AIAA AVIATION Forum
June 15-19, 2020, VIRTUAL EVENT
AIAA AVIATION 2020 FORUM

10.2514/6.2020-3139

Coupled Discrete Adjoints for Multiphysics in SU2

Ole Burghardt[*] and Nicolas R. Gauger[†]
Chair for Scientific Computing, TU Kaiserslautern, 67663 Kaiserslautern, Germany

Pedro Gomes[‡] and Rafael Palacios[§]
Department of Aeronautics, Imperial College London, SW7 2AZ London, U.K.

Tobias Kattmann[¶]
Bosch Corporate Research - Fluid Dynamics, 71272 Renningen, Germany

Thomas D. Economon[‖]
SU2 Foundation, San Francisco, CA, 94158, U.S.A.

Many optimization problems in fluid dynamics are naturally constituted in a larger multiphysics setting, the most prominent ones nowadays being fluid-structure interaction and conjugate heat transfer or combinations of them. In the context of partitioned solution approaches, and from a software perspective, this means that two or more solvers are coupled by exchanging data at common physical boundaries during the simulation.
Optimization methods relying on discrete adjoint solutions have to consider such couplings in order to give accurate gradients. This article presents a methodology for how coupled multiphysics adjoints can be computed efficiently in a way that is also independent of the choice or configuration of the underlying physical problem.
Based on an implementation in SU2 (an open-source multiphysics simulation and design software[1]) we demonstrate how the same algorithm can be applied for shape sensitivity analysis on a conjugate heat transfer as well as a fluid-structure interaction test case.

I. Introduction

Especially in the engineering sciences that are concerned with flow problems, one finds important cases where multiphysics effects play a crucial role in the design of new parts. For instance, it could be a turbine blade that heats up in a high-pressure flow (and thus deforms or even undergoes physical damage) or an airfoil that deforms under flow tractions exerted on its mechanical structure. In,[2] the authors studied the case of a heat sink device where heated fins in a variable density incompressible flow show inhomogeneous temperature distributions which can have an non-negligible influence on their optimal design.
On a more abstract level, a multiphysics problem is just a problem that is set up by combining different physical domains, hereafter referred to as *zones*, where we assume that each is governed by a partial-differential equation. All zones therefore have their own solver based on appropriate discretizations. In fluid dynamics we are most commonly dealing with Navier-Stokes or RANS equations but one can also think of solid zones where we are concerned with elastic deformations or heat conduction.
In general, let us denote such solvers by a function

$$\mathcal{G}_{(\boldsymbol{x})} : \mathbb{R}^{mn} \to \mathbb{R}^{mn} \tag{1}$$

[*]PhD Candidate, Chair for Scientific Computing.
[†]Professor, Chair for Scientific Computing, AIAA Associate Fellow.
[‡]PhD Student, Department of Aeronautics.
[§]Professor of Computational Aeroelasticity, Department of Aeronautics, AIAA Associate Fellow.
[¶]PhD Student, Chair for Scientific Computing.
[‖]Executive Director, AIAA Senior Member.

that maps an intermediate solution $\boldsymbol{u}^i$ to $\boldsymbol{u}^{i+1}$, where $\boldsymbol{x} \in \mathbb{R}^m$ denotes an arbitrary but fixed design parameter vector (we think of mesh coordinates unless otherwise specified) and n is the number of components (formerly the number of equations of the discretized PDE problem).

We assume that the solution $\boldsymbol{u} = u(\boldsymbol{x})$ to the PDE problem is given by the fixed point of $\mathcal{G}$, this incurs no loss of generality since common primal solution methods can easily be put in this form. In a practical sense, $\boldsymbol{u}$ is the first state for which a suitable convergence criteria is met, for example

$$\|\mathcal{G}_{(\boldsymbol{x})}(\boldsymbol{u}^i) - \boldsymbol{u}^i\| < \varepsilon_{\text{tol}}$$

where ε_{tol} is a prescribed accuracy. Still, for all mathematical considerations, we assume that $\mathcal{G}_{(\boldsymbol{x})}(\boldsymbol{u}) = \boldsymbol{u}$. We can keep this notation for multiphysics as well by assuming that we are using a partitioned approach and regarding $\mathcal{G}(\boldsymbol{u}^i)$ as the vector whose components are the iterates in each zone, that is

$$\begin{pmatrix} \boldsymbol{u}_{(1)}^{i+1} \\ \vdots \\ \boldsymbol{u}_{(r)}^{i+1} \end{pmatrix} = \begin{pmatrix} \mathcal{G}_{(1)}(\boldsymbol{u}^i) \\ \vdots \\ \mathcal{G}_{(r)}(\boldsymbol{u}^i) \end{pmatrix}. \tag{2}$$

Note that the inputs

$$\boldsymbol{u}^i = (\boldsymbol{u}_{(1)}^i, \ldots, \boldsymbol{u}_{(r)}^i)$$

of each solver on the right hand side are (possibly) the solution iterates of all zones as these dependencies arise from the exchange of solution values (or their derived quantities) at common physical boundaries and we wish to leave open which kind of couplings a user of the program might have chosen.
We then regard $\mathcal{G}_{(k)}$ as a fixed-point iterator with respect to the solution variables $\boldsymbol{u}_{(k)}^i$ and all $\boldsymbol{u}_{(l)}^i$ with $k \neq l$ as solver parameters, this distinction will later be used to define cross derivatives in the context of adjoints.
Throughout this article, subscripts will indicate the zone number, whereas superscripts refer to the iteration count unless otherwise specified.

Objective functions, $\tilde{J}$, based on solutions of PDE problems can be said do depend only on the design parameters x since the solution u, and all other derived quantities, are also defined by the parameters, i.e. $u = u(x)$. However, in deriving the adjoint solution method it is useful to consider the functional of interest a function of the parameters and the solution. We denote this by defining an objective function $J : \mathbb{R}^m \times \mathbb{R}^{mn} \to \mathbb{R}$ and setting

$$\tilde{J}(x) := J(x, u(x))$$

accordingly. Naive differentiation of $\tilde{J}$ at $\boldsymbol{x}$ by application of the chain rule,

$$D\tilde{J}(\boldsymbol{x}) = \frac{\partial}{\partial x} J(\boldsymbol{x}, \boldsymbol{u}) + \frac{\partial}{\partial u} J(\boldsymbol{x}, \boldsymbol{u}) \cdot \frac{\partial}{\partial x} u(\boldsymbol{x}), \tag{3}$$

results in a computationally intractable problem for large m, as the last term in (3) requires differentiation of the converged solution (involving the complex program $\mathcal{G}_{(\boldsymbol{x})}$) with respect to every solution variable. Thus an approach that efficiently captures the implicit dependence on the solution variables is needed.

