector b, i.e.

$$w = A^{-1}b \tag{32}$$

It can be shown[35] that the application of reverse mode AD of this basic operation yields

$$\bar{b} = A^{-T}\bar{w} \tag{33}$$
$$\bar{A} = -\bar{b}\bar{w}^T \tag{34}$$

where the adjoint values $\bar{\cdot}$ can be interpreted in the same way as in Table 2, in the sense that we can consider equation (32) as a single statement that occurs somewhere in the code. Consequently, when we evaluate equations (33) and (34) during the adjoint interpretation, $\bar{w}$ should already hold non-trivial values. An advantage of using this analytic derivative is that we can reduce the runtime of the adjoint solver by reducing the number of iterations for solving the linear system (33). Similar to the flow solver, typically a couple of preconditioned FGMRES or BCGSTAB iterations (usually in the range of 2-5) are sufficient in order to achieve stable convergence.

Adjoint MPI

SU2 is parallelized using MPI for the application in HPC environments. A key factor for the efficiency of the adjoint solver the correct differentiation of the parallel communication. For that reason we use the AdjointMPI[31] library that provides for every MPI routine a version to be called in the forward section and its matching implementation call during the adjoint interpretation. A small subset of corresponding calls is shown in Table 3. We simplified the listed MPI calls by omitting parameters that are not relevant for this discussion. In order to keep the changes to the original code as small as possible and to improve the readability SU2 provides a wrapper class `SU2_MPI`, that chooses the correct MPI calls (either the original calls or the overloaded calls provided by AMPI) depending on the datatype used during the compilation of the software suite.

MPI Call	Adjoint MPI Call	Wrapper Function Call
`MPI_Send(V)`	`MPI_Recv(V)`	`SU2_MPI::Send(V)`
`MPI_Recv(V)`	`MPI_Send(V)`	`SU2_MPI::Recv(V)`
`MPI_Bcast(V)`	root: `MPI_Recv(V)`, not root: `MPI_Send(V)`	`SU2_MPI::Bcast(V)`
`MPI_Reduce(V)`	`MPI_Bcast(V)`	`SU2_MPI::Reduce(V)`

Table 3: Exemplary MPI routines and their corresponding routines during the adjoint interpretation.

AD Tool Wrapper

The above introduced Expression Template approach is implemented in the open-source C++ library *Code-Differentiation Package* (CoDiPack) TODO REF. This library provides a special datatype and is automatically included during the compilation if AD support is requested by the user. For developers of SU2 there is no need to deal with this library explicitly which is why there are simple wrapper routines for the most important features available. These are for example the following:

- `AD::RegisterInput(su2double &var)` - Registers the variable as input, i.e. sets the index to a non-zero value. The exact value is determined by the AD tool.
- `AD::StartRecording()` - Starts the recording by enabling the traversal of the computational graphs of each subsequent expression to store the partial derivatives.

- `AD::StopRecording()` - Stops the recording of information.
- `AD::ComputeAdjoint()` - Interprets the stored information to compute the adjoints according to Table 2.
- `AD::Reset()` - Deletes all stored information, i.e. the adjoint values and the partial derivatives to enable a new recording.
- `AD::ClearAdjoints()` - Sets the adjoint values to zero but keeps the derivative information, thereby enabling a new interpretation with a different seed vector $\bar{y}$.

Since the actual interpretation of the adjoints is done by the AD tool, we need some functions to set and extract the derivative information. To account for other datatypes (like for example types that implement the forward mode of AD or types for the complex step method) these functions are enclosed in the namespace `SU2_TYPE`:

- `SU2_TYPE::SetDerivative(su2double &var, double &val)` - Sets the adjoint value of a variable before calling `AD::ComputeAdjoint()`.
- `SU2_TYPE::GetDerivative(su2double &var)` - Returns the interpreted adjoint value of a variable after a call to `AD::ComputeAdjoint()`.

