PADERBORN UNIVERSITY
The University for the Information Society

Efficient algorithmic differentiation of CAD frameworks

Mladen Banović

Dissertation

Advisor: Prof. Dr. Andrea Walther

Paderborn, December 19, 2019

Acknowledgments

The past four years at Paderborn University (UPB) have been an unforgettable and wonderful journey of my life. I owe my deepest gratitude to my advisor Prof. Dr. Andrea Walther for giving me an opportunity to work under her guidance in one of the most inspiring and challenging ventures I have encountered so far.

This research is part of the IODA project — *Industrial Optimal Design using Adjoint CFD*. IODA is a *Marie Skłodowska-Curie Innovative Training Network* (ITN) funded by the European Union's Horizon 2020 research and innovation program under the Grant Agreement No. 642959. I find IODA ITN to be a brilliant concept that provided me a working experience both in academic and industrial environment. One of the key aspects of an ITN is mobility. That is, during my fellowship I have taken a number of secondments to IODA associates: Open CASCADE (OCC), Queen Mary University of London (QMUL), Rolls-Royce Deutschland (RRD) and the von Karman Institute for Fluid Dynamics (VKI). Being internationally mobile while doing research is relevant both for professional and personal development. I am grateful to many people with whom I collaborated and from whom I have learned a lot. Without these collaborations, certain results achieved in this work would not be possible. Beside providing the partnership opportunities, the ITN offered various trainings and workshops to elevate research and transferable skills. Here, I want to thank the project coordinator Dr. Jens-Dominik Müller (QMUL) and the project administrator Susan Barker (QMUL) for their organization and enthusiasm, as well as the European Commission for funding the whole program.

During my research stay at OCC (France) in 2015 I received a great support from Dr. Herve Legrand and Sergey Slyadnev, which eventually helped me to overcome the core assignment of my IODA fellowship — algorithmic differentiation of the Open CASCADE Technology CAD kernel. I would like to extend my deepest gratitude to IODA fellows Orest Mykhaskiv (QMUL) and Salvatore Auriemma (OCC), as well as their advisor Dr. Jens-Dominik Müller, who worked with me on a regular basis from that point until the end of the project.

Regarding the collaboration with VKI, I want to thank Dr. Ismael Sanchez Torreguitart and Prof. Dr. Tom Verstraete for their encouragement and knowledge sharing.

I would like to offer my special thanks to Dr.-Ing. Marcus Meyer (RRD) for inviting me to Rolls-Royce premises where I had the opportunity to work on industrial problems. His support is highly appreciated as well as the joint work with Ilias Vasilopoulos (RRD). Moreover, advices from Dr.-Ing. Peter Flassig (RRD) and Dr.-Ing. André Huppertz (RRD) have been a valuable help in understanding design principles of turbo-machinery blades.

Thanks should also go to present and former colleagues of the group 'Mathematics and its Applications' at UPB for their insightful comments. I would like to recognize the assistance I received from Dr. Kshitij Kulshreshtha about the field of algorithmic differentiation and the software tool ADOL-C.

Last, but not least, my heartfelt appreciation goes to my whole family in Croatia, especially to my sister Ružica Banović who has moved two years ago to Germany to live with me. I am thankful to them for believing in me and giving me constant support in pursuing this life-changing adventure.

Abstract

Computer Aided Design (CAD) systems and tools are considered essential for industrial design. They construct and manipulate the geometry of a certain component with an arbitrary set of design parameters. However, contrary to e.g. Computational Fluid Dynamics (CFD) solvers, shape sensitivities for gradient-based optimization of CAD-parametrized geometries have been so far only available with inaccurate finite differences. Here, Algorithmic Differentiation (AD) is applied to three CAD libraries to obtain the exact derivative information. This represents the first time that AD has been integrated into a CAD framework.

First, the open-source CAD kernel *Open CASCADE Technology* (OCCT) is differentiated using the AD software tool ADOL-C (*Automatic Differentiation by OverLoading in C++*). Furthermore, the differentiated OCCT is coupled with the discrete adjoint CFD solver STAMPS (*Source-Transformation Adjoint Multi-Physics Solver*), developed at Queen Mary University of London and also produced by AD. This achievement represents the first example of a complete differentiated design chain built from generic, multi-purpose tools. The design chain is demonstrated on the gradient-based shape optimization of two turbo-machinery test-cases: a U-bend cooling channel and the TU Berlin (TUB) *TurboLab* stator. In both cases, the aim is to minimize the total pressure loss. Moreover, the optimization framework is extended to support assembly constraints of the TUB stator test-case. In particular, there are four mounting bolts (cylinders) that are pierced inside the volume of the blade and they serve to mount the blade to the stator assembly. Their optimal position during the aerodynamic shape optimization — that ensures there is no collision between the optimal blade and the cylinders — is accomplished using the derivative information from the differentiated OCCT.

Second, the reverse mode of ADOL-C is integrated into the in-house CAD and mesh generation tool CADO (*Computer Aided Design and Optimization*) developed at the von Karman Institute for Fluid Dynamics. The differentiated CADO tool is used for the derivative computation of the LS89 axial turbine test-case. However, due to large memory consumption of the differentiated sources, the source code is modified by exploiting the structure of the mesh generation algorithm to benefit from an improved efficiency.

Finally, the Rolls-Royce in-house airfoil design and blade generation tool *Parablading* is differentiated using the AD software tools ADOL-C and Tapenade. The differentiated Parablading tool is coupled with a discrete adjoint CFD solver that is part of the Rolls-Royce in-house HYDRA suite of codes, also produced by AD. This differentiated design chain is utilized to perform gradient-based shape optimization of the TUB *TurboLab* stator test-case with the aim to minimize the total pressure loss and exit angle deviation objectives. The resulting shape is entirely different from conventional results achieved so far.

Keywords: Algorithmic Differentiation, CAD kernel, Adjoint CFD method, Gradient-based optimization.

Zusammenfassung

Computer Aided Design (CAD) Systeme und Werkzeuge sind essenzielle Bestandteile in Prozessen des industriellen Designs. Diese konstruieren und modifizieren die Geometrie von Komponenten mit verschiedensten Designparametern. Im Gegensatz zum großen Bereich der numerischen Strömungsmechanik (*Computational Fluid Dynamics* — CFD) wurden die geometrische Sensitivitäten — für Gradienten-basierte Optimierung von CAD-parametrisierter Geometrie — bislang nur mit ungenauen Finiten Differenzen berechnet. In der vorliegenden Arbeit wird das Algorithmische Differenzieren (AD) auf drei CAD Ansätze angewandt, um die exakten Ableitungen zu berechnen. Dies ist das erste Mal, dass AD in ein CAD Framework integriert wird.

Im ersten Teil der vorliegenden Arbeit wird der Open-Source CAD Kernel Open CASCADE Technology (OCCT) durch die Anwendung des AD Werkzeugs ADOL-C (*Automatic Differentiation by OverLoading in C++*) differenziert. Außerdem wird das differenzierte OCCT mit dem diskreten adjungierten CFD Werkzeug STAMPS (*Source-Transformation Adjoint Multi-Physics Solver*) gekoppelt, welches an der Queen Mary University of London entwickelt wurde und AD verwendet, um Adjungierte zu berechnen. Dieses Ergebnis stellt das erste Beispiel einer vollständig differenzierten Design Kette dar, welche auf generischen Werkzeugen basiert. Diese Design Kette wird zur Gradienten-basierten Formoptimierung zweier industrierelevanter Beispiele genutzt: ein U-Rohr Kühlkanal und der TU Berlin (TUB) *TurboLab* Stator. In beiden Fällen ist es das Ziel, den allgemeinen Druckverlust zu minimieren. Anschließend wird der Optimierungsprozess weiterentwickelt, um geometrische Nebenbedingungen des TUB Stator Beispiels berücksichtigen zu können. Es gibt vier Befestigungsschrauben (Zylinder), die in der Schaufel angebracht sind. Diese verbinden die Schaufel mit dem kompletten Stator. Die optimale Position während der aerodynamischen Formoptimierung wird durch die Nutzung der Ableitung des differenzierten OCCTs erreicht. Dadurch kann garantiert werden, dass es zu keinem Durchdringen der optimierten Schaufel und den Zylindern kommt.

Im zweiten Teil der Arbeit wird der Rückwärtsmodus von ADOL-C in das proprietäre CAD und Gittergenerierung Werkzeug CADO (*Computer Aided Design and Optimization*), welches am *von Karman Institute for Fluid Dynamics* entwickelt wurde, integriert. Das differenzierte CADO Werkzeug wird für die Berechnung der Ableitung des *LS89 axial turbine* Beispiels genutzt. Aufgrund des großen Speicherplatzbedarfs des differenzierten Programms wird der Code so modifiziert, dass die Struktur des Gittergenerierung-Algorithmus ausgenutzt wird, um die Effizienz zu erhöhen.

Abschließend wird das proprietäre Rolls-Royce Schaufel Design Werkzeug Parablading betrachtet. Hier finden die AD Werkzeuge ADOL-C und Tapenade Einsatz. Das differenzierte Parablading Werkzeug wird mit einem diskreten adjungierten CFD Werkzeug, welches Teil der proprietäre Rolls-Royce HYDRA Bibliothek ist und mit Hilfe von AD entwickelt wurde, gekoppelt. Die differenzierte Design Kette wird genutzt, um eine Gradienten-basierte Optimierung des TUB *TurboLab* Stator Beispiels durchzuführen. Das Ziel ist es, den Druckverlust und die Abweichung des Winkels zu minimieren. Die erzielten Ergebnisse unterscheiden sich fundamental von der bislang möglichen Formoptimierung.

Stichworte: Algorithmisches Differenzieren, CAD Kernel, Adjungierter CFD Ansatz, Gradienten-basierte Optimierung

Contents

List of Figures

List of Tables

Nomenclature

List of abbreviations

2-D	two-dimensional
3-D	three-dimensional
AD	Algorithmic/Automatic Differentiation
ADOL-C	Automatic Differentiation by OverLoading in C++
BFGS	Broyden-Fletcher-Goldfarb-Shanno
BRep	Boundary Representation
CAD	Computer Aided Design
CADO	Computer Aided Design and Optimization
CAE	Computer Aided Engineering
CAM	Computer Aided Manufacturing
CFD	Computational Fluid Dynamics
CSM	Computational Structural Mechanics
ESR	Early Stage Researcher
FD	Finite Differences
GUI	Graphical User Interface
IGES	Initial Graphics Exchange Specification
IODA	Industrial Optimal Design using Adjoint CFD
ITN	Innovative Training Network
L-BFGS-B	Limited-memory BFGS with Box constraints

LE	Leading Edge
NSPCC	NURBS-based Parametrization with Complex Constraints
NURBS	Non-Uniform Rational B-Spline
OCC	Open CASCADE
OCCT	Open CASCADE Technology
PDE	Partial Differential Equation
PS	Pressure Side
QMUL	Queen Mary University of London
RAM	Random-Access Memory
RANS	Reynolds-Averaged Navier-Stokes
RRD	Rolls-Royce Deutschland
S-D	Steepest-Descent
SLSQP	Sequential Least SQuares Programming
SS	Suction Side
STAMPS	Source-Transformation Adjoint Multi-Physics Solver
STEP	STandard for the Exchange of Product model data
STL	STereoLithography
Tcl	Tool Command Language
TE	Trailing Edge
TUB	Technical University of Berlin
UPB	Paderborn University
VKI	von Karman Institute for Fluid Dynamics

1 Introduction

1.1 Motivation

Computer Aided Design (CAD) systems and tools are considered essential for industrial design. They create and manipulate the geometric representation of a certain component with an arbitrary set of design parameters. This geometric representation serves to generate a computational domain (mesh) for Computer Aided Engineering (CAE). CAE packages such as Computational Fluid Dynamics (CFD) or Computational Structural Mechanics (CSM) use these meshes when solving the governing partial differential equations (PDEs) to evaluate the design performance. After a number of design iterations, the selected CAD model can be manufactured using Computer Aided Manufacturing (CAM). This complete process represents the so-called CAD/CAE/CAM workflow. The first two elements of this workflow (CAD and CAE) form the so-called design chain — illustrated in Fig. 1.1.

To obtain an optimal design, one typically employs shape optimization techniques that are nowadays profoundly used in aeronautical and automotive industry. With advances in computational power and numerical simulation methods (e.g. CFD, CSM), the product development cycle is driven by shape optimization methods, especially in its early development phase to accelerate design studies and reduce experimental cost.

Optimization algorithms systematically explore the design space to find the optimal solution. They are divided into two categories: gradient-free [CSV09] and gradient-based optimization methods.

The gradient-free methods, such as Particle Swarm Optimization [Ken10] or Evolutionary Algorithms [Bac96], require only the *primal* value of the objective function to

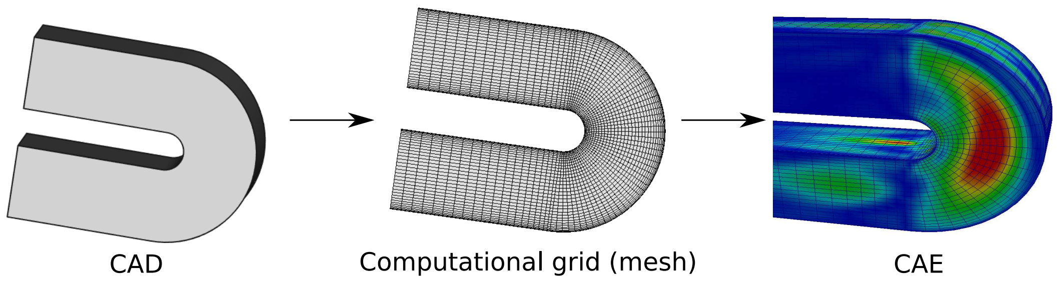


Figure 1.1: Design chain

be optimized. They treat each element of the chain in a *black-box* fashion and hence it is straightforward to build the CAD/CAE optimization workflow. Nevertheless, the gradient-free approaches require a substantial amount of function evaluations. When these evaluations involve CAE tools such as CFD — that often have rich design spaces and large computational cost — the gradient-free methods become prohibitively expensive for industrial applications.

To reduce the cost of the gradient-free approaches in aerodynamic shape optimization, one employs surrogate methodologies to build meta-models that mimic the behavior of expensive CFD simulations [IQ13]. Here, the challenge is to develop a meta-model that is as accurate as possible, by running only a few high-fidelity simulations during the whole optimization process. Therefore, the chosen data-set to train the meta-model highly influences the search direction and quality of the optimum. Nevertheless, this approach is useful for searching global behavior of the system being optimized, especially for the preliminary design studies.

On the other hand, the gradient-based methods are recognized for their computational efficiency when optimizing cases with many design variables and therefore are considered in this work. Although gradient-based non-linear solvers converge to a locally optimal point, they are acceptable for industrial applications where the initial design is based on engineering knowledge and experience, thus characterized by satisfying aerodynamic properties. Therefore, even a couple of incremental design updates during the optimization would already provide an improved design in terms of its performance. However, to involve these methods into the shape optimization, gradients need to be computed for each element in the design chain.

Over the past few decades, the adjoint approach [Pir74, Jam89, GDMP03] in CFD codes is considered as *state-of-the-art* for evaluating gradients because its computational cost is independent on the number of design variables and near-constant to the *primal* evaluation.

Today, industrial application of gradient-based optimization is primarily limited by the immaturity of the parametrization tools that define the design space. That is, computing CAD gradients still remains challenging.

So far, the derivative computation for a complete design chain, ranging from the parametrization to desired objective value, has not been demonstrated with exact gradient computation. Robinson et al. [RAC$^+$12] used rather inaccurate finite differences (FD) to evaluate gradients in commercial CAD systems. An alternative is to use non-CAD parametrization [GWMW07] that can be entirely differentiated, however offering limited versatility. Most importantly, the optimal shape exists as a deformed mesh rather than in a CAD format, which implies a severe shortcoming for the industrial CAD/CAE/CAM workflow. This gap is closed by the results presented in this work which, among others, is the algorithmic differentiation (AD) of involved CAD libraries.

1.2 Parametrization approaches: CAD-free vs. CAD-based

CAD-free methods, also referred to as node- or mesh-based approaches, are typically used within gradient-based optimization workflow. Here a displacement of every grid point coordinate is considered as a design variable, thus providing the richest design space that optimizers can explore. However, to avoid oscillatory shapes, one requires regularization of the gradients, often chosen as implicit [JV00] or explicit [JM08] smoothing. Other popular approaches to manipulate grid points are: (i) auxiliary grid perturbations such as Free-Form deformation (FFD) which interpolates the deformation of the domain within the control lattice of a volume spline [SP86, JA07], (ii) a globally interpolated distortion field from radial basis functions (RBF) [DBVdSB06] or (iii) lattices of Hicks-Henne bumps [HH78].

A major drawback of the CAD-free methods is that the optimal shape exists only as a mesh, rather than in a CAD-specific format. Importing it back to a CAD system for further investigation is a difficult task. Usually the importing step involves significant approximations that impair the optimality of the shape. Another difficulty is to include geometric constraints that are crucial for the optimization of industrial components.

On the contrary, CAD-based methods resolve these issues by keeping a CAD framework in the design loop. Therefore, the optimal shape exists in the CAD-specific format and it is available for further multi-disciplinary analysis and finally manufacturing.

1.3 Current approaches to evaluate derivatives in CAD environment

To integrate the parametric CAD description into a gradient-based optimization framework, one requires the calculation of the so-called *shape sensitivities* with respect to design parameters of the model to be optimized. However, computing CAD gradients is a difficult task and usually this information is not provided in commercial CAD systems, e.g. SIEMENS NX, SolidWorks or CATIA V5.

Agarwal et al. [ARA$^+$18b] compute the shape sensitivity (also referred to as *design velocity*) in the commercial CAD systems through FD. Here, a step size (perturbation) has to be carefully chosen for each design parameter used in a CAD feature tree to limit truncation errors. Moreover, to avoid any issues with possible topology changes (patch renumbering), they approximate the original and perturbed geometries using surface tessellation of linear triangular elements (also referred to as the faceted representation). This introduces an additional step of surface-to-surface projections to compute the distances between the unperturbed and perturbed surface facets. Nevertheless, this method fits in a wide range of industrial applications where closed-source CAD software is commonly used.

Xu et al. [XJM13, XRMM15] use NURBS (*Non-Uniform Rational B-Spline*) patches in the CAD-native boundary representation (BRep). They developed a geometric kernel, namely the 'NURBS-based Parametrization with Complex Constraints' (NSPCC), which considers only the BRep as given, e.g. in the standardized STEP (*STandard for the Exchange of Product model data*) format, therefore ignoring any internal representation of a CAD system. For this reason, the method is considered as vendor-neutral. For computing the derivatives, the NSPCC kernel is differentiated using the source-transformation AD tool Tapenade.

Sanchez Torreguitart et al. [STVM18] differentiated the in-house CAD kernel CADO (*Computer Aided Design and Optimization*) developed at the von Karman Institute for Fluid Dynamics (VKI), using the AD tool ADOL-C (*Automatic Differentiation by OverLoading in C++*). They performed aerodynamic CAD-based shape optimization of the LS89 axial turbine nozzle guide vane profile [ALR90], where the gradients are computed with respect to engineering design parameters such as axial chord length, trailing edge radius, etc.

Dannenhoffer and Haimes used the open-source CAD kernel Open CASCADE Technology (OCCT) to develop a fully-parametric, feature-based solid-modeling system with web-based user interface [HD13]. To obtain the derivatives they applied analytic differentiation to simple geometrical shapes (such as circles or cylinders), while finite differences are used for the more complex geometries [DH15]. They also considered AD of the OCCT code, but did not demonstrate an implementation due to the high complexity of the source code.

As demonstrated in this study, the AD of OCCT is indeed feasible. Here, the most-widely used CAD kernel OCCT is differentiated in forward and reverse mode of the AD tool ADOL-C. This work is the first instance a fully developed CAD kernel has been differentiated.

Comparing to other CAD-based approaches, the application of AD to a complete CAD kernel offers a number of significant advantages. As opposed to the FD approaches [RAC$^+$12, ARA$^+$18b], the AD method does not require the computation of perturbed geometries, therefore avoiding the risk of topology changes. Furthermore, the geometric derivatives are not affected by truncation or projection errors, but are exact up to machine accuracy. Finally, the computational efficiency of the AD method can be demonstrated to be superior compared to the FD approach, especially when using a large number of design variables.

1.4 Thesis contribution

This work improves CAD-based optimization workflows using gradient-based methods by applying the AD technique to the existing CAD libraries, in order to compute the *shape sensitivities* that are typically not available with the exact gradient computation. The AD-enabled CAD packages are successfully coupled with discrete adjoint CFD solvers (that are also produced by AD), thus providing complete differentiated design chains at hand. These differentiated design chains are utilized to perform aerodynamic gradient-based optimization of industrial components.

There are several novel contributions of this dissertation:

- AD of the general-purpose open-source CAD kernel OCCT.
- Validation of the differentiated OCCT in the aerodynamic gradient-based optimization of parametric CAD models.[1]
- Integration of geometric constraints in the gradient-based optimization using the differentiated version of the OCCT kernel.[1]
- Improved AD of the VKI in-house CAD and mesh generation tool CADO.[1]
- AD of an industrial CAD tool, namely the Rolls-Royce in-house airfoil design and blade generation tool Parablading.
- Validation of the differentiated Parablading in the aerodynamic gradient-based optimization of an industrial component.[1]

1.4.1 Other contributions and collaborations

Applications of the differentiated CAD libraries into the gradient-based optimization workflows have been conducted among partners within the project IODA[2] (*Industrial Optimal Design using Adjoint CFD*) funded by the European Commission. During the project, the author has taken a number of secondments to IODA associates: Open CASCADE (OCC), Queen Mary University of London (QMUL), Rolls-Royce Deutschland (RRD) and von Karman Institute for Fluid Dynamics (VKI). These research activities and collaborations made a significant contribution to this disser-

[1]Read Sec. 1.4.1

tation — certain results would not be even possible without them.

First, three early-stage researchers (ESRs) performed a short research activity (the second half of the year 2015) together at OCC premises to start with the differentiation of the OCCT kernel: Mladen Banović (ESR12, Paderborn University) — the author of this dissertation, Orest Mykhaskiv (ESR2, QMUL) and Salvatore Auriemma (ESR9, OCC). With support of Sergey Slyadnev (OCC), several differentiation approaches resulting in different code modifications were discussed. After an agreement, the chosen differentiation approach was performed by the author at Paderborn University (UPB) to integrate the AD software tool ADOL-C into the OCCT sources. The geometric derivatives were successfully verified against finite differences, using the following parametric models: (i) a U-bend cooling duct and (ii) the TU Berlin (TUB) *TurboLab* stator. These parametric models were designed and coded in OCCT by ESR9. Moreover, ESR2 created an optimization framework to couple the differentiated OCCT with a discrete adjoint CFD solver developed at QMUL. The author also contributed to the optimization framework with the derivative computation of the parametric models. This framework has been used for a gradient-based optimization of the parametric U-bend and TUB stator models and the results are elaborated in common publications [ABM$^+$16, BMA$^+$18, MBA$^+$18, ABW$^+$18]. Furthermore, in a collaboration with ESR9, the author expanded the gradient-based optimization framework to handle assembly (geometric) constraints of the TUB stator test-case, using the derivative information from the differentiated OCCT.

Second, the VKI in-house CAD and grid generation tool CADO was differentiated using the reverse mode of ADOL-C in a collaboration with Ismael Sanchez Torreguitart (ESR13, VKI) and used for the derivative computation of the LS89 axial turbine test-case. However, the reverse differentiated CADO sources consumed a large amount of memory (more than 30 GB of memory). For this reason, further work was carried out by the author to perform structure exploitation, i.e. to modify the reverse differentiated CADO sources to increase their efficiency.

Finally, the AD of the industrial airfoil design tool Parablading was performed solely by the author at RRD. However, its integration into the Rolls-Royce optimization

[2]https://ioda.sems.qmul.ac.uk/

workflow was carried out in a collaboration with Ilias Vasilopoulos (ESR11, RRD) to optimize the TUB stator component. The results are published as a scientific article in the 'Optimization and Engineering' journal [BVWM19].

1.5 Thesis structure

The thesis is structured as follows.

Chapter 2 explains the parametrization principles used to construct three CAD models optimized in this work. First, the parametric models of the U-bend cooling duct and the TUB stator blade are defined using the OCCT kernel. Second, the TUB stator blade is also parametrized using the Rolls-Royce in-house airfoil design and blade generation tool Parablading. Although the author did not parametrize these models, it is important to explain how the mentioned components are constructed because the underlying CAD sources were differentiated to compute the geometric sensitivities.

Chapter 3 describes the algorithmic differentiation of the OCCT CAD kernel. It offers a discussion about several investigated approaches to differentiate OCCT among with compile- and run-time issues faced upon the differentiation process. After the successful differentiation, the primal functionality is validated using an automated test system. Moreover, the gradients evaluated with AD are verified against FD, by using the parametric models of the U-bend duct and the TUB stator blade. Finally, Chapter 3 presents performance tests of the differentiated OCCT.

Chapter 4 presents the governing equations for a flow problem and its adjoint, together with an assembly of the relevant derivatives. That is, it explains how the differentiated OCCT is coupled with the STAMPS (*Source-Transformation Adjoint Multi-Physics Solver*) flow solver to form the gradient-based shape optimization framework. Finally, it shows an application of the complete differentiated design chain to the gradient-based optimization of the U-bend duct and the TUB stator blade.

Chapter 5 expands the optimization workflow explained in Chapter 4 by handling assembly constraints defined for the TUB stator test-case. In particular, the blade has to accommodate four mounting bolts (cylinders) that serve to attach the blade to its casing. This is a rather challenging geometric requirement to fulfill because the position of the cylinders is arbitrary in the blade whose shape is changing during the optimization process. The proposed method to tackle this constraint, i.e. that ensures there is no interference between the optimal blade and the cylinders, is elaborated in the chapter, together with the optimization results.

Chapter 6 explains improved AD of the VKI in-house CAD and grid generation tool CADO. The CADO sources are differentiated using the reverse mode of ADOL-C to compute the derivative information of the LS89 axial turbine test-case. Nevertheless, the differentiated CADO tool required a large amount of memory (more than 30 GB). Therefore, the structure exploitation of the differentiated sources is performed to benefit from an improved efficiency.

Chapter 7 describes the algorithmic differentiation of the Rolls-Royce in-house airfoil design and blade generation tool Parablading. Moreover, it presents the derivative verification and the performance tests of the differentiated sources. After the successful differentiation, the Parablading tool is coupled with a discrete adjoint CFD that is part of the Rolls-Royce in-house HYDRA suite of codes. Furthermore, the gradient-based shape optimization is performed on the TUB stator test-case and the optimization results are presented.

Finally, Chapter 8 offers conclusions and a discussion about possible future directions in this field of research.

2 Test-cases

As explained in Sec. 1.4, the author did not contribute to definitions of parametric CAD models used in the gradient-based optimization framework. However, it is important to give a brief description of parametrization principles, as the underlying source code was differentiated in order to compute the CAD sensitivities. The author used the parametric models to verify correctness of the computed derivatives and to measure performance of the differentiated sources.

2.1 Parametrization with OCCT

Two industrial components were parametrized in the original OCCT by Salvatore Auriemma (OCC): a U-bend cooling duct and the TUB stator as well established benchmark [MV]. The detailed parametrization principles can be found in [BMA$^+$18, ABW$^+$18].

2.1.1 U-bend

The U-bend duct is a typical cooling channel used in a turbine blade application. The baseline geometry, shown in Fig. 2.1, consists of a circular U-part with attached inlet and outlet legs. The attached legs are not modified during the optimization and therefore the parametrization is only defined on the U-part.

The three-dimensional (3-D) U-part shape is based on a cross-sectional design approach — also known as *lofting* operation in CAD. This approach constructs the final B-Spline surfaces by taking as inputs n two-dimensional (2-D) slices generated along a guiding *path-line*. Each slice lies on a plane that is orthogonal to the

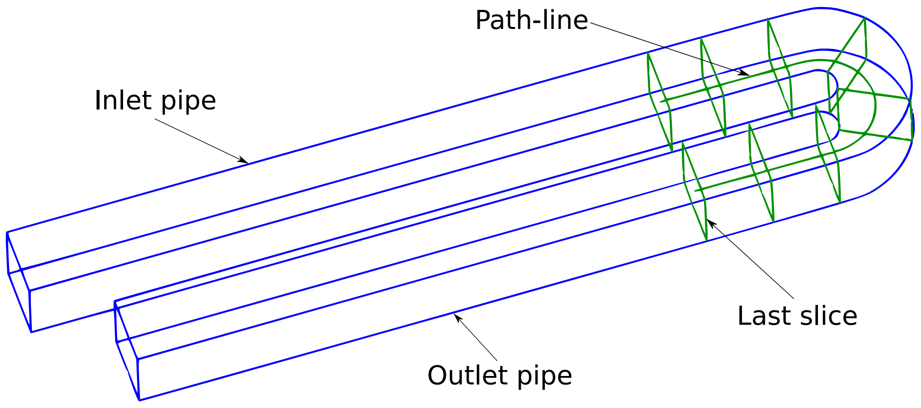


Figure 2.1: U-bend parametrization in OCCT[3]

path-line. The path-line is described as a B-spline curve and it is not manipulated during the optimization. The slice consists of four Bézier curves forming a closed wire and having in total twelve control points. Each control point has its own law of evolution along the path-line that determines its position in the specific plane. The laws of evolution are defined as B-spline curves and their control points are the actual design parameters considered in the optimization, in this example 96 degrees of freedom.

2.1.2 TU Berlin stator

The TUB *TurboLab* stator is a compressor stator designed at the Chair for Aero Engines of the TU Berlin. It is used in modern jet engines compressors with the purpose to turn the incoming flow of 42° whirl angle into an axial direction at the outlet. Complete description of the test-case as well as the corresponding CAD geometry (given in STEP, IGES and parasolid formats) can be found in [MV].

2-D profile parametrization

The parametrization starts by defining a camber-line as a B-spline curve consisting of seven control points. Orthogonally to the camber-line, eight control points are

[3]Picture provided by Salvatore Auriemma

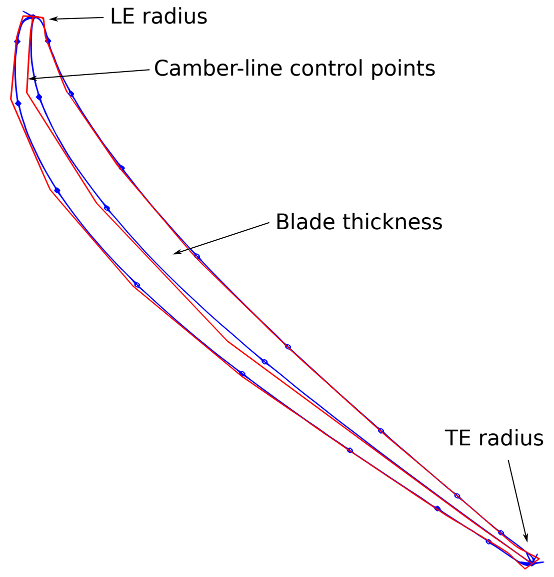


Figure 2.2: TUB 2-D section parametrization in OCCT[4]

generated both for suction and pressure side B-spline curves. Finally, the suction and pressure B-splines are smoothly joined at the leading and trailing edge (LE and TE) of the camber-line, using a specified radius of curvature.

The 2-D profile, illustrated in Fig. 2.2, consists of 23 parameters: (i) 13 parameters to control the camber-line and (ii) ten parameters to control thickness, where two of them define LE and TE radii.

3-D parametrization

Identical to the U-bend test-case (described in Sec. 2.1.1), the 3-D blade parametrization is also based on the cross-sectional design approach. As illustrated in Fig. 2.3, a number of 2-D slices is distributed along a blade span that is defined as a B-spline curve, also referred to as the *path-line*. Each 2-D section parameter is characterized by a law of evolution along the path-line. The laws of evolution are defined as B-spline curves, consisting of eight control points each. Certain control point parameters are the actual design parameters of the optimization. In total there are

[4]Picture provided by Salvatore Auriemma

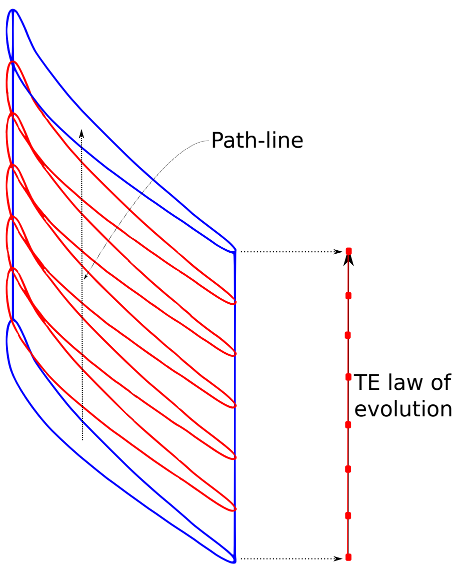


Figure 2.3: TUB blade parametrization in OCCT[5]

184 design parameters (23 section parameters $\times$ 8 law of evolution parameters).

2.2 TU Berlin stator parametrization in Rolls-Royce workflow

Since the TUB stator test-case has been also optimized in the Rolls-Royce workflow, its geometry has been re-parametrized with the in-house CAD tool Parablading. Here, the author gives a brief description of the parametrization, more details can be found in [BF10].

2-D profile parametrization

The stator's blade geometry is characterized by a set of individual 2-D sections. The 2-D section parametrization is shown in Fig. 2.4. It consists of the following individual design parameters:

[5]Picture provided by Salvatore Auriemma

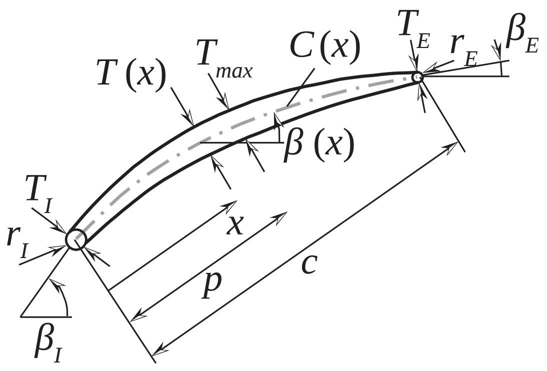


Figure 2.4: TUB 2-D section parametrization in Parablading[6]

- leading/trailing edge radii (r_I/r_E),
- inlet/exit blade angles (β_I/β_E),
- maximum thickness (T_{max}).

