CAD-integration for Gradient-based Shape Optimization using the Algorithmically Differentiated pythonOCC Library

Mladen Banović[1], Adam Büchner[1], Thomas E. Hafemann[1], Patrick Wegener[2], and Arthur Stück[1]

[1] German Aerospace Center (DLR)
Institute of Software Methods for Product Virtualization
Nöthnitzer Str. 46b, 01187 Dresden, Germany
Mladen.Banovic@dlr.de
[2] German Aerospace Center (DLR)
Institute of Aerodynamics and Flow Technology
Lilienthalplatz 7, 38108 Braunschweig, Germany

Abstract. pythonOCC is a library that provides Python bindings for the Open CASCADE Technology (OCCT) C++ geometric modelling kernel. To integrate it into a gradient-based shape optimization, one requires to compute the so-called geometric sensitivities, e.g., derivatives of surface nodes with respect to the design parameters. To obtain this information, pythonOCC and OCCT were algorithmically differentiated. Here, they are modularly integrated in the form of a CAD plugin into a framework for multidisciplinary design analysis and optimization (MDAO) based on the DLR's FlowSimulator HPC ecosystem. The CAD plugin allows a robust and metadata-enabled mesh-to-CAD association between MPI domain-decomposed mesh objects and the underlying CAD patches, as well as the computation of geometric sensitivities. The framework integration of pythonOCC is demonstrated in a context of gradient-based, aerodynamic shape optimization for a RAE2822 configuration in fully turbulent, transonic flow.

Keywords: Gradient-based optimization, CAD-based optimization, Algorithmic Differentiation (AD), Geometric sensitivity, Open CASCADE Technology (OCCT)

1 Introduction

Computer-Aided Design (CAD) plays a vital role in simulation-based design analysis and optimization, which is often a multidisciplinary task and can involve multiple levels of fidelity. This calls for flexible and user-friendly CAD solutions linked to simulation and optimization.

A CAD-based approach pursued in this work offers several advantages. First, a CAD geometry is integrated into an MDAO framework, which makes it a central cornerstone coupled between different computational meshes and disciplines

such as CFD (*Computational Fluid Dynamics*) or CSM (*Computational Structural Mechanics*). Second, it allows a straightforward definition of *part* or *assembly* constraints that have to be fulfilled during an optimization. Third, an optimal shape exists in a CAD-specific format as opposed to CAD-free approaches that optimize directly on the computational meshes.

Nonetheless, an implementation of the CAD-based approach comes with a number of technical challenges, two of which are addressed in this study. In particular, one requires the following capabilities: (i) a robust and systematic association between the computational meshes and underlying CAD topological entities and (ii) the computation of the geometric (CAD) sensitivities in the context of a gradient-based shape optimization that is considered in this work.

Gradient-based optimization methods are recognized for their efficiency—which is beneficial when a design chain is associated with high-fidelity coupled simulations that typically impose significant computational costs. However, to employ these methods, one requires to compute derivatives for each component of the design chain in order to accumulate the total derivatives of objective functionals and constraints. In CFD packages, the adjoint approach is considered as state-of-the-art for computing gradients of an objective function w.r.t. mesh node coordinates [11, 7, 16]. The next challenge is the computation of CAD sensitivities to systematically complement the sensitivity chain.

In conjunction with black-box CAD software (e.g., proprietary tools), the derivatives are usually approximated using Finite Differences (FD) [1]. In this case, the derivatives are affected by truncation or cancellation errors. A systematic fine-tuning of step sizes can quickly become tedious. Another issue is computational efficiency as the costs of computing derivatives linearly scale with the total number of design parameters. Nevertheless, this method is widely used for industrial applications where closed-source CAD solutions are employed.

On the contrary, one can apply Algorithmic Differentiation (AD) to the source-code of the CAD software (given that it is available) in order to compute exact sensitivities. Moreover, the computational efficiency of reverse-mode AD is superior to FD when many design parameters are considered. For instance, this was demonstrated in the past by differentiating a general-purpose and open-source CAD kernel Open CASCADE Technology (OCCT) [3].

Today, Python-controlled MDAO processes and frameworks are well established in conjunction with complex industrial workflows. For this reason, the CAD library pythonOCC [15] is employed in this work as it offers Python extensions for the OCCT kernel. AD of such hybrid Python/C++ environments represents a challenge and requires a mixed-language AD approach [14]. The forward differentiation of pythonOCC was tackled in [2] by implementing a Python extension for the AD tool ADOL-C. Here, the AD-enabled OCCT and pythonOCC are employed to differentiate a functionality from the TiGL geometry library [18]. Moreover, we demonstrate a systematic integration of the differentiated pythonOCC software as a CAD component into a gradient-based MDAO workflow.

This paper is structured as follows. Section 2 describes a verification use-case considered in this work. Section 3 explains how the geometric sensitivities are computed and verified. Section 4 elaborates the CAD-integration of the differentiated OCCT/pythonOCC into an MDAO framework. Moreover, it clarifies how the AD-enabled framework is employed for a gradient-based shape optimization, while the optimization results are given in Sect. 5. Finally, conclusions are drawn in Sect. 6.

2 Use-Case Parametrization

A simplified wing geometry considered for the validation of this study is described as follows. Its 2-D base profile is an RAE2822 airfoil parametrized by a method named Class-Shape Transformation (CST) [12]. This method allows analytic representations of various 2-D and 3-D shapes ensuring smooth surfaces, using only a modest number of design parameters.