IV.B. Top-Level Approach

Using the methods described in the previous subsection we can easily construct the adjoint solver derived in Section III. This is based on the fact, that by correctly initializing the adjoint values of the objective function and the flow solution (i.e. setting $\bar{y}$), the adjoint interpretation in Table 2 yields the gradient of the shifted Lagrangian needed for iteration (18) (see Albring et al.[18]). Figure 3 illustrates this top-level approach. We start with registering the numerically converged flow solution U^* as input. A subsequent call to the flow iteration routine and the evaluation of the objective function will record the information that is necessary to compute the gradient of any linear combination of G and J with respect to U^*. Starting with some initial adjoint vector $\bar{U}^n$ we can perform iteration (18) by simply setting $\bar{U}^+$ and $\bar{W}$ to the correct values and running the adjoint interpretation until we have achieved convergence. To compute $\bar{X}$ we use a single evaluation with X instead of U^* registered as input once we have converged the adjoint solver. Since this iteration is done on an abstract level, it is not necessary to know the exact structure of G. Additionally any changes to G or J are automatically incorporated for the adjoint iteration.

An important fact is that it is also possible to easily realize the simplified iteration (21) using this approach. First note that dropping the last term in (20) is equivalent to assuming that the preconditioner P is constant, i.e. that it is independent of U. Thus it is equivalent to setting $\bar{A} = 0$ instead of evaluating equation (34) in the analytic differentiation of the linear solver during the adjoint interpretation.

V. Applications

In this section we will give an overview of some aerodynamic applications of the discrete adjoint solver.

V.A. Optimization of the LM1021 in Inviscid Flow

The conceptual supersonic transport design, known as Lockheed Martin (LM) 1021, was developed for the NASA N+2 Supersonic Validations program and was design to produce very low sonic boom. It was chosen as a test case for the 2014 AIAA Sonic Boom Prediction Workshop, where it was found that viscosity was not important at full-scale[37] due to the very thin boundary layer. Here we use this case as an optimization test case to show the feasiblity of the adjoint solver for realistic configurations. The gradients for the inviscid case were validated in Albring et al.[18] showing a good agreement with the finite difference gradient.

As free-stream values we consider the cruise-conditions at a Mach number of 1.6 and an angle of attack of 2.1°. The computational mesh consists of 5,730,841 interior elements and the aircraft is discretized using 214,586 boundary elements. For the spatial integration we use a central scheme with artifical dissipation in combination with an implicit Euler method for the pseudo-time stepping. For the adjoint solver we use the simplified iteration (21).