II. Discrete Adjoints

The difficulty of complex implicit dependencies of quantities (that are needed to compute an objective J) on some design parameters x has – in the context of fluid dynamics – early been approached through adjoints by e.g. Jameson who introduced them in[3] as a Lagrange multiplier for certain (potential) flow solution contraints to be fulfilled. The approach presented in this section will abstractly follow this idea. For an introduction to the relation of adjoint design approaches and present CFD solvers, see for example the overview by M. Giles.[4]

In the context of discrete adjoints, let us directly define the Lagrangian $L : \mathbb{R}^m \times \mathbb{R}^{mn} \to \mathbb{R}$ as

$$L(x, u) := J(x, u) + (\mathcal{G}(x, u) - u) \cdot \lambda. \tag{4}$$

Restricted to the set of flow solutions, L equals J, independent of the choice of λ. In particular, we have that

$$D\tilde{J}(\boldsymbol{x}) = \frac{\mathrm{d}}{\mathrm{d}x} L(\boldsymbol{x}, \boldsymbol{u}).$$

If we now could find λ such that $\frac{\partial}{\partial u} L = 0$, which is equivalent to

$$\nabla_u J(\boldsymbol{x}, \boldsymbol{u}) \overset{!}{=} -D_u \mathcal{R}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda, \tag{5}$$

where

$$\mathcal{R}(x, u) := \mathcal{G}(x, u) - u, \tag{6}$$

$D\tilde{J}(\boldsymbol{x})$ would be easy to compute as no terms $\frac{\partial}{\partial x} u(\boldsymbol{x})$ appear anymore, that is

$$\begin{aligned} \nabla \tilde{J}(\boldsymbol{x}) &= \nabla_x L(\boldsymbol{x}, \boldsymbol{u}) \\ &= \nabla_x J(\boldsymbol{x}, \boldsymbol{u}) + D_x \mathcal{R}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda. \end{aligned} \tag{7}$$

In this work algorithmic differentiation (AD) is used for numerical evaluation of the partial derivatives in (7), in the context of SU2 this has been suggested and applied by Albring et al.[5]
There, the authors successfully developed a discrete adjoint solver in SU2 for aerodynamic shape optimizations by implementing the fixed-point iteration equivalent for (5),

$$\lambda \overset{!}{=} \nabla_u J(\boldsymbol{x}, \boldsymbol{u}) + D_u \mathcal{G}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda. \tag{8}$$

Here, for each new iterate λ_{i+1}, one needs to evaluate $D_u \mathcal{G}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda_i$, which naturally fits the formulation of the reverse mode of AD, though some more attention has to be paid as we are dealing with multiple zones. Abstractly this means that also our adjoint solution λ now consists of the adjoint solutions of all zones,

$$\lambda = (\lambda_{(1)}, \ldots, \lambda_{(r)})$$

and that a fixed-point iteration update in (8) is an update for each and every zone,

$$\lambda_{(k)}^{(i+1)} = \frac{\partial J}{\partial u_{(k)}}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) + \frac{\partial \mathcal{G}}{\partial u_{(k)}}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda^{(i)}, \tag{9}$$

where the full vectors λ and $\partial\mathcal{G}/\partial u_{(k)}$ appear on the right side. In keeping with primal solution algorithms, more intricate update schemes can be proposed. Most importantly those where we carry out multiple updates with respect to one zone before updating the adjoints variables of the others. With i being the (outer) iteration count and L the number of repeated inner updates, this gives an alternative to (9),

$$\begin{cases} \lambda_{(k)}^{(l+1,i)} &= \frac{\partial J}{\partial u_{(k)}}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) + \frac{\partial \mathcal{G}_{(k)}}{\partial u_{(k)}}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda_{(k)}^{(l,i)} + \sum_{j \neq k} \frac{\partial \mathcal{G}_{(j)}}{\partial u_{(k)}}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda_{(j)}^{(i)} \\ \lambda_{(k)}^{(i+1)} &= \lambda_{(k)}^{(L,i)} \end{cases} \tag{10}$$

which, to the authors' experience, can lead to a more stable procedure for $L > 1$ (note for $\lambda^{(0,i)} := \lambda^{(i)}$ and $L = 1$, we recover Eq. (9)). For every outer iteration i we have to evaluate Eq. (10) for each zone k which raises the question of how to efficiently manage the sum of off-diagonal contributions, i.e. keeping it updated with values already at $i + 1$ (from previously updated zones). This will be detailed in section III B.
All in all, even the specific algorithm is up to configuration, and all we need is a functionality to evaluate derivatives of the form

$$\frac{\partial \mathcal{G}_{(j)}}{\partial u_{(k)}}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda_{(j)} \tag{11}$$

for a given pair (j, k) and a vector $\lambda_{(j)}$ at low computational cost.

III. Algorithmic differentiation of a multiphysics solver

A. Storing and evaluating derivative information via AD

Derivatives as in (11) can be evaluated by letting $\mathcal{G}$ calculate a solution update $(\boldsymbol{x}, \boldsymbol{u})$ from a previous iterate for all zones (as in (2)) while saving the kind of operations that are executed along their execution (or directly their derivatives) to an external data structure so that we can later evaluate the overall derivatives using the chain rule. There are many such tools available but as in Albring et al.,[5] CoDiPack[6] was chosen for its performance and convenience to use within *C++* codes.
More precisely and in order to introduce some AD vocabulary that is taken from,[7] the computer program $\mathcal{G}$ is a map $\mathcal{G} : \mathbb{R}^a \to \mathbb{R}^b$ that can be represented as a sequence of l statements $\varphi_i : \mathbb{R}^{n_i} \to \mathbb{R}$.
For each i, we refer to v_i as the output values in the image of φ_i. Its input values are denoted by w_i which would be just a vector of n_i preceding output values, denoted by

$$w_i := (v_j)_{j \prec i} \in \mathbb{R}^{n_i}.$$

Evaluating $\mathcal{G}$ with respect to some input values $u_i \in \mathbb{R}^a$ and output values $y_i \in \mathbb{R}^b$ can now be described as

$$\begin{array}{lll} v_i & = u_i & i = 1 \dots a \\ v_{i+a} & = \varphi_i(w_i) & i = 1 \dots l \\ y_i & = v_{a+l-i+1} & i = 1 \dots b. \end{array}$$

Figure 1. Primal run of $\mathcal{G}$.

