Technische Universität Kaiserslautern

Combining Parameterizations, Sobolev Methods and Shape Hessian Approximations for Aerodynamic Design Optimization

Fachbereich Mathematik
Technische Universität Kaiserslautern

eingereicht von
Thomas Karl Oskar Dick

Vom Fachbereich Mathematik der Technischen Universität Kaiserslautern genehmigte Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Doctor Rerum Naturalium, Dr. rer. nat.)

1. Gutachter: Prof. Dr. Nicolas R. Gauger
2. Gutachter: Jun.-Prof. Dr. Martin Siebenborn
Datum der Disputation: 14.10.2022

DE-386

Abstract

Aerodynamic design optimization, considered in this thesis, is a large and complex area spanning different disciplines from mathematics to engineering. To perform optimizations on industrially relevant test cases, various algorithms and techniques have been proposed throughout the literature, including the Sobolev smoothing of gradients. This thesis combines the Sobolev methodology for PDE constrained flow problems with the parameterization of the computational grid and interprets the resulting matrix as an approximation of the reduced shape Hessian.

Traditionally, Sobolev gradient methods help prevent a loss of regularity and reduce high-frequency noise in the derivative calculation. Such a reinterpretation of the gradient in a different Hilbert space can be seen as a shape Hessian approximation. In the past, such approaches have been formulated in a non-parametric setting, while industrially relevant applications usually have a parameterized setting. In this thesis, the presence of a design parameterization for the shape description is explicitly considered. This research aims to demonstrate how a combination of Sobolev methods and parameterization can be done successfully, using a novel mathematical result based on the generalized Faà di Bruno formula. Such a formulation can yield benefits even if a smooth parameterization is already used.

The results obtained allow for the formulation of an efficient and flexible optimization strategy, which can incorporate the Sobolev smoothing procedure for test cases where a parameterization describes the shape, e.g., a CAD model, and where additional constraints on the geometry and the flow are to be considered. Furthermore, the algorithm is also extended to One Shot optimization methods. One Shot algorithms are a tool for simultaneous analysis and design when dealing with inexact flow and adjoint solutions in a PDE constrained optimization. The proposed parameterized Sobolev smoothing approach is especially beneficial in such a setting to ensure a fast and robust convergence towards an optimal design.

Key features of the implementation of the algorithms developed herein are pointed out, including the construction of the Laplace-Beltrami operator via finite elements and an efficient evaluation of the parameterization Jacobian using algorithmic differentiation. The newly derived algorithms are applied to relevant test cases featuring drag minimization problems, particularly for three-dimensional flows with turbulent RANS equations. These problems include additional constraints on the flow, e.g., constant lift, and the geometry, e.g., minimal thickness. The Sobolev smoothing combined with the parameterization is applied in classical and One Shot optimization settings and is compared to other traditional optimization algorithms. The numerical results show a performance improvement in runtime for the new combined algorithm over a classical Quasi-Newton scheme.

Zusammenfassung

Aerodynamische Designoptimierung ist ein weitreichendes, komplexes Arbeitsfeld, welches verschiedene Disziplinen von der Mathematik bis zu den Ingenieurwissenschaften involviert. Um Optimierungen in industriell relevanten Fällen zu betrachten, stehen in der Literatur eine große Bandbreite an möglichen Algorithmen zur Verfügung. Unter anderen ist hier das Sobolevglätten des Gradienten zu nennen. Die vorliegende Arbeit verbindet die Sobolevmethode für Strömungsprobleme, welche durch partielle Differenzialgleichungen beschränkt sind, mit der Parametrisierung des numerischen Netzes und interpretiert die resultierende Gesamtmatrix als eine Approximation der reduzierten Shape-Hessematrix.

Traditionell vermeiden Sobolev Methoden Regularitätsverluste und helfen, hochfrequente Fehler bei der Ableitungsberechnung zu reduzieren. Eine solche Sobolev Neuinterpretation des Gradienten in einem anderen Hilbertraum kann auch als eine Approximation des Shape-Hesseoperators betrachtet werden. Bisherige Ergebnisse hierzu berücksichtigen jedoch nicht die Parametrisierung, welche für industriell relevante Anwendungen stets gegeben ist. Ziel dieser Forschung ist es, die genannten Punkte erfolgreich zu kombinieren. Hierzu wird ein neues mathematisches Resultat verwendet, welches auf einer Verallgemeinerung der Faà di Bruno Formel basiert. Eine solche Formulierung kann auch dann vorteilhaft für die Optimierung sein, wenn die verwendete Parametrisierung selbst bereits glatt ist.

Für Testfälle, in denen die Form durch eine Parametrisierung, z. B. ein CAD Modell, beschrieben wird, erlauben die vorgestellten Resultate Sobolevglätten effizient und flexible in das Optimierungsverfahren einzubinden. Darüber hinaus kann der Algorithmus für die Anwendung von One Shot Verfahren erweitert werden. Diese erlauben einen gleichzeitigen Simulations- und Designprozess unter Verwendung von inexakten, approximierten Strömungs- und adjungierten Lösungen. Parametrisiertes Sobolevglätten ist für solche Situationen besonders geeignet, da es eine schnelle und gleichzeitig robuste Konvergenz des Verfahrens garantiert.

Kernpunkte der Implementierung des hier entwickelten Algorithmus werden in der vorgestellten Arbeit beschrieben. Insbesondere die Berechnung des Laplace-Beltrami Operators mit Hilfe einer Finite-Elemente-Methode und der Einsatz von Algorithmischem Differenzieren zur effizienten Auswertung von Ableitungen der Parametrisierung. Die neuentwickelten Algorithmen werden zum Test auf Widerstandsminimierungsprobleme für dreidimensionale, turbulente Reynolds-gemittelte Navier-Stokes-Gleichungen angewendet. Die beschriebenen Testfälle enthalten darüber hinaus weitere Nebenbedingungen, wie konstanten Auftrieb für die Strömung und minimale Dicken für die Flügelgeometrie. Parametrisiertes Sobolevglätten wird hier sowohl in einem klassischen als auch in einem One Shot Algorithmus getestet und die Ergebnisse in beiden Fällen werden mit anderen traditionellen Verfahren verglichen. Diese Vergleiche zeigen, dass der neue kombinierte Algorithmus die Laufzeit gegenüber bekannten Quasi-Newtonverfahren verbessert.

Acknowledgement

I want to thank all the people who supported me in the past years. Either in working on and writing this thesis or otherwise.

Foremost, I would like to thank my advisor Prof. Dr. Nicolas R. Gauger, for his support in my work, the freedom in research he provided, and the inspiration to do this thesis. Furthermore, I would like to thank Dr. Stephan Schmidt for the many fruitful discussions, the research support, and his invaluable advice on shape derivatives and optimization. His cooperation has proven highly beneficial in this project. Also, I would like to thank Jun.-Prof. Dr. Martin Siebenborn for accepting the role as a second referee.

I would also like to thank my colleagues at the Scientific Computing group for providing such a great work atmosphere. Especially, I want to thank Dr. Lisa Kusch for the discussions and her advice on One Shot optimization. Furthermore, I thank Dr. Ruben Sánchez Fernández for his help with finite elements solvers, Dr. Max Sagebaum for the fruitful discussions about algorithmic differentiation, and Payam Dehpanah for his help in creating numerical meshes.

Next, I want to thank all the researchers working in the SU2 community for their work. The multiphysics analysis and design optimization software they provide enabled the implementation and numerical results present in this thesis. As a last researcher, I want to mention the late Dr. Bernhard Eisfeld, formerly from the German Aerospace Center (DLR) Braunschweig, for his valuable feedback on turbulence in the ONERA M6 test case.

I would also like to thank Dr. Raul Epure and Dr. Tobias Seidel for proofreading parts of this thesis.

Finally and naturally, I would like to thank my parents for their support and encouragement throughout my life, especially during the last couple of years when I was working on this thesis.

Contents

List of Symbols

In this thesis the following notation is used to ensure a unified style. The meaning of symbols is explicitly introduced in the chapters of their introduction or for reappearances the relevant chapter is referenced.

I. Aerodynamic Symbols:

Ω	flow volume
Γ	design surface
Θ	volume of the flow obstacle to be designed
n	surface normal vector
v	flow speed
ρ	density
W	flow field, analytic solution of the flow equation
Re	Reynolds number
α	angle of attack
c_P, c_D, c_L, c_M	aerodynamic pressure, drag, lift and pitching moment coefficients

II. Optimization Functions and Variables:

F	objective function
u	state variable, solution of the discrete flow equation
x	vector of mesh coordinates
p	design parameters for optimization
M	mesh parameterization, mapping $p \mapsto x$
H	discrete flow equations
G	iterative flow solver, pseudo time-stepping of the finite volume method
E	equality constraint for optimization
C	inequality constraint for optimization
$B_\varepsilon(x)$	ε-neighborhood around a point x
S_*	feasible set, i.e., the set of possible values for variable $*$

III. Derivatives and Algorithmic Differentiation:

$\mathrm{D}_x f$	partial derivative of f w.r.t. x, for finite-dimensional vector spaces the Jacobian matrix
$\mathscr{D}\mathscr{F}(\Omega; v)$	shape derivative or directional derivative of a functional f in direction v
$\frac{\partial}{\partial x_i}$	partial derivative of a scalar function w.r.t. the i-th component of the input vector x
$\frac{\mathrm{d}}{\mathrm{d}x_i}$	total derivative of a scalar function w.r.t. the i-th component of the input vector x
d_α^w	weak derivative w.r.t. a multiindex α
φ	elementary statement in algorithmic differentiation
z_k	intermediate result value in algorithmic differentiation

IV. Discrete Adjoint Calculus and One Shot Optimization:

L	Lagrangian function
λ	adjoint state variable
I	identity operator, for finite dimensions the identity matrix
$\varepsilon_1, \varepsilon_2$	coefficients in the Laplace-Beltrami operator

V. Function spaces and Sobolev smoothing:

L^p	space of p-integrable functions
H^k	Sobolev space of order k
ϕ	test function
M_S	surface parameterization, mapping p to the surface mesh nodes
M_V	mesh deformation, mapping surface to volume mesh nodes
$\mathscr{B}$	approximation of the reduced shape Hessian on the mesh
B	parameterized (hybrid) Laplace-Beltrami operator

Chapter 1

Introduction

Starting from the title of this thesis, ‘Combining Parameterizations, Sobolev Methods and Shape Hessian Approximations for Aerodynamic Design Optimization’, this work aims to combine different topics and apply them together. To better explain this, an overview of the state of the art and references to the relevant literature are given for each topic.
Beginning with introducing the intended application in aerodynamic shape optimization, the adjoint optimization framework and its extension to One Shot methods are discussed. This is followed by an overview of the role of design parameterization in shape optimization. Next, relevant results on shape calculus and shape Hessians are listed. In particular, results investigating a connection to partial differential operators, like Sobolev methods, and their potential uses to improve optimization algorithms. After presenting an overview of the existing literature, the research objectives and the scientific contributions of this thesis are listed. This is then followed by an explanation of the structural layout of this work.

1.1 Background of This Thesis

This section describes the current state of the art in the research fields this thesis builds upon. It lists previous publications by various authors, some of which inspired this work with their research on related topics. A particular inspiration for this thesis was the joint work by Kusch, Schmidt, and Gauger on approximated Newton methods for shape optimization via smoothing [70, 71].

Aerodynamic Shape Optimization

Design optimization, also known as shape optimization, is a broad term used in engineering [84, 15, 16] that generally refers to improving and possibly optimizing the geometry of an object with respect to one or multiple of its properties. In aerodynamic applications, a component which is in contact with a fluid, e.g., wings, turbine blades, and rotors, should be optimized. The property of interest is derived from the interaction of the component with the fluid, e.g., drag and lift forces, structural loads, or heat exchange. Such examples can span a wide variety of questions, like increasing the efficiency of modern jet engines by improving the turbine blades, as done by Backhaus, Schmitz, et al. [12], or reducing the noise of an airframe, see Zhou, Albring, et al. [126], or improving the electricity production of a wind power plant, see King, Dykes, et al. [68].

Of the different algorithms proposed for shape optimization, the adjoint framework for derivative-based optimization has proven itself as one of the most successful [86]. Mainly, because of its independence from the number of design parameters, in terms of computational cost. The framework can be divided into discrete adjoint methods, pioneered by Giles [46], and continuous adjoint methods, pioneered by Jameson [62]. Both have their advantages and disadvantages. However, discrete adjoints have some benefits in industrial applications. Mainly, when set up using algorithmic differentiation [49], they allow to create consistent and efficient adjoint solvers for the simulation algorithms in an automated way [4]. Such approaches have been deployed in large scale industrial applications, where the above mentioned adjoint property of being independent of the number of design variables in terms of computational cost is particularly interesting [86].

One Shot Optimization

The One Shot optimization algorithm has been discussed extensively for its great potential to improve adjoint-based aerodynamic shape optimization. The term itself was first proposed by Ta'asan [119]. A good introduction and an overview of the historical development can be found in the overview paper by Bosse, Gauger, et al. [23]. The idea can be motivated in two different ways, first as a Newton step on the whole KKT system, with extensive convergence analysis done by Hamdi and Griewank [51, 52]. The second approach views the method as a pseudo time-stepping for the design equation, as done by Ta'asan [120], or Hazra, Schulz, Brezillion, and Gauger [55]. Both approaches are equivalent, ultimately resulting in the same algorithm, which was applied in various aerodynamic settings [55, 24]. This also includes more complicated partial differential equation constrained optimization methods, e.g., for unsteady aerodynamics by Günther [50], or even the design of tokamak nuclear fusion reactors by Blommaert [19].

Of course, many extensions and refinements to the original One Shot idea were introduced over the years, with some of the more noteworthy being multistep algorithms, as studied by Özkaya [92], and even asynchronous execution of the involved solvers, as suggested by Bosse [21]. Multiple authors also studied the incorporation of additional constraints into this optimization [54, 22, 72], either by treating them in an SQP like fashion when updating the design, as done by Hazra and Schulz [54], or by adding them to the Lagrangian of the original problem via multipliers, as introduced by Kusch, Walther, et al. [124, 72].

Most of this research emphasizes the adequate choice of a design preconditioner to ensure the convergence and stability of the algorithm. Therefore, as part of the presented thesis, the introduced Sobolev smoothing motivated techniques are tested as preconditioners within the One Shot framework. Multistep One Shot algorithms are used and the additional design constraints are handled following the ideas in [54], since they can be easily extended to include multiple additional constraints, including geometric inequalities, without additional changes to the underlying adjoint solver.

Parameterization

In shape optimization, the shape to be designed is the component's geometry which has to be described. In industrial and engineering applications, this is done in a computer-based format, usually given by a CAD model [112]. Such a model uses a mathematical parameterization and describes the shape by several discrete values, called the design parameters. Naturally, the optimization then becomes a question of finding the optimal set of design parameters. Derivative-based algorithms have to consider this when deriving their formulation of gradients. However, these typically only compute first order derivatives. The matter becomes much more involved when considering higher order differentiation as computational costs increase [48].

Theoretically, only requiring correct generalization of the chain rule, the exact formula for multidimensional higher order derivatives is known as the generalized Faà di Bruno formula [31, 38]. It can be adapted and simplified in the presented context to incorporate the parameterization into the formulation of an approximated reduced shape Hessian matrix.

Special focus is given to keeping these results independent from the particular choice of parameterization. For the results presented in this thesis, Hicks-Henne functions [57] and FFD boxes [111] are used since these approaches are representative of a wide array of common parameterizations used throughout the aerodynamics community. Other parameterizations can be used as well, as long as their first order derivatives are available.

Shape Hessians and Sobolev Smoothing

Next, shape calculus, especially regarding shape Hessian approximation and Sobolev smoothing, has to be considered. A general overview of the topic can be found in multiple textbooks [116, 84, 6]. Such mathematical formulations usually require very complex differential calculus and a deep understanding of the functions and operators involved. Naturally, this can be a significant drawback for the fast adaptation of such work in numerical applications.

Discretization of the calculated derivatives on the computational mesh often leads to high-frequency oscillations in the sensitivities and subsequent induced errors. This has led to several techniques to increase the regularity of the search direction and smoothen the sensitivities, with Sobolev smoothing being one of the most popular ones. An introduction can be found in the books of Faragó and Karátson [39], or Neuberger [88]. A related idea was first proposed in a CFD context by Jameson [62], although not fully formulated back then.

Most actual research papers on shape Hessians focus on applying general mathematical techniques to derive the Hessian formulation for a specific problem. This is either done in a completely continuous formulation or as a free node optimization on the mesh level. For example, Schmidt [105] presented a method to derive weak and strong formulations of shape Hessians automatically and applied this to an iso-perimeter problem and incompressible Navier-Stokes equations. Until now, no research has explicitly considered the effect of the geometry parameterization on the formulation of a smoothing operator as a shape Hessian approximation.

An approach to formulate the operator symbol for the Hessian of a drag minimization problem with Euler equations was done by Arian and Ta'asan [10], where the investigation of a 'small disturbance problem' leads to the smoothing of sensitivities on the design surface in a chord-wise direction while coarsening them in a span-wise direction. Following up on this, Arian and Vatsa [11] developed a smoothing operator for shape optimization and proposed to remesh the deformed area after the update step. Based upon these results, Kusch, Schmidt, and Gauger [70, 71] investigated the drag minimization problem for Stokes equations. The authors were able to formulate the operator symbol for the Hessian in local coordinates and approximate it in terms of a Laplace-Beltrami operator. A result that inspired the research done in this thesis

Other researchers proposed alternative partial differential operators as well to improve the regularity of the search direction, e.g., using a Stecklov-Poincaré type metric proposed by Schulz and

Siebenborn [110], or a p-Laplace problem suggested in a more recent study by Müller, Kühl, et al. [85].

1.2 Research Objectives

This thesis aims to combine different research areas relevant to aerodynamics shape optimization. First, the discrete adjoint optimization framework, and especially the One Shot optimization approach, are extended by a novel optimization technique. This technique combines the Sobolev smoothing method for sensitivities on the surface or volume level with design parameterizations of the computational mesh. The aim is to interpret this combination as a shape Hessian approximation and deploy it in a flexible optimization framework, with the potential to perform simultaneous analysis and design. Furthermore, all of this is formulated in a general way to allow for the extension of additional constraints straight away. To achieve this stated objective and demonstrate the presented methodology's capabilities, this thesis begins by formulating the mathematical framework, discusses the implementation, and finally presents numerical experiments.

Mathematically, to achieve such a novel Sobolev smoothing methodology, a theorem on the connection of the discretized shape Hessian and the second order derivatives with respect to the design parameters is introduced and proven. These theoretical results are used to construct an efficient yet computationally cheap, Sobolev smoothing operator on the space of design parameters. A special focus is given to incorporating the new results into a flexible reduced SQP optimization framework. Issues, such as additional constraints and the extension to One Shot optimization, are discussed and practical solutions are formulated.

The presented algorithms are implemented within a state of the art software framework for aerodynamic optimization. This implementation's key ideas and relevant features are highlighted and addressed in detail.

Numerical experiments are performed for relevant reference test cases using the newly developed algorithms. The results are examined and compared to similar, commonly used optimization techniques, both in terms of mathematical performance and computational cost. This includes investigating the improvement in the objective function, adherence to constraints, and convergence behavior. In addition, computational costs are examined in terms of runtime and iteration counts on different processor architectures on an HPC cluster. These numerical evaluations and results are discussed with respect to applicability in the initially proposed aerodynamic shape optimization.

The novelty of the presented thesis is the successful combination of Sobolev smoothing with a design parameterization and its application as a shape Hessian approximation for aerodynamic design optimization. The work combines all of this within a flexible One Shot algorithm to achieve considerable benefits for industrially relevant applications. Thus, not only pure Sobolev gradients are used, but also a powerful optimization algorithm for constrained shape optimization problems is formulated.

1.3 Structure of This Thesis

The layout of this thesis aims first to establish all fundamental topics and then introduce the results step by step. To this end, the earlier chapters may initially stand on their own, but they will all be interconnected later on.

- **Chapter 2:** This chapter serves as an in-depth introduction to the scientific background. Fundamental concepts and the related notation are introduced, including aerodynamic concepts, such as flow equations, aerodynamic functionals, and turbulence modeling. This is extended by the basic notations for computational meshes and finite volume flow solvers. Afterwards, the notation for mathematical optimization and derivative-based optimality criteria is stated, and basic concepts from shape calculus are introduced. Here, the role of design parameterization is pointed out and the parameterizations used in this thesis are derived from geometric modeling. Finally, the discrete and continuous formulations of the optimization problem are compared to each other.

- **Chapter 3:** This chapter aims at an introduction of the discrete adjoint optimization framework. First, the adjoint calculus is derived from the optimality conditions. This is done in a free node formulation. Next, special attention is given to presenting the effects a parameterization has on this formulation. Afterwards, an overview of algorithmic differentiation is combined with an explanation of how it can be used to calculate the derivatives appearing in the adjoint formulation. With this, the basic iterative algorithms for solving the adjoint equation are formulated. At the end of the chapter, the mathematical framework is used to construct a reduced SQP optimization algorithm, allowing the basic adjoint method to consider additional constraints.

- **Chapter 4:** This chapter enhances the discrete adjoint methodology into the One Shot optimization framework. A short overview of the convergence analysis for the simultaneous analysis and design strategy is given and then the One Shot algorithms used in this thesis are formulated. In the end, the incorporation of additional optimization constraints into the presented setting is discussed as well.

- **Chapter 5:** This chapter has two key objectives. First, different results on shape Hessians and their connection to elliptic smoothing operators are presented, motivating the approximation by such an operator. The Sobolev smoothing technique for function spaces and its application to free node optimization are formulated. The second half of this chapter combines these results with the design parameterization. The central theorem for the connection of reduced shape Hessian matrices and discrete second order derivatives with respect to the parameters is derived from the Faà di Bruno formula and proven. At last, this is combined with the optimization algorithms for adjoint and One Shot optimization derived in Chapters 3 and 4.

- **Chapter 6:** After establishing the new algorithm for parameterized Sobolev smoothing in the previous chapter, this chapter explains the implementation. It starts by introducing the existing software frameworks in which the algorithms will be implemented. Afterwards, the implementation for the computation of the smoothing operator based on finite elements is discussed. This includes presenting the necessary changes to the existing adjoint solver and how to compute and incorporate the derivatives of the parameterization. Finally, the optimization algorithms introduced in Chapters 3 and 4 can be realized by building upon these newly established implementations.

- **Chapter 7:** In this chapter, the methodology derived in this thesis is applied to relevant test cases from aerodynamic shape optimization. Two different test cases are considered and the discussion is split into two parts for each of them. First, the newly developed preconditioner is applied for classical adjoint optimization with exact, fully converged functions and gradients. Second, it is tested for One Shot optimization. In both areas, the performance is evaluated in several ways. The observed improvement of the optimization and the convergence behavior of the implementation are compared to other relevant algorithms. In addition, the computational costs, e.g., the runtime, are evaluated on different hardware architectures.
- **Chapter 8:** In the final chapter, the scientific results of this thesis are discussed. Beginning with the theoretical results, followed by a discussion of the numerical experiments and their findings. Afterwards, conclusions are drawn from these results on how to apply the new algorithms for relevant aerodynamic shape optimization problems. Finally, a short outlook on future research questions in the area is given, including ideas for potential improvements of the presented algorithms.

Chapter 2

Fundamentals

This chapter introduces the background of this work from different fields of research. This introduction aims to state the critical concepts used throughout this thesis and define the notation.
The first Section 2.1 gives a brief overview of aerodynamics, including the formulation of the flow equations as conservation laws. Additionally, some essential aerodynamic functionals are defined since they are used as objective functions and constraints in the numerical test cases for this work. At last, the Reynolds averaging of the Navier-Stokes equations is stated and some information about the turbulence models, used for numerical test cases, is given. In the second Section 2.2, the fundamental ideas of a finite volume flow solver are introduced, as far as they are relevant for the computations done in this work. This is complemented by a discussion on the role of the computational mesh, which will become relevant when formulating the optimization problem. The third Section 2.3 revolves around the concepts from mathematical optimization used in this thesis. This begins with the basic formulation of optimality criteria in optimization. Next, the definitions from shape calculus for a continuous shape Hessian are stated, which are relevant later for understanding results from the literature on how the Laplace-Beltrami operator can model the operator symbol of such a Hessian. This is followed by a detailed discussion of the shape parameterization and its role in describing the design, including the formulation of the parameterizations used in this thesis. Finally, the discrete and continuous approaches to optimization are compared to each other.

2.1 Aerodynamics

This thesis discusses methods for aerodynamic shape optimization problems in the context of engineering and industrial application. Much of the focus of this thesis lies on the construction of algorithms, mathematical aspects of the partial differential equation (PDE) constrained optimization problems, and the computationally efficient application of these ideas in relevant simulation software. However, such a discussion still requires an understanding of the underlying PDE structure. This section introduces essential concepts from aerodynamics to understand the flow equations involved and the aerodynamic functionals used in the optimization problems. Note that it is not part of this overview to discuss these physical equations' exact modeling or derivation.

2.1.1 Flow Equations

The term flow equation refers to the physical conservation laws dictating the state and movement of a fluid, i.e., a gas or a liquid. Typical quantities described by this are mass, momentum, angular momentum, and energy, whose conservation form the main laws of classical mechanics [73] and have a deep connection to the symmetry of such physical systems via the famous Noether theorem [91]. In mathematical terms, the equations take the form of partial differential equations. The following notation is based on the introductory textbook by Blazek [18, Chapter 2]. Further explanations can be found in many standard textbooks, e.g., Anderson [8], or Schröder [107].

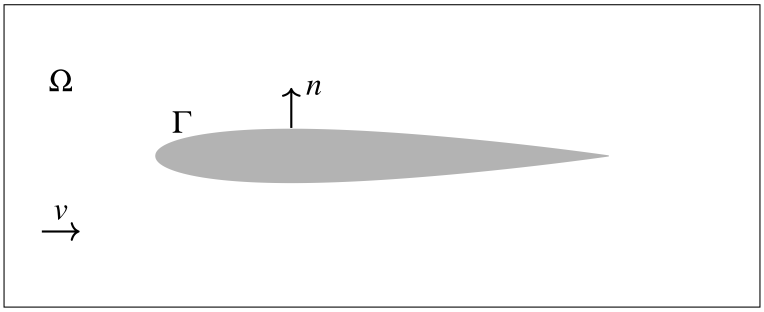


Figure 2.1: Flow domain around an airfoil.

As shown in Figure 2.1, the flow can be described in terms of several quantities, including the flow volume Ω, the boundary Γ, the normal vector n, and the flow speed v. Following the idea of a conservation law, some physical quantities must be preserved in an arbitrary flow volume. Listing these conserved physical quantities gives three independent equations.

1. The *continuity equation* is the conservation of mass. For a fluid with density ρ, this means that the rate of mass change over time must be equal to the mass flow over the boundaries of the control volume
$$\int_\Omega \frac{\partial}{\partial t}\rho \,\mathrm{d}x + \oint_{\partial\Omega} \rho \langle v, n\rangle \,\mathrm{d}s = 0. \tag{2.1}$$

2. The *momentum equation* is the conservation of the flow momentum. According to Newton's second law, a change in momentum must equal the forces acting on the fluid. Overall, summation of the involved forces results in an equation in integral form
$$\int_\Omega \frac{\partial}{\partial t}(\rho v) \,\mathrm{d}x + \oint_{\partial\Omega} \rho v \langle v, n\rangle \,\mathrm{d}s = \int_\Omega \rho F_E \,\mathrm{d}x - \oint_{\partial\Omega} pn \,\mathrm{d}s + \oint_{\partial\Omega} \tau n \,\mathrm{d}s. \tag{2.2}$$
Here, three source terms are present on the right-hand side, representing different kinds of forces. The term F_E are the external (volume) forces acting on the fluid body, e.g., gravity. The second term is the pressure distribution across the surface, resulting in a distributed force, while in the last term of the expression τ is the so-called viscous stress tensor, related to the fluid's internal friction. It is worthwhile noting that the viscous stress tensor plays an essential role in turbulence modeling, see Subsection 2.1.3.

3. The *energy equation* is the conservation of the total energy E, as stated by the first law of thermodynamics. Normally, the ideal gas law
$$p = \rho R T, \tag{2.3}$$

where R is the specific gas constant, is assumed to simplify the formulation. In addition, it is assumed that

$$p = (\gamma - 1)\rho(E - \frac{1}{2}\langle v, v\rangle) \text{ and } \rho H_E = \rho E + p, \tag{2.4}$$

where γ denotes the heat capacity ratio of the fluid and H_E the enthalpy. Also, assume that the heat transfer on the surface is modeled by Fourier's law

$$q_S = -k\langle \nabla T, n\rangle, \tag{2.5}$$

where T is the total temperature and k is the thermal conductivity coefficient. Then the temperature can be expressed as

$$T = \frac{p}{R\rho}. \tag{2.6}$$

With these connections, the conservation of energy can be written as

$$\int_\Omega \frac{\partial}{\partial t}(\rho E)\,\mathrm{d}x + \oint_{\partial\Omega} \rho H_E \langle v, n\rangle \,\mathrm{d}s = \oint_{\partial\Omega} k\langle \nabla T, n\rangle \,\mathrm{d}s + \oint_{\partial\Omega} (\tau v) n \,\mathrm{d}s. \tag{2.7}$$

Combining all three equations (2.1), (2.2), and (2.7) into one overall set of equations results in the famous *Navier-Stokes equations*, which govern the behavior of a fluid according to classical physics

$$\int_\Omega \frac{\partial}{\partial t} W \,\mathrm{d}x + \oint_{\partial\Omega} (F_C - F_V)\,\mathrm{d}s = \int_\Omega Q \,\mathrm{d}x. \tag{2.8}$$

The short formulation, in Equation (2.8), utilizes several simplified terms, so one should clarify their meaning. W is the vector of *conservative variables*

$$W = (\rho,\ \rho v_1,\ \rho v_2,\ \rho v_3,\ \rho E)^T. \tag{2.9}$$

Where v_i is the component of the velocity in the i-th unit dimension. The two flux terms have slightly more involved formulas. To express them in a more readable way, the velocity normal to the surface is denoted by $v_n = \langle v, n\rangle$ and the Einstein sum convention is used, stating that a sum is taken over all indices which appear multiple times in an expression. Then the flux terms are

$$F_C = \begin{bmatrix} \rho v_n \\ \rho v_1 v_n + n_1 p \\ \rho v_2 v_n + n_2 p \\ \rho v_3 v_n + n_3 p \\ \rho H_E v_n \end{bmatrix}, \qquad F_V = \begin{bmatrix} 0 \\ \tau_{1,i} n_i \\ \tau_{2,i} n_i \\ \tau_{3,i} n_i \\ n_i\left(\tau_{i,j} v_j + k\frac{\partial}{\partial x_i} T\right) \end{bmatrix}. \tag{2.10}$$

Finally, the source term on the right-hand side of the equations is given by

$$Q = (0,\ \rho F_{E,1},\ \rho F_{E,2},\ \rho F_{E,3},\ 0)^T. \tag{2.11}$$

Instead of the conservation formulation, also referred to as weak formulation, given in Equation (2.8), the Navier-Stokes equations are often expressed in differential form. For this, the solution is assumed to contain no shocks and a Newtonian fluid is assumed. Therefore, the viscous stress tensor can be written as

$$\tau_{ij} = \mu \left(\left(\frac{\partial}{\partial x_j} v_i + \frac{\partial}{\partial x_i} v_j \right) - \frac{2}{3} \frac{\partial}{\partial x_k} v_k \delta_{ij} \right). \tag{2.12}$$

Here, the scalar constant μ is known as the shear viscosity.

When using the notation stated above, the Navier-Stokes equations are transformed into a system of partial differential equations, which can be written separated into the three components for mass, momentum, and energy [107, Chapter 4, Pages 45-48].

$$\begin{aligned} \frac{\partial}{\partial t}\rho + \operatorname{div}(\rho v) &= 0 \\ \frac{\partial}{\partial t}(\rho v) + \operatorname{div}(\rho v \otimes v) &= \rho F_E - \nabla p + \nabla^T \tau \\ \frac{\partial}{\partial t}(\rho E) + \operatorname{div}((\rho E + p)v) &= -\operatorname{div}(k\langle \nabla T, n\rangle) - \operatorname{div}(\tau v) \end{aligned} \tag{2.13}$$

Here, $\otimes$ denotes the outer product between two vectors. Due to the complicated nature of the Navier-Stokes equations and the abundance of physical phenomena they describe, e.g., shocks, vortices, turbulence, boundary layers, etc., many simplifications have been introduced in the past. One of the most famous are the *Euler equations*. They neglect the viscous terms in the equations to describe an inviscid fluid, thus taking the form

$$\int_\Omega \frac{\partial}{\partial t} W \,\mathrm{d}x + \oint_{\partial\Omega} F_C \,\mathrm{d}s = \int_\Omega Q \,\mathrm{d}x. \tag{2.14}$$

While solutions of these equations do not show turbulence or boundary layer effects, they can still model shocks. Therefore, they are a valid approximation in various applications, like laminar flows. The Euler equations can be expressed in a differential form as well.

$$\begin{aligned} \frac{\partial}{\partial t}\rho + \operatorname{div}(\rho v) &= 0 \\ \frac{\partial}{\partial t}(\rho v) + \operatorname{div}(\rho v \otimes v) &= \rho F_E - \nabla p \\ \frac{\partial}{\partial t}(\rho E) + \operatorname{div}((\rho E + p)v) &= -\operatorname{div}(k\langle \nabla T, n\rangle) \end{aligned} \tag{2.15}$$

Of course, more complex physical models exist to express more complicated flows, e.g., turbomachinery applications [34, 121], non-equilibrium flows [78], etc., but these applications are beyond the scope of this work.