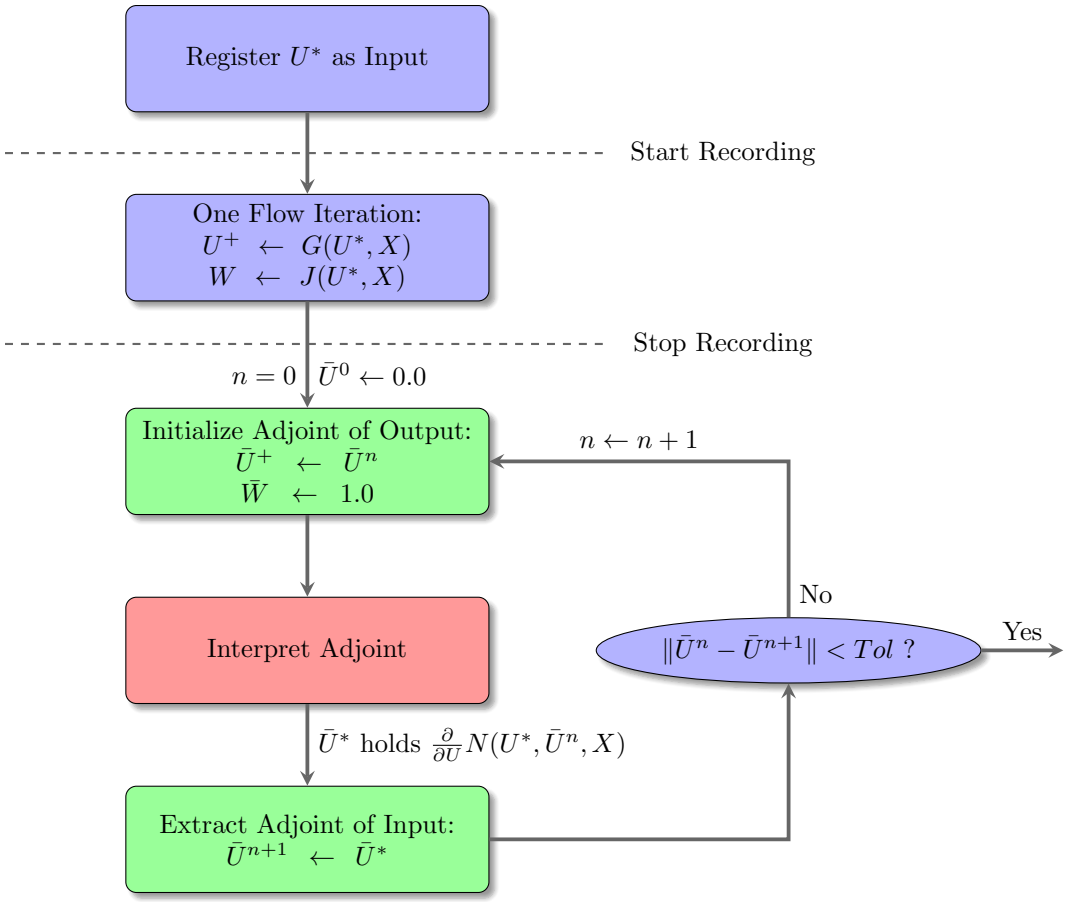


Figure 3: Top-Level Approach of the Discrete Adjoint Solver in SU2

To measure the parallel performance of the adjoint solver we have run the case using 32, 64, 128 and 256 processors, which is equivalent of using 2, 4, 8 and 16 nodes. Each node is equipped with 16 Intel E5-2640v3 processors and 64GB of memory. Figure 4 shows the resulting speedup (left) and the parallel efficiency (right). It is clearly visible that the adjoint solver offers a better scalability and efficiency than the flow solver, which is probably due the slightly increased workload. Note that at 256 cores each processor has roughly 9000 points to process, so that 70% efficiency still means reasonable scaling. The runtime ratio

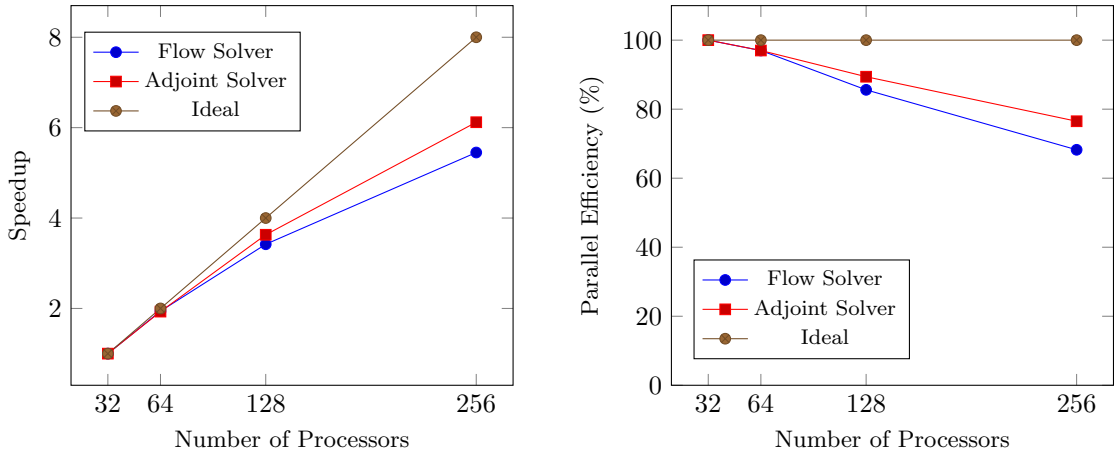


Figure 4: Speedup (left) and parallel efficiency (right) for the LM1021 test case.

between one iteration of the adjoint solver and one iteration of the flow solver on 64 processors is given by

$$\frac{t_{\text{adjoint}}}{t_{\text{flow}}} = 1.17. \tag{35}$$

For the optimization we want to reduce the drag while applying realistic constraints in terms of lift, moment and thickness. More specific we constraint the lift to be larger than $C_L = 0.132$, which equals to 95% of the initial lift while the pitching moment should stay below the initial value of $C_{M_y} = 0.18$. The maximum thickness at five different sections should not be reduced by 90% of initial value. We use the already present optimization framework of SU2 which uses an SQP method with BFGS approximation to reduce the objective function. The wing shape is parameterized using the Free-Form deformation approach with a total of 264 design variables where a movement was only allowed in the z-direction. The region affected by shape modifications is highlighted by the box in Figure 7.