The chord length c is set to be constant. Extra parameters serve to control camber-line and thickness distribution. However, instead of controlling the camber-line $C(x)$ directly, Bestle et al. [BF10] and Vasilopoulos et al. [VFM17] prefer to use the camber-line angle distribution $\beta(x)$. Moreover, they normalize both camber-line angle $\beta(x)$ and thickness $T(x)$ distributions using the blade inlet and exit angles (β_I/β_E), maximum thickness (T_{max}) and the chord length c:

$$\tilde{\beta}(\tilde{x}) = \frac{\beta_I - \beta(\tilde{x})}{\beta_I - \beta_E} \in [0,1], \quad \tilde{x} = \frac{x}{c} \in [0,1] \tag{2.1}$$

$$\tilde{T}(\tilde{x}) = \begin{cases} \dfrac{T(\tilde{x}) - T_I}{T_{max} - T_I} & \text{for } 0 \leq \tilde{x} \leq \tilde{p}, \\ \dfrac{T(\tilde{x}) - T_E}{T_{max} - T_E} & \text{for } \tilde{p} < \tilde{x} \leq 1 \end{cases} \in [0,1], \quad \tilde{p} = \frac{p}{c} \in [0,1] \tag{2.2}$$

where $\tilde{p}$ is the normalized position of the maximum thickness ($\tilde{T}_{max}$) for the non-dimensional thickness distribution $\tilde{T}(\tilde{x})$. The normalized distributions are characterized by two-dimensional B-spline curves whose control points are indirectly modified during the optimization.

[6]Picture provided by Ilias Vasilopoulos

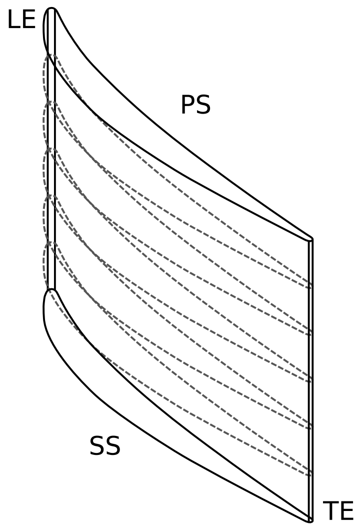


Figure 2.5: TUB blade parametrization in Parablading

3-D parametrization

As well as the U-bend and TUB parametric models described in Sec. 2.1, here also is the 3-D blade parametrization based on the lofting design approach. That is, a number of 2-D sections is radially stacked along the blade span. These sections are provided to an algorithm of Parablading that constructs four B-spline surfaces: leading edge (LE), trailing edge (TE), suction side (SS) and pressure side (PS), as illustrated in Fig. 2.5.

There are two additional stacking parameters to move the 2-D section: axial and circumferential shifts. Each 2-D profile parameter is manipulated by a law of evolution in the radial direction. The laws of evolution are defined as cubic splines and their control points are the actual design parameters of the optimization. There are 28 design parameters per section:

- five individual design parameters (r_I, r_E, β_I, β_E, T_{max}),
- ten control point parameters for the non-dimensional camber-line angle distribution,
- eleven control point parameters for the non-dimensional thickness distribution

- two stacking parameters: axial and circumferential shifts.

The blade has 21 sections, seven of which are considered as design sections. Other sections are moved accordingly to intermediate cubic spline values to ensure smooth surfaces. For this set-up, a potential number of design parameters is 196. However, the stacking parameters of the hub and tip sections are kept constant, therefore yielding the total number of 192 degrees of freedom.

3 Algorithmic differentiation of OCCT

3.1 Introduction to OCCT

The OCCT kernel [Ope] is the most-widely used open-source general-purpose CAD library. It consists of thousands of classes developed in C++ to provide solutions in surface and solid modeling, 2-D and 3-D visualization, and data exchange in standardized CAD formats.

The C++ sources of OCCT are organized into packages, the packages belong to toolkits and the toolkits are formed into modules. As illustrated on a dependency graph in Fig. 3.1, the OCCT library consists of seven modules, briefly explained as follows:

- **Foundation classes** — this module contains low-level classes such as primitive data types and structures, collections, linear algebra calculations, numerical algorithms, basic geometry types and geometric computations, exceptions handling, and memory management which includes smart pointers (also referred to as *handles*) to support dynamically created objects.
- **Modeling data** — provides data structures to implement a boundary representation (BRep) of a 3-D object. The BRep structure represents the shape as a composition of geometry and topology. The geometry is a mathematical description of an object, e.g. represented in a form of curves and surfaces. The topology defines connection between different geometrical entities. Besides necessary data structures, this module provides algorithms to create parametric curves and surfaces by approximation or interpolation of a given point set, algorithms of direct construction to build elementary geometric entities (e.g. lines, curves, planes, circles, cones, etc.), conversion of curves and sur-

faces to a NURBS form and computation of extrema between objects, i.e. the minimum distance between geometric entities.

- **Modeling algorithms** — provide a wide range of geometric and topological routines used in geometric modeling. This module is divided into two submodules: low-level and top-level routines. The low-level routines include algorithms for: (i) intersection between two curves, intersection between two surfaces or between a curve and a surface, (ii) projection of points to curves and surfaces or projection of a 3-D curve to a surface, (iii) line and circle construction considering constraints and (iv) conversion of a shape to the NURBS form. The top-level routines are used for: (i) direct construction of primitives (e.g. boxes, cylinders, spheres, etc.), (ii) fillets/chamfers to create smooth or sloped transitions between edges of a 2-D object or between adjacent faces of a 3-D object, (iii) linear and general-form sweeps to construct prisms and pipes, (iv) lofting (also known as the *skinning* algorithm) that constructs a shape out of a given set of 2-D cross-sections (as mentioned in Sec. 2, this algorithm is highly beneficial for this work), (v) Boolean operations (union, intersection and difference) that create a new shape from the given input shapes, (vi) meshing functionality to deal with tessellated representations of objects in a form of triangular surfaces.
- **Visualization** — provides data structures and tools for a graphical representation of an object (shape, mesh, etc.). This module contains low-level routines to work with the geometry and the topology and high-level routines to support a real-time rendering of objects using ray tracing.
- **Data exchange** — implements a set of interfaces to import or export data from or to various CAD libraries, e.g. STEP and IGES (*Initial Graphics Exchange Specification*) data formats. It relies on the *Shape Healing* library that provides a functionality to correct and adjust the topology and the geometry of shapes imported to OCCT from different CAD software tools. Another important feature implemented in this module is an STL (*STtereoLithography*) converter that translates OCCT shapes to the STL file format (triangular surfaces), which is commonly used in computer-aided manufacturing and 3-D printing.

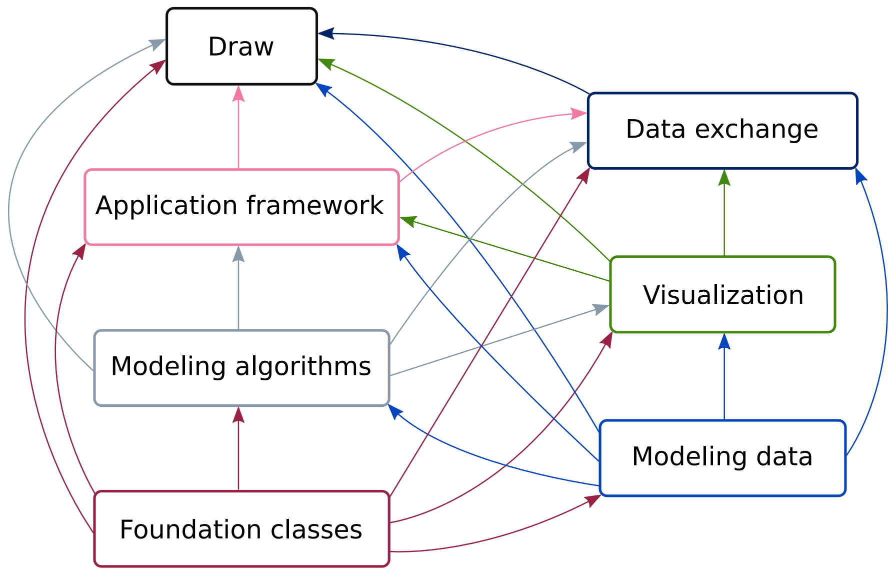


Figure 3.1: OCCT modules

- **Application framework** (also referred to as OCAF) — offers services to support a development of CAD applications based on the OCCT library. It handles user-specific data attributes and provides functionalities like open/save, copy/paste and undo/redo. Additionally, it is also possible to track and alter a modification history of a model (also known as *feature tree*), therefore directly influencing its parametrization.
- **Draw Test Harness** (also referred to as Draw) — a test environment implemented as a command interpreter with a graphical system. It offers a set of predefined commands based on Tcl (*Tool Command Language*) and a number of 2-D and 3-D viewers to test all libraries. Furthermore, the user can add commands to verify or demonstrate custom functionalities.

OCCT does not provide a graphical user interface (GUI) to design/parametrize components. For this reason, the user has to write a piece of code to actually construct objects. One possibility is to directly use the Draw module and its predefined set of Tcl commands, while the other option is to write the parametrization using the C++ language that can be beneficial for advanced and tailored applications like the ones developed in this study.

Regarding the users that prefer to have a GUI for the design purposes, while still having the OCCT kernel capabilities, there is a solution — FreeCAD [FC12]. FreeCAD is a powerful general-purpose 3-D modeling system that is completely open-source. It uses OCCT as the CAD engine and has a complete GUI implemented in the Qt framework.

3.2 Introduction to algorithmic differentiation

Algorithmic (Automatic) Differentiation (AD) is a technique to accurately and efficiently evaluate derivatives of a function given by a computer program. Any computer program, no matter how complicated, can be viewed as a sequence of elementary arithmetic operations ($+$, $-$, $*$, $/$) and elementary functions (e.g. `sin`, `cos`, `log`, `exp`, etc.). Let us assume there is a function $f(x)$ defined as follows:

$$f(x) = f_n(f(_{n-1}(f_{n-2}(\dots f_1(x))))) \ ,$$

where $f_1, f_2, \dots, f_n$ represent a sequence of elementary operations and functions that calculate intermediate results of the computer program. The derivatives of these statements can be easily calculated and coupled using the chain rule of calculus:

$$\frac{df(x)}{dx} = \frac{\partial f_n}{\partial f_{n-1}} \cdot \frac{\partial f_{n-1}}{\partial f_{n-2}} \cdot \dots \cdot \frac{\partial f_1(x)}{\partial x} \ .$$

In principle, AD software tools exploit this fact to generate a differentiated code that is able to compute the derivatives accurately up to a floating point round-off.

There are two basic modes of AD to compute the derivatives: *forward* (tangent) and *reverse* (adjoint). Both modes have been used in this work to compute first-order derivatives, however one can combine them to calculate higher-order derivatives. The forward mode evaluates Jacobian-vector products (also referred to as the *scalar* forward mode) and Jacobian-matrix products (also referred to as the *vector* forward mode), while the reverse mode evaluates vector-Jacobian products (also referred to as the *scalar* reverse mode) and matrix-Jacobian products (also referred to as the *vector* reverse mode).

A comprehensive introduction to AD can be found in the literature [GW08, Nau12], where the temporal complexity bounds for each of the modes as well as the expected memory requirements are elaborated.

3.3 Introduction to ADOL-C

ADOL-C is an open-source AD software tool, developed and maintained at Paderborn University in the research group led by Prof. Dr. Andrea Walther. It computes first and higher derivatives of vector functions that are defined by computer programs written in C or C++ [WG12].

ADOL-C defines the class `adouble` that augments the most common operators and elementary functions to enable the computation of derivatives. This is also referred to as the *operator-overloading* concept and it is typically allowed in object-oriented languages like C++.

To differentiate a certain code with ADOL-C, one has to replace the declaration types of all relevant *real* variables (e.g. `float`, `double`) with the `adouble` type. The relevant (also referred to as *active*) variables are: inputs (independents), outputs (dependents) and all intermediate variables that depend on independent variables and influence the output variables. A compiler error arises if certain intermediate variables depend on *active* variables but are not declared as *adoubles*, since the `adouble` class on purpose does not support implicit and explicit type conversion to native data types.

Based on the differentiation options, the `adouble` class has two kinds of implementations:

- *traceless* (differentiation mode: forward),
- *trace-based* (differentiation modes: forward and reverse).

These options impose different ways of derivative computation. The *traceless* option computes the derivatives directly together with the function (*primal*) evaluation. It is easier to use and understand, since all operators implement both *primal* and derivative statements. On the other hand, the *trace-based* option uses the operator-overloading to generate an internal representation of the code to be differentiated — called *trace*. After the *trace* is created, one calls ADOL-C drivers to evaluate the derivatives. Moreover, this option is more powerful than the *traceless* one as it features the reverse mode of AD that has a potential to dramatically reduce the temporal complexity of the derivative computation.

Both *traceless* and *trace-based* options support the derivative computation in *scalar* (one direction) and *vector* mode (many directions). For example, when using the *scalar* forward mode of AD, one has to execute the differentiated sources separately with respect to each independent variable to compute the corresponding derivatives. As this evaluation process can become very time consuming, a more efficient option is to integrate the *vector* forward mode of AD that enables the derivative computation with respect to many design parameters simultaneously, i.e. in a single code execution.

3.3.1 Traceless differentiation variant

The *traceless adouble* class is defined in the `adtl` namespace (`<adolc/adtl.h>` header). It contains a *double*-type pointer to an array of size $p + 1$, named *adval*, where the zeroth element represents the real part of an *adouble* object (the *primal* value), while the other elements correspond to directional derivatives. The number p is a user-defined constant and it represents the desired number of directions.

By default the constant p is set to 1 which implies that the *scalar* mode of the derivative computation is being used. Therefore, the *vector* mode is activated when $p > 1$. In that case, the i-th element of the *adval* array corresponds to the i-th direction, where $i = 1, \dots, p$, meaning that each direction has its own dedicated element in the *adval* array. This enables simultaneous derivative computation with respect to many design variables.

There are two possibilities to set the number of directions. The simpler option is to use the `adtl::setNumDir(`*p*`)` method. However, it is important to call this method before any *adouble* initialization such that dynamically allocated memory assigned to *adval* pointers is consistent during the program execution. Otherwise, changing the number p at any other part of the program execution can cause a run-time exception because the pointers get corrupted. Another obstacle are *static* variables that are initialized even before the `main` function is executed, which means there is a possibility that *static adoubles* have a different number p than the one requested by the user. This issue was encountered in the differentiated version of OCCT. For this reason, the recommended way that avoids any chance of memory corruption is to change the number of directions directly in the ADOL-C sources (file `adouble_tl.cpp`). To apply the change, one has to type the `make && make install` command in ADOL-C directory to re-compile the *traceless* library, which is finished in a couple of seconds.

To get and set the *primal* value of an *adouble* object, one calls the methods *getValue* and *setValue*, respectively. Likewise, the methods *getADValue* and *setADValue* serve to get and set the derivative components. Other possibilities to manipulate these values are via *adouble* constructor or assignment operator (=). More details can be found in the ADOL-C manual provided with the sources.

3.3.1.1 Boost memory management

A complex application like OCCT may consist of thousands of *adouble* objects that are dynamically allocated using the `new` operator. When using this operator, the memory management depends on the implementation of the C++ standard library as well as on the operating system. For a fast and efficient memory management, one can employ pool-based allocators in conjunction with the C++ standard library containers. One of such pool-based allocators is implemented in the Boost C++ framework, namely the *Boost.Pool*, and it can be used in the *traceless* ADOL-C library.

The Boost pool allocator does not change the behavior of the operating system, but it adds a layer between the application and the operating system. When a program starts, the pool allocator requests a memory block from the operating system (e.g. using the operator `new`). Moreover, this memory is partitioned into segments with the same size. The size of a segment is set accordingly to the number of directions p, i.e. to accommodate $p + 1$ `double` elements. Every time the application requests the memory from the *Boost.Pool* to initialize an *adouble* object, the framework accesses the next free segment and assigns it to the *adval* pointer. Therefore, the segments are just marked as *used* or *free* on the pool layer without actually dealing with the operating system. This approach is very useful if many objects of the same size have to be created or destroyed frequently, since the memory can be provided or released quickly.

The *Boost.Pool* allocator is initialized in the `adouble_tl.cpp` file with the default maximum number of 10,000 segments, that can be changed by the user. For the OCCT differentiation purposes, this number is sufficient.

3.3.2 Trace-based differentiation variant

The *trace-based adouble* class is defined in the `<adolc/adouble.h>` header. As opposite to the *traceless adouble* class, the *trace-based* variant does not calculate the derivatives *on-the-fly* with the *primal* evaluation. Here, first the internal representation of the function to be differentiated — *trace* — is generated using the operator-overloading concept.

The *trace* is a sequential data set that contains all *adouble*-related instructions recorded during an active section of the code to be differentiated. It is divided into four internal arrays: (i) *operation* — contains unique identifiers of operations (instructions), (ii) *location* — contains unique identifiers of *adouble* objects, (iii) *value* — contains values of constants that contribute to the derivative computation (variables declared as e.g. *int*, *float* or *double*) and (iv) a *Taylor* buffer. The latter is optional and serves to keep the values of intermediate results. All buffers can be written to four corresponding files on a hard drive, however it is preferred to keep them in the random-access memory (RAM) due to faster performance of the differentiated sources.

The default array length of buffers is defined in `<adolc/internal/usrparms.h>` with parameters `OBUFSIZE`, `LBUFSIZE`, `VBUFSIZE`, and `TBUFSIZE`, that are by default set to 524288. They can be changed in the header before compiling the ADOL-C library, however the recommended way is to create the `.adolcrc` file in the directory where the differentiated sources are executed. In this file, the user can set all buffer parameters using the notation: `"`*`VARIABLE`*`" = "`*`VALUE`*`"`, where the quotation marks are mandatory. The ADOL-C library searches for this file right at the beginning of the program execution and initializes all buffers accordingly. In the case that application requirements exceed the predefined buffer lengths, the buffers are saved to the hard drive.

It is important to note that one has to carefully choose the buffer lengths because ADOL-C *trace* manager may request more memory than it is available on the operating system, which could cause a run-time exception. The exact number of bytes depends on the operating system and it is calculated as follows: `OBUFSIZE * sizeof(unsigned char) + LBUFSIZE * sizeof(unsigned int) + VBUFSIZE * sizeof(double) + TBUFSIZE * sizeof(double)`. When using the default lengths, the *trace* takes approximately 10 MB of memory. This is sufficient for simple applications, however the differentiated OCCT sources required a few GB of memory to store the *trace*. For this reason, the buffer lengths are tailored accordingly to application demands. To find out the exact lengths of buffers required by an application, one can call the function `printTapeStats(std::cout, tag)` after the tracing process, which prints information about the *trace* status to the standard output. This information is beneficial, when entering custom buffer lengths into the `.adolcrc` file, which was exactly the procedure for dealing with the differentiated OCCT.

The active section of the code to be differentiated is marked with ADOL-C routines *trace_on(tag,keep)* and *trace_off(file)*. Pairs of these routines can appear anywhere in the code, however they must not overlap. The parameter *tag* is a non-negative integer that identifies a certain *trace* and therefore is considered to be unique. Otherwise, if the user provides the same tag to many traces, their content will be overwritten. The flag *keep* should be given as either 1 (*true*) or 0 (*false*). In the case it is *true*, the *primal* values of intermediate results are saved to the *Taylor* buffer during tracing. This is useful when evaluating the derivatives using the reverse mode of AD right after the *trace* is generated. The flag *file* indicates whether the *trace* should be stored on the hard drive, no matter how large the buffers are.

To mark the independent and dependent variables of the active section, one uses bitwise shift operators `<<=` and `>>=`, respectively. For this purpose, they are overloaded in the *trace-based adouble* class. One declares the independent variables (*independents*) immediately after calling the `trace_on` routine and the dependent variables (*dependents*) at the end of the active section, i.e. before calling the `trace_off` routine.

Once the *trace* is successfully generated, one calls the ADOL-C drivers to evaluate the *primal* and the derivatives up to an arbitrary order using the forward/reverse mode of AD. There is a vast set of ADOL-C drivers available that are explained in the manual provided together with the sources. These routines can be also used to evaluate the *trace* at a different set of arguments, i.e. different values of the independent variables than the original ones that were used during the *tracing* process. This approach can significantly reduce run times, however it is important that a control flow of a program stays unaltered, e.g. comparison operators involving *adoubles* yield the same results. If there is a change in the control flow, ADOL-C gives a warning that the *tracing* process has to be repeated.

3.3.2.1 Activity analysis

Activity analysis is one of the recent features of ADOL-C implemented by Dr. Kshitij Kulshreshtha. Its purpose is to improve the computation efficiency of the *trace-based* differentiation variant when dealing with the differentiated sources that contain a large amount of *adoubles* that are in fact constants and therefore should not be treated as differentiable quantities. This situation typically happens when differentiating a certain source code by using the so-called *typedef* differentiation approach, explained in Sec. 3.4.6, where almost all *real* variables are re-declared with the *adouble* type. However, the activity analysis discovers all constant *adoubles* during the tracing process and deals with them accordingly.

In principle, every *adouble* object used in the active section has a corresponding *char* variable stored in the ADOL-C buffer memory that can be either 1 (*true*) or 0 (*false*). At the beginning of the tracing process, this activity state is set to *true* for all independent variables once they are marked with the `<<=` bitwise shift operator. Later on, all intermediate *adoubles* inherit the activity state from their predecessors. Based on the activity states, only the operations with at least one active *adouble* are recorded on the *trace*, otherwise they are ignored. Moreover, in case for operations with two *adouble* operands where only one is active, the other one is treated as a constant, e.g. like a regular *double* variable.

Although the activity analysis imposes additional *if*-statements for each overloaded operator of the *adouble* class to test the activity status of operands, and therefore may increase the tracing time, it reduces the overall *trace* memory consumption. Furthermore, the invested effort for the activity analysis upon tracing pays-off when evaluating a smaller *trace* with the ADOL-C drivers, which improves the performance of the differentiated sources.

By default, the activity analysis is disabled and one can enable it, while configuring the ADOL-C makefile, with a flag `--enable-activity-tracking`. The activity analysis is successfully validated and verified with the differentiated OCCT sources by the author and its performance is presented in Sec. 3.6.2.

3.4 Approaches of differentiating OCCT

As mentioned in the Sec. 3.3, the ADOL-C library is integrated into OCCT by injection of its specific *adouble* type instead of the native *real* type used by OCCT. That is, one has to use the specific ADOL-C type as the declaration type to all relevant *real* variables, i.e. variables that depend on the design variables and influence the output variables. Otherwise, if the *adouble* chain is broken in some part of the code, the derivative values will be incorrect. When applying this to a complicated *object-oriented* code like OCCT, the process of replacing the declaration types is not simple. Different approaches of ADOL-C integration are investigated and discussed in this section.

To perform the algorithmic differentiation of the OCCT kernel using ADOL-C, several possible approaches with respect to the code modification were investigated:

1. **Code duplication**: duplicate original entity classes and introduce *AD* specialized properties and methods.
2. Use an **inheritance model** to implement child classes of the original entities, defining extra *AD* properties and methods.
3. Create completely isolated entry points designed for *AD* only — **controller** approach.
4. **Templating** approach: apply generic programming (C++ templates) to every original class, such that it becomes a class template.
5. **Typedef** approach: redefine the *Standard_Real* type used in OCCT to be the *adouble* type of ADOL-C.

3.4.1 A sample class

To demonstrate AD in principle and explain every code modification approach, let us consider a geometrical entity class that corresponds to a two-dimensional B-spline curve — *Geom2d_BSplineCurve*. A shortened and simplified presentation of the complex *Geom2d_BSplineCurve* class is shown in Listing 3.1. The aim is to differentiate the shape of a curve with respect to its properties/parameters:

- poles — collection of two-dimensional Cartesian points (class: *gp_Pnt2d*),
- weights — collection of *Standard_Real* values (in OCCT, *Standard_Real* is the alias for the native *double* data-type).

Listing 3.1: Geom2d_BSplineCurve class (simplified)

```
class Geom2d_BSplineCurve : public Geom2d_BoundedCurve
{
public:
  Geom2d_BSplineCurve(TColgp_Array1OfPnt2d& Poles,
    TColStd_Array1OfReal& Weights, TColStd_Array1OfReal& Knots,
    int Degree, bool Periodic = Standard_False, ...);
  void D0 (const Standard_Real U, gp_Pnt2d& P) const Standard_OVERRIDE;
private:
  Handle(TColgp_HArray1OfPnt2d) poles;
  Handle(TColStd_HArray1OfReal) weights;
};
```

The *gp_Pnt2d* class is briefly described in Listing 3.2. It supports operations like rotation, translation, mirroring, etc., but uses another entity to store its coordinates — *gp_XY*. The *gp_XY* class represents a Cartesian coordinate entity in two-dimensional space and supports basic algebraic operations. As private fields, it contains two *Standard_Real* variables (Xp, Yp).

Listing 3.2: gp_Pnt2d class (simplified)

```
class gp_Pnt2d
{
public:
  gp_Pnt2d();
  gp_Pnt2d(const gp_XY& Coord);
  gp_Pnt2d(const Standard_Real Xp, const Standard_Real Yp);
private:
  gp_XY coord;
};
```

Furthermore, as shown in Listing 3.1, OCCT features special collection (array) data-types: *TColgp_HArray1OfPnt2d* and *TColStd_HArray1OfReal*. An instance of such collections is managed by *handles*. The *handle* is a native smart-pointer of OCCT used for automatic memory management of shared objects. Additionally, OCCT also features collections that are not managed by *handles*, e.g. *TColgp_Array1OfPnt2d* and *TColStd_Array1OfReal*.

The following sections describe differentiation approaches with respect to the sample class *Geom2d_BSplineCurve* and its parameters (*poles* and *weights*).

3.4.2 The code duplication approach

The **code duplication** approach creates *AD*-classes from the existing original entities and introduces *AD*-specialised properties and methods. To differentiate the sample class using this approach, several steps are required. The first step is to create new classes:

- *gp_XY_AD* — the two-dimensional Cartesian coordinate entity that has (*Xp*, *Yp*) variables declared as *adoubles*,
- *gp_Pnt2d_AD* — the two-dimensional Cartesian point class that uses the previously implemented *gp_XY_AD* class to store its coordinates.

The next step is to define new collection data-types managed by the OCCT *handles* (*TColgp_HArray1OfPnt2d_AD* and *TColStd_HArray1OfReal_AD*) as well as collection data-types not managed by the OCCT *handles* (*TColgp_Array1OfPnt2d_AD* and *TColStd_Array1OfReal_AD*), in order to manipulate with *gp_Pnt2d_AD* and *adouble* objects.

Some additional methods are needed. A point on a curve is calculated in the *D0* method. Therefore, a new method is introduced in the *Geom2d_BSplineCurve_AD* class — *D0_AD*. As the name implies, this method works with *AD* poles and weights. Starting from this function, the *adouble* injection, i.e. the change of the variable type, has to be consistently propagated to all dependent routines such that the chain rule is not broken. It is important to note that these routines are not just in the B-spline class, but also in other classes that provide B-spline evaluation algorithms. Hence, they are all differentiated using the same approach.

Finally, the new *AD*-class named *Geom2d_BSplineCurve_AD* is presented in Listing 3.3. As one can notice, the elements of the source code of the original *D0* method are still present as well as the original collections of poles and weights. The reason for keeping the *D0* method is that the *Geom2d_BSplineCurve_AD* class inherits from an abstract class where the *D0* method is defined as *pure virtual.* Hence, there is a requirement that any derived class that inherits from *Geom2d_Curve* has to implement such defined methods. Not only *D0* is required to be implemented by the parent abstract classes, but also many other methods, e.g. *D1*, *D2* and *D3*, have to be implemented. This implies that the original collections of poles and weights are necessary, because they are used in the original methods like *D0.* One option is to keep them in the *AD*-class and synchronize them with the *AD*-collections, whereas the other possibility is to add a certain code in each original method that will create '*double*' versions of poles and weights *on-the-fly* by copying the values from the existing *AD*-collections wherever they are requested.

Listing 3.3: Geom2d_BSplineCurve_AD class (simplified)

```
class Geom2d_BSplineCurve_AD : public Geom2d_BoundedCurve
{
public:
  Geom2d_BSplineCurve_AD(TColgp_Array1OfPnt2d_AD& Poles,
    TColStd_Array1OfReal_AD& Weights, TColStd_Array1OfReal& Knots,
    int Degree, bool Periodic = Standard_False, ...);
  void D0 (const Standard_Real U, gp_Pnt2d& P) const Standard_OVERRIDE;
  void D0_AD (const adouble U, gp_Pnt2d_AD& P) const;
private:
  Handle(TColgp_HArray1OfPnt2d) poles;
  Handle(TColStd_HArray1OfReal) weights;
  Handle(TColgp_HArray1OfPnt2d_AD) poles_AD;
  Handle(TColStd_HArray1OfReal_AD) weights_AD;
};
```

The code duplication approach may be useful for exploratory differentiation, e.g. for a proof of concept for the algorithmic differentiation of the sample class, but to use it on a full-scale differentiation of OCCT is not reasonable. The first and most important concern is the maintenance of the differentiated code alongside with the original code development. The second issue, noticeable in the B-spline example, is that the approach imposes duplicate collections of poles and weights in the same object, which can be error-prone. Simply replacing the original collections with the

AD collections is not feasible because it will lead to compilation errors. The original signatures should remain untouched, otherwise one has to modify the signatures of the parent classes as well as other source files that are using them. Another concern that needs to be addressed is data consistency: if the original collections are modified, one has to make sure that at the same time the *AD* collections are updated as well. Without implementing a synchronization between these collections, this approach is not a good choice because one has to track the original collections throughout the computation. Moreover, the same object has duplicated methods (e.g. the *D0* and *D0_AD* methods). For these reasons, the code duplication approach is rejected for the differentiation of the full OCCT kernel.

3.4.3 The inheritance approach

An example of differentiating the *Geom2d_BSplineCurve* class with the **inheritance** approach is shown in Listing 3.4. The difference between this approach and the code duplication approach presented in Listing 3.3 is that the *Geom2d_BSplineCurve_AD* class now inherits from the original *Geom2d_BSplineCurve* class instead of the original parent class *Geom2d_BoundedCurve*. Therefore, it is easier to distinguish between the *AD* versions and the original entities, but this approach is not a good choice because it does not represent the real inheritance model:

- The final object will contain duplicate collections of poles and weights.
- The final object will contain duplicate methods (e.g. the *D0* and *D0_AD* methods).
- The majority of methods from a parent class will not be used (e.g. in the case of B-spline curve, only the *D0* method is interesting for differentiation).

Listing 3.4: Geom2d_BSplineCurve_AD class that inherits from Geom2d_BSplineCurve (simplified)

```
class Geom2d_BSplineCurve_AD : public Geom2d_BSplineCurve
{
public:
  Geom2d_BSplineCurve_AD(TColgp_Array1OfPnt2d_AD& Poles,
    TColStd_Array1OfReal_AD& Weights, TColStd_Array1OfReal& Knots,
    int Degree, ...);
  void D0_AD (const adouble U, gp_Pnt2d_AD& P) const;
private:
  Handle(TColgp_HArray1OfPnt2d_AD) poles;
  Handle(TColStd_HArray1OfReal_AD) weights;
};
```

This approach shares the major drawbacks with the code duplication approach that are related to duplicate properties and methods in the same object. Therefore, it is also rejected as a solution for the differentiation of the full OCCT kernel.

3.4.4 The controller approach

The **controller** approach introduces new lightweight classes — *controllers*, which are isolated entry points for AD only. Considering the two-dimensional B-spline curve class, a new controller class *Geom2d_BSplineCurve_ADController* consists of:

- a pointer to the original curve (entity),
- the *AD* collections as properties, together with corresponding methods for getting and setting them (also known as *getters* and *setters*),
- designated methods, like the *D0_AD* (almost exact copy of the original one), which are modified for dealing with *adoubles*.

An example of the controller class for *Geom2d_BSplineCurve* is shown in Listing 3.5. The controller class does not inherit any OCCT classes, therefore no additional methods have to be implemented, except the ones needed for AD. To further simplify the controller implementation, it may be declared as a *friend* of the original curve in order to obtain a direct access to its private and protected members.

Listing 3.5: Geom2d_BSplineCurve_ADController class (simplified)

```
class Geom2d_BSplineCurve_ADController
{
public:
  Geom2d_BSplineCurve_ADController(
    const Handle(Geom2d_BSplineCurve)& C);
  void D0_AD (const adouble U, gp_Pnt2d_AD& P) const;
private:
  Handle(Geom2d_BSplineCurve) curve; //original curve
  Handle(TColgp_HArray1OfPnt2d_AD) poles;
  Handle(TColStd_HArray1OfReal_AD) weights;
};
```

Although the controller approach yields better code in terms of readability and usability in comparison with the two previously described approaches, this approach is not selected here. All three approaches explained so far require an additional effort to differentiate extra methods, i.e. *PrepareEval*, *BuildEval* and *Eval*, used in the recursion tree of the *D0* method. These methods are members of the *BSplCLib* class (*B-spline Curve Library*). Therefore also a substantial amount of code duplication arises which could be reduced for some cases by reorganizing and modifying the original OCCT sources in a generic (*template*) way. Before doing so, one does not know how many methods except *D0* would have to be differentiated and how deeply into the call tree the sources need to be edited to propagate the *adouble* injection to all extra *AD* methods. Once again, it would be very hard and time consuming to maintain such a code.

3.4.5 The templating approach

In order to avoid the code duplication as much as possible, one could consider the **templating** approach to apply a generic programming (C++ templates) to the OCCT classes such that they become class templates, as shown in Listing 3.6. This approach is ideal from the maintenance point of view because there would be no additional *AD* classes introduced. Instead, one would choose between the original or differentiated functionality just by specifying a template parameter, i.e.:

- *Geom2d_BSplineCurve<double>*,
- *Geom2d_BSplineCurve<adouble>*.

The template-based approach allows keeping only one version of the OCCT library in the final application. That is, one does not have to recompile the source code to switch between the original and AD version (and vice-versa). This allows to use the AD entities only when the gradients are required, otherwise, the original OCCT functionality is sufficient. In this way it is possible to avoid unnecessary performance overhead introduced when using the AD classes in the whole geometric kernel.

The code duplication risk is minimal, but still exists, because *adouble* objects cannot just fit everywhere. There are a lot of places in the code, some of which will be mentioned later, where one must interfere and add/modify some code lines in order to make everything work. For such cases, one has to use *template specialization*, which is somehow similar to function overloading and allows the user to write specific method implementations that will be used for *adoubles*. In most of the cases, these specific implementations will not be that much different from the original ones, which is why they impose a certain code duplication. Furthermore, it is important to note that these methods have to be maintained together with the templated code.

Although the final outcome of using this approach is advantageous and reduces the code duplication as much as possible, it is the most intrusive way of differentiating OCCT because it requires to change (*template*) the complete OCCT code structure. Moreover, it reduces significantly the readability of the kernel's code, especially when applying the template specialization for the cases where *adoubles* require additional code modifications. Therefore, it is not selected to differentiate the OCCT kernel.