If during its evaluation the derivative information of all its statements was stored, that is the values

$$\frac{\partial}{\partial v_j} \varphi_i(w_i), \tag{12}$$

we could then obtain all intermediate derivatives with respect to the v_i (indicated by bars and commonly simply called "the adjoint values") and an arbitrary but fixed $\bar{\lambda} \in \mathbb{R}^b$ by executing

$$\begin{array}{lll} \bar{v}_{a+l-i+1} & = \bar{y}_i & i = b \dots 1 \\ \bar{v}_j & = \bar{v}_j + \bar{v}_{i+n} \cdot \frac{\partial}{\partial v_j} \varphi_i(w_i), \bar{v}_i = 0 & i = l \dots 1 \\ \bar{u}_i & = \bar{v}_i & i = a \dots 1, \end{array}$$

Figure 2. Reverse mode run.

giving

$$\bar{u} = D_u \mathcal{G}^T(\boldsymbol{x}, \boldsymbol{u}) \cdot \bar{\lambda}, \tag{13}$$

which is exactly what we need for (8).
The information (12) is stored in a tape. To reference it to the adjoint values, they are internally assigned indices, incremented for each and every statement being executed. To have access to the desired derivatives, all input values have to be registered which will assign them indices before running the program. The same accounts for the output values to indicate where the adjoint values $\bar{\lambda}$ have to be applied.
Therefore, to proceed from the black-box evaluation in (8) to the modular evaluation that we need for (11), all we need to do is to *store solution variable indices in a way that we can relate them to their zones and solvers.*
Another aspect we have to consider is the transfer of solution data between the solvers across zones. We will record them in a designated section within the tape. That way we can be sure that we are always evaluating diagonal and cross derivatives at the original solution state $\boldsymbol{u} = \boldsymbol{u}^n$ (and not at an updated one which would lead to incorrect derivatives).

B. Adjoint evaluation algorithm, subroutines and tape layout

For our implementation of Eq. (10) we will build the adjoint solution update algorithm upon the following subroutines:

- `ComputeAdjointsObjectiveFunction`: Initialises the adjoint value of J with `1.0` and then evaluates all derivatives $\frac{\partial J}{\partial u}(\boldsymbol{x}, \boldsymbol{u})$ for all variables in all zones.
- `ComputeAdjoints`: Takes the zone index j as an argument and will then initialize the adjoint values at indices of $\boldsymbol{u}_{(j)}^{(n+1)}$ (that we have saved during recording) with values of $\lambda_{(j)}^{i}$ to then evaluate the derivatives $\frac{\partial \mathcal{G}_{(j)}}{\partial u}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda_{(j)}$ for all variables in all zones.
- `Iterate`: Takes the zone index k as an argument and extracts the current derivative values (be it $\frac{\partial \mathcal{G}_{(j)}}{\partial u_{(k)}}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda_{(j)}$ or $\frac{\partial J}{\partial u_{(k)}}(\boldsymbol{x}, \boldsymbol{u})$) at indices of $\boldsymbol{u}_{(k)}^{n}$ to the adjoint solution vector in zone k.
- `GetExternal`: Besides the adjoint solution vector, every zone needs a separate data structure that we refer to as `External`. The reason being that we can only evaluate derivatives for individual pairs (j, k), though for a full adjoint solution update in zone k we need all cross contributions from zones $j \neq k$, and the objective function gradient, this sum is stored in `External`.
- `UpdateCrossTerm`: As soon as the adjoints of zone j are updated, we want to update its cross contributions to zones $k \neq j$, to then perform the update of the next zones using the most current value of its `External`. This is accomplished by adding the difference between the current cross term ($\frac{\partial \mathcal{G}_{(j)}}{\partial u_{(k)}}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda_{(j)}$, as retrieved by `Iterate` from the AD tape) and the previous cross term, to the `External` of zone k. This strategy requires us to keep a table of past cross terms (formerly a block sparse matrix with vector blocks) but it allows inclusion of under/over-relaxation and the objective function gradient to be computed only once. The subroutine takes zone indices j and k as inputs and will update the `External` vector in zone k with derivatives with respect to zone j.

An outline of the algorithm to obtain the adjoint solution as the fixed point of the sequence $\lambda^{(i)}$ is given in terms of these subroutines by

```
ComputeAdjointsObjectiveFunction();
for $j \leftarrow 1$ to $r$ do
    AddSolutionToExternal($k$, Iterate($k$));
end
for $i \leftarrow 1$ to OuterIter do
    for $j \leftarrow 1$ to $r$ do
        for $l \leftarrow 0$ to InnerIter do
            $\lambda[j]$ = Sum($\lambda[j]$, GetExternal($j$));
            ComputeAdjoints($j$, $\lambda[j]$);
            $\lambda[j]$ = Iterate($j$);
        end
        for $k \leftarrow 1$ to $r$ do
            if $k \neq j$ then
                $\lambda[k]$ = Iterate($k$);
                UpdateCrossTerm (j,k);
            end
        end
    end
end
```

Algorithm 1: Discrete adjoint multizone driver algorithm.

The number of inner iterations L can be set by the user of the program and can serve as a stabilizer for the coupled outer loop.
In order to evaluate derivatives for a pair (j, k), the `ComputeAdjoints` does not have to go through the whole tape including all solvers but just through the necessary parts (mainly $\mathcal{G}_{(j)}$) which will be computationally more efficient. To do so, the tape is set up in a defined way so that we can refer to specific section afterwards. The layout can be summarized as in Fig. 3.