The convergence history of the flow and adjoint solver during the first design iteration is shown in Figure 5. As expected, the simplified version of iteration (18) offers the same convergence rate as the flow solver. Since the sensitivity $\bar{X}$ is only computed after we have converged the adjoint solver, we plot the derivative $\frac{dC_D}{dMa}$ in order to have an additional coefficient to check the convergence. Note that this value can be computed at no cost, by just registering it as an additional input at the beginning of the iteration shown in Figure 3. We see that this derivative needs slightly more iterations until it settles compared to C_D. In Figure 6 the optimization history is plotted. Each iteration represents a evaluation of a design, which might not allows induce a subsequent gradient evaluation due to the step size computation of the line search algorithm. However, the objective function is reduced in each step, indicating that the line search method is able to easily find a suitable direction. Furthermore the constraints are all maintained well. In total the optimization achieved a reduction of 9 ΔC_D compared to the initial design. A comparison of the baseline and optimized pressure distribution is shown in Figure 7. It can be clearly seen that the optimized design offers a smoother distribution of the pressure coefficient. However, to maintain the lift and pitching moment a low pressure region is concentrated near the leading edge. The resulting drag surface sensitivity $\bar{X}$ on the wing is plotted in Figure 8. In this case there is barely a difference visible, however, in general this sensitivity can be a valuable contribution to the design process, as it immediately visualizes areas that have the largest influence on the objective function.

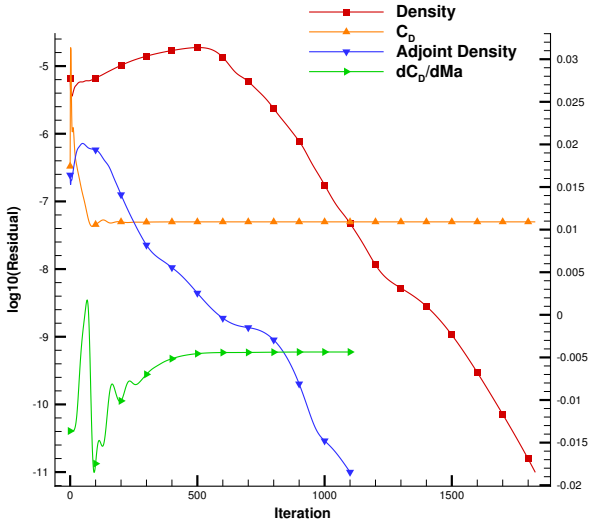


Figure 5: Residual history of the flow and adjoint solver for the LM1021 case in the first design iteration.

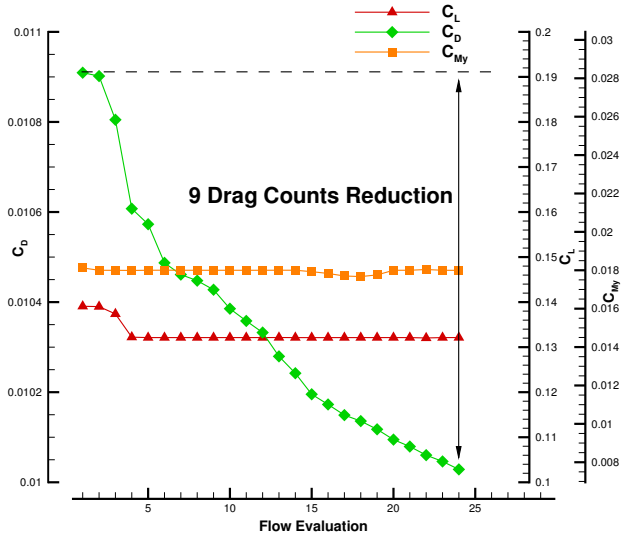


Figure 6: Optimization history for the LM1021 case.

V.B. Sensitivity Analysis of the OneraM6 Wing in Turbulent Flow

As another application we consider the OneraM6 wing in turbulent flow conditions. The Onera M6 wing is a classic CFD validation case for external flows because of its simple geometry combined with complexities of transonic flow, i.e. local supersonic flow, shocks, and turbulent boundary layers separation. The setup of