Listing 3.6: Geom2d_BSplineCurve template class (simplified)

```
template<class T>
class Geom2d_BSplineCurve : public Geom2d_BoundedCurve<T>
{
public:
  Geom2d_BSplineCurve(NCollection_Array1<gp_Pnt2d<T>>& Poles,
    NCollection_Array1<T>& Weights, TColStd_Array1OfReal& Knots,
    int Degree, ...);
  void D0 (const T U, gp_Pnt2d<T>& P) const Standard_OVERRIDE;
private:
  Handle(NCollection_SharedArray1<gp_Pnt2d<T>>) poles;
  Handle(NCollection_SharedArray1<T>) weights;
};
```

3.4.6 The typedef approach

The **typedef** specifier is a reserved keyword in C/C++ used to declare an alias for another data type. OCCT defines the alias *Standard_Real* for the *double* data-type and it has been used as an entry point for AD, changing *Standard_Real* to become the alias for the *adouble* type of ADOL-C. In the so-called typedef approach, almost all declarations of *doubles* variables are replaced by the declaration of *adouble* variables. The main advantage of this approach is that the code modification should be minimal with the drawback of sacrificing memory and efficiency to some extent because almost all *double* variables, even the ones that are not needed for AD, become *adouble* objects. With the minimal code modification, it should be also straightforward to maintain the differentiated code alongside the original code development. Moreover, the typedef approach is significantly less error-prone compared to the methods described previously. That is, the risk of a human mistake which can lead to unnoticed breaking of the chain rule is minimal. Therefore, it is a very appropriate solution for differentiating large and complex source codes like OCCT.

Although the idea about the *typedef* approach looks simple, it is not as straightforward as one would expect. The differentiation of OCCT version 7.0 involved a significant amount of code modification and even after the successful compilation, a large number of run-time errors had to be resolved during the testing phase.

It is important to note that the default tests provided with the OCCT kernel are related only to the *primal* functionality. Even though the *primal* tests went successfully as described in Sec. 3.5.1, this achievement does not imply the successful differentiation as these tests use only the *primal*. To make sure that the AD version is sound, it is necessary to add extra test cases which will validate the derivatives. In this study, the AD functionality of OCCT was successfully validated using only the parametric U-bend and TUB stator models, as elaborated in Sec. 3.5.2 and Sec. 3.5.3, respectively.

3.4.7 Compile and run-time issues of OCCT differentiation

This section describes the difficulties faced during the *typedef* implementation and the corresponding solutions. Some of the compile-time issues were related to:

- On a very low level of the OCCT kernel, static assertion is used to check whether the size of *Standard_Real* is equal to the size of 2×*Standard_Integer* (which is the *typedef* for *int* type). Once the *typedef Standard_Real* corresponds to *adouble*, the static assertion fails, therefore yielding a compile-time error. The reason of having such an assertion is due to a definition of the *union* with two members of {*Standard_Real*, 2×*Standard_Integer*}. *Union* is a user-defined type in C/C++ in which all its members occupy the same physical space in memory. Until C++11 standard, *unions* were allowed to store only primitive data types. Without further investigation how to integrate a non-primitive type with C++11 standard, the author kept the *double* data-type here because the *union* appears only in the low level of OCCT that is not related to the modeling algorithms. This affects the class *FSD_BinaryFile*, where the *union* is used to inverse bytes of a *real* number. To overcome compile-time issues, keeping the *double* variables results in a certain modification of other sources that used the *union* in order to create a sort of bridge between *doubles* and *adoubles*.

- Hundreds of places in the OCCT code involve explicit/implicit conversion of *Standard_Real* to *int* type. In the terms of algorithmic differentiation, this could cause a problem because an integer object does not carry along the derivative information. By doing such a type change, the computational graph of the function to be differentiated is disconnected in that part. Hence, the chain rule is broken and wrong derivatives may be computed as a result. This is the reason why such a cast is not automatically supported in the *adouble* class. Being aware of the risk, the type-casting has been allowed simply by using the *getValue* method on the *adouble* object which extracts the primal (*double*) value of it. A small example where this could cause a problem is related to a method of the *BndLib_Box2dCurve* class, as shown in Listing 3.7. Let us assume that the input *adouble* variables `aT` and `aPeriod` carry the derivative information. Such a derivative information will not be propagated in the line where `k` is computed because the expression `(-aT/aPeriod)` is converted to *int*. On the following line, the result `aTRet` is computed using the previously described variable `k` with the truncated derivative information. On the other hand, there is no problem if the inputs `aT` and `aPeriod` are not 'active', i.e. if they do not carry derivative information. Since this issue is application-dependent, the derivatives have to be carefully verified or the original code structure has to be modified to avoid such problems.
- *Standard_Real* is not the only *typedef* for *double* in OCCT. Although it is broadly used in OCCT, there are many other *typedefs*: *Quantity_Acceleration*, *Quantity_Area*, *Quantity_Coefficient*, *doublereal*, *GLdouble*, etc. Most of them are replaced with *Standard_Real* to keep consistency across the OCCT kernel, but there are also exceptions where the native *double* type is required. These exceptions mostly relate to packages that belong to the Visualization module of OCCT, e.g. the *OpenGL* package which uses both *doubles* and *floats*. This

means that the *adouble* presence is reduced in such packages as much as possible (which is reasonable), but not entirely. The issue lies in the fact that visualization packages use geometric entities like point, vector and axis, that already contain *adouble* objects because they use *Standard_Real typedef*. To overcome this problem is not simple, but it is successfully resolved by introducing intermediate variables and breaking the chain rule wherever required.

- Functions that are part of external libraries cannot work with *adoubles*. Therefore, the compiler reports type mismatch in such places. Depending on the function arguments, whether they are pointers or not, it was necessary to substitute *adoubles* with *doubles* or call the *getValue* method on the *adouble* objects. This includes also functions like `modf` (decomposes a number into integer and fractional parts) and `fmod` (calculates remainder of the floating point division operation). They are C-functions (defined in the header *cmath*) and the differentiation rule for them is simply not defined. Therefore, similarly to the type casting described above, the chain rule is broken when using these functions.
- Functions for printing to an output, like *sprintf*, cannot accept *adouble* as an argument. However, the *sprintf* function is used in a lot of places in OCCT, mostly referred to printing standard CAD output formats like STEP and IGES. Since *sprintf* is a C-function and cannot be overloaded in ADOL-C, the *getValue* method was used here as well.

Listing 3.7: *BndLib_Box2dCurve* class method (simplified)

```
Standard_Real BndLib_Box2dCurve::AdjustToPeriod(
  const Standard_Real aT,
  const Standard_Real aPeriod)
{
  Standard_Integer k;
  Standard_Real aTRet;
  aTRet=aT;
  if (aT<0.) {
    //possible derivative loss:
    k=1+(Standard_Integer)(-aT/aPeriod).getValue();
    aTRet=aT+k*aPeriod;
  }
  //...
  return aTRet;
}
```

Furthermore, some of the run-time issues were related to:

- The left and right shift operators (<< and >>) for printing to an output are overloaded in the *adouble* class. Since they are also used in the OCCT output system, the files were corrupted by *adoubles*, because the derivative information is also printed. There is no need that files like STEP contain such an additional information. Hence, the solution was just to extract the *primal* values of *adoubles* using the *getValue* method wherever necessary.
- C dynamic memory allocation is used in the OCCT kernel. This causes errors once the *adoubles* are initialized. An *adouble* object has to be initialized using its constructor, such that the correct memory amount is allocated. The C-function *malloc* does not achieve that, triggering 'segmentation fault' errors upon program execution. For this reason, the functions *malloc/free* are replaced with the C++ operators *new/delete*. Moreover, the C-functions *memset* and *memcpy* are replaced with *for*-loops in order to manually assign or copy the values from one pointer to another. Otherwise, a memory exception occurs. Even more complex low-level memory management is used in one specific package of OCCT that is non-differentiable with ADOL-C, as described in Sec. 3.5.1.
- In many places of the OCCT code, the explicit conversion from *real* numbers to *adoubles* is required in the case where these numbers are passed as arguments to the specific overloaded methods. For example, consider the *SetCoord* method which is overloaded in OCCT in two different ways: (i) the first option *SetCoord(Standard_Integer, Standard_Real)* sets a coordinates value by its index and (ii) the second option *SetCoord(Standard_Real, Standard_Real)* specifies each coordinate value independently. A case in the code where developer uses the second method with *real* numbers, e.g. *SetCoord(6., 8.)*, is not correctly identified in the differentiated sources because the compiler calls the method with primitive types as arguments, which is the first method. Without any interaction this may not produce a runtime error, but certainly the values are wrong. The correct way in such cases is to add an explicit conversion, i.e. *SetCoord((Standard_Real) 6., (Standard_Real) 8.)*.

Both *traceless* and *trace-based* differentiation variants have been integrated into OCCT by using the *typedef* approach, therefore yielding two different versions of the differentiated OCCT sources. Their validation is described in the following section.

3.5 Verification of differentiated OCCT

3.5.1 Primal functionality validation of differentiated OCCT

After the successful compilation of the differentiated OCCT kernel, the first step is to verify the original (*primal*) functionality. For such purposes, OCCT provides its own *Automated Testing System* which consists of more than 10,000 tests related to all OCCT modules. As mentioned in Sec. 3.4.7, a large number of run-time errors had to be resolved during the testing phase. A very small number of issues could not be resolved, see Table 3.1 for details.

Table 3.1: OCCT *Automated Testing System* final results

Tests marked: OK	Tests marked failed: FAILED	**Success rate**
11,306	374	**97%**

The majority of the tests marked *failed* are related to the package *AdvApp2Var* which deals with an approximation of functions based on Legendre polynomials. This package is a legacy code that was firstly written in Fortran and later automatically translated to C code. It involves low-level memory management instructions which make *adouble* handling quite difficult. Whenever it is used by the testing system, a run-time memory exception occurs. There are two possibilities to solve this problem: (i) rewrite the package such that it follows the C++ standards for memory management or (ii) use the source-transformation AD tool Tapenade to differentiate the *AdvApp2Var* source package and couple it with ADOL-C.

The following section elaborates the verification of derivatives, using the parametric models of U-bend and TUB stator test-cases described in Sec. 2.1.

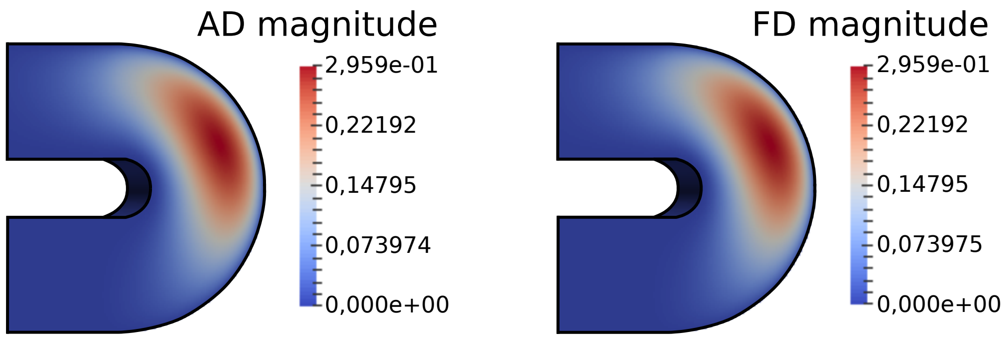


Figure 3.2: U-part geometric sensitivities evaluated with AD (left) and FD (right)

Table 3.2: AD and FD values comparison for several U-bend point coordinates

AD value	FD value	Abs. difference
0.000384**36**	0.000384**26**	1.02e-07
0.063391**89**	0.063391**25**	6.41e-07
0.1561**5249**	0.1561**4906**	3.43e-06
0.1287**4039**	0.1287**3815**	2.24e-06
⋮	⋮	⋮
0.2745**9387**	0.2745**8089**	1.30e-05[*]

[*]Maximal difference

3.5.2 Gradient verification using U-bend parametrization

The derivative verification is firstly performed on the U-bend test-case described in Sec. 2.1.1. As a representative example, let us compare the surface sensitivities with respect to one design parameter calculated with the *traceless* forward mode of AD and finite differences, shown in Fig. 3.2. As noticeable, the overall magnitude plots coincide to a very high extent. Furthermore, the quantitative comparison presented in Table 3.2 confirms mutual agreement.

Moreover, the same surface sensitivities are also verified with a Taylor test:

$$f(x+h) - f(x) - h\frac{\partial f}{\partial x}(x) = \mathcal{O}(h^2). \tag{3.1}$$

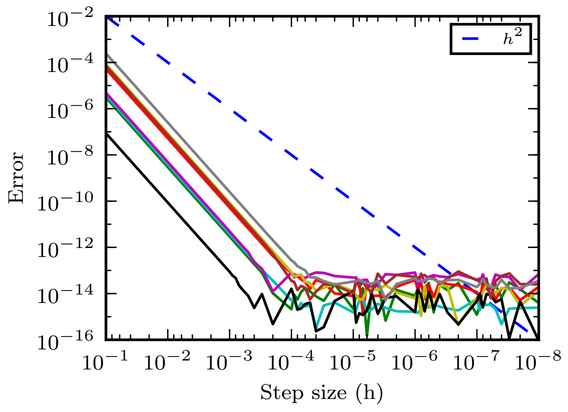


Figure 3.3: Taylor test overview for eight U-bend surface point coordinates

The Taylor test is evaluated on a number of arbitrary surface point coordinates with a range of step sizes $h \in [10^{-1}, 10^{-10}]$. The error plots (the left-hand side of Eq. (3.1)) in eight surface point coordinates are presented in Fig. 3.3. Here, one observes even better convergence than the theoretical convergence rate of h^2. This behavior continues until $h = 10^{-4}$, where the errors reach machine precision.

Once the *traceless* forward mode of AD is successfully verified, the next step is to compare the derivatives against the *trace-based* differentiation modes. The results presented in Table 3.3 and Table 3.4 show some small disparity (close to the machine precision) between the gradients. This is due to different implementations of certain overloaded operators (*power* and *division*) in ADOL-C with respect to the *trace-based* and *traceless* options. Not only the derivative calculation but also the primal evaluation is affected in the same order of magnitude. The differences between the *trace-based* and the *traceless* variants are therefore small enough not to yield radically different CAD models, and hence both can be used equally.

This verification ensures the correctness of the computed derivatives only for the computational path exercised by the U-bend geometry. Although this test-case does use a lot of methods from the OCCT kernel, it represents a very small part of the complete OCCT capability. Clearly, adding definitions for all possible input and output variables to all regression tests is not a feasible task. An unanswered challenge to the AD community is how to not just automatically produce the derivative code,

Table 3.3: AD traceless-forward and AD trace-based forward gradient comparison for several U-bend point coordinates

Traceless-forward gradient	Trace-based forward gradient	Abs. difference
1.03293863915**149**e-01	1.03293863915**283**e-01	1.34e-13
2.3670708698791**7**e-01	2.3670708698791**3**e-01	4.00e-15
1.286827619752**10**e-01	1.286827619752**25**e-01	1.50e-14
1.256524426**70046**e-02	1.256524426**69980**e-02	6.60e-15
⋮	⋮	⋮
2.2469959328**0905**e-01	2.2469959328**1250**e-01	3.45e-13*

*Maximal difference

Table 3.4: AD traceless-forward and AD trace-based reverse gradient comparison for several U-bend point coordinates

Traceless-forward gradient	Trace-based reverse gradient	Abs. difference
1.03293863915**149**e-01	1.03293863915**222**e-01	7.30e-14
2.3670708698791**7**e-01	2.3670708698791**2**e-01	5.00e-15
1.2868276197521**0**e-01	1.2868276197521**6**e-01	6.00e-15
1.256524426700**46**e-02	1.256524426700**53**e-02	7.00e-16
⋮	⋮	⋮
2.2469959328**0905**e-01	2.2469959328**1155**e-01	2.50e-13*

*Maximal difference

but also to derive relevant derivative regression tests from existing primal tests.

3.5.3 Gradient verification using TU Berlin stator blade parametrization

The same derivative verification procedure is repeated for the TUB stator test-case described in Sec. 2.1.2, since its parametrization uses slightly different OCCT modeling functionalities. The surface sensitivities with respect to one design parameter calculated with the *traceless* forward mode of AD and FD are shown in Fig. 3.4. They show mutual agreement, as confirmed by the quantitative comparison presented in Table 3.5.

Furthermore, the TUB blade surface sensitivities are also verified with the Taylor test (presented in Eq. (3.1)). The Taylor test is evaluated on a number of arbitrary

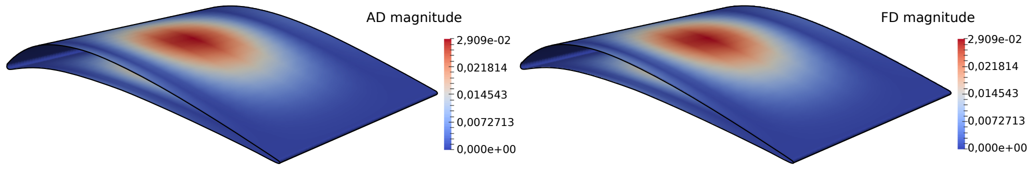


Figure 3.4: TUB stator blade geometric sensitivities evaluated with AD (left) and FD (right)

Table 3.5: AD and FD values comparison for several TUB stator blade point coordinates

AD value	FD value	Abs. difference
-3.96647**98813**e-07	-3.96647**15956**e-07	8.29e-13
6.45152**77484**e-07	6.45152**82006**e-07	4.52e-14
-8.242**1051717**e-06	-8.242**0958947**e-06	9.28e-12
-3.00799**39641**e-05	-3.00799**16336**e-05	2.33e-11
⋮	⋮	⋮
2.5489**612529**e-03	2.5489**583777**e-03	2.88e-09*

*Maximal difference

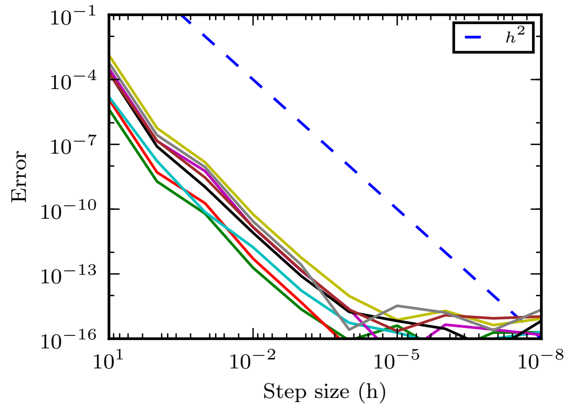


Figure 3.5: Taylor test overview for eight TUB blade surface point coordinates

surface point coordinates with a range of step sizes $h \in [10^1, 10^{-14}]$. The error plots (the left-hand side of Eq. (3.1)), presented in Fig. 3.5, follow the theoretical convergence rate of h^2. This behavior continues until $h = 10^{-4}$, where the errors reach machine precision.

The following step is to compare the derivatives evaluated with the *traceless* forward mode of AD against the *trace-based* differentiation variants. The results presented in Table 3.6 and Table 3.7 show only small discrepancies between the *traceless* and *trace-based* options, that are close to machine precision and therefore can be ignored.

Table 3.6: AD traceless-forward and AD trace-based forward gradient comparison for several TUB blade point coordinates

Traceless-forward gradient	Trace-based forward gradient	Abs. difference
1.2223149473367**50**e-05	1.2223149473367**28**e-05	2.20e-19
-2.6752511177383**95**e-06	-2.6752511177383**55**e-06	4.02e-20
-6.0582425223061**52**e-04	-6.0582425223061**83**e-04	3.04e-18
-4.4525814163383**72**e-07	-4.4525814163383**55**e-07	1.69e-21
⋮	⋮	⋮
2.523646741428**816**e-04	2.523646741428**715**e-04	1.01e-17*

*Maximal difference

Additionally to the original *trace-based* variants, the TUB stator derivatives are

Table 3.7: AD traceless-forward and AD trace-based reverse gradient comparison for several TUB blade point coordinates

Traceless-forward gradient	Trace-based reverse gradient	Abs. difference
1.222314947336**750**e-05	1.222314947336**682**e-05	6.81e-19
-2.675251117738**395**e-06	-2.675251117738**477**e-06	8.17e-20
-6.058242522306**152**e-04	-6.058242522306**268**e-04	1.16e-17
-4.452581416338**372**e-07	-4.452581416338**406**e-07	3.44e-21
⋮	⋮	⋮
1.76238488602**5022**e-03	1.76238488602**4947**e-03	7.50e-17*

*Maximal difference

Table 3.8: Activity analysis forward AD derivative verification on several TUB blade point coordinates

Original trace-based forward AD	Activity forward AD	Abs. difference
1.22231494733672**8**e-05	1.22231494733672**5**e-05	3.05e-20
-2.675251117738**355**e-06	-2.675251117738**024**e-06	3.31e-19
-6.0582425223061**83**e-04	-6.0582425223061**70**e-04	1.30e-18
-4.4525814163383**55**e-07	-4.4525814163383**28**e-07	2.70e-21
⋮	⋮	⋮
1.6985627017923**27**e-03	1.6985627017923**31**e-03	4.12e-18*

*Maximal difference

also calculated with the *activity analysis* feature to improve the performance of the differentiated OCCT sources, as explained in Sec. 3.3.2. Since the activity analysis imposes certain modifications to the *trace-based adouble* operators, the blade surface derivatives are verified against the original *adouble* class. The results presented in Table 3.8 and Table 3.9 show mutual agreement between the original ADOL-C sources and the activity analysis feature, with only very small discrepancies close to the machine precision. Therefore, both variants can be used equally.

To summarize, all ADOL-C differentiation options are successfully verified on the differentiated OCCT algorithms used for the parametric TUB stator blade model.

Table 3.9: Activity analysis reverse AD derivative verification on several TUB blade point coordinates

Original trace-based reverse AD	Activity reverse AD	Abs. difference
1.2223149473366**82**e-05	1.2223149473366**96**e-05	1.41e-19
-3.0255062517**50016**e-07	-3.0255062517**49999**e-07	1.75e-21
-8.0421849533121**62**e-06	-8.0421849533121**91**e-06	2.88e-20
-4.45258141633840**6**e-07	-4.45258141633840**4**e-07	2.12e-22
⋮	⋮	⋮
1.4317190660315**24**e-04	1.4317190660315**44**e-04	2.01e-18*

*Maximal difference

3.6 Performance tests of differentiated OCCT

3.6.1 Performance of U-bend CAD application

An optimization example is developed to measure performance of the differentiated OCCT sources by computing a large number of derivatives. The example is a typical, often executed task in CAD, the so-called 'surface fitting'. It is used to find a set of design parameters in parametrization P to match a certain geometry T, with the following optimization problem:

$$\min_{x\in\mathbb{R}^{96}} f(x) = \sum_{i=1}^{12000} \|P_i(x) - T_i\|^2 \quad \text{s.t.} \quad lb_j \leq x_j \leq ub_j, \quad j = 1, \ldots, 96. \tag{3.2}$$

where j is the number of the design variables, i is the index of one of 12,000 sampling points distributed uniformly over the surface, $P_i(x)$ and T_i are the points on the original and target (perturbed) surfaces respectively, lb_j and ub_j are lower and upper box limits for the j-th design parameter.

The verification proceeds as follows:

1. Construct two U-bends: original and perturbed, see Fig. 3.6.
2. Sample both final B-spline surfaces with 12,000 pairs of $(u_i,\ v_i)$ parametric coordinates. These parametric coordinates are later used the in B-spline algorithms to evaluate the corresponding 3-D points $P_i(x)$ and T_i.

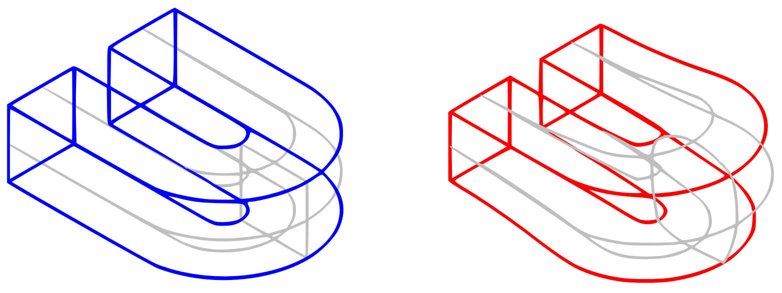

(a) Original U-bend shape (b) Perturbed U-bend shape

Figure 3.6: CAD optimization with two U-bends

3. Define an objective function as in Eq. 3.2.
4. Declare the original design parameters x as the independent variables of the system.
5. Compute gradients with ADOL-C.
6. Minimize the objective function $f(x)$ by using the limited-memory BFGS optimization algorithm with box constraints (L-BFGS-B) [ZBLN97].

Before analyzing the performance, the surface fitting example was executed with the *traceless* forward mode and the *trace-based* reverse mode of AD. The L-BFGS-B optimizer converged at the same point using the gradients provided by both differentiation modes. This step served as an additional effort to validate the derivatives.

The performance of the differentiated OCCT sources has been analyzed and compared to the original sources. The time required for a single geometry optimization iteration — the objective function value and the corresponding derivatives — has been measured. All three original differentiation modes of ADOL-C (the *traceless* forward mode and the *trace-based* forward and reverse modes) have been evaluated in the tests.

It is important to note that the measurements related to the *trace-based* variants include the time for generating the trace (*tracing*) on every iteration. Typically, when dealing with simple functions that do not involve code branching, the *tracing* is

Table 3.10: U-bend single optimization iteration timings for original and differentiated sources with number of directions $p = 1$ (*scalar* mode)

Computation mode	Avg. time [s]	Run-time ratio
Original sources (primal)	0.421	
Traceless forward AD	1.260	2.99
Trace-based forward AD	4.726	11.23
Trace-based reverse AD	4.660	11.07

Timings are averaged on 10 measurements. Tracing time: 3.9s, run-time ratio: 9.25.

Table 3.11: U-bend single optimization iteration timings with original and differentiated sources with number of directions $p = 96$ (*vector* mode)

Computation mode	Avg. time [s]	Run-time ratio
Original sources (primal)	0.421	
Traceless forward AD	12.597	29.92
Trace-based forward AD	18.132	43.07
Trace-based reverse AD	4.654	11.05

Timings are averaged on 10 measurements. Tracing time: 3.9s, run-time ratio: 9.25.

performed only once at the beginning, i.e. on the initial iteration. Later on, the same *trace* can be re-used to evaluate the function and its derivatives for different values of arguments x. This capability can significantly reduce the run-time. However, the OCCT geometry computation is a highly non-linear algorithm and a different computational path may be traversed at each design iteration, which disables the possibility to re-use the *trace*. For this reason, the *tracing* process is repeated for every iteration and therefore its required time is calculated in the measurements.

The results, i.e. quantitative comparisons of the average timings and run-time ratios (t_{ad}/t_{primal}), where the derivatives have been computed in one direction (*scalar* mode) as well as in 96 directions (*vector* mode), are shown in Table 3.10 and Table 3.11, respectively.

According to the theory [GW08], the run-time ratio between the derivative computation in the forward mode of AD and the *primal* evaluation should be in range $[1 + p, 1 + 1.5p]$, where p is the number of directions. Furthermore, the run-time ratio between the derivative computation in the reverse mode of AD and the *primal*

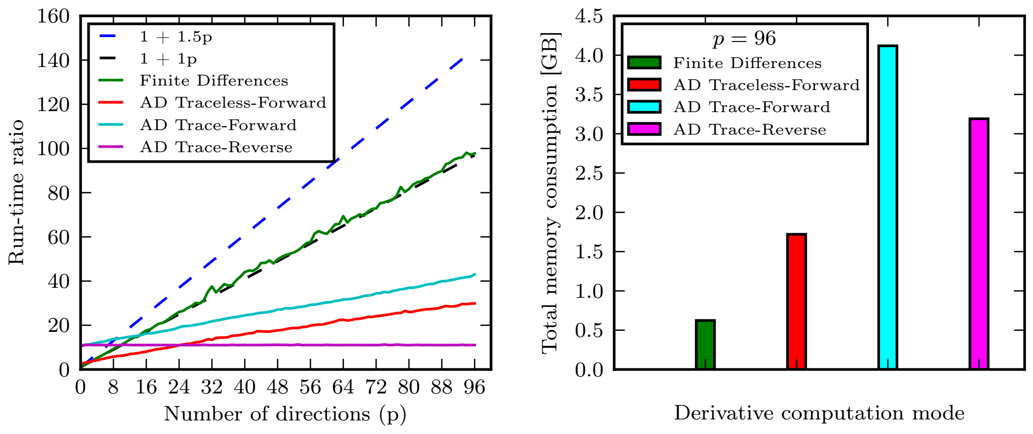


Figure 3.7: Summary of run-time ratios (left) and total memory requirements (right) for U-bend example

evaluation should be in range $[1 + 2q, 1.5 + 2.5q]$, where q is the number of adjoints. In the U-bend test example one has $q = 1$, therefore the expected range is [3, 4].

Comparing this with the results in Table 3.10 and Table 3.11, one can state that the differentiated OCCT sources yield run-time ratios that are even below the theoretical lower range boundaries for the forward mode. One reason for this is that the derivation of the theoretical bounds assumes a rather pessimistic runtime ratio for nonlinear univariate operations. Therefore, much better run-time ratios achieved with the *traceless* forward mode might be connected to the limited use of these costly operations by OCCT. Alternatively, one might also assume that compiler optimization could be a reason for this good run-time ratio. However, a similar effect, i.e. a better run-time ratio than predicted by the theory, is observable also for the *trace-based* forward mode where compiler optimization is not available in a comprehensive fashion due to the used overloading approach. Finally, the reverse mode of AD obtains a 63% improved efficiency in contrast to the *traceless* forward mode of AD for $p = 96$ design variables.

An overview of all run-time ratios evaluated in the range of 1 to 96 directions, together with the memory requirements with respect to the maximal number of directions, is shown in Fig. 3.7. Additionally to the AD run-time ratios, the run-time ratios for FD are also shown. To evaluate the memory requirements (with $p = 96$), the same U-bend application has been profiled with the profiling tool *Massif* — which is a part of the *Valgrind* tools for debugging and profiling [NS07]. The sources have been compiled with two compilers: *g++ v4.8.5* and *clang++ v3.7.0*, showing similar results. To summarize, AD requires more memory than the FD approach, but it performs significantly faster, especially when using many directions.

3.6.2 Performance of TU Berlin stator CAD application

Analogously to the U-bend CAD application, the surface fitting algorithm is developed for the TUB stator test-case. Here, the optimization problem is defined as follows:

$$\min_{x\in\mathbb{R}^{184}} f(x) = \sum_{i=1}^{1800} \|P_i(x) - T_i\|^2 \quad \text{s.t.} \quad lb_j \le x_j \le ub_j, \quad j = 1, \dots, 184. \tag{3.3}$$

where j is the number of the design variables, i is the index of one of 1,800 sampling points distributed uniformly over the surface, $P_i(x)$ and T_i are the points on the original and target (perturbed) surfaces respectively, lb_j and ub_j are the lower and upper box limits for the j-th design parameter.

The verification follows the workflow described in the previous section, using the original (P) and perturbed (T) geometries that are presented in Fig. 3.8. In comparison with Eq. (3.2), here the objective function depends on 184 design variables, while the number of sampling points is reduced to 1,800.

The time required for a single geometry optimization iteration — the objective function value and the corresponding derivatives — has been measured. Additionally to the *traceless* and original *trace-based* differentiation variants, the *activity analysis* feature is also considered in the measurements.

The results, i.e. quantitative comparisons of the average timings and the run-time ratios (t_{ad}/t_{primal}), where the derivatives have been computed in one direction (*scalar* mode) as well as in 184 directions (*vector* mode), are shown in Table 3.12 and Table 3.13, respectively.

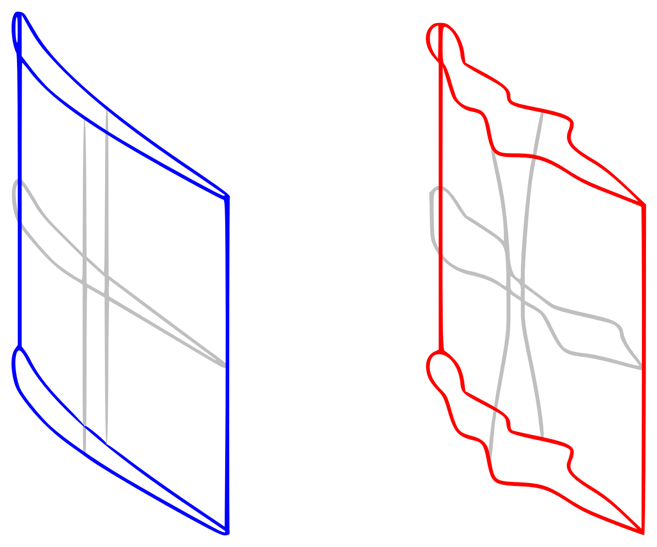


Figure 3.8: CAD optimization with two TUB stator blades

Table 3.12: TUB stator single optimization iteration timings for original and differentiated sources with number of directions $p = 1$ (*scalar* mode)

Computation mode	Avg. time [s]	Run-time ratio
Original sources (primal)	0.047	
Traceless forward AD	0.489	10.42
Trace-based forward AD	2.881	61.45
Trace-based forward *activity* AD	1.844	39.34
Trace-based reverse AD	2.921	62.30
Trace-based reverse *activity* AD	1.904	40.62

Timings are averaged on 10 measurements.

Table 3.13: TUB stator single optimization iteration timings with original and differentiated sources with number of directions $p = 184$ (*vector* mode)

Computation mode	Avg. time [s]	Run-time ratio
Original sources (primal)	0.047	
Traceless forward AD	7.134	152.16
Trace-based forward AD	10.776	229.84
Trace-based forward *activity* AD	5.260	112.19
Trace-based reverse AD	2.897	61.79
Trace-based reverse *activity* AD	2.244	47.87

Timings are averaged on 10 measurements.

For this test-case, the results are not as good as the ones presented in Sec. 3.6.1. That is, the *scalar* run-time ratios for forward mode of AD (presented in Table 3.12) are much higher than the upper theoretical limit of 2.5. The reason for such a slow-down could be that the compiler is not able to optimize complex blade construction algorithms as good as it does for the U-bend parametrization code. However, this pays off when using forward vector mode of AD with the complete design space of 184 parameters, where the differentiated sources yield a run-time ratio that is even better than the lower theoretical expectation.

Considering the reverse mode of AD, its performance is significantly slower than the upper theoretical boundary of 4. Nevertheless, the results presented in Table 3.13 show that the original reverse mode of AD obtains a 59.4% improved efficiency in contrast to the *traceless* forward mode of AD. Once the activity analysis is enabled, it improves the computation efficiency by additional 22.5% in contrast to the original reverse mode of AD.