The CST representation for a 2-D curve is defined separately for the suction and the pressure side of the wing profile, each having four Bernstein coefficients as the design parameters. To ensure a curvature-continuous leading edge, the first coefficient of the pressure side is derived from the first coefficient of the suction side (i.e., they have same values, but the opposite sign). Therefore, the total number of design parameters is equal to seven. An additional parameter defined here is the trailing edge thickness, however it is kept as a constant.

The CST analytical description does not exist in the OCCT/pythonOCC library and thus has to be converted in a compatible shape representation, which is the B-spline class in this case. For this purpose, a functionality from the TiGL geometry library is employed, namely the `CCSTCurveBuilder` class. This class first approximates the CST curve representation using Chebyshev polynomials. Depending on the desired tolerance for the approximation, multiple Chebyshev segments can be created. From each segment, a B-spline object equivalent to a Bézier curve is constructed, where the Chebyshev coefficients are transformed into the resulting control point coordinates. Finally, these B-splines are joined with C1 continuity into a single B-spline curve.

The two resulting B-spline curves (the suction and pressure sides) can then be used in pythonOCC. From here they are converted into edges and, together with the trailing edge, joined as a wire (edges and wires are topological representations). Since the default chord length is equal to one, the previously constructed wire is first scaled to a target value to create the base profile. After that, it is cloned, scaled again and transformed to the wing-tip position to create the tip profile. These two profiles are given as inputs to the *lofting* algorithm (implemented in the class `BRepOffsetAPI_ThruSections`) that constructs the final wing shape, as presented in Fig. 1.

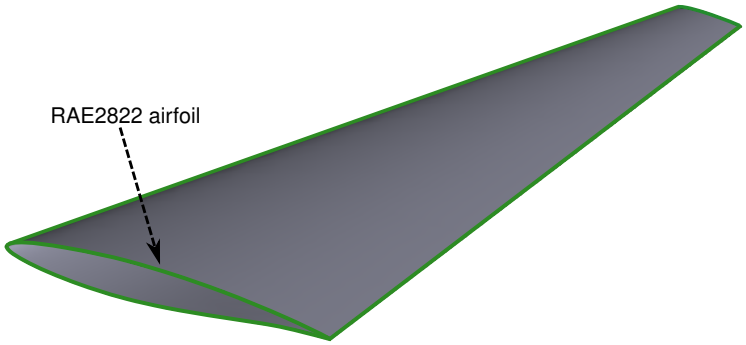

Fig. 1. Tapered 3-D wing geometry using an RAE2822 section.

3 AD-based Calculation of Geometric Sensitivities

The AD process to achieve the exact computation of geometric sensitivities in OCCT and pythonOCC (*v7.6.2*) is illustrated in Fig. 2 and explained as follows. First, the AD software tool employed in this study is ADOL-C [19]. This tool is based on the *operator-overloading* concept and it computes first and higher-order derivatives of vector functions written in C++. To integrate ADOL-C into a certain code, one replaces the declaration types of almost all *real* variables (e.g., declared as `float` or `double`) with the ADOL-C data-type `adouble`. There are two implementations of the `adouble` class (defined in different headers), offering distinct ways of derivative computation:

- *traceless*: provides the forward mode of AD,
- *trace-based*: provides the forward and the reverse mode of AD.

The *traceless* `adouble` class was considered as the first step of pythonOCC differentiation due to a few practical reasons. The *traceless* option computes derivatives *on-the-fly* along the *primal* evaluation and it is straightforward to extract the derivative information at any point in the computation. For this reason, it is easier to debug, which can be useful when differentiating complex hybrid environments such as pythonOCC. Finally, the forward mode of AD can serve as a reference when validating derivatives computed using the reverse mode of AD.

Second, the integration of ADOL-C into OCCT was performed by changing the alias `Standard_Real` (defined in OCCT) from `double` to `adouble`. This swap of data-types triggered a large amount of compile- and run-time issues that were tackled in order to finalize the differentiation process. More details about the common issues and their solutions can be found in [3].

Third, a Python interface for the *traceless* `adouble` class was generated using the software development tool SWIG (*Simplified Wrapper and Interface Generator*) [4]. SWIG creates a wrapper code for a wide variety of scripting languages (in this case Python) by parsing the so-called interface files (typically denoted with a suffix `.i`). In this interface file, SWIG expects C/C++ declarations and

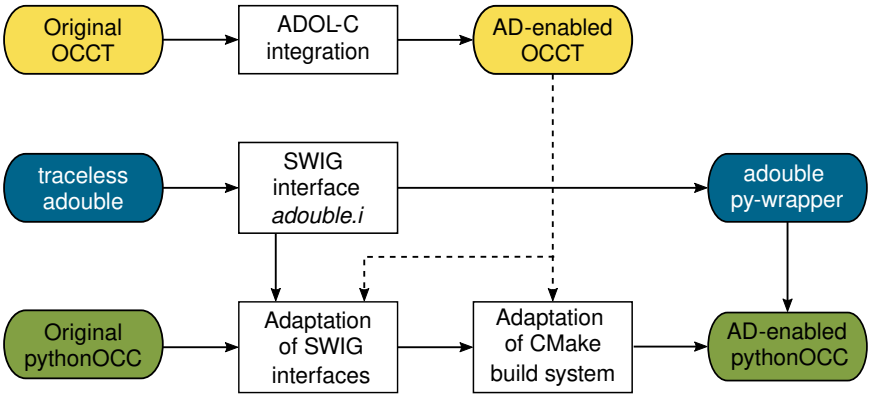

Fig. 2. AD of OCCT and pythonOCC.

special SWIG directives preceded by the % delimiter. Such file, namely the `adouble.i` interface, was developed for this purpose and its example can be found in [2]. Afterwards, one can invoke SWIG to build the desired `adoble` wrapper.