2.1.2 Aerodynamic Functionals

While the last Subsection 2.1.1 explains the equations describing the flow of a fluid in a given geometry, the solutions of such partial differential equations are highly nonlinear functions. From a purely theoretical viewpoint, the continuous vector field W contains all the necessary information in the conservative variables. Nonetheless, they are not stated in the most convenient form for an engineer or anyone else interested in optimizing an aerodynamic shape. The total forces imposed by

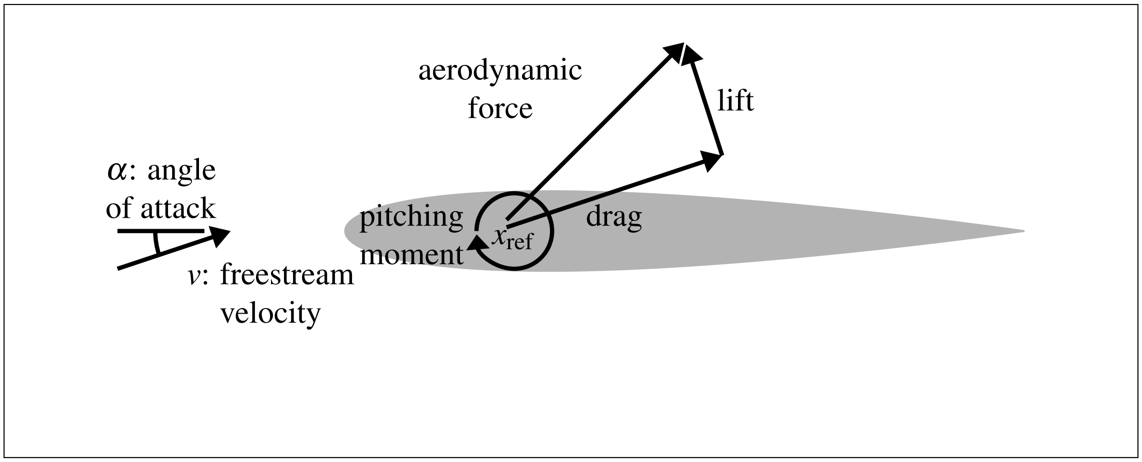


Figure 2.2: Aerodynamic forces acting on an airfoil.

the fluid on the geometry are usually what one is interested in. They can be stated in terms of scalar, dimensionless aerodynamic coefficients, which allow a comparable measure for the aerodynamic performance of different shapes. In this subsection, a few of them are defined, as long as they are relevant to the scope of this thesis. Regardless, one should remember that many more such coefficients and efficiency functionals are known throughout the engineering community. For example, multiple efficiency definitions are used to evaluate the performance of turbomachinery [34, Chapter 2].

The force acting on an object moving through a fluid depends on the pressure distribution p and the shear stress tensor τ over the surface

$$F = \int_{\Gamma} -pn + \tau n \, \mathrm{d}s. \tag{2.16}$$

It is a major goal in aerodynamics to calculate the pressure p and shear stress tensor τ and to derive expressions for the forces from them. In general three-dimensional settings, this can get very involved and the following discussion is simplified to shorten the notation here. The complete force calculation can be found in the introductory textbooks by Blazek [18, Chapter 2], or Anderson [8, Chapter 1]. The total force can be subdivided into a superposition of forces in different directions, see Figure 2.2. In general, the component of this force parallel to the freestream direction is called the *drag*, while the orthogonal component is named the *lift*. Also, a common technique is to express these in a dimensionless form to avoid the influence of the size, as making an object smaller naturally reduces the total force.

First, consider reference values derived from the freestream condition and the shape to calculate a dimensionless expression for the force.

- p_∞: The static pressure in the freestream.
- ρ_∞: The fluid density in the freestream.
- v_∞: The freestream velocity, i.e., the negative velocity of the shape through the fluid.
- l_{ref}: The reference length of the shape, which gives a notion of the overall size.

- S_{ref}: The reference surface area of the shape. For two-dimensional cases $S_{\text{ref}} = l_{\text{ref}}$.

Some additional terms can be defined to simplify the notation using these quantities.

- *dynamic pressure:* $q_\infty = \frac{1}{2}\rho_\infty v_\infty^2$
- *pressure coefficient:* $c_P = \frac{(p-p_\infty)}{q_\infty}$
- *skin friction coefficient:* $c_\tau = \frac{\tau}{q_\infty}$

These dimensionless numbers link the static pressure and friction in the evaluation point with the freestream conditions, i.e., the flow solution in an infinite distance to any obstacle. So now the three most common aerodynamic coefficients from engineering can be calculated as parts of the total force and defined as integrals over the flow surface.

1. The *drag coefficient* c_D gives a dimensionless measure of the drag. Let i_{v_∞} denote the unit vector in direction of the freestream velocity. For a given angle of attack α and following standard convention, this is $i_{v_\infty} = (\cos(\alpha), \sin(\alpha))^T$ in two dimensions and $i_{v_\infty} = (\cos(\alpha), 0, \sin(\alpha))^T$ in three dimensions. Then

   $$c_D = \frac{1}{S_{\text{ref}}} \int_\Gamma -c_P \langle i_{v_\infty}, n\rangle + \langle i_{v_\infty}, c_\tau n\rangle \,\mathrm{d}s. \tag{2.17}$$

   This is the resistance of the given shape against being moved through the fluid. Compared to classical friction, for solid on solid movement, here the resistance is caused by viscous and convective forces. However, the drag is sensitive to effects like shocks and turbulent effects that would not occur for any solid friction.

2. The *lift coefficient* c_L is a dimensionless measure for the lift produced by a shape. This is the force acting perpendicular to the drag. Following the previous convention, one can define the vector $i_{\perp v_\infty}$ as $i_{\perp v_\infty} = (-\sin(\alpha), \cos(\alpha))^T$ in two dimensions and $i_{\perp v_\infty} = (-\sin(\alpha), 0, \cos(\alpha))^T$ in three dimensions. Then

   $$c_L = \frac{1}{S_{\text{ref}}} \int_\Gamma -c_P \langle i_{\perp v_\infty}, n\rangle + \langle i_{\perp v_\infty}, c_\tau n\rangle \,\mathrm{d}s. \tag{2.18}$$

   For level flight, this force acts in the opposite direction as gravity, e.g., in aircraft wings, where it is of key importance. There are also applications in which a negative lift, meaning in the same direction as gravity, might be desired, e.g., the spoiler of a race car.

3. The *pitching moment* $c_M(x_{\text{ref}})$, with respect to a reference point x_{ref}, is given by

   $$c_M(x_{ref}) = \frac{1}{S_{\text{ref}}\, l_{\text{ref}}} \int_\Gamma -c_P((x - x_{\text{ref}}) \times n) + ((x - x_{\text{ref}}) \times (c_\tau n)) \,\mathrm{d}s. \tag{2.19}$$

   The components of this vector are the pitching moments around the corresponding coordinate axis and this dimensionless constant is used to illustrate the torque imposed by the aerodynamic forces acting on a shape.

This is only a tiny glimpse at the vast amount of possible functionals used in aerodynamic shape optimization. Nonetheless, they give a good example to the reader about the kind of functionals relevant to the scope of this work. In particular, they demonstrate what mathematical form the objective functions and the constraints used for aerodynamic shape optimization might take. For the rest of this thesis, optimization problems are generally formulated with such quantities as objectives and constraints, thus requiring the application of nonlinear optimization techniques for real-valued functions.

2.1.3 Turbulence Modeling

One of the most significant problems when dealing with computational fluid dynamics is the treatment of turbulence. Introductions to turbulent flows can be found in many textbooks, such as Pope [96] or Wilcox [125]. A vast quantity of research has been done on this topic and many possible approaches are known. The most accurate way would be to use a computational mesh that is fine enough to resolve all relevant turbulent effects in the flow solution completely. This approach is known as the *direct numerical solution* (DNS) and requires an unfeasible amount of computational power, even in relatively simple cases. Therefore, it is usually inapplicable for larger simulations, outside of academic test settings.

A much more practical idea is to solve an averaged version of the flow equations and compute an additional turbulence model, which imitates the main properties of the turbulent behavior. For this, another set of equations is introduced, the so-called *Reynolds averaged Navier-Stokes equations* (RANS). Their main purpose is to run computations with coarser computational meshes while still capturing the major flow properties in the solution.

Of course, not all flows are turbulent by nature. A useful indication for the occurrence of turbulence is a high *Reynolds number*

$$\mathrm{Re} = \frac{\rho v_\infty l_{\mathrm{ref}}}{\mu}. \tag{2.20}$$

The dimensionless Reynolds number describes the ratio of inertia to viscous forces and plays an important role in fluid dynamics. It turns out that flows with low Reynolds numbers tend to be laminar, while high Reynolds numbers are a sign of turbulent flows.

In the case of turbulent flows, an averaging approach can be used to split the occurring terms into two parts. For example, a quantity x might be written as

$$x = \overline{x} + x'. \tag{2.21}$$

With $\overline{x}$ being the time average of x and x' being the fluctuation around this average. The time average itself is defined as

$$\overline{x} = \lim_{\delta t \to \infty} \frac{1}{\delta t} \int_{\delta t} x \,\mathrm{d}t. \tag{2.22}$$

Averaging over time is appropriate for stationary turbulence phenomena, but the same basic principle can be applied to spatial or other averaging if the kind of flow requires it, see Wilcox [125, Section 2.1]. For a shorter notation, it is useful to calculate the mass weighted time average as well, where a quantity is written as

$$x = \widetilde{x} + x'', \tag{2.23}$$

with the average being defined via

$$\widetilde{x} = \frac{\lim\limits_{\delta t \to \infty} \frac{1}{\delta t} \int_{\delta t} \rho x \, \mathrm{d}t}{\lim\limits_{\delta t \to \infty} \frac{1}{\delta t} \int_{\delta t} \rho \, \mathrm{d}t}. \tag{2.24}$$

The original Navier-Stokes equations (2.13) can be averaged into the RANS equations, using a combination of time and mass weighted averages in a process known as Favre averaging, see for example Eisfeld [36] or Wilcox [125, Section 5.2]. For brevity, expressions are once again written using the Einstein sum convention.

$$\begin{aligned}
\frac{\partial}{\partial t}\overline{\rho} + \frac{\partial}{\partial x_k}(\overline{\rho}\ \widetilde{v_k}) &= 0 \\
\frac{\partial}{\partial t}(\overline{\rho}\ \widetilde{v_i}) + \frac{\partial}{\partial x_k}(\overline{\rho}\ \widetilde{v_i}\ \widetilde{v_k}) + \frac{\partial}{\partial x_k}(\overline{\rho v_i'' v_k''}) &= -\frac{\partial}{\partial x_i}\overline{p} + \frac{\partial}{\partial x_k}\overline{\tau_{ik}} + \overline{\rho} F_E \\
\frac{\partial}{\partial t}(\overline{\rho}\widetilde{E}) + \frac{\partial}{\partial x_k}(\overline{\rho}\ \widetilde{H_E}\ \widetilde{v_k}) + \frac{\partial}{\partial x_k}\overline{(\rho\ H_E'' v_k'')} &= \frac{\partial}{\partial x_k}(\overline{\tau_{ik}}\widetilde{v_i}) + \frac{\partial}{\partial x_k}\overline{(\tau_{ik} v_i'')} - \frac{\partial}{\partial x_k}\overline{(q_S)}_k
\end{aligned} \tag{2.25}$$

These equations have a similar structure to the original Navier-Stokes equations. However, new terms appear for the turbulence and therefore, there are more unknowns. These terms are known as the *Reynolds stress tensor*

$$\overline{\rho v_i'' v_k''} = \overline{\rho}\widetilde{v_i'' v_k''}, \tag{2.26}$$

or the *specific Reynolds stress tensor*

$$\widetilde{R_{ik}} = \widetilde{v_i'' v_k''}. \tag{2.27}$$

The main idea behind any turbulence model is to replace these terms in the RANS equations with an approximation expressed in terms of the other variables. The aim is to give a closure for the system, i.e., have as many equations as there are unknowns. Two popular turbulence models are used for the test cases in this work.

1. The Spalart–Allmaras (SA) model [117] introduces a new variable called the turbulent eddy viscosity $\widehat{v}$ and a transport equation for it. Because of this, it is normally referred to as a one equation model.

2. The Menter Shear Stress Transport (SST) model [83] blends two turbulence models into each other, with both of them being two-equation models. First, a k-ω turbulence model is used for the inner boundary layer. This model is based on the turbulent kinetic energy k and a dissipation rate ω for this energy. Second, a k-ε model is used in the flow domain consisting of two PDEs to describe the behavior of the turbulent kinetic energy k and a different rate of dissipation ε.

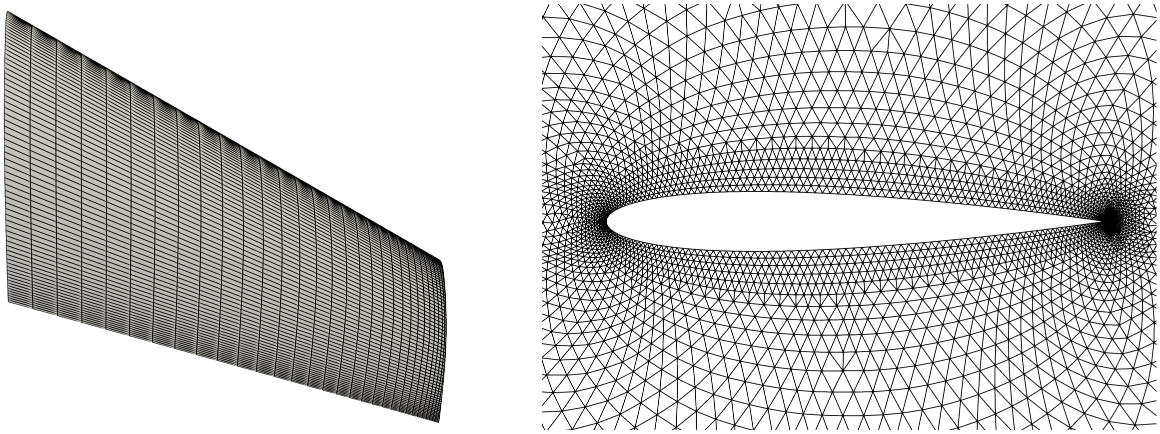
Naturally, each turbulence model represents a compromise between simplicity and accuracy. Higher degrees of accuracy, and thereby better flow solutions, can be achieved by using more involved Reynolds stress models [37, 61]. However, this requires additional knowledge and effort for the implementation and increases computational costs. Also, it is worth noting that recent developments try to employ machine learning to enhance the accuracy of existing turbulence models [95].

2.2 Computational Fluid Dynamics

Establishing the physical model and formulating partial differential equations for the flow problem is only the first step of doing actual shape optimization with them. Next, the equations introduced in Subsection 2.1.1 have to be solved. Unfortunately, a proof for the existence and uniqueness of solutions to the Navier-Stokes equations remains one of the biggest open problems in mathematics [40], so it should not be expected to have an analytical solution at hand for optimization. Nonetheless, a wide variety of numerical schemes for approximate solutions is available. These approaches vary and can range from finite volume methods [18, 76], or discontinuous Galerkin methods [30], over lattice Boltzmann methods [27], all the way to meshfree and particle simulations [75]. In this thesis, the simulations are done using the state of the art CFD framework SU2 [35, 93], more details on this will be given in Subsection 6.1.1. The SU2 flow solver uses finite volume methods, which have proven to be very powerful in the past and offer great potential for using adjoint approaches.

2.2.1 Computational Meshes

Before discussing the numerical schemes themselves, some words about spatial discretization are helpful. Computational *meshes* or *grids* arise as a natural consequence of numerical representation in an algorithm [18, 58, 74]. Finding a good way to represent a function inside a computer is a relevant research topic in its own right, but in most cases, this is solved by storing a triangulation of the domain and storing discrete function values on the vertexes or cells. Applying this principle to the spatial domain directly leads to the flow domain being represented by a set of coordinate points called the *nodes*, which form the mesh.



(a) Structured rectangular grid.

(b) Unstructured triangular grid.

Figure 2.3: Examples for a structured grid (surface cells on an ONERA M6 wing) and an unstructured grid (cells around a NACA 0012 airfoil).

Different types of grids are available, which can be generally divided into structured and unstructured grids [17, 26]. Figure 2.3 shows a visualization for both types of meshes.

1. Structured grids form a regular pattern with their mesh cells. Meaning all nodes are ordered, which simplifies the formulation of numerical schemes, e.g., loops and other accesses, since neighboring cells can be identified from the indices of the involved nodes.

2. Unstructured grids do not have this regularity allowing for a higher degree of freedom. This is used to better adapt to the shape of the flow domain Ω and especially the boundary. In addition, it can be easier to adapt the resolution to a finer scale in some crucial areas. This property can be seen in Figure 2.3, where the unstructured grid has a very fine resolution around the front and trailing edge of an airfoil. Nevertheless, unstructured meshes are more complicated for implementation since the connectivity between nodes has to be stored and can no longer be computed by simple node index arithmetic.

The test cases in this thesis use unstructured meshes and the unstructured SU2 solver, see Palacios, Colonno, et al. [93, Section 4]. In particular, an edge-based structure on a dual grid, as visualized in Figure 2.4. Median-dual control volumes are constructed by connecting the primal grid cell centers of all cells around one node and a vertex-based numerical scheme is formulated based on this dual mesh.

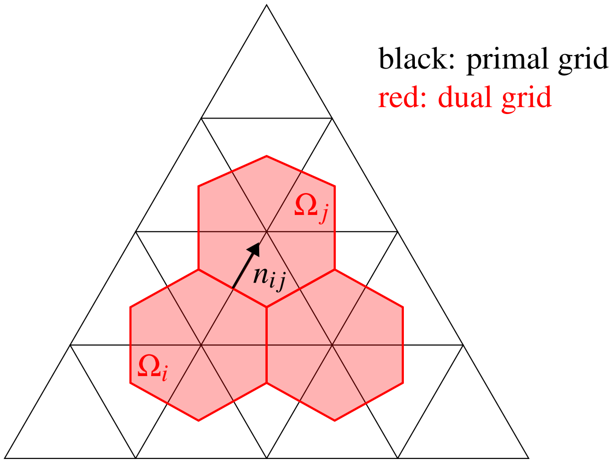


Figure 2.4: Schematic image of the primal and dual grid and control volumes.

2.2.2 Finite Volume Methods

The SU2 flow solver [35, 93] used for this work is based on the *finite volume method* [18, 76]. Here, a brief summary of the structure and principles of such schemes is given, as far as they are relevant for this work. For a detailed description of the numerical schemes, see Palacios, Colonno, et al. [93, Section 4]. The main idea of the finite volume method is to discretize the flow equation in two stages.

First, the spatial domain of the flow problem is discretized by a computational mesh, as introduced in the last Subsection 2.2.1. This leads to discrete values for the conservative variables in each control volume, therefore the name finite volume. Taking a look at the structure of the Navier-Stokes equations (2.8) in integral form, they should hold on each control volume Ω_i in the mesh,

$$\int_{\Omega_i} \frac{\partial}{\partial t} W \,\mathrm{d}x + \oint_{\partial\Omega_i} (F_C - F_V)\,\mathrm{d}s = \int_{\Omega_i} Q\,\mathrm{d}x. \tag{2.28}$$

Following the dual mesh based formulation in [93], the spatial terms in Equation (2.28) can be replaced by discrete quantities leading to a semi-discretized formulation

$$\frac{\partial}{\partial t}W_i|\Omega_i| + \sum_{j\in\mathscr{N}(i)} \left(\bar{F}_C(W_i,W_j) + \bar{F}_V(W_i,W_j)\right) - Q_i|\Omega_i| = \frac{\partial}{\partial t}W_i|\Omega_i| + R_i(W) = 0. \tag{2.29}$$

Here, the terms $\bar{F}_C(W_i,W_j)$ and $\bar{F}_V(W_i,W_j)$ represent the convective and viscous fluxes from the i-th to the j-th control volume and Q_i is a source term, also see Figure 2.4. Their exact computation varies depending on the finite volume scheme. For the test cases in this thesis, the *Jameson-Schmidt-Turkel* (JST) scheme is utilized. This scheme was originally developed for structured meshes [66], with newer versions for unstructured meshes being available [65, 63].
The important point is that the spatial terms can be combined into the residual $R_i(W)$, leading to an *ordinary differential equation* (ODE) in time. Equation (2.29) can then be solved by a numerical time integration method. Here, the implicit Euler method is shown. Although, it is important to remark that most implementations also offer explicit Runge-Kutta methods.

$$\frac{|\Omega_i|}{\delta t_i^n}\left(W_i^{n+1} - W_i^n\right) = -R_i(W^{n+1}) \tag{2.30}$$

The unknown value of the residual at the next time step $R(W^{n+1})$ can be computed by linearization

$$\begin{aligned} R_i(W^{n+1}) &= R_i(W^n) + \frac{\partial}{\partial t}R_i(W^n)\delta t_i^n + \mathscr{O}((\delta t_i^n)^2) \\ &= R_i(W^n) + \sum_{j\in\mathscr{N}(i)} \frac{\partial}{\partial W_j}R_i(W^n)(W_j^{n+1} - W_j^n) + \mathscr{O}((\delta t_i^n)^2). \end{aligned} \tag{2.31}$$

Inserting Equation (2.31) into Equation (2.30) results in a linear system of equations for the time update $\delta W_j^n = \left(W_j^{n+1} - W_j^n\right)$,

$$\left(\frac{|\Omega_i|}{\delta t_i^n}\delta_{ij} + \frac{\partial}{\partial W_j}R_i(W^n)\right)\delta W_j^n = -R_i(W^n). \tag{2.32}$$

At last, the solution for the next timestep can be computed with a linear equation solver. Due to the nature of the underlying equations, the matrix in Equation (2.32) is very high-dimensional and has a sparse structure. This means that iterative solvers are preferable, e.g., the Generalized Minimal Residual Method (GMRES) [102] or the Lower-Upper Symmetric-Gauss-Seidel Method (LU-SGS) [67].
The example above shows how finite volume methods can be used to compute a solution to the flow equations, yet the kind of solution one is interested in may vary. One import type are *steady state solutions*, that is solutions to Equation (2.29) where $\frac{\partial}{\partial t}W = 0$. Such situations arise naturally in many engineering and industrial applications, e.g., consider the case of a plane flying at a constant speed and altitude. Here, the source terms in the flow equation are constant over time and the solution at each position should have this property too. For this scenario, the ODE solver from Equation (2.30) is mathematically just a fixed point iteration to find the steady state solution.

Definition 2.2.1 (pseudo time-stepping)**.** *Assume that W^* is a steady state solution which solves $R_i(W^*) = 0$, then time iteration for the ODE from Equation* (2.29) *converges against W^*, if the numerical time integration method is stable, i.e.,*

$$\lim_{n\to\infty} W_i^n = W_i^*. \tag{2.33}$$

This procedure is known as pseudo time-stepping.

Remark 2.2.2. *From now on, it is assumed that the flow problems in this thesis have a steady state solution computed by pseudo time-stepping with a finite volume scheme.*
This solution can be represented by storing the values of W_i^* on all cells Ω_i into a vector which is denoted by $u \in \mathbb{R}^{n_u}$.

For finite volume methods, one usually assumes consistency and accuracy of the obtained solution. Let $W|_{\text{mesh}} \in \mathbb{R}^{n_u}$ be a vector containing the exact solution in the control volumes. Then the error in a suitable norm should be smaller than a given tolerance

$$||u - W|_{\text{mesh}}|| < tol_1. \tag{2.34}$$

Since the exact solution is oftentimes not available and experimental data is hard to obtain, mesh refinement studies are used instead. The basic idea is that a consistent method must converge against the exact solution if the size of the mesh cells approaches 0. Therefore, one constructs a series of refined meshes and accepts the solution if it does not change too much under further mesh refinement.

2.3 Fundamentals of Optimization

In this subsection, basic topics from mathematical optimization are summarized. The topics are ordered in four subsections, first defining the fundamental criteria for optimality in numerical optimization in Subsection 2.3.1. This is followed by a definition of the continuous shape Hessian in Subsection 2.3.2, which will help understand the motivation of approximating Hessian operator symbols later in this thesis. Then the role a mesh parameterization plays in describing the design and how this affects the optimization are discussed in Subsection 2.3.3, followed by a comparison of discrete and continuous optimization in Subsection 2.3.4.

2.3.1 Numerical Optimization

Since the topic of this thesis are methods for aerodynamic shape optimization and their applications, a major recurring point throughout this work will be concepts from mathematical optimization. Therefore, it seems appropriate to start by introducing the basic ideas and state fundamental theorems for optimization. These results can be found in many standard-issue textbooks. Within this thesis, the notation is inspired by the books of Nocedal and Wright [90, Chapter 12] and Luenberger and Ye [77, Chapter 11].

Definition 2.3.1 (optimization problem)**.** *Let* $F : \mathbb{R}^n \to \mathbb{R}$*,* $C_i : \mathbb{R}^n \to \mathbb{R}$ *for* $i \in \mathscr{I}_C$*,* $E_j : \mathbb{R}^n \to \mathbb{R}$ *for* $j \in \mathscr{I}_E$ *be functions,* $\mathscr{I}_C = \{1,\dots,n_C\}$ *and* $\mathscr{I}_E = \{1,\dots,n_E\}$ *be sets of indices, then*

$$\begin{aligned} &\min_{x \in S} F(x) \\ s.t.\ &\forall i \in \mathscr{I}_C :\ C_i(x) \geq 0 \\ &\forall j \in \mathscr{I}_E :\ E_j(x) = 0 \end{aligned} \tag{2.35}$$

is called a constrained optimization problem.

Here, x is the *optimization variable*, F the *objective function*, C_i and E_j the *inequality and equality constraint functions*, and $S \subset \mathbb{R}^n$ compact and simply connected the *design space*.
It is worth noting that the notation of the constraints can be simplified to get more convenient expressions when formulating optimization criteria and algorithms. For this, they are combined into vector-valued functions

$$C(x) := \begin{bmatrix} C_1(x) \\ \vdots \\ C_{n_C}(x) \end{bmatrix}, \quad E(x) := \begin{bmatrix} E_1(x) \\ \vdots \\ E_{n_E}(x) \end{bmatrix}. \tag{2.36}$$

A point $z \in S$ is called *feasible* if it fulfills the constraints, i.e.,

$$\forall i \in \mathscr{I}_C : C_i(z) \geq 0 \wedge \forall j \in \mathscr{I}_E : E_j(z) = 0. \tag{2.37}$$

Naturally, this defines the *feasible set*,

$$S_f = \{z \in S \mid \forall i \in \mathscr{I}_C : C_i(z) \geq 0 \wedge \forall j \in \mathscr{I}_E : E_j(z) = 0\}. \tag{2.38}$$

Throughout the rest of this thesis, the feasible set S_f is assumed to be compact and non-empty.
Also, in constrained optimization some of the inequality constraints might be precisely zero for a feasible point z. The notation of an *active set* $S_{\mathscr{A}}$ is often used to mark for which inequality constraint indices the constraint value is equal to zero

$$S_{\mathscr{A}} = \{i \in \mathscr{I}_C \mid C_i(z) = 0\} \cup \mathscr{I}_E. \tag{2.39}$$

Remark 2.3.2. *In this work, equality constraints are always assumed to be active.*

The fundamental question in optimization is whether or not a feasible solution to the optimization problem in Definition 2.3.1 exists and how to find it?
For an investigation of this question, possible solutions can be classified into two different types.

- A point $x^*_{\text{global}} \in S_f$ is called a *global optimum*, if it has the smallest objective function value among all feasible points
$$\forall z \in S_f : F(x^*_{\text{global}}) \leq F(z). \tag{2.40}$$
- A point $x^* \in S_f$ is called a *local optimum*, if
$$\exists \varepsilon > 0 : \forall z \in (S_f \cap B_\varepsilon(x^*)) : F(x^*) \leq F(z), \tag{2.41}$$
where $B_\varepsilon(x^*)$ is the ε-neighborhood around x^*, with respect to the standard Euclidean norm.

While it is known from basic analysis that a global optimum must exist if F is continuous and S_f is compact and non-empty, this does not state how such an optimum can be found. If F is a differentiable function, results from multi-dimensional analysis can be applied to give conditions that any locally optimal point must fulfill. For unconstrained optimization problems, there is a well-known necessary condition for optimality.

Proposition 2.3.3 (unconstrained first order necessary optimality condition)**.** *Assume $\mathscr{I}_C = \mathscr{I}_E = \emptyset$. Let x^* be a local minimum of the objective function F, then $\mathrm{D}_x F(x^*) = 0$.*

Remark 2.3.4. *There are many equivalent notations for Proposition 2.3.3 throughout the literature. It can also be stated in terms of the gradient $\nabla F(x^*) = \mathrm{D}_x F(x^*)^T = 0$, or the directional derivative in arbitrary directions v, i.e., $\forall v \in S : \mathrm{D}_x F(x^*)v = 0$.*

When dealing with a constrained optimization problem, there can be minima for which the derivatives of the objective function do not vanish. In fact, the gradient of F only has to be zero inside the feasible set S_f to fulfill the optimality condition, while on the boundary of the feasible set S_f, it is sufficient that all directional derivatives pointing inside the set are larger or equal zero.
This implicit dependency on the constraints can be problematic in practice since a closed formula for S_f is often unknown, and it is preferable to have a different formulation. The introduction of a Lagrangian function for the optimization problem allows for such an explicit formulation.

Definition 2.3.5 (Lagrangian function)**.** *The function*

$$L(x,\lambda) = F(x) + \lambda_E^T E(x) + \lambda_C^T C(x) \tag{2.42}$$

is called the Lagrange function or Lagrangian associated with the optimization problem in Definition 2.3.1. The entries of the vector $\lambda = [\lambda_E, \lambda_C] \in \mathbb{R}^{n_E+n_C}$ are called Lagrange multipliers.

While everything up to this point holds without further assumptions, additional regularity conditions are necessary to formulate optimality conditions with the Lagrange function.

Definition 2.3.6 (LICQ condition)**.** *For a local optimum x^* and active set $S_{\mathscr{A}}$, the gradients of all active constraints*

$$\{\mathrm{D}_x E_1(x^*), \ldots, \mathrm{D}_x E_{n_E}(x^*), \mathrm{D}_x C_1(x^*), \ldots, \mathrm{D}_x C_{|S_{\mathscr{A}}|}(x^*)\} \tag{2.43}$$

must be linearly independent. This is called the linear independent constraint qualification (LICQ) condition.

Using the Definitions 2.3.5 and 2.3.6 from above, the following famous theorem can be stated.

Theorem 2.3.7 (Karush-Kuhn-Tucker condition)**.** *Let $x^* \in S_f$ be a local minimum, $F(x), C(x), E(x)$ be continuously differentiable in x^*, and the LICQ condition from Definition 2.3.6 be fulfilled, then there exists a unique Lagrange multiplier λ^* such that*

$$\mathrm{D}_x L(x^*,\lambda^*) = \mathrm{D}_x F(x^*) + (\lambda_E^*)^T \mathrm{D}_x E(x^*) + (\lambda_C^*)^T \mathrm{D}_x C(x^*) = 0 \tag{2.44}$$

$$C(x^*) \geq 0 \tag{2.45}$$

$$E(x^*) = 0 \tag{2.46}$$

$$\forall i \in \mathscr{I}_C : \lambda_i^* \leq 0 \tag{2.47}$$

$$\forall i \in \mathscr{I}_C : (\lambda_i^*)^T C_i(x^*) = 0 \tag{2.48}$$

These equations are called the KKT (Karush-Kuhn-Tucker) conditions.

Proof. A proof for this theorem is given in many introductory textbooks for optimization, e.g., Nocedal and Wright [90, p. 331-342]. □

Remark 2.3.8. *The equations* $\forall i \in \mathscr{I}_C : (\lambda_i^*)^T C_i(x^*) = 0$ *imply that the Lagrange multipliers for inactive constraints are* 0*, and thus the values of these constraints do not effect the value of the Lagrangian* $L(x^*, \lambda^*)$*. This is known as complementarity.*

For twice differentiable functions, further optimality conditions can be stated using the Hessian matrix of the Lagrangian function $\mathrm{D}_{xx}L$. It is important to note that these are sufficient, and not only necessary, conditions. Once again, it is helpful to start with the condition statement for the unconstrained case.

Proposition 2.3.9 (unconstrained second order sufficient optimality condition)**.** *Let* $x^* \in S_f$ *be a critical point and let for all directions* $v \in \mathbb{R}^n$ *hold*

$$\begin{aligned} \mathrm{D}_x F(x^*) &= 0 \\ \text{and } \; v^T \mathrm{D}_{xx} F(x^*) v &> 0, \end{aligned} \tag{2.49}$$

then x^* *is a local minimum.*

Proof. See Luenberger and Ye [77, Section 7.3]. □

To state this condition for the constrained case, the Hessian matrix of the Lagrange function has to be positive semidefinite in all feasible directions. The feasible directions can be qualified by introducing a set

$$W := \left\{ w \in \mathbb{R}^n \;\middle|\; \begin{array}{l} \forall j \in \{1, \dots, n_E\} : \mathrm{D}_x E_j(x^*)^T w = 0 \\ \forall i \in S_{\mathscr{A}} \text{ with } \lambda_i^* < 0 : \mathrm{D}_x C_i(x^*)^T w = 0 \\ \forall i \in S_{\mathscr{A}} \text{ with } \lambda_i^* = 0 : \mathrm{D}_x C_i(x^*)^T w \geq 0 \end{array} \right\}. \tag{2.50}$$

This definition looks confusing at first, but essentially it consists of all directions in the tangent cone of the feasible set for which it is unclear whether F increases or not. So now, second order conditions can be stated for the constrained problem similar to the KKT conditions using the Lagrangian.

Theorem 2.3.10 (second order sufficient optimality condition)**.** *Let* x^*, λ^* *fulfill the KKT conditions and* $F(x), E(x), C(x)$ *be twice continuously differentiable in* x^**. If*

$$\forall w \in W, w \neq 0 \;:\; w^T \mathrm{D}_{xx} L(x^*, \lambda^*) w > 0 \tag{2.51}$$

then x^* *is a local minimum.*

Proof. See Nocedal and Wright [90, Theorem 12.6]. □

Almost all derivative-based optimization algorithms are designed to find points fulfilling these optimality conditions. The second order condition is superior in the sense that it is a sufficient condition, meaning it already implies optimality. However, it is considerably harder to computationally find points fulfilling Theorem 2.3.10, than to compute points fulfilling Theorem 2.3.7.