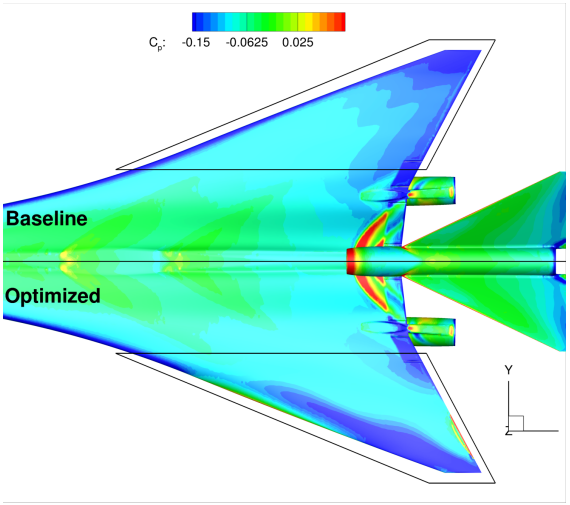


Figure 7: C_p distribution on the upper wing surface of the LM1021.

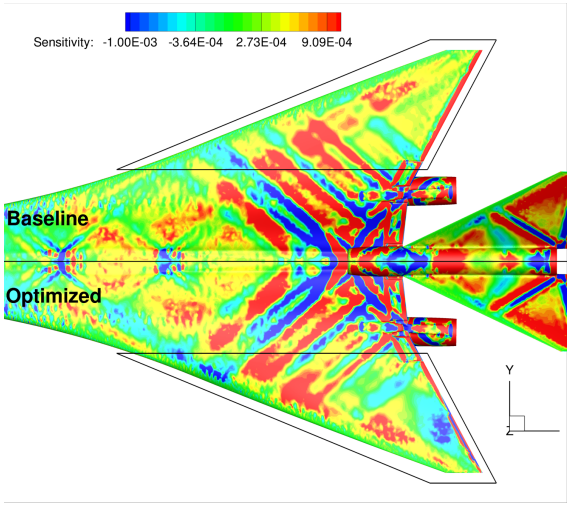


Figure 8: Surface sensitivity $\bar{X}$ on the upper wing surface of the LM1021.

this case closely follows the wind tunnel tests documented by Schmitt and Charpin[38] which are summarized in Table 4. For the spatial discretization we again use a central scheme with artifical dissipation coupled with an implicit Euler method for the pseudo-time integration. The turbulence model is the SST $k-\omega$ model. To validate the gradient we have chosen a rather coarse mesh with 43008 elements. The wing shape

Ma	Reynolds Number	Angle of Attack (°)
0.8395	11.72×10^6	3.06

Table 4: Flow conditions for the OneraM6 case.

is parameterized using 74 FFD control points. A comparison with the complex step method is shown in Figure 9 and Figure 10 for the drag gradient and the lift gradient, respectively. We have a perfect agreement between both methods, except for some control points. However, these points are located near the trailing edge of the wing where the flow and turbulence quantities are not converged reasonable well.

Due to the increasing demand for large-scale flow computations one of the requirements of this approach is the capability of performing the sensitivity analysis or even design optimization also for these cases. As a first step we used a mesh with 27286490 elements. To validate the resulting flow solution we compare the pressure distribution at sections $y/b = 0.65$ and $y/b = 0.9$ against experimental data taken from Schmitt and Charpin[38] shown in Figure 11 and Figure 12, respectively. At both sections we have a good agreement with the data. The pressure distribution over the wing is shown in Figure 13 where the lambda shock configuration is clearly visible. Figure 14 depicts the corresponding drag surface sensitivity $\bar{X}$ computed using the discrete adjoint solver.

VI. Outlook

In this paper we presented the derivation of a discrete adjoint solver that efficiently makes use of Algorithmic Differentiation, thereby avoiding the error-prone and time-consuming construction of the Jacobi matrix. We shortly discussed the implementation inside the open-source framework SU2 that is freely available online (`https://www.github.com/su2code/SU2`) and showed the top-level approach that enables an automatic adaption to changes or modifications in the flow solver. Furthermore we applied the adjoint solver to an optimization of a realistic configuration and validated the gradients also for turbulent flows. Future developments will include improvements to the performance and usability and applications to multi-physics problems like CAA and FSI.