Finally, the differentiation of pythonOCC was performed as follows. It started by modifying the `CMakeLists.txt` file to add paths to the ADOL-C installation directory and the `adouble.i` interface. Moreover, the pythonOCC modules were linked against the ADOL-C library and the differentiated OCCT libraries. Next, the pythonOCC sources located in the `SWIG_files` directory were modified in this way. An include statement for the *traceless* `adouble` (`adolc/adtl.h`) was added to a top-level header, namely the `headers/Standard_module.hxx` file. The entry point for AD is the `wrapper/Standard.i` interface, in which `adouble.i` was imported. In the same file, the alias `Standard_Real` was changed from `double` to `adouble`. Afterwards, pythonOCC contains about 300 interface files corresponding to each OCCT source package. They were checked and accordingly adapted such that each class and its methods match the signatures of the differentiated OCCT sources.

3.1 Differentiation of the CST functionality

As mentioned in Sect. 2, the CST parametrization is employed in this work. A C++ implementation for this functionality is available in the `CCSTCurveBuilder` class of TiGL and compatible with OCCT. Briefly, it offers: (i) a user-defined constructor that accepts a vector of design parameters (among other arguments) and (ii) a method named `Curve` that returns a *handle* to the resulting `Geom_BSplineCurve` object (*handles* are smart pointers of OCCT).

There are in total six source files that enable the CST functionality. They are extracted from TiGL as a standalone project for the purposes of differentiation. Otherwise, the whole TiGL library would have to be differentiated, which is out of the scope of this study.

This extracted code is differentiated using ADOL-C and the amount of introduced changes is minimal, such as swapping the declaration type `double` with the alias `Standard_Real`. Moreover, it is linked against the differentiated OCCT. Another code adaptation is related to the *class* function of CST which is defined as follows:

$$C(\psi) = \psi^{N1}(1-\psi)^{N2}, \tag{1}$$

where:

- $\psi \in [0., 1.]$ is the chord-normalized coordinate,
- $N1 = 0.5$ and $N2 = 1.0$ are the exponents of the *class* function that in this setup ensure a round nose airfoil and a sharp trailing edge [12].

Due to $N1 = 0.5$, the *class* function is not differentiable at the leading edge, thus causing the derivative to become `NaN` (*Not a Number*) during program execution. To avoid having `NaN` values, an *if*-statement is added to the *class* function in order to return zero both for the *primal* and derivative values when $\psi = 0$. Elseways, the code related to Eq. 1 is executed as originally intended along with the derivative computation.

The next step is to generate a Python extension of the CST C++ implementation that is compatible with pythonOCC. This is achieved by implementing a SWIG interface for the `CCSTCurveBuilder` class. A code snippet of this interface is presented in Listing 1.1 and described as follows. Using the `%module` directive, one can define the module name as shown in Line 1. Lines 3–5 represent a block whose body (Line 4) is copied (as is stands) into the wrapper file and it is required to compile the interface. The same rule applies for Lines 8–11. As mentioned previously, the method `Curve` returns a *handle* to a B-spline object. There is an established procedure in pythonOCC how to deal with *handles* such that the user does work with them on the Python layer. For this purpose, one adds SWIG directives as shown in Lines 14–15. Furthermore, Lines 18–21 allow to declare an `std::vector` of *adoubles* (i.e., design parameters) in Python, which is needed for the `CCSTCurveBuilder` constructor. Next, the import directive (Line 24) serves to collect information from another SWIG interface (in this case the `Geom.i` file of pythonOCC) without generating any wrapper code. Finally, Line 26 instructs SWIG to generate the desired Python extension for the `CCSTCurveBuilder` class.

Listing 1.1. SWIG interface file *tigl_cst_builder.i* (simplified).

```
1  %module tigl_cst_builder
2
3  %{
4  #include "geometry/CCSTCurveBuilder.h"
5  %}
6
7  // common include dependencies from pythonOCC
8  %{
9  #include "headers/Standard_module.hxx"
10 #include "headers/Geom_module.hxx"
```

```
%}

// eliminate usage of OCCT's Handle on the Python layer
%include common/OccHandle.i
%wrap_handle(Geom_BSplineCurve)

// allow a vector of adoubles
%include "std_vector.i"
namespace std {
  %template(adVector) vector<adouble>;
};

// import interface for the Geom_BSplineCurve type
%import wrapper/Geom.i
// generate a wrapper for the CCSTCurveBuilder
%include "geometry/CCSTCurveBuilder.h"
```

As one can notice in Listing 1.1, the headers and interfaces of pythonOCC (in this case the differentiated version) are employed to create the Python wrapper for CST. This way, the CST functionality is compatible with pythonOCC as the same types are being used. Furthermore, the propagation of sensitivities from Python to C++ and vice-versa is ensured by ADOL-C and its Python extension.

3.2 Verification of geometric sensitivities

The correctness of the AD-computed derivatives is verified using the wing geometry described in Sect. 2. The corresponding AD-workflow when using the *scalar* forward mode of ADOL-C is described as follows. First, to activate a certain design parameter (i.e., mark it as an *independent* variable), one calls the method `setADValue` on the corresponding `adouble` object to set the derivative seed equal to one. After the seeding, one executes the parametrization code that now employs the AD-enabled CST and pythonOCC to create the wing shape and access its topological data structure. Finally, one obtains the underlying geometry—surfaces in this case—to calculate point coordinates (i.e., *dependent* variables) and their corresponding partial derivatives w.r.t. the activated design parameter. To extract the derivative information, one calls the `getADValue` method on an `adouble` object.