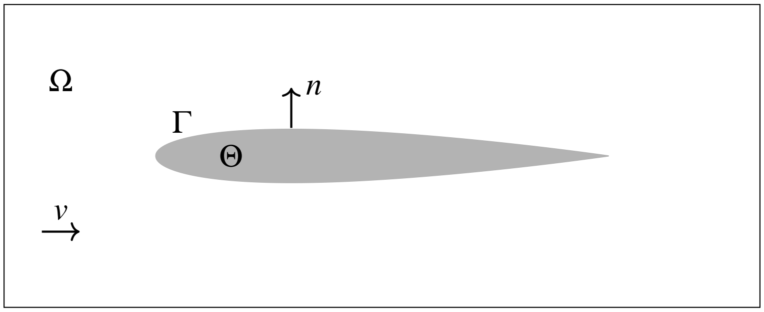


Figure 2.5: Flow area, design boundary, and flow obstacle.

2.3.2 Shape Calculus and Shape Hessians

In Subsection 2.3.1, the fundamentals of numerical optimization for a finite-dimensional setting are introduced. Nonetheless, this thesis is partially inspired by and deals with approximations of shape Hessians. Formulating a shape optimization problem in a continuous setting will be done in the following Subsection 2.3.4. But first, it is important to define differentiability for a functional $\mathscr{F}$ with respect to a domain Ω. Here, the introduction follows the book by Sokolowski and Zolesio [116, Chapter 2].

The situation is shown in Figure 2.5. It is generally assumed that the flow domains are from a set of admissible shapes, $\Omega \in S_\Omega \subset \mathscr{P}(\mathbb{R}^d)$. For the definition of shape derivatives, assume that Ω is a closed manifold with boundary Γ piecewise in class C^2, with intersections in C^1, see the discussion in [116, Chapter 2]. Also, the flow domain Ω is part of a compact, simply connected domain called the hold-all, i.e., $\Omega \subset \mathscr{M}(\Omega)$. Then let $\mathscr{F} : S_\Omega \to \mathbb{R}, \Omega \mapsto \mathscr{F}(\Omega)$ be a functional depending on the flow field in Ω. At first, a variation in the domain must be defined. For first order derivatives, the following class of deformations is used.

Definition 2.3.11 (perturbation of identity)**.** *Let $v \in C^2(\mathscr{M}(\Omega), \mathbb{R}^d)$ be a differentiable vector field, then*

$$\Omega(t)[v] := \{x + tv(x) | x \in \Omega\} \tag{2.52}$$

is a deformed domain.

Remark 2.3.12. *In Definition 2.3.11, the vector fields v have to be twice continuously differentiable to ensure that admissible shapes are mapped to admissible shapes. The boundary Γ being in class C^2, with intersections in C^1, is necessary in case the functional $\mathscr{F}$ is defined by a boundary integral on Γ. For more details on the regularity requirements see [116, Section 2.32].*

Using the perturbation of identity, it is possible to define the derivative canonically as the limit with respect to change in the domain.

Definition 2.3.13 (shape derivative)**.** *Let $v \in C^2(\mathscr{M}(\Omega), \mathbb{R}^d)$ be an arbitrary vector field, then the shape derivative or Eulerian derivative in direction v is defined as the one-sided limit*

$$\mathscr{D}\mathscr{F}(\Omega; v) = \lim_{t \to 0^+} \frac{\mathscr{F}(\Omega(t)[v]) - \mathscr{F}(\Omega)}{t}. \tag{2.53}$$

The functional $\mathscr{F}$ is called shape differentiable if:

1. *The shape derivative* $\mathscr{D}\mathscr{F}(\Omega;v)$ *exists for all* $v \in C^2(\mathscr{M}(\Omega),\mathbb{R}^d)$.
2. *The mapping* $\mathscr{D}\mathscr{F}(\Omega) : C^2(\mathscr{M}(\Omega),\mathbb{R}^d) \to \mathbb{R} : v \mapsto \mathscr{D}\mathscr{F}(\Omega;v)$ *is linear and continuous.*

With this notion of differentiability in place, one might be inclined to directly define the second order derivative as applying the shape differentiation twice.

Remark 2.3.14. *Applying the shape derivative in a direction twice does not lead to a well-defined second order derivative. The mapping is not bilinear in* v_1, v_2 *because the second perturbation of identity would be applied to an already deformed manifold.*

Instead, a new definition is necessary, being linear in two arguments, to define second order differentiation.

Definition 2.3.15 (double perturbation of identity)**.** *Let* $v_1, v_2 \in C^2(\mathscr{M}(\Omega),\mathbb{R}^d)$ *be differentiable vector fields, then*

$$\Omega(t_1,t_2)[v_1,v_2] := \{x + t_1 v_1(x) + t_2 v_2(x) | x \in \Omega\} \tag{2.54}$$

is a doubly deformed domain.

Definition 2.3.16 (second order shape derivative and shape Hessian)**.** *Let* $v_1, v_2 \in C^2(\mathscr{M}(\Omega),\mathbb{R}^d)$ *be two arbitrary vector fields, then the second order shape derivative in those directions is defined as*

$$\mathscr{D}^2\mathscr{F}(\Omega;v_1,v_2) = \lim_{t_2 \to 0^+} \lim_{t_1 \to 0^+} \frac{\mathscr{F}(\Omega(t_1,t_2)[v_1,v_2]) - \mathscr{F}(\Omega(t_1)[v_1]) - \mathscr{F}(\Omega(t_2)[v_2]) + \mathscr{F}(\Omega)}{t_1 t_2}. \tag{2.55}$$

This limit defines a bilinear mapping in the two arguments v_1, v_2.
Assume that $(\chi, \langle \bullet, \bullet \rangle)$ *is a suitable Hilbert space, with* $\chi \subset C^2(\mathscr{M}(\Omega),\mathbb{R}^d)$*, then there exists a unique representation*

$$\mathscr{D}^2\mathscr{F}(\Omega;v_1,v_2) = \langle \mathrm{Hess}\,\mathscr{F}(\Omega) v_1, v_2 \rangle. \tag{2.56}$$

The operator $\mathrm{Hess}\,\mathscr{F}(\Omega)$ *is commonly called the shape Hessian of* $\mathscr{F}$.

2.3.3 Shape Parameterization

For engineering applications, the design has to be stored and processed in a computer and thus can not be described in the mathematical terms of domains and manifolds. Instead, a parameterization has to be used, which is oftentimes done in the form of a *computer-aided design* (CAD) tool [112]. To better understand how the connection between shape, parameterization, and computational mesh is relevant for this work, these concepts are explained here in more detail.

1. The *shape* refers to the object itself, which is assumed to be a compact manifold Θ with a closed boundary Γ piecewise in class C^2, with intersections in C^1. Γ is also called the surface for obvious reasons, see the situation depicted in Figure 2.5. Subsection 2.3.4 discusses this setting and how to formulate an optimization problem there in more detail. However, it is challenging to do analytic computations on this level.

2. *Parameterization* itself is a word used in various contexts, from mathematics, physics, meteorology, and other sciences. Here, it is used in its geometric or mathematical meaning, which is giving a parametric equation to describe the geometry. A parametric equation refers to a description that gives the desired object as a function of a finite number of real-valued parameters. Usually, one also demands those equations to be explicit algebraic expressions, but implicit formulations are also possible.

3. The *computational mesh* is, by definition, a triangulation of the flow area Ω. Therefore, the surface cells are naturally a triangulation of Γ. For the present context, this means that a parameterization will consist of two combined mappings, one from the design parameters to the surface nodes of the computational mesh and one for the internal deformation of the volume mesh according to the surface movement.

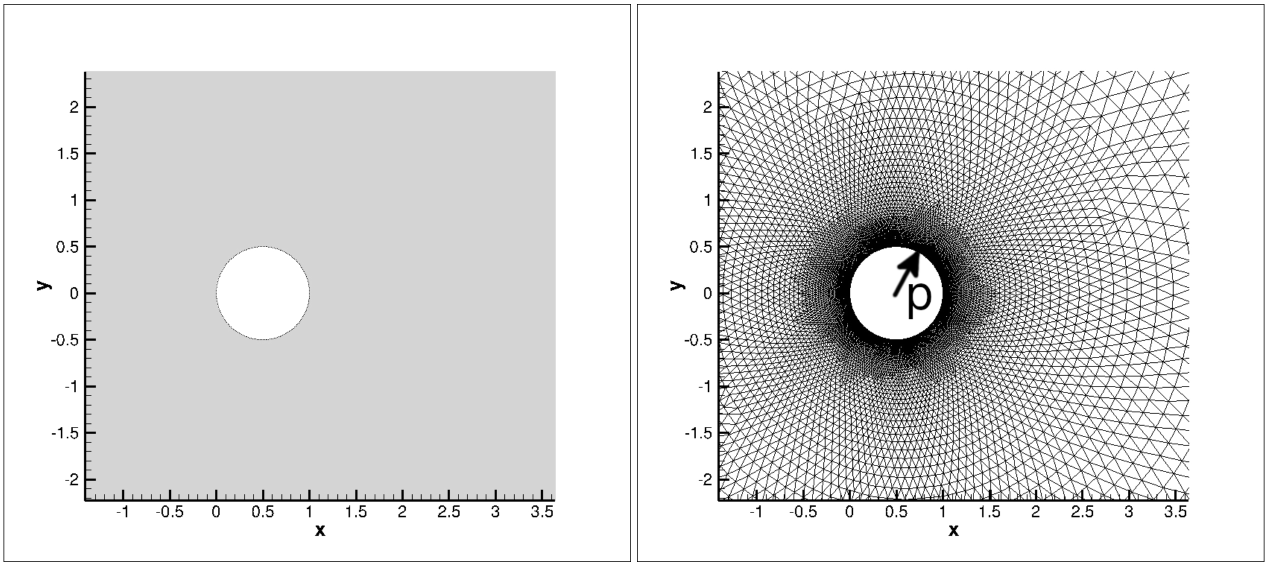


Figure 2.6: Example of a shape and the associated mesh.

To better visualize these points consider the trivial example shown in Figure 2.6, where the shape itself is just a circle with a fixed radius. Here, the design parameter is the radius p that describes the surface coordinates by the equation

$$\sqrt{(x-0.5)^2+y^2}=p \tag{2.57}$$

and an example of a mesh can be seen on the right side of the figure.

The key ideas introduced in this thesis are focused on this concept of parameterization. In particular, how parameterization has to be taken into account and influences the results when applying certain types of shape optimization techniques. When working on a discrete level, it is not apparently clear at what level of discretization the optimization is done. Two possibilities for this can be summed up as follows.

1. Using the computational mesh from the CFD solver as a discrete representation, see Section 2.2 for more details. The obvious advantage is that the discretized flow equations are already

expressed on this level, which helps with consistency. This approach can yield a very well attuned solution, but it is undesirable from an engineering viewpoint. An optimized mesh geometry can not easily be returned to an industrial design or manufactured and projecting it back into a manufacturing-friendly form typically leads to a performance loss.
Furthermore, using all node coordinates as a high-dimensional design space leads to problems with regularity and the optimization process will suffer from high-frequency noise if the movement of vertices is not restricted.

2. Alternatively, a parameterization can be used as a discrete representation of the geometry, meaning that the mesh and thereby the entire design depend on a vector of *design parameters* p. In this case, the use of gradient-based optimization requires further projections, via the derivatives of the parameterization. For example, CAD representations common throughout the industrial design process might be used in such a way.

For theoretical work and academic test cases, mathematicians prefer to directly work on the mesh coordinates. This is called the *free node* approach and simplifies the process since the computational mesh and the design parameters are equal, omitting the necessity of a parameterization. However, it should be clear by now that a parameterization is the norm in industrial applications and that a discrete set of parameters represents the object to be designed.
Naturally, there is an enormous amount of theory on the properties of such a description and how to find a good representation. The different types of parameterizations used in this work are introduced in the following. The objective is to highlight how the design parameters p are connected to the Cartesian coordinates of a computational mesh via parameterization and introduce their role in the formulation of shape optimization problems. Ultimately, this results in the next definition.

Definition 2.3.17 (mesh equation)**.** *Let* $p \in \mathbb{R}^{n_p}$ *be a vector of design parameters and* $M : \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ *be a differentiable function, such that for the mesh coordinates* $x \in \mathbb{R}^{n_x}$ *it holds that*

$$x = M(p), \tag{2.58}$$

then Equation (2.58) *is called the mesh equation.*

At this point, it should be mentioned that the parameterization consists of two steps. First, only the nodes s on the design surface Γ are directly determined via parameter changes

$$M_{\text{S}} : \mathbb{R}^{n_p} \to \Gamma; p \mapsto s. \tag{2.59}$$

This part is the surface parameterization. All the other node coordinates x in the flow volume Ω must be computed from s. This can be done by shifting the mesh nodes in a smooth deformation according to the change in surface coordinates

$$M_{\text{V}} : \Gamma \to \Omega; s \mapsto x. \tag{2.60}$$

The process is referred to as mesh deformation. In this thesis, a linear elasticity approach is used. This can be calculated by a finite elements approach, where the stiffness can be based on different settings, e.g., wall distance or inverse cell volume. A stiffness matrix $A \in \mathbb{R}^{n_s \times n_x}$ is set up and the resulting linear equation system is solved,

$$Ax = s. \tag{2.61}$$

Together, such a two-staged parameterization approach expresses Equation (2.58) by a successive computation

$$x = M(p) = M_{\mathrm{V}}(M_{\mathrm{S}}(p)). \tag{2.62}$$

In the rest of this section, the surface parameterizations M_{S} for the test cases in Chapter 7 are presented.

Hicks-Henne Functions

The first parameterization used in this thesis are *Hicks-Henne bump functions* [57]. While they are an old concept going back to the 70s, they are still a popular choice in research since they allow for an easy setup of test cases. Starting with them also allows the introduction of concepts in a simple formulation.

Consider an airfoil in two dimensions, as shown in Figure 2.2. A deformed airfoil can be created by adding a smooth bump function to the surface. Assume that an airfoil is scaled to a unit coordinate system υ_1, υ_2, such that the angle of attack is $\alpha = 0^\circ$ and the chord length is $l_c = 1$. This can be done using the transformation

$$\begin{pmatrix} \upsilon_1 \\ \upsilon_2 \end{pmatrix} = \begin{pmatrix} \cos(\alpha)\left(\frac{x}{l_c}\right) + \sin(\alpha)\left(\frac{y}{l_c}\right) \\ -\sin(\alpha)\left(\frac{x}{l_c}\right) + \cos(\alpha)\left(\frac{y}{l_c}\right) \end{pmatrix}. \tag{2.63}$$

Then the functions

$$f(\upsilon_1) = \left[\sin(\pi \upsilon_1)^{\frac{\log(0.5)}{\log(a)}}\right]^b, \quad \upsilon_1 \in [0,1] \tag{2.64}$$

are called Hicks-Henne bump functions, where a is the position of the maximum and b is the width of the bump. An exemplary sample of Hicks-Henne functions is shown in Figure 2.7.

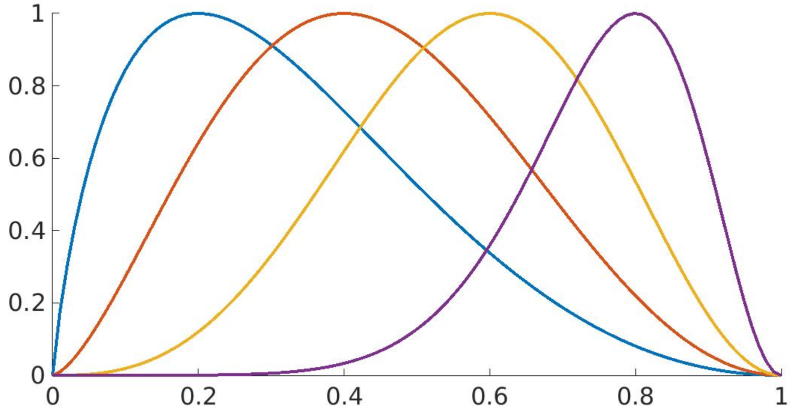


Figure 2.7: Multiple Hicks-Henne bump functions.

Given a number of these functions n_p, with maxima spaced equally across the upper and lower sides of the airfoil, new Cartesian coordinates (x, y) of the surface points can be calculated by adding

Hicks-Henne functions in the unit coordinate system and projecting them back. Define the auxiliary function

$$\Upsilon : [0,1] \to \mathbb{R}, \upsilon_1 \mapsto \begin{cases} \upsilon_2 = \upsilon_{2,0} + \sum_{i=0}^{\lfloor \frac{np}{2} \rfloor} p_i f_i(\upsilon_1), & \text{for } \upsilon_{2,0} \geq 0 \\ \upsilon_2 = \upsilon_{2,0} - \sum_{i=0}^{\lfloor \frac{np}{2} \rfloor} p_i f_i(\upsilon_1), & \text{for } \upsilon_{2,0} < 0 \end{cases}, \tag{2.65}$$

then the new Cartesian coordinates can be written as

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} l_c(\cos(\alpha)\upsilon_1 - \sin(\alpha)\Upsilon(\upsilon_1)) \\ l_c(\sin(\alpha)\upsilon_1 + \cos(\alpha)\Upsilon(\upsilon_1)) \end{pmatrix}. \tag{2.66}$$

The weights p_i used in this sum are the design parameters and form a vector $p = (p_1, \dots, p_{n_p})^T$, which uniquely defines any given design.
Extensions to three dimensions are possible by modifying different cross sections along a wing with Hicks-Henne functions and interpolating in between. Similar approaches are still used in industrial applications today. While Hicks-Henne functions are no longer the preferred method of choice, it is still common in turbomachinery design to describe a turbine blade with a series of airfoils and interpolation in between them. Such airfoils are then parameterized by quantities such as their chord length, the opening angle at the leading and trailing edge, and the curvature of the surface.

Free-Form Deformation

The second parameterization presented in detail is the *free-form deformation* (FFD) [111, 97]. This idea is based on embedding the geometry inside a cuboid with evenly distributed control points on the surface of the cuboid, see Figure 2.8. Deformation can be achieved by moving the control points and smoothly shifting the cuboid's interior. The original method uses Bernstein polynomials to construct a trivariate tensor, though other interpolations have been developed since, e.g., NURBS-based FFD, volume-preserving FFD, etc. In general, the surface of the deformed object can be seen as a hyper patch, an idea that is a generalization of the famous Bézier curves in two dimensions.

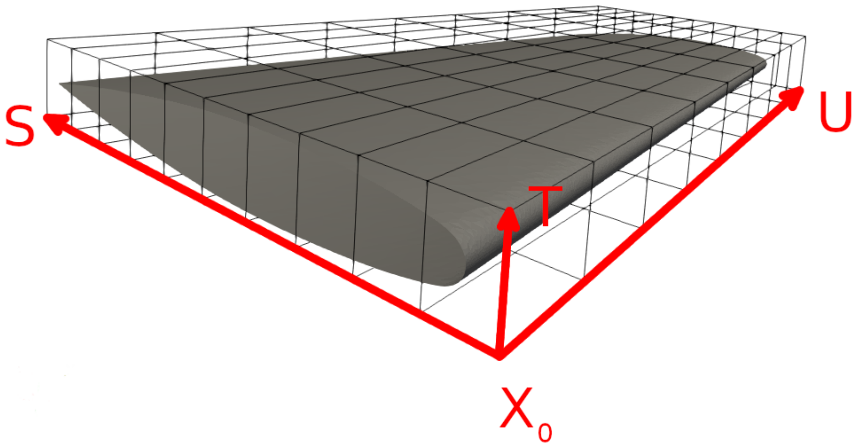


Figure 2.8: The free-form deformation box with a local coordinate system.

To achieve such a deformation with high enough derivative continuity, i.e., second order, the interpolation inside the FFD box must be constructed accordingly. Here, the original formulation is stated, as introduced by Sederberg [111]. Assume that a point X is given, then its position can be expressed in terms of a local coordinate system (S,T,U), as shown in Figure 2.8,

$$X = X_0 + aS + bT + cU. \tag{2.67}$$

The parametric coordinates (a,b,c) can be calculated for X by

$$a = \frac{\langle T \times U, (X - X_0)\rangle}{\langle T \times U, S\rangle}, \quad b = \frac{\langle S \times U, (X - X_0)\rangle}{\langle S \times U, T\rangle}, \quad c = \frac{\langle S \times T, (X - X_0)\rangle}{\langle S \times T, U\rangle}. \tag{2.68}$$

Distributing control points in an equidistant pattern across the FFD box gives the positions

$$P_{ijk} = X_0 + \frac{i}{n_S}S + \frac{j}{n_T}T + \frac{k}{n_U}U. \tag{2.69}$$

To calculate a deformed point X, the deformation is first applied to the control points P_{ijk}. The Cartesian coordinates for X are then found by evaluating Bernstein polynomials, with the parametric coefficients for the old position of X computed by Equations (2.68).

$$X_{ijk} = \sum_{i=0}^{n_S} \binom{n_S}{i} (1-a)^{(n_S-i)} (a)^i \left(\sum_{j=0}^{n_T} \binom{n_T}{j} (1-b)^{(n_T-j)} (b)^j \left(\sum_{k=0}^{n_U} \binom{n_U}{k} (1-c)^{(n_U-k)} (c)^k P_{ijk} \right) \right) \tag{2.70}$$

As mentioned, different basis functions for interpolation have been applied over the years to achieve certain levels of continuity, or other desirable properties, such as constant volume. Overall, FFD as a deformation technique has a lot of nice mathematical properties, allowing for application in optimization while still providing a powerful design tool. For this work, it is particularly relevant that an implementation of FFD using the Bernstein polynomials from Equation (2.70) can be easily differentiated. Furthermore, note that the transformation is linear in the control coordinates. A desirable property for the design process, where repeated shifts in the geometry should add up.

2.3.4 Basic Discrete and Continuous Shape Optimization

With the basic notation for numerical optimization and shape calculus in place, the next logical step is to introduce the concept of shape optimization [116, 84]. Shape optimization problems include many of the most crucial questions in engineering since most desired behaviors can be expressed by minimizing some function with respect to the shape under construction constraints. Considering aerodynamic shape optimization, it is apparent that a designer would be interested in optimizing the properties of the shape with respect to the flow surrounding it, as described by the flow equations. Some examples would be minimizing the drag of an aircraft, increasing the efficiency of a turbine blade, the noise reduction of wind energy turbines, and many more. Additional problems can be formulated based on other physical state equations, such as the optimization of structural properties like stiffness or weight, or the maximization of heat exchange on a cooling surface. In the case of aerodynamics, the relevant PDE is a flow equation, as introduced in Section 2.1, while for structural problems, it would be some form of stiffness equation. Also, in recent years the consideration of coupled problems, where one might have multiple interacting state equations, has become more and

more relevant.
In this Subsection, the connection between the two Subsections 2.3.1 and 2.3.2 is discussed in more detail. The objective is to find a methodology to apply the results from numerical optimization to a problem governed by differentiable shape functionals. As a starting point, to formulate the continuous optimization problems mathematically, it is necessary to define which shape is actually meant. In Section 2.1, the flow domain Ω already appeared when formulating the flow equations. Taking a look at Figure 2.5, several equivalent formulations are possible.

1. One could describe the shape by the flow domain Ω, which allows for an easy formulation of the state equations.

2. An engineer might be more interested in the flow obstacle Θ, i.e., the designed and built part.

3. Also, one can describe the entire shape by the design ´surface $\Gamma := \partial\Omega = \partial\Theta$. The functionals from Subsection 2.1.2 are typically integrals over this domain.

Here, the problem is set up in terms of Ω, although one should keep in mind that the descriptions using Θ or Γ are equally valid as well. Assuming that the PDE constraint $\mathscr{Q}(W,\Omega)$ has a solution of sufficient regularity, then Sokolowski and Zolesio [116, Section 2.32] state that the shape optimization problem can be written as follows.

Definition 2.3.18 (constrained shape optimization problem)**.** *Let $S_\Omega \subset \mathscr{P}(\mathbb{R}^d)$ be a set of admissible shapes, i.e., compact sets with a closed boundary Γ piecewise in class C^2, with intersections in C^1, and let $\mathscr{F} : S_\Omega \to \mathbb{R}, \Omega \mapsto \mathscr{F}(\Omega)$ be a real-valued shape functional and $\mathscr{Q}(W,\Omega)$ be a set of flow equations, then*

$$\begin{aligned} &\min_{\Omega,W} \mathscr{F}(\Omega) \\ &s.t.\ \mathscr{Q}(W,\Omega) = 0 \end{aligned} \tag{2.71}$$

is a shape optimization problem.

Solving this continuous functional formulation of the shape optimization problem has a couple of drawbacks attached to it.

1. How to generalize all the mathematical properties for optimality and optimization introduced in Subsection 2.3.1?
   While optimality and derivatives in a finite-dimensional $\mathbb{R}$ vector space are straightforward, their formulation in a general shape setting in a well-defined way is a very involved task [116].

2. The second question is how to formulate a suitable optimization algorithm for Definition 2.3.18?
   This is directly linked to the first point and requires much additional mathematical theory to do successfully. Usually, one would be interested in constructing an improving sequence of designs. However, mathematically it is not easy to formulate convergence for such a sequence.

3. At last, there remains the problem of how to compute numerical solutions to the involved equations?
   To put it briefly, one wants to do computations with the geometry and use numerical simulations to determine its properties. For this, a discrete representation of the shape must be stored in a computer.

A natural solution is to replace the whole continuous setting with a discrete formulation. In the case of aerodynamic shape optimization, this is closely linked to the discretization of the flow equations and their numerical solution. The evaluation of the aerodynamic functionals requires solving the discretized flow equations in any case and this is done numerically on a computational mesh, see Subsection 2.2.1.

Assume that there exists a computational mesh on the domain Ω, represented by the vector $x \in \mathbb{R}^{n_x}$ containing the coordinates of each of the mesh nodes, then the optimization problem 2.3.18 can be rewritten in discrete form. This means that the nature of the design variables changes, as the control is no longer a set Ω, but instead a discrete vector $x \in \mathbb{R}^{n_x}$ of mesh coordinates. Also, the discrete flow solution $u \in \mathbb{R}^{n_u}$ is a vector containing constant values for all state variables on each of the dual mesh cells, approximating the steady state solution W^*, similar to Subsection 2.2.2. Then it is possible to replace the continuous flow equation with a discrete approximation

$$\mathscr{Q}(W,\Omega) = 0 \Leftrightarrow H(u,x) = 0, \tag{2.72}$$

where the function $H : \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_u}$ is called the *discrete flow equation*. Solutions can be computed with finite volume methods and then used to evaluate the aerodynamic functionals necessary for the optimization process.

With this, the discrete definition of the shape optimization problem can be given.

Definition 2.3.19 (flow constrained discrete shape optimization problem)**.** *Let $x \in S_x \subset \mathbb{R}^{n_x}$ be the vector of coordinates of the mesh vertexes and $u \in S_u \subset \mathbb{R}^{n_u}$ the corresponding steady state solution for this mesh. In addition, let $F : \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}$ be a discretization of the functional $\mathscr{F}$, and $H(u,x)$ be the discrete flow equations, then*

$$\begin{aligned} &\min_{u,x} F(u,x) \\ &s.t. \;\; H(u,x) = 0 \end{aligned} \tag{2.73}$$

is called the discrete shape optimization problem.

This discretization changes the nature of the problem dramatically since all relevant functionals have been replaced by functions on finite-dimensional, real-valued vectors. Using the mesh coordinates directly as the control is called *free node optimization*.

Now, the optimality conditions from Subsection 2.3.1 can be applied, as long as the derivatives of F and H with respect to x are available. To compute them, one needs to solve the implicit Equation (2.72) for u first and then evaluate F. This process, and how to remove the cross dependencies on u, will be discussed at length in Chapter 3.

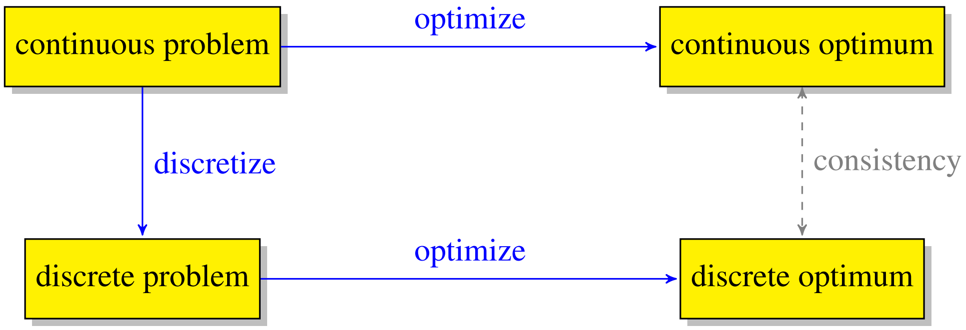


Figure 2.9: Principles of discrete vs. continuous optimization.

The discretization approach explained in this subsection is only part of a more fundamental distinction in the underlying philosophy on when it is best to apply discretization. Take a look at Figure 2.9. For aerodynamic shape optimization, two principally different methods are available. One follows the principle of ‘First discretize, then optimize’, where the results from Subsection 2.3.1 are applied to Definition 2.3.19 to find optimal solutions for the discrete problem. Such an approach will be used in this thesis and is extensively discussed when introducing the discrete adjoint methodology in Chapter 3. The alternative is to ‘First optimize, then discretize’, meaning shape calculus is applied to find an optimal solution to the continuous problem in Definition 2.3.18.

Chapter 3

Discrete Adjoint Optimization

This chapter introduces the adjoint methodology and presents its application to the shape optimization problem introduced in Subsection 2.3.4. To achieve this, the chapter introduces the discrete adjoint formulation for a free node optimization in Section 3.1 and continues with a discussion on the influence the shape parameterization has on the adjoint framework in Section 3.2. Section 3.3 shows how algorithmic differentiation is used to compute the derivatives in the adjoint equation efficiently. This knowledge is then used to formulate iterative adjoint solution strategies in Section 3.4, which are incorporated into a complete optimization algorithm in Section 3.5.

In Subsection 2.3.4, the differences between ‘First discretize, then optimize’ and ‘First optimize, then discretize’ were already discussed. These approaches lead to the two methodologies commonly known as the *discrete adjoint* and the *continuous adjoint* approach. Both have been extensively discussed and compared in the literature in the past [86, 46, 84].

The discrete adjoint approach is used throughout this work, as it allows for the application of very efficient, existing adjoint solvers based on algorithmic differentiation. Such solvers benefit from being compatible with a wide variety of different flow equations and providing a computationally efficient solution scheme [4, 5].

3.1 Adjoint Derivation for Free Node Optimization

The role of design parameterization in comparison to a free node formulation is a crucial point of discussion in this thesis. To better understand this, the discretized optimization problem 2.3.19 is combined with a parameterization of the computational mesh to form the following definition.

Definition 3.1.1 (parameterized discrete shape optimization problem)**.** *Let* $x \in S_x \subset \mathbb{R}^{n_x}$ *be the coordinates of a mesh triangulation of* Ω*,* $u \in S_u \subset \mathbb{R}^{n_u}$ *the discrete flow solution,* $F : \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}$ *be a discrete objective function,* $N : \mathbb{R}^{n_p} \times \mathbb{R}^{n_x} \to \mathbb{R}$ *be a parameterization of the computational mesh with parameters* $p \in S_p \subset \mathbb{R}^{n_p}$*, and* $H : \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_u}$ *a discretization of the flow equations on* Ω*, then*

$$\begin{aligned} &\min_{u,x,p} F(u,x) \\ &s.t.\;\; N(x,p) = 0 \\ &\qquad H(u,x) = 0 \end{aligned} \tag{3.1}$$

is called the parameterized discrete shape optimization problem.

The mesh parameterization can be written as an explicit equation

$$N(x,p) = 0 \Leftrightarrow x = M(p), \tag{3.2}$$

following Equation (2.58) and the reader might ask why this is not used immediately to remove the dependence on x? However, it is beneficial for the analysis in later chapters of this thesis to keep the dependency on the mesh coordinates when deriving the discrete adjoint approach in a free node formulation and apply the chain rule afterwards.
Also, certain aspects of the flow solver can be exploited to provide a robust and efficient adjoint framework for the optimization problem, crucially that the flow equations can be expressed in a fixed point formulation. This is because numerical solution schemes for conservation laws, e.g., flow equation, are usually iterative by design. For example, the finite volume methods applied in computational fluid dynamics have this property, as shown in Subsection 2.2.2. These schemes discretize the spatial derivatives first and then apply a pseudo time-stepping scheme, which results in a fixed point iteration for the state variables when computing a steady state solution. In a mathematical sense this means there exists a function $G : \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_u}$, such that

$$H(u,x) = 0 \Leftrightarrow u = G(u,x). \tag{3.3}$$

Throughout the rest of this work, the fixed point iteration G is assumed to be continuously differentiable in u and x and contractive in u, i.e.,

$$\|\mathrm{D}_u G(u,x)\|_2 < 1. \tag{3.4}$$

The fixed point formulation will be crucial throughout the rest of this work since it ensures the numerical stability of the adjoint solvers in this chapter and also helps in formulating the algorithms in Chapter 4.
For simplicity, the adjoint method is first formulated with the coordinates of the mesh nodes from the triangulation x of Ω as the design space in a free node optimization. The influence of the mesh equation (2.58) will be added afterwards in Section 3.2. Consider the simplified free node optimization problem of Definition 3.1.1 with the fixed point formulation of the flow constraint,

$$\begin{aligned} &\min_{u,x} F(u,x) \\ &s.t. \;\; G(u,x) = u. \end{aligned} \tag{3.5}$$

Assuming that the flow equation constraining the problem has a unique solution for a given mesh, derivative-based optimization techniques can be applied. In contrast to the basic optimization concepts in Subsection 2.3.1, the objective function $F(u,x)$ now depends on two vectors of variables u and x. However, this is not a true increase in the degrees of freedom since the variables are implicitly linked by the flow equation $G(u,x) = u$. Therefore, only x can be modified within the optimization process and the total derivative of the objective function with respect to x is required to formulate optimality conditions. This fixation of u by x via an equation can be exploited mathematically by applying the following theorem.