An overview of all run-time ratios evaluated in the range of 1 to 184 directions is shown in Fig. 3.9. Additionally to the AD run-time ratios, the run-time ratios for FD are also shown. As one can notice, the best performance for this example is achieved when computing the gradients with respect to the complete design space using the reverse mode of AD with enabled *activity analysis*.

The same TUB stator application is profiled with the *Massif* profiling tool to evaluate the maximal memory requirements of the differentiated sources when computing

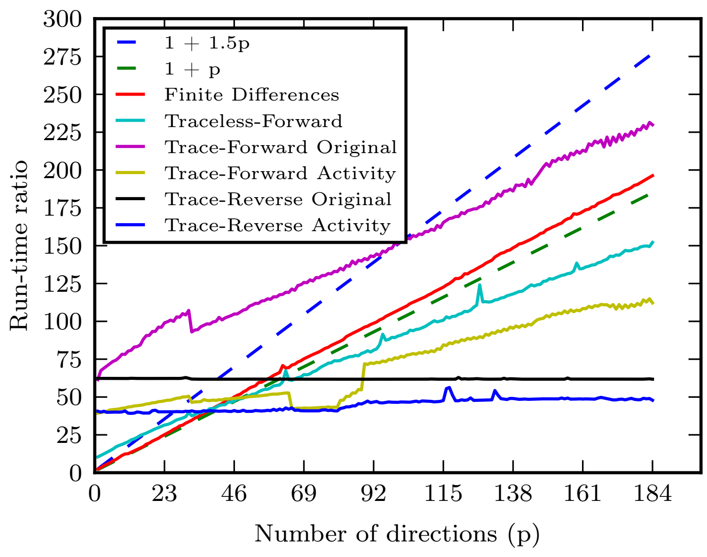


Figure 3.9: Summary of run-time ratios for TUB stator example

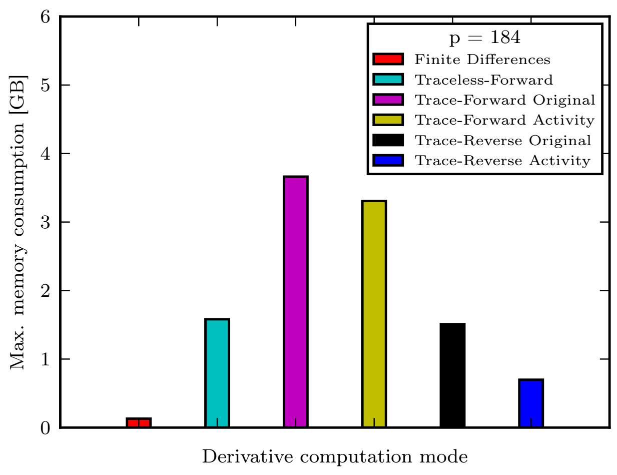


Figure 3.10: Summary of memory requirements for TUB stator example

Table 3.14: ADOL-C trace buffers

	Original ADOL-C		Activity analysis	
Buffer type	Size	Memory [MB]	Size	Memory [MB]
Operation	103900078	99.09	38079347	36.32
Location	163085895	622.12	71488624	272.71
Value	3643416	27.80	9810737	74.85
Taylor	96946436	739.64	35656343	272.04

the derivatives in 184 directions, as shown in Fig. 3.10. The lowest memory consumption of the differentiated sources is achieved when using the reverse mode of AD with activity analysis.

In addition to the memory profiling with *Massif*, the ADOL-C *trace* buffers' sizes are examined and written in Table 3.14. To retrieve these numbers, one calls the function of ADOL-C named `printTapeStats`. In comparison to the original ADOL-C *trace-based* variant, the activity analysis requires much less resources, i.e. the size of *operation*, *location* and *Taylor* buffers is significantly reduced. Only the *value* buffer is increased, but this is exactly the expected behavior. As explained in Sec. 3.3.2, the activity analysis discovers all *adoubles* that are constants and stores only their values in the *value* buffer (hence reducing their footprint in other buffers). Therefore, when evaluating the *trace*, all constant *adoubles* are treated in the same way as regular *double* variables. When using the activity analysis for this particular case, the total memory consumed by the *trace* is reduced from 1488.65 MB to 655.92 MB, i.e. by 56%. This also justifies its faster evaluation than the original *trace-based* code (shown in Fig. 3.9).

3.7 Summary

The AD of OCCT was achieved by integrating the AD tool ADOL-C into its sources. During the differentiation, a large number of compile- and run-time errors had to be resolved, which resulted in a considerable amount of code modification. Both *traceless* and *trace-based* ADOL-C features were integrated into OCCT, leading to two different versions of the differentiated sources.

After the differentiation, the *primal* functionality was validated with the automated testing system of OCCT. Next, the derivatives computed with AD were successfully verified against FD, using the parametric models of U-bend and TUB stator.

The same test-cases have been employed to measure performance of the differentiated sources. Regarding the U-bend test-case, the reverse mode of AD obtains a reduction of 63% in run-time contrary to the *traceless* forward mode of AD for $p = 96$ design parameters. Regarding the TUB stator test-case, the reverse mode of AD yields a reduction of 59.4% in contrast to the *traceless* forward mode of AD for $p = 184$ design parameters. For this test-case, the *activity analysis* feature of ADOL-C has been employed, which resulted in a faster execution and reduced memory requirements of the reverse differentiated OCCT sources.

4
Aerodynamic shape optimization with differentiated OCCT

4.1 Mathematical formulation of CAD-based optimization with adjoint method

Aerodynamic performance of a given CAD model is usually described with a scalar objective function J such as drag, lift or total pressure loss. Here, the optimization problem can be stated as follows:

$$\min_{\alpha} \; J(U(X(\alpha)), X(\alpha)) \,, \tag{4.1}$$

$$R(U(X(\alpha)), X(\alpha)) = 0 \,. \tag{4.2}$$

In our context Eq. (4.2) denotes the system of steady-state Reynolds-Averaged Navier-Stokes equations (RANS), where the flow residual R is driven to zero by an iterative solution procedure. Both the objective function J and the residual R are functions of the flow state variable U and the mesh coordinates X, which successively depend on CAD design parameters α. Differentiating the system Eq. (4.1)-(4.2) with respect to α yields:

$$\frac{dJ}{d\alpha} = \left[\frac{\partial J}{\partial U}\frac{\partial U}{\partial X} + \frac{\partial J}{\partial X}\right]\frac{\partial X}{\partial \alpha} \,. \tag{4.3}$$

$$\frac{dR}{d\alpha} = \left[\frac{\partial R}{\partial U}\frac{\partial U}{\partial X} + \frac{\partial R}{\partial X}\right]\frac{\partial X}{\partial \alpha} = 0 \,. \tag{4.4}$$

The latter can be written in a form of a linear system:

$$A\,u = b \,, \tag{4.5}$$

where

$$A = \frac{\partial R}{\partial U}, \quad u = \frac{\partial U}{\partial X} \quad \text{and} \quad b = -\frac{\partial R}{\partial X} \, .$$

The solution u is obtained by solving the linear system presented in Eq. (4.5). Later, it can be used to compute the objective function gradient:

$$\frac{dJ}{d\alpha} = \left[g \, u + \frac{\partial J}{\partial X} \right] \frac{\partial X}{\partial \alpha} \, , \tag{4.6}$$

where

$$g = \frac{\partial J}{\partial U} \, .$$

The described workflow represents the so-called tangent-linear approach (or forward mode) to compute the gradient [XJM13]. However, the right-hand side of the linear system in Eq. (4.5), as well as the solution u depends on X. Therefore, the solution u needs to be recomputed for each variable X solving Eq. (4.5) [CJM11]. This causes the gradient computational cost to scale linearly with respect to the number of mesh variables, which is expensive for industrial applications where the mesh size is typically quantified in thousands or more likely millions when dealing with high-fidelity simulations.

A more efficient approach to compute the gradient is the adjoint method [GDMP03, YMJC11, XJM13, XRMM15]. With this approach, one computes the transposed gradient which allows regrouping of the terms in Eq. (4.6):

$$\begin{aligned} \frac{dJ}{d\alpha}^{\top} &= \frac{\partial X}{\partial \alpha}^{\top} \left[(g \, A^{-1} \, b)^{\top} + \frac{\partial J}{\partial X}^{\top} \right] \\ &= \frac{\partial X}{\partial \alpha}^{\top} \left[b^{\top} \, A^{-\top} \, g^{\top} + \frac{\partial J}{\partial X}^{\top} \right] . \end{aligned} \tag{4.7}$$

To avoid computing the inverse of the matrix A in Eq. (4.7), one applies the following expression:

$$A^{-\top} \, g^{\top} \equiv \nu \, . \tag{4.8}$$

Multiplying Eq. (4.8) from the left side by $A^{\top}$, one retrieves a linear system:

$$A^\top \nu = g^\top \;\equiv\; \frac{\partial R}{\partial U}^\top \nu = \frac{\partial J}{\partial U}^\top \,. \tag{4.9}$$

Eq. (4.9) is referred to as the adjoint equation. The adjoint solution ν can be utilized in Eq. (4.6) to compute the gradient by applying the following equivalence:

$$g\,u = (g^\top)^\top u = (A^\top \nu)^\top u = \nu^\top A\,u = \nu^\top b\,. \tag{4.10}$$

Using this equivalence statement (Eq. (4.10)), the total gradient in Eq. (4.6) is reformulated as follows:

$$\frac{dJ}{d\alpha} = \left[\nu^\top b + \frac{\partial J}{\partial X}\right]\frac{\partial X}{\partial \alpha}\,. \tag{4.11}$$

Another approach to derive the adjoint equation is by means of the Lagrange calculus, where ν is the so-called Lagrange multiplier. A comprehensive explanation of this method can be found in [Gun02].

The benefit of using the adjoint approach is that the linear system presented in Eq. (4.9) is independent of the mesh points X contrary to the direct approach defined in Eq. (4.5). In the case of a scalar objective function J, the adjoint linear system requires only a single evaluation. The adjoint sensitivity defined in Eq. (4.11) still depends on the source term b which depends on X, however b is not involved any more in an expensive linear system solution process and the term $\nu^\top b$ can be efficiently computed using the reverse mode of AD. That is the reason why the adjoint method is advantageous for applications such as CFD that typically have a few objective functions (outputs) and many design variables (inputs).

Typically, the parametrization code is decoupled from the CFD tool, as it is the case in this study. Therefore, the adjoint formulation explained above computes only the volumetric sensitivity dJ/dX (instead of $dJ/d\alpha$). Applying the chain rule, the total sensitivity can be assembled as follows:

$$\frac{dJ}{d\alpha} = \frac{dJ}{dX}\frac{dX}{dX_S}\frac{dX_S}{d\alpha}\,, \tag{4.12}$$

where X_S represents the mesh coordinates of the design surface.

The term dX/dX_S describes the relation between volume and surface mesh perturbation, i.e. how the computational grid is affected by the change in the design surface. One possibility to compute it is to differentiate a mesh generation algorithm. This can be useful if the mesh generation is repeated at every design update during the optimization. A more efficient approach is to utilize mesh morphing techniques that deform volume mesh nodes accordingly to the design surface perturbation. The common mesh deformation algorithms are: linear elasticity [Dwi09], spring analogy [Blo00] and inverse-distance weighting [WB09].

By differentiating the mesh morphing algorithm, one can combine the retrieved sensitivity with the adjoint volumetric sensitivity to assemble the adjoint surface sensitivity:

$$\frac{dJ}{dX_S} = \frac{dJ}{dX}\frac{dX}{dX_S} \; . \tag{4.13}$$

By employing the adjoint surface sensitivity in Eq. (4.12), the total gradient is rewritten as follows:

$$\frac{dJ}{d\alpha} = \frac{dJ}{dX_S}\frac{dX_S}{d\alpha} \; . \tag{4.14}$$

The first term in Eq. (4.14), the so-called *CFD sensitivity*, corresponds to the sensitivity of the objective function with respect to displacements of the surface grid points X_S. The second term in Eq. (4.14), the so-called *CAD sensitivity*, represents the geometric derivative of the surface grid points X_S with respect to design parameters α. After obtaining the total gradient, one can use it in gradient-based optimization loop:

$$\alpha^{(n+1)} = A\big(\alpha^{(n)}, \frac{dJ}{d\alpha}(\alpha^{(n)})\big) \; , \tag{4.15}$$

with A as an iterative optimization algorithm.

4.2 STAMPS flow solver

The CFD optimization group at Queen Mary University of London develops the open-source discrete adjoint CFD solver STAMPS [MGX+16, MHM18]. This CFD solver employs a typical edge-based vertex-centered finite volume formulation to solve the primal system of equations (defined in Eq. (4.2)) on unstructured grids.

Its source code is developed in the Fortran90 language. This enables the application of the source-transformation AD — in particular the AD tool Tapenade is used — in order to produce the adjoint code. Contrary to the operator-overloading concept, the source-transformation AD enables a lower memory footprint of the differentiated sources, which is extremely beneficial for applications such as CFD. As reported in [MHM18], the adjoint solve requires only 15% more memory comparing to the primal evaluation on the TUB stator test-case.

4.3 Gradient-based shape optimization framework

As mentioned in Sec. 1.4.1, Orest Mykhaskiv (QMUL) developed the gradient-based shape optimization framework used in this work as illustrated in Fig. 4.1. The author contributed to the *CAD sensitivity* evaluation part and its integration into the optimization framework. The framework development is based on the object-oriented programming paradigm, such that different tools or optimization methods could be connected to it. Therefore, the workflow presented in Fig. 4.1 represents the general CAD-based optimization loop.

The optimization process is described as follows. The baseline CAD model is constructed using the provided initial set of design parameters α. The CAD geometry is exported to a STEP file which is used by a mesh generator to create the computational grid for the CFD tool. The mesh points that belong to the design surface are projected to the B-spline surface of the model to find corresponding (u, v) parametric coordinates. Typically, the CAD model has several faces (B-spline surfaces). Therefore, in addition to the parametric coordinates, one stores the face index where the certain point is projected. This information is later required to compute the CAD sensitivity. Afterwards, one executes the CFD tool (in this case the STAMPS flow solver) that solves the primal and the adjoint equations. As the output, one obtains the objective function value and the adjoint volumetric sensitivity. The following step is to map the volumetric sensitivity to the adjoint surface sensitivity. For this purpose, STAMPS provides the mesh morphing capability that is also algorithmi-

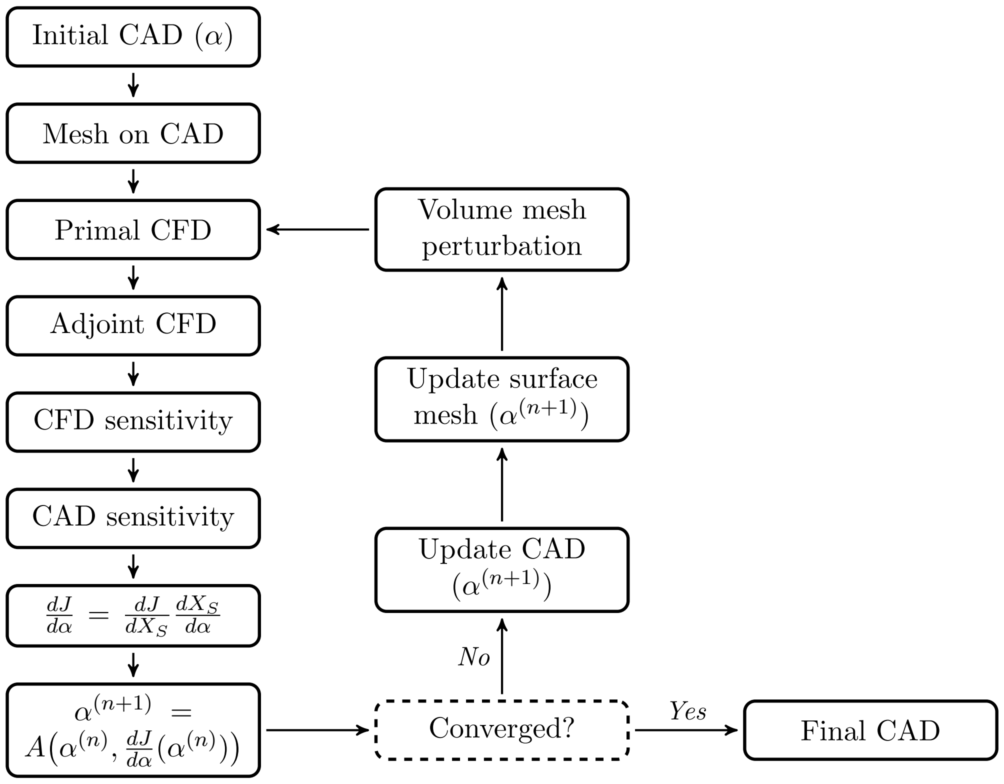

Initial CAD (α)
Mesh on CAD
Primal CFD
Adjoint CFD
CFD sensitivity
CAD sensitivity
$\frac{dJ}{d\alpha} = \frac{dJ}{dX_S}\frac{dX_S}{d\alpha}$
$\alpha^{(n+1)} = A\big(\alpha^{(n)}, \frac{dJ}{d\alpha}(\alpha^{(n)})\big)$
Converged?
Yes
Final CAD
No
Update CAD ($\alpha^{(n+1)}$)
Update surface mesh ($\alpha^{(n+1)}$)
Volume mesh perturbation

Figure 4.1: Gradient-based optimization workflow

cally differentiated. In this work, the inverse-distance weighting method is used. Once the CFD sensitivity is assembled, there are two possibilities to compute the total gradient $dJ/d\alpha$:

1. Compute the CAD sensitivity by using the differentiated OCCT kernel in the *traceless* forward mode of ADOL-C and couple both sensitivities at the end.
2. Use the differentiated OCCT kernel in the *trace-based* reverse mode of ADOL-C, thus having a full reverse mode differentiation of the complete differentiated design chain to compute the desired gradient.

When choosing the first approach, the CAD and CFD parts can be executed in parallel since their sensitivity computation is independent of each other. In this case, the total gradient is calculated at the end just by multiplying the two sensitivities. On the other hand, the second approach requires that the CFD sensitivity is computed first and then propagated as a derivative seed vector to the reverse differentiated OCCT. Basically, the derivative seed vector is given to an ADOL-C driver routine that evaluates the *trace* by using the reverse mode of AD and gives the total gradient as the output.

The total gradient is provided to an optimizer to update the values of the design parameters α. There are three optimization methods used in this work: steepest-descent (also referred to as gradient-descent), L-BFGS-B and SLSQP (*Sequential Least SQuares Programming*).

The next step is to update the CAD geometry and the surface mesh. The surface displacements are propagated into the interior domain by the mesh morphing algorithm. After the volume mesh perturbation is completed, the CFD tool is executed again and the complete process is repeated until the convergence is achieved. As already stated, the benefit of using the CAD-based optimization methods is that one retrieves the optimal shape in the CAD-specific format.

4.4 U-bend optimization results

The gradient-based optimization methodology elaborated in the previous section is applied to the U-bend test-case whose parametrization is described in Sec. 2.1.1. The complete case description can be found in [Ver] with the corresponding related research [VCB+13, CVB+13]. The objective function to minimize is the total pressure loss between the inlet and the outlet pipe. The U-bend is a rather challenging case for the CFD solver, since the flow completely changes its direction after the U-part.

The computational grid for the CFD tool was created in ANSYS ICEM CFD and has approximately 170,000 cells. The CFD computations are performed by the discrete adjoint solver STAMPS.

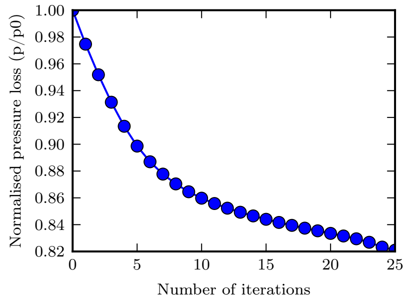
STAMPS includes several methods for robust volume mesh movement. As mentioned previously, the inverse-distance weighting method is used in this work. Therefore, at each optimization step, the surface mesh is recalculated on the updated CAD geometry given by the OCCT kernel and then the resulting surface displacements are propagated into the interior domain of the volume mesh.

Two optimization methods were applied: L-BFGS-B and steepest descent (S-D). The efficient L-BFGS-B algorithm was taking too large steps in the initial iteration that generated a U-bend shape not suitable for the volume mesh movement and therefore breaking the optimization process. On the other hand, the S-D method is considered to be inferior in performance, however its explicit control of the design steps made parametrization updates and corresponding volume mesh movement more robust and hence was finally used to drive the optimization.

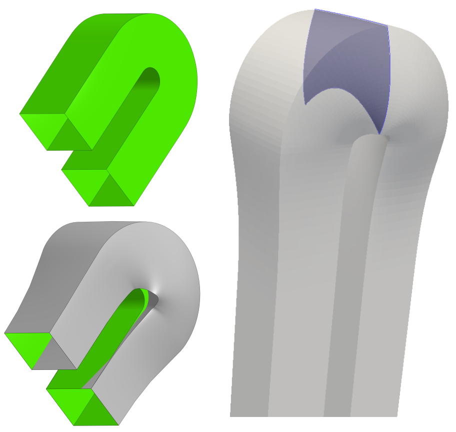
The optimization history, yielding 18% improvement, is shown in Fig. 4.2 together with a comparison between the baseline and the optimal geometry. The design iterations broke down at this stage because the mesh quality of the deformed mesh became too poor for the flow solver to converge. As there is an exact CAD description of the U-bend geometry, one could re-mesh and run further optimization steps. Nevertheless, re-meshing is currently a manual step and not included in the automated design algorithm.

The dominant flow phenomenon in the baseline geometry is the separation after the U-part and downstream in the outlet leg, which significantly adds to the total pressure loss. As shown in Fig. 4.3, the CAD-based optimization managed to reduce its size, resulting in much lower loss of the total pressure. This is mainly due to the increase in inner radius of the bend and the cross-sectional area of the U-part.

[7]Picture 4.2 (b) provided by Orest Mykhaskiv



(a) Objective function



(b) Initial (green) and optimized (gray) CAD model

Figure 4.2: U-bend optimization results[7]

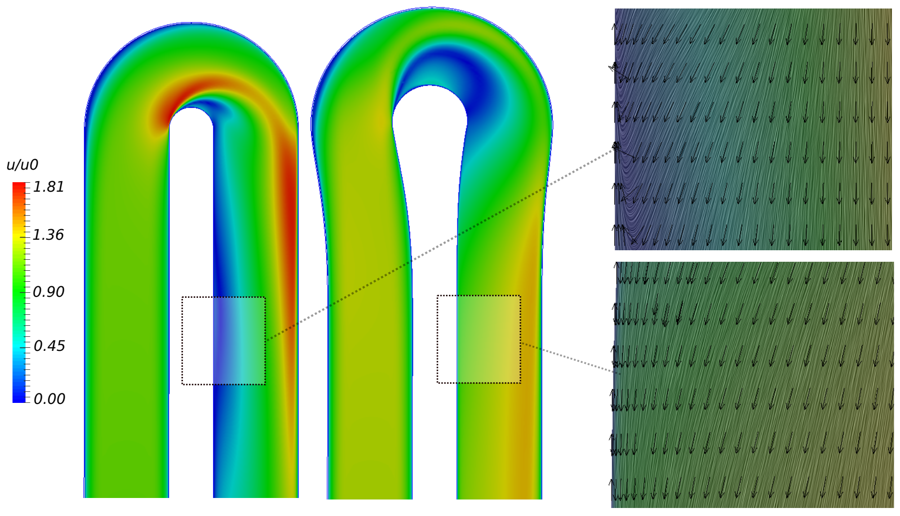


Figure 4.3: Left: Baseline and optimized mid-span velocity magnitude; Right: Flow streamlines in the outlet leg of U-bend[8]

4.5 TU Berlin stator optimization results

The gradient-based optimization framework (described in Sec. 4.3) is also utilized to perform aerodynamic shape optimization of the TUB compressor stator blade elaborated in Sec. 2.1.2. The task of the stator is to turn the incoming flow with a whirl angle of 42° into the axial direction with a minimal total pressure loss. Therefore, the objective function to minimize is the total pressure loss between the inlet and the outlet CFD domain. The second objective is to minimize the flow angle deviation from the axial direction at the outlet, however it is not considered in this optimization.

The optimization was performed in two stages, with low-fidelity and high-fidelity CFD simulations performed by STAMPS. The computational grids were created in ANSYS ICEM CFD and have approximately 20,000 and 400,000 cells for the low-fidelity and the high-fidelity CFD simulation, respectively.

There are several geometric/manufacturing constraints that have to be satisfied during the optimization:

1. G2 continuity is imposed on every blade section to ensure a smooth blade surface. In the context of a single blade section, the G2 continuity or *curvature continuity* means that the suction and the pressure B-spline curves have a common point (e.g. leading edge), their tangent vectors lie along the same direction and the curvature change at that point is equal for both curves. In other words, when two curves join at a certain point, their angles are identical as well as their radii. The G2 continuity is respected both for the leading and the trailing edge (LE and TE) and guaranteed by the parametrization.
2. The axial chord of the blade has to be kept constant: the axial-coordinate of the last camber-line control point is set to be equal to the axial-coordinate of the first control point plus the constant axial chord value.
3. Minimum blade thickness distribution.
4. Minimum LE and TE radii.
5. Minimum thickness near the hub and the shroud to accommodate four mount-

[8]Picture provided by Orest Mykhaskiv

ing bolts (cylinders) that serve to mount the blade to its casing.

The first two requirements are embedded in the parametrization. The third and the fourth constraint are handled by the optimization algorithm. For this purpose, the L-BFGS-B algorithm is used as the optimizer, since it allows to impose bounding ranges (lower and upper limits) for the CAD parameters α. The last prerequisite that the optimal blade has to accommodate four cylinders is rather challenging to fulfill, however a suitable solution is elaborated in Chapter 5.

The CAD-based optimization with the low-fidelity CFD simulation yields 13.72% reduction of the objective function after eleven iterations, as shown in Fig. 4.4. A comparison between the baseline and the optimal geometry is presented in Fig. 4.5. As one can notice, the optimization reduces LE and TE radii as well as the blade thickness, while respecting the predefined minimum values.

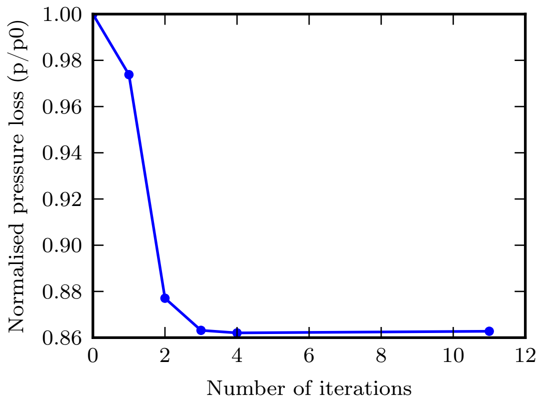


Figure 4.4: TUB stator optimization history (low-fidelity CFD simulation)

The optimization with the high-fidelity CFD simulation converged after 23 iterations and yields 6.7% improvement of the objective function, as presented in Fig. 4.6. As shown in Fig. 4.7, the optimal blade is a bit thinner comparing to the original one with slightly bended trailing edge at the mid-span. It achieves a signification reduction of the flow separation at the shroud (tip).

[9]Picture provided by Orest Mykhaskiv
[10]Picture provided by Orest Mykhaskiv

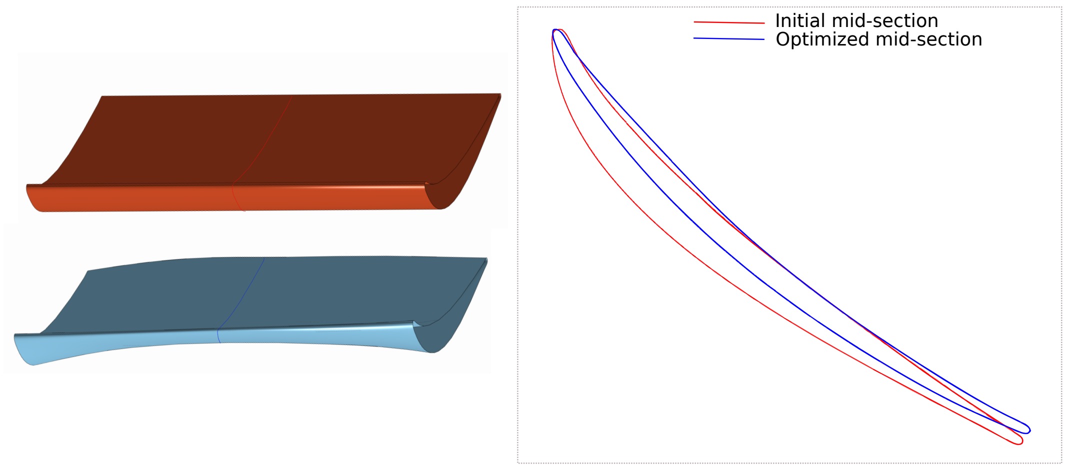


Figure 4.5: Left: Baseline (red) and optimal (blue) TUB blade geometry (low-fidelity CFD simulation); Right: Comparison of mid-sections[9]

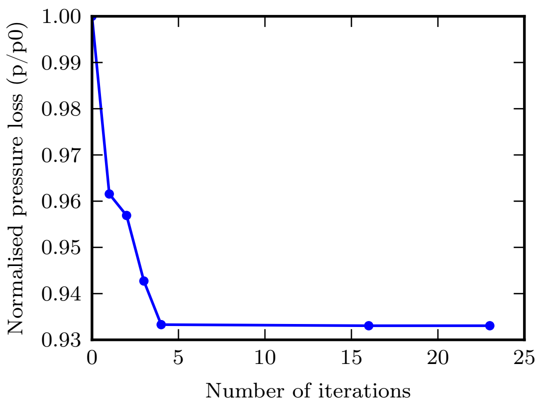


Figure 4.6: TUB stator optimization history (high-fidelity CFD simulation)

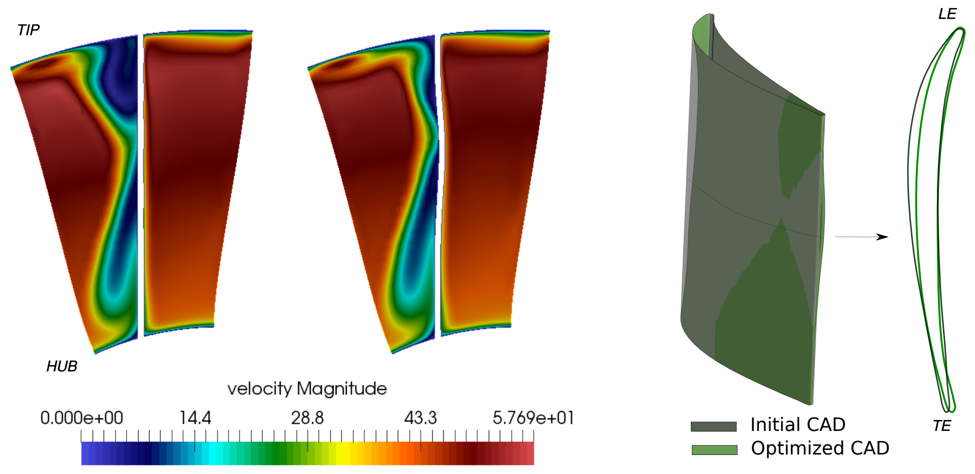


Figure 4.7: Left: Baseline and optimal velocity distribution at TE (high-fidelity CFD simulation); Right: Comparison of baseline and optimal geometry and mid-sections[10]

4.6 Summary

In this chapter it is described how to formulate the CAD-based optimization with the adjoint CFD method. Furthermore, the gradient-based shape optimization workflow developed in this work is presented. In particular, the differentiated OCCT kernel is coupled with the discrete adjoint CFD solver STAMPS, that is also produced by AD. This differentiated design chain is demonstrated on the aerodynamic shape optimization of the U-bend and TUB stator parametric models, in order to minimize the total pressure losses. The following optimization methods were used: S-D and L-BFGS-B. The U-bend optimization with S-D converged after 25 iterations yielding 18% reduction of the objective. Regarding the TUB stator test-case, the optimization was performed in two stages, both using L-BFGS-B as the optimizer. First, the optimization with the low-fidelity CFD simulation converged after eleven iterations decreasing the total pressure loss between the inlet and the outlet by 13.72%. Second, the optimization with the high-fidelity CFD simulation converged after 23 iterations and yields 6.7% improvement of the objective function.

5 TU Berlin stator optimization with assembly constraints

Assembly CAD models are complex structures defined in a CAD system with the purpose to describe how a set of individual 3-D components are put together in order to construct a whole product. They are crucial part of a product's design process since they define relations between all components in the product (i.e. how the individual components are connected), as well as their position and volume to be occupied. These CAD files also serve as inputs to manufacture the final product with CAM.

Due to their complexity, shape optimization in industrial workflows is typically performed on the extracted individual components of the product (i.e. on a component-by-component basis) rather than considering the highly-detailed assembly CAD models. Once an individual component is optimized, its CAD model needs to be re-inserted to the assembly CAD model to ensure the fitting with the surrounding components.

A collision arises when the optimized model occupies the same physical space as one of the neighboring elements. In this case, the optimized shape has to be modified until all collisions disappear. However, any post-optimization modifications impose additional costs as they require further analysis (e.g. with CAE tools) and may impair the optimal solution. Therefore, it is important to respect assembly constraints during the shape optimization process.

Another type of assembly constraint is related to fasteners such as bolts, screws and nuts, that serve to connect various components in the product's assembly. Therefore, one has to ensure the fitting of the fasteners in the optimal shape as well. This kind

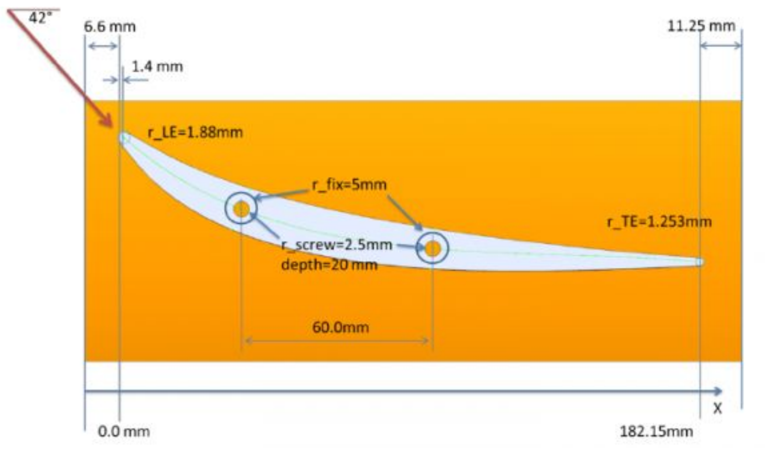


Figure 5.1: Top-view on the stator vane[11]

of requirement is defined for the TUB stator assembly.