As a representative example, the geometric sensitivities are calculated w.r.t. one design parameter (a Bernstein coefficient) and compared against finite differences. As presented in Fig. 3, the overall magnitude plots of the AD- and FD-sensitivities match to a very high extent. The reason is that the design parameters have a linear influence on the CST function and subsequently on the resulting B-spline surface (for this parametrization). One could argue that the FD-computed sensitivities themselves would be sufficiently accurate for such cases. Nonetheless, one of the goals of this study is to demonstrate that AD can be successfully integrated into a hybrid Python/C++ geometry modeling workflow where various libraries are involved. For reference, a verification study for

the AD-enabled pythonOCC can be found in [2], in which a nonlinear influence of a design parameter on the geometric sensitivities was observed.

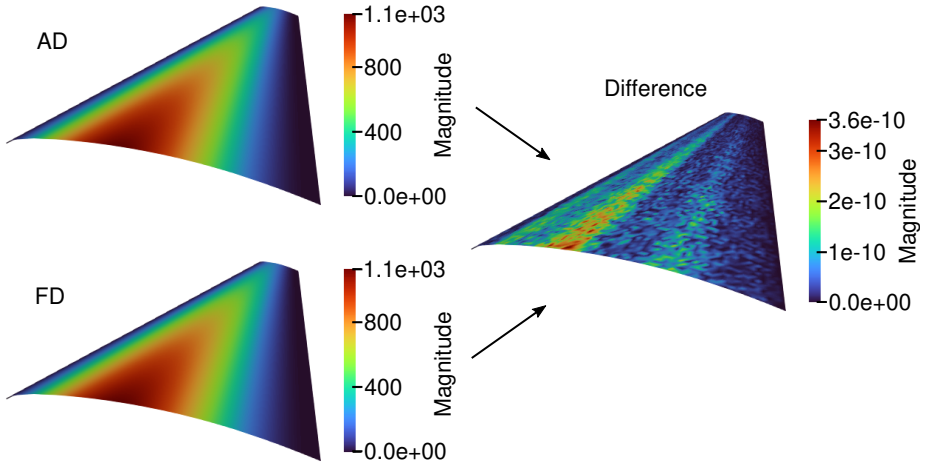

Fig. 3. Geometric sensitivities: AD vs. FD.

4 CAD-integration of the Differentiated pythonOCC into Optimization Workflows

This section elaborates on several ingredients that are used in conjunction with the AD-enabled OCCT/pythonOCC for a gradient-based shape optimization.

4.1 MDAO Framework based on FlowSimulator and its Components

FlowSimulator is DLR's centralized simulation framework, optimized for HPC applications aiming at the implementation of high-fidelity multidisciplinary simulations in the field of aeronautical design. Its core functionalities are implemented in C++, whereas a Python layer is used to control the multidisciplinary workflows.

The FlowSimulator's backbone is its data manager (abbreviated as FSDM) that allows to share large mesh datasets in memory among the simulation plugins in an MPI-parallel way. Additionally, it provides utility functions, such as conversion by reference of various datasets to *numpy* arrays. On top of the FSDM, one integrates various components (disciplines) in the form of the so-called plugins. That is, all plugins that work with meshes (e.g., a CFD solver, mesh deformation, etc.) communicate via FSDM in an MPI-distributed way.

Furthermore, on the basis of the FlowSimulator ecosystem, one can perform multidisciplinary analyses and optimizations in conjunction with gradient-enabled MDAO-frameworks like OpenMDAO [8] or GEMSEO [6]. For this purpose, a unified interface named FSMDAO is developed. The component interface systematically defines inputs and outputs for the MDAO workflow integration. For instance, each plugin requires the computation of the *primal* and the corresponding partial derivatives. Combined with OpenMDAO, the FSMDAO plugin allows an automated verification of the total gradient or the component-wise partial derivatives, which is typically evaluated before running an optimization process. A general representation of the framework is illustrated in Fig. 4. More details can found in [5, 9].

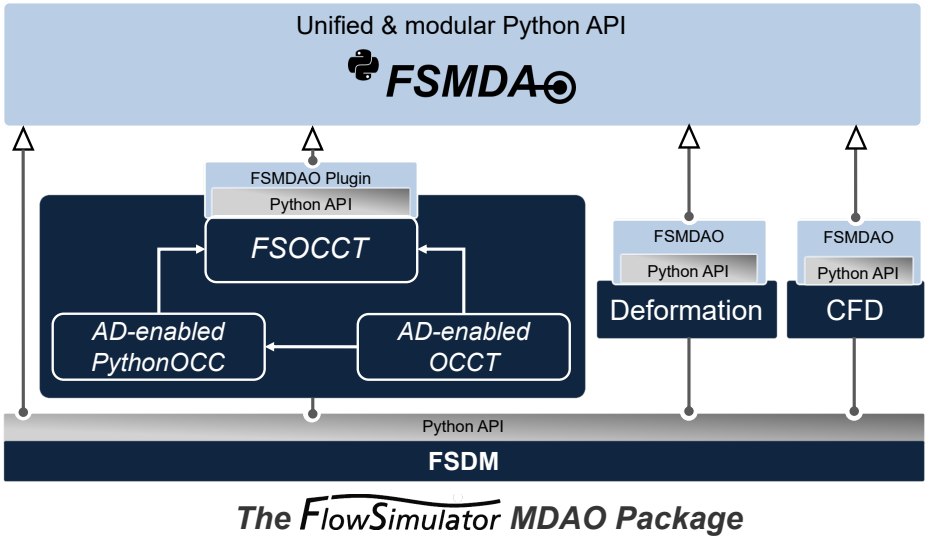

Fig. 4. FlowSimulator MDAO framework and main components.