Theorem 3.1.2 (implicit function theorem)**.** *Let* $H : S_{(u,x)} \to \mathbb{R}^{n_u}, (u,x) \mapsto H(u,x)$ *be continuously differentiable on* $S_{(u,x)} \subseteq \mathbb{R}^{n_u} \times \mathbb{R}^{n_x}$, $H(u_*,x_*) = 0$ *at a point* $(u_*,x_*) \in S_{(u,x)}$, *and* $\mathrm{D}_u H$ *invertible in* (u_*,x_*). *Then there exists an open set* $B_\varepsilon(u_*) \times B_\varepsilon(x_*) \subseteq S_{(u,x)}$ *and a unique function* $\varphi : B_\varepsilon(x_*) \to B_\varepsilon(u_*)$, *such that:*

1. $\forall x \in B_\varepsilon(x_*) : H(\varphi(x),x) = 0$
2. $\mathrm{D}_x u_* = \mathrm{D}_x \varphi(x_*) = -\mathrm{D}_u H(\varphi(x_*),x_*)^{-1} \mathrm{D}_x H(\varphi(x_*),x_*)$.

Proof. A proof can be found in many introductory textbooks on analysis, e.g., Forster [42, Page 104]. □

Essentially, this theorem states that u can be locally expressed in terms of x and that there is a connection between the derivatives. The connection to the flow equation $H(u,x) = 0$ becomes apparent when examining the total derivative of the objective function. The total derivative with respect to the i-th component x_i, is given by the chain rule

$$\frac{\mathrm{d}}{\mathrm{d}x_i} F(x_i) = \frac{\partial}{\partial x_i} F(u,x) + \sum_{j=1}^{n_u} \frac{\partial}{\partial u_j} F(u,x) \frac{\partial}{\partial x_i} u_j. \tag{3.6}$$

The central issue with evaluating this expression for practical applications is the partial derivative of the flow variables with respect to the coordinates, i.e., the Jacobian $\mathrm{D}_x u$. This matrix can be evaluated in several ways.

1. It is possible to adequately evaluate the derivative by a *direct method*, e.g., finite differences, algorithmic differentiation, etc. However, since the matrix has dimensions $\mathbb{R}^{n_u} \times \mathbb{R}^{n_x}$ this process is expensive to compute, requiring $O(n_x)$ evaluations of the flow equation. Furthermore, since it is almost impossible to give an explicit expression for $u(x)$, it is hard to apply knowledge from analysis to simplify the expression.

2. The more computationally efficient way to solve this problem is to exploit the implicit function Theorem 3.1.2. This method is commonly known as the *adjoint method*.
   By the implicit function theorem $H(u,x) = 0$ and $\mathrm{D}_u H$ having full rank imply that a function $u(x)$ exists locally and that $\mathrm{D}_x u(x) = -\mathrm{D}_u H(u,x)^{-1} \mathrm{D}_x H(u,x)$. Inserted into Equation (3.6) this leads to

   $$\begin{aligned} \mathrm{D}_x F(u(x),x) &= \mathrm{D}_x F(u,x) - \mathrm{D}_u F(u,x) \mathrm{D}_u H(u,x)^{-1} \mathrm{D}_x H(u,x) \\ &= \mathrm{D}_x F(u,x) + \Psi^T \mathrm{D}_x H(u,x), \end{aligned} \tag{3.7}$$

   where Ψ is the solution of the *adjoint equation*

   $$\mathrm{D}_u H(u,x)^T \Psi = -\mathrm{D}_u F(u,x)^T. \tag{3.8}$$

   This approach allows the total derivative of the objective function to be computed by one linear equation system solve for Equation (3.8), instead of potentially n_x evaluations to compute $\mathrm{D}_x u$ directly.

Both formulations compute the derivative $\mathrm{D}_x F(u(x),x)$, as long as the flow equation can be solved. Using the results from Subsection 2.3.1, optimality conditions for constraint problems can be stated by introducing Lagrange multipliers and a Lagrange function, see Definition 2.3.5. The Lagrangian function in this case is

$$L : \mathbb{R}^{n_u} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}, (u, \lambda, x) \mapsto F(u,x) + \lambda^T (G(u,x) - u), \tag{3.9}$$

directly yielding the KKT conditions as formulated in Theorem 2.3.7:

$$\begin{aligned} &\mathrm{D}_\lambda L = 0 \Leftrightarrow u = G(u,x) && \text{(state equation)} \\ &\mathrm{D}_u L = 0 \Leftrightarrow \lambda = \mathrm{D}_u G(u,x)^T \lambda + \mathrm{D}_u F(u,x)^T && \text{(adjoint equation)} \\ &\mathrm{D}_x L = 0 \Leftrightarrow 0 = \mathrm{D}_x G(u,x)^T \lambda + \mathrm{D}_x F(u,x)^T && \text{(design equation)} \end{aligned} \tag{3.10}$$

Here, the LICQ condition from Definition 2.3.6 reduces to $\mathrm{D}_u(G(u,x) - u)$ having full rank, i.e., being invertible, in a local minimum. The use of a KKT formulation is a well-known approach for the adjoint setup and various gradient-based optimization frameworks exist to solve those equations [4]. The second equation in the KKT conditions and Equation (3.8) are both called adjoint equations since they are equivalent formulations of each other, as stated by the following corollary.

Corollary 3.1.3. *Let $\mathrm{D}_u(G(u,x) - u)$ be invertible, then for the two formulations of the adjoint equation, it holds that*

$$\mathrm{D}_u H(u,x)^T \Psi = -\mathrm{D}_u F(u,x)^T \Leftrightarrow \lambda = \mathrm{D}_u G(u,x)^T \lambda + \mathrm{D}_u F(u,x)^T \tag{3.11}$$

for the vectors $\Psi = \lambda$ being equal.

Proof.

$$\begin{aligned} &\mathrm{D}_u H(u,x)^T \Psi = -\mathrm{D}_u F(u,x)^T \Leftrightarrow (\mathrm{D}_u(G(u,x) - u))^T \Psi = -\mathrm{D}_u F(u,x)^T \\ &\Leftrightarrow (\mathrm{D}_u G(u,x) - \mathrm{I})^T \Psi = -\mathrm{D}_u F(u,x)^T \Leftrightarrow \Psi = \mathrm{D}_u G(u,x)^T \Psi + \mathrm{D}_u F(u,x)^T. \end{aligned} \tag{3.12}$$

Meaning both adjoint equations are equivalent for $\Psi = \lambda$. □

Since the Lagrangian is an exact penalty function finding a local minimum which fulfills the optimality conditions from Subsection 2.3.1 is equivalent to solving the original minimization problem with the flow constraint. In terms of a mathematical statement, this is expressed by the following corollary.

Corollary 3.1.4. *Assume that (u_*, x_*, λ_*) solve the flow equation $u_* = G(u_*, x_*)$ and the adjoint equation $\lambda_* = \mathrm{D}_u G(u_*, x_*)^T \lambda_* + \mathrm{D}_u F(u_*, x_*)^T$ respectively, then for the design equation it holds that*

$$\mathrm{D}_x L(u_*, \lambda_*, x_*) = \mathrm{D}_x F(u_*, x_*). \tag{3.13}$$

Proof. By definition

$$\mathrm{D}_x L(u_*, \lambda_*, x_*) = (\lambda_*)^T \mathrm{D}_x G(u_*, x_*) + \mathrm{D}_x F(u_*, x_*) \tag{3.14}$$

together with Corollary 3.1.3 and Equation (3.8)

$$\lambda_*^T \mathrm{D}_x G(u_*, x_*) + \mathrm{D}_x F(u_*, x_*) = \mathrm{D}_u F(u_*, x_*) \left(\mathrm{D}_u \left(G(u_*, x_*) - u_*\right)\right)^{-1} \mathrm{D}_x G(u_*, x_*) + \mathrm{D}_x F(u_*, x_*) \tag{3.15}$$

and by Theorem 3.1.2 it follows that

$$\mathrm{D}_u F(u_*, x_*) \left(\mathrm{D}_u \left(G(u_*, x_*) - u_*\right)\right)^{-1} \mathrm{D}_x G(u_*, x_*) + \mathrm{D}_x F(u_*, x_*) = \mathrm{D}_u F(u_*, x_*) \mathrm{D}_x u_* + \mathrm{D}_x F(u_*, x_*). \tag{3.16}$$

Since $u_* = u(x_*)$ solves the flow equation, it follows that total and partial derivatives with respect to x are the same, i.e.,

$$\mathrm{D}_u F(u_*, x_*) \mathrm{D}_x u_* + \mathrm{D}_x F(u_*, x_*) = \mathrm{D}_x F(u_*(x_*), x_*). \tag{3.17}$$

□

With the KKT conditions, it is possible to show the major stability advantage of the fixed point formulation for adjoint optimization.

Proposition 3.1.5. *If the fixed point operator of the numerical flow solver is contractive, i.e.,* $\|\mathrm{D}_u G(u,x)\|_2 < 1$, *then the fixed point iteration for the adjoint equation*

$$\lambda_{n+1} := \mathrm{D}_u G(u,x)^T \lambda_n + \mathrm{D}_u F(u,x)^T \tag{3.18}$$

converges to a unique solution

$$\lim_{n \to \infty} \lambda_n = \lambda^*. \tag{3.19}$$

Proof. The adjoint vector λ is in $\mathbb{R}^{n_u}$, which is a Banach space, and for the operator norm of the right-hand side of the adjoint equation, it holds that

$$\|\mathrm{D}_\lambda (\mathrm{D}_u G(u,x)^T \lambda + \mathrm{D}_u F(u,x)^T)\|_2 = \|\mathrm{D}_u G(u,x)\|_2. \tag{3.20}$$

Utilizing the contractivity of G from equation (3.4), Banach's fixed point theorem implies that the adjoint equation has a unique fixed point, see Forster [42, Page 103]. □

This means the adjoint fixed point iteration inherits the contractive behavior of the flow iteration. For stability, this is a major advantage compared to solving the linear system in Equation (3.8), which is known to be ill-conditioned in many cases [5].

3.2 Reduced Gradients and Design Parameterization

The adjoint formulation above yields a way to compute sensitivities for optimization with respect to the mesh coordinates. As explained in Subsection 2.3.4, discrete design optimization aims to find optimal design parameters p. Taking this into account, the design equation (3.10) must be adjusted by an additional projection step.

Here, the mesh equation $x = M(p)$ comes into account. Assuming the existence and uniqueness of the flow and adjoint solutions implies that the objective function solely depends on the design parameters. This means that it is possible to define a function

$$\tilde{F}(p) := F(u((M(p)), M(p)), \tag{3.21}$$

where the notation $u(M(p))$ refers to the implicit flow solution for the equation $u = G(u, M(p))$, which exists by the implicit function theorem 3.1.2. While an explicit analytic expression for $\tilde{F}$ is not available, this theorem also enables expressing the derivatives of $\tilde{F}$ in terms of F. Overall, the optimization problem from Definition 3.1.1 can alternatively be written as an unconstrained optimization problem

$$\min_{p \in S_p} \tilde{F}(p). \tag{3.22}$$

The derivatives of $\tilde{F}(p)$ are referred to as reduced derivatives. A similar approach for the Lagrangian allows the replacement of dependencies on x with dependencies on p when stating optimality conditions.

$$L : \mathbb{R}^{n_u} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}, (u, \lambda, p) \mapsto F(u, M(p)) + \lambda^T (G(u, M(p)) - u), \tag{3.23}$$

with the corresponding KKT conditions:

$$\begin{aligned}
&\mathrm{D}_\lambda L(u, \lambda, x) = 0 \Leftrightarrow u = G(u, x) && \text{(state equation)} \\
&\mathrm{D}_u L(u, \lambda, x) = 0 \Leftrightarrow \lambda = (\mathrm{D}_u G(u, x))^T \lambda + (\mathrm{D}_u F(u, x))^T && \text{(adjoint equation)} \\
&\mathrm{D}_x L(u, \lambda, M(p)) \mathrm{D}_p M(p) = 0 \Leftrightarrow && \text{(design equation)} \\
&\quad 0 = \left(\lambda^T \mathrm{D}_x G(u, M(p)) + \mathrm{D}_x F(u, M(p))\right) \mathrm{D}_p M(p).
\end{aligned} \tag{3.24}$$

Going a step further, if $\lambda(u(M(p)), M(p))$ denotes the corresponding solution of the adjoint equation from (3.24), a reduced Lagrangian can be formulated

$$\tilde{L} : \mathbb{R}^{n_p} \to \mathbb{R}, p \mapsto L(u(M(p)), \lambda(u(M(p)), M(p)), M(p)). \tag{3.25}$$

Application of the chain rule allows for the following formulation.

Corollary 3.2.1. *Let $\tilde{F}, \tilde{L}$ be defined as stated above, then*

$$\begin{aligned}
\mathrm{D}_p \tilde{F}(p) =& \mathrm{D}_x F(u, x) \mathrm{D}_p M(p) = \mathrm{D}_x F(u(M(p)), M(p)) \mathrm{D}_p M(p) \\
\mathrm{D}_p \tilde{L}(p) =& \mathrm{D}_x L(u, \lambda, x) \mathrm{D}_p M(p) \\
=& \left[\lambda(u(M(p)), M(p))^T \mathrm{D}_x G(u(M(p)), M(p)) + \mathrm{D}_x F(u(M(p)), M(p))\right] \mathrm{D}_p M(p),
\end{aligned} \tag{3.26}$$

and for the two reduced functions the equation

$$\mathrm{D}_p \tilde{L}(p) = \mathrm{D}_p \tilde{F}(p) \tag{3.27}$$

holds true.

Proof. If u, λ are the flow and adjoint solutions from Equation 3.24, then the stated result is just Corollary 3.1.4 together with the application of the chain rule for $x = M(p)$. □

Corollary 3.2.1 means the derivatives of the reduced functions can be computed by evaluating the KKT system. This implies that any derivative-based optimization algorithm that uses the adjoint methodology has to operate in three steps:

1. Create a mesh from the current design.

2. Solve the state and adjoint equations from the KKT system and evaluate the design equation.
3. Project the sensitivities back by evaluating Equation (3.26).

Traditionally, only the second step is considered in the development of adjoint solvers, while the other two are left as separate tasks provided by the optimization framework. However, this paradigm is changed to a certain degree in this work since the projection of sensitivities from the mesh to the design parameters will play a crucial role in formulating Sobolev smoothing for parameterized optimization later on.

The permanent switch between regular and reduced functions can be inconvenient for notation. To simplify this, it is possible to introduce the notion of a reduced gradient.

Definition 3.2.2 (reduced gradient)**.** *Let $F : \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}, (u,x) \mapsto F(u,x)$ be a function, where u,x are implicitly dependent on p by the mesh and flow equations, then*

$$\tilde{D}_p F(u,x) := \left(\mathrm{D}_x F(u,x) - \mathrm{D}_u F(u,x)(\mathrm{D}_u(G(u,x)-u))^{-1}\mathrm{D}_x G(u,x)\right)\mathrm{D}_p M(p) \tag{3.28}$$

is called the reduced gradient of F.

This is strongly linked to the adjoint calculus already introduced and the theory established so far directly leads to the following proposition.

Proposition 3.2.3. *For the reduced gradient from Definition 3.2.2 applied to the objective function $F(u,x)$ the following equations holds,*

$$\tilde{D}_p F(u,x) = \mathrm{D}_p \tilde{F}(p). \tag{3.29}$$

Until now, only KKT conditions, i.e., first order optimality conditions, are considered. However, Theorem 2.3.10 from Section 2.3.1 stated second order conditions as well, raising the question of what can be said about the Hessian when working with discrete adjoint methods.

Differentiating the Lagrangian function twice gives a condition for the Hessian with respect to u and x. The Hessian matrix can be split into four blocks.

$$\mathrm{D}_{(u,x)^2} L(u,\lambda,x) = \begin{bmatrix} \mathrm{D}_{uu}G(u,x)^T\lambda + \mathrm{D}_{uu}F(u,x)^T & \mathrm{D}_{ux}G(u,x)^T\lambda + \mathrm{D}_{ux}F(u,x)^T \\ \mathrm{D}_{ux}G(u,x)^T\lambda + \mathrm{D}_{ux}F(u,x)^T & \mathrm{D}_{xx}G(u,x)^T\lambda + \mathrm{D}_{xx}F(u,x)^T \end{bmatrix} \tag{3.30}$$

Theorem 2.3.10 introduced the set W, representing all feasible directions within a tangent cone of the constraints. Similarly, the flow constraint $G(u,x) = u$ and the implicit function Theorem 3.1.2 can be utilized to formulate the second order optimality condition in terms of x only.

Corollary 3.2.4 (adjoint second order optimality condition)**.** *Assume that for a critical point u^*, λ^*, x^* solve the flow, adjoint, and design equations respectively. If*

$$\forall v_x \in S_x : \; v_x^T Z^T \mathrm{D}_{(u,x)^2} L(u,\lambda,x) Z v_x > 0, \tag{3.31}$$

then x^* is a local minimum. Here,

$$Z = \left[(-\mathrm{D}_u G - \mathrm{I})^{-1}\mathrm{D}_x G, \mathrm{I}\right] \tag{3.32}$$

is the projection onto the feasible subspace via the implicit function theorem 3.1.2, and v_x are the feasible mesh movements.

There are several remarks worth pointing out about this result.

- The central matrix in this condition is the discretized equivalent of the continuous shape Hessian. Meaning for $\operatorname{Hess}\mathscr{F}(\Omega)$ from Definition 2.3.16, it holds that
$$\operatorname{Hess}\mathscr{F}(\Omega) \approx Z^T \mathrm{D}_{(u,x)^2} L(u,\lambda,x) Z. \tag{3.33}$$
- Usually, one assumes that the adjoint equation is fulfilled, leading to the following result. Let u^*, λ^* be the flow and adjoint solutions respectively, then
$$\mathrm{D}_{xx} L(u^*,\lambda^*,x) = Z^T \mathrm{D}_{(u,x)^2} L(u^*,\lambda^*,x) Z. \tag{3.34}$$
This matrix is referred to as the *reduced shape Hessian* in this thesis.
- Due to the high dimensions of u and x, this second order optimality condition is prohibitively expensive to compute for practical application test cases.

Finding a connection between $\mathrm{D}_{pp}\tilde{F}(p)$ and $D_{xx}L(u,\lambda,x)$ leads directly to the central results of this thesis. The computation of this result and a detailed analysis are presented in Chapter 5.
Here, the following subsections show how the KKT system can be solved in a computationally efficient way, to set up a fast and stable adjoint solver and use the resulting adjoint solution and sensitivities in an optimization algorithm.

3.3 Discrete Adjoints by Algorithmic Differentiation

The derivation of discrete adjoint methods in the previous sections introduced a multitude of derivative expressions. These derivatives' fast, accurate, and reliable computation is a crucial prerequisite for successful discrete adjoint optimization. In fact, it is naturally an important aspect of all derivative-based optimization algorithms. While various differentiation methods exist, one of the most accomplished ones is *algorithmic differentiation* (AD) [49, 87].
To understand how the discrete adjoint optimization benefits from AD, the core principles of the method are explained in more detail, which will help the reader to understand the algorithms presented throughout the rest of this thesis. Once the key ideas of AD are introduced, using them to evaluate the adjoint and design equations in the KKT system will become obvious.
In short, AD means differentiating a computer program, or more precisely, differentiating the elementary statements a computer program calculates and connecting them via the chain rule. Every numerical simulation code will necessarily be built out of simple mathematical statements like $(+,*,\sqrt{\bullet},\exp,\log)$, which for the rest of this section will be denoted by φ_k. Assuming for a function $f:\mathbb{R}^n \to \mathbb{R}^m, x \mapsto y$ there exists an implementation to evaluate the function value y, then this implementation can be seen as a series of elementary statements.
$$y = f(x) = \varphi_K \circ \varphi_{K-1} \circ \cdots \circ \varphi_1(x). \tag{3.35}$$

This can also be written as an algorithm with intermediate values $z_k = \varphi_k(z_{k-1})$.

Algorithm 3.3.1 function evaluation

```
$z_1 = \varphi_1(x)$
for $k = 2, \ldots, K-1$ do
    $z_k = \varphi_k(z_{k-1})$
$y = \varphi_K(z_{K-1})$
```

AD evaluates the derivatives for each of those elementary statements to compute the overall derivative of f. Let $f' := \mathrm{D}_x f$ be a shorthand notation for the derivative of f, then if written down a complete chain rule evaluation looks like

$$f'(x) = (\varphi_K' \circ \varphi_{K-1} \circ \cdots \circ \varphi_1(x))(\varphi_{K-1}' \circ \cdots \circ \varphi_1(x)) \circ \cdots \circ \varphi_1'(x), \tag{3.36}$$

or with intermediate values z_k

$$f'(x) = \bigcirc_{k=1}^{K} \varphi_k'(z_{k-1}) = \varphi_K'(z_{K-1}) \circ \cdots \circ \varphi_1'(x). \tag{3.37}$$

This formulation leaves two possible ways to evaluate the derivative expression. First, to evaluate Equation (3.37) from the inside out, called the *forward mode* of AD, also commonly referred to as tangent or tangent-linear mode. Here, the evaluation chain starts with the innermost statement $\varphi_1'(x)$, or if seen as a computer program with the first executed statement. Naturally, this requires a starting value for the derivatives $\dot{x}$, which is called the *seeding* of variable x and expressed in terms of Equation (3.37). Then the forward mode computes a value

$$\dot{y} = \left(\bigcirc_{k=1}^{K} \varphi_k'(z_{k-1}) \right) \dot{x}. \tag{3.38}$$

The procedure is also shown in Algorithm 3.3.2.

Algorithm 3.3.2 forward mode

```
$z_1 = \varphi_1(x)$
$\dot{z}_1 = \varphi_1'(\dot{x})$
for $k = 2, \ldots, K-1$ do
    $z_k = \varphi_k(z_{k-1})$
    $\dot{z}_k = \varphi_k'(z_{k-1})\dot{z}_{k-1}$
$y = \varphi_K(z_{K-1})$
$\dot{y} = \varphi_K'(\dot{z}_{K-1})$
```

It is possible to express the forward mode in terms of the Jacobian of the original function f,

$$\dot{y} = \mathrm{D}_x f \dot{x} = f'(x)\dot{x}, \tag{3.39}$$

meaning the forward mode of AD computes a matrix vector product between the Jacobian and the seeding vector $\dot{x}$.

The second way to evaluate the chain rule is by starting with the outermost statement. This is called the *reverse mode* of AD, also sometimes referred to as adjoint mode. It can be written down similarly to the forward mode yielding

$$\bar{x}^T = \bar{y}^T \left(\bigcirc_{k=1}^{K} \varphi_k'(z_{k-1}) \right), \tag{3.40}$$

or as a procedure shown in Algorithm 3.3.3.

Algorithm 3.3.3 reverse mode

```
$z_1 = \varphi_1(x)$
for $k = 2, \dots, K-1$ do
    $z_k = \varphi_k(z_{k-1})$
$y = \varphi_K(z_{K-1})$
$\bar{z}_{K-1}^T = \bar{y}^T \varphi_K'(z_{K-1})$
for $k = K-1, \dots, 2$ do
    $\bar{z}_{k-1}^T = \bar{z}_k^T \varphi_k'(z_{k-1})$
$\bar{x}^T = \bar{z}_1^T \varphi_1'(x)$
```

The critical difference to the forward mode is that the sensitivity information in Algorithm 3.3.3 is passed through the source code in reverse order, flipping around all the data dependencies. This idea has been discovered several times by different authors [47]. In terms of the Jacobian, the reverse mode computes the transposed matrix vector product from the forward mode

$$\bar{x}^T = \bar{y}^T \mathrm{D}_x f = \bar{y}^T f'(x). \tag{3.41}$$

Some important observations can be drawn from the two modes of AD.

- Both modes offer matrix vector products with the Jacobian, thus the full derivative matrix can be calculated by n forward mode or m reverse mode evaluations.

- In reverse mode, the intermediate values z_k are required in the second loop to calculate the derivatives. This can lead to increased memory consumption or the need to recompute intermediate values. A whole subfield of AD, so-called *checkpointing techniques*, is concerned with balancing runtime and memory for such situations, see Griewank and Walther [49, Chapter 12].

- In the formulation of adjoint methods in Sections 3.1 and 3.2, many matrix vector products with the transposed of different Jacobians appeared throughout the formulas. Indeed, this connection is the primary reason why the reverse mode of AD is such an efficient tool for implementing discrete adjoint optimization algorithms.

Overall, it can be concluded that both modes of AD have their respective fields of application. For example, the reverse mode is heavily utilized in training neural networks for machine learning, where it is called backpropagation [14].

The most important advantage AD offers is precision. In contrast to other techniques, AD calculates the derivatives of a computer program up to machine precision. This means that if an implementation for a function f is available, the use of AD will compute a machine accurate derivative of the

implemented routine. This property is especially beneficial in numerical applications, where equations are solved approximately since it always guarantees that the derivatives are consistent with the corresponding approximated function values.
After introducing the theoretical modes, a look at the actual implementation is necessary. In Algorithms 3.3.2 and 3.3.3 the general procedures are stated, yet it is unclear how to translate them into actual source code. How can an AD tool, i.e., a program that takes an implementation of Algorithm 3.3.1 and creates an implementation of Algorithms 3.3.2 or 3.3.3, be built?
Two prominent approaches for this are widely used, both accompanied by their own sets of benefits and contraries.

- *Operator overloading* is based on the idea of overloading the elementary statements to do the computation directly. This requires the programming language to support the overloading of operators and functions for custom data types. Overloading is a common feature in programming languages like C++, where a custom AD type can replace the floating-point type in the original code. For this AD type all of the elementary statements are overloaded to not only compute the value $z_k = \varphi_k(z_{k-1})$, but to also evaluate the derivative $\dot{z}_k = \varphi_k'(z_{k-1})\dot{z}_{k-1}$ as well or to store the reverse statement $\bar{z}_{k-1}^T = \bar{z}_k^T \varphi_k'(z_{k-1})$ for later evaluation. Operator overloading tools are normally easier to maintain for a large project. Once the floating-point types are exchanged, the operator overloading tool automatically keeps itself up to date with any source code changes made to the original code. The original and the differentiated codes are compiled simultaneously, instead of having to process every change of the original code by separate measures. However, this benefit in maintenance comes at a cost since it is significantly harder, if not impossible, to optimize the code structure for the reverse AD evaluation, e.g., reordering of functions, loop restructuring, etc. Additionally, the overloading can lead to less compiler optimization and problems with the memory layout and access patterns.
  In the implementations for this thesis, the operator overloading tool CoDiPack is used for AD [103]. A more detailed overview of this tool can be found in Subsection 6.1.2 and aspects of the application of AD will be discussed as a part of Section 6.2.

- *Source transformation* uses the whole source code for the function evaluation as its input. A source transformation tool usually reads the source files and outputs a new source file, containing code for the derivative calculation. The source transformation method is nonintrusive by definition, as the original source files are not changed in any way. This allows the AD tool to perform a lot of possible code optimizations and to do massive changes in the code structure first. Nonetheless, such advantages are hard to exploit since all the knowledge to handle any possible form of source code must be built into the AD tool. In general, source transformation tools have potentially higher performance for the derivative evaluation, but buy this ability by being much more restrictive to the kinds of source code they accept and by being significantly more challenging to implement. For example, as of 2021, there are very efficient tools for C or Fortran code, but still no source transformation tool can handle heavily templated C++ code. A common example of source transformation tools is Tapenade [53] developed by the french INRIA institute.

The actual implementation and application of both types of AD tools are a research field in their own right. To conclude this section, the points discussed so far are compiled into a short comparison of AD in contrast to other derivative computation techniques. Also see Figure 3.1, where some of these different key concepts for derivative computation approaches are visualized.

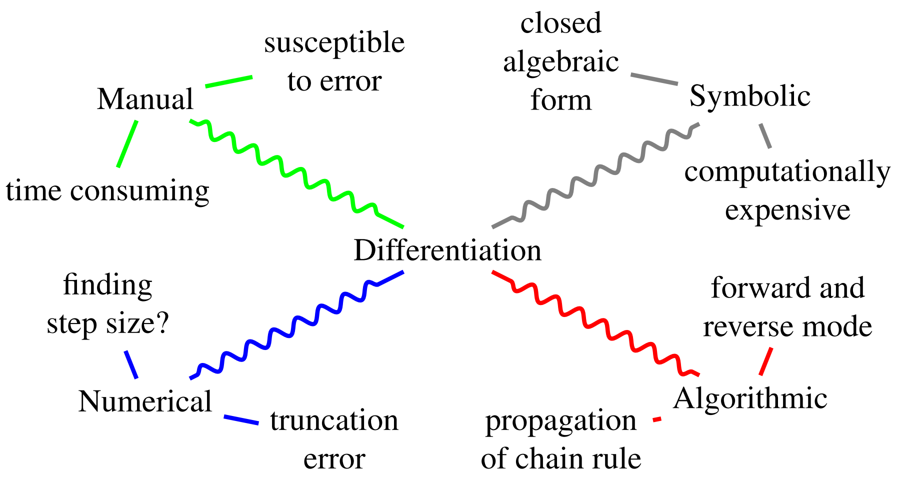


Figure 3.1: Different methods for differentiation.

- AD does not form a difference quotient of the change in function value between two points, like *finite differences* would do, thereby avoiding any issues with rounding errors. Note however, that the forward mode of AD introduced in this section is equivalent to the concept of dual numbers, in particular complex step approximation [80].

- AD does not *symbolically* differentiate, as it does not work on the complete mathematical function, but rather on individual statements and their implementation. To clarify this point, take a look at the statement $x = x + 1$. While this is unsolvable, if seen as a mathematical equation, it is trivial to handle as a computer statement from an AD perspective by setting $\dot{x} = \dot{x}$ or $\bar{x} = \bar{x}$.

- AD does not *manually* differentiate. Instead, the AD tool works completely autonomously, in contrast to the programmer manually deriving the derivative and then implementing a function for it, thus gaining a considerable advantage in development time and cost.

These points should clarify the contrast between AD and other common methods for implementing derivatives. Now that the fundamental concepts of AD are established, recall the KKT system from Equation (3.10). As the names suggest, the adjoint equation from the discrete adjoint approach and the adjoint mode of AD are closely connected. In fact, when analyzing the structure of the adjoint fixed point equation, the transposed Jacobian $\mathrm{D}_u G(u,x)^T$ of the flow iteration is at its center. This is ideal for employing the reverse mode of AD, with the current adjoint vector λ as seeding. See Equation (3.41) for comparison. In Section 3.4, different ways to formulate an algorithm around this idea are introduced. Using the AD methods shown here, these algorithms are designed to solve the adjoint and design equations to get accurate derivatives.

3.4 Discrete Adjoint Algorithms and Reverse Accumulation

In this subsection, different strategies for solving the adjoint equation of the KKT conditions (3.10) are introduced. They are based on applying the algorithmic differentiation techniques from Section

3.3 to the discrete adjoint framework from Section 3.1. All of these strategies aim at computing a solution λ^* of the adjoint fixed point equation

$$\lambda^* = \mathrm{D}_u G(u,x)^T \lambda^* + \mathrm{D}_u F(u,x)^T, \tag{3.42}$$

but differ in their implementation. As a starting point the numerical scheme for the flow solver, to calculate a steady state solution with pseudo time-stepping, is formulated in pseudocode in Algorithm 3.4.1.

Algorithm 3.4.1 Flow solver

```
input Initial values $x, u_0$
for i=0,...,I-1 do
    $u_{i+1} = G(u_i, x)$
$y = F(u_I, x)$
return $u_I, y$
```

Throughout this work, iterative processes in algorithms are written as for-loops. In an actual implementation, these would be while-loops with suitable convergence criteria to determine that $u_I \approx u_*$. This notation was chosen since the number of flow iterations I is important in formulating the adjoint algorithms in the rest of this chapter. In practice, I is set to whatever step number the flow convergence criteria are fulfilled.

At the core of the adjoint fixed point equation is a matrix vector product with the Jacobian of the flow iteration $\mathrm{D}_u G(u,x)^T \lambda$. Recalling Section 3.3, this term can be calculated by the reverse mode of AD, which is the main reason why the reverse mode of AD is so efficient when working with discrete adjoint methods.

The first algorithm treats the whole flow solver and the evaluation of the objective function as a black box and differentiates the whole process. In pseudocode, this results in Algorithm 3.4.2.

Algorithm 3.4.2 Black box algorithm

```
input Initial values $x, u_0, \bar{y}$
start recording of the AD tape ↻
for i=0,...,I-1 do
    $u_{i+1} = G(u_i, x)$
$y = F(u_I, x)$
end recording
start evaluation of the AD tape ↺
$\bar{u}_I = \mathrm{D}_u F(u_I, x)^T \bar{y}$
$\bar{x} = \mathrm{D}_x F(u_I, x)^T \bar{y}$
for i=I-1,...,0 do
    $\bar{u}_i = \mathrm{D}_u G(u_i, x)^T \bar{u}_{i+1}$
    $\bar{x} += \mathrm{D}_x G(u_i, x)^T \bar{u}_{i+1}$
end evaluation
return $u_I, y, \bar{x}$
```

No other variables are taken into account when calculating y, so it is clear that for $\bar{y} = 1$ the value of $\bar{x}$ is exactly the derivative $\mathrm{D}_x L(u,x)$ required in an optimization process, i.e., differentiating the flow solver G by reverse mode AD in black box fashion yields the required derivative. This can be seen by the actions of the AD tool depicted in gray. The whole flow simulation is recorded and then evaluated using the reverse mode, see Algorithm 3.3.3. Analysis of the black box algorithm reveals that the dependencies on the variables u_i are an issue in practice, as they lead to high computational costs. For extensive simulations, a single flow solution is quite large in terms of memory consumption and the necessity to reuse all intermediate u_i values in the second loop thus leads to an unfeasibly high cost. See the previous Section 3.3 and the checkpointing techniques mentioned there for a possible solution. However, checkpointing still increases the memory and runtime significantly. It would therefore be highly preferable to have a better solution.

The most widely adopted practical solution is the so-called *reverse accumulation* approach, as formulated by Christianson [28, 29]. This idea allows for the efficient differentiation of iterative procedures, as long as they are exactly converged to a fixed point. The procedure is shown in Algorithm 3.4.3.