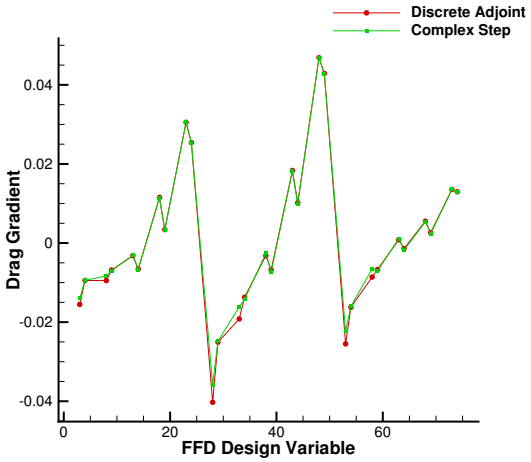


Figure 9: Drag sensitivity validation.

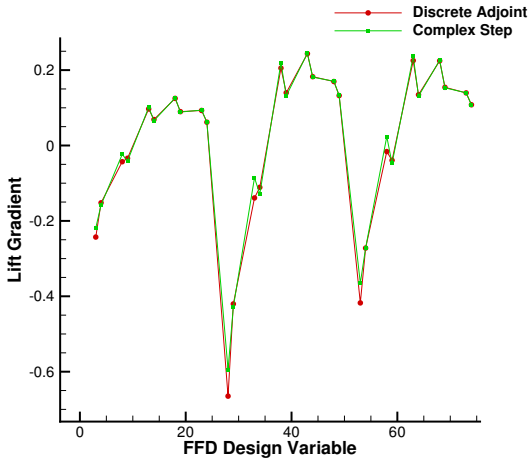


Figure 10: Lift sensitivity validation.

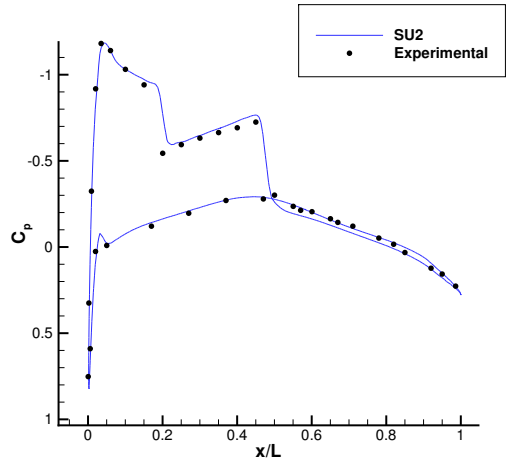


Figure 11: Pressure distribution at $y/b = 0.65$ of the OneraM6 wing.

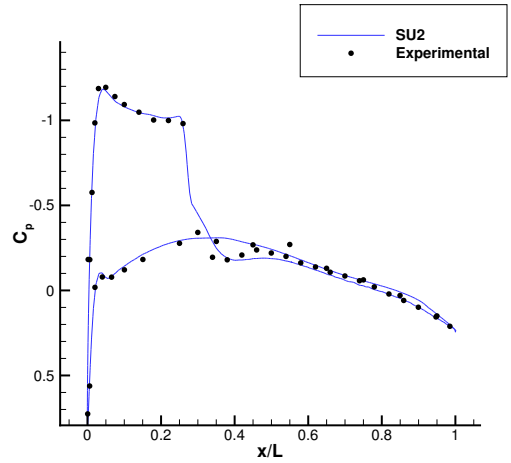


Figure 12: Pressure distribution at $y/b = 0.9$ of the OneraM6 wing.

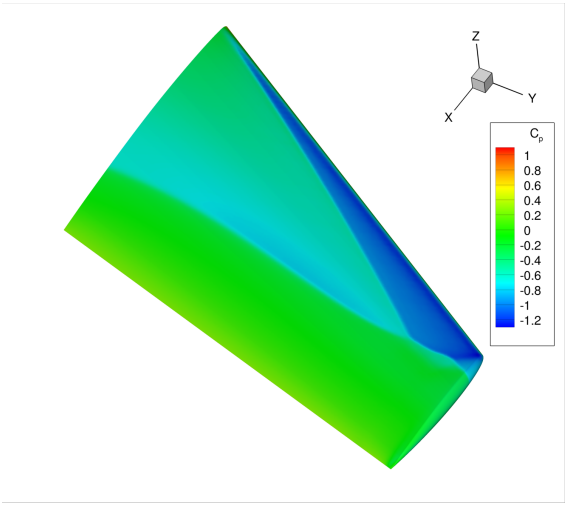


Figure 13: C_p distribution on the OneraM6 wing.