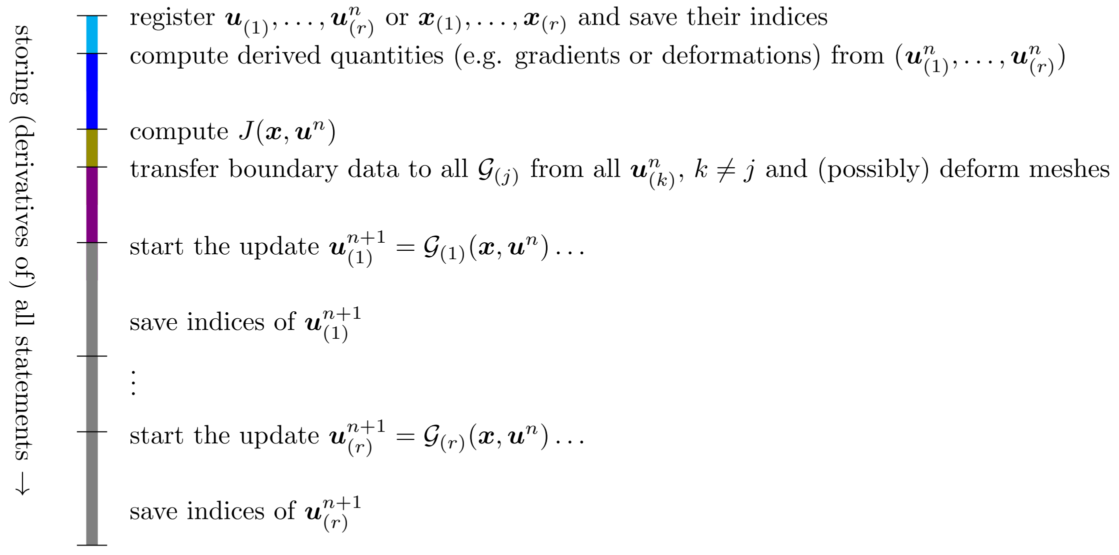
storing (derivatives of) all statements $\rightarrow$
register $\boldsymbol{u}_{(1)}, \ldots, \boldsymbol{u}^n_{(r)}$ or $\boldsymbol{x}_{(1)}, \ldots, \boldsymbol{x}_{(r)}$ and save their indices
compute derived quantities (e.g. gradients or deformations) from $(\boldsymbol{u}^n_{(1)}, \ldots, \boldsymbol{u}^n_{(r)})$
compute $J(\boldsymbol{x}, \boldsymbol{u}^n)$
transfer boundary data to all $\mathcal{G}_{(j)}$ from all $\boldsymbol{u}^n_{(k)}$, $k \neq j$ and (possibly) deform meshes
start the update $\boldsymbol{u}^{n+1}_{(1)} = \mathcal{G}_{(1)}(\boldsymbol{x}, \boldsymbol{u}^n) \ldots$
save indices of $\boldsymbol{u}^{n+1}_{(1)}$
$\vdots$
start the update $\boldsymbol{u}^{n+1}_{(r)} = \mathcal{G}_{(r)}(\boldsymbol{x}, \boldsymbol{u}^n) \ldots$
save indices of $\boldsymbol{u}^{n+1}_{(r)}$

Figure 3. Proposed tape layout for a multiphysics solver.

A helper function `AD_ComputeAdjoints` with two arguments – the point where it enters the tape for derivative evaluation and where it leaves it, illustrated by horizontal hyphens in Fig. 3 – provides an interface to the AD tool to evaluate only a certain scope within the tape. This makes the implementation of `ComputeAdjoints` very simple. E.g. in order to compute $\frac{\partial \mathcal{G}_{(j)}}{\partial u}^{\mathsf{T}}(\boldsymbol{x}, \boldsymbol{u}) \cdot \lambda_{(j)}$, we need three calls of `AD_ComputeAdjoints` in case we incorporate the transfer of boundary data to have derivatives ready for extraction:

```
InitializeAdjoint(j, λ[j]);
AD_ComputeAdjoints(Iterator_Offset +(j − 1) ∗ 2 + 1, Iterator_Offset +(j − 1) ∗ 2);
AD_ComputeAdjoints(Transfer, Objective_Function);
AD_ComputeAdjoints(Derived_Quantities, Start);
```

In practice the `Transfer` region of the tape is only evaluated in the last inner iteration (see algorithm 1), as for FSI problems this section includes the (computationally expensive) mesh deformation algorithm, and in general one only needs the section when cross terms are to be extracted from the tape. Furthermore, during the recording of the deformation operation, we introduce the simplification that the deformation stiffness matrix is a constant (i.e. not a function of the variables) as this nearly halves the memory usage. This approximation is consistent with what is done to project the volumetric sensitivities onto the design surface modified by the shape parameterization approach. For similar memory usage concerns, and general differentiability of the methods involved, we also assume the interpolation operators used to transfer fluid forces and structural displacements between domains to be constant.

IV. Applications

In this section, two multiphysics examples (one for CHT, one for FSI) will showcase the multiphysics adjoint functionality being of such type where there are are only two zones and $\mathcal{G}_{(1)}$ is a flow solver.
In the first case, the second zone is a rigid material that conducts heat and transfers it into the fluid ($\mathcal{G}_{(2)}$ is a heat solver), whereas in the second case we have an elastic material that deforms under flow tractions ($\mathcal{G}_{(2)}$ is a FEM elasticity solver).
For both cases there exist previous studies that show reasonable results in terms of correctness of adjoints and derived sensitivities. In,[8] the authors present an implementation for CHT discrete adjoints in the open-source software OpenFOAM[9] for analyzing temperature sensitivities in a heat sink by succesfully applying a black-box approach. In contrast, let us now demonstrate the modular approach of the present paper by

writing out the defining fixed-point iteration in the case $r = 2$ as

$$\begin{pmatrix} \lambda_{(1)}^{i+1} \\ \lambda_{(2)}^{i+1} \end{pmatrix} = \begin{pmatrix} \frac{\partial J}{\partial u_{(1)}} & \frac{\partial J}{\partial u_{(2)}} \end{pmatrix}^{\mathsf{T}} + \begin{pmatrix} \frac{\partial \mathcal{G}_{(1)}}{\partial u_{(1)}} & \frac{\partial \mathcal{G}_{(1)}}{\partial u_{(2)}} \\ \frac{\partial \mathcal{G}_{(2)}}{\partial u_{(1)}} & \frac{\partial \mathcal{G}_{(2)}}{\partial u_{(2)}} \end{pmatrix}^{\mathsf{T}} \begin{pmatrix} \lambda_{(1)}^{i} \\ \lambda_{(2)}^{i} \end{pmatrix}.$$

By design of Algorithm 1, the matrix-vector multiplication on the right hand side is not done in one single operation but is split into different steps.
For example, to obtain the updated adjoint solution for zone 1, $\lambda_{(1)}^{i+1}$, $\lambda_{(1)}^{i}$ is updated by evaluating the diagonal term $\frac{\partial \mathcal{G}_{(1)}}{\partial u_{(1)}}^{\mathsf{T}} \cdot \lambda_{(1)}^{i}$ and adding the external contribution from the last outer iteration, $\frac{\partial \mathcal{G}_{(2)}}{\partial u_{(1)}}^{\mathsf{T}} \cdot \lambda_{(2)}^{i-1}$ (via `GetExternal`) for InnerIter times.
Note that right after having updated $\lambda_{(1)}^{i}$ to $\lambda_{(1)}^{i+1}$, the cross term $\frac{\partial \mathcal{G}_{(1)}}{\partial u_{(2)}}^{\mathsf{T}} \cdot \lambda_{(1)}^{i}$ – that will act as a contribution to $\lambda_{(2)}^{i+1}$ – also is updated (via `UpdateCrossTerm`).