Algorithm 3.4.3 Reverse Accumulation algorithm

```
input Initial values $x, u_0, \bar{y}$
for i=0,...,I-1 do
    $u_{i+1} = G(u_i, x)$
start recording of the AD tape ⟳
$u^{I+1} = G(u^I, x)$
$y = F(u_I, x)$
end recording
for j=0,...,J-1 do
    $\lambda_{j+1} = \mathrm{D}_u G(u_I, x)^T \lambda_j + \mathrm{D}_u F(u_I, x)^T \bar{y}$
$\bar{x} = \mathrm{D}_x G(u_I, x)^T \lambda_J + \mathrm{D}_x F(u_I, x)^T \bar{y}$
return $u_I, \lambda_J, \bar{x}$
```

Basically, the algorithm linearizes the algorithmic differentiation of an iterative process in the last step, i.e., when convergence is reached, resulting in the following theorem.

Theorem 3.4.1 (Reverse Accumulation)**.** *Let the two iterations*

$$u_{i+1} = G(u_i, x) \text{ and } \lambda_{k+1} = \mathrm{D}_u G(u_I, x)^T \lambda_k + \mathrm{D}_u F(u_I, x)^T \bar{y} \tag{3.43}$$

be strict contractions, i.e., $\|G(u,x)\| < 1$ *and* $\|\mathrm{D}_u G(u,x)\| < 1$. *Furthermore, let the solution of the flow iteration in Algorithm 3.4.3 be a fixed point* u_*, *i.e.,*

$$\lim_{I \to \infty} u_I = u_*. \tag{3.44}$$

Then for Algorithm 3.4.2 the following limit exists

$$\lim_{I \to \infty} \left(\mathrm{D}_x F(u_I, x)^T \bar{y} + \sum_{i=0}^{I-1} \mathrm{D}_x G(u_i, x)^T \bar{u}_{i+1} \right) = \bar{x}_{BB}, \tag{3.45}$$

and for Algorithm 3.4.3 the following limit exists

$$\lim_{I\to\infty}\lim_{J\to\infty}\left(\mathrm{D}_x G(u_I,x)^T\lambda_J + \mathrm{D}_x F(u_I,x)^T\bar{y}\right) = \bar{x}_{RA}. \tag{3.46}$$

Both limits have the same value, meaning both algorithms calculate the same result

$$\bar{x}_{BB} = \bar{x}_{RA}. \tag{3.47}$$

Proof. See the convergence proof of Algorithm 3.1 in the paper by Christianson [29]. □

Clearly, this approach is highly beneficial in terms of memory. Using a good AD tool, only one recording of an AD tape, for the flow solver in the converged fixed point, is necessary, as shown in gray in Algorithm 3.4.3. Then, the adjoint iteration can be done by repeated tape evaluation. However, the flow solver must be convergent up to machine precision for Theorem 3.4.1 to apply. For industrial test cases, this is often impossible due to numerical approximations in the finite volume schemes and test case settings, which can cause serious convergence issues for the computation of adjoints with reverse accumulation. Numerical stabilization techniques for this situation have been proposed in the past, e.g., in a paper by Albring, Dick, and Gauger [3].

Viewing the two algorithms under the aspect of solving the KKT conditions, it becomes obvious they both sequentially evaluate the KKT conditions. First, the flow equation and then the adjoint equation are solved. The design equation is evaluated afterwards, allowing for an optimization step based on $\bar{x}$ to update the design.

An interesting alternative approach is to solve the fixed point equations for flow and adjoint together in a coupled iteration. This is called the *piggyback method* since the adjoint is set up on top of the flow solver itself. As a mathematical equation, this is a new fixed point problem

$$\begin{bmatrix} u \\ \lambda \end{bmatrix} = \begin{bmatrix} G(u,x) \\ \mathrm{D}_u G(u,x)^T\lambda + \mathrm{D}_u F(u,x)^T \end{bmatrix}. \tag{3.48}$$

The AD tool can record the evaluation of the flow iterator, i.e., the first line of the equation, and immediately evaluate one adjoint step, i.e., the second line.

Algorithm 3.4.4 Piggyback algorithm

```
input Initial values $x, u_0, \bar{y}$
for i=0,...,I-1 do
    start recording of the AD tape ↻
    $u_{i+1} = G(u_i,x)$
    $y = F(u_i,x)$
    end recording
    start evaluation of the AD tape ↺
    $\lambda_{i+1} = \mathrm{D}_u G(u_i,x)^T\lambda_i + \mathrm{D}_u F(u_i,x)^T\bar{y}$
    end evaluation
$\bar{x} = \mathrm{D}_x G(u_I,x)^T\lambda_I + \mathrm{D}_x F(u_I,x)^T\bar{y}$
return $u_I, \lambda_I, \bar{x}$
```

The main difference is at what point the derivatives of G and F are evaluated. Reverse accumulation evaluates them exclusively in the converged fixed point u_*, while piggyback uses the current intermediate value of the state variable u_i. For reference, see how in each piggyback iteration in Algorithm 3.4.4 the AD tool records the flow solver and then immediately evaluates the tape, as shown in gray. An additional distinction is that the objective function F is now evaluated in every step. However, the overhead can be neglected since the cost is normally small compared to the flow iteration G. A detailed convergence analysis of the piggyback iteration can be found in the paper by Griewank and Faure [48, Corollary 3]. For the purpose of this work, it is sufficient to remark that if the contractivity assumptions stated in Theorem 3.4.1 are fulfilled, then Algorithms 3.4.3 and 3.4.4 converge towards the same fixed point values for $u_*, \lambda_*, \bar{x}$.

A final note to end this subsection is that there are two possible strategies for setting up the piggyback iteration. The first uses the flow solution from the previous step in the adjoint iteration, resulting in the equation

$$\lambda_{i+1} = \mathrm{D}_u G(u_i, x)^T \lambda_i + \mathrm{D}_u F(u_i, x)^T \bar{y}. \tag{3.49}$$

This is referred to as a Jacobi type iteration. The second always uses the latest information available. The first part of the loop in Algorithm 3.4.4 calculates u_{i+1}, so this value is clearly available and the adjoint could take the form

$$\lambda_{i+1} = \mathrm{D}_u G(u_{i+1}, x)^T \lambda_i + \mathrm{D}_u F(u_{i+1}, x)^T \bar{y}. \tag{3.50}$$

In the literature, this is called a Seidel type iteration. For comparison, see the similarity with iterative Jacobi and Seidel solvers for linear equation systems, e.g., in the book by Saad [101, Chapter 4].

3.5 Reduced SQP Optimizer

Optimizing highly complex and nonlinear problems, like the ones typical in aerodynamic shape optimization, requires close attention when designing the optimization algorithm. In this section, the results achieved so far are combined to introduce a fast and efficient optimization framework. The adjoint problem in Equation (3.42) can be solved by any of the algorithms from Subsection 3.4 and then reduced gradients can be computed, as shown in Subsection 3.2. Furthermore, additional constraints must be considered since they are practically always required in larger application test cases.

In Chapter 5, a new Hessian approximation technique, using parameterized Sobolev gradient reinterpretation, is presented. Currently, it is enough to assume that an approximation of the second order derivatives is available for the optimization algorithm, fitting naturally into a Quasi-Newton setting. Quasi-Newton methods are a powerful tool in optimization, whenever the second order derivatives of the Lagrangian function are not directly available, but some level of approximation can be obtained. One of the more general approaches is the *sequential quadratic programming* (SQP) framework [90, Chapter 18]. In particular, a partially reduced SQP approach is used, based on the previous work of Schulz [108] and Schmidt [104]. For a start, consider the definition of the complete optimization problem with additional equality and inequality constraints, see Equation (2.36).

Definition 3.5.1 (constrained shape optimization problem)**.** *Using the notation from Definition 3.1.1, let there be sets of additional equality constraints* $E : \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_E}$ *and inequality constraints*

$C : \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_C}$, then the problem

$$\begin{aligned}
&\min_{u,x,p} F(u,x) && \textit{(objective function)}\\
&s.t.\ \ M(p) = x && \textit{(mesh equation)}\\
&\qquad G(u,x) = u && \textit{(flow equation)}\\
&\qquad E(u,x) = 0 && \textit{(equality constraint)}\\
&\qquad C(u,x) \geq 0 && \textit{(inequality constraint)}
\end{aligned} \tag{3.51}$$

is called the constrained shape optimization problem.

The key idea of an SQP method is to find an optimal point fulfilling the first and second order optimality conditions from Propositions 2.3.3 and 2.3.10, by solving a sequence of simplified, approximate problems. In particular, the Lagrangian is approximated by a second order (quadratic) Taylor expansion and the constraints are approximated by their first order (linear) Taylor expansions. Since inequality constraints require extra care when formulating the method, the derivation is first shown only considering equality constraints.

Consider the extended Lagrangian function depending on the flow state u, the adjoint state λ, the parameters p, and the multiplier θ. Here, the mesh equation is explicitly evaluated. This is defined by

$$\begin{aligned}
L^{\text{Ext}} :&\mathbb{R}^{n_u} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_E} \times \mathbb{R}^{n_p} \to \mathbb{R},\\
&(u,\lambda,\theta,p) \mapsto F(u,M(p)) + \theta^T E(u,M(p)) + \lambda^T (G(u,M(p)) - u).
\end{aligned} \tag{3.52}$$

According to Subsections 3.1 and 3.2, KKT conditions can be formulated. However, since the mesh equation is already implicitly inserted in the definition of the extended Lagrangian, additional care is required when forming derivatives for the KKT conditions. For brevity, the dependence on p is sometimes omitted, writing x instead of $M(p)$ again.

$$\begin{aligned}
&\mathrm{D}_\lambda L^{\text{Ext}}(u,\lambda,\theta,x) = 0 \Leftrightarrow u = G(u,x) && \text{(state equation)}\\
&\mathrm{D}_u L^{\text{Ext}}(u,\lambda,\theta,x) = 0 \Leftrightarrow && \text{(adjoint equation)}\\
&\quad \lambda = (\mathrm{D}_u G(u,x))^T \lambda + (\mathrm{D}_u F(u,x))^T + (\mathrm{D}_u E(u,x))^T \theta &&\\
&\mathrm{D}_\theta L^{\text{Ext}}(u,\lambda,\theta,x) = 0 \Leftrightarrow 0 = E(u,x) && \text{(equality constraint)}\\
&\mathrm{D}_x L^{\text{Ext}}(u,\lambda,\theta,M(p))\mathrm{D}_p M(p) = 0 \Leftrightarrow && \text{(design equation)}\\
&\quad 0 = \left(\lambda^T \mathrm{D}_x G(u,M(p)) + \mathrm{D}_x F(u,M(p)) + \theta^T \mathrm{D}_x E(u,M(p))\right) \mathrm{D}_p M(p) &&
\end{aligned} \tag{3.53}$$

An optimization algorithm needs to calculate a solution to these KKT conditions. One possible approach is to apply a Newton method, see [104, Section 8.3]. The expressions for such a Newton step can be simplified to a certain extent by replacing dependencies on x with dependencies on p.

$$\begin{bmatrix}
\mathrm{D}_{uu}L^{\text{Ext}} & \mathrm{D}_{up}L^{\text{Ext}} & (\mathrm{D}_u G - \mathrm{I})^T & (\mathrm{D}_u E)^T\\
\mathrm{D}_{pu}L^{\text{Ext}} & \mathrm{D}_{pp}L^{\text{Ext}} & (\mathrm{D}_p G)^T & (\mathrm{D}_p E)^T\\
\mathrm{D}_u G - \mathrm{I} & \mathrm{D}_p G & 0 & 0\\
\mathrm{D}_u E & \mathrm{D}_p E & 0 & 0
\end{bmatrix}
\begin{bmatrix}\delta u\\ \delta p\\ \delta\lambda\\ \delta\theta\end{bmatrix}
=
\begin{bmatrix}-(\mathrm{D}_u L^{\text{Ext}})^T\\ -(\mathrm{D}_p L^{\text{Ext}})^T\\ -H(u,p)\\ -E(u,p)\end{bmatrix} \tag{3.54}$$

This Jacobian matrix contains the Hessian of the extended Lagrangian function as its upper left block. Also, note that it is a mixed Hessian with derivatives with respect to u and p. Assuming

$(\mathrm{D}_u G - \mathrm{I})$ to be invertible, the expression can be simplified significantly using the reduced gradient from Definition 3.2.2.

$$\tilde{D}_p F = \left(\mathrm{D}_x F - \mathrm{D}_u F (\mathrm{D}_u G - \mathrm{I})^{-1} \mathrm{D}_x G\right) \mathrm{D}_p M(p) \tag{3.55}$$

For the second order derivatives of the Lagrangian $\tilde{D}_{pp} L^{\mathrm{Ext}}$, a similar expression for the reduced shape Hessian is derived, using projections similar to the ones seen in Corollary 3.2.4. Let B be an approximation of this reduced Hessian matrix with respect to p, then this fulfills

$$\begin{bmatrix} (-\mathrm{D}_u G - \mathrm{I})^{-1} \mathrm{D}_p G & \mathrm{I} \end{bmatrix} \begin{bmatrix} \mathrm{D}_{uu} L^{\mathrm{Ext}} & \mathrm{D}_{up} L^{\mathrm{Ext}} \\ \mathrm{D}_{pu} L^{\mathrm{Ext}} & \mathrm{D}_{pp} L^{\mathrm{Ext}} \end{bmatrix} \begin{bmatrix} (-\mathrm{D}_u G - \mathrm{I})^{-1} \mathrm{D}_p G \\ \mathrm{I} \end{bmatrix} = \tilde{D}_{pp} L^{\mathrm{Ext}} \approx B \tag{3.56}$$

and if used in Equation (3.54) the linear system simplifies to

$$\begin{bmatrix} B & \tilde{D}_p E^T \\ \tilde{D}_p E & 0 \end{bmatrix} \begin{bmatrix} \delta p \\ \delta \theta \end{bmatrix} = \begin{bmatrix} -(\tilde{D}_p L^{\mathrm{Ext}})^T \\ -E \end{bmatrix}. \tag{3.57}$$

The last expression of interest is $\tilde{D}_p E$. Since the evaluation of the constraint is similar to the objective function, evaluating the reduced gradient means solving an adjoint problem for each of the constraints. A Lagrangian function, including the flow equation, can be constructed accordingly for each of them.

$$\forall k \in \{1, \ldots, n_E\} : L^{E_k}(u, \lambda^{E_k}, p) = E(u, M(p))_k + (\lambda^{E_k})^T (G(u, M(p)) - u) \tag{3.58}$$

Now the reduced gradient can be computed by Corollary 3.2.1 and Proposition 3.2.3,

$$\tilde{D}_p E_k(u, x) = \tilde{D}_p L^{E_k}(u, \lambda^{E_k}, p). \tag{3.59}$$

The fact that the dependency on the flow variables themselves has completely vanished from the formulation by using the discrete adjoint method gives this approach its name, *reduced SQP* method. While SQP methods were initially designed to use the exact second order derivatives $\tilde{D}_{pp} L^{\mathrm{Ext}}$, they have advanced since and are a general tool using all sorts of approximated Hessian matrices in a Quasi-Newton like fashion.

To better demonstrate the connection to the classical SQP algorithm, consider the quadratic optimization problem used in each optimization step.

$$\begin{aligned} &\min_{v \in S_v} \frac{1}{2} v^T \tilde{D}_{pp} L^{\mathrm{Ext}} v + \tilde{D}_p F v && \text{(objective function)} \\ &s.t.\ \ \tilde{D}_p E v + E = 0 && \text{(equality constraint)} \end{aligned} \tag{3.60}$$

Here, $S_v \subset \mathbb{R}^{n_p}$ is a suitable set of feasible design updates. Minimizing such a quadratic approximation of the Lagrangian under linearized constraints is the basic idea of SQP methods and the quadratic optimization problem has a unique solution $(v, \mu)^T$, fulfilling the relation

$$\begin{aligned} \tilde{D}_{pp} L^{\mathrm{Ext}} v + \tilde{D}_p F^T + \tilde{D}_p E^T \mu &= 0 \\ \tilde{D}_p E v + E &= 0. \end{aligned} \tag{3.61}$$

This can be written as a linear equation system, which is then solved in each optimization step.

$$\begin{bmatrix} \tilde{D}_{pp} L^{\mathrm{Ext}} & \tilde{D}_p E^T \\ \tilde{D}_p E & 0 \end{bmatrix} \begin{bmatrix} v \\ \mu \end{bmatrix} = \begin{bmatrix} -\tilde{D}_p F^T \\ -E \end{bmatrix} \tag{3.62}$$

Since $B \approx \tilde{D}_{pp}L^{\mathrm{Ext}}$, the solutions of Equations (3.57) and (3.62) can be identified with each other, resulting in

$$\begin{aligned} \delta p &= v \\ \theta + \delta\theta &= \mu. \end{aligned} \tag{3.63}$$

The remaining open question is if the linear system always has a unique solution? The following theorem can guarantee this.

Theorem 3.5.2. *Let v, μ fulfill the KKT conditions for the quadratic problem* (3.60)*, such that*

1. *For the current p, the rows of $\tilde{D}_pE$ are linear independent. This is equivalent to the LICQ condition 2.3.6.*
2. *$\forall k = \{1, \dots, n_E\}$ and $\forall w \neq 0$ with $\tilde{D}_pE_k^T w = 0$, it holds that $w^T \tilde{D}_{pp}L^{Ext}w > 0$.*

Then the matrix in Equation (3.62) *is invertible.*

Proof. This theorem, together with a detailed proof, can be found in the book by Geiger and Kanzow [44, Theorem 5.28]. ☐

Now that the process is clear, the overall optimization algorithm can be formulated. Algorithm 3.5.1 shows the full optimization procedure for an equality constraint problem.

Algorithm 3.5.1 reduced SQP method for equality constraint optimization

```
input Initial design variables $p_0$, set iteration counter $i = 0$
while err $\geq$ tol do
    $x_i = M(p_i)$                                              ▷ compute a deformed mesh
    for $j = 0, 1, ..., J_u$ do                                 ▷ solve the flow equation
        $u_{j+1} = G(u_j, x_i)$
    $u_i = u_{J_u}$;  $y = F(u_i, x_i)$
    for $j = 0, 1, ..., J_\lambda$ do                           ▷ solve the adjoint equations
        $\lambda_{j+1} = \mathrm{D}_uG(u_i, x_i)^T \lambda_j + \mathrm{D}_uF(u_i, x_i)^T$
    $\lambda_i = \lambda_{J_\lambda}$
    for $k = 0, 1, ..., n_E$ do
        for $j = 0, 1, ..., J_k$ do
            $\lambda_{j+1}^{E_k} = \mathrm{D}_uG(u_i, x_i)^T \lambda_j^{E_k} + \mathrm{D}_uE_k(u_i, x_i)^T$
        $\lambda_i^{E_k} = \lambda_{J_k}^{E_k}$
    $\tilde{D}_pF = \left(\lambda_i^T \mathrm{D}_xG(u_i, x_i) + \mathrm{D}_xF(u_i, x_i)\right) \mathrm{D}_pM(p_i)$      ▷ evaluate the design equations
    for $k = 0, 1, ..., n_E$ do
        $\tilde{D}_pE_k = \left((\lambda_i^{E_k})^T \mathrm{D}_xG(u_i, x_i) + \mathrm{D}_xE_k(u_i, x_i)\right) \mathrm{D}_pM(p_i)$
    solve: $\begin{bmatrix} B_i & \tilde{D}_pE^T \\ \tilde{D}_pE & 0 \end{bmatrix} \begin{bmatrix} v \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} -\tilde{D}_pF^T \\ -E \end{bmatrix}$      ▷ compute design update
    $p_{i+1} = p_i + v$                                         ▷ update the design
    $i = i + 1$
return $p_i$
```

Returning to Definition 3.5.1, the only step missing is the introduction of inequality constraints into Algorithm 3.5.1. From the literature [90, Chapter 18], it is known that the quadratic problem assembled in each SQP step in Equation (3.60) has to be extended. Doing so leads to the formulation of a convex optimization problem.

$$\begin{aligned} \min_{v \in S_v} \ & \frac{1}{2} v^T B v + \tilde{D}_p F v \\ s.t. \ & \tilde{D}_p E v + E = 0 \\ & \tilde{D}_p C v + C \geq 0 \end{aligned} \tag{3.64}$$

While it is no longer possible to calculate an optimum for this by solving a linear system of equations, it is still possible to apply a convex optimizer for the subproblem in each step of the SQP algorithm. The expanded SQP algorithm is then given as an extension of Algorithm 3.5.1. Overall, Algorithm 3.5.2 shows how this is set up, following the general layout of an SQP method.
Some important points about this algorithm should be remarked upon.

1. The flow solution is still the steady state solution computed by a standard iterative finite volume flow solver.

2. For the adjoint solution, the reverse accumulation Algorithm 3.4.3 is used to obtain fast and consistent solutions.

3. A solution for the quadratic problem (3.64) can be calculated by any convex optimization package, many of which come equipped with special solvers for this kind of problem. The inequality constraints can be treated by an active set approach, where only the constraints close to zero are taken into account when computing the update v.

4. In practice, a globalization strategy for the optimizer may be necessary. Furthermore, it is oftentimes helpful to do a line search on the solution of the quadratic problem v. This is done if the quadratic approximation is not very accurate with respect to the overall problem, or if the length of v is large, since then the optimizer generally should check if the proposed step v indeed yields good descent.

This thesis applies the reinterpretation of the gradient in a Sobolev space, by parameterized smoothing, to compute a matrix B and apply it in the SQP algorithms shown above. It thereby replaces traditional gradient descent optimizations with an approximated Newton algorithm for aerodynamic shape optimization problems.

Algorithm 3.5.2 reduced SQP method for mixed constraint optimization

```
input Initial design variables $p_0$, set iteration counter $i = 0$
while err $\geq$ tol do
    $x_i = M(p_i)$                                                  ▷ compute a deformed mesh
    for $j = 0, 1, ..., J_u$ do                                     ▷ solve the flow equation
        $u_{j+1} = G(u_j, x_i)$
    $u_i = u_{J_u};\ \ y = F(u_i, x_i)$
    for $j = 0, 1, ..., J_\lambda$ do                               ▷ solve the adjoint equations
        $\lambda_{j+1} = \mathrm{D}_u G(u_i, x_i)^T \lambda_j + \mathrm{D}_u F(u_i, x_i)^T$
    $\lambda_i = \lambda_{J_\lambda}$
    for $k = 0, 1, ..., n_E$ do
        for $j = 0, 1, ..., J_k$ do
            $\lambda_{j+1}^{E_k} = \mathrm{D}_u G(u_i, x_i)^T \lambda_j^{E_k} + \mathrm{D}_u E_k(u_i, x_i)^T$
        $\lambda_i^{E_k} = \lambda_{J_k}^{E_k}$
    for $l = 0, 1, ..., n_C$ do
        for $j = 0, 1, ..., J_l$ do
            $\lambda_{j+1}^{C_l} = \mathrm{D}_u G(u_i, x_i)^T \lambda_j^{C_l} + \mathrm{D}_u C_l(u_i, x_i)^T$
        $\lambda_i^{C_l} = \lambda_{J_l}^{C_l}$
    $\tilde{D}_p F = \left(\lambda_i^T \mathrm{D}_x G(u_i, x_i) + \mathrm{D}_x F(u_i, x_i)\right) \mathrm{D}_p M(p_i)$      ▷ evaluate the design equations
    for $k = 0, 1, ..., n_E$ do
        $\tilde{D}_p E_k = \left((\lambda_i^{E_k})^T \mathrm{D}_x G(u_i, x_i) + \mathrm{D}_x E_k(u_i, x_i)\right) \mathrm{D}_p M(p_i)$
    for $l = 0, 1, ..., n_C$ do
        $\tilde{D}_p C_l = \left((\lambda_i^{C_l})^T \mathrm{D}_x G(u_i, x_i) + \mathrm{D}_x C_l(u_i, x_i)\right) \mathrm{D}_p M(p_i)$
    solve the quadratic problem: $\min_{v \in S_v} \frac{1}{2} v^T B_i v + \tilde{D}_p F v$      ▷ compute design update
                                 $s.t.\ \ \tilde{D}_p E v + E = 0$
                                 $\tilde{D}_p C v + C \geq 0$
    $p_{i+1} = p_i + v$                                             ▷ update the design
    $i = i + 1$
return $p_i$
```

Chapter 4

One Shot Optimization

One Shot optimization simultaneously computes solutions for all nonlinear equations in the KKT system, i.e., the flow, adjoint, and design equations [23, 43, 52, 119]. Different variations of the general One Shot idea have been deployed in various settings for PDE constrained optimization in the past [55, 24] and a detailed overview and discussion of the different One Shot approaches can be found in [23].

Contrary to the algorithms presented in Section 3.5, which calculate converged state and adjoint solutions and use them to update the design, One Shot methods apply a simultaneous update in the flow, adjoint, and design variables, intending to perform a simultaneous analysis and design (SAND). This gives the algorithm excellent theoretical potential for an overall speedup in runtime if individual design updates are computationally cheap. However, the necessity to operate with approximated, i.e., non-converged, objective function and gradient values will increase the number of optimization steps. Overall, this trade-off must be carefully balanced to exploit the One Shot optimization to its maximum potential.

Section 4.1 will derive the basic equations for the One Shot method. Then, the actual algorithms are formulated in Section 4.2, partially utilizing the AD-based discrete adjoint algorithms from Chapter 3. Finally, the incorporation of additional constraints into the One Shot algorithm is discussed in Section 4.3.

4.1 Derivation of the One Shot Method

The previous Section 3.4 introduced a series of algorithms to evaluate the adjoint equation and use the result in the design equation. With the sensitivities from the design equation in the KKT system (3.10), it is possible to do a descent step on the mesh coordinates in a free node fashion. If feasibility in the flow and adjoint states in each design step is left aside for a moment, one might combine this with the piggyback method (3.48) into a single iteration

$$\begin{bmatrix} u_{i+1} \\ \lambda_{i+1} \\ x_{i+1} \end{bmatrix} = \begin{bmatrix} G(u_i, x_i) \\ \mathrm{D}_u G(u_i, x_i)^T \lambda_i + \mathrm{D}_u F(u_i, x_i)^T \\ x_i - P_i^{-1} \left(\mathrm{D}_x G(u_i, x_i)^T \lambda_i + \mathrm{D}_x F(u_i, x_i)^T \right) \end{bmatrix}. \tag{4.1}$$

Here, P_i is some suitable preconditioner. This iteration establishes a coupled fixed point formulation for flow, adjoint, and design to solve all three KKT conditions at once. It is therefore named One Shot

method.
Before introducing further analysis, it is worth pointing out that this iteration might be interpreted from different angles. For example, it is possible to view Equation (4.1) as a Quasi-Newton method for the KKT system, e.g., in the papers by Hamdi and Griewank [51, 52].
An alternative interpretation is to view Equation (4.1) as an explicit Euler step for finding stationary points of the following differential algebraic equation

$$\begin{bmatrix} u_{i+1} \\ \lambda_{i+1} \\ \frac{\mathrm{d}}{\mathrm{d}t} x \end{bmatrix} = \begin{bmatrix} G(u_i, x_i) \\ \mathrm{D}_u G(u_i, x_i)^T \lambda_i + \mathrm{D}_u F(u_i, x_i)^T \\ -\mathrm{D}_x L(u, \lambda, x)^T \end{bmatrix}. \tag{4.2}$$

Since the fixed point equations for flow and adjoint are already derived from pseudo time-stepping via an iterative finite volume method, it is possible to view Equation (4.1) as a simultaneous pseudo time-stepping for the state, adjoint, and design. Many authors have suggested this in the past, including Ta'asan [120] and Hazra, Schulz, Brezillion, and Gauger [55].
Both approaches are equivalent and in the following, the analysis uses the Quasi-Newton interpretation. To simplify the notation, it is helpful to define the *shifted Lagrangian* function

$$N(u, \lambda, x) = L(u, \lambda, x) + \lambda^T u = F(u, x) + \lambda^T G(u, x), \tag{4.3}$$

which allows for the expression of the original KKT conditions in a shortened form:

$$\begin{aligned} &\mathrm{D}_\lambda L = 0 \Leftrightarrow u = G(u, x) && \text{(state equation)} \\ &\mathrm{D}_u L = 0 \Leftrightarrow \lambda = \mathrm{D}_u N(u, \lambda, x)^T && \text{(adjoint equation)} \\ &\mathrm{D}_x L = 0 \Leftrightarrow 0 = \mathrm{D}_x N(u, \lambda, x)^T && \text{(design equation)} \end{aligned} \tag{4.4}$$

The convergence properties of the One Shot iteration (4.1) were investigated at length in the work of Hamdi and Griewank [51, 52]. Obviously, they depend heavily on the choice of a suitable preconditioner P. To understand what a good preconditioner has to offer, it is helpful to give an outlook on the actual convergence analysis. The basic idea is to show that solving Equation (4.1) is equivalent to minimizing a *doubly augmented Lagrangian function*

$$L^{\mathrm{Aug}}(u, \lambda, x) = L(u, \lambda, x) + \frac{\alpha}{2} \|G(u, x) - u\|^2 + \frac{\beta}{2} \|\mathrm{D}_u N(u, \lambda, x)^T - \lambda\|^2. \tag{4.5}$$

The gradient of L^{Aug} might be written as

$$\begin{bmatrix} \mathrm{D}_u L^{\mathrm{Aug}} \\ \mathrm{D}_\lambda L^{\mathrm{Aug}} \\ \mathrm{D}_x L^{\mathrm{Aug}} \end{bmatrix} = -M s(u, \lambda, x), \tag{4.6}$$

with

$$M = \begin{bmatrix} \alpha\,(\mathrm{I} - \mathrm{D}_u G)^T & -\mathrm{I} - \beta \mathrm{D}_{uu} N & 0 \\ -\mathrm{I} & \beta\,(\mathrm{I} - \mathrm{D}_u G) & 0 \\ -\alpha \mathrm{D}_x G^T & -\beta \mathrm{D}_{ux} N^T & P \end{bmatrix}, \tag{4.7}$$

and

$$s(u, \lambda, x) = \begin{bmatrix} G(u, x) - u \\ \mathrm{D}_u N(u, \lambda, x)^T - \lambda \\ -P_i^{-1} \mathrm{D}_x N(u, \lambda, x)^T \end{bmatrix}. \tag{4.8}$$

Here, the vector s is the increment vector in each One Shot iteration step. In their analysis, Hamdi and Griewank derived conditions on α, β, such that the doubly augmented Lagrangian L^{Aug} is an exact penalty function. In [52], they propose the following preconditioner,

$$P = \frac{1}{\sigma}\left(\alpha (\mathrm{D}_x G)^T \mathrm{D}_x G + \beta (\mathrm{D}_{ux} N)^T \mathrm{D}_{ux} N + \mathrm{D}_{xx} N\right), \tag{4.9}$$

where ρ is the spectral radius of G and

$$\sigma = 1 - \rho - \frac{\left(1 + \frac{\beta}{2}\|\mathrm{D}_{uu} N\|\right)^2}{\alpha\beta(1-\rho)}. \tag{4.10}$$

If the Hessian of the shifted Lagrangian $\mathrm{D}_{xx} N$ is positive semi-definite this is a valid choice.

Remark 4.1.1. *This preconditioner is strongly linked to the Hessian of the augmented Lagrangian, since* $P \approx \mathrm{D}_{xx} L^{\mathrm{Aug}}$*, and strict equality holds if flow and adjoint feasibility are fulfilled.*

Sometimes it can be beneficial to take a step back and look at results from different angles. The theoretical descent conditions from the literature are great from a mathematical viewpoint, as they give precise requirements for a good preconditioner. However, the individual terms involved are highly impractical to evaluate. Second order derivatives of the shifted Lagrangian, like $\mathrm{D}_{uu} N$, $\mathrm{D}_{ux} N$, or $\mathrm{D}_{xx} N$, are too expensive to compute in most cases. Therefore, many implementations end up using BFGS approximations of $\mathrm{D}_{xx} L^{\mathrm{Aug}}$, or heuristics for α, β to calculate the preconditioner.
This thesis takes a different route and in Chapter 5 a new approximation of the Hessian, via Sobolev smoothing, is derived from well-established shape optimization theory. The constructed matrix is positive definite by design and has strong smoothing properties for fluctuations in the sensitivities. This is backed up by strong numerical results for the effectiveness of this preconditioner in practical One Shot applications, presented in Chapter 7.

4.2 One Shot Algorithms

Having discussed the One Shot iteration in Equation (4.1), the next task is to translate this into an algorithm for practical application. Direct integration into an iterative procedure yields the most basic version.

Algorithm 4.2.1 Jacobi One Shot

```
input Initial values $x_0$
for $i = 0, 1, \dots, I-1$ do
    $u_{i+1} = G(u_i, x_i)$
    $\lambda_{i+1} = \mathrm{D}_u G(u_i, x_i)^T \lambda_i + \mathrm{D}_u F(u_i, x_i)^T$
    $x_{i+1} = x_i - P_i^{-1}\left(\mathrm{D}_x G(u_i, x_i)^T \lambda_i + \mathrm{D}_x F(u_i, x_i)^T\right)$
return $x_I$
```

This original One Shot algorithm can be significantly extended and optimized to increase performance.

1. As mentioned in the introduction of this chapter, the One Shot method aims at achieving an optimal trade-off between the accuracy of the flow and adjoint solution and the design updates. In many cases, it is beneficial to do more than one step for the flow and adjoint iterations before updating the mesh. This is especially true in industrial applications, where external parameterizations, e.g., CAD tools and mesh creation software, are involved in the mesh deformation, making the computation costly. Such algorithms are known as multistep One Shot algorithms and have been studied in the past, e.g., Özkaya [92].
   Depending on the number of flow and adjoint iterations used in each step, they act on a scale from pure One Shot as in Algorithm 4.2.1 to classic, fully converged optimization.

2. Algorithm 4.2.1 is a natural extension of the Jacobi type version of the piggyback algorithm and it is also possible to implement this as a Seidel type, by always using the latest information available. While this might seem advantageous at first, the Jacobi type can offer better potential for parallelization. That is, one might run the three iterators simultaneously on different machines. In fact, it has been suggested in experimental research to run the three separate solvers asynchronous and only exchange the variables u, λ, x via interfaces during the iteration at specified points [21].
   For this thesis, the implementation is set up to use a coupled Jacobi type piggyback driver, where the resulting u_{i+1} and λ_{i+1} are then used to compute an update in the design. A detailed explanation of the implementation is given in Subsection 6.2.2.