The components/plugins employed in this work are related to the AD-enabled CAD (and mesh-to-CAD association), mesh deformation and CFD. They are briefly described in the following.

First, the integration of the differentiated OCCT into the FlowSimulator ecosystem was enabled by developing a CAD plugin named FSOpenCascade (or shortly FSOCCT). This plugin provides a robust metadata-supported mesh-to-CAD link, as well as the computation of geometric sensitivities. For the purposes of this study, FSOCCT is extended to support the differentiated pythonOCC. A detailed explanation follows in Sect. 4.2.

Second, the mesh deformation functionality is provided by the FSMeshDeformation plugin. It uses a linear elasticity approach to propagate the displacements of the surface mesh nodes into the volume mesh. Since the deformation

consists of a linear problem, the derivative of the mesh coordinates w.r.t. the surface displacements is calculated analytically using the solution of the linear system [17].

Finally, the new-generation CFD software CODA [13] is developed as part of a collaboration between the French Aerospace Lab ONERA, the German Aerospace Center (DLR), Airbus and their European research partners. It is a compressible, cell-centered CFD code implementing both traditional Finite Volume (FV), as well as modal and nodal Discontinuous Galerkin (DG) methods. CODA is a CFD *library* and an FSDM plugin by design, providing modular and fine-grained access to its residuals and other quantities. A key feature of CODA is its derivative support using AD, which automatically covers all implemented PDEs, convection schemes, turbulence models and boundary conditions. CODA provides exact derivatives of its residual, integral quantities and surface loads with respect to the flow solution and CFD mesh quantities, which can be accessed individually on the Python layer via a derivative API. In addition, the differentiation of CODA enables a robust convergence of the linear solver used in the non-linear solution process to machine precision. Like the other quantities, this linear system can be accessed via the Python API FSMDAO.

4.2 CAD Plugin: FSOpenCascade

FSOpenCascade provides access to BRep (*Boundary Representation*) data structures of OCCT and associated functionalities related to FSDM [9]. Its main feature is a metadata-supported mesh-to-CAD mapping that involves the projection of the surface nodes onto the underlying CAD objects, giving the FlowSimulator's plugins access to the real geometric representation. By employing the AD-enabled OCCT, it can also provide sensitivity calculations for mapped mesh nodes and a given CAD parametrization.

The mapping process between an existing mesh and the related CAD model is typically done as a preliminary step. Metadata information, such as face labels or colors, is collected from the CAD and mesh objects. Labels from the CAD model are cross-referenced with the labeling of mesh groups, such as the ones used for defining boundary conditions. A direct association between objects is only possible if the mesh grouping is in accordance with CAD labels and patches, or was created by the same CAD model. To address cases where the direct association is not possible, a search algorithm based on a projection distance is used. Mapped nodes are sorted according to the topological entities they are associated with, thus allowing *vertex*, *edge* or *face* nodes. This association maintains features within the mesh, such as leading and trailing edges or a wing-body intersection edge. Each mapped node also stores the associated parametric (u,v) coordinates used to locate the node on the corresponding CAD object. The parametric coordinates are required to compute surface displacements on an updated geometry (which is the *primal* of the CAD plugin) and to obtain the derivatives.

FSOpenCascade provides a virtual base class for which the user implements the `build` method that defines the CAD parametrization. Here it is important

to use the alias `Standard_Real` for the declaration of *real* variables such that the parametrization code can be used both with or without the AD capabilities. Moreover, the base class already provides a method named `get_sensitivities` that deals with the AD tool ADOL-C, runs the differentiated CAD parametrization and retrieves the derivative information.

The same functionality is used for the integration of AD-enabled pythonOCC (and the CST parametrization in this case) by creating a Python wrapper using SWIG. On the Python layer, the `build` method is now a placeholder for pythonOCC-based parametrizations.

From the AD perspective, the process of computing the sensitivities is illustrated in Fig. 5 and described as follows. The AD seeding and the differentiated CAD parametrization (defined in the `build` method) are executed on the Python layer. Next, the resulting topological structure that holds the sensitivity information is transferred to the C++ layer. There, the `get_sensitivities` method takes over the time-consuming operation: it iterates through the mapped nodes to re-evaluate their Cartesian coordinates and the corresponding partial derivatives.

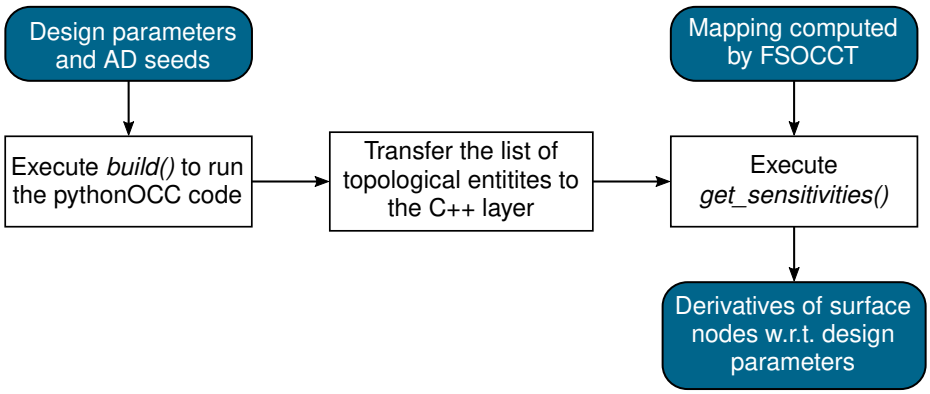

Fig. 5. AD sensitivity calculation with pythonOCC and FSOpenCascade.