Note the physical interpretation of such cross terms. In CHT,

- $\frac{\partial \mathcal{G}_{(1)}}{\partial u_{(2)}}$ constitutes the flow solver's dependence on the temperature distribution at the interface and
- $\frac{\partial \mathcal{G}_{(2)}}{\partial u_{(1)}}$ constitutes the heat solver's dependence on the heat fluxes at the interface,

whereas in FSI,

- $\frac{\partial \mathcal{G}_{(1)}}{\partial u_{(2)}}$ constitutes the flow solver's dependence on the fluid mesh displacements with respect to the initial (undeformed) grid coordinates. The displacements are an explicit function of the initial coordinates and of the fluid-structure interface displacements. In previous implementations in SU2 so called three-field approaches were used (mesh displacements being the third) but here the two-field approach naturally fits the concept of *zone*.
- $\frac{\partial \mathcal{G}_{(2)}}{\partial u_{(1)}}$ constitutes the elasticity solver's dependence on the flow tractions at the interface.

To find optimal shapes with respect to a prescribed quantity of interest, we start with an initial design (e.g. in the first example, this will simply be a cylinder) based on which we compute the coupled adjoint solution for all zones.
In order to transform this information into shape gradients, we need a shape parametrization in the first place. In a simple but very instructional case, this could just be given by the surface node coordinates. Later on, one should work with much smaller gradient dimensions that are more suitable for optimization algorithms. In any case, let α be the vector of shape parameters and

$$(x_{(1)}, \ldots, x_{(r)}) = M(\alpha) \tag{14}$$

the map that gives us all domains in terms of their meshes. (Regarding the first option, where α is the vector of all surface node positions, M could be a mesh deformation algorithm based on an inverse volume-elasticity solver.)
Instead of obtaining gradients with respect to (volume) mesh coordinates x as in Eq. (4), we now want them with respect to α, so we set up an adapted version,

$$L(x, u, \alpha) := J(x, u, \alpha) + (\mathcal{G}(x, u) - u) \cdot \lambda + (M(\alpha) - x) \cdot \mu. \tag{15}$$

As before, in order to have

$$D\tilde{J}(\alpha) = \frac{\partial}{\partial \alpha} L(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\alpha})$$

We now do not only require $\frac{\partial}{\partial u} L \overset{!}{=} 0$ (which gives us the exact same adjoint equation for λ) but also $\frac{\partial}{\partial x} L \overset{!}{=} 0$.
We achieve this by solving for the appropriate μ,

$$\mu = \frac{\partial J}{\partial x}^{\mathsf{T}} (\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\alpha}) + \frac{\partial \mathcal{G}}{\partial x}^{\mathsf{T}} (\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\alpha}) \cdot \lambda, \tag{16}$$

which is just an evaluation without the need of an iterative scheme. We then have

$$\boxed{\nabla_\alpha L(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\alpha}) = DM(\boldsymbol{\alpha})^\mathsf{T} \cdot \mu} \tag{17}$$

which is the sum of the zone-wise contributions $DM^\mathsf{T}_{(k)}(\boldsymbol{\alpha}) \cdot \mu_{(k)}$.

A. Conjugate heat transfer test case

This test case is for steady state, turbulent, incompressible flow through a 2D array of heated cylinders. A typical pin array for e.g. power electronics cooling can consist of tens up to a couple of thousand individual pins. Instead of a simulation on the entire pin array a common simplification is to only use a characteristic unit cell, see Fig. 4. Through mirroring at symmetry boundaries and translating at periodic boundaries one recovers an approximation to the initial full array. The presented simulation setup includes several simplifications which have to be validated using a full size pin array. It is however within reason to presume that general design rules can be derived from such a reduced, fast to evaluate, unit cell approach.
The geometry of the simulation domain is fully characterized by three quantities: distance between pin midpoints 6.44 mm (three neighboring pins form an equilateral triangle), the inner pin radius 0.6 mm and outer pin radius 2 mm. Inlet and outlet in the fluid domain are prescribed as periodic boundaries.[10] Symmetry boundaries on top and bottom are indicated in Fig. 4 and handling of the interfaces between fluid and solid is described in literature.[2] On the inner pin arc in the solids a heatflux of 5e5 W/m^2 is prescribed and all other non-interface walls are adiabatic.