3. Last, the mesh parameterization has to be incorporated when updating the design. In Subsection 3.2, the importance of projecting the mesh sensitivities back onto the design variables was established. Considering this, the algorithm must be extended by projecting gradients onto design parameters, updating them, and then deforming the mesh consistently.

A parameterized multistep One Shot algorithm is formulated by bringing all these points together. This is achieved in Algorithm 4.2.2, which serves as the backbone for One Shot optimization when deriving a constrained formulation in Section 4.3.

Algorithm 4.2.2 Multistep One Shot with mesh update

```
input Initial values $p_0$
for $i = 0, 1, \dots, I-1$ do
    $x_i = M(p_i)$
    for $j = 0, 1, \dots, J-1$ do
        $u_{j+1} = G(u_j, x_i)$
        $\lambda_{j+1} = \mathrm{D}_u G(u_j, x_i)^T \lambda_j + \mathrm{D}_u F(u_j, x_i)^T$
    $\delta p = \mathrm{D}_p M(p_i)^T \left(\mathrm{D}_x G(u_J, x_i)^T \lambda_J + \mathrm{D}_x F(u_J, x_i)^T\right)$
    $p_{i+1} = p_i - B_i^{-1} \delta p$
return $p_I$
```

As seen previously, the convergence of One Shot algorithms depends on the correct choice of the preconditioners B_i. Note that this is now a matrix with dimensions $n_p \times n_p$, so it must act in the parameter space. In Chapter 5, a new combination of shape derivative smoothing techniques and parameterization is introduced, enabling an efficient construction of B_i for every design update and allowing Algorithm 4.2.2 to run on a smooth trajectory towards feasibility and optimality simultaneously.

4.3 Constrained One Shot Optimization

One important aspect missing in the formulation of the One Shot algorithm so far is the incorporation of additional constraints for the optimization. In aerodynamic shape optimization, most relevant test cases require additional restrictions during the design process. These can range in complexity starting with simple direct limits on the design parameters, e.g., with $b_l, b_u \in \mathbb{R}^{n_p}$,

$$b_l \leq p \leq b_u. \tag{4.11}$$

Such simple limits and other more complex geometric constraints to the design and the mesh, e.g., minimal thickness, limited length, etc., can be expressed in terms of algebraic functions depending only on the geometric coordinates

$$C_{geo}(x, p) \geq 0. \tag{4.12}$$

Finally, there might be restrictions on the aerodynamic properties, depending on the calculated flow field. In this case, evaluating them has similar costs to an evaluation of the objective function, for example keeping a constant lift or limiting the pitching moment of a wing

$$C_{aero}(u,x) \geq 0. \tag{4.13}$$

The treatment of constraints for One Shot optimization has been studied from different aspects [72, 22, 54]. Existing approaches focus mainly on equality constraints and integrating them into the formulation of the algorithm. Although the proposed methods could potentially be generalized, currently no complete convergence analysis for inequality constraints in this setting has been presented.

Throughout the literature, there are two main ways for constraint treatment. In this work, they are called the direct and indirect approaches, respectively. The direct treatment uses the constraint values and gradients for a linear approximation of the constraints in a quadratic subproblem when computing a design update [54, 109]. It coincides with the reduced SQP optimizer introduced in Section 3.5, if the inner piggyback iteration from the multistep One Shot method is converged in the flow and adjoint solutions. That means, if the iteration count J in Algorithm 4.2.2 is chosen large enough, the constrained multistep One Shot algorithm derived in the following becomes a reduced SQP optimization again.

Recall Definition 3.5.1 from Section 3.5:

$$\begin{aligned} \min_{u,x,p} \; & F(u,x) && \text{(objective function)} \\ s.t. \;\; & M(p) = x && \text{(mesh equation)} \\ & G(u,x) = u && \text{(flow equation)} \\ & E(u,x) = 0 && \text{(equality constraint)} \\ & C(u,x) \geq 0 && \text{(inequality constraint)} \end{aligned} \tag{4.14}$$

For each component of the constraints E_k and C_l, additional adjoint problems are set up by defining a Lagrangian and shifted Lagrangian to compute a reduced gradient. For example,

$$\begin{aligned} L^{E_k}(u,\lambda^{E_k},x) &= E_k(u,x) + (\lambda^{E_k})^T(G(u,x)-u) \\ N^{E_k}(u,\lambda^{E_k},x) &= E_k(u,x) + (\lambda^{E_k})^T G(u,x) \\ \tilde{D}_p E_k &= \mathrm{D}_x N^{E_k}(u,\lambda^{E_k},x)\mathrm{D}_p M(p). \end{aligned} \tag{4.15}$$

Using a reduced SQP algorithm for the optimization problem (4.14) means that each design update step solves a quadratic approximation of the real problem.

$$\begin{aligned} &\min_{v\in S_v} \frac{1}{2}v^T B v + \tilde{D}_p F v \\ &s.t.\ \ \tilde{D}_p E v + E = 0 \\ &\qquad \tilde{D}_p C v + C \geq 0 \end{aligned} \tag{4.16}$$

Integrating the solution of this quadratic problem as the design update into the One Shot algorithm results in the complete method used in this thesis, see Algorithm 4.3.1.

Algorithm 4.3.1 One Shot with direct treatment of constraints

```
input Initial values $p_0$
for $i = 0,1,...,I-1$ do
    $x_i = M(p_i)$                                             ▷ compute a deformed mesh
    for $j = 0,1,...,J-1$ do                                   ▷ iterate multiple piggyback steps
        $u_{j+1} = G(u_j,x_i)$
        $\lambda_{j+1} = \mathrm{D}_u N(u_j,\lambda_j,x_i)^T$
        for $k = 0,1,...,n_E$ do
            $\lambda_{j+1}^{E_k} = \mathrm{D}_u N^{E_k}(u_j,\lambda_j^{E_k},x_i)^T$
        for $l = 0,1,...,n_C$ do
            $\lambda_{j+1}^{C_l} = \mathrm{D}_u N^{C_l}(u_j,\lambda_j^{C_l},x_i)^T$
    $\tilde{D}_p F = \mathrm{D}_x N(u_J,\lambda_J,x_i)\mathrm{D}_p M(p_i)$         ▷ evaluate the design equations
    for $k = 0,1,...,n_E$ do
        $\tilde{D}_p E_k = \mathrm{D}_x N^{E_k}(u_J,\lambda_J^{E_k},x_i)\mathrm{D}_p M(p_i)$
    for $l = 0,1,...,n_C$ do
        $\tilde{D}_p C_l = \mathrm{D}_x N^{C_l}(u_J,\lambda_J^{C_l},x_i)\mathrm{D}_p M(p_i)$
    Compute $B_i$                                              ▷ calculate the preconditioner
    solve the quadratic problem: $\min_{v\in S_v} \frac{1}{2}v^T B_i v + \tilde{D}_p F v$     ▷ compute design update
                                 $s.t.\ \tilde{D}_p E v + E = 0$
                                       $\tilde{D}_p C v + C \geq 0$
    $p_{i+1} = p_i + v$                                        ▷ update the design
return $p_I$
```

Observe, how the direct approach computes multiple adjoint states λ, λ^{E_k}, and λ^{C_l} by separate adjoint computations. This allows the algorithm to use existing adjoint solvers out of the box, thus making the implementation relatively easy and allowing for parallel evaluation of the different adjoints. Nevertheless, it is important to keep in mind that this increases the computational cost significantly. In a worst-case scenario, the optimization has to run $n_E + n_C + 1$ different adjoint solvers, each with their own cost and overheads for initialization, file IO, etc. However, this can be alleviated if the constraints do not depend on the flow state. From the different types of inequality constraints shown at the beginning of this section, only aerodynamic constraints, as in Equation (4.13), need their own adjoint state. For box and geometric constraints, as in Equations (4.11) and (4.12), the gradient can be computed directly.

The alternative is to use an indirect treatment of the constraints, as analyzed by Kusch, Walther, et al. [72]. Here, the equality constraints are integrated by extending the existing Lagrangian function. Note that this is the same extended Lagrangian as in the original introduction of SQP methods in Equation (3.52).

$$\begin{aligned} L^{\text{Ext}}(u,\lambda,x) &= F(u,x) + \theta^T E(u,x) + \lambda^T (G(u,x) - u) \\ N^{\text{Ext}}(u,\lambda,x) &= F(u,x) + \theta^T E(u,x) + \lambda^T G(u,x) \end{aligned} \tag{4.17}$$

It is then possible to use Algorithm 4.2.2 directly, where the derivatives of the shifted Lagrangian N are replaced by the derivatives of its extended counterpart N^{Ext}.

Algorithm 4.3.2 One Shot with indirect treatment of constraints

```
input Initial values $p_0$
for $i = 0, 1, ..., I - 1$ do
    $x_i = M(p_i)$
    for $j = 0, 1, ..., J - 1$ do
        $u_{j+1} = G(u_j, x_i)$
        $\lambda_{j+1} = \mathrm{D}_u N^{\text{Ext}}(u_j, \lambda_j, x_i)^T$
    $\tilde{D}_p N^{\text{Ext}} = \mathrm{D}_x N^{\text{Ext}}(u_J, \lambda_J, x_i) \mathrm{D}_p M(p_i)$
    compute preconditioners $B_i, \widehat{B}_i$
    $p_{i+1} = p_i - B_i^{-1} (\tilde{D}_p N^{\text{Ext}})^T$
    $\theta_{i+1} = \theta_i - \widehat{B}_i^{-1} E(u_J, x_J)$
return $p_I$
```

This keeps the cost from the incorporation of the additional constraints low, but may require significant changes to the discrete adjoint implementation to be able to evaluate and differentiate the expression $F(u,x) + \theta^T E(u,x)$ and its associated adjoint. Furthermore, $\widehat{B}_i^{-1}$ has to be chosen carefully and the conditions on the choice of α and β are stricter.

Chapter 5

Shape Hessian approximation, Sobolev Smoothing, and Parameterization

Sobolev gradient smoothing, interpreted as a shape Hessian approximation, can be combined with a parameterization to offer significant benefits for design optimization. To demonstrate this central result of the presented thesis, all of these parts are brought together here, demonstrating how they can be combined successfully.

Sobolev gradient smoothing is based on deriving connections between gradients with respect to different function spaces. Central to this approach is the reinterpretation of the gradient in different scalar products. By choosing the right Hilbert space, and therefore the right scalar product, the regularity of the problem can be significantly increased, naturally leading to faster convergence rates for derivative-based optimization schemes.

In optimization, the use of second order derivative information can lead to crucial increases in speed. To this end, many different approximations of Hessian matrices have been proposed to formulate approximated Newton methods, including the transfer of theoretical results on the shape Hessian to discrete adjoint optimization.

Approximate Newton methods need to take the crucial role of the design parameterization into account. That means any form of higher order derivative information must not only consider derivatives with respect to surface nodes, but with respect to design parameters. Sobolev smoothing can be applied here to offer benefits, even if the parameterization used is smooth already.

This chapter creates a synthesis of these different approaches. It starts with a motivational background on the approximation of shape Hessian operators and their connection to the Laplace-Beltrami operator in Section 5.1. Next, it recapitulates the theory of Sobolev smoothing for a free node formulation in Section 5.2. All of this leads up to the central result of this work, a new methodology for Sobolev smoothing in the context of parameterization. To achieve this, the equation connecting the reduced shape Hessian and the second order derivatives with respect to the parameters is derived, and the Laplace-Beltrami operator from Sobolev smoothing is inserted into this equation in Section 5.3. At last, the incorporation of the new method into the optimization algorithms presented before is discussed in Section 5.4, including One Shot algorithms for simultaneous analysis and design.

5.1 Reduced Shape Hessian approximation

In this section, the background on shape Hessian approximation is further discussed by surveying the relevant literature. The aim is to motivate the approximation of the reduced shape Hessian by the Laplace-Beltrami operator. This includes results on the continuous shape Hessian operator, see Definition 2.3.16, and on the discrete reduced shape Hessian, see Equation (3.34).

Before the different results are listed, the shape calculus established in Subsection 2.3.2 is used to calculate the Hessian operator for an example. Consider the iso-perimeter problem, also called 'Dido's problem', where the aim is to minimize the surface of an object Ω, while keeping a constant volume c_{vol}.

$$\begin{aligned} \min_{\Omega} \mathscr{F}(\Omega) &= \int_{\Gamma} 1 \, \mathrm{d}s \\ s.t. \int_{\Omega} 1 \, \mathrm{d}x &= c_{\text{vol}} \end{aligned} \tag{5.1}$$

Given a Lagrangian function for this problem

$$L(\Omega, \lambda) = \int_{\Gamma} 1 \, \mathrm{d}s + \lambda \left(\int_{\Omega} 1 \, \mathrm{d}x - c_{\text{vol}} \right), \tag{5.2}$$

it is possible to calculate shape derivatives. A detailed analysis and discussion of the problem and the associated calculations are given in the paper by Schmidt [105]. Applying Definitions 2.3.13 and 2.3.16 and simplifying the resulting terms yields the following expressions. For the shape derivative it holds that

$$\mathscr{D}L(\Omega, \lambda; v) = \int_{\Gamma} \langle v, n \rangle (\kappa + \lambda) \, \mathrm{d}s. \tag{5.3}$$

Here, n is the surface normal and κ is the curvature of the surface Γ. Calculating the shape Hessian results in

$$\mathscr{D}^2 L(\Omega, \lambda; v_1, v_2) = \int_{\Gamma} \langle v_1, n \rangle \langle v_2, n \rangle (\lambda \kappa + \kappa^2) + \langle \nabla_{\Gamma} \langle v_1, n \rangle, \nabla_{\Gamma} \langle v_2, n \rangle \rangle \, \mathrm{d}s. \tag{5.4}$$

Here, ∇_{Γ} denotes the tangential gradient on the surface. Looking at the Hessian expression, it can be identified as the weak formulation of a PDE. Assume that v_* is unknown, then

$$\begin{aligned} \forall v_2 \in C_0^{\infty}(\mathscr{M}, \mathbb{R}^{n_d}) : \int_{\Gamma} \langle v_*, n \rangle \langle v_2, n \rangle (\lambda \kappa + \kappa^2) + \langle \nabla_{\Gamma} \langle v_*, n \rangle, \nabla_{\Gamma} \langle v_2, n \rangle \rangle \, \mathrm{d}s = 0 \\ \Leftrightarrow \left((\lambda \kappa + \kappa^2) \mathrm{I} + \Delta_{\Gamma} \right) (v_*)_n = 0, \end{aligned} \tag{5.5}$$

where $(v_*)_n$ denotes the part of v_* in direction of n. It is worth keeping this PDE in mind, as the partial differential operator, known as the *Laplace-Beltrami* operator, reappears throughout this thesis.

For the connection to aerodynamic design optimization problems, one might expect a connection to the drag reduction problem in aerodynamics, as drag reduction, seen from a trivial viewpoint, usually corresponds with keeping the surface smooth and the front cross section small.

In general, the nature of the Navier-Stokes equations makes the explicit formulation of the shape Hessian for arbitrary flow equations and objective functions almost impossible. However, finding good approximations for relevant special cases is an open research topic and extensive work in this

field has been done in the past. In the scope of this thesis, some of the results are worth showing in more detail, so this section aims to give an overview of the relevant literature.
Inspiration for this thesis was drawn from the work by Kusch, Schmidt, and Gauger [70, 71] on the drag minimization problem for Stokes equations. Here, the key points of the underlying analysis are highlighted to clarify how they apply in the current context. The core idea is to make use of the symbol of the Hessian matrix. The symbol of an operator can be seen as the response of that operator applied to a wave function.

Definition 5.1.1 (operator symbol)**.** *Let $O : \mathrm{S} \to \mathbb{R}$ be a scalar-valued operator on a suitable space of periodic functions S and $g = e^{-i\omega x}$ be a wave function with*

$$O(g) = \sigma_O e^{-i\omega x}, \tag{5.6}$$

then σ_O is called the symbol of operator O.

This concept is based on Fourier analysis, where it is widely used to calculate the effect of an operator on the basis functions of a Fourier transformation, e.g., wave functions. Application to aerodynamic shape optimization problems can be credited to Arian and Ta'asan [10]. In the context of shape optimization, a small, smooth perturbation of a shape at one point is equivalent to adding a wave function there. For comparison, see the similarity to the differentiable vector field in the perturbation of identity 2.3.11, where a wave function could be inserted.
For the application of a Quasi-Newton step in the optimization algorithm, the symbol of the inverse Hessian $\sigma_{H^{-1}}$ is mimicked. In [71], a drag minimization problem with Stokes equations for the flow was investigated and it was shown that

$$\sigma_H = \beta_1 + \beta_2 \omega, \tag{5.7}$$

for suitably chosen parameters β_1, β_2, depending on the analytic expression for the drag and the current shape. Thus, any choice of a preconditioner $\mathscr{B}$ for a Quasi-Newton optimization method should mimic the behavior of the inverse Hessian symbol

$$\sigma_{B^{-1}} \approx \sigma_{H^{-1}} = \frac{1}{\beta_1 + \beta_2 \omega}. \tag{5.8}$$

To obtain this kind of behavior with a simple and computationally cheap preconditioner, $\mathscr{B}$ is chosen to be a modification of the Laplace-Beltrami operator

$$\mathscr{B} = \varepsilon_1 \mathrm{I} + \varepsilon_2 \Delta_\Gamma. \tag{5.9}$$

For more details on the background of this particular choice of operator, see Section 5.2. The inverse of the Laplace-Beltrami operator has the symbol

$$\sigma_{B^{-1}} = \frac{1}{\varepsilon_1 + \varepsilon_2 \omega^2}. \tag{5.10}$$

While the two symbols are different, the factors β_1, β_2 and $\varepsilon_1, \varepsilon_2$ can be chosen to make up for the discrepancy, allowing for Hessian like behavior when applied to the gradient for an approximated Newton step. It is important to note that this choice has some interesting properties derived from

Fourier analysis. Using only even-ordered derivatives can prevent a phase shift in the wave function and for the correct scaling, it is important to choose the parameters $\varepsilon_1, \varepsilon_2$ accordingly. Since the parameters β_1, β_2 were calculated from the analytic expression for the drag in a laminar Stokes flow, this is a non-trivial task in practice. For the presented thesis, the values $\varepsilon_1, \varepsilon_2$ in the optimization are instead obtained from a parameter study. Nonetheless, there are theoretical results available for choosing local approximations of β_1, β_2 and while our solver uses global values, it seems possible to extend the implementation for the use of local values for $\varepsilon_1, \varepsilon_2$ on each mesh cell. See the discussion in Chapter 8 for possible extensions of the presented results in this direction.

The ideas above are based on an older result worth highlighting in this context. Arian and Ta'asan [10] have done extensive work with potential flows and inviscid Euler flows. Their research gives a detailed theoretical insight into the problem. The authors follow the approach of investigating a 'small disturbance problem', meaning they add a small disturbance to an optimal design Γ^* and flow state W^*.

$$\begin{aligned} \Gamma &= \Gamma^* + \delta \alpha n \\ W &= W^* + \delta \tilde{U} + O(\delta^2) \end{aligned} \tag{5.11}$$

Here, n is the outer surface normal vector and δ is a small positive number. Similar to the Fourier idea presented before, this change can be seen as adding a wave function and can be analyzed in local coordinates around the perturbation. If $\alpha > 0$ is a design parameter defining the wave's amplitude, then this defines a half-space of deformations and allows for a minimization problem with Taylor expansions for flow, adjoint, and design equations. Solving those leads to equations for the Hessian with respect to the control α, from which the operator symbol can be determined.

Using the same notation as above, the operator symbol of the Hessian for Euler equations is derived as

$$\sigma_H = c_1 \frac{k_1^4}{k_1^2(1-c_2)+k_2^2}, \tag{5.12}$$

where k_1, k_2 are the wave numbers from the basis functions of the Fourier analysis and c_1, c_2 are constants. When working in local coordinates on the surface there are two dimensions marked by the subscripts, one in flow direction and one orthogonal to it. Going back to the previous notation this implies that $k_1 = \frac{\omega_1}{h_1}$ is a local chord-wise perturbation and $k_2 = \frac{\omega_2}{h_2}$ is a local span-wise perturbation, with mesh cell sizes h_1, h_2 in the respective directions. In case of a two-dimensional airfoil there is no span-wise perturbation, leading to

$$\sigma_H = c_1 \frac{k_1^2}{(1-c_2)} \tag{5.13}$$

and for the inverse

$$\sigma_{H^{-1}} = c_1 \frac{(1-c_2)}{k_1^2}. \tag{5.14}$$

This means that the operator symbol smooths in chord direction, similar to the Laplace-Beltrami operator shown above. Using the results from Arian and Ta'asan [10], Arian and Vatsa developed a method for shape optimization with Euler equations [11]. In their algorithm, they propose using an operator of the form $c_3 \mathrm{I} + c_4 \Delta_\Gamma$ on the surface to smooth the continuous shape derivative formulation and then remesh the geometry after the design update.

Finally, a more recent result by Müller, Kühl, et al. [85] deserves mentioning. In this work, the authors apply a p-Laplace problem as a relaxation step in the steepest descent algorithm. The focus is on free node optimization, where the method shows promising results in terms of preserving edges in the design and maintaining a good mesh quality throughout the optimization. These results are also connected to another recent study by Deckelnick, Herbert, et al. [32] investigating properties of Sobolev gradient descent in a $\mathrm{W}^{1,\infty}$ topology for shape optimization.

The whole process of calculating shape Hessian operator symbols involves complex calculations and is so far only understood for some special cases. While it has great potential in giving second order information for continuous or free node formulations, there is still no general solution. For example, the exact Hessian symbol for drag minimization with Navier-Stokes or RANS equations is still an open research topic.

All of these works share the use of Laplacian smoothing, in one form or another, hinting at a deeper connection between the Hessian of the drag minimization problem for different fluid equations and the Laplace-Beltrami operator. Therefore, an approximation of the discretized, reduced shape Hessian, in Equation (3.34), should have the same properties and behavior. This idea is used in this thesis when Sobolev smoothing is combined with the parameterization in Section 5.3.

5.2 Function Spaces and Sobolev Smoothing

This section introduces the core idea of the Sobolev gradient smoothing method. This approach has been used in different PDE constraints optimization settings in the past and a good overview of the general idea, as well as the theoretical background, can be found in the books of Neuberger [88], or Faragó and Karátson [39]. An introduction to Sobolev spaces can be found in many sources, the standard one being the textbook by Adams and Fournier [2]. This only deals with Sobolev spaces of scalar-valued functions and for vector-valued functions the results presented below are based on the book by Hytönen, van Neerven, et al. [59, Chapter 2].

In Chapter 3, a large emphasis was given to the computation of gradients for shape optimization problems using the adjoint method. From functional analysis, it is a well-known that the gradient of a function dependents on the underlying Hilbert space and its scalar product. The derivatives discussed in this thesis until now are directional derivatives and in a finite-dimensional real-valued vector space, the gradients are expressed via the canonical Euclidean scalar product. Sobolev smoothing aims to increase the regularity of gradient-based descent steps, in an optimization, by using the Sobolev space H^1-gradient instead. Therefore, the mentioned relevant Hilbert spaces and their respective scalar products are presented. With this, the general idea of Sobolev smoothing and the reinterpretation of derivatives in the sense of an H^1-gradient are demonstrated.

For the rest of this section, assume that $\zeta \subset \mathbb{R}^{n_\zeta}$ is a compact subset of a finite-dimensional real-valued vector space. Here, the theory is introduced for vector-valued functions $f : \zeta \to \mathbb{R}^d$. For the application of this thesis, the dimension d will be 2 or 3, depending on the smoothing being performed on a surface or in a volume.

Definition 5.2.1 (L^p spaces)**.** *Let f be a measurable function on ζ, then the class of all functions for which*

$$\left(\int_\zeta \|f(x)\|_p^p \,\mathrm{d}x \right)^p < \infty \tag{5.15}$$

is called the $\mathrm{L}^p(\zeta, \mathbb{R}^d)$ space. Here, $\|\bullet\|_p$ denotes the p-norm on $\mathbb{R}^d$.

Corollary 5.2.2. *The functional*

$$\| \bullet \|_{\mathrm{L}^p} = \left(\int_\zeta \|f(x)\|_p^p \,\mathrm{d}x \right)^{\frac{1}{p}} \tag{5.16}$$

defines a norm on $\mathrm{L}^p(\zeta, \mathbb{R}^d)$*, such that the space is a Banach space.*

Proof. The proof for $\mathrm{L}^p(\zeta, \mathbb{R})$ is given in Adams and Fournier [2, Theorem 2.16]. It carries over verbatim to the vector-valued case, when the absolute value $|f(x)|$ is replaced by the respective p-norm $\|f(x)\|_p$. □

The L^p spaces from Definition 5.2.1 have nice mathematical properties, with the most relevant of them being the space of square-integrable functions $\mathrm{L}^2(\zeta, \mathbb{R}^d)$.

Definition 5.2.3 (L^2 scalar product)**.** *Let* $\mathrm{L}^2(\zeta, \mathbb{R}^d)$ *be the space of square-integrable functions by Definition 5.2.1, then the mapping*

$$\langle \bullet, \bullet \rangle_{\mathrm{L}^2} : \mathrm{L}^2(\zeta, \mathbb{R}^d) \times \mathrm{L}^2(\zeta, \mathbb{R}^d) \to \mathbb{R}; \langle f, g \rangle_{\mathrm{L}^2} \mapsto \int_\zeta \langle f(x), g(x) \rangle_2 \,\mathrm{d}x, \tag{5.17}$$

where $\langle \bullet, \bullet \rangle_2$ *denotes the standard scalar product on* $\mathbb{R}^d$*, defines a scalar product on* $\mathrm{L}^2(\zeta, \mathbb{R}^d)$*.*

Corollary 5.2.4. *The space* $\mathrm{L}^2(\zeta, \mathbb{R}^d)$ *of square-integrable functions, together with the norm introduced by the scalar product from Definition 5.2.3, is a Hilbert space.*

Proof. The proof of this statement follows directly from Corollary 5.2.2 and Adams and Fournier [2, Corollary 2.18]. □

Next, to introduce weak differentiability for a vector-valued function define a multiindex $\alpha \in \mathbb{N}^{n_\zeta}$. Then the derivative w.r.t. a multiindex can be defined by componentwise partial differentiation

$$\mathrm{D}_\alpha = \frac{\partial^{|\alpha|}}{\partial x_1^{\alpha_1} \dots \partial x_{n_\zeta}^{\alpha_n}}. \tag{5.18}$$

For vector-valued functions, this is applied componentwise in each dimension.

Example 5.2.5. *Let* $u : \mathbb{R}^2 \mapsto \mathbb{R}^2; \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto \begin{pmatrix} x_1 \sin(x_2) \\ x_2^2 \end{pmatrix}$ *be a function and* $(0,1)$ *be a multiindex, then*

$$\mathrm{D}_{(0,1)} u(x) = \frac{\partial}{\partial x_2} \begin{pmatrix} u_1(x) \\ u_2(x) \end{pmatrix} = \begin{pmatrix} x_1 \cos(x_2) \\ 2x_2 \end{pmatrix}. \tag{5.19}$$

This means that for $f \in C^1(\zeta, \mathbb{R}^d)$ a first order componentwise partial derivative $\mathrm{D}_\alpha f \in C^0(\zeta, \mathbb{R}^d)$ is a column of the Jacobian matrix.

Definition 5.2.6 (weak differentiability)**.** *Let* $f, g \in \mathrm{L}^1(\zeta, \mathbb{R}^d)$ *be integrable functions. If*

$$\forall \phi \in C_0^\infty(\zeta, \mathbb{R}) : \int_\zeta g(x) \phi(x) \,\mathrm{d}x = (-1)^{|\alpha|} \int_\zeta f(x) \mathrm{D}_\alpha \phi(x) \,\mathrm{d}x, \tag{5.20}$$

then g *is called the weak derivative of* f *of order* α*, denoted by* $d_\alpha^w f$*.*

The definition above is consistent with the definition of a distributional derivative, see [59, Definition 2.5.1]. Also, if the function f has a classical derivative $\mathrm{D}_\alpha f$, then this naturally fulfills Definition 5.2.6 by partial integration.

Definition 5.2.7 (Sobolev spaces)**.** *The Sobolev space of weak differentiable functions is defined by*

$$\mathrm{W}^{1,p}(\zeta,\mathbb{R}^d) := \left\{ f \in \mathrm{L}^p(\zeta,\mathbb{R}^d) \;\middle|\; \forall 0 \leq |\alpha| \leq 1 : d^w_\alpha f \in \mathrm{L}^p(\zeta,\mathbb{R}^d) \right\}. \tag{5.21}$$

Definition 5.2.8 (Sobolev norm)**.** *For a weak differentiable function f, the Sobolev norm*

$$\|f\|_{\mathrm{W}^{1,p}} = \left(\sum_{0\leq|\alpha|\leq 1} \|d^w_\alpha f\|^p_{\mathrm{L}^p} \right)^{\frac{1}{p}}, \tag{5.22}$$

can be defined using the L^p-norm $\|\bullet\|_{\mathrm{L}^p}$ from 5.2.2.

Corollary 5.2.9. *The space $\mathrm{W}^{1,p}(\zeta,\mathbb{R}^d)$ together with the norm $\|\bullet\|_{\mathrm{W}^{1,p}}$ forms a Banach space.*

Proof. For a proof of this statement, see the book by Hytönen, van Neerven, et al. [59, Section 2.5]. ☐

While Sobolev spaces have many additional, interesting mathematical properties, for the scope of this work one particular Sobolev space is relevant. It can be once again defined by a scalar product. To simplify the following notation, let α_i be a multiindex of order $|\alpha_i| = 1$ with the i-th component being 1, then d^w_i denotes the weak derivative with respect to the i-th variable, i.e., for differentiable functions $d^w_i f = d^w_{\alpha_i} f = \frac{\partial}{\partial x_i} f$.

Definition 5.2.10 (H^1 space and scalar product)**.** *The mapping*

$$\begin{aligned} \langle \bullet,\bullet \rangle_{\mathrm{H}^1} &: \mathrm{W}^{1,2}(\zeta,\mathbb{R}^d) \times \mathrm{W}^{1,2}(\zeta,\mathbb{R}^d) \to \mathbb{R}; \\ \langle f,g \rangle_{\mathrm{H}^1} &\mapsto \int_\zeta \langle f(x),g(x)\rangle_2 \,\mathrm{d}x + \int_\zeta \sum_{i=1}^{n_\zeta} \langle d^w_i f(x), d^w_i g(x)\rangle_2 \,\mathrm{d}x \end{aligned} \tag{5.23}$$

defines a scalar product on $\mathrm{W}^{1,2}(\zeta,\mathbb{R}^d)$, where $\langle\bullet,\bullet\rangle_2$ denotes the Euclidean scalar product on $\mathbb{R}^d$. The $\mathrm{H}^1(\zeta,\mathbb{R}^d)$ Sobolev space is defined via

$$\mathrm{H}^1(\zeta,\mathbb{R}^d) := \left\{ f \in \mathrm{W}^{1,2}(\zeta,\mathbb{R}^d) \;\middle|\; \langle f,f\rangle_{\mathrm{H}^1} < \infty \right\}. \tag{5.24}$$

Corollary 5.2.11. *$\mathrm{H}^1(\zeta,\mathbb{R}^d)$ is a Hilbert space with respect to the scalar product $\langle\bullet,\bullet\rangle_{\mathrm{H}^1}$.*

Proof. See Hytönen, van Neerven, et al. [59, Section 2.5]. ☐

Next, the connection between the choice of the underlying Hilbert space and scalar product and the gradient of a function is discussed. The well-known Riesz representation theorem shows how the directional derivative and the gradient of a function are connected.

Theorem 5.2.12 (Riesz representation theorem)**.** *Let $L : V \to \mathbb{R}$ be a linear functional on a Hilbert space V, then there exists a unique representative $z \in V$ such that*

$$\forall v \in V : \; L(v) = \langle v,z\rangle_V. \tag{5.25}$$

Proof. This well-known theorem can be found in many textbooks on functional analysis, e.g., Reed and Simon [98, Theorem II.4] or Heuser [56, Theorem 26.1]. □

Example 5.2.13 (Riesz representation of the directional derivative)**.** *The directional derivative of any functional is a linear mapping of the direction and its Riesz representation is the gradient*

$$\mathscr{D}f(x;v) = \langle \nabla f, v\rangle. \tag{5.26}$$

For a scalar-valued function $f \in C^1(\mathbb{R}^n,\mathbb{R})$, i.e., in finite dimensions, this directional derivative takes the form of the Jacobian

$$\mathscr{D}f(x;v) = \mathrm{D}_x f \cdot v. \tag{5.27}$$

When using the Euclidean scalar product in $\mathbb{R}^n$, then combining the above leads to

$$\mathrm{D}_x f = (\nabla f)^T. \tag{5.28}$$

When dealing with shape optimization problems, e.g., as in Definition 2.3.18, it is possible to find a gradient representation for a given scalar product. Here, the direction takes the form of a vector field on the domain, see Definition 2.3.11. Thus, requiring a vector-valued Sobolev space $\mathrm{H}^1(\zeta,\mathbb{R}^d)$ for the gradient representation. As will be discussed in the following, this will result in componentwise Laplacian smoothing.

Corollary 5.2.14 (Sobolev smoothing)**.** *Let $\mathscr{F} : S_\zeta \to \mathbb{R}, \zeta \mapsto \mathscr{F}(\zeta)$ be a shape differentiable functional, then the Riesz representation $\nabla_{\mathrm{H}^1}\mathscr{F}$ of $\mathscr{D}\mathscr{F}(\zeta;\bullet)$, with respect to the inner product* (5.23)*, can be obtained as the solution of*

$$\forall j = \{1,\ldots,n_\zeta\}, v \in \mathrm{H}^1(\zeta,\mathbb{R}^d) : \mathscr{D}\mathscr{F}(\zeta;v_j e_j) = \int_\zeta \left((\nabla_{\mathrm{H}^1}\mathscr{F})_j v_j + \sum_{i=1}^{n_\zeta} d_i^w (\nabla_{\mathrm{H}^1}\mathscr{F})_j d_i^w v_j \right) \mathrm{d}x, \tag{5.29}$$

where v_j is the j-th component of v and e_j is the j-th unit vector.