As described in Sec. 4.5, the TUB stator test-case has several manufacturing constraints to be respected during the aerodynamic shape optimization. While the majority of them are handled by the parametrization itself and by imposing lower and upper bounds on the design space, the fitting of the fasteners in the optimal blade required a more sophisticated solution elaborated in this chapter.

In particular, there are four mounting bolts (cylinders) that are pierced inside the volume of the blade and they serve to mount the blade to the stator assembly. As shown in Fig. 5.1, a cylinder has a radius of 5 mm and a depth of 20 mm to allow cutting the screw thread at both hub (inner part of the stator) and shroud (outer part of the stator). There are two cylinders per side of the blade. Their position is arbitrary, however the axial distance between the two cylinders (in correspondence to the hub or the shroud) has to be at least 60 mm.

The proposed approaches to tackle the assembly fit constraint during the optimization are explained in the following section. Salvatore Auriemma developed the method to measure the geometric constraints using the OCCT kernel, while the author contributed to gradient computation and integration of constraints into the shape optimization framework.

[11]Picture taken from the TUB test-case description [MV]

5.1 Implementation of stator assembly constraints

5.1.1 Intersection approach

Let us consider one cylinder that is placed inside the volume of a perturbed blade such that the constraint is already violated, as shown in Fig. 5.2. One possibility to define the constraint is to measure the intersection between the blade and the cylinder. For this purpose, the OCCT kernel provides the *BRepAlgoAPI_Section* class which belongs to Boolean operations. Given two shapes, it calculates vertices and edges that are the result of a conjunction between those shapes.

The next step is to analyze the collections of resulting edges with the *ShapeAnalysis_FreeBounds* class. It provides a method for connecting edges to wires. The wire is a sequence of edges connected by their vertices. It can happen that there are several wires found by the algorithm, e.g. if the cylinder intersects the blade both on the suction and the pressure side.

There are open and closed wires. For this reason, one has to loop over all wires provided by the *ShapeAnalysis_FreeBounds* algorithm in order to identify open wires. The open wires have to be manually closed, i.e. by creating a straight line between the first and the last vertex of the wire and joining the newly created edge with the original wire.

Once ensured that all wires are closed, one can calculate the area bounded by every wire. This can be achieved with the method *ContourArea* implemented in the *ShapeAnalysis* class. Finally, one can sum over all evaluated areas to retrieve the total intersection area that represents the constraint.

To validate the proposed approach for implementing the assembly constraint, an optimization problem is defined to find the position of the cylinder where the objective function (intersection area) is minimal. That is, the blade parameters are fixed, while the center coordinates of the cylinder (x, y) are defined as the independent variables of the system (the z-coordinate is constant). The initial position of the cylinder is presented in Fig. 5.2.

The L-BFGS-B algorithm is used as the optimizer, where the lower and the upper bounds of the (x, y) coordinates are defined such that the cylinder does not end up

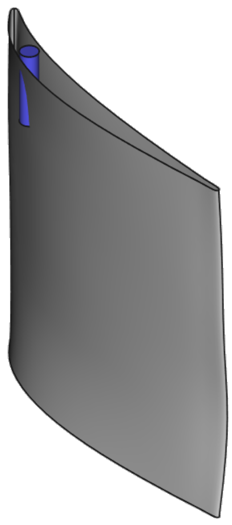


Figure 5.2: Example of constraint violation between TUB blade and one cylinder

outside of the blade. Although in this case the objective function would be zero, the solution is not valid. Therefore, a movement of the cylinder is limited to a few millimeters in x- and y-direction. The derivatives of the objective function with respect to (x, y) variables are calculated using the reverse mode of AD.

However, the optimization did not converge. The first assumption was that the gradient information provided by the differentiated OCCT could be wrong, so the same optimization was repeated with the original OCCT sources and the FD approach to compute the derivatives. Still, the result was the same as with the differentiated OCCT. This required further investigation of the intersection algorithm.

The first step was to evaluate the total intersection area (the primal) in many points in order to generate a 3-D plot. That is, the center coordinates of the cylinder (x, y) were moved in the neighborhood defined with the following range: $[x - 2.5,\ x + 2.5]$ and $[y - 2.5,\ y + 2.5]$, using a step size that produced 10^4 samples for the plot presented in Fig. 5.3. The red dot in the middle represents the initial position of the cylinder (as in Fig. 5.2). As one can notice, the objective function is very noisy and contains a large number of spikes, making it unstable for the gradient-based optimization. The actual number of spikes is unknown as it may differ with coarser or finer step size for (x, y) coordinates.

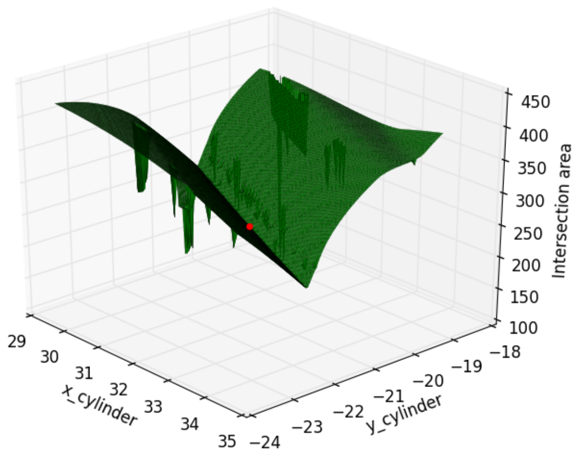


Figure 5.3: Total intersection area with respect to (x, y) coordinates of the cylinder

One of such locations where a spike arises was extracted to visualize the blade and the cylinder, as well as all edges found by the *BRepAlgoAPI_Section* algorithm. This example is presented in Fig. 5.4. As one can notice, the intersection edge on the 'B' side is not computed by the intersection algorithm, therefore causing the drop in the objective function.

The problem was reported to Open CASCADE engineers to find an appropriate solution. However, the conclusion is that the OCCT intersection algorithm works poorly in the tangential cases and that is the reason why it failed to find all intersection edges. This is a general issue reported by Barnhill et al. [BFJP87], where they stated that one class of examples difficult for surface-surface intersection algorithms are surfaces which are tangent to each other. Due to this obstacle, the intersection approach to define assembly constraint was rejected. The working alternative is elaborated in the following section.

[12]Picture provided by Salvatore Auriemma

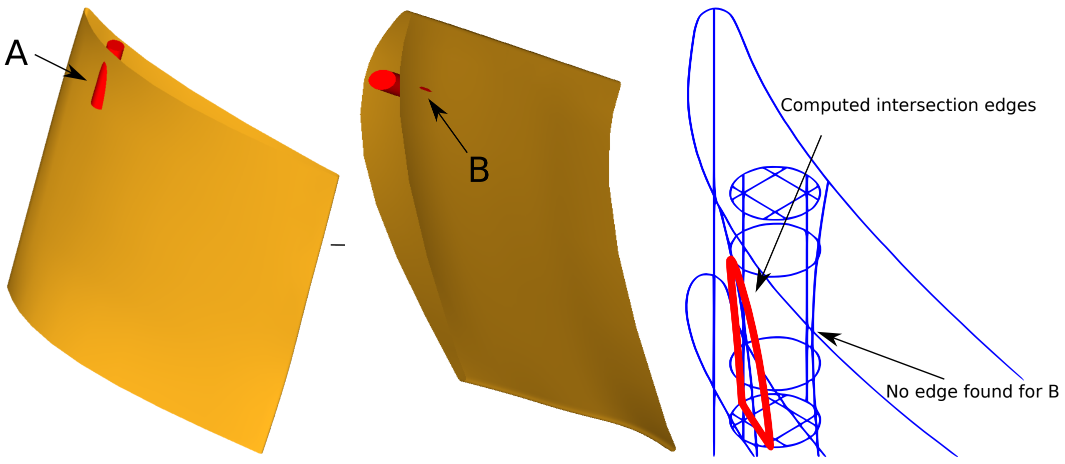


Figure 5.4: *BRepAlgoAPI_Section* algorithm fails to compute all intersection edges[12]

5.1.2 Interference detection approach based on distance between shapes

Another possibility to treat the assembly constraints is the interference detection that is available in certain commercial CAD systems. Agarwal et al. [ARA18a] proposed a CAD-based shape optimization framework that exploits this capability. They use the interference detection tool in CATIA V5 which returns the minimum distance required to translate a component to avoid collision. In the case there is no collision between the observed shapes, it returns the allowed (clearance) distance. These values serve to define inequality constraints of the optimization. Furthermore, Agarwal et al. use FD to compute the gradient of the constraints with respect to design parameters of a CAD model. They have successfully optimized an automotive ventilation duct, the so-called S-bend, while respecting adjacent components in the assembly. The similar concept is explained in this section to implement the TUB bolts constraint.

To describe the algorithm, let us consider a relation between one cylinder and the blade. First, a point cloud is distributed uniformly on the cylindrical surface, as shown in Fig. 5.5. Second, each point is projected to the suction and the pressure side of the blade in order to evaluate all distances between the mesh and both sides of the blade. Finally, only two distances are selected to define two inequality

[13]Picture provided by Salvatore Auriemma

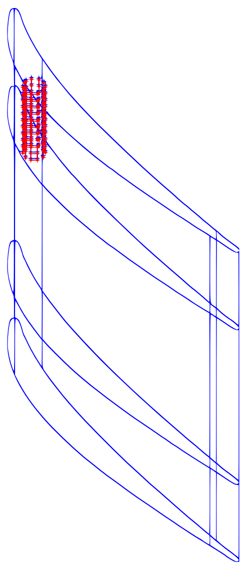


Figure 5.5: Example of point cloud for one cylinder[13]

constraints: $C_1 \geq 0$ and $C_2 \geq 0$, as illustrated in Fig. 5.6.

The selection algorithm works on the following criteria:

- In the case the cylinder is completely inside the volume of the blade, two mesh points are identified at the minimum distance to the suction and pressure sides and these distances are assigned to C_1 and C_2, respectively. Here, both constraints are satisfied, as illustrated in Fig. 5.7a.
- If there is a collision between the shapes, e.g. at the suction side of the blade such that C_1 is not respected, the algorithm identifies the mesh point at the maximum distance to the suction side. This distance is multiplied with -1 and assigned to C_1, as shown in Fig. 5.7b, while C_2 still has the positive minimum distance to the pressure side. For this example, the negative value of C_1 implies to the optimizer that only the constraint C_1 is violated.

Following the same procedure, the interference detection approach is implemented for all cylinders, yielding the number of eight inequality constraints ($C_{1,\ldots,8}$). Nevertheless, this is not the total number of inequality constraints — there are two additional constraints described in the next section.

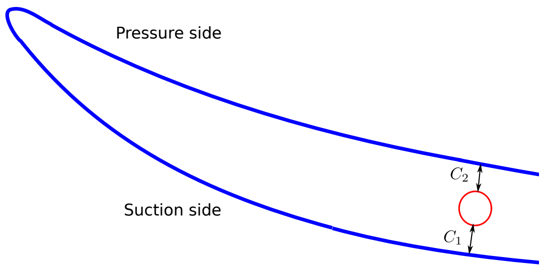
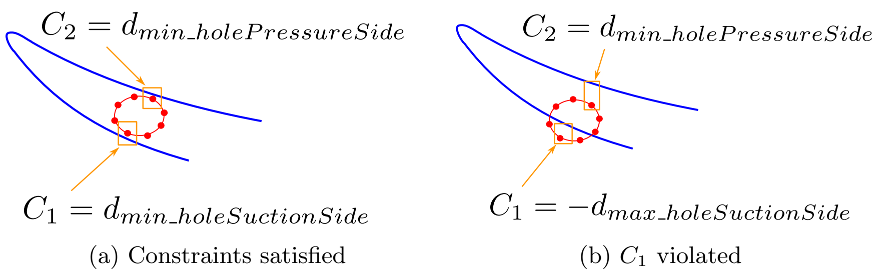


Figure 5.6: Two distances between cylinder and blade[14]


$C_2 = d_{min\_holePressureSide}$
$C_1 = d_{min\_holeSuctionSide}$
$C_2 = d_{min\_holePressureSide}$
$C_1 = -d_{max\_holeSuctionSide}$

(a) Constraints satisfied (b) C_1 violated

Figure 5.7: Inequality constraints definition[15]

5.1.3 Cylinder positioning during shape optimization

The position of all four cylinders is arbitrary as long as they are inside the volume of the blade and the axial distance between cylinder pairs (in correspondence to the hub and the shroud) is at least 60 mm. The proposed solution to allow the movement of the cylinders while respecting this constraint is defined as follows:

1. During the cross-sectional build-up of the blade geometry, two additional B-spline curves are constructed on the bottom (hub) and top (shroud) sections. They are created in the middle between the corresponding suction and pressure B-splines and therefore referred to as ‘mid-lines’, as illustrated in Fig. 5.8.
2. The cylinders are allowed to slide on the mid-line. This is achieved by defining the center coordinates of the cylinder as functions of the parametric coordinate u of the mid-line $(x(u), y(u))$. Since each cylinder has the corresponding

[14]Picture provided by Salvatore Auriemma
[15]Picture provided by Salvatore Auriemma

parametric coordinate u to define its center, there are four additional design parameters considered in the optimization ($u_{1,\dots,4}$).

3. Finally, the axial distance constraints C_9 and C_{10} are defined in Eq. (5.1) and Eq. (5.2), respectively.

$$C_9 = x(u_2) - x(u_1) - 60 \geq 0, \tag{5.1}$$

$$C_{10} = x(u_4) - x(u_3) - 60 \geq 0. \tag{5.2}$$

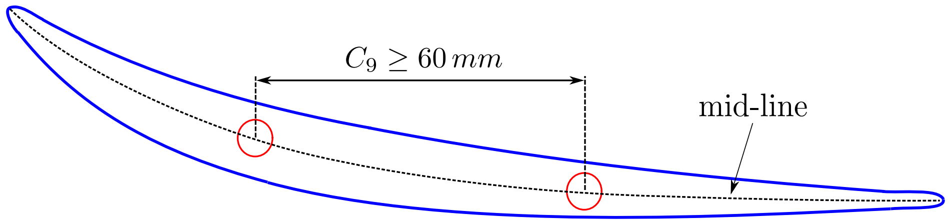


Figure 5.8: Mid-line example and minimum distance between two cylinders

Additionally to the inequality constraints, the parameters u are bounded with lower and upper limits. In this way, every cylinder moves in its own half of the blade and therefore it can not end up on the opposite side of the mid-line (which can happen during the optimization if there are no imposed bounds). It also ensures that the expressions $x(u_2) - x(u_1)$ in Eq. (5.1) and $x(u_4) - x(u_3)$ in Eq. (5.2) are always positive. That is, the values of C_9 and C_{10} become negative only if the axial distance is lower than 60 mm, which would imply that the constraint is violated.

5.2 Optimization results

The gradient-based optimization framework described in Sec. 4.3 has been expanded to support all inequality constraints explained in this chapter. Here, the SLSQP algorithm from the Python's *SciPy* library is chosen as the optimizer. Since the optimization framework used in this work is originally implemented in C++, it required further development to couple Python functions used by the SLSQP optimizer with their corresponding C++ implementations.

The optimization problem is defined as follows:

$$\begin{aligned} \min_{\alpha \in \mathbb{R}^{188}} \quad & J(\alpha) \\ \text{s.t.} \quad & C_i(\alpha) \geq 0, \quad i = 1, \ldots, 10 \\ & lb_j \leq \alpha_j \leq ub_j, \quad j = 1, \ldots, 188 \end{aligned}$$

As opposed to the TUB stator optimization explained in Sec. 4.5, here the total number of design parameters is 188 instead of 184 (4 additional parameters serve to move the cylinders during the optimization) and there are 10 inequality constraints to ensure the fitting of cylinders inside the volume of the optimal blade.

The gradients of the objective function and the constraints are computed using the reverse mode of AD. The derivatives of the objective function with respect to the last four design parameters are equal to zero, i.e. the flow solver is not aware of cylinders and in principle it should not be because the cylinders have to be inside the optimal blade and therefore have no influence on the computational grid analyzed by CFD. The derivatives of the geometric constraints are calculated with respect to the complete design space, i.e. both blade and cylinder design parameters as they all contribute to the gradient information.

The optimization was performed two times. First, the low-fidelity CFD simulation with STAMPS used a computational grid of approximately 20,000 cells. This optimization served as a proof of concept that the assembly constraints can be implemented. Second, the high-fidelity CFD simulation used approximately 800,000 cells which required the computation to be executed by Salvatore Auriemma on a cluster at Queen Mary University of London. The objective function to minimize is the total pressure loss between the inlet and the outlet of the stator.

The CAD-based optimization with the low-fidelity CFD simulation yields 12.3% reduction of the objective function after 18 iterations. The optimization history with constraints is presented in Fig. 5.9, while the comparison between the initial and the optimal blade shape is shown in Fig. 5.10. Here, a height of the cylinders exceeds the blade volume due to presentation purposes, i.e. to better perceive their positions. As visible in Fig. 5.10, the optimizer pushes the cylinders towards the middle of the optimal blade due to a reduction of the blade's thickness, however respecting the minimal axial distance of 60 mm.

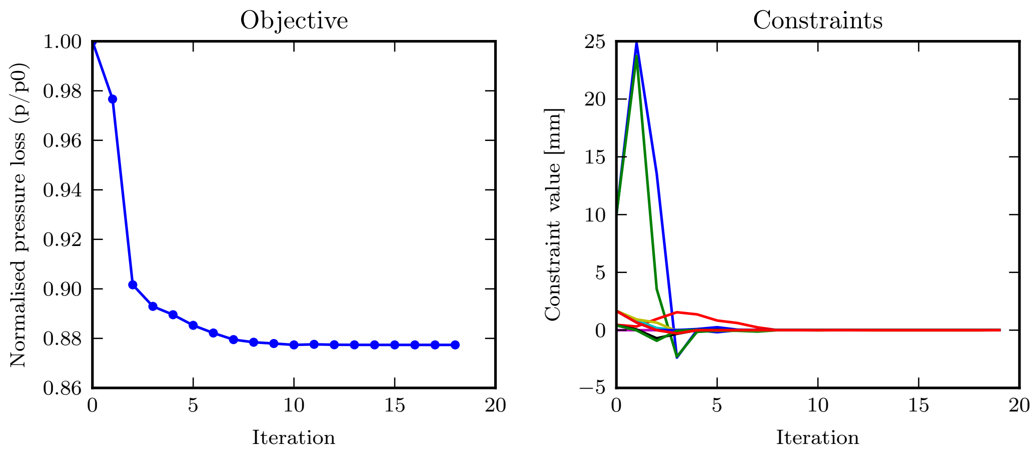


Figure 5.9: TUB stator optimization history with assembly constraints (low-fidelity CFD simulation)

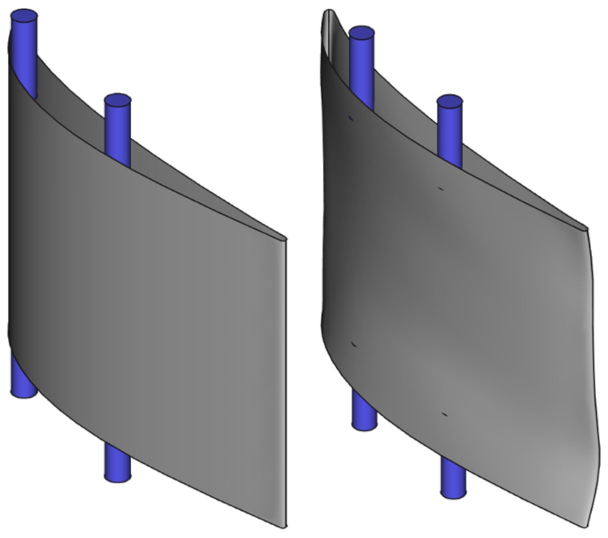


Figure 5.10: Comparison between initial (left) and optimal (right) blade geometry with cylinders (low-fidelity CFD simulation)

Table 5.1: Constraints values for baseline and optimal geometry

Constraint	Initial value [mm]	Optimal value [mm]
C_1	0.42	0.10
C_2	0.46	0.63
C_3	1.66	1.33
C_4	1.68	0.46
C_5	0.42	0.10
C_6	0.46	1.52
C_7	1.65	1.68
C_8	1.68	0.40
$C_9{}^*$	9.90	5.97
$C_{10}{}^*$	9.90	2.36

* One should add 60 mm to get the actual axial distance between cylinders

As one can notice, there are small dots on the optimal blade (Fig. 5.10) that indicate possible violation of constraints, however this is only due to a visualization with FreeCAD. That is, all constraints are satisfied as presented in the plot on the right side of Fig. 5.9.

The CAD-based optimization with the hight-fidelity CFD simulation converged after 42 iterations, reducing the total pressure loss objective by 14%. The optimization history is presented in Fig. 5.11, while the baseline and optimal TUB blade geometries are compared in Fig. 5.12. The optimal blade has a curved trailing edge and it is thinner than the original one, especially when looking at the mid-sections. One can also notice that the position of the bolts is slightly modified during the optimization.

The objective function does not drop continuously during the optimization due to violation of certain constraints. Nevertheless, the optimizer recovers successfully, satisfying all constraints in the end, as shown in Table 5.1.

Furthermore, the comparison between the initial and the optimal mid-span velocity distribution is shown in Fig. 5.13. Here one observes that the separation bubble (blue region in the initial case) is significantly reduced in the final iteration. This contributes to the reduction of the objective function.

[16]Picture provided by Salvatore Auriemma

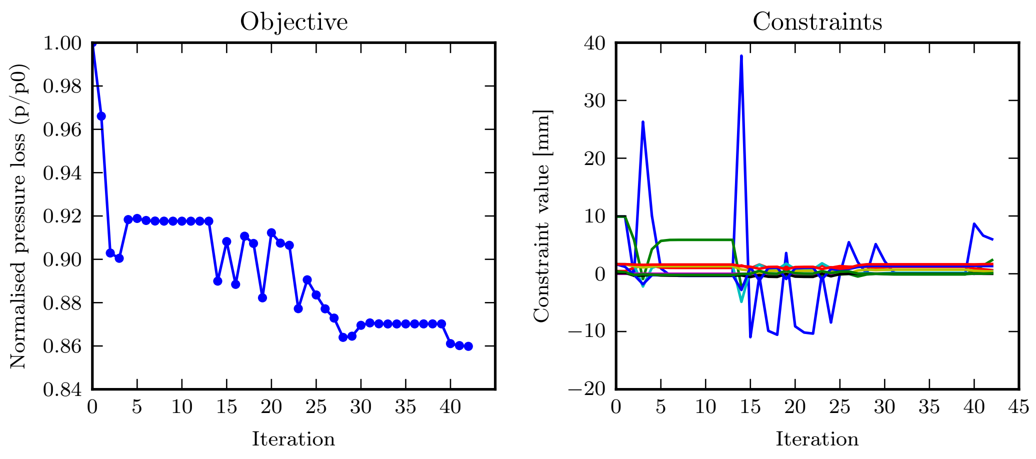


Figure 5.11: TUB stator optimization history with assembly constraints (high-fidelity CFD simulation)

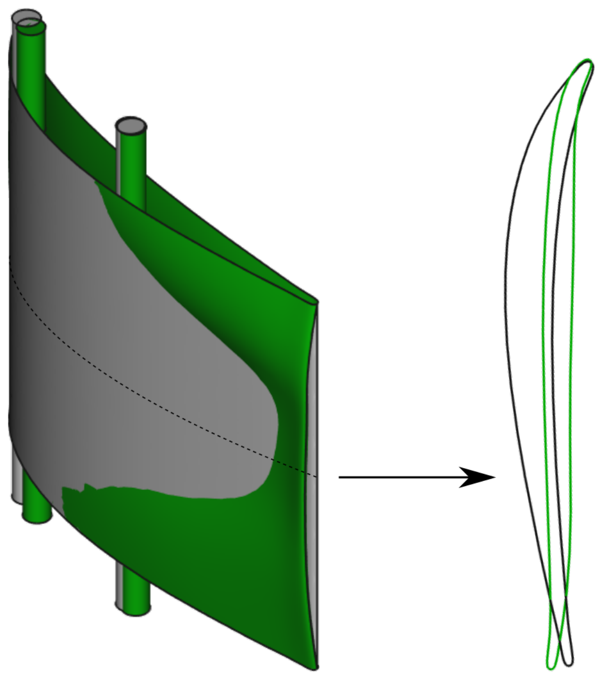


Figure 5.12: Comparison of baseline (gray) and optimal (green) TUB blade geometry and mid-sections (high-fidelity CFD simulation)

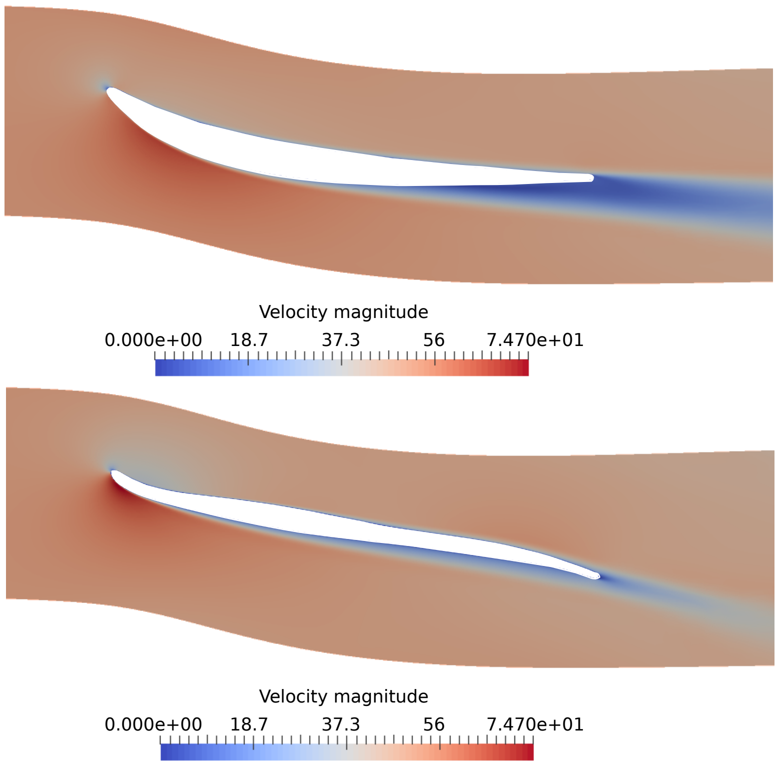


Figure 5.13: Comparison between the mid-span velocity distribution of the initial (top) and final (bottom) iteration of the optimization[16]

5.3 Summary

This chapter proposed an approach to deal with the assembly constraints of the TUB stator during the aerodynamic shape optimization. The optimal blade has to accommodate four mounting bolts (cylinders) — two at the hub and two at the shroud of the stator. Moreover, their position in the blade is arbitrary, however the axial distance between the corresponding cylinder pairs has to be at least 60 mm. For this purpose, ten inequality constraints were developed and provided to the SLSQP optimizer together with the derivative information calculated with the differentiated OCCT kernel. The gradient-based optimization was performed in two stages. First, the optimization with the low-fidelity CFD simulation yields 12.3% reduction of the objective function after 18 iterations. Second, the optimization with the high-fidelity CFD simulation results in 14% reduction of the objective function after 42 iterations. The developed approach ensures there is no collision between the blade and the cylinders.

6 Improved AD of the VKI in-house CAD and grid generation tool

CADO (*Computer Aided Design and Optimization*) is a computer aided design and optimization tool utilized for design of turbomachinery components at the von Karman Institute for Fluid Dynamics [Ver10]. It consists of the following components: a Computer Aided Graphical Design (CAGD) library used for airfoil and blade design, a mesh generation tool (both for the fluid domain and the solid domain) and CAE packages (CFD and CSM).

So far, the CADO framework has been mainly used in a gradient-free optimization context by applying a meta-model assisted evolutionary algorithm to drive the design process. A recent development of a discrete adjoint CFD solver, presented by Mueller et al. [MV17], enabled a shift towards gradient-based optimization methods.

To complete the chain rule of derivatives, ranging from the CAD parametrization to the desired objective function value, one requires the derivative information from the other elements in the VKI's design chain. For this purpose, Sanchez Torreguitart et al. [STVM16] performed the algorithmic differentiation of the CAD kernel and the mesh generator using the *traceless* forward mode of ADOL-C. They used the typedef approach (explained in Sec. 3.4.6), i.e. the alias `cado::Real`, to inject the *traceless adouble* class into the CADO sources. The complete differentiated design chain was demonstrated on the gradient-based optimization of the LS89 axial turbine profile [STVM18].

The next step was to integrate the reverse mode of AD into the CAD and the grid generation tool and couple them with the adjoint CFD solver in order to have the complete reverse differentiated design chain and possibly benefit from an improved efficiency. This stage was successfully achieved during a collaboration between UPB and VKI.

First, one has to replace the declaration type *cado::Real* to become the alias for the *trace-based adouble* class which is implemented in a different header of ADOL-C (`<adolc/adouble.h>`). Every additional source code modification caused by the type replacement was wrapped using the preprocessor directives with conditional compilation to enable simple switching between the original and different versions of the differentiated sources. Second, in the *trace-based* version of ADOL-C one has to use the ADOL-C drivers to compute the derivatives. That is, the CFD sensitivity is evaluated first and then propagated as a derivative seed information to one of the ADOL-C routines, e.g. `fos_reverse` (first-order scalar reverse mode of AD). Such a routine evaluates the *trace* which contains the necessary data about executed operations and intermediate variables in the CAD kernel and the mesh generator. Finally, one obtains the gradient of the objective function, e.g. total pressure loss, with respect to the design space defined at the CAD level. The verification of gradients as well as the performance test of the differentiated sources are described by Ismael Sanchez Torreguitart [ST19].

The performance of the reverse differentiated sources was not satisfactory because the *trace* size took approximately 36 GB of memory. Such a large *trace* cannot fit into the RAM and therefore is stored on a hard drive. This has a negative impact of the performance of the differentiated sources, because reading the *trace* from the hard drive is slower than reading the *trace* from the RAM. The following step was to investigate the CADO sources in order to locate the cause for such an increased memory requirement.

The CADO workflow that creates a computational domain for the CFD tool can be divided into the following steps: (i) construct a 2-D geometry of the LS89 airfoil, (ii) generate a 2-D block-structured mesh, (iii) perform mesh smoothing and (iv) extrude the 2-D mesh computed in the previous step to construct a quasi 3-D mesh that is required by the CFD tool (in sense that the geometry still remains 2-D). The mesh smoothing part is an iterative process that executes almost an identical code sequence 300 times and it was discovered as the main cause for having the large *trace* files. Therefore, the aim is to exploit its structure in order to benefit from an improved efficiency of the differentiated sources.

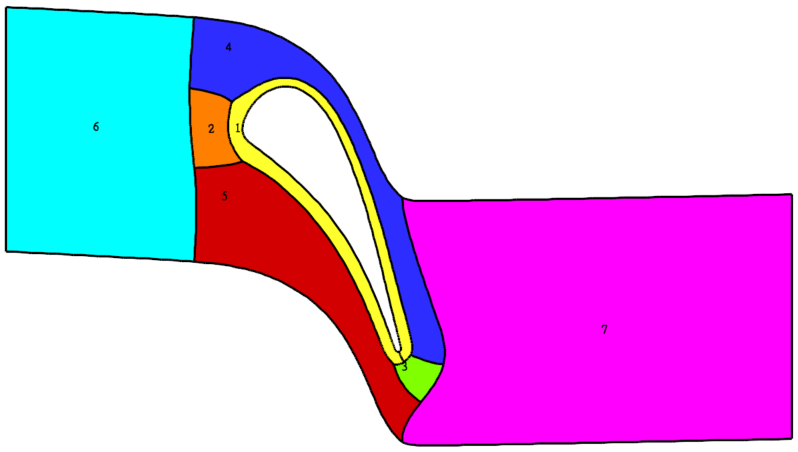


Figure 6.1: LS89 multi-block mesh topology[17]

6.1 Mesh generation of LS89 axial turbine profile

The computational grid of the LS89 airfoil is generated by splitting the domain into several structured blocks that are independent from each other, however sharing common interfaces. This is also referred to as the multi-block approach. Contrary to the single-block approach, it allows more control over cell size and shape, which is beneficial when optimizing a complex geometry [ST19].

The LS89 multi-block grid topology is presented in Fig. 6.1. A single grid is constructed around the LS89 profile (also referred to as the O-grid in [ST19]). There are additional six grids, four of which are placed around the O-grid and two of which serve as the inlet and the outlet domain (these six grids are also referred to as the H-grids in [ST19]).

There are two steps to generate the initial grid. First, the grid points are distributed on the boundary edges of all blocks. After that, the internal points are calculated from the boundary points using the Transfinite Interpolation (TFI) equations [TSW98].

However, as stated in [ST19], when using this method there is no guarantee that the grid lines will not intersect or that corners from the side will not be propagated inside

[17]Picture provided by Ismael Sanchez Torreguitart

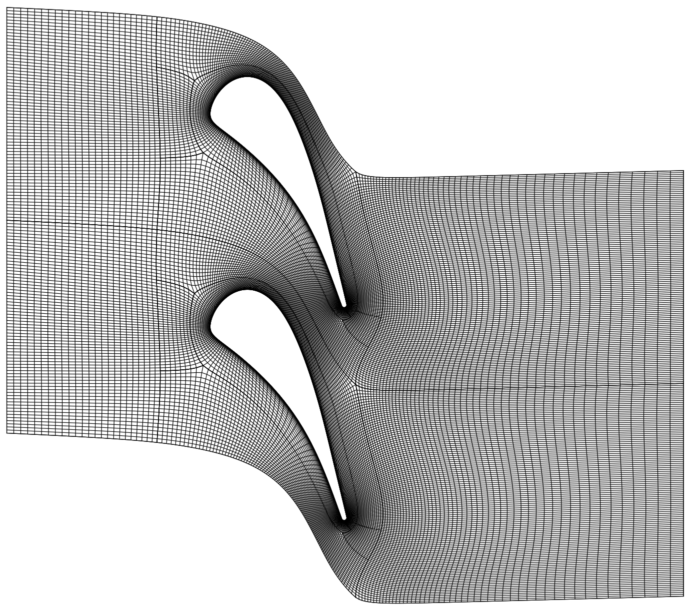


Figure 6.2: LS89 smoothed multi-block mesh[18]

the domain. Therefore, it is advantageuous to smooth the grid blocks afterwards by solving a PDE of a certain type — elliptic in this case.

In particular, the elliptical equations being solved are the Poisson equations and their application to the structured grid generation was proposed by Thompson et al. [TTM74]. The right-hand side of the Poisson equation (also referred to as the source term) is used to control geometrical aspects of the mesh cells. That is, the source terms enable control over the mesh cell orthogonality, clustering and smoothness. Two approaches are used in CADO to define the source terms: (i) the Steger and Sorenson [SS79] method which is employed for the O-grid and (ii) the Laplace method which basically sets all source terms to zero and is mainly used for the H-grids.