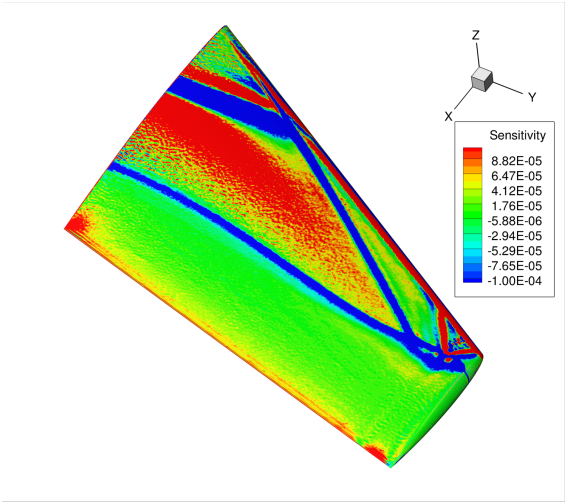


Figure 14: Surface sensitivity $\bar{X}$ on the OneraM6 wing.

Acknowledgements

The authors would like to thank the whole SU2 team at Stanford University, especially Dr. Thomas Economon for providing the test cases and his continuous support.

References

[1]Anthony Jameson. 'Success and Challenges in Computational Aerodynamics.' AIAA Paper 87-1184, 1987.

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

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

[4]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.

[5]Francisco Palacios, Thomas D. Economon, Andrew Wendorff and Juan J. Alonso. 'Large-scale aircraft design using SU2'. AIAA Paper 2015-1946, 2015.

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

[7]Zhoujie Lyu, Gaetan Kenway, Cody Paige, Joaquim R. Martins. 'Automatic Differentiation Adjoint of the Reynolds-Averaged Navier-Stokes Equations with a Turbulence Model'. 43rd AIAA Fluid Dynamics Conference and Exhibit, 2013.

[8]Eric J. Nielsen, Boris Diskin. 'Discrete Adjoint-Based Design for Unsteady Turbulent Flows on Dynamic Overset Unstructured Grids', AIAA Journal, Vol. 51, No. 6., pp. 1355-1373, 2013.

[9]P.A Newman, G. J.-W. Hou, A. C. Taylor III and V. M. Korivi. 'Observations on Computational Methodologies for use in Large-Scale, Gradient-Based, multi-disciplinary design incorporating advanced CFD codes'. NASA Technical Memorandum 104206, 1992.

[10]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.

[11]Dimitri J. Mavriplis, "Formulation and Multigrid Solution of the Discrete Adjoint Problem on Unstructured Meshes." Computational Fluid Dynamics 2004. Springer Berlin Heidelberg, 2006. 663-668.

[12]Jacque E.V. Peter and Richard P. Dwight. 'Numerical Sensitivity Analysis for Aerodynamic Optimization: A Survey of Approaches'. Computers and Fluid, Vol. 39(3), pp.373-391, 2010.

[13]Nielsen, E.J., Lu, J., Park, M.A. and Darmofal, D.L.. 'An implicit, exact dual adjoint solution method for turbulent flows on unstructured grids'. Computers & fluids, 33(9), pp.1131-1155, 2004.

[14]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.

[15]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.

[16]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.

[17]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.

[18]Tim Albring, Max Sagebaum, Nicolas R. Gauger. 'Development of a Consistent Discrete Adjoint Solver in an Evolving Aerodynamic Design Framework', AIAA Paper 2015-3240, 2015.

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

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

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

[22]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.

[23]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.

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

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

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

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

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

[29]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.

[30]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.

[31]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.

[32]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

[33]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.

[34]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.

[35]Mike B. Giles. 'Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation'. Advances in Automatic Differentiation, pp. 35-44, 2008.

[36]J. M. Morgenstern, M. Buonanno, and F. Marconi. 'Full Configuration Low Boom Model and Grids for 2014 Sonic Boom Prediction Workshop'. AIAA Paper 2013-0647, 2013.

[37]John M. Morgenstern, Michael Buonanno and Frank Marconi. "Full Configuration Low Boom Model and Grids for 2014 Sonic Boom Prediction Workshop." AIAA Paper 2013-647, 2013.

[38]V. Schmitt and F. Charpin, 'Pressure Distributions on the ONERA-M6-Wing at Transonic Mach Numbers', Experimental Data Base for Computer Program Assessment. Report of the Fluid Dynamics Panel Working Group 04, AGARD AR 138, 1979.