Proof. Since $C^1(\zeta,\mathbb{R}^d)$ is dense in $\mathrm{H}^1(\zeta,\mathbb{R}^d)$, the shape derivative can be extended for $v \in \mathrm{H}^1(\zeta,\mathbb{R}^d)$. By Theorem 5.2.12, the shape derivative of $\mathscr{F}$ can then be expressed in terms of a scalar product with a unique representative denoted by $\nabla_{\mathrm{H}^1}\mathscr{F}$,

$$\mathscr{D}\mathscr{F}(\zeta;v) = \langle \nabla_{\mathrm{H}^1}\mathscr{F}, v\rangle_{\mathrm{H}^1} = \int_\zeta \langle \nabla_{\mathrm{H}^1}\mathscr{F}, v\rangle_2 \,\mathrm{d}x + \int_\zeta \sum_{i=1}^{n_\zeta} \langle d_i^w(\nabla_{\mathrm{H}^1}\mathscr{F}), d_i^w v\rangle_2 \,\mathrm{d}x. \tag{5.30}$$

The inner Euclidean scalar products can be written in terms of the components,

$$\begin{aligned}\mathscr{D}\mathscr{F}(\zeta;v) &= \int_\zeta \sum_{j=1}^{n_\zeta} (\nabla_{\mathrm{H}^1}\mathscr{F})_j v_j \,\mathrm{d}x + \int_\zeta \sum_{i=1}^{n_\zeta}\sum_{j=1}^{n_\zeta} d_i^w (\nabla_{\mathrm{H}^1}\mathscr{F})_j d_i^w v_j \,\mathrm{d}x \\ &= \sum_{j=1}^{n_\zeta} \left(\int_\zeta \left((\nabla_{\mathrm{H}^1}\mathscr{F})_j v_j + \sum_{i=1}^{n_\zeta} d_i^w (\nabla_{\mathrm{H}^1}\mathscr{F})_j d_i^w v_j \right) \mathrm{d}x \right).\end{aligned} \tag{5.31}$$

Every v can be written as a linear combination $v = \sum_{j=1}^{n_\zeta} v_j e_j$ and the directional derivative is linear. Therefore, inserting $v_j e_j$ gives

$$\mathscr{D}\mathscr{F}(\zeta;v_j e_j) = \int_\zeta \left((\nabla_{\mathrm{H}^1}\mathscr{F})_j v_j + \sum_{i=1}^{n_\zeta} d_i^w (\nabla_{\mathrm{H}^1}\mathscr{F})_j d_i^w v_j \right) \mathrm{d}x, \tag{5.32}$$

which is the weak formulation of Sobolev smoothing for the j-th component. □

Equation (5.29) is a weak formulation of a vector-valued partial differential equation. It can be discretized using finite elements on the CFD mesh. For the discretized finite elements representation, the summation over weak spatial derivatives in all variables becomes the scalar product with the spatial gradient. Therefore, it can be written more compact as

$$\mathscr{D}\mathscr{F}(\zeta;v_j e_j) = \int_\zeta \big((\nabla_{\mathrm{H}^1}\mathscr{F})_j v_j + \langle \nabla_x (\nabla_{\mathrm{H}^1}\mathscr{F})_j, \nabla_x v_j\rangle_2\big)\,\mathrm{d}x, \tag{5.33}$$

where ∇_x is the spatial gradient w.r.t. the Euclidean scalar product. Interpreted as a strong PDE this corresponds to

$$\forall j = \{1,\ldots,n_\zeta\}, v \in \mathrm{H}^1(\zeta,\mathbb{R}^d):\ (\mathrm{I}-\Delta)(\nabla_{\mathrm{H}^1}\mathscr{F})_j = \mathscr{D}\mathscr{F}(\zeta;v_j e_j), \tag{5.34}$$

with zero Neumann boundary conditions. In case the smoothing is done on the design surface, i.e., $\zeta = \Gamma$, the spatial gradients ∇_x are replaced with their tangent form $\nabla_{\partial\zeta}\mathscr{F} = \nabla_x\mathscr{F} - \langle \mathscr{F}(\zeta,n),n\rangle$. This results in the Laplace-Beltrami operator in the strong formulation,

$$\forall j = \{1,\ldots,n_\zeta\}, v \in \mathrm{H}^1(\zeta,\mathbb{R}^d):\ (\mathrm{I}-\Delta_{\partial\zeta})(\nabla_{\mathrm{H}^1}\mathscr{F})_j = \mathscr{D}\mathscr{F}(\zeta;v_j e_j), \tag{5.35}$$

Corollary 5.2.15. *Scaling the Euclidean inner products in Equation* (5.23) *by constant factors* $\varepsilon_1, \varepsilon_2 \in \mathbb{R}_{\geq 0}$ *leads to a new scalar product*

$$\langle f,g\rangle = \int_\zeta \langle f(x), \varepsilon_1 g(x)\rangle_2\,\mathrm{d}x + \int_\zeta \sum_{i=1}^{n_\zeta} \langle d_i^w f(x), \varepsilon_2 d_i^w g(x)\rangle_2\,\mathrm{d}x \tag{5.36}$$

and to a scaling of the smoothing operator $(\varepsilon_1\mathrm{I} - \varepsilon_2\Delta)$. *Thereby, scaling the Laplace-Beltrami operator can be interpreted as a change in the scalar product.*

Proof. The proof of Corollary 5.2.14 holds verbatim for the new scalar product and since everything is linear, the scaling factor can be pulled in front of the integrals. □

At this point, it is important to speak about the intended application and the dimensions involved. Let $F(u,x)$ be a discrete version of $\mathscr{F}(\zeta)$ on the CFD mesh. The design equation in the adjoint method from Chapter 3 evaluates a directional derivative of the Lagrangian, or the objective function respectively, if flow and adjoint equations are fulfilled, see Corollary 3.1.4. This discrete derivative

$$\mathrm{D}_x L(u,\lambda,x) = \mathrm{D}_x F(u,x) \tag{5.37}$$

has the form of an n_ζ-dimensional vector in each mesh cell, where the j-th entry represents the sensitivity of the objective w.r.t. movement in the j-th spatial direction. This motivates the following simplified notation.

Remark 5.2.16. *From now on*

$$(\varepsilon_1\mathrm{I} - \varepsilon_2\Delta)(\nabla_{\mathrm{H}^1} L) = \mathrm{D}_x L(u,\lambda,x)^T \tag{5.38}$$

is used as a shortened notation for Equations (5.34) *and* (5.35)*, discretized by a finite elements approach on the CFD mesh.*
The operator $(\varepsilon_1\mathrm{I} - \varepsilon_2\Delta)$ *is understood to act componentwise in each direction and* Δ *can stand for the volume or surface Laplace operator depending on the setting. In numerical computations, it will be replaced by the finite elements stiffness matrix.*

Subsection 5.1 showed connections between the Hessian matrix of shape optimization problems and the Laplace-Beltrami operator. Both Newton methods and gradient descent in the Sobolev space H^1 aim to achieve higher regularity for the search direction. The connection between the two becomes more apparent if both update formulas are compared side-by-side. Assume a Newton method is used for minimizing L. For simplicity, write $L(x) := L(u, \lambda, x)$ assuming that accurate flow and adjoint solutions are used. Then a single step of the procedure looks like

$$x_{k+1} = x_k + \delta x \quad \text{with} \quad \mathrm{D}_{xx} L(x_k) \delta x = -\mathrm{D}_x L(x_k)^T. \tag{5.39}$$

The Sobolev gradient steepest descent step gives

$$x_{k+1} = x_k + \delta x \quad \text{with} \quad \delta x = -\nabla_{\mathrm{H}^1} L(x_k). \tag{5.40}$$

Combining Equations (5.40) and (5.38) leads to

$$\begin{aligned} \delta x &= -(\varepsilon_1 \mathrm{I} - \varepsilon_2 \Delta)^{-1} \mathrm{D}_x L(x_k)^T && \text{for Sobolev gradient descent,} \\ \delta x &= -(\mathrm{D}_{xx} L(x_k))^{-1} \mathrm{D}_x L(x_k)^T && \text{for Newton updates.} \end{aligned} \tag{5.41}$$

Such a side-by-side comparison motivates the approximation of the reduced Hessian of the Lagrangian function on the mesh $\mathrm{D}_{xx} L$ with the Laplace-Beltrami operator, i.e.,

$$\mathrm{D}_{xx} L(u, \lambda, x) \approx (\varepsilon_1 \mathrm{I} - \varepsilon_2 \Delta). \tag{5.42}$$

This idea is commonly referred to as *Sobolev smoothing* in the literature. In the past, the method has been successfully applied to shape optimization using a free node formulation, e.g., by Schmidt, Ilic, et al. [106].

Finally, note how the Sobolev smoothing procedure, the approximated Hessian operator symbols from Section 5.1, and the iso-perimeter problem (5.5) all lead to the Laplace-Beltrami operator, hinting at a deeper connection between the reinterpretation of the gradient in the Sobolev space and the Hessian matrix.

5.3 Combination of Reduced Shape Hessian and Parameterization

This section introduces new calculations, demonstrating how an approximated Hessian or a smoothing operator can be combined with the design parameterization. The results in this and the following section in this chapter have been published by Dick, Schmidt, and Gauger [33].

- In Subsection 2.3.2, the continuous formulation of the shape Hessian was introduced. Considering that there is a flow domain and a design surface, with $\partial\Omega = \Gamma$, the operator can be formulated on the surface $\mathrm{Hess}\,\mathscr{F}(\Gamma)$, or in the volume $\mathrm{Hess}\,\mathscr{F}(\Omega)$, where it typically has a rank deficit.

- In Section 3.2, the discretized reduced shape Hessian $\mathrm{D}_{xx} L(u, \lambda, x)$ appeared in the context of formulating optimality criteria for discrete adjoint methods. For the discrete optimization problem, this matrix is the respective approximation of the Hessian operator from the continuous formulation. Nonetheless, due to the high dimensions of this matrix, it is rarely explicitly computed in numerical optimization algorithms.

For the practical application of shape optimization to industrially relevant test cases, incorporating a mesh parameterization is imperative. It can be described by

$$x = M(p), \tag{5.43}$$

where $p \in \mathbb{R}^{n_p}$ is the design parameter vector. Section 3.2 already discussed how this affects the calculation of projected, reduced gradients and when formulating second order derivatives, a similar dilemma between two paradigms arises.

1. In the CFD simulation, the computational mesh is the discrete triangulation of the shape. Thus the reduced shape Hessian $\mathrm{D}_{xx}L(u,\lambda,x)$ is a natural approximation of the continuous shape Hessian on the level of mesh node coordinates.
2. For an efficient optimization algorithm, one would naturally want to formulate an approximated Newton step with respect to the design parameters p. This implies that any approximation of the Hessian should operate on the sensitivities of a reduced objective function with respect to the design parameters $\mathrm{D}_{pp}\tilde{F}(p)$.

Traditionally, the focus for Sobolev smoothing methods has been on free node formulations where the optimization works on the mesh coordinates directly, as stated in Section 5.2. Here, this is translated into the situation of a parameterized mesh description by deriving the connection between the two Hessian formulations.

Theorem 5.3.1 (discrete parameterization of the shape Hessian). *Assume that $\tilde{F} : \mathbb{R}^{n_p} \to \mathbb{R}, p \mapsto y$ is twice continuously differentiable for $p \in S_p \subset \mathbb{R}^{n_p}$ in the space of possible parameters, then for the second order derivatives of $\tilde{F}$ the following equation holds*

$$\mathrm{D}_{pp}\tilde{F}(p) = \mathrm{D}_p M(p)^T \mathrm{D}_{xx} L(u,\lambda,x)\, \mathrm{D}_p M(p) + \sum_{k=1}^{n_x} \frac{\partial}{\partial x_k} L(u,\lambda,x)\, \mathrm{D}_{pp} M_k(p). \tag{5.44}$$

Proof. For simplicity, consider the individual components of the derivatives, where the notation $(\cdot)_i$ means the i-th component. Corollary 3.2.1 states the connection between the total derivatives of $\tilde{F}$ and the partial derivatives of L. Consequently, applying the chain rule yields

$$\frac{\mathrm{d}}{\mathrm{d}p_i}\tilde{F}(p) = \left(\mathrm{D}_x L(u,\lambda,M(p))\mathrm{D}_p M(p)\right)_i = \sum_{k=1}^{n_x} \frac{\partial}{\partial x_k} L(u,\lambda,M(p))\, \frac{\partial}{\partial p_i} M_k(p). \tag{5.45}$$

Here, the adjoint calculus with the Lagrangian is crucial to remove dependencies on the flow state u and transform total into partial derivatives.

Now take a look at the (i,j)-th component of the second order derivative, i.e., the Hessian matrix.

$$\begin{aligned} \frac{\mathrm{d}^2}{\mathrm{d}p_i \mathrm{d}p_j}\tilde{F}(p) &= \frac{\mathrm{d}}{\mathrm{d}p_i}\left(\frac{\mathrm{d}}{\mathrm{d}p_j}\tilde{F}(p)\right) \\ &= \frac{\mathrm{d}}{\mathrm{d}p_i}\left(\sum_{k=1}^{n_x} \frac{\partial}{\partial x_k} L(u,\lambda,M(p))\, \frac{\partial}{\partial p_j} M_k(p)\right) \end{aligned} \tag{5.46}$$

Applying the product rule to each component of this sum results in

$$\frac{\mathrm{d}^2}{\mathrm{d}p_i \mathrm{d}p_j}\tilde{F}(p) = \sum_{k=1}^{n_x}\left(\frac{\mathrm{d}\partial}{\mathrm{d}p_i \partial x_k}L(u,\lambda,M(p))\frac{\partial}{\partial p_j}M_k(p) + \frac{\partial}{\partial x_k}L(u,\lambda,M(p))\frac{\partial^2}{\partial p_i \partial p_j}M_k(p)\right). \tag{5.47}$$

For the mixed second order derivative $\frac{\mathrm{d}\partial}{\mathrm{d}p_i \partial x_k}L(u,\lambda,x)$, the chain rule can be applied again to replace p_i with terms of x.

$$\begin{aligned}\frac{\mathrm{d}^2}{\mathrm{d}p_i \mathrm{d}p_j}\tilde{F}(p) = &\sum_{k=1}^{n_x}\left(\sum_{l=1}^{n_x}\frac{\partial^2}{\partial x_l \partial x_k}L(u,\lambda,M(p))\frac{\partial}{\partial p_i}M_l(p)\frac{\partial}{\partial p_j}M_k(p)\right) \\ &+\sum_{k=1}^{n_x}\frac{\partial}{\partial x_k}L(u,\lambda,M(p))\frac{\partial^2}{\partial p_i \partial p_j}M_k(p)\end{aligned} \tag{5.48}$$

Reordering the scalar multiplications yields

$$\begin{aligned}\frac{\mathrm{d}^2}{\mathrm{d}p_i \mathrm{d}p_j}\tilde{F}(p) = &\sum_{k=1}^{n_x}\sum_{l=1}^{n_x}\frac{\partial}{\partial p_i}M_l(p)\frac{\partial^2}{\partial x_l \partial x_k}L(u,\lambda,M(p))\frac{\partial}{\partial p_j}M_k(p) \\ &+\sum_{k=1}^{n_x}\frac{\partial}{\partial x_k}L(u,\lambda,M(p))\frac{\partial^2}{\partial p_i \partial p_j}M_k(p).\end{aligned} \tag{5.49}$$

The appearing sums are the result of matrix vector multiplications. Therefore, the expression for the complete Hessian matrix is

$$\mathrm{D}_{pp}\tilde{F}(p) = \mathrm{D}_p M(p)^T \mathrm{D}_{xx}L(u,\lambda,M(p))\,\mathrm{D}_p M(p) + \sum_{k=1}^{n_x}\frac{\partial}{\partial x_k}L(u,\lambda,M(p))\,\mathrm{D}_{pp}M_k(p), \tag{5.50}$$

proofing the statement. □

In the formulation presented here, Theorem 5.3.1 was first published by the author of this thesis in [33, Theorem 1]. The equation in Theorem 5.3.1 can be interpreted as a special case of the generalized Faà di Bruno formula [31, 38, 79, 114].

The statement shows the full effect of a parameter variation on the second order derivatives. However, the second term in Equation (5.44) will vanish in many cases, as the following corollary shows.

Corollary 5.3.2. *Let* $M : \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ *be a linear function, then*

$$\mathrm{D}_{pp}\tilde{F}(p) = \mathrm{D}_p M(p)^T \mathrm{D}_{xx}L(u,\lambda,x)\,\mathrm{D}_p M(p). \tag{5.51}$$

Alternatively, for arbitrary parameterizations M, let p^* *be an optimum of the Lagrangian L, then the same equation holds true*

$$\mathrm{D}_{pp}\tilde{F}(p^*) = \mathrm{D}_p M(p^*)^T \mathrm{D}_{xx}L(u,\lambda,x)\,\mathrm{D}_p M(p^*). \tag{5.52}$$

Proof. The first statement is trivial, since for a linear parameterization the second order derivatives must vanish, i.e., $\mathrm{D}_{pp}M_k(p) = 0$ and therefore,

$$\sum_{k=1}^{n_x}\frac{\partial}{\partial x_k}L(u,\lambda,x)\,\mathrm{D}_{pp}M_k(p) = 0 \tag{5.53}$$

proving the statement.
For the second statement, an optimal point p^* must fulfill the KKT conditions from Equation (3.10) and therefore $\mathrm{D}_x L = 0$. This implies that all individual components $\frac{\partial}{\partial x_k} L(u, \lambda, x) = 0$ and that

$$\sum_{k=1}^{n_x} \frac{\partial}{\partial x_k} L(u, \lambda, x) \, \mathrm{D}_{pp} M_k(p^*) = 0, \tag{5.54}$$

completing the proof. □

The impact of Corollary 5.3.2 will be discussed in more detail in the following subsection, but first the smoothing procedure on the parameter level is formulated and Corollary 5.3.2 helps to simplify this significantly. Now, the formula for the first order derivative of $\tilde{F}(p)$ already appeared in this work, as it can be expressed simply by a multiplication of Jacobian matrices

$$\mathrm{D}_p \tilde{F}(p) = \mathrm{D}_x L(u, \lambda, x) \mathrm{D}_p M(p). \tag{5.55}$$

Combining Equations (5.44) and (5.55) allows to transform a Newton step w for the design parameters

$$\mathrm{D}_{pp} \tilde{F}(p) w = -\mathrm{D}_p \tilde{F}(p)^T, \tag{5.56}$$

into

$$\mathrm{D}_p M(p)^T \mathrm{D}_{xx} L(u, \lambda, x) \mathrm{D}_p M(p) w = -\mathrm{D}_p M(p)^T \, \mathrm{D}_x L(u, \lambda, x)^T. \tag{5.57}$$

Here, the reduced shape Hessian with respect to the mesh $\mathrm{D}_{xx} L(u, \lambda, x)$ is used to formulate the update with respect to the design parameters, thereby eliminating the conflicting paradigms described above. This means approximations of the reduced shape Hessian $\mathscr{B} \approx \mathrm{D}_{xx} L(u, \lambda, x)$ w.r.t. the mesh coordinates can be utilized in the formula

$$\mathrm{D}_p M(p)^T \mathscr{B} \mathrm{D}_p M(p) w = -\mathrm{D}_p M(p)^T \, \mathrm{D}_x L(u, \lambda, x)^T, \tag{5.58}$$

to compute an update step w. Using the results on Sobolev smoothing from the previous Section 5.2, the Laplace-Beltrami operator from Equation (5.42) is used.

$$\mathrm{D}_p M(p)^T (\varepsilon_1 \mathrm{I} - \varepsilon_2 \Delta) \mathrm{D}_p M(p) w = -\mathrm{D}_p M(p)^T \mathrm{D}_x L(u, \lambda, x)^T \tag{5.59}$$

This equation is the central result presented in this thesis. It enables the Sobolev reinterpretation of the gradient for arbitrary finite-dimensional spaces of design parameters. The geometric structure necessary to formulate a meaningful Laplace operator Δ is kept by approximating the mesh Hessian, while simultaneously taking arbitrary, linear parameterizations into account and removing the need to work on a free node parameterization.

Surface and Volume Mesh Formulation

Corollary 5.3.2 shows how the second order derivatives of the parameterization can be neglected, but so far in this Section 5.3, the mesh parameterization was treated as a single mapping. Subsection 2.3.3 explained the mesh parameterization in detail and introduced the distinction between the surface parameterization M_{S} and the volume parameterization, or mesh deformation, M_{V}. On the other hand, Sobolev smoothing in a free node formulation is traditionally associated with the surface mesh sensitivities [106]. This leaves two open questions.

1. Is the Laplace-Beltrami operator formulated on the surface Γ or in the flow volume Ω?
2. How does this choice affect the formulations in Theorem 5.3.1 and Corollary 5.3.2?

For the first question, it is essential to note that the analytic shape Hessian $\operatorname{Hess}\mathscr{F}$ oftentimes has a rank deficit when formulated in the flow volume [105], i.e., moving interior points of the flow domain does not affect objective functions depending on a surface integral. Nonetheless, the discrete reduced shape Hessian $\mathrm{D}_{xx}L(u,\lambda,x)$ can be approximated on a volume level.
For the second question, recall how the parameterization of the individual mesh components and especially their Hessian matrices w.r.t the parameters $\mathrm{D}_{pp}M_k$ are a part of Equation (5.44). Because of the two-staged parameterization approach from Equation (2.62), each M_k is a combined mapping

$$M_k : \mathbb{R}^{n_p} \to \mathbb{R}; p \mapsto (M_\mathrm{V})_k(M_\mathrm{S}(p)). \tag{5.60}$$

Since these are scalar-valued functions, applying the Faà di Bruno formula a second time yields

$$\mathrm{D}_{pp}M_k(p) = \mathrm{D}_pM_\mathrm{S}(p)^T\mathrm{D}_{ss}(M_\mathrm{V})_k(s)\,\mathrm{D}_pM_\mathrm{S}(p) + \sum_{l=1}^{n_s}\frac{\partial}{\partial s_l}(M_\mathrm{V})_k(s)\,\mathrm{D}_{pp}(M_\mathrm{S})_l(p). \tag{5.61}$$

Insertion of Equation (5.61) into Equation (5.44) gives an extended formulation of the original result. Thus, the complete second order derivatives are

$$\begin{aligned}\mathrm{D}_{pp}\tilde{F}(p) = {} & \mathrm{D}_pM_\mathrm{S}(p)^T\mathrm{D}_sM_\mathrm{V}(s)^T\mathrm{D}_{xx}L(u,\lambda,x)\,\mathrm{D}_sM_\mathrm{V}(s)\mathrm{D}_pM_\mathrm{S}(p) + \\ & \sum_{k=1}^{n_x}\frac{\partial}{\partial x_k}L(u,\lambda,x)\,\mathrm{D}_pM_\mathrm{S}(p)^T\mathrm{D}_{ss}(M_\mathrm{V})_k(s)\,\mathrm{D}_pM_\mathrm{S}(p) + \\ & \sum_{k=1}^{n_x}\frac{\partial}{\partial x_k}L(u,\lambda,x)\left(\sum_{l=1}^{n_s}\frac{\partial}{\partial s_l}(M_\mathrm{V})_k(s)\,\mathrm{D}_{pp}(M_\mathrm{S})_l(p)\right).\end{aligned} \tag{5.62}$$

Now, Corollary 5.3.2 can be applied to the parameterizations from Subsection 2.3.3. In the surface case, there is no internal mesh deformation M_V to consider. The Hicks-Henne functions, as in Equation (2.65), and the FFD boxes, as in Equation (2.70), are both clearly linear in the design parameters p and so their second order derivatives $\mathrm{D}_{pp}M_\mathrm{S}(p)$ must vanish. Let $(\varepsilon_1\mathrm{I} - \varepsilon_2\Delta_\Gamma) \in \mathbb{R}^{n_s \times n_s}$ be a discretization of the Laplace-Beltrami operator restricted to the surface, then the smoothing equation takes the form

$$\mathrm{D}_pM_\mathrm{S}(p)^T(\varepsilon_1\mathrm{I} - \varepsilon_2\Delta_\Gamma)\mathrm{D}_pM_\mathrm{S}(p)w = -\mathrm{D}_pM(p)^T\mathrm{D}_xL(u,\lambda,x)^T. \tag{5.63}$$

Next, for the volume case the same arguments about the surface parameterization M_S still hold. Here, an additional cross derivative term remains from Equation (5.62),

$$\sum_{k=1}^{n_x}\frac{\partial}{\partial x_k}L(u,\lambda,x)\,\mathrm{D}_pM_\mathrm{S}(p)^T\mathrm{D}_{ss}(M_\mathrm{V})_k(s)\,\mathrm{D}_pM_\mathrm{S}(p). \tag{5.64}$$

In this work, the mesh deformation is computed via a linear elasticity approach, see Equation (2.61). This means evaluating $x = M_\mathrm{V}(s)$ involves solving a linear equation system, which is clearly a linear process. Therefore, the terms $\mathrm{D}_{ss}(M_\mathrm{V})_k(s)$ will vanish, resulting in

$$\mathrm{D}_pM_\mathrm{S}(p)^T\mathrm{D}_sM_\mathrm{V}(s)^T(\varepsilon_1\mathrm{I} - \varepsilon_2\Delta)\mathrm{D}_sM_\mathrm{V}(s)\mathrm{D}_pM_\mathrm{S}(p)w = -\mathrm{D}_pM(p)^T\mathrm{D}_xL(u,\lambda,x)^T, \tag{5.65}$$

where Δ is the volume Laplace operator. Both formulations will be tested in Chapter 7, although Equation (5.63) is closer to the original theoretical results about Sobolev smoothing.

5.4 Inclusion into SQP algorithms

To use the results of this chapter in a practical, aerodynamic optimization application, the parameterized reinterpretation of the gradient must be combined with computing the search direction in an optimization algorithm. To achieve this goal, recall the SQP methods in Section 3.5. Here, the second order derivative information is accounted for by the reduced Hessian of a Lagrangian function, and thus it seems only natural to use the left-hand side matrix from Equation (5.59) in Algorithm 3.5.2.

$$\tilde{D}_{pp}L^{\mathrm{Ext}} \approx \bar{B} = \mathrm{D}_p M(p)^T(\varepsilon_1 \mathrm{I} - \varepsilon_2 \Delta)\mathrm{D}_p M(p) \tag{5.66}$$

This choice fits naturally into the basic concept of Quasi-Newton methods and the approximation can also be used together with inexact function values and gradients since it does not suffer from poor derivative approximation to the same degree as other Hessian approximation techniques. Equation (5.66) is hence a suitable preconditioner for constrained One Shot optimization using Algorithm 4.3.1 as well. This section discusses some of the implications Sobolev smoothing as a Hessian approximation has for the optimization algorithm in more detail.

The Marathos effect

In numerical optimization, the term Lagrangian is used in different contexts for different functions, which may differ in the constraints they take into account. Consider how the original Lagrangian for the discrete adjoint framework from Section 3.1 and the extended Lagrangian from the SQP algorithm in Section 3.5 are connected

$$L^{\mathrm{Ext}}(u,\lambda,p) = L(u,\lambda,p) + \theta^T E(u,M(p)). \tag{5.67}$$

This difference can have severe implications when applying SQP algorithms since the quadratic subproblem in each SQP iteration expects an approximation of $\tilde{D}_{pp}L^{\mathrm{Ext}}$. Assuming that for a given objective function one can approximate $\tilde{D}_{pp}L$ by some matrix $\bar{B}$, then the second order information about the equality constraint term E should be considered by the optimization algorithm as well,

$$\tilde{D}_{pp}L^{\mathrm{Ext}}(u,\lambda,p) = \tilde{D}_{pp}L(u,\lambda,p) + \theta^T \tilde{D}_{pp}E(u,M(p)) \approx \bar{B} + \theta^T \tilde{D}_{pp}E(u,M(p)). \tag{5.68}$$

Neglecting the term $\theta^T \tilde{D}_{pp}E(u,M(p))$ can lead to undesirable numerical behavior and for larger design update steps v the optimizer will violate the equality constraints since the curvature of the constraint is not taken into account. While such difficulties can be overcome by choosing small design updates v, this significantly slows down the SQP algorithm, which is undesirable too.

This behavior is closely linked to similar issues, where SQP methods struggle to adhere to highly nonlinear constraints $E(u,x)$. Those are well known in the literature and are referred to as the *Marathos effect*, see [90, Section 18.11] for details. While some heuristics to avoid this issue are known, they usually deal with iterative Hessian approximations, e.g., BFGS updates, or adapt the used merit function for line searches. Unfortunately, this means they cannot easily be applied to the situation at hand.

Another idea is to extend the approximation $\bar{B}$ by a regularization term $c\mathrm{I}$, with an adequate constant $\varepsilon_3 \approx \|\theta^T \tilde{D}_{pp}E(u,M(p))\|$. This approach will be discussed at length in the next section. As an overall expression the approximated, extended Lagrangian becomes

$$\tilde{D}_{pp}L^{\mathrm{Ext}}(u,\lambda,p) = \tilde{D}_{pp}L(u,\lambda,p) + \theta^T \tilde{D}_{pp}E(u,M(p)) \approx \bar{B} + \varepsilon_3 \mathrm{I}, \tag{5.69}$$

and it is then possible to apply Algorithms 3.5.1 and 3.5.2 with $B = \bar{B} + \varepsilon_3 \mathrm{I}$.

Hybrid Laplace-Beltrami operator

In Section 5.3, the combination between the parameterization and the Hessian on the mesh was introduced and the following equation for the connection between both sides was established,

$$\mathrm{D}_{pp}\tilde{F}(p) = \mathrm{D}_p M(p)^T \mathrm{D}_{xx} L(u,\lambda,x)\, \mathrm{D}_p M(p). \tag{5.70}$$

Using Sobolev smoothing to approximate the Hessian of an objective function results in a scaled Laplace-Beltrami operator,

$$\mathrm{D}_{pp}\tilde{F}(p) \approx \mathrm{D}_p M(p)^T (\varepsilon_1 \mathrm{I} - \varepsilon_2 \Delta)\, \mathrm{D}_p M(p). \tag{5.71}$$

The key motivation behind this is a reinterpretation of the directional derivative in a different scalar product resulting in the H^1-gradient. In Section 5.1, this was investigated by a literature survey on operator symbols for the exact Hessian of a drag minimization problem. Until now, the presented theory was limited to free node optimization, meaning that the mesh coordinates are also the design parameters, but this can be extended with the new results from this thesis.

When dealing with a parameterization, an interested reader might ask why it is impossible to directly state a Laplace-Beltrami operator on the design parameters. Obviously, a naive notation like

$$\mathrm{D}_{pp}\tilde{F}(p) \approx \varepsilon_3 \mathrm{I}_p - \varepsilon_4 \Delta_p, \tag{5.72}$$

makes no sense since a meaningful mathematical operator Δ_p cannot be defined for arbitrary parameters. The components of the vector p can be an arbitrary set of real-valued numbers from a variety of ranges and there is no need for an attached geometric structure. On the contrary, the Laplace operator requires a differentiable manifold to be defined, e.g., in the case of the mesh coordinates, this is part of the spaces $\mathbb{R}^2$ or $\mathbb{R}^3$ in which the mesh is embedded, depending on the dimension of the test case.

Yet, the Laplace-Beltrami operator is a linear combination and in a numerical sense, the two parts serve two distinct functions. First, the Laplace part has smoothing properties and dissipates numerical noise and high-frequency errors on the mesh. Second, the identity part serves a regularizing purpose, ensuring that the new gradient in the Sobolev space H^1 is still close to the original derivatives.

A close examination reveals that there are three individual components to consider.

1. The term $\varepsilon_1 D_p M(p)^T \mathrm{I} D_p M(p)$ is the identity operator on the mesh projected to the parameters, which arises naturally from using Sobolev smoothing on the mesh. Some observations about this expression can be made.

    - There is a connection to the spectral norm of the parameterization since it is the square root of the largest eigenvalue in the spectrum

    $$\|D_p M(p)\|_2 = \sqrt{\sigma(D_p M(p)^T \mathrm{I} D_p M(p))} \tag{5.73}$$

    and as such, preconditioning with this matrix scales the gradient.

    - The resulting matrix is not necessarily diagonal dominant. Instead, a resemblance to the form of a covariance matrix, with correlations between parameters, could be seen in its entries

    $$\left(D_p M(p)^T \mathrm{I} D_p M(p)\right)_{ij} = \sum_{k=1}^{n_x} D_{p_i} M_k(p) D_{p_j} M_k(p). \tag{5.74}$$

  - If the derivatives from the adjoint solver have high sensitivities in one component, they can be spread to other components when using the inverse of this matrix as a preconditioner. In that sense, it is similar to the dissipation property of the Laplace operator, albeit it lacks the former's regularity.

2. The term $\varepsilon_2 D_p M(p)^T \Delta D_p M(p)$, as the discretized Laplace operator on the mesh projected to the parameters, is the closest available approximation for a Laplace operator on the parameters. In that sense, it has several properties.
   - As the smoothing part of the equation, this term helps with regularity and error dampening.
   - In the original Sobolev reinterpretation, higher order derivative information is introduced by the Laplace operator.
   - Numerical experiments show that for too large values of ε_2 this component will cause too much smoothing and slow down the progress in the optimization process.

3. The parameter identity $\varepsilon_3 \mathrm{I}_p$ can be interpreted in different ways.
   - Mathematically, it is the trivial, scaled identity on each component of the design parameter vector.
   - Using only this identity I_p as the matrix B in the reduced SQP methods transforms them into a projected gradient descent.
   - The discussion earlier in this section shows how it can be beneficial to approximate the Hessian of the equality constraints, from the extended Lagrangian $L^{\mathrm{Ext}}(u, \lambda, p)$, by $\theta^T \tilde{D}_{pp} E \approx \varepsilon_3 \mathrm{I}_p$ with $\varepsilon_3 \approx \|\theta^T \tilde{D}_{pp} E\|$.
   - Numerical experiments reveal that including the parameter identity I_p, with a small ε_3, helps to regularize the SQP optimization and keep the constraints.