The corresponding PDEs used in the CADO mesh smoothing part can be found in [STVM16, ST19]. An example of the final grid after smoothing is shown in Fig. 6.2.

[18]Picture provided by Ismael Sanchez Torreguitart

The next step was to perform the structured AD of the CADO sources such that the iterative process of solving governing PDEs is exploited from the AD perspective.

6.2 Structure-exploiting AD of mesh smoothing process

Instead of having a single large *trace* file which contains the differentiation data about the complete CADO workflow (including 300 iterations of solving the mesh smoothing PDEs), the differentiated CADO sources were modified in order to separate the gradient evaluation on three *traces*, as shown in Fig. 6.3. The *traces* are formed as follows:

1. Trace *T1* computes the LS89 geometry and initializes mesh blocks, interfaces and smoothers.
2. Trace *T2* performs a single 2-D mesh smoothing iteration.
3. Trace *T3* takes the final 2-D mesh after smoothing as input and produces a 3-D mesh that is required by the CFD tool.

Since the main motivation is to detach the mesh smoothing iteration from the rest of the code, every calculation that happens before it goes on the *trace* T1, while everything what happens after the mesh smoothing goes on the *trace* T3. However, the decoupling process is not straightforward because CADO is an object-oriented C++ library. It means that the data about mesh blocks, mesh interfaces and smoothing algorithms is encapsulated in objects.

The smoothing process is actually a *for*-loop that executes a code sequence 300 times (this number is a user-defined variable). In the loop's body, the *Update* method is called on every smoother object. After the update of seven mesh blocks is finished, the mesh interfaces are updated as well. The aim is to eliminate the *for*-loop and trace only the instructions called in its body. Later on, the *trace* T2 can be repeatedly evaluated using the ADOL-C drivers which can speed-up the execution of the differentiated sources.

Simply putting the *trace_on* and *trace_off* ADOL-C commands around this code block would not work, because the tracing process would miss the information about the independent and dependent variables and how they relate to the underlying

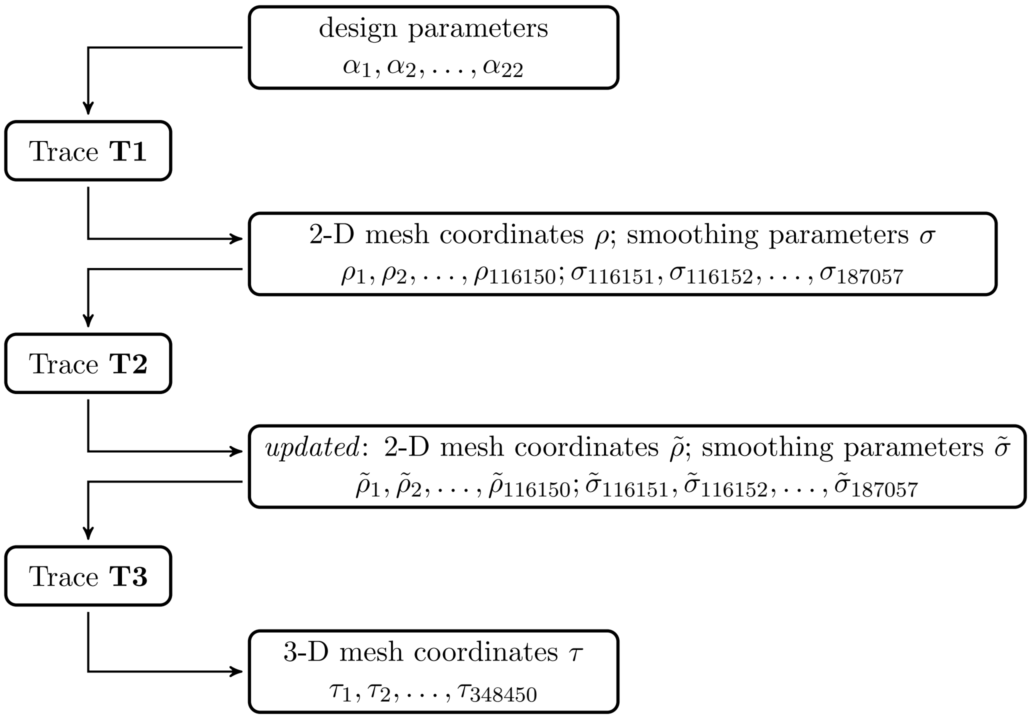

design parameters
$\alpha_1, \alpha_2, \ldots, \alpha_{22}$
Trace **T1**
2-D mesh coordinates ρ; smoothing parameters σ
$\rho_1, \rho_2, \ldots, \rho_{116150}; \sigma_{116151}, \sigma_{116152}, \ldots, \sigma_{187057}$
Trace **T2**
updated: 2-D mesh coordinates $\tilde{\rho}$; smoothing parameters $\tilde{\sigma}$
$\tilde{\rho}_1, \tilde{\rho}_2, \ldots, \tilde{\rho}_{116150}; \tilde{\sigma}_{116151}, \tilde{\sigma}_{116152}, \ldots, \tilde{\sigma}_{187057}$
Trace **T3**
3-D mesh coordinates τ
$\tau_1, \tau_2, \ldots, \tau_{348450}$

Figure 6.3: Improved trace structure of differentiated CADO

code of every *Update* method. To complete this computational graph which defines a single mesh smoothing iteration, one has to investigate how the mesh smoother objects correlate with the mesh blocks and whether there are additional variables encapsulated in the smoother objects that transmit the derivative information from one iteration to another.

In order the simplify the explanation, let us consider an example where only the *O*-grid around the LS89 profile is updated. The simplified code snippet of structuring ADOL-C *traces* in CADO sources is shown in Listing 6.1.

The *trace* T1 (Line 1 – Line 25) includes everything from defining the design parameters to initializing the mesh, mesh interfaces and mesh smoother objects. In addition to marking the active sections of the code to be differentiated (with *trace_on*/*trace_off* routines), one has to mark the independent and dependent variables using the bit-wise shift operators (`<<=` and `>>=`), as described in Sec. 3.3.2. The left-hand side of the operator `<<=` has to be an *adouble* object, while the right-hand side has to be a *primitive*-type variable. The opposite rule applies to the `>>=` operator. An example of marking the design parameters as the independent variables is shown in Line 3 – Line 4. The design parameters are simply re-set using the same values, however they become *activated* from the AD perspective. For this test-case, there are 22 design parameters, however CADO supports advanced LS89 parametrization principles that can involve up to a few hundred parameters.

After constructing the LS89 airfoil geometry and the initial mesh block, a smoother object is initialized (Line 14) by taking a reference to the mesh block in its constructor. The constructor performs additional operations: it initializes the Steger and Sorenson source terms and other variables like the default cell thickness which is calculated from the initial grid. Such variables are crucial for the gradient computation and need to be considered when marking the independent/dependent variables of the ADOL-C *traces*. They are commonly referred to as the smoothing parameters σ in Fig. 6.3.

Before ending the T1 tracing process, all mesh point coordinates ρ and the smoothing parameters σ (that are previously extracted from the smoother object) are marked as the dependent variables. An example of marking the *Block1* is shown in Line 17 – Line 23 of Listing 6.1.

Listing 6.1: Structured AD of CADO workflow (simplified)

```
trace_on(1);
// declare independents for trace T1
m_AxialChordLength <<= m_AxialChordLength.getValue();
m_LERadius <<= m_LERadius.getValue();
// etc. (there are 22 design parameters)

// construct geometry using the previously activated design parameters
Geo airfoil = ConstructGeometry(designParameters);
// generate mesh that contains faces: Block1, Block2 etc.
Mesh2D m_MeshFluid = ConstructMultiBlock(airfoil);
Face2D Block1 = m_MeshFluid.GetBlock1();

// initialize smoother object with a reference to Block1
Smoother2DStegerSorenson Smoother_Block1(Block1);

// declare dependents for trace T1
double output;
for(int i = 0; i < Block1.GetNPntI(); ++i) { // X direction
  for(int j = 0; j < Block1.GetNPntJ(); ++j) { // Y direction
    Block1.GetPnt(i,j).X() >>= output;
    Block1.GetPnt(i,j).Y() >>= output;
  }
}
//... extract and mark the smoothing parameters sigma as dependents
// end the tracing process
trace_off();
// start the middle trace T2
trace_on(2);
// declare independents for T2
for(int i = 0; i < Block1.GetNPntI(); ++i) {
  for(int j = 0; j < Block1.GetNPntJ(); ++j) {
    cado::Real x, y;
    x <<= Block1.GetPnt(i,j).X().getValue();
    y <<= Block1.GetPnt(i,j).Y().getValue();
    Block1.SetPnt(i, j, Vertex2D(x, y));
  }
}
//... declare the sigma parameters as independents
//... provide them to the smoother

//perform the update method
Smoother_Block1.Update();

// declare dependents for T2
for(int i = 0; i < Block1.GetNPntI(); ++i) {
  for(int j = 0; j < Block1.GetNPntJ(); ++j) {
```

```
47     Block1.GetPnt(i,j).X() >>= output;
48     Block1.GetPnt(i,j).Y() >>= output;
49   }
50 }
51 // ... extract the sigma parameters and mark them as dependents
52 // end the middle trace T2
53 trace_off();
54 // start the trace T3
55 trace_on(3);
56 // declare independents for T3 - only the mesh data is relevant
57 for(int i = 0; i < Block1.GetNPntI(); ++i) {
58   for(int j = 0; j < Block1.GetNPntJ(); ++j) {
59     cado::Real x, y;
60     x <<= Block1.GetPnt(i,j).X().getValue();
61     y <<= Block1.GetPnt(i,j).Y().getValue();
62     Block1.SetPnt(i, j, Vertex2D(x, y));
63   }
64 }
65 // construct 3D block
66 // iterate through the mesh coordinates and mark them as dependents
67 // end trace T3
68 trace_off();
```

The middle *trace* T2 is defined as follows. The O-grid and the σ parameters, that are indicated as the dependents of T1, have to be marked as the independents of T2 (Line 30 – Line 37). The order in the marking process is important, as it helps to couple the *trace* evaluation later. Moreover, the activated σ parameters have to be set in the smoother object. After that, the *Update* method can be executed (Line 42). Once the mesh is updated, its coordinates and the σ parameters are labeled as the dependents of T2 (Line 45 – Line 50).

Finally, the *trace* T3 starts by marking only the mesh coordinates as the independents. Here, the σ parameters are not relevant any more as they do not contribute to the construction of a 3-D block. As already mentioned, the 3-D mesh is created by extruding the 2-D mesh in the z-direction. However, all LS89 design parameters are two-dimensional, i.e. there is no design parameter that influences the z-direction. Therefore, the derivatives of the z-coordinates with respect to any design parameter are always zero. For this reason, only the $(x,\ y)$ coordinates of the 3-D mesh are labeled as the dependents of T3. This avoids any unnecessary computation.

During the tracing process, it was observed that all *traces* were stored onto the hard drive. To increase the efficiency of the differentiated sources, one can customize ADOL-C configuration such that the *traces* fit into the RAM, which is explained in the following section.

6.2.1 Tailoring trace size to application requirements

To ensure that all *traces* fit into the RAM (which would enable faster execution), one has first to investigate the size of *trace* buffers for each of the *traces*. As explained in Sec. 3.3.2, the *trace* has four internal arrays: *operation*, *location*, *value* and the *Taylor* buffer. The default array length of these buffers is equal to 524288 (2^{19}). To obtain the actual size of each *trace*, one can call the function `printTapeStats(std::cout, tag)` right after the *trace_off* routine.

If these array lengths are not exceeded during the *trace* generation, the buffers will be kept in the RAM. Otherwise, any buffer that is larger than its predefined size is stored on the hard drive. Since the default array lengths are too small for such an application as the differentiated CADO, one has to customize the buffers' size to keep all *traces* in the memory.

There are two possibilities to customize the array lengths. As mentioned in Sec. 3.3.2, one can create the `.adolcrc` file in the working directory where the executable can be found. In this file, the user can define parameters `OBUFSIZE`, `LBUFSIZE`, `VBUFSIZE` and `TBUFSIZE`. The ADOL-C tool reads the file `.adolcrc` right at the beginning of the program execution to properly set-up the *trace* manager with the user-defined values. After that, every *trace* is initialized with these predefined array lengths. However, this is not an appropriate solution when having multiple *traces*, because the *traces* T1, T2 and T3 have different memory requirements. Therefore, the recommended approach to tailor the array lengths independently for each *trace* is to do it directly while calling the *trace_on* routine.

An example of customizing the buffers of the *trace* T2 is shown in Listing 6.2. To compute the actual *trace* size in bytes, one has to apply the expression shown in Line 9 – Line 11. For this particular case, the *trace* T2 takes approximately 143 MB.

Listing 6.2: Customizing trace buffers

```
trace_on(2,0,
          6765000,  //size of the operation buffer
          17381000, //size of the location buffer
          2029000,  //size of the value buffer
          7626000   //size of the taylor buffer
          );

// actual size in bytes (B) - depends on operating system
unsigned t2size = 6765000 * sizeof(unsigned char) +
                  17381000 * sizeof(unsigned int) +
                  2029000 * sizeof(double) + 7626000 * sizeof(double);
```

The following section describes how to couple the ADOL-C drivers in order to evaluate the generated *traces*.

6.2.2 Coupling ADOL-C drivers to evaluate derivatives

The *traces* are evaluated by employing the ADOL-C drivers. Two of such drivers are used in the differentiated CADO sources: (i) `zos_forward` — evaluates only the *primal* and (ii) `fos_reverse` — evaluates the first-order scalar *adjoint*. Their signatures are presented in Listing 6.3 and Listing 6.4, respectively.

Listing 6.3: ADOL-C `zos_forward` driver (evaluates $y = F(x)$)

```
int zos_forward(tag, m, n, keep, x, y)
short int tag;  // trace identifier
int m;          // number of dependents
int n;          // number of independents
int keep;       // flag for reverse mode preparation
double x[n];    // vector of independents
double y[m];    // resulting vector of dependents
```

Listing 6.4: ADOL-C `fos_reverse` driver (evaluates $z^\top = u^\top F'(x)$)

```
int fos_reverse(tag, m, n, u, z)
short int tag;  // trace identifier
int m;          // number of dependents
int n;          // number of independents
double u[m];    // weight vector
double z[n];    // resulting adjoint
```

The input arguments used by the ADOL-C drivers are given as follows. The *traces* T1, T2 and T3 have identifiers 1, 2 and 3, respectively. The number of independents (n) or dependents (m) for each of the *traces* is defined in Fig. 6.3, where the *trace* structure is presented. For example, the *trace* T1 has $n = 22$ and $m = 187057$, while the *trace* T2 has $n = m = 187057$. The argument *keep* is used only by the forward mode drivers. When it is equal to 1, the `zos_forward` driver stores the required intermediate variables during the forward sweep and prepares the *trace* for the reverse mode. The vector of independents, e.g. regarding the *trace* T2, contains the values of the 2-D mesh point coordinates ρ and the smoothing parameters σ. It is important to note that the values of this vector are set in the same order as they are labeled as the independent variables during the tracing process. If this criterion is not fulfilled, the *trace* might be evaluated without errors, but the computed values will be wrong in the end. The weight vector u (Line 5 of Listing 6.4) can contain, for instance, the adjoint values provided by the CFD solver — which is the case for the *trace* T3. Again, the order of setting the values of the weight vector has to be the same as the dependent variables are marked during the tracing process.

The developed workflow to compute the gradient of the objective function J (e.g. the total pressure loss) with respect to the LS89 design parameters α is presented in Fig. 6.4. The green arrows represent the forward sweeps, while the red arrows represent the reverse (or backward) sweeps.

First, the adjoints of the *trace* T3 have to be evaluated. To achieve that, the following procedure is performed:

1. The forward sweep is executed on T1 by taking the design parameters α as inputs. The output of T1 is stored because it will be required more than once during the whole process.
2. The output of T1 is given as input to the forward sweep of T2 which computes the mesh smoothing iteration for $k = 1$. Next, the output of the first iteration is provided as input to the forward sweep of T2 to execute the second iteration ($k = 2$) and so forth, until $k = 300$.
3. The forward sweep of T3 with $keep = 1$ takes the final output of T2 as input. After it is completed, the reverse sweep of T3 can be executed by taking the adjoint information from the CFD tool as the weight vector u. As the result,

one retrieves the adjoints from the *trace* T3.

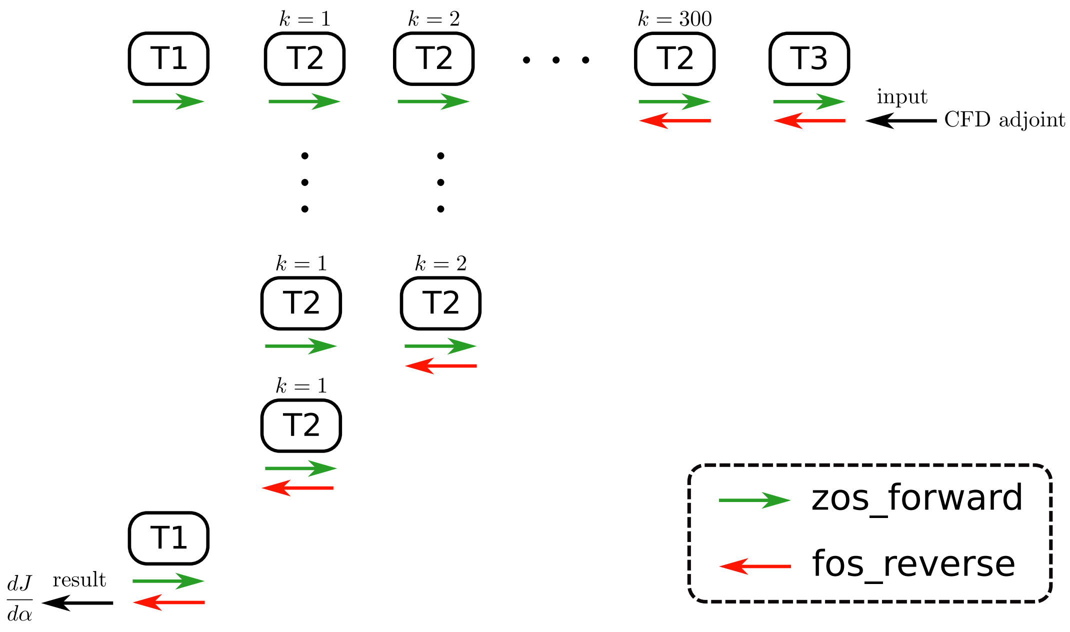


Figure 6.4: ADOL-C driver coupling

The following task is to compute the adjoints of all mesh smoothing iterations, which requires much more effort than the previous steps. For example, to compute the adjoints of T2 for $k = 300$, one has first to execute 300 forward sweeps of T2 ($k = 1, 2, \ldots 300$), while setting the *keep* flag to 1 only during the last iteration ($k = 300$) in order to prepare the *trace* for the reverse sweep. Similarly, to compute the adjoints of T2 for $k = 299$, one has to start from the beginning and execute 299 forward sweeps of T2 ($k = 1, 2, \ldots, 299$), while setting $keep = 1$ during the last iteration ($k = 299$). This pattern can be also noticed in Fig. 6.4. The amount of the forward sweeps of T2 required to reverse differentiate the mesh smoothing part can be calculated as the sum of arithmetic progression $(1, 2, \ldots, 300)$:

$$S_n = \frac{n}{2}(a_1 + a_n) = \frac{300}{2}(1 + 300) = 45150\,.$$

Although a single *trace* execution time with ADOL-C is negligible for T2, the large number of 45150 forward sweep executions slows down the performance of the differentiated sources. With this approach, it takes approximately 20 minutes on an average desktop computer (with Intel Core i5 processor) to compute the adjoints of the mesh smoothing process, which is about 300 times slower than an execution of the original code.

Nevertheless, there is a solution to reduce the number of T2 forward sweeps. That is, the output values of certain forward sweeps can be copied to vectors which will be used as snapshots (also known as checkpoints). A checkpoint represents a record of values of all variables at a certain time of the program execution. The aim is to distribute a number of checkpoints in the mesh smoothing part such that the forward sweeps of T2 do not have to start from the beginning ($k = 1$) every time when computing the adjoints of the mesh iteration k.

Two checkpointing strategies are employed in this work: *equidistant* and *binomial*. The equidistant (or *one-level*) checkpointing technique is the simplest one and it distributes checkpoints equidistantly over a given interval. It can be useful when the total number of iterations is know a priori (which is equal to 300 in this case) and the *trace* evaluation time is almost the same for every iteration. Here, a distribution of checkpoints is determined by the user-defined step size s such that the output of any iteration k, which is a multiple of s, is stored as a checkpoint. An example of the ADOL-C driver coupling together with the equidistant checkpoint distribution is illustrated in Fig. 6.5, where the step size $s = 3$. With this approach, the total number of T2 forward sweeps would be reduced from 45150 to 600, which would significantly speed-up the execution of the differentiated sources.

A more sophisticated solution is the binomial checkpointing [WG04], which is implemented in the algorithm named *Revolve* [GW00]. *Revolve* provides the optimal strategy of distributing the checkpoints such that the computational effort is minimized. This algorithm is integrated into the differentiated CADO sources to efficiently compute the adjoints of the mesh smoothing process.

The checkpoints require additional memory, which is discussed in Sec. 6.3 together with run-time tests.

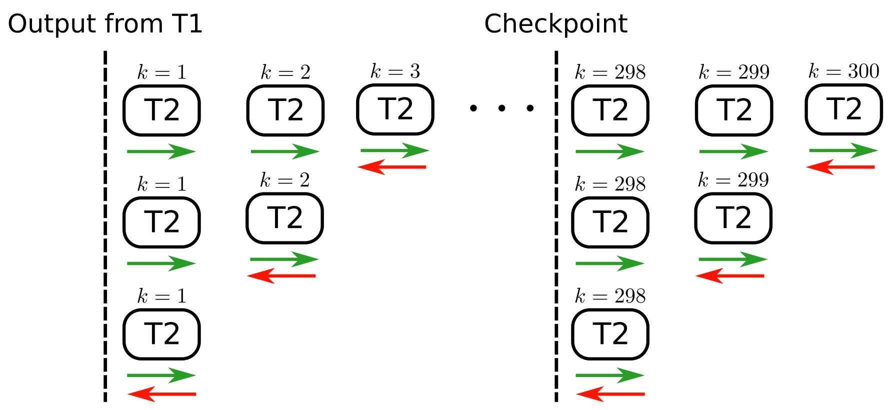


Figure 6.5: ADOL-C driver coupling with checkpoints

6.2.3 Handling conditional branches in mesh smoothing

After the successful coupling of the ADOL-C drivers, it was still not possible to re-evaluate the *trace* T2 for all mesh smoothing iterations. That is, while progressing from one iteration to another, the `zos_forward` driver stopped the evaluation and reported that code branching is detected. This can happen when evaluating the same *trace* with a different values of independent variables than the ones that were used for generating the *trace*. The reason is that a different set of values could traverse different paths of the program execution caused by e.g. *if*-statements. If this happens, the *trace* does not contain further information about the code branches that were not executed during the *trace* recording process. That is, the information stored on the *trace* always follows the *primal* program flow originated by a specific set of independent variables. This behavior represents a general issue for operator-overloading AD tools when reusing the *trace* with arbitrary input values.

ADOL-C detects certain code branching only if an *adouble* object is evaluated in the *if*-statement. Otherwise, when evaluating only primitives or other data-types in the conditional statements, ADOL-C does not store any kind of information that the branching occurred. This could lead to wrong derivatives when re-evaluating the *trace* for arbitrary arguments and the user will not even receive a warning. Therefore, it is important to be very familiar with the differentiated source code and check if every relevant *if*-statement evaluates the *adouble* object in its condition (instead of a primitive variable). Once this is ensured, the ADOL-C tool stores the information about the condition being evaluated and also the result of the condition such that it can warn the user if a different path of the program flow is traversed.

In the case the user receives such a warning, there are two possibilities to deal with this problem. The simpler option is to re-trace the code with the current set of independent variables and after that to call the desired ADOL-C driver to evaluate the *trace*. However, this solution is not preferred as it would significantly slow down the performance of the differentiated sources and the whole previous effort about decoupling the *traces* would probably be worthless. Another possibility is to replace simple branches with conditional assignments that are specially treated by ADOL-C.

For this purpose, ADOL-C provides a function named `condassign(a, b, c, d)`. It corresponds to the following conditional (ternary) operator in C++:

$$a = (b > 0) \ ? \ c \ : \ d;$$

where all arguments are actually *adoubles*. If the statement $b > 0$ yields *true*, $a = c$, otherwise, $a = d$. The header `<adolc/adouble.h>` contains also the corresponding definition for the passive arguments (primitive data-types) such that the modified code can be tested against the original code by considering only the primal values. This function is employed in the CADO sources to treat simple *if*-statements. An example is presented in Listing 6.5 as a comparison between the original code (Line 2 – Line 8) and the modified (differentiated) code (Line 10 – Line 11).

Listing 6.5: Conditional assignments in ADOL-C

```
 1 //original code
 2 if ( m_Pt(i,0) > 0.0 ) {
 3   x_xi = x_xi_forward;
 4   y_xi = y_xi_forward;
 5 } else {
 6   x_xi = x_xi_backward;
 7   y_xi = y_xi_backward;
 8 }
 9 // modified code with condassign function of ADOL-C
10 condassign(x_xi, m_Pt(i,0), x_xi_forward, x_xi_backward);
11 condassign(y_xi, m_Pt(i,0), y_xi_forward, y_xi_backward);
```

Table 6.1: Gradient comparison ($dJ/d\alpha$) between black-box reverse AD and structured reverse AD

Black-box reverse AD	Structured reverse AD	Abs. difference
1.729432013772**604**e+00	1.729432013772**566**e+00	3.80e-14
-1.068101293938237e-02	-1.068101293938237e-02	0.00e+00
1.72449547356**3913**e-02	1.72449547356**4018**e-02	1.05e-15
6.540462563168633e-01	6.540462563168633e-01	0.00e+00
⋮	⋮	⋮
-3.824269579188**263**e+01	-3.824269579188**149**e+01	1.14e-12*

*Maximal difference

After the *condassign* function was integrated into the differentiated CADO sources, all mesh smoothing iterations were successfully computed by reusing the *trace* T2 without any warnings or errors.

6.2.4 Gradient verification

The structured *trace* workflow presented in Fig. 6.4 to compute $dJ/d\alpha$ is validated against the original ('black-box') reverse mode of AD where a single *trace* includes everything. The gradient comparison is shown in Table 6.1. As one can notice, there are very small discrepancies with respect to a few design parameters. However, they are close to machine precision and therefore acceptable.

6.3 Performance tests of differentiated CADO

The performance of the reverse differentiated CADO sources is measured while computing the total gradient $dJ/d\alpha$. Here the CFD sensitivity is previously computed and just loaded from a file, such that almost all computational resources are dedicated to the CAD, mesh generation and mesh smoothing parts.

There are two types of the LS89 parametrization, with the total number of 22 and 230 design parameters, respectively. They execute different paths in the geometry construction code, thus influencing only the *trace* T1, while the *traces* T2 and T3 remain the same.

Table 6.2: Run-time and memory requirements of differentiated CADO with respect to LS89 profile defined by 22 design parameters

AD mode	Avg. run-time* [s]	Run-time ratio	Memory [GB]
Primal	3.26		0.07
Traceless forward vector	96.21	29.51	0.56
Black-box reverse	130.43	40.01	36.14
Structured reverse	1229.23	377.06	1.69
Equidistant checkpointing	56.93	17.46	1.77
Binomial checkpointing	47.63	14.61	1.78

*Run-time averaged on 5 measurements

Five different derivative computation modes are considered for measuring the performance: (i) *traceless* forward vector mode (with 22 and 230 directions, depending on the parametrization type), (ii) black-box reverse mode, (iii) structured reverse mode, (iv) structured reverse mode with the equidistant checkpointing and (v) structured reverse mode with the binomial checkpointing.

Regarding the equidistant checkpointing, the checkpoints are taken every five steps of the mesh smoothing process, which results in the total number of 900 T2 forward sweeps and 59 checkpoints. The output values of the *trace* T1 are not deleted after its evaluation and they serve as the initial checkpoint (otherwise there would be in total 60 checkpoints instead of 59). For this setup, the checkpoints occupy 84.2 MB of memory (each checkpoint is a *double*-type array of 187057 elements).

Concerning the binomial checkpointing, the algorithm *Revolve* is initialized with two arguments (numbers): the total number of 300 mesh smoothing steps and the total number of 60 checkpoints. The checkpoints occupy 85.6 MB of memory. As opposed to the equidistant checkpointing, the *Revolve* algorithm executes only 538 T2 forward sweeps.

Table 6.2 and Table 6.3 compare average run-time, run-time ratio (t_{AD}/t_{primal}) and the peak memory requirements (evaluated with the profiling tool *Massif*) of the differentiated CADO sources with respect to LS89 profiles defined by 22 and 230 design parameters, respectively. Additionally, the run-time and memory requirements of the *primal* code are also shown. Furthermore, the information from Table 6.2 and Table 6.3 is visualized in Fig. 6.6 and Fig. 6.7, respectively, such that one can easily

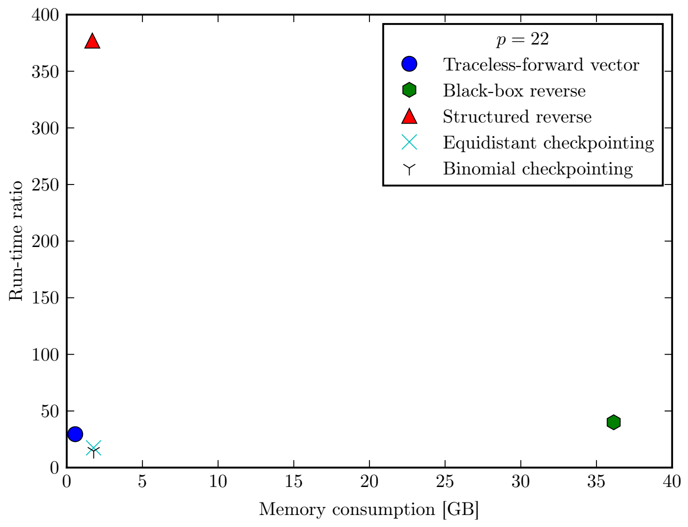


Figure 6.6: Run-time ratio vs. memory consumption for different AD modes with respect to $p = 22$ directions (corresponds to Table 6.2)

observe a trade-off between the run-time ratio and the memory consumption with respect to different AD modes. The number of directions p for which the derivatives are computed is equal to 22 and 230, regarding the performance results presented in Table 6.2 and Table 6.3, respectively.

Analyzing the data of Table 6.2, one can notice that the structured reverse mode with the binomial checkpointing requires approximately three times more memory than the *traceless* forward AD, however it obtains a memory reduction of 95% comparing to the original reverse mode of AD. Moreover, it yields the lowest run-time comparing to other AD modes.

When using the design space of 230 variables, the memory requirements of the reverse differentiated sources increase by approximately 6 GB, as visible in Table 6.3. This is caused by a larger amount of instructions recorded on the *trace* T1 due to a more complex LS89 geometry. Nevertheless, the *traces* T2 and T3 require the same amount of memory as in the previous case because the mesh part does not change. Moreover, Table 6.3 shows a significant advantage in run-time of the structured reverse mode with the binomial checkpointing over other AD modes. Even though

Table 6.3: Run-time and memory requirements of differentiated CADO with respect to LS89 profile defined by 230 design parameters

AD mode	Avg. run-time* [s]	Run-time ratio	Memory [GB]
Primal	4.03		0.07
Traceless forward vector	989.66	245.57	5.00
Black-box reverse	157.60	39.11	42.06
Structured reverse	1247.08	309.45	7.58
Equidistant checkpointing	73.16	18.15	7.66
Binomial checkpointing	63.59	15.78	7.72

*Run-time averaged on 5 measurements

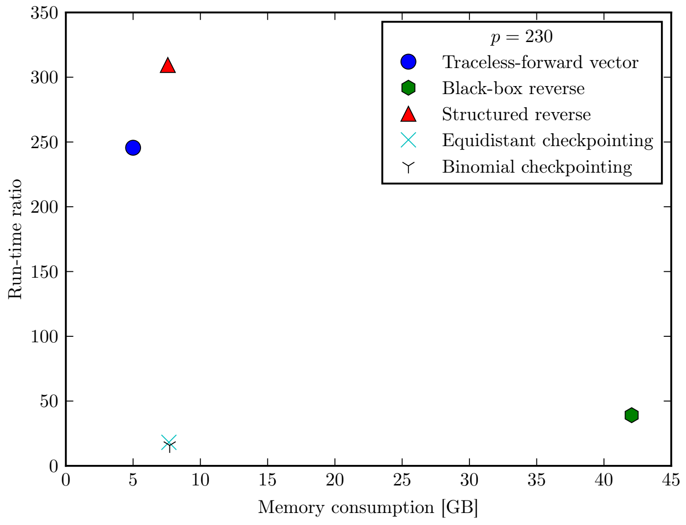


Figure 6.7: Run-time ratio vs. memory consumption for different AD modes with respect to $p = 230$ directions (corresponds to Table 6.3)

it requires approximately 1.5 times more memory than the *traceless* forward mode, the execution of *traces* is very fast because they fit in the RAM. Furthermore, it yields a memory reduction of 82% comparing to the black-box reverse mode of AD for this particular case.

6.4 Summary

The structure-exploiting AD has been successfully applied to the reverse differentiated CADO sources to reduce large memory requirements imposed by the 'black-box' reverse differentiation where a single *trace* contains the whole information about the geometry, the mesh generation and the mesh smoothing. However, this achievement imposed a significant amount of code modification to split the initial large *trace* into three smaller *traces*. The main effort was invested into the 2-D mesh smoothing part such that the middle *trace* corresponds only to a single mesh smoothing iteration. Once this is achieved, one can employ the ADOL-C drivers to re-use the same *trace* in order to compute the gradients of all mesh smoothing iterations.

Additionally to the structured reverse AD, two checkpointing techniques were employed to reduce the total number of *trace* evaluations, namely the equidistant and binomial checkpointing. The second approach allows the optimal distribution of checkpoints to minimize the computational efforts for evaluating the adjoints of the mesh smoothing process. The total number checkpoints is equal to 60 and they occupy approximately 85 MB of additional memory which is negligible comparing to the total amount of memory required by the differentiated sources.

Two types of the LS89 parametrization are considered for measuring the performance. When using the first parametrization with 22 design variables, the structured AD with the binomial checkpointing reduces the memory consumption from 36.14 GB to only 1.78 GB, i.e. by 95%. When considering the parametrization with 230 design variables, the structured AD with the binomial checkpointing reduces the memory consumption from 42.06 GB to 7.72 GB, i.e. by 82%. Moreover, its performance prevails over any other differentiation mode used in this test-case.