It is important to note that the interface of the CAD plugin is compatible with the FSMDAO, such that the inputs (the design parameters) and outputs (the *primal* and the geometric sensitivities) can be directly employed in conjunction with the OpenMDAO framework. This rule also applies for CAD constraints.

5 Validation Study of the Gradient-based Shape Optimization

In order to validate the suggested approach and the software implementation, we considered the drag minimization of a 2-D RAE2822 use-case, cf. [9, 10],

in transonic, turbulent flow conditions at Reynolds number $Re = 6.5$ million and Mach number $Ma = 0.73$. The negative extension of the Spalart–Allmaras turbulence model (*SA-neg*) was used in combination with a 2$^{\text{nd}}$-order finite-volume discretization on unstructured cell-centered grids of 2560 volume cells. A shock is observed at the suction side of the airfoil, as seen in Fig. 7b and Fig. 8b for the initial pressure coefficient distribution. Changes in the geometry, introduced during the optimization, minimize the drag by reducing the shock.

There are several constraints considered in this work. First, an additional functionality is introduced to allow the trimming of the angle of attack for a target lift coefficient of $c_l = 0.71$, defined as an equality constraint that depends on the CFD state. Second, an inequality constraint is defined to keep the pitching moment at the aerodynamic center (located at 25% of the chord length) equal or above of $c_{My} = -0.092$. Third, the area of the airfoil should be larger or equal to the initial value. The latter is implemented as a CAD-based constraint. Additionally, bounds are defined for the (CST) design parameters to avoid invalid geometries or negative mesh cells during the optimization. As mentioned in Sect. 2, there are in total seven design parameters.

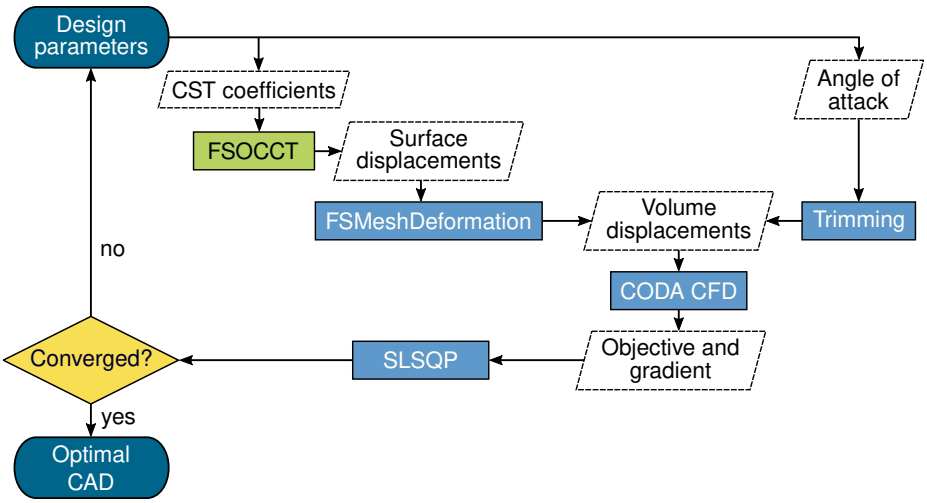

Fig. 6. Shape optimization workflow.

Two configurations of constraints are defined in this study and used separately in the optimization runs: (i) the initial run considers only the target lift constraint and (ii) the second run additionally includes the pitching moment and the CAD-based area constraint. These constraints require the computation of their derivatives, which is done directly in the FSMDAO plugin. Regarding the CAD-based constraint, this requirement is fulfilled using the AD version of pythonOCC.

The optimization is driven by the SLSQP (*Sequential Least SQuares Programming*) algorithm. An overview of the optimization workflow is presented in Fig. 6, while the results are presented in Fig. 7 and Fig. 8.

In the initial optimization, one observes a reduction of the drag coefficient of approx. 25% (Fig. 7c). This outcome, however, also decreases the pitching moment. It is important to note that the initial lift is below the requirement, which then alters the angle of attack to achieve the desired lift. Fig. 7b shows the surface distribution of the pressure coefficient for the initial and final designs. The final design presents a smoother distribution for the suction side of the airfoil. In total there were 50 SLSQP iterations, with 43 of them being gradient calculations.