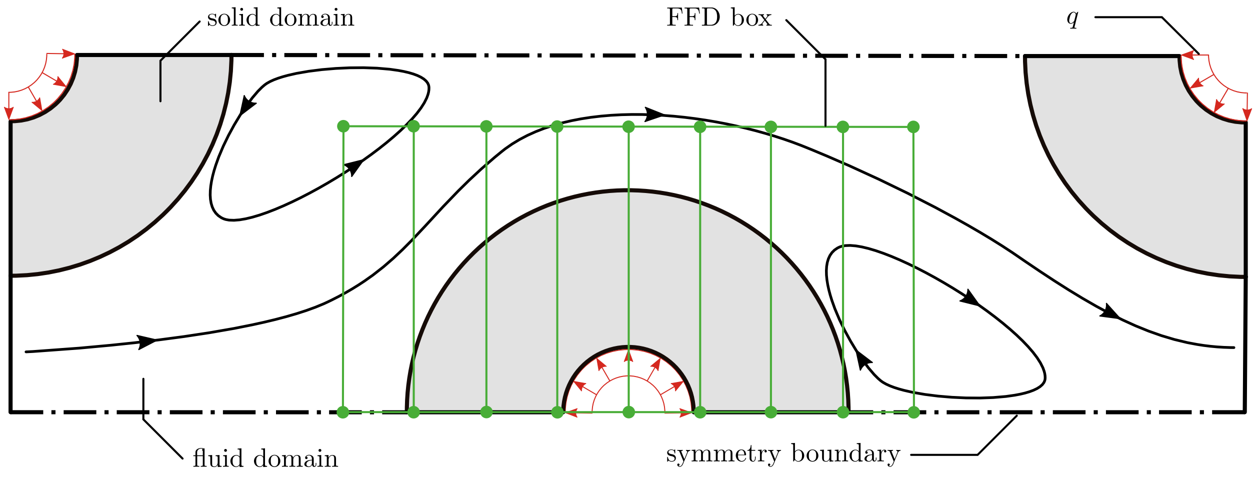

Figure 4. Simulation setup schematic. Heatflux boundary conditions with magnitude q indicated with red arrows and symmetry planes with a dash-dotted line. The FFD box in green encloses the half pin.

The pressure drop over the domain is set to 208 Pa and the inlet bulk temperature to 338 K. Other constant fluid properties are density $\rho = 1045\,\text{kg/m}^3$, specific heat $c_p = 3540\,\text{J/(kg K)}$, laminar viscosity $\mu_{dyn} = 1.385\text{e-}3\,\text{Pa s}$, laminar and turbulent Prandtl numbers $\text{Pr}_{lam} = 11.7$ and $\text{Pr}_{turb} = 0.9$. The fluids material properties represent Glysantin in a 50/50 mixture with water, a commonly used liquid for automotive cooling circuits. The constant material properties in the solid represent Aluminium: density $\rho = 2719\,\text{kg/m}^3$, specific heat $c_p = 871\,\text{J/(kg K)}$ and thermal conductivity $\lambda = 200\,\text{W/(m K)}$. Convective fluxes are discretized using a Flux-Difference-Splitting (FDS) with second order accuracy achieved by a MUSCL approach.[11] No limiters are used for variable reconstruction and gradients are computed via the Green-Gauss theorem. The Menter SST turbulence model without wall function is used.
The computational mesh is fully structured, e.g. only quadrilaterals are used, and consists of 8477 elements in the fluid and 6747 elements in the solid. A $y^+ < 1$ is obtained everywhere but the mesh is too coarse in order to ensure a mesh independent solution. The purpose here is solely to verify the gradient accuracy of the discrete adjoint method. Nonetheless temperature contour lines are given in Fig. 5 that show a thin temperature boundary layer on the middle pin and a noticeable heat convection into the fluid body at the

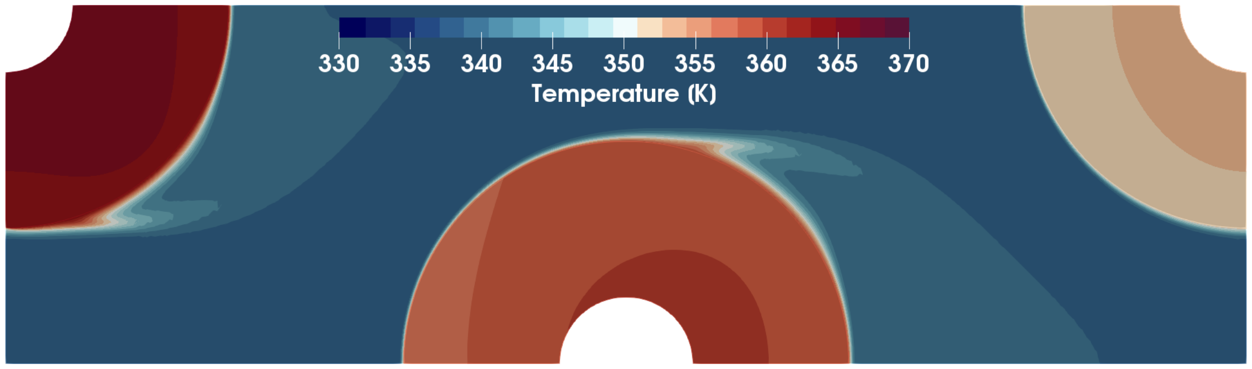

Figure 5. Temperature contour lines.

separation point at approximately 1 o'clock. Note that the quarter pins are not periodically connected in this simulation which leads to a considerable temperature difference compared to the examined half pin.
As geometry parametrization a free-form-deformation (FFD) box encloses the half-pin in the center of the simulation domain, see Fig. 4 and consists of 18 control points. For the gradient validation only the vertical movement of the upper row of control points is considered i.e. 9 design variables are examined. Since a constant heatflux is prescribed, the temperature of the pin is determined by the cooling performance of the pin shape, i.e. the geometry of the fluid-solid interface. Therefore the average temperature on the inner arc of the center pin is set as an objective function and the interface as the region to be designed. Other boundaries are fixed to preserve symmetry. Note that for simplicity the gradient only considers changes to the center pin and not equivalent changes to the quarter pins.
The gradient obtained via the discrete adjoint method shows a good agreement to first order finite differences with stepsize 10^{-8}, see Fig. 7. The gradients are nondimensionalized from K/m by multiplying with outer pin diameter divided by objective function value of 360.12 K. The finite difference step size was determined by continuously lowering the stepsize until rounding errors become dominant, see Fig. 6 for one example evaluation. The CHT adjoint testcase has a maximum memory consumption of 2 GB which is approximately 2 times the memory used by the primal evaluation. The same ratio is observed in the compute time which is about 1000 CPU seconds for the adjoint evaluation and was run on 14 cores (Intel Xeon Gold 6132 CPU). A compute time comparison highly depends on convergence criteria and is only given to show the comparable compute cost between primal and adjoint evaluation.

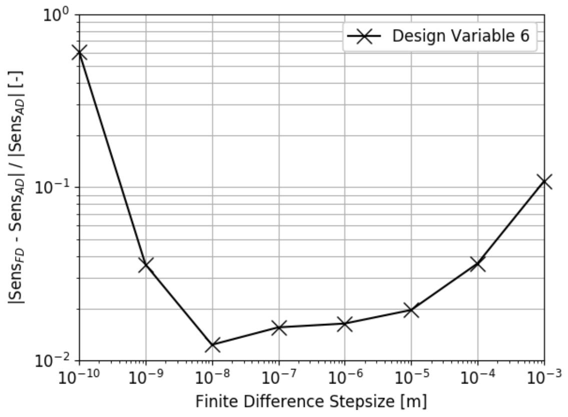

Figure 6. Relative error of the 1st order finite difference sensitivity to the discrete adjoint sensitivity at decreasing stepsize.