7

Algorithmic differentiation of an industrial airfoil design tool

Based on the OCCT differentiation experience, an industrial CAD tool has been differentiated, namely the Rolls-Royce in-house airfoil design and blade generation tool Parablading. This achievement represents the first example of applying AD to an industrial design workflow. The differentiated Parablading tool is coupled with a discrete adjoint CFD solver that is part of the Rolls-Royce in-house HYDRA suite of codes, also produced by AD. This complete differentiated design chain is utilized to perform the gradient-based optimization of the TUB stator test-case to minimize the total pressure loss and exit whirl angle objectives. The TUB stator parametrization in Parablading is explained in Sec. 2.2.

7.1 Parablading differentiation

7.1.1 Introduction to Tapenade

Tapenade [HP13] is the *source-transformation* AD tool developed at INRIA. It computes the first-order derivatives of computer programs written in Fortran and C. As opposed to the operator-overloading concept (e.g. used in ADOL-C), the source (program) transformation generates intermediate source files. In order to differentiate a certain program with Tapenade, the user provides its source files to the AD tool as inputs. Tapenade parses the input files to generate an internal representation. This internal representation serves to build the differentiated code as the output. Such an output usually follows the original code structure with additional derivative statements that are readable to the end-user. Similarly to ADOL-C, Tapenade supports the forward and the reverse mode of AD, together with the *scalar* and *vector* derivative computation options.

To differentiate a certain function/routine with Tapenade, one calls the Tapenade executable with the following basic set of arguments:

- input language, only if Tapenade cannot determine it from the file extension,
- name of the routine to be differentiated with a list of arguments to be treated as independent (input) / dependent (output) variables,
- differentiation mode: forward or reverse,
- filename where the routine is located.

If the function to be differentiated employs other functions that are implemented in different source files, the user can also provide a list of filenames where the functions found in the call-graph are located. In this way, Tapenade is able to differentiate them as well. However, if certain functions cannot be differentiated due to the fact that they are not found in the provided files, the AD tool gives a warning about these issues.

Optional arguments serve to specify output language, the *vector* mode differentiation and the output filename/directory where the differentiated routine(s) should be stored. The detailed explanation is available in the Tapenade tutorial whose location is provided with the sources.

7.1.2 Mixed-language AD of Parablading

The Parablading tool is differentiated in the forward mode of AD, using the AD software tools ADOL-C and Tapenade. The reason for applying two different AD techniques comes from the fact that Parablading is developed both in C++ and Fortran. Such a code environment requires mixed-language algorithmic differentiation.

Bischof et al. [BJMS95] applied this approach to differentiate a 3-D volume grid generation software. They combined AD tools ADIC [BRMO97] and ADIFOR [BKMC96] to compute the derivative information from the source code developed in C and Fortran, respectively. Moreover, Pascual et al. [PH18] utilized the mixed-language AD approach to differentiate the *CalculiX* finite element library. They de-

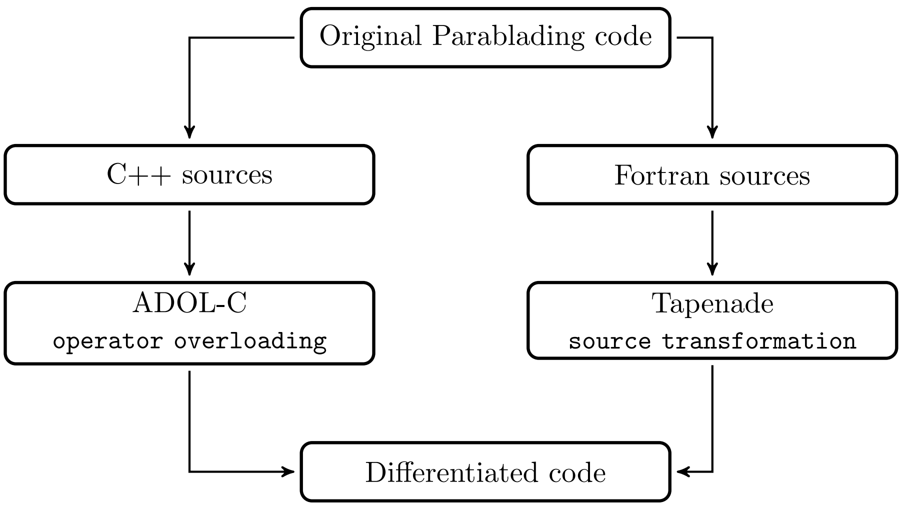


Figure 7.1: Mixed-language AD applied to Parablading

veloped an extension for the Tapenade tool to differentiate a code written in C and Fortran. However, object-oriented languages like C++ are so far not supported by the source-transformation tools like Tapenade. Therefore, the operator-overloading AD tool ADOL-C has been used to differentiate a large part of the Parablading source code written in C++, while Tapenade has been used to differentiate the rest of the sources written in Fortran, as shown in Fig. 7.1.

ADOL-C has been integrated in Parablading by using the *typedef* approach (explained in Sec. 3.4.6) that was proven to be successful with the OCCT CAD kernel. As opposed to OCCT, Parablading did not have any existing alias for the *double* data-type. For this reason, the alias `doublereal` — as shown in Listing 7.1 — is defined. The alias corresponds to the *traceless adouble* class of ADOL-C. After this, the differentiation is executed by replacing the declaration types of all relevant *real* variables in the sources with the *doublereal* type.

There are several practical reasons for choosing the *traceless adouble* as the starting point of the Parablading differentiation. First, the forward mode of AD serves as a reference to later verify the derivatives calculated with the reverse mode of AD. Second, the *traceless* forward mode performs faster than the *trace-based* forward mode (as demonstrated with the differentiated OCCT kernel in Sec. 3.6). Furthermore, it is simpler to debug the differentiated code that uses the *traceless adouble* class because the primal and the derivative information can be easily extracted from the *adouble* objects by calling the methods *getValue* and *getADValue*, respectively. The debugging of the differentiated code that employs the *trace-based* differentiation modes requires a few more steps to obtain the derivatives as mentioned in Sec. 3.3.2. Finally, due to the coupling with a CFD solver which runtime is dominant for the whole optimization process, the runtime increase caused by using the forward mode of AD in comparison to the reverse mode of AD just for the CAD part is insignificant.

Listing 7.1: Entry point for AD

```
#define AUTODIFF 1 //or 0
#if AUTODIFF
typedef adtl::adouble doublereal;
#else
typedef double doublereal;
#endif
```

In addition to the defined alias for the *traceless adouble* class, the author used *preprocessor directives* (as visible in Listing 7.1). The preprocessor directives are source lines preceded by a hash sign (#) and analyzed before compilation. One of their features is *conditional compilation* that allows to perform or skip the compilation of a certain program part based on user-defined criteria. In this case, the condition is based on a flag (macro), e.g. *AUTODIFF*, and its state (0 or 1).

This approach is very useful for the Parablading differentiation, because all source code modifications imposed by the differentiation are wrapped with the preprocessor directives in order to distinguish between the original and the differentiated sources. Such a coding style with the conditional compilation may not look elegant, but allows to switch between the original and the differentiated sources (and vice-versa) just by activating/deactivating the flag *AUTODIFF*. The maintenance aspects of this solution are discussed in Sec. 7.1.4.

The differentiated Parablading features both *scalar* and *vector* variants to compute the derivatives. The switching process from the *scalar* to the *vector* mode of AD is straightforward in ADOL-C, i.e. it does not impose modifications to the differentiated sources (as explained in Sec. 3.3.1). On the other hand, Tapenade requires re-differentiation of the sources with the `-multi` flag turned on in order to support the *vector* mode of AD. For this reason, the Parablading AD development is divided into two branches that correspond to the *scalar* and the *vector* mode of AD, leading to two different versions of the differentiated Parablading tool. The derivatives computed with the vector forward mode of AD have been successfully verified against the scalar forward mode of AD.

It is not straightforward to link ADOL-C and Tapenade in a single differentiated code because they do not use the same data format. While the *traceless* option of ADOL-C encapsulates both primal and derivative information in the *adouble* object (internally it is an array of size $p + 1$), Tapenade works only with primitive data types and treats the primal and the derivative information as separate variables. Therefore, any relevant data needs to be propagated manually from one AD tool to another.

On the contrary, when using the *trace-based* differentiation option, ADOL-C provides an elegant solution to couple the functions that are differentiated manually or by other AD tools. This concept is known as *external function differentiation*, where the user provides the required derivatives by using a certain interface defined by ADOL-C. However, this interface is not available for the *traceless* option used to differentiate the Parablading sources.

As an example of linking *traceless* ADOL-C and Tapenade in the Parablading source code, let us consider a Fortran function from SLATEC library [VH82] that computes point coordinates on a B-spline curve — named `dbvalu` [DB77]. The coupling example of the *scalar* differentiated version `dbvalu_d` is shown in Listing 7.2.

First, the information from the *adouble* array `knot` is extracted to the *double*-type arrays `knotPrimal` and `knotDerivative` (Line 20 – Line 25). The same procedure is repeated for the control points `cp` and parameter `t`. The next step is to call the differentiated routine `dbvalu_d` with the previously created variables (Line 34 – Line 37). Finally, the results `dbvaluPrimal` and `dbvaluDerivative` are copied to the *adouble* object `result` (Line 39 – Line 40). For comparison purposes, Listing 7.2 also shows the original code for executing the `dbvalu` function (Line 3 – Line 10) to give an impression of code modifications that were required upon differentiation.

Listing 7.2: Linking ADOL-C and Tapenade in *scalar* mode

```
#if !AUTODIFF
//original code
inline double dbValue(std::vector<double> &knot,
    std::vector<double> &cp, int *degree, int *deriv, double *t,
    std::vector<double> &work) {
  int M = static_cast<int>(cp.size());
  int inbv(1);
  return dbvalu_(&knot[0], &cp[0], &M, degree, deriv, t,
                    &inbv, &work[0]);
}
#else
//differentiated code in scalar mode
inline doublereal dbValue(std::vector<doublereal> &knot,
    std::vector<doublereal> &cp, int *degree,
    int *deriv, doublereal *t,
    std::vector<double> &work, std::vector<double> &workd) {
  int M = static_cast<int>(cp.size());
  int inbv(1);

  std::vector<double> knotPrimal(knot.size());
  std::vector<double> knotDerivative(knot.size());
  for(size_t i = 0; i < knot.size(); ++i) {
    knotPrimal[i] = knot[i].getValue();
    knotDerivative[i] = knot[i].getADValue(0);
  }

  //...
  //perform the same extracting procedure for control points cp
  //...

  double tPrimal = (*t).getValue();
  double tDerivative = (*t).getADValue(0);
  double dbvaluPrimal;
  double dbvaluDerivative = dbvalu_d_(&knotPrimal[0],
    &knotDerivative[0], &cpPrimal[0], &cpDerivative[0],
    &M, degree, deriv, &tPrimal, &tDerivative,
    &inbv, &work[0], &workd[0], &dbvaluPrimal);

  doublereal result = dbvaluPrimal;
  result.setADValue(0,dbvaluDerivative);

  return result;
}
#endif
```

Once the vector mode of AD is employed, the linking process between ADOL-C and Tapenade changes. First of all, the signature of the `dbvalu` routine differentiated in the vector forward mode (named `dbvalu_dv`) is different than its *scalar* version. Listing 7.3 presents the definition of the `dbvalu_dv` routine implemented in Fortran. The routine does not return the derivative value *dbvalud* as it is the case in the *scalar* version, because the derivative is now represented by an array instead of a single value. Therefore, the reference to *dbvalud* array can be found in the argument list (Line 2). Another difference is the last argument *nbdirs* (Line 2) which is only relevant for the vector mode of AD. It represents the number of directions for which the derivatives should be evaluated.

Listing 7.3: Fortran routine `dbvalu_dv` and its argument types

```
 1 SUBROUTINE DBVALU_DV(knots, knotsd, cp, cpd, m, degree, deriv,
 2   t, td, inbv, work, workd, dbvalu, dbvalud, nbdirs)
 3 INCLUDE 'DIFFSIZES.inc'
 4
 5 INTEGER m, degree, deriv, inbv
 6 DOUBLE PRECISION knots, cp, work, t
 7 DOUBLE PRECISION knotsd, cpd, workd, td(nbdirsmax)
 8 DIMENSION knots(*), cp(*), work(*)
 9 DIMENSION knotsd(nbdirsmax,*), cpd(nbdirsmax,*), workd(nbdirsmax,*)
10 DOUBLE PRECISION dbvalud(nbdirsmax)
11 DOUBLE PRECISION dbvalu
12 INTEGER nbdirs
13 ...
14 END
```

Furthermore, Listing 7.3 shows the declarations of arguments and their dimensions (Line 5 – Line 12). There, one can notice the constant named *nbdirsmax*. This constant is defined in the file '*DIFFSIZES.inc*' (Line 3) and it represents the maximal number of directions used by the differentiated Parablading. The following relation must hold: $nbdirs <= nbdirsmax$. Otherwise, a runtime error may occur if the program tries to access elements that are outside of array bounds. For the TUB stator test-case, the *nbdirsmax* is set to 196, because this is the total number of design parameters. However, the user can for example compute the derivatives only with respect to a group of 28 parameters. In that case, the value of *nbdirs* can be set to 28. Later in the body of the routine, every loop that iterates through the derivative directions stops at the element index equal to *nbdirs*.

Another important observation in Listing 7.3 is that the array of *knots* has a corresponding matrix of derivatives *knotsd* (Line 9), such that each row of this matrix corresponds to one derivative direction. The same applies for control points *cp*.

Fortran stores matrices (or higher dimensional arrays) as a linear sequence of elements. Moreover, array storage in Fortran is *column-major*, which means that adjacent elements of a column reside next to each other. This is important to know when propagating the information from the *adouble* objects to one-dimensional arrays that the Fortran routines expect.

Listing 7.4 presents a coupling example of the *vector* differentiated `dbvalu_dv` routine. The coupling follows the same procedure as shown in Listing 7.2, however there is a difference in dimensions of the variables that correspond to derivatives (a single value becomes a vector, a vector becomes a matrix). For example, the code in Listing 7.4 shows how to properly extract the derivative information from the *adouble* vector `knot` to the *double* vector `knotDerivative` (Line 13 – Line 19), which is actually a matrix in the Fortran routine `dbvalu_dv`. On the C++ side, one has to ensure that the matrix of derivatives is stored as a one-dimensional array by respecting the *column-major* storage order required by Fortran. As mentioned previously, the signature of `dbvalu_dv` is different than its *scalar* version and this also affects how it is called from the C++ side (Line 34 – Line 37). Finally, the resulting derivatives represented by the vector `dbvaluDerivative` are copied using the *for*-loop to the *adouble* object result (Line 40 – Line 42).

Listing 7.4: Linking ADOL-C and Tapenade in *vector* mode

```
 1 //differentiated code in vector mode
 2 inline doublereal dbValue(std::vector<doublereal> &knot,
 3     std::vector<doublereal> &cp, int *degree,
 4     int *deriv, doublereal *t,
 5     std::vector<double> &work, std::vector<double> &workd) {
 6   int M = static_cast<int>(cp.size());
 7   int inbv(1);
 8
 9   const size_t numDir = adtl::getNumDir(); //equals to 196
10
11   std::vector<double> knotPrimal(knot.size());
12   //matrix of derivatives in 1-D representation
13   std::vector<double> knotDerivative(knot.size() * numDir);
14   for(size_t i = 0; i < knot.size(); ++i) {
15     knotPrimal[i] = knot[i].getValue();
```

```
    for(size_t j = 0; j < numDir; ++j) {
      knotDerivative[i * numDir + j] = knot[i].getADValue(j);
    }
  }

  //...
  //perform the same extracting procedure for control points cp
  //...

  double tPrimal = (*t).getValue();
  std::vector<double> tDerivative(numDir);
  for(size_t j = 0; j < numDir; ++j) {
    tDerivative[j] = (*t).getADValue(j);
  }
  double dbvaluPrimal;
  double dbvaluDerivative(numDir, 0.0);

  const int numDirInt = static_cast<int>(numDir);
  dbvalu_dv_(&knotPrimal[0], &knotDerivative[0],
             &cpPrimal[0], &cpDerivative[0], &M, degree, deriv,
             &tPrimal, &tDerivative, &inbv, &work[0], &workd[0],
             &dbvaluPrimal, &dbvaluDerivative[0], &numDirInt);

  doublereal result = dbvaluPrimal;
  for(size_t j = 0; j < numDir; ++j) {
    result.setADValue(j,dbvaluDerivative[j]);
  }

  return result;
}
```

7.1.3 Parablading differentiation issues

The AD of Parablading involved a significant amount of code modification. Even after the successful compilation, the derivatives appeared to be incorrect when verifying against FD, which required a broad investigation of the complete source code to resolve the problems. This section briefly summarizes the difficulties faced upon Parablading differentiation.

7.1.3.1 ADOL-C integration issues

The *adouble* class cannot fit everywhere in the Parablading code without user's interaction. Some of the compile-time issues were related to:

- Standard output/string streams: the *adouble* class supports the standard C++ output stream `ofstream`, but not the standard C-style printing functions, e.g. `sprintf`, because C functions are not allowed to be overloaded. Furthermore, Parablading is developed using the *Qt* framework and when the *adouble* object is propagated to one of the *Qt*-classes, e.g. `QTextStream` or `QString`, a compile-time error occurs saying that such an operation is not defined for an operand of type *adouble*. Solution to all these issues is very simple and requires only to call the *getValue* method on the *adouble* object to extract its *primal* part. By extracting the *primal* part, one looses the derivative information, however in this case it is not even relevant to the output system, as e.g. the standard CAD output formats (STEP or IGES) do not require this information anyway.
- Type-casting: many places in the Parablading source code involve explicit/implicit conversion of the type *adouble* to primitive data-types, i.e. *int* and *float*. Although the type-casting could be overloaded in ADOL-C, it is omitted for a reason that *integer* and *float* variables do not carry along the derivative information. By doing such a type change, the chain rule gets disconnected in that part and the derivatives may be incorrect at the end. Therefore, the user needs to be aware of such situations and allow the conversion only if there is no risk of breaking the chain rule. In that case, the solution is simply to call the *getValue* method on the *adouble* object. However, the critical case where the type conversion produced wrong derivatives was encountered in the *BladeSection* class. The latter contains control point coordinates that serve to compute the TUB blade geometry as well as for the purposes of 3D-rendering with OpenGL. The control point coordinates are declared as arrays of *GLfloat* data-type which is the OpenGL alias for the *float* data-type. The first try was to leave the original declarations unchanged and allow the type-casting. However, the derivatives appeared to be all zeros at the end, which means that the chain rule was completely broken. Therefore, from the AD perspective,

such important entities should be treated as differentiable quantities. For this reason, the declaration types of all arrays were changed from *GLfloat* to *doublereal*. This modification caused many compile-time errors with classes that deal with visualization. They were resolved by introducing intermediate variables and breaking the chain rule only where it did not affect the derivative computation.

- Namespaces: common mathematical functions (e.g. `sin`, `exp`, etc.) have the prefix `std` (e.g. `std::sin`, `std::exp`, etc.) which causes compile-time errors once the *adoubles* are present. The common mathematical functions are overloaded in the `adtl` namespace of ADOL-C. To tackle this issue, one has to remove the `std` namespace from these functions (e.g. `std::sin` becomes `sin`) and add the corresponding `using` statements in the global header (e.g. `using std::sin`). In this way, both original and differentiated sources compile without errors, i.e. *Argument-Dependent Lookup* of C++ finds the appropriate functions based on their argument types (whether they are primitive types or *adoubles*).

7.1.3.2 Tapenade differentiation issue

In total there were 38 Fortran routines differentiated with Tapenade version *v3.12*, both in the *scalar* and *vector* mode variants. Only three of them are considered as the top-level routines since they are called from the C++ side. In almost all cases Tapenade generates correctly differentiated code. Here, only one example is explained in which the differentiation produced wrong output — the `dfspvn` routine from SLATEC library.

The part where a problem arises is shown in Listing 7.5. Line 2 introduces the `SAVE` statement that is used for variables that retain their values even after a certain routine finishes its execution. In other words, this is a way of declaring *static* variables in Fortran. The `DATA` statement (Line 3) initializes a variable with a particular value — in this case elements of `deltam` and `deltap` arrays are set to zero. If both `DATA` and `SAVE` statements are applied to the same variable, the `DATA` statement is executed only once during the initial routine call and ignored on any future calls, such that the static variables can keep their values.

Listing 7.5: dfspvn.f (SLATEC library)

```
DIMENSION DELTAM(20),DELTAP(20)
SAVE J, DELTAM, DELTAP
DATA J/1/,(DELTAM(I),I=1,20),(DELTAP(I),I=1,20)/40*0.0D0/
```

Instead of generating the same workflow for the corresponding AD quantities — `deltamd` and `deltapd` arrays — Tapenade performs incorrect differentiation, as shown in Listing 7.6. Rather than using the `DATA` statement to initialize the content of the `deltamd` and `deltapd` arrays, it initializes them to zero using the `DO` loop. In this way, their content will be erased on each subsequent routine call, i.e. the AD arrays loose their *static* property. The correct solution is shown in Listing 7.7.

Listing 7.6: dfspvn_d.f - incorrect differentiation

```
DIMENSION deltam(20),deltap(20)
DIMENSION deltamd(20),deltapd(20)
SAVE j, deltam, deltap
SAVE deltamd, deltapd
DATA j/1/,(deltam(i),i=1,20),(deltap(i),i=1,20)/40*0.0D0/
DO ii1=1,20
  deltapd(ii1) = 0.D0
ENDDO
DO ii1=1,20
  deltamd(ii1) = 0.D0
ENDDO
```

Listing 7.7: dfspvn_d.f - correct differentiation

```
DIMENSION deltam(20),deltap(20)
DIMENSION deltamd(20),deltapd(20)
SAVE j, deltam, deltap
SAVE deltamd, deltapd
DATA j/1/,(deltam(i),i=1,20),(deltap(i),i=1,20)/40*0.0D0/
DATA (deltamd(i),i=1,20), (deltapd(i),i=1,20) /40*0.0d0/
```

Both Tapenade *v3.12* and *v3.13* produced the same error with the differentiation of the `dfspvn` routine. However, the problem was reported to one of Tapenade main developers (Dr. Laurent Hascoët) and the corresponding patch is included in the next release.

7.1.3.3 Obstacles between AD and original workflow

After resolving all integration issues related to the AD tools, the derivatives computed with AD still appeared to be zero or much different than the ones evaluated with FD. This required further analysis of the parametrization workflow and a couple of places were found in the code that blocked AD sensitivity propagation.

Here, let us consider one issue that led the geometric sensitivities to be zero. It is related to a method that is used for setting a blade section parameter value, as shown in Listing 7.8. The method *setValue* has a conditional statement that checks whether the absolute difference between the current (`dValue`) and the new value is bigger than a certain tolerance (10^{-14}). Only when the tolerance criterion is satisfied, the parameter value is updated. Due to these circumstances, the method becomes an obstacle when computing sensitivities with AD.

For example, Listing 7.9 shows a code snippet to activate (in terms of AD) the axial shift parameter of a blade section. First, one retrieves a pointer to the *Parameter* object from the blade section (Line 2). From this object, it is necessary to extract the *adouble* variable which represents the parameter value (Line 3). The next step is to put the AD seed into the *adouble* object (Line 5 – Line 6). Finally, such an activated variable is placed back to the parameter object (Line 8) using the *setValue* method defined in Listing 7.8.

Since the *primal* value of the axial shift parameter is unaffected — AD does not impose any perturbations to the original parameters — the tolerance criterion of the *setValue* method (Listing 7.8) is not fulfilled and therefore the section parameter is actually not updated with the given AD seed. That is, the AD sensitivity propagation is blocked and the AD value of the axial shift remains 0 instead of 1. As a consequence, the blade construction algorithm is executed without any AD seeds which causes the geometric derivatives to be equal to zero.

Another similar case was detected in the code for setting the parameters of the blade inlet/exit angles, where the tolerance criterion was significantly larger (10^{-4}). All these barriers are imposed in the original code due to efficiency reasons, i.e. to prevent the blade reconstruction for small perturbations of the design parameters. Therefore, the differentiated Parablading code was additionally modified (simply by ignoring the tolerance criteria) in order to enable the sensitivity propagation.

Listing 7.8: Original code for setting parameter value

```cpp
void Parameter::setValue( const doublereal val )
{
  if (Abs(dValue-val) > 1e-14){
    // set current value
    dValue = val;
  }
}
```

Listing 7.9: Example of placing AD seed in design parameter

```cpp
//fetch parameter from section and extract its value
Parameter *axialShiftParam = bladeSection.parameter(parameterIndex);
doublereal axialShift = axialShiftParam->getValue();
//put AD seed (scalar mode)
double seed = 1.;
axialShift.setADValue(&seed);
//put activated axialShift back to parameter using the setValue method
axialShiftParam->setValue(axialShift);
```

When using FD, the aforementioned procedure for setting the design parameter is not considered to be a problem since the FD approach introduces a step size to the parameter value in order to compute a perturbed blade geometry. This step size is most likely bigger than 10^{-14} and therefore satisfies the tolerance criterion. However, it is important to note that for the second case, where the tolerance equals to 10^{-4}, one could obtain wrong sensitivities even with FD. That is, the chosen perturbation size has to be bigger than 10^{-4} in order to trigger the update procedure with respect to the blade inlet/exit angles.

7.1.4 Maintenance aspects

The proposed differentiation concept that combines the *typedef* approach together with the preprocessor directives enables straightforward maintenance of the differentiated sources, as both original and differentiated code are implemented on a single git repository branch. That is, any update to the primal code can be immediately validated if it works in the differentiated version just by activating the *AUTODIFF* flag. In this way, every member of the Parablading development team is involved in the differentiation process, meaning that the new features become available in the differentiated sources with a minimal effort.

The expected places in the code where compile-time errors may arise with further Parablading development are described in Sec 7.1.3. To fix them, one has to introduce additional preprocessor directives, however in most of the cases the solution is simple such as calling the *getValue* method on the *adouble* object to extract its primal part. Most likely the compile-time issues will be only related to the C++ side, i.e. the integration of *adoubles* in the new features, because the Fortran routines belong to a legacy code that is probably not going to change. For this reason, the differentiation with Tapenade and the mapping from C++ to Fortran (and vice-versa) is done only in the beginning. However, further differentiation of the Fortran routines and coupling with the C++ side will be required when switching to the reverse mode of AD for computing the derivatives.

An alternative to the proposed differentiation approach is to create separate branches that would serve only for the AD purposes. In this case, one has to maintain permanent parallel branches with respect to the master branch. Here, the preprocessor directives are not required any more. However, it would be helpful to use the alias *doublereal* in all branches, even though it would correspond to the *double* data-type in the original sources. This would help to reduce the difference between the master and AD branches and also the amount of conflicts that one will face during the merging process. This concept is useful when only a few persons maintain the differentiated versions instead of the whole team.

7.2 Verification of differentiated Parablading

7.2.1 Geometric derivative validation

The correctness of the computed derivatives is verified using the TUB blade parametric model described in Sec. 2.2. As a representative example, the derivatives of the surface point coordinates with respect to axial shift of the 8th blade section are compared. A quantitative comparison of derivatives is presented in Table 7.1 and shows mutual agreement. Furthermore, as hinted by Fig. 7.2, the overall magnitude plots for the same design parameter match to a very high extent.

The same surface sensitivities are also verified with the Taylor test (presented in Eq. (3.1)). The Taylor test was performed on a number of arbitrary surface points

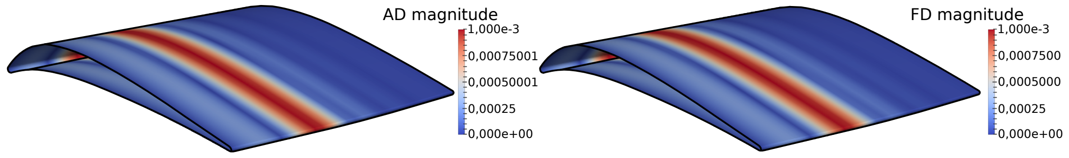


Figure 7.2: TUB surface sensitivities evaluated with AD (left) and FD (right)

Table 7.1: AD and FD values comparison for several TUB surface point coordinates

AD value	FD value	Abs. difference
-8.36954**5**e-09	-8.36954**6**e-09	1.07e-15
-2.2943**42**e-06	-2.2943**51**e-06	9.02e-12
-1.01146**3**e-05	-1.01146**5**e-05	1.88e-11
9.5689**27**e-04	9.5689**59**e-04	3.20e-09
⋮	⋮	⋮
1.491**902**e-04	1.492**034**e-04	1.32e-08*

*Maximal difference

with a range of step sizes $h \in [10^1,\ 10^{-6}]$ taken in millimeters. The error plots (the left-hand side of Eq. (3.1)) in eight surface point coordinates are presented in Fig. 7.3. Here one observes even slightly better convergence than the theoretical convergence rate of h^2. This behavior continues until $h \in [10^{-2},\ 10^{-4}]$ (the axial shift parameter influences differently the chosen surface point coordinates), where the errors reach machine precision.

7.2.2 Performance test

Performance of the differentiated Parablading sources is measured using the code for the actual optimization workflow that computes 33,000 surface points (x, y, z) and their corresponding derivatives. Table 7.2 presents quantitative comparisons of average timings (based on 10 measurements) and run-time ratios (t_{ad}/t_{primal}), where the derivatives are computed in 1 direction (*scalar* mode) and 196 directions (*vector* mode).

According to the theory [GW08], the run-time ratio between the derivative compu-

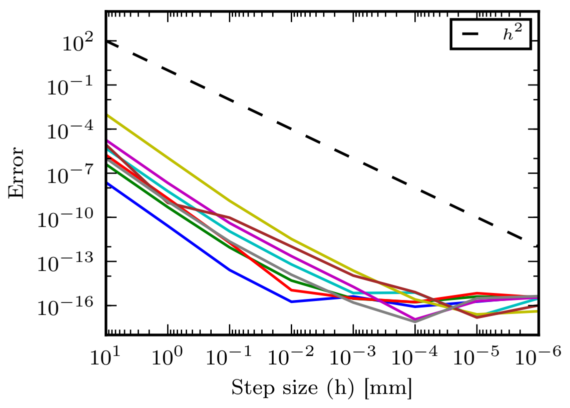


Figure 7.3: Taylor test for eight TUB surface point coordinates

Table 7.2: Original vs. differentiated sources with the number of directions $p = 1$ (*scalar* mode) and $p = 196$ (*vector* mode)

	Primal	Scalar-forward AD	Vector-forward AD ($p = 196$)
Avg. time [s]	8.12	36.17	1929.30
Run-time ratio		4.45	237.52 $\Rightarrow$ 1.21 per direction

Timings are averaged on 10 measurements.

tation in the forward mode of AD and the function (primal) evaluation should be in the range $[1 + p, 1 + 1.5p]$, where p is the number of directions. Comparing this with the results in Table 7.2, one observes that the *scalar* mode exceeds the upper theoretical limit of 2.5, while the *vector* mode satisfies the theoretical expectations. The reason for slower performance of the *scalar* mode is that the theoretical expectations do not assume memory copying between one AD tool to another, as it was the case here due to ADOL-C and Tapenade linking.

As mentioned in Sec. 7.1.3.2, there are three top-level Fortran routines for which the *C++ to Fortran* interface exists. They serve for: (i) computing points on a B-spline curve and (ii) fitting a B-spline curve to a given data-set. For each blade section, the top-level Fortran routines are executed more than ten times. Considering that the total number of sections is 21, one can notice that there are hundreds of memory copying operations involved just for a single blade construction. Such an overhead justifies the slower performance of the *scalar* mode. However, this pays-off when using the *vector* mode due to compiler optimization.

An overview of all run-time ratios evaluated in the range of 1 to 196 directions together with the memory requirements with respect to the maximal number of directions is shown in Fig. 7.4. Additionally to the AD run-time ratios, the run-time ratios for forward and central FD (2^{nd} order) are also shown. As one can notice, the *traceless* forward mode of AD performs accordingly to the theoretical expectations. For this particular test-case, it is slower than forward FD, but much more efficient than central FD that is typically used in the Rolls-Royce optimization workflow.

To evaluate the peak memory requirements of the *primal* and the vector forward mode of AD with the vector size $p = 196$, the Parablading has been profiled with the profiling tool *Massif.* The memory requirements are compared in Fig. 7.4. To compute a rough estimate of the required memory b_A per vector entry in the differentiated sources, one can apply the following expression:

$$\begin{aligned} & b_N + b_A + b_A * p = m_{AD} \\ \Leftrightarrow \quad & b_A = (m_{AD} - (b_N + b_A))/p \, , \end{aligned}$$

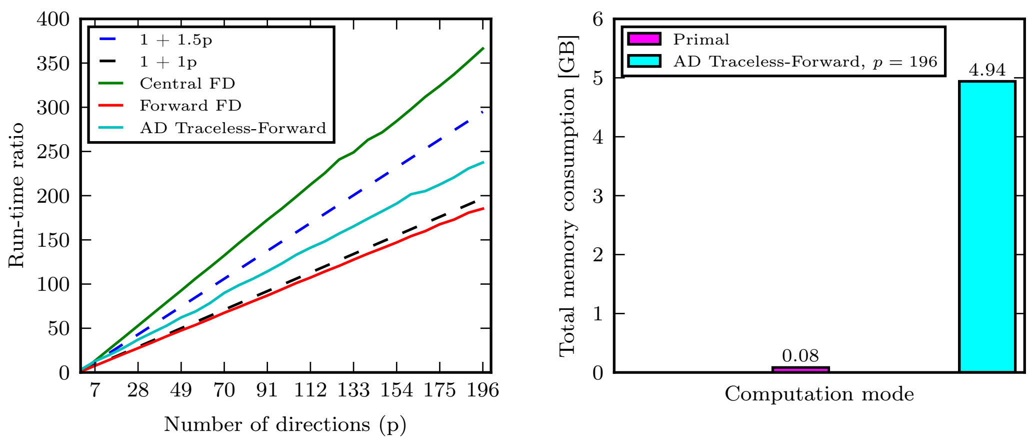


Figure 7.4: Summary of run-time ratios (left) and total memory requirements (right) for TUB test-case

where b_N is the base memory overhead of non-active data, b_A is the base memory of active data and m_{AD} is the total memory required by the differentiated sources (4.94 GB). Considering that the primal memory $(b_N + b_A)$ is 0.08 GB and p equals to 196, one retrieves that b_A equals to 0.0248 GB.

The run-time ratio graph in Fig. 7.4 also shows that the high overhead caused by the scalar forward mode of AD is no longer visible at $p = 28$. It means that one could also run the differentiated sources 7 times while setting at each step a different set of design parameters as the independent variables of the system. In this way, the Jacobian matrix is computed in blocks of 28 rows. Every run of the differentiated sources would require approximately 0.77 GB of memory. That is, the total memory required by the differentiated sources would be reduced while keeping almost the same performance as running the vector forward mode of AD in 196 directions.