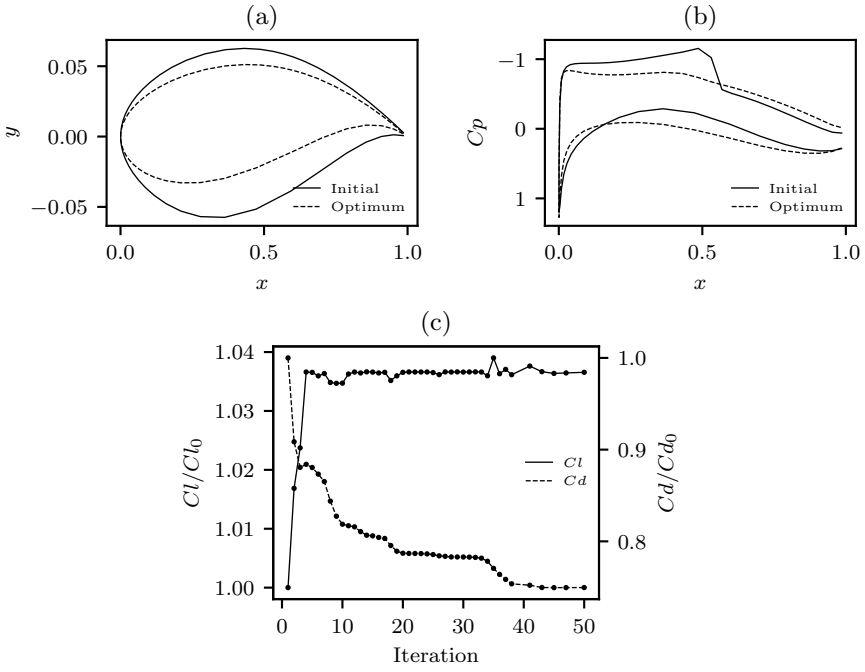

Fig. 7. Optimization results with constrained lift. a) Initial vs. optimal surface mesh. b) Initial vs. optimal pressure coefficient distribution over chord length. c) Optimization history (gradient calculations are marked with dots).

In the second optimization run, the shape variations (shown in Fig. 8a) are smaller compared to the previous optimization. Fig. 8b shows a similar smoothing of the pressure coefficient distribution at the suction side of the airfoil. The minimum pressure coefficient is a bit lower than for the previous case. The optimization results in approx. 9% reduction of the drag. In total there were

123 SLSQP iterations, where 50 were gradient calculations. Here, the additional constraints resulted in more line searches between design iterations than in the previous case.

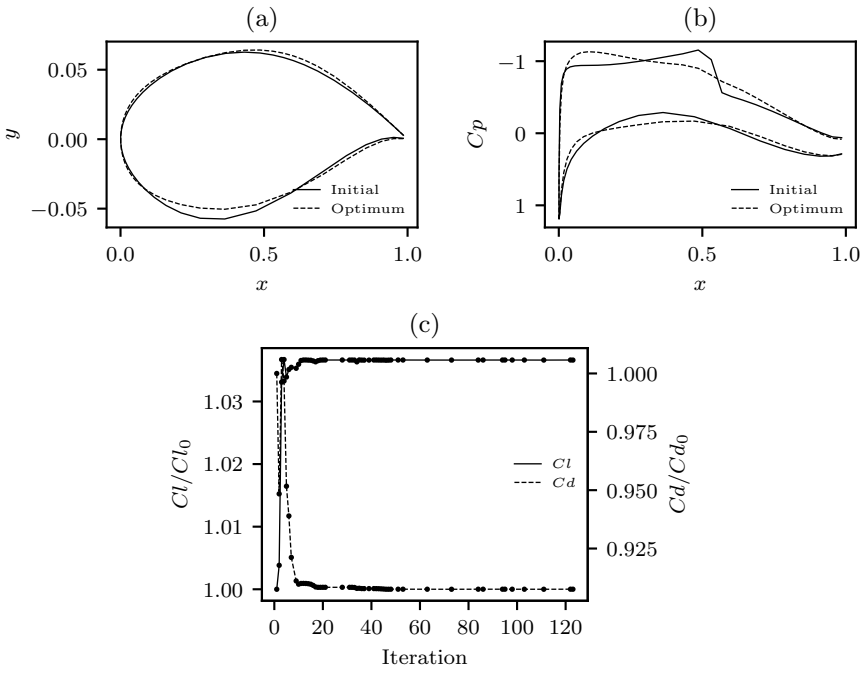

Fig. 8. Optimization results with constrained lift, pitching moment and area. a) Initial vs. optimal surface mesh. b) Initial vs. optimal pressure coefficient distribution over chord length. c) Optimization history (gradient calculations are marked with dots).

6 Conclusion and Outlook

This paper presents the AD-enabled computation of CAD sensitivities in a hybrid Python/C++ geometry modeling environment, where pythonOCC and OCCT are combined with the CST functionality of TiGL. The CST functionality is differentiated with ADOL-C and linked against the differentiated OCCT. Additionally, its Python extension is generated using SWIG by importing the SWIG interfaces of ADOL-C and pythonOCC, thus making it compatible with the differentiated pythonOCC.

The differentiated CAD software stack is employed to parametrize a tapered wing geometry and to compute the corresponding geometric sensitivities. Fur-

thermore, the AD-computed sensitivities are successfully verified against finite differences.

The CAD-integration of the differentiated pythonOCC (and CST) into the optimization framework is achieved by extending the CAD plugin FSOpenCascade. The CAD plugin allows the mesh-to-CAD association and the computation of geometric sensitivities by employing the differentiated OCCT. Here, its Python layer is generated using SWIG, where the `build` method serves as the entry-point for pythonOCC-based parametrizations. When considering AD, the Python layer allows the AD seeding of design parameters and the construction of the desired CAD model. Afterwards, the resulting topological data structure holding the sensitivity information is transferred to the C++ layer where Cartesian points and their partial derivatives are calculated.

Currently, the computation of pythonOCC sensitivities is done using the forward mode of AD, while the differentiated OCCT supports both the forward and the reverse mode of AD. Future work will be related to the reverse differentiation of pythonOCC. For this purpose, a Python extension for the *trace-based* `adouble` class has to be considered.