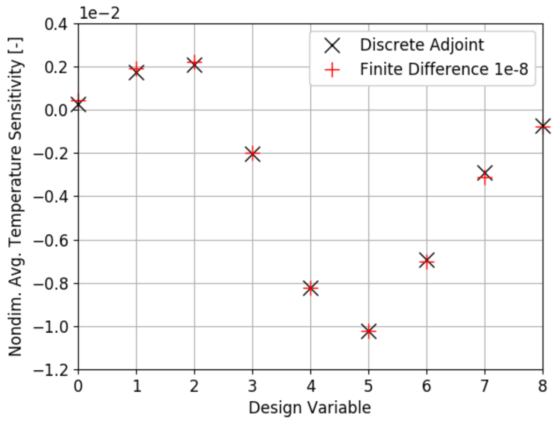

Figure 7. Nondimensional sensitivity for the discrete adjoint solver and finite differences. Objective function is average temperature on the inner middle pin.

B. Fluid-structure interaction test case

To verify the shape sensitivities for FSI problems we consider the simulation of a flexible wing at low Mach number (0.6) and 4 degrees angle-of-attack (AoA). The wing geometry (Fig. 8) is generated by *lofting* two symmetric 4-digit NACA profiles, a 0.25 m chord 9% thickness profile at the root, and a 0.175 m chord 7.2% thickness profile at the tip. The wing span is 1 m, with the 0.25c line swept back 5 degrees, and a linear twist distribution of -3 degrees to prevent static divergence. The primal simulation methodology has been described in the literature,[12] thus here we describe only the choice of numerical schemes. We approximate the wing structure as a neo-Hookean hyper-elastic solid with elasticity modulus of 7.5 GPa and Poisson's ratio of 0.35, with these properties the vertical displacement of the wing tip is approximately 20% of chord (see Fig. 9). On the fluid side air at standard temperature and pressure is considered. Both fluid and structural grids are composed mostly of hexahedra, the former having 832 000 nodes and the latter 174 000 nodes respectively. We note that the fluid grid is not sufficiently fine for detailed flow field analysis, as $y^+ \approx 5$ and we use Menter's SST turbulence model without wall functions. However, it is adequate for verification of discrete adjoint sensitivities (whose accuracy is not mesh dependant). Convective fluxes are discretized with a second order Roe scheme, the gradients are computed via the Green-Gauss theorem, and the flow variable reconstruction limited with Venkatakrishnan and Wang's limiter.

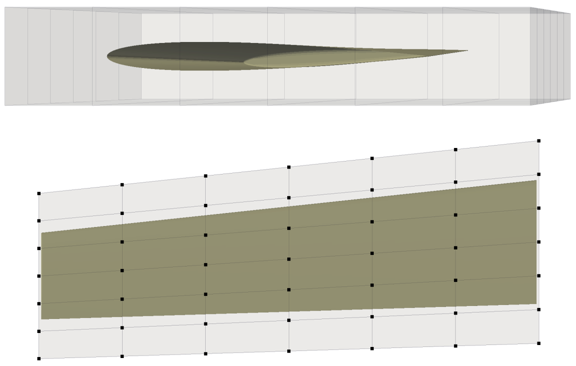

Figure 8. Wing geometry and free-form-deformation box, viewed from the wing tip and the top.

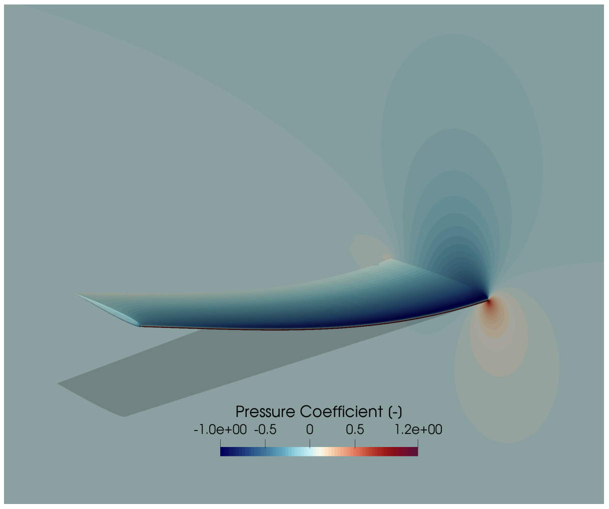

Figure 9. Deformed configuration of the wing and pressure coefficient contours.

The free-form-deformation (FFD) box shown in Fig. 8 has 98 control points, rather than verifying derivatives for each point (and each Cartesian coordinate), we define stretch (acting on chord) and translation variables for each spanwise section of points. We will refer to these 14 variables as S_{0-6} and T_{0-6} respectively (starting the numbering at the root) and normalize them taking as reference value 0.05 m (20% of root chord). The gradients of drag coefficient (C_d) and elastic energy are verified, we chose these functionals as the first is fluid oriented, whereas the second is mostly structural. The values of the functionals are also normalized (by 0.01 and 75 J respectively) which leaves both gradients dimensionless and of comparable magnitudes.
A suitable finite difference step size was determined by conducting a convergence study, starting with a step size of 0.05 and halving it each time we obtained second order central approximations to the gradients. At step sizes below 0.0125 the approximations start to diverge due to the typical limitations of finite differences, as the variation of the functionals approaches the accuracy to within which the primal problem can be converged ($\approx 10^{-6}$ change over FSI iterations). Therefore gradients obtained with step size of 0.025 were considered. The error associated with the approximation, used to compute lower and upper confidence bounds, was estimated by taking the average plus three standard deviations of the differences between second and fourth order central approximations. The latter being computed using also the function values at ± 0.05. The results for drag are plotted in Fig. 10, together with the sensitivity values computed via the discrete adjoint approach. Fig. 11 shows the results for the elastic energy functional.
The agreement between finite difference approximations and adjoint sensitivities of drag is better for the translation variables (T_{0-6}), to some extent this was expected due to the larger magnitude of those deriva-

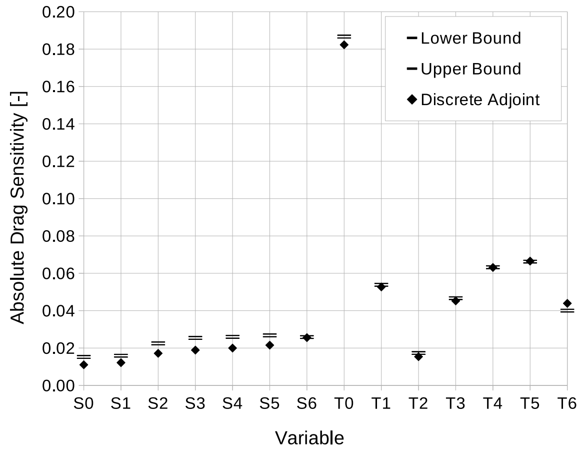

Figure 10. Absolute value of drag sensitivities, from the adjoint solver, and finite difference error bounds.