7.3 Gradient-based optimization of TU Berlin stator

7.3.1 Objective functions

The gradient-based optimization of the baseline stator geometry is performed with respect to two criteria. The first objective to be minimized is the total pressure loss between inlet and outlet, defined by the loss coefficient:

$$\omega = \frac{p_{t,I} - p_{t,E}}{p_{t,I} - p_{s,I}} \times 100\%$$

where p_s and p_t denote the static and total pressure computed using mass-averaging over the corresponding cross section at inlet (I) and exit (E) of the CFD domain. The second objective is the flow angle deviation from the axial direction at the stator outlet A_E, which takes into account both the circumferential and radial components:

$$\alpha_E = \sqrt{\frac{\int\limits_{A_E} \dot{m}\alpha^2 \,\mathrm{d}A_E}{\int\limits_{A_E} \dot{m}\,\mathrm{d}A_E}} \quad , \quad \alpha = \cos^{-1}\left(\frac{\vec{V}\cdot\vec{i}}{|\vec{V}|}\right) = \cos^{-1}\left(\frac{u}{\sqrt{u^2+v^2+w^2}}\right)$$

where u, v, w denote the velocity components and $\dot{m}$ the mass flow rate. Only design point operating conditions are considered in this case, with a uniform inlet whirl angle profile of 42°.

These two objectives are contradicting since the higher turning — which is required here to achieve a more axial outflow — produces more losses. Normally, this would lead to a multi-objective optimization problem. However, since gradient-based optimizers can deal with single-objective optimization problems, an augmented objective function to be minimized is defined using the weighted sum approach: $F_{aug} = \omega + c\,\alpha_E$, where the weight $c = 1$ (this value was chosen by Ilias Vasilopoulos after conducting preliminary studies).

7.3.2 CFD setup and optimization workflow

To build the optimization workflow, the following Rolls-Royce in-house tools were used:

- PADRAM meshing tool [SL03] — generates a block-structured 1.9 million node hexahedral mesh.
- HYDRA primal solver [Lap04] — a steady RANS compressible flow solver with the Spalart-Allmaras turbulence model.
- HYDRA adjoint solver [Gil02] — a discrete adjoint solver which provides the volumetric sensitivities. These sensitivities are then projected onto the surface of the stator using the inverse operation of a spring-based mesh deformation algorithm.

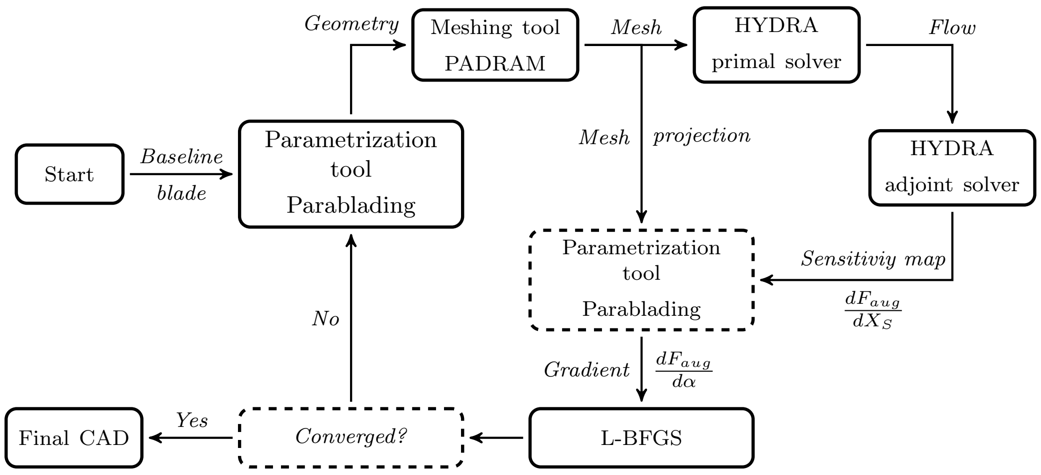


Figure 7.5: Gradient-based optimization workflow

The automated gradient-based optimization workflow is presented in Fig. 7.5. First, the mesh is generated using the given baseline geometry and provided to the primal solver. The primal solver performs the flow simulation until convergence criteria are met. Subsequently, the adjoint solver reads in the flow solution and executes the two adjoint simulations in parallel (one for ω and one for α_E), which result in the mesh sensitivities $\left(\frac{d\omega}{dX_S}, \frac{d\alpha_E}{dX_S}\right)$. These sensitivity components are added to compute the augmented sensitivity $\left(\frac{dF_{aug}}{dX_S}\right)$, following the same weighted sum expression as for the objective functions.

Once this process is completed, the differentiated Parablading computes the geometric sensitivity $\left(\frac{dX_S}{d\alpha}\right)$ for each design parameter using the forward vector mode of AD. Furthermore, it reads in the augmented sensitivity $\left(\frac{dF_{aug}}{dX_S}\right)$ and computes the total gradient:

$$\frac{dF_{aug}}{d\alpha} = \frac{dF_{aug}}{dX_S}\frac{dX_S}{d\alpha} \ .$$

Although the computational cost for the geometry part scales with the number of design parameters, the total cost for the 192 parameters (obtained in the forward mode of AD) is negligible compared to the cost of the primal and adjoint solutions of the flow simulation. Nevertheless, it would be beneficial to integrate the reverse mode of AD into the geometry part such that the computational cost does not scale with the number of design parameters α. Moreover, the reverse mode of AD would reduce the temporal complexity of the derivative computation in this case because there is only one output function F_{aug} opposite to 192 inputs (α).

Finally, the gradient is passed to an L-BFGS optimizer which updates the design space. The whole process is wrapped in Python and is repeated until the optimization convergence criteria are met. A similar optimization has been performed in [VFM17], where the geometric sensitivities were computed using 2^{nd} order finite differences.

7.3.3 Optimization results

The gradient-based optimization converged after 22 *BFGS* steps and yields 14.37% reduction of the augmented objective function F_{aug} (in blue), as shown in Fig. 7.6. This result was mainly affected by the minimization of α_E (in green), which is remarkably reduced by 42.84%. On the other hand, the contradicting objective ω (in red) is only decreased by 0.45%. For this particular configuration of the augmented objective, where the weighting factor $c = 1$, the optimizer chooses to compromise with respect to the total pressure loss in order to massively reduce the flow deflection. The final outcome of decreasing the exit angle deviation from the axial direction about $2°$, while keeping the total pressure loss practically constant, is clearly beneficial for this industrial application.

Figure 7.7 compares the baseline and optimal stator blade geometry. One can notice that the optimization resulted with a leaned blade, where the S-shaped trailing edge is significantly responsible for reducing the exit flow angle deviation. This can be

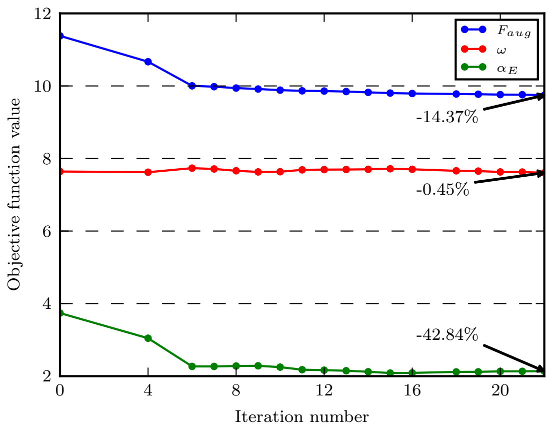


Figure 7.6: Optimization history

seen in Fig. 7.8, where the distribution of the whirl angle at the exit plane of the CFD domain is plotted for both baseline and optimum blades. The optimal stator blade obtains an average whirl angle much closer to zero.

7.4 Summary

The AD of Parablading CAD tool has been performed by combining two different AD concepts in the source code which is developed both in C++ and Fortran. The C++ sources are differentiated with the operator-overloading AD tool ADOL-C, while the Fortran sources are differentiated with the source-transformation AD tool Tapenade. There are two versions of the differentiated sources to support the derivative computation both in the *scalar* and *vector* forward mode of AD.

The issues faced upon the differentiation are discussed together with the corresponding solutions. Even the original parametrization workflow was slightly modified to ensure the correct propagation of sensitivities. The geometric derivatives computed with AD were successfully verified against FD.

All code modifications imposed by the differentiation are wrapped with the preprocessor directives, to enable easy switching between the original and AD sources. The

[19]Picture provided by Ilias Vasilopoulos

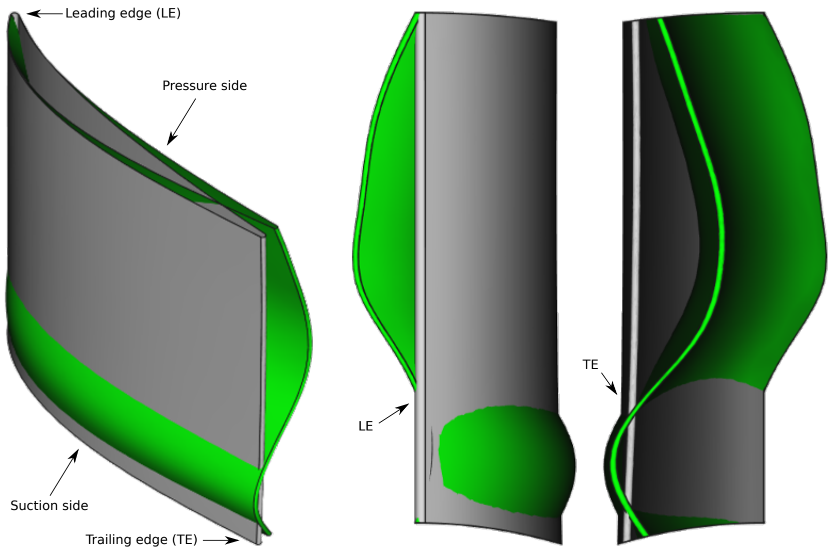


Figure 7.7: Baseline (grey) vs. optimal (green) geometry

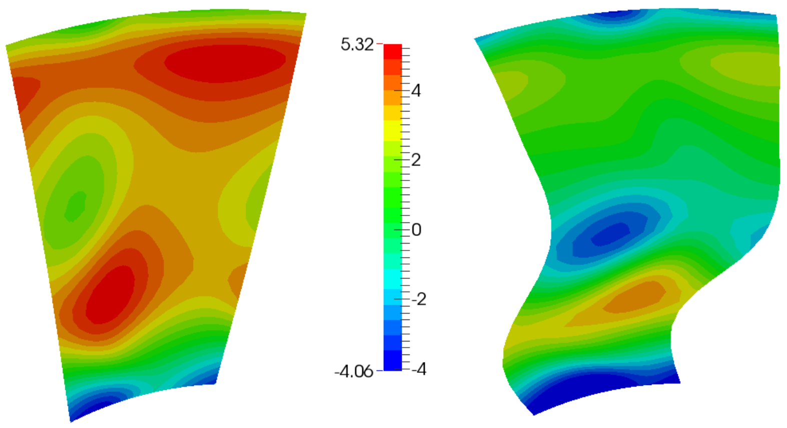


Figure 7.8: Exit whirl angle distribution for baseline (left) and optimum (right) blade[19]

maintenance aspects of this approach are described.

The differentiated Parablading tool has been coupled with the HYDRA adjoint CFD solver, that is also produced by AD. This work demonstrates that AD can be entirely integrated into an industrial environment.

This differentiated design chain is used to perform the aerodynamic optimization of the TUB stator blade. Two contradicting terms: the total pressure loss (ω) and the exit whirl angle distribution (α_E), are joined using the weighted sum approach to the single-objective function. This augmented objective function, together with its corresponding gradients, is provided to the L-BFGS optimizer that converged after 22 iterations. The optimization was highly influenced by α_E that was reduced by 42.84%, while keeping the contradicting objective ω almost constant (0.45% reduction).

8 Conclusion

Gradient-based optimization methods are recognized for their computational efficiency, especially when considering test-cases with a large number of design variables. Over the past decades, adjoint methods have emerged as the most effective methods to compute gradients in CFD codes since they can compute the gradients with respect to an arbitrary number of design parameters at near-constant computational cost similar to the *primal* evaluation. Coupling adjoint CFD approaches with a geometrical parameterization of a component — usually provided in a CAD framework — enables optimization based on the complete design chain, but includes additional challenging aspects. To obtain a gradient for the complete design chain, one requires the computation of shape (or geometric) sensitivities with respect to its parametrization. However, this information is usually not provided within a CAD system/tool. Typically, it is computed with rather inaccurate FD. This gap, i.e. the AD of the involved CAD libraries, is closed by the results presented in this work, which are suitable to a wide variety of applications.

Chapter 3 presented the AD of the open-source CAD kernel OCCT *v7.0* using the AD software tool ADOL-C, however requiring a significant amount of code modification. This achievement represents the first example of applying AD to a general-purpose CAD library. The correctness of the computed derivatives has been verified using the parametric models of the U-bend cooling duct and the TUB *TurboLab* stator (described in Chapter 2), but the derivative validation of the rest of differentiated OCCT sources still remains challenging.

The OCCT kernel offers the *Automated Testing System* that consists of thousands of tests to validate its *primal* functionality, from which the derivative tests could be implemented. However, doing such a job manually would require a tremendous amount of work hours. An unanswered challenge to the AD community is how to develop an automated workflow that would derive relevant derivative regression tests from the existing primal tests.

The reverse mode differentiation of OCCT yields a significant reduction in the temporal complexity of the derivative computation. Compared to the *traceless* vector forward mode of AD, one benefits from an improved efficiency by: (i) 63% for the U-bend test-case (96 design variables) and (ii) 59.4% for the TUB stator test-case (184 design variables).

Since ADOL-C is integrated into OCCT by the *typedef* approach, its new functionality *activity analysis* was employed and tested with the differentiated OCCT in order to discover *adoubles* that should not be treated as differentiable quantities. That is, any *adouble* found in the computational graph which does not depend on the independent variables is treated as a constant. This feature reduces the amount of information stored on the *trace* and improves the performance of the reverse differentiated OCCT. Its performance was evaluated using the TUB stator test-case. Once the activity analysis is enabled, one benefits from an improved efficiency by 22.5% in comparison to the original reverse mode of AD.

The differentiated OCCT has been coupled with the discrete adjoint CFD solver STAMPS, that is also produced with AD, as elaborated in Chapter 4. This achievement demonstrates for the first time the differentiation of a complete design chain built from generic, multi-purpose components, which can be applied to a very wide variety of shape parametrizations expressed in CAD. This chain is utilized to perform the gradient-based shape optimization of the U-bend and TUB stator test-cases, to minimize the total pressure losses. The optimization of the U-bend duct yields 18% reduction of the objective function. Regarding the TUB stator test-case, the optimization was executed two times with respect to the low-fidelity and high-fidelity CFD simulations, reducing the total pressure loss between the inlet and the outlet by 13.72% and 6.7%, respectively.

In Chapter 5, the previously developed gradient-based optimization workflow was expanded in order to support assembly constraints of the TUB stator. In particular, the blade has to accommodate four mounting bolts (cylinders) during the shape optimization (two on the hub and two on the shroud). The cylinders are allowed to move, however the minimum axial distance between the hub and shroud pairs has to be 60 mm. For this purpose, ten inequality geometric constraints were implemented to ensure the fitting of cylinders inside the volume of the optimal blade. The whole optimization workflow is wrapped in Python and driven by the SLSQP optimizer, which handles the inequality constraints using the derivative information provided by the differentiated OCCT. The optimization yields 12.3% reduction of the objective function when employing the low-fidelity CFD simulation and 14% reduction with respect to the high-fidelity CFD simulation, while respecting all constraints.

This achievement demonstrates that the differentiation of a complete CAD kernel is feasible and can be applied to industrial cases. The last two chapters are dedicated to the AD of specialized turbo-machinery CAD tools.

First, improved AD of the VKI in-house CAD and mesh generation tool CADO (Chapter 6) was achieved by exploiting the structure of the 2-D mesh smoothing process. The original 'black-box' reverse mode of ADOL-C required a large amount of memory to store the *trace* (more than 36 GB). For this reason, the differentiated CADO sources were modified to split the computation on three smaller *traces*, where the middle *trace* represents a single mesh smoothing iteration and can be re-evaluated many times with ADOL-C drivers to compute the final smoothed mesh. In addition to the *trace* structuring, two checkpointing techniques (equidistant and binomial) were employed to store snapshots during the execution of the differentiated sources, such that the gradients evaluated with the reverse mode can be computed more efficiently. These upgrades yield a substantial reduction of the memory requirements in comparison with the black-box reverse AD. The performance was measured using the two types of LS89 axial turbine parametrization. When considering the parametrization with 22 design variables, the structured AD with the binomial checkpointing reduces the memory consumption from the original 36.14 GB to 1.78 GB (by 95%). Next, when considering the finer parametrization with 230 design variables, the structured AD with the binomial checkpointing reduces the memory footprint from the original 42.06 GB to 7.72 GB (by 82%). Moreover, its performance is significantly faster than any other differentiation mode used in CADO.

Second, the Rolls-Royce in-house airfoil design and blade generation tool Parablading is differentiated using the AD software tools ADOL-C and Tapenade (Chapter 7). The two different AD concepts have been successfully integrated into the Parablading source code, however imposing a lot of changes with respect to the original sources. Even a few places in the original parametrization workflow were found that blocked the propagation of AD sensitivities. All differentiation issues have been resolved and the geometric derivatives computed with AD have been successfully verified against FD.

The code changes imposed upon AD were wrapped with preprocessor directives for conditional compilation such that the user can easily switch between the original and differentiated functionality (and vice-versa) just by changing a single flag. If such coding style could be integrated into a master development branch, it would allow simpler maintenance of the differentiated sources.

The differentiated Parablading tool has been coupled with the HYDRA adjoint CFD solver, that is also produced with AD, thus providing a complete differentiated design chain at hand. This work demonstrates that AD can be entirely integrated into an industrial environment.

This differentiated design chain was demonstrated on the gradient-based optimization of the TUB stator. Two contradicting terms: total pressure loss (ω) and exit whirl angle distribution (α_E), were joined using a weighted sum approach to a single-objective function. This objective, together with its corresponding gradients, was provided to the L-BFGS optimizer that converged after 22 iterations. The optimization was highly influenced by α_E that was reduced by 42.84%, while keeping the contradicting objective ω almost constant (0.45% reduction).

This thesis demonstrates the feasibility of applying AD to the CAD frameworks. The involvement of CAD systems/tools in the design chain avoids the effort and errors associated with the CAD-free (mesh-based) methods when transforming the optimal mesh back to the CAD format. That is, the developed CAD-based workflows in this work keep the design tool into the optimization loop, therefore providing the optimal shape in the CAD format that is convenient for further analysis in a multi-disciplinary environment and finally manufacturing. Application of AD to the CAD rather than FD not only produces exact shape sensitivities (up to the machine-precision), but also reduces its computational cost, especially when employing the reverse mode of AD. These advancements enable a step change in industrial shape optimization with the gradient-based methods.

8.1 Future work and research direction

In this study, the OCCT kernel *v7.0* is differentiated and this version was taken in 2015 when the IODA ITN started. Since then, there has been a large amount of updates in the OCCT sources. Unfortunately, they are not included in the AD version, which is therefore slowly becoming outdated. For this reason, the author is starting with the differentiation of the latest version — currently *v7.3*. In the meantime, the *typedef* approach (together with the preprocessor directives) to differentiate the sources is going to be discussed with Open CASCADE developers to find an agreement about maintaining the differentiated version such that it regularly reflects the updates from the master branch. A long-term strategy could be to make the AD efforts open-source, such that the OCCT community can contribute to its development. Definitely, the AD of OCCT is not a one-person job when preferring to catch-up with the regular updates on the original sources.

Another challenge is to develop an automated testing suite for validating the derivatives from the existing *primal* regression tests. The *Automated Testing System* of OCCT involves thousands of tests written in Tcl that execute C++ functionality in the background. The first step would be to identify which tests are beneficial for the derivative validation. Next, the set of Tcl capabilities has to be extended in order to support additional operations such as defining the independent and dependent variables of the system as well as getting and setting their AD values.

As mentioned in Sec. 3.5.1, there is one source package (*AdvApp2Var*) in OCCT that could not be differentiated with ADOL-C. The sources of this package are actually a legacy code that was firstly written in Fortran and later translated to C. They involve a complex pointer arithmetic (low-level memory management) that results in a memory corruption of *adouble* objects. Either one could rewrite the package such that it follows the C++ standards for memory management or differentiate it with Tapenade which would require additional coupling efforts between ADOL-C and Tapenade.

The Rolls-Royce in-house CAD tool Parablading is differentiated using the AD tools ADOL-C and Tapenade, however the geometric derivatives are computed only with the forward mode of AD. The following step is to integrate the reverse mode of AD to benefit from an improved efficiency.

The knowledge gained by the AD of CAD libraries investigated in this work could be applied to the differentiation of commercial (closed-source) CAD systems. Since the adjoint CFD methods are widely adopted in industrial gradient-based optimization workflows, the next advancement is expected to be in efficient and exact calculation of CAD gradients, not only in academia but also in industry.

Bibliography

[ABM$^+$16] S. Auriemma, M. Banović, O. Mykhaskiv, H. Legrand, J.-D. Müller, and A. Walther. Optimisation of a U-bend using CAD-based adjoint method with differentiated CAD kernel. In *ECCOMAS Congress*, 2016.

[ABW$^+$18] S. Auriemma, M. Banović, A. Walther, O. Mykhaskiv, and J.-D. Müller. Applications of differentiated CAD kernel in gradient-based aerodynamic shape optimisation. In *2018 Joint Propulsion Conference*. American Institute of Aeronautics and Astronautics, 2018.

[ALR90] T. Arts, M. Lambertderouvroit, and A. W. Rutherford. Aero-thermal investigation of a highly loaded transonic linear turbine guide vane cascade. A test case for inviscid and viscous flow computations. *NASA STI/Recon Technical Report N*, 91, 1990.

[ARA18a] D. Agarwal, T. T. Robinson, and C. G. Armstrong. A CAD Based Framework for Optimizing Performance While Ensuring Assembly Fit. In *Recent Advances in Intelligent Manufacturing*, pages 73–83. Springer, 2018.

[ARA$^+$18b] D. Agarwal, T. T. Robinson, C. G. Armstrong, S. Marques, I. Vasilopoulos, and M. Meyer. Parametric design velocity computation for CAD-based design optimization using adjoint methods. *Engineering with Computers*, 34(2):225–239, 2018.

[Bac96] T. Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford university press, 1996.

[BF10] D. Bestle and P. Flassig. Optimal aerodynamic compressor blade design considering manufacturing noise. In *8th Association for Structural and Multidisciplinary Optimization in the UK/International Society for Structural and Multidisciplinary Optimization (ASMO-UK/ISSMO) Conference on Engineering Design Optimization, London, July*, pages 8–9, 2010.

[BFJP87] R. E. Barnhill, G. Farin, M. Jordan, and B. R. Piper. Surface/surface intersection. *Computer Aided Geometric Design*, 4(1-2):3–16, 1987.

[BJMS95] C. H. Bischof, W. T. Jones, A. Mauer, and J. Samareh. Application of automatic differentiation to 3-D volume grid generation software. *CFD for Design and Optimization*, 232:17–22, 1995.

[BKMC96] C. H. Bischof, P. Khademi, A. Mauer, and A. Carle. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering*, 3(3):18–32, 1996.

[Blo00] F. J. Blom. Considerations on the spring analogy. *International journal for numerical methods in fluids*, 32(6):647–668, 2000.

[BMA$^+$18] M. Banović, O. Mykhaskiv, S. Auriemma, A. Walther, H. Legrand, and J.-D. Müller. Algorithmic differentiation of the Open CASCADE Technology CAD kernel and its coupling with an adjoint CFD solver. *Optimization Methods and Software*, pages 1–16, 2018.

[BRMO97] C. H. Bischof, L. Roh, and A. J. Mauer-Oats. ADIC: an extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience*, 27(12):1427–1456, 1997.

[BVWM19] M. Banović, I. Vasilopoulos, A. Walther, and M. Meyer. Algorithmic differentiation of an industrial airfoil design tool coupled with the adjoint CFD method. *Optimization and Engineering*, Nov 2019.

[CJM11] F. Christakopoulos, D. Jones, and J.-D. Müller. Pseudo-timestepping and verification for automatic differentiation derived CFD codes. *Computers & Fluids*, 46(1):174–179, 2011.

[CSV09] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization.* Society for Industrial and Applied Mathematics, 2009.

[CVB$^+$13] F. Coletti, T. Verstraete, J. Bulle, T. Van der Wielen, N. Van den Berge, and T. Arts. Optimization of a U-Bend for Minimal Pressure Loss in Internal Cooling Channels - Part II: Experimental Validation. *Journal of Turbomachinery*, 135(5):051016, 2013.

[DB77] C. De Boor. Package for calculating with B-splines. *SIAM Journal on Numerical Analysis*, 14(3):441–472, 1977.

[DBVdSB06] A. De Boer, M. S. Van der Schoot, and H. Bijl. New method for mesh moving based on radial basis function interpolation. In *ECCOMAS CFD 2006: Proceedings of the European Conference on Computational Fluid Dynamics, Egmond aan Zee, The Netherlands, September 5-8, 2006*. Delft University of Technology; European Community on Computational Methods in Applied Sciences (ECCOMAS), ECCOMAS, 2006.

[DH15] J. Dannenhoffer and R. Haimes. Design Sensitivity Calculations Directly on CAD-based Geometry. *53rd AIAA Aerospace Sciences Meeting, AIAA SciTech Forum*, 2015. AIAA 2015-1370.

[Dwi09] R. P. Dwight. Robust Mesh Deformation using the Linear Elasticity Equations. In H. Deconinck and E. Dick, editors, *Computational Fluid Dynamics 2006*, pages 401–406, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[FC12] D. Falck and B. Collette. *FreeCAD [How-To]*. Packt Publishing Ltd., 2012.

[GDMP03] M. B. Giles, M. C. Duta, J.-D. Müller, and N. A. Pierce. Algorithm developments for discrete adjoint methods. *AIAA journal*, 41(2):198–205, 2003.

[Gil02] M. B. Giles. On the iterative solution of adjoint equations. In *Automatic Differentiation of Algorithms*, pages 145–151. Springer New York, 2002.

[Gun02] M. Gunzburger. *Perspectives in Flow Control and Optimization*. Society for Industrial and Applied Mathematics, 2002.

[GW00] A. Griewank and A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Society for Industrial Mathematics, 2nd edition, 2008.

[GWMW07] N. R. Gauger, A. Walther, C. Moldenhauer, and M. Widhalm. Automatic differentiation of an entire design chain for aerodynamic shape optimization. In *New Results in Numerical and Experimental Fluid Mechanics VI*, pages 454–461. Springer, 2007.

[HD13] R. Haimes and J. Dannenhoffer. The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry. *21st AIAA Computational Fluid Dynamics Conference, Fluid Dynamics and Co-located Conferences*, 2013. AIAA 2013-3073.

[HH78] R. M. Hicks and P. A. Henne. Wing design by numerical optimization. *Journal of Aircraft*, 15(7):407–412, 1978.

[HP13] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3):20:1–20:43, May 2013.

[IQ13] E. Iuliano and D. Quagliarella. Proper Orthogonal Decomposition, surrogate modelling and evolutionary optimization in aerodynamic design. *Computers & Fluids*, 84:327–350, 2013.

[JA07] S. Jakobsson and O. Amoignon. Mesh deformation using radial basis functions for gradient-based aerodynamic shape optimization. *Computers & Fluids*, 36(6):1119–1136, 2007.

[Jam89] A. Jameson. Aerodynamic design via control theory. In *Recent advances in computational fluid dynamics*, pages 377–401. Springer, 1989.

[JM08] A. Jaworski and J.-D. Müller. Toward modular multigrid design optimisation. In C. Bischof and J. Utke, editors, *Lecture Notes in Computational Science and Engineering*, volume 64, pages 281–291, "New York, NY, USA", 2008. Springer.

[JV00] A. Jameson and J. C. Vassberg. Studies of alternative numerical optimization methods applied to the brachistochrone problem. *Computational Fluid Dynamics Journal*, 9(3):281–296, 2000.

[Ken10] J. Kennedy. Particle swarm optimization. *Encyclopedia of machine learning*, pages 760–766, 2010.

[Lap04] L. Lapworth. Hydra-CFD: A Framework for Collaborative CFD Development. In *International Conference on Scientific and Engineering Computation (IC-SEC)*, 2004.

[MBA$^+$18] O. Mykhaskiv, M. Banović, S. Auriemma, P. Mohanamuraly, A. Walther, H. Legrand, and J.-D. Müller. NURBS-based and parametric-based shape optimization with differentiated CAD kernel. *Computer-Aided Design and Applications*, pages 1–11, 2018.

[MGX$^+$16] J.-D. Müller, M. Gugala, S. Xu, J. Hückelheim, P. Mohanamuraly, and O. R. Imam-Lawal. Introducing STAMPS: an open-source discrete adjoint CFD solver using source-transformation AD. In *11th ASMO UK/ISSMO/NOED2016: International Conference on Numerical Optimisation Methods for Engineering Design*, 2016.

[MHM18] J.-D. Müller, J. Hückelheim, and O. Mykhaskiv. STAMPS: a Finite-Volume Solver Framework for Adjoint Codes Derived with Source-Transformation AD. In *2018 Multidisciplinary Analysis and Optimization Conference*, 2018.

[MV] J.-D. Müller and T. Verstraete. AboutFlow benchmark test-case: TU Berlin TurboLab stator. `http://aboutflow.sems.qmul.ac.uk/events/munich2016/benchmark/testcase3/`. Accessed 23 October 2018.

[MV17] L. Mueller and T. Verstraete. CAD integrated multipoint adjoint-based optimization of a turbocharger radial turbine. *International Journal of Turbomachinery, Propulsion and Power*, 2(3):14, 2017.

[Nau12] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2012.

[NS07] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, pages 89–100. ACM, 2007.

[Ope] Open CASCADE. https://www.opencascade.com/. Accessed 17 July 2017.

[PH18] V. Pascual and L. Hascoët. Mixed-language automatic differentiation. *Optimization Methods and Software*, 33(4-6):1192–1206, 2018.

[Pir74] O. Pironneau. On optimum design in fluid mechanics. *Journal of Fluid Mechanics*, 64(1):97–110, 1974.

[RAC$^+$12] T. T. Robinson, C. G. Armstrong, H. S. Chua, C. Othmer, and T. Grahs. Optimizing Parameterized CAD Geometries Using Sensitivities Based on Adjoint Functions. *Computer-Aided Design and Applications*, 9(3):253–268, 2012.

[SL03] S. Shahpar and L. Lapworth. PADRAM: Parametric design and rapid meshing system for turbomachinery optimisation. In *ASME Turbo Expo*, 2003.

[SP86] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. *ACM SIGGRAPH computer graphics*, 20(4):151–160, 1986.

[SS79] J. L. Steger and R. L. Sorenson. Automatic mesh-point clustering near a boundary in grid generation with elliptic partial differential equations. *Journal of Computational Physics*, 33(3):405–410, 1979.

[ST19] I. Sanchez Torreguitart. *Efficient CAD based adjoint optimization of turbomachinery using an adaptive shape parameterization.* PhD thesis, 05 2019.

[STVM16] I. Sanchez Torreguitart, T. Verstraete, and L. Mueller. CAD Kernel and Grid Generation Algorithmic Differentiation for Turbomachinery Adjoint Optimization. In *ECCOMAS Congress*, 2016.

[STVM18] I. Sanchez Torreguitart, T. Verstraete, and L. Mueller. Optimization of the LS89 axial turbine profile using a CAD and adjoint based approach. *International Journal of Turbomachinery, Propulsion and Power*, 3(3):20, 2018.

[TSW98] J. F. Thompson, B. K. Soni, and N. P. Weatherill. *Handbook of grid generation*. CRC press, 1998.

[TTM74] J. F. Thompson, F. C. Thames, and C. W. Mastin. Automatic numerical generation of body-fitted curvilinear coordinate system for field containing any number of arbitrary two-dimensional bodies. *Journal of computational physics*, 15(3):299–319, 1974.

[VCB$^+$13] T. Verstraete, F. Coletti, J. Bulle, T. Vanderwielen, and T. Arts. Optimization of a U-Bend for Minimal Pressure Loss in Internal Cooling Channels - Part I: Numerical Method. *Journal of Turbomachinery*, 135(5):051015, 2013.

[Ver] T. Verstraete. AboutFlow benchmark test-case: VKI U-bend. `http://aboutflow.sems.qmul.ac.uk/events/munich2016/benchmark/testcase1/`. Accessed 21 February 2017.

[Ver10] T. Verstraete. CADO: a computer aided design and optimization tool for turbomachinery applications. In *2nd Int. Conf. on Engineering Optimization, Lisbon, Portugal, September 6-9*, 2010.

[VFM17] I. Vasilopoulos, P. Flassig, and M. Meyer. CAD-based Aerodynamic Optimization of a Compressor Stator using Conventional and Adjoint-driven Approaches. In *ASME Turbo Expo*, 2017.

[VH82] W. H. Vandevender and K. H. Haskell. The SLATEC Mathematical Subroutine Library. *SIGNUM Newsl.*, 17(3):16–21, September 1982.

[WB09] J. Witteveen and H. Bijl. *Explicit Mesh Deformation Using Inverse Distance Weighting Interpolation*. 2009.

[WG04] A. Walther and A. Griewank. Advantages of Binomial Checkpointing for Memory-reduced Adjoint Calculations. In M. Feistauer, V. Dolejší, P. Knobloch, and K. Najzar, editors, *Numerical Mathematics and Advanced Applications*, pages 834–843, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[WG12] A. Walther and A. Griewank. *Getting Started with ADOL-C*, pages 181–202. Chapman & Hall/CRC Computational Science. Dagstuhl Seminar Proceedings 09061, 2012.

[XJM13] S. Xu, W. Jahn, and J.-D. Müller. CAD-based shape optimisation with CFD using a discrete adjoint. *Int. J. Numer. Meth. Fluids*, 74(3):153–68, 2013.

[XRMM15] S. Xu, D. Radford, M. Meyer, and J.-D. Müller. CAD-Based Adjoint Shape Optimisation of a One-Stage Turbine with Geometric Constraints. *ASME Turbo Expo 2015*, 2C: Turbomachinery, 2015.

[YMJC11] G. Yu, J.-D. Müller, D. Jones, and F. Christakopoulos. CAD-based shape optimisation using adjoint sensitivities. *Computers & Fluids*, 46(1):512–516, 2011. 10th ICFD Conference Series on Numerical Methods for Fluid Dynamics (ICFD 2010).

[ZBLN97] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.