Finally, we carried out a CAD-based shape optimization using the framework based on the FlowSimulator HPC ecosystem and OpenMDAO. To demonstrate and validate the suggested optimization method and its implementation, the drag of the RAE2822 airfoil in transonic, fully turbulent flow was minimized in combination with a number of constraints. The area of the airfoil, for instance, was considered in a CAD-based way providing the corresponding derivatives w.r.t. design parameters by means of the differentiated pythonOCC. In the future, more complex 3-D aircraft geometries will be considered with reverse-mode multidisciplinary sensitivity analyses using the presented gradient-enabled MDAO framework approach.

References

1. Agarwal, D., Robinson, T.T., Armstrong, C.G., Marques, S., Vasilopoulos, I., Meyer, M.: Parametric design velocity computation for CAD-based design optimization using adjoint methods. Engineering with Computers **34**(2), 225–239 (2018). DOI 10.1007/s00366-017-0534-x. URL https://doi.org/10.1007/s00366-017-0534-x
2. Banović, M., Hafemann, T., Stück, A.: Algorithmic Differentiation of the pythonOCC Geometric Modeling Library. In: 9th European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS 2024 (2024). DOI https://doi.org/10.23967/eccomas.2024.197. URL https://elib.dlr.de/210295/
3. Banović, M., Mykhaskiv, O., Auriemma, S., Walther, A., Legrand, H., Müller, J.D.: Algorithmic differentiation of the Open CASCADE Technology CAD kernel and its coupling with an adjoint CFD solver. Optimization Methods and Software **33**(4-6), 813–828 (2018). DOI https://doi.org/10.1080/10556788.2018.1431235
4. Beazley, D.M.: Automated scientific software scripting with SWIG. Future Generation Computer Systems **19**(5), 599–609 (2003). DOI https://doi.org/10.1016/S0167-739X(02)00171-1

5. Ehrmanntraut, S., Büchner, A., Gottfried, S., Stück, A.: Scalable Framework Integration of CODA for a Multidisciplinary Preconditioned Matrix-Free Newton-Krylov Method. In: STAB/DGLR Symposium 2022 (2024). DOI https://doi.org/10.1007/978-3-031-40482-5
6. Gallard, F., Vanaret, C., Guénot, D., Gachelin, V., Lafage, R., Pauwels, B., Barjhoux, P.J., Gazaix, A.: GEMS: A Python Library for Automation of Multidisciplinary Design Optimization Process Generation. In: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference (2018)
7. Giles, M.B., Duta, M.C., Müller, J.D., Pierce, N.A.: Algorithm developments for discrete adjoint methods. AIAA journal **41**(2), 198–205 (2003)
8. Gray, J.S., Hwang, J.T., Martins, J.R.R.A., Moore, K.T., Naylor, B.A.: OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. Structural and Multidisciplinary Optimization **59**(4), 1075–1104 (2019). DOI 10.1007/s00158-019-02211-z
9. Hafemann, T., Banović, M., Büchner, A., Ehrmanntraut, S., Höing, C., Gottfried, S., Stück, A.: A CAD-enabled MDAO Framework Approach for Gradient-based Aerodynamic Shape Optimization. In: 9th European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS 2024 (2024). DOI https://doi.org/10.23967/eccomas.2024.199
10. He, X., Li, J., Mader, C.A., Yildirim, A., Martins, J.R.: Robust aerodynamic shape optimization—from a circle to an airfoil. Aerospace Science and Technology **87**, 48–61 (2019). DOI https://doi.org/10.1016/j.ast.2019.01.051. URL https://www.sciencedirect.com/science/article/pii/S1270963818319072
11. Jameson, A.: Aerodynamic design via control theory. In: Recent advances in computational fluid dynamics, pp. 377–401. Springer (1989)
12. Kulfan, B., Bussoletti, J.: "Fundamental" Parameteric Geometry Representations for Aircraft Component Shapes. In: 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference (2006). DOI https://doi.org/10.2514/6.2006-6948
13. Leicht, T., Vollmer, D., Jägersküpper, J., Schwöppe, A., Hartmann, R., Fiedler, J., Schlauch, T.: DLR-Project DIGITAL-X – Next Generation CFD Solver FLUCS. DLRK (2016). URL https://elib.dlr.de/111205/
14. Pascual, V., Hascoët, L.: Mixed-language automatic differentiation. Optimization Methods and Software **33**(4-6), 1192–1206 (2018). DOI 10.1080/10556788.2018.1435650. URL https://doi.org/10.1080/10556788.2018.1435650
15. Paviot, T.: pythonocc (2022). DOI https://doi.org/10.5281/zenodo.7471333
16. Pironneau, O.: On optimum design in fluid mechanics. Journal of Fluid Mechanics **64**(1), 97–110 (1974)
17. Rempke, A.: Deformation of CFD Meshes with Anisotropic Cells in a Viscous Boundary Layer Using Line-Implicit Methods. In: 23rd STAB/DGLR Symposium on New Results in Numerical and Experimental Fluid Mechanics XIV, *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, vol. 154. Springer Nature Switzerland AG (2023). DOI https://doi.org/10.1007/978-3-031-40482-5_26. URL https://elib.dlr.de/198222/
18. Siggel, M., Kleinert, J., Stollenwerk, T., Maierl, R.: TiGL: An Open Source Computational Geometry Library for Parametric Aircraft Design. Mathematics in Computer Science **13**(3), 367–389 (2019). DOI https://doi.org/10.1007/s11786-019-00401-y
19. Walther, A., Griewank, A.: Getting started with ADOL-C. In: Combinatorial scientific computing, Chapman & Hall/CRC Computational Science, pp. 181–202. Taylor & Francis (2012). DOI https://doi.org/10.1201/b11644