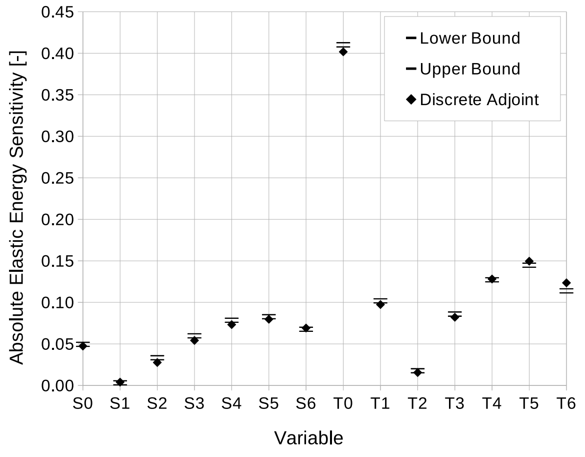

Figure 11. Absolute value of elastic energy sensitivities, from the adjoint solver, and finite difference error bounds.

tives, and the way those variables are constructed from the FFD control points (a translation variable moves all the points in a section by an equal amount, whereas a stretch variable moves points away from the centerline of the box proportionally to their distance to it). The overall agreement is also better for the elastic energy functional, this is also expected as the elastic energy depends strongly on pressure and structural deformation (i.e. almost directly on the solution variables) whereas drag at subsonic speed depends mostly on viscous effects (and therefore on gradients of the solution variables).
Since due to the typical limitations of finite differences, it was not possible to obtain derivative approximations with a small-enough step size (the one used is 2.5% of the plausible range for use in optimization) we cannot fully attribute the discrepancies to the memory-saving approximations introduced in subsection III B. Our current work includes improving the forward-mode AD capabilities of SU2 to be able to accurately differentiate the more challenging linear systems that arise from structural problems (including mesh deformation).
A typical challenge of implementing an algorithmic differentiation-based discrete adjoint solver in a non intrusive manner, i.e. not requiring extensive modification of the architecture of the code, and/or of the implementation of the algorithms, is maintaining a low memory footprint. Running the adjoint FSI problem used for verification requires 113 GB of memory, or approximately 4 times the memory required to run the primal simulation. We note however that while multi-grid acceleration was used for the primal solution, it was not for the adjoint solution, doing so roughly doubles the memory footprint of the fluid problem, which would increase the adjoint-to-primal memory ratio to ≈ 6. Notwithstanding not using multi-grid, the adjoint solution takes approximately the same time as the primal, about 15 minutes running on 48 cores (4 Intel Xeon E5-2650v4 CPU).

V. Conclusions

This paper has presented a methodology on how to set up and implement an algorithm to compute discrete adjoints for multiphysics problems using algorithmic differentiation, giving accurate gradients for optimization purposes.
By design, no a priori knowledge on the physical setup of the problem was required so that we could apply our implementation for a conjugate heat transfer and a fluid-structure interaction problem without running different code sections for one or the other.
To the authors' knowledge, no such unifying approach has been pursued so far within the optimization community. Besides all apparent advantages of using a common code base (like ease of maintenance and compatibility), it also opens the way to include further kinds of physics to ultimately tackle complex multiphysics optimization problems where many individual solution algorithms would repeatedly demand a notable amount of implementation effort.

VI. Acknowledgements

The authors would like to gratefully acknowledge the support of the Bayerische Forschungsstiftung (BFS AZ-1232-16) as well as the computational resources provided by the RHRK high performance computing center via the Elwetritsch high performance cluster at the TU Kaiserslautern.

References

[1]Thomas D Economon, Francisco Palacios, Sean R Copeland, Trent W Lukaczyk, and Juan J Alonso. Su2: An open-source suite for multiphysics simulation and design. *Aiaa Journal*, 54(3):828–846, 2015.

[2]Ole Burghardt, Nicolas R Gauger, and Thomas D Economon. Coupled adjoints for conjugate heat transfer in variable density incompressible flows. In *AIAA Aviation 2019 Forum*, page 3668, 2019.

[3]Antony Jameson. Aerodynamic design via control theory. *Journal of scientific computing*, 3(3):233–260, 1988.

[4]M.B. Giles and N.A. Pierce. Adjoint equations in cfd-duality, boundary conditions and solution behaviour. In *13th Computational Fluid Dynamics Conference*, page 1850, 1997.

[5]T. Albring, M. Sagebaum, and N.R. Gauger. Efficient aerodynamic design using the discrete adjoint method in SU2. *AIAA Paper 2016-3518*, 2016.

[6]M. Sagebaum, T. Albring, and N. R. Gauger. High-Performance Derivative Computations using CoDiPack. *ACM Transactions on Mathematical Software*, 2019.

[7]Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2008.

[8]Markus Towara, Johannes Lotz, and Uwe Naumann. Discrete adjoint approaches for cht applications in openfoam.

[9]Henry G Weller, Gavin Tabor, Hrvoje Jasak, and Christer Fureby. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Computers in physics*, 12(6):620–631, 1998.

[10]S.V. Patankar, C.H. Liu, and E.M. Sparrow. Fully developed flow and heat transfer in ducts having streamwise-periodic variations of cross-sectional area. *Journal of Heat Transfer*, 99(2):180–186, 1977.

[11]Thomas D. Economon. Simulation and adjoint-based design for variable density incompressible flows with heat transfer. *AIAA Journal*, 58(2):757–769, 2020.

[12]Pedro Gomes and Rafael Palacios. Aerodynamic driven multidisciplinary topology optimization of compliant airfoils. In *AIAA Scitech 2020 Forum*.