Overall, this leaves three components from which to build a suitable Hessian approximation. To achieve the best possible numerical performance, the optimization in this thesis uses a linear combination of all three. This means that for the approximated second order derivative matrix B in Algorithms 3.5.1, 3.5.2, and 4.3.1 the following formula is used,

$$B = \mathrm{D}_p M(p)^T (\varepsilon_1 \mathrm{I} - \varepsilon_2 \Delta) \mathrm{D}_p M(p) + \varepsilon_3 \mathrm{I}_p. \tag{5.75}$$

Because this term combines operators from the mesh level and the design parameters into one approximation, it is called the *hybrid Laplace-Beltrami operator*.

It can be motivated by the fact that parameterized Sobolev reinterpretation of the gradient does not directly imply on which level the identity lives. For classical free node optimization, such a situation would not occur, as only the space of mesh coordinates, as a triangulation of the surface manifold, exists there. Furthermore, using the design parameter identity can be motivated as a regularization for the equality constraints in the SQP framework.

As mentioned, when using a pure parameter identity $B = \mathrm{I}_p$, the SQP algorithms are equivalent to projected gradient descent methods. Therefore, the hybrid Laplace-Beltrami approach offers the user a variety of possible methods, ranging from fully-approximated Newton algorithms to pure gradient descent. Utilizing this variety and choosing suitable values $\varepsilon_1, \varepsilon_2, \varepsilon_3$ for a given test case helps the optimization to achieve maximum performance.

Chapter 6

Implementation

This chapter presents an overview of the implementations done as part of this thesis. Starting by giving an introduction of the different used software tools in Section 6.1, and how they are utilized to implement the algorithms and run the test cases. Following this introduction, the implementations of new solvers and driver routines inside the SU2 executables are discussed in Section 6.2, as well as presenting the optimizer implementation done in the Python script-based FADO package in Section 6.3.

The source code of SU2 is available under `https://github.com/su2code/SU2` and FADO is available under `https://github.com/pcarruscag/FADO`. The implementations used can be found in the authors respective forks of these repositories.

A brief discussion of key features of the implementation presented in this chapter has been previously published in the appendix of the paper by Dick, Schmidt, and Gauger [33].

6.1 Used Software Packages

Here, the most important software packages used are listed and a short explanation of their functionality, features, and code structure is given. This includes the SU2 framework, as the CFD code used in this thesis to calculate flow and adjoint solutions, CoDiPack and MeDiPack, as they provide the necessary AD capabilities, and the FADO framework, as an optimization interface in Python.

6.1.1 SU2

The key aspects of this work, namely the development and implementation of a new Sobolev smoothing method incorporating design parameterizations, are done inside the context of aerodynamic shape optimization. Such an optimization requires solving the underlying flow equations, presented in Subsection 2.1.1, and evaluating aerodynamic functionals, e.g., the ones introduced in Subsection 2.1.2. Furthermore, the derivatives of those functionals must be computed, which is done using the discrete adjoint approach discussed at length in Chapter 3.

All of the mentioned tasks are computed using the SU2 code [93, 94, 35], a free, open-source multiphysics package that provides various solvers. These include flow solvers for the most common compressible and incompressible flow equations, like Euler, Navier-Stokes, and RANS equations and corresponding turbulence model solvers, see Section 2.1. In addition, a variety of solvers for

other relevant sets of partial differential equations from engineering and physics are available. These include elasticity and conjugate heat transfer solvers and their corresponding coupling with the flow solvers.
Especially relevant for this work is the included discrete adjoint solver [4] since it provides the capability to differentiate the flow solver itself in a robust and computationally efficient way. The underlying mathematical structure is based on the fixed point formulation of the flow problem discussed in Subsection 3.1. Algorithmic differentiation is applied to compute the appearing derivative terms, following the theoretical introduction of the topic provided in Subsection 3.3. It should be emphasized that algorithmic differentiation fits nicely into a discrete adjoint approach since it adheres to the fundamental principle of 'first discretize, then optimize', and thus its use results in very efficient adjoint code. For comparison, the quotient between a flow simulation and an according adjoint evaluation in SU2 is down to ca. 2.3 in runtime [4] and ca. 5 to 10 in terms of memory consumption [103].

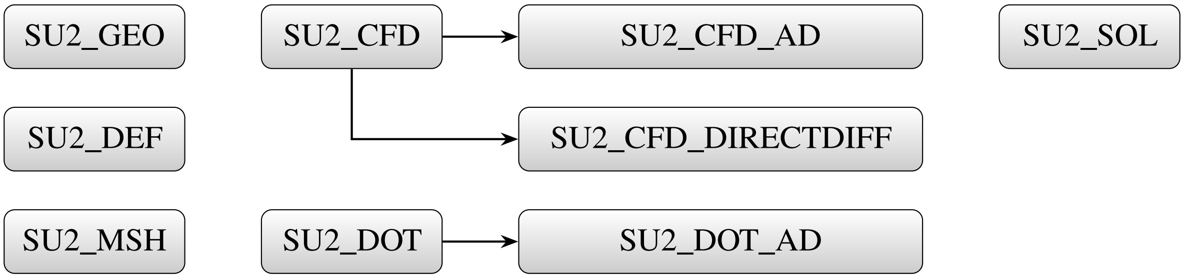


Figure 6.1: Overview of C++ executables in SU2.

The basic code structure of SU2 determines how the algorithms established so far can be implemented. Viewing it from a top-down perspective, there are several distinct executables shown in Figure 6.1. They are all implemented in C++, heavily utilizing class structures and templates, and each of these could be compiled as a stand-alone tool, although they are designed to work together in sequential order. The most relevant executables used here for the implementation are:

1. **SU2_CFD:** This can be seen as the core of SU2 since it provides the direct execution of the different solvers. Simulations with the flow, turbulence, and additional multiphysics solvers are run using this executable.

2. **SU2_CFD_AD:** This executable provides the discrete adjoint versions of the solvers from SU2_CFD. It is created from the differentiated code of the primal solvers using algorithmic differentiation, see the following Subsection 3.3. It is called to do the sensitivity analysis and the derivative calculations. Additionally, it can also run the simulations itself, though this will be slower than running SU2_CFD due to overhead.

3. **SU2_DEF:** As an implementation of the supported parameterizations, this executable takes the design parameters as input and calculates the mesh deformation. The output is a new mesh file containing the deformed design, which can then be used for flow and adjoint simulations.

4. **SU2_DOT_AD:** This is another executable using algorithmic differentiation. It provides a differentiated calculation of the mesh parameterization, i.e., it can calculate derivatives for the deformation process. Note that the SU2_DOT_AD executable provides a differentiation of the mesh deformation done by AD. In comparison, the SU2_DOT executable computes them by finite differences instead.

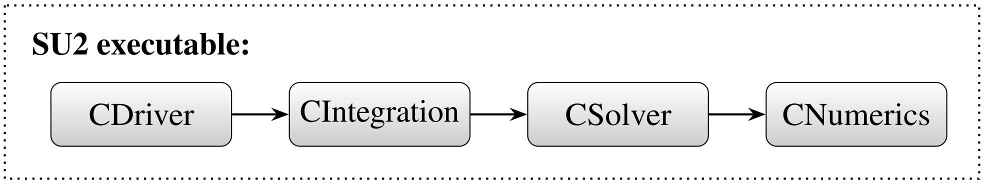


Figure 6.2: Class hierarchy in SU2.

Clearly, the executables SU2_CFD and SU2_CFD_AD contain the core functions for solvers and simulations, so the optimal place to implement Sobolev gradient treatment is as a part of the discrete adjoint capability. To better illustrate their underlying working principles, look at the class hierarchy of SU2_CFD and SU2_CFD_AD, as depicted in Figure 6.2. The different classes are organized following a pattern of generalization, from general physics to small scale numerical formulas on the individual mesh cells.

1. **CDriver:** Driver classes represent the kind of physical problem. This includes options for whether this is a primal simulation, e.g., flow simulations, or an adjoint run. The driver class holds the spatial distribution of the problem, e.g., being all on a single zone, for flow solutions, or having multiple zones, for fluid-structure interaction. Additionally, the driver controls the time dependency of the physics. For example, this could be a steady state problem with pseudo time-stepping or a time dependent, unsteady problem.

2. **CIntegration:** Integration classes manage the individual timesteps for the simulation, or pseudo timesteps in case of steady state problems. This class calls the individual solvers, manages the data exchange between them, and utilizes their output to execute a numerical time integration step.

3. **CSolver:** Solver classes implement a specific set of governing equations. Possible options include flow equations, turbulence models, structural mechanics, etc. They manage everything necessary for one evaluation of the iterative fixed point solver and store the current, temporary solution fields.

4. **CNumerics:** In the numerics classes, the core routines of the individual solution schemes are implemented. They work in close conjunction with the solver classes to provide the discrete solution scheme for the physical problem.

While there are other classes for different purposes, such as the geometric representation of the dual mesh, solution and history file output, etc., this brief overview should be sufficient to understand the implementation steps discussed throughout the rest of this chapter.
Finally, it is worth mentioning that the SU2 framework also comes equipped with a series of Python

scripts. They provide easier access, allowing for the call to and execution of the compiled main executables, and include optimization capabilities. Several different optimizers, stemming from the SciPy optimization package [123], are provided in the standard version of the interface. However, SciPy's own included second order Quasi-Newton method, the SLSQP algorithm [69], does not allow the user to provide Hessian approximations. More importantly, the existing Python scripts are unsuited for applying One Shot methods. Therefore, a different interface is used in this thesis, which will be introduced in Subsection 6.1.3.

6.1.2 CoDiPack and MeDiPack

In the presented work, all flow calculations and adjoint evaluations are done using the SU2 software, as stated in Subsection 6.1.1. The standard AD tool included within SU2 is *CoDiPack*[1] [103], the name being an abbreviation for 'Code Differentiation Package'. CoDiPack is an operator overloading AD tool for C++ code, especially designed for fast AD support in high performance computing environments. The tool is based on expression templates and static polymorphism, both features of modern C++ code, and comes in the form of a header only library making integration into existing code very simple and straightforward. Support includes the forward and reverse mode of AD, including some useful features and optimizations for the reverse mode that are relevant here. This includes the possibility to run multiple reverse mode evaluations of the same recorded primal evaluation and, in addition, forward evaluations of the recorded tape from reverse mode. Subsection 6.2.1 will further discuss how those features are utilized in this work.

Parallelization is one aspect of modern software architecture, especially in high performance computing, where AD traditionally struggles. Parallel computations are necessary to run large simulations and optimizations, e.g., the kind encountered in CFD applications. However, many AD tools lack proper coverage of common parallelization techniques, like message passing, shared memory, or vectorization. In the case of parallelization via message passing (MPI) for CoDiPack the additional tool *MeDiPack*[2], standing for 'Message Differentiation Package', is available. MeDiPack offers strong support for AD with MPI, by providing a C++ library for overloading MPI communication. The communication interface provided by MeDiPack replaces the standard MPI calls in the source code with overloaded communication calls, keeping the same general layout and structure. The process is similar to replacing the floating-point type with CoDiPack, therefore being non-intrusive to the code and user friendly for the developer. With this approach, a majority of ca. 80% of the MPI 3.1 standard is covered.

When working together, MeDiPack can automatically set up the adjoint communication calls CoDiPack needs for reverse mode derivative calculations. In such a fashion, it is possible to differentiate complex codes with many MPI communications while keeping the additional work for the developer at a minimum. In SU2, the communication routines are automatically replaced by MeDiPack routines, if the correct build options for AD are set, thus providing differentiated MPI code out of the box. Since the test cases of this work are generally done on a high performance cluster architecture, and the methodology aims at industrially relevant applications, the availability of parallel AD tools is an essential ingredient of the overall implementation.

---

[1] CoDiPack – Code Differentiation Package, `https://www.scicomp.uni-kl.de/software/codi/`

[2] MeDiPack – Message Differentiation Package, `https://www.scicomp.uni-kl.de/software/medi/`

For the proposed implementation of parameterized Sobolev smoothing, special focus must be given to the efficient computation of the mesh parameterization Jacobian $\mathrm{D}_p M(p)$ and its transposed $\mathrm{D}_p M(p)^T$. In Section 6.2.1, the use of CoDiPack for this calculation is further discussed.

6.1.3 FADO

As the topic of this thesis is the use of a combination of Sobolev smoothing with a shape parameterization in an optimization process, it is necessary to implement the Algorithms 3.5.2 and 4.3.1 from previous chapters. This means that the optimizer must compute flow simulations and discrete adjoints, e.g., by reverse accumulation or piggyback, and use different gradients, e.g., fully converged, accurate derivatives, or approximated, intermediate derivatives.

As mentioned, SU2 itself includes several Python scripts to automate various tasks. However, they have a couple of limiting factors attached to them. Because of this, the optimization algorithms are implemented in another extended Python framework. To better understand the involved reasoning, first specify the key requirements for the optimization interface.

1. Provide an interface between the optimizer and the flow and adjoint solver executables. Functions, constraints, and gradients should be available to the optimization algorithm via wrapped function calls in Python.
2. Include support for multiple adjoint solver settings and methods, i.e., work with reverse accumulation or piggyback. Additionally, allow the optimizer to call newly implemented solvers, e.g., to compute the Laplace-Beltrami operator.
3. Support for geometric evaluations, via the mesh parameterization, to compute additional constraints depending on the geometry and their respective derivatives.
4. Handling of the required file I/O and management of (sub-)directories. The optimizer calculates multiple designs throughout an optimization. These intermediate deformed meshes, flow and adjoint solutions, and other files should be stored in an organized subdirectory structure.
5. Control and guide the optimization process. Everything should be callable and controlled by a command script. The script should provide all information the optimization backend needs and control the data flow.

In this thesis, the relatively recent FADO[3] framework is used to fulfill these requirements and implement the optimization algorithms. FADO stands for 'Framework for Aerostructural Design Optimization' and is originally designed to couple different solvers and optimization packages for multiphysics problems. It offers a flexible toolbox to couple the SU2 executables with arbitrary optimization packages in Python and provides several beneficial features.

1. Customized, user-defined command line calls to the executables. While the standard SU2 scripts use the same parameters, e.g., executable names and number of MPI processes, for all calls in the optimization, in FADO it is instead possible to completely customize all options for each individual call to the different solvers.

---

[3] FADO - Framework for Aerostructural Design Optimization, https://github.com/su2code/FADO

2. Adaptation of the configuration file via a label replacement system. In FADO, the user provides a configuration template containing labels for key settings, e.g.,

   ```
   % Mathematical problem (DIRECT, DISCRETE_ADJOINT)
   MATH_PROBLEM= __MATH_PROBLEM__
   %
   % Number of iterations for single-zone problems
   ITER= __ITER__
   ```

   These labels can then be specified, i.e., they are replaced with different, individual options for each function or gradient evaluation call. The label replacement system allows users to change every SU2 solver setting quickly and efficiently.

3. Active checks for changed design variables and keeping track of computed solutions. The toolbox stores, whether or not the function or gradient computation has been called already for certain values and starts a new simulation only if the design has changed. Also, these features keep track of the solution files and can use them as restart points for simulations in the next optimization step.

4. Custom handling of (sub-)directories and file I/O. Within FADO, the user can specify where simulations are done by providing a subdirectory name in which to run the executable, as well as input files. After termination, it is possible to run user provided postprocessing commands, e.g., copying back results.

These features will become helpful in Section 6.3, where they are employed in implementing the general structure of a reduced SQP algorithm. The points listed here will help ensure that the SQP algorithm is flexible enough to work with fully converged solutions and intermediate values in a One Shot process.

6.2 Implementations in SU2

After introducing the used software packages, the new implementations in SU2 for this thesis are presented in this section. An overview of the different solvers and how they fit into the overall framework is given, including the implementation of new solver classes and the extension of the existing ones. This section's objective is not to state the complete source code, as this would be too long, but to highlight the relevant ideas for a reproduction.

6.2.1 Gradient Smoothing Solver

In Chapter 5, the mathematical formulation for combining a Sobolev reinterpretation of the gradient with the parameterization was introduced. The next logical step is to implement a solver for the key equation to conduct an approximated Newton step, see Equation (5.59) in Section 5.3.

$$\mathrm{D}_p M(p)^T (\varepsilon_1 \mathrm{I} - \varepsilon_2 \Delta) \mathrm{D}_p M(p) w = -\mathrm{D}_p M(p)^T \mathrm{D}_x L(u, \lambda, x)^T \tag{6.1}$$

In the following, the major steps in the implementation of such a new solver class are outlined. The CGradientSmoothing solver is introduced as a new specialization of the basic CSolver class of SU2.

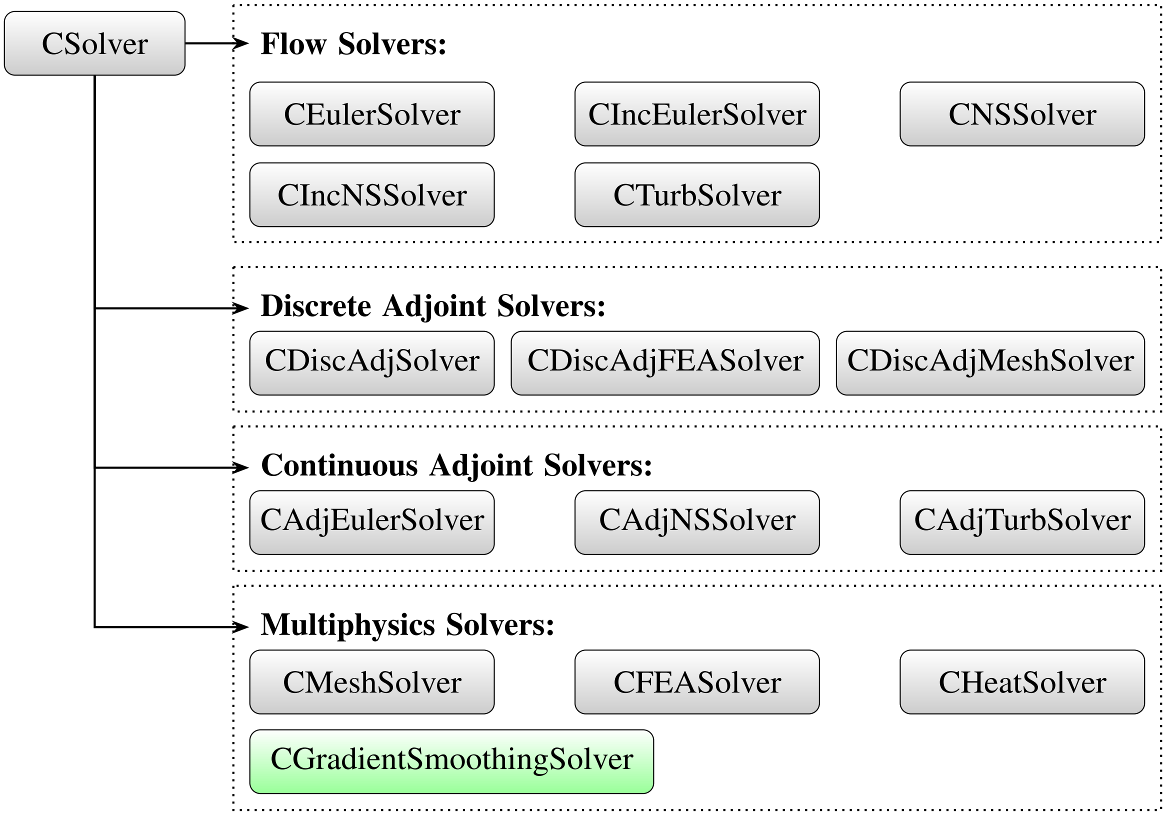


Figure 6.3: Solver class structure in SU2.

The hierarchy of different classes in SU2 was explained in Subsection 6.1.1 and shown in Figure 6.3. The solver classes, derived from the CSolver template, can be roughly grouped into different types for flow, turbulence, adjoints, etc. The new CGradientSmoothing solver class, which is necessary for the implementation, is shown in green in Figure 6.3. It is intended to fulfill three distinct steps, aiming to solve Equation (6.1).

1. A discrete representation of the Laplace-Beltrami operator $(\varepsilon_1 \mathrm{I} - \varepsilon_2 \Delta)$, by an expansion of the SU2 internal finite element numerics.
2. An efficient AD-based implementation for the evaluation of arbitrary matrix vector products with the parameterization Jacobian $\mathrm{D}_p M(p)$ and its transposed $\mathrm{D}_p M(p)^T$.
3. An overarching structure which enables the solution of the complete linear equation system for different settings and parameters.

The means to achieve this goal are outlined in the following.

Extension of the linear finite element solver

As the central operator in Equation (6.1) $(\varepsilon_1 \mathrm{I} - \varepsilon_2 \triangle)$ must be computed and naturally, a solver for this equation should possess some properties.

1. As the term is an elliptic partial differential operator and contains the Laplace operator as the main derivative part, it seems adequate to use a finite element method to solve it. Such methods have proven capable of solving elliptic PDEs and are widely used throughout the engineering community.
2. To achieve a high computational performance, the finite element code should be fully integrated into the existing SU2 structure. Either by implementing it inside the framework or by including an external C++ implementation. The first approach is chosen within this work, although the second one seems feasible as well, considering the high number of available open-source C++ libraries for finite elements, e.g., deal.II [13].

To achieve the best possible performance, while keeping good maintainability, the already existing linear finite element solver in SU2 is used as a basis for the implementation. This solver was initially introduced by Sánchez [41] for structural mechanics. In its original form, it is intended for the analysis of linear and special cases of nonlinear elasticity equations and uses linear finite elements. The implementation is based on the equations derived in the book by Bonet and Wood [20]. Nonetheless, since the solver was initially intended for structural mechanics, a couple of extensions are necessary to enable the computation of the operator in system (6.1).

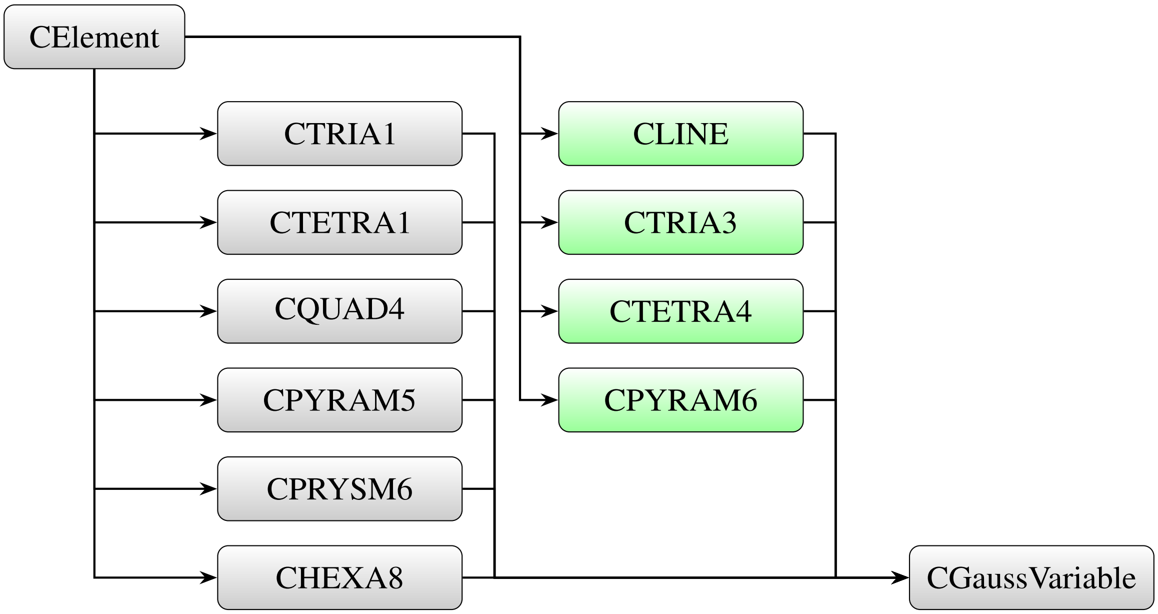


Figure 6.4: Element class structure in SU2.

Consider the weak partial differential equation seen in Equation (5.29). A general finite elements approach for discretizing the Laplace-Beltrami operator can be written as

$$\forall \phi \in \mathscr{V}(\Omega) : \varepsilon_1 \int_\Omega f(x)\phi \, \mathrm{d}x - \varepsilon_2 \int_\Omega \langle \nabla_x f(x), \nabla_x \phi \rangle_2 \, \mathrm{d}x = \int_\Omega g(x)\phi \, \mathrm{d}x. \tag{6.2}$$

The basic idea of finite elements is to find a solution of the weak formulation for all test functions ϕ from a vector space $\mathscr{V}(\Omega)$, called the test space. To do this, choose a finite-dimensional space called ansatz space, spanned by a set of basis functions, and express $f(x)$ in terms of that basis. Many approaches with different function spaces exist, leading to different finite element methods. Following a Ritz-Galerkin approach, the test and ansatz space are the same, see [25, Chapter II.4]. In the SU2 implementation, piecewise linear basis functions are used.

Definition 6.2.1 (linear ansatz space)**.** *Let $\mathscr{T}$ be a triangulation of Ω, then*

$$S_{\mathscr{T}} = \left\{ \phi \in C^0(\Omega) \;\middle|\; \phi|_T \in P_1,\ T \in \mathscr{T} \right\} \tag{6.3}$$

is the space of all continuous, piecewise linear functions on the triangulation that are polynomials of order one, denoted by P_1, on each element of the triangulation.

Choosing a canonical basis of hat functions $\{\phi_i \in S_{\mathscr{T}}\}_{i=1,\dots,n_a}$, i.e., functions which are 1 on one node of $\mathscr{T}$ and 0 on all others, and searching for the best approximation of f in terms of $S_{\mathscr{T}}$ leads to the problem of finding coefficients a_i, such that

$$f(x) = \sum_{i=0}^{n_a} a_i \phi_i. \tag{6.4}$$

The functions $\phi_i \in S_{\mathscr{T}}$ are called ansatz functions. Insertion into the weak formulation yields a system of equations.

$$\forall j = \{1,\dots,n_a\} : \varepsilon_1 \int_\Omega \sum_{i=0}^{n_a} a_i \phi_i(x)\phi_j(x)\,\mathrm{d}x - \varepsilon_2 \int_\Omega \sum_{i=0}^{n_a} a_i \langle \nabla_x \phi_i(x), \nabla_x \phi_j(x)\rangle_2\,\mathrm{d}x = \int_\Omega g(x)\phi_j(x)\,\mathrm{d}x \tag{6.5}$$

This can be rewritten into a linear system of equations

$$Aa = b, \tag{6.6}$$

with the so-called *stiffness matrix*

$$A_{ij} = \varepsilon_1 \int_{T_j} \phi_i(x)\phi_j(x)\,\mathrm{d}x - \varepsilon_2 \int_{T_j} \langle \nabla_x \phi_i(x), \nabla_x \phi_j(x)\rangle_2\,\mathrm{d}x, \tag{6.7}$$

and the right hand side

$$b_j = \int_{T_j} g(x)\phi_j(x)\,\mathrm{d}x. \tag{6.8}$$

Solving the linear equation system for $a = (a_1,\dots,a_{n_a})^T$ is the discrete version of solving the partial differential equation (6.2). Therefore, it is possible to view the stiffness matrix A as the discrete representation of the Laplace-Beltrami operator.

However, there is a catch when calculating the matrix A. In Equation (6.7), the evaluation of the following integral is needed.

$$\int_{T_j} \phi_i(x)\phi_j(x)\,\mathrm{d}x \tag{6.9}$$

Note that this is a major difference from the old SU2 finite element solver. There, only integrals over $\nabla_x \phi_i(x)\phi_j(x)$ or similar derivative expressions are calculated.

If the functions ϕ_i are polynomials of degree one, then the integral in Equation (6.9) must be evaluated by a second order quadrature formula. For the implementation of Equation (6.9) in SU2, this means that the quadrature rules for numerical integration must be changed from first to second order. Several new element type classes are introduced to the solver to enable such a higher order integration, as shown in Figure 6.4 in green. The number in the classes name stands for the number of Gauss quadrature points used for numerical integration. The exact second order quadrature formulas for reference integrals on different kinds of elements can be found in the book by Stroud [118, Chapter 8].

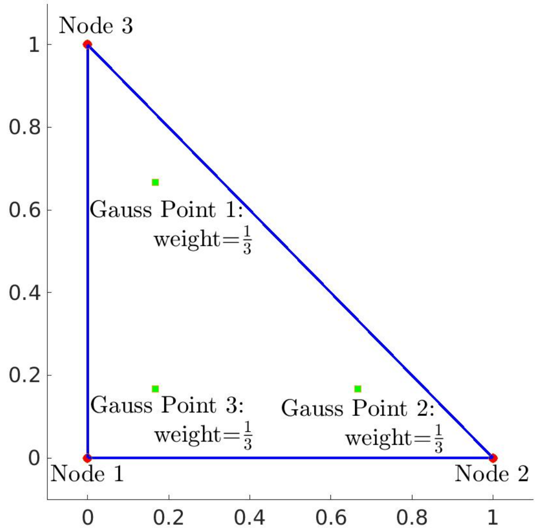


Figure 6.5: Triangular reference element with second order Gauss points.

Unusually, the equation needs a second order quadrature while working with linear elements. In finite elements, one would typically expect higher order quadrature formulas to be used when working with higher order elements.

An explanatory example can be given for a reference triangular element, e.g., the one depicted in Figure 6.5. A second order quadrature rule on the reference triangle uses 3 Gauss points at the coordinates $(\frac{2}{3}, \frac{1}{6})$, $(\frac{1}{6}, \frac{2}{3})$, and $(\frac{1}{6}, \frac{1}{6})$, each with a respective weight of $\frac{1}{3}$, to calculate the exact value for integrals over polynomials of degree 2. Similar formulas can be implemented for other types of elements too.

The next issue is that the finite elements approach described so far has not taken the curvature of the surface into account. The formulation established above is valid if the area Ω in which the PDE is assembled is planar. In Section 5.3, the gradient smoothing procedure was discussed in detail and two approaches were mentioned. First, the Laplace-Beltrami operator might be assembled on the volume grid, i.e., a planar 2D or 3D area depending on the test case dimension. Second, it is also possible to formulate the method on the design surface. This approach can have advantages, like a better approximation of the shape Hessian, but it means that the domain for the PDE becomes a curved manifold embedded in a higher-dimensional space. For example, one might imagine the surface of a wing to visualize this fact. Clearly, this is a 2D manifold embedded inside a 3D geometry and since the surface mesh is stored as a subset of the volume mesh, this means that the node coordinates are also 3D points.

For finite element methods, different approaches are available for dealing with the change in dimension when the triangulation elements are embedded in a space with one dimension more. The base solver presented above uses linear finite elements and since the individual elements shall remain planar objects, the use of more complex elements is ruled out. Instead, the change in dimensions is accounted for in the transformation to the reference element, where such a mapping requires adaptation in the integral transformation [99]. Without loss of generality, the explanation is done for triangular elements on an embedded surface. It can be extended to other types of elements as well and it reduces naturally in the case of a 1D boundary curve embedded in a 2D plane.

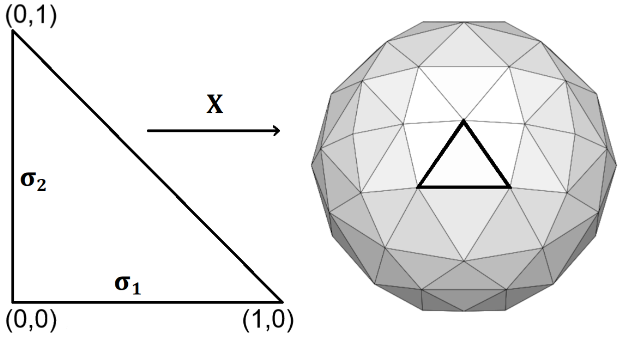


Figure 6.6: Mapping from reference to surface element.

When evaluating an integral like $\int_{T_j} \phi_i(x)\phi_j(x)\,\mathrm{d}x$, it is not directly calculated on the triangle T_j associated with the hat function ϕ_j. Instead, a coordinate transformation is performed and the numerical integration formula is evaluated on the reference triangle. In Figure 6.6, the mapping is visualized. The local coordinate transformation

$$X : (\sigma_1, \sigma_2) \to (X_1, X_2, X_3) \tag{6.10}$$

has the property of changing the dimension of the problem. This must be taken into account, and so a natural generalization of the transformation formula for integrals has to be applied. From the basic analysis, the reader should be familiar with the absolute value of the Jacobian determinant $|\det(J)|$, appearing in the transformation formula for integrals. Obviously, this term can not be applied directly since a closer look at the Jacobian of the coordinate transformation shows that it is not a square matrix.

$$J = \begin{bmatrix} \frac{\partial}{\partial\sigma_1}X_1 & \frac{\partial}{\partial\sigma_2}X_1 \\ \frac{\partial}{\partial\sigma_1}X_2 & \frac{\partial}{\partial\sigma_2}X_2 \\ \frac{\partial}{\partial\sigma_1}X_3 & \frac{\partial}{\partial\sigma_2}X_3 \end{bmatrix} \tag{6.11}$$

When incorporating a dimension change the transformation formula for integrals changes too. The absolute value of the Jacobian determinant $|\det(J)|$ is replaced by the more general Gram determinant $\sqrt{\det(J^T J)}$. The functions and there respective derivatives must be adapted as well. Suppose that a function $\phi(x)$ is defined on T_j, then the pullback of that function on the reference triangle T is defined as $\Phi(\sigma) := \phi(X(\sigma))$. In addition, the gradients are linked using the Moore–Penrose inverse $J^\dagger := (J^T J)^{-1} J^T$, with

$$\nabla_x \phi(x) = (J^\dagger)^T \nabla_\sigma \Phi(\sigma). \tag{6.12}$$

In total, the integral transformation for the stiffness matrix entries A_{ij} is given by,

$$\begin{aligned} A_{ij} &= \varepsilon_1 \int_{T_j} \phi_i(x)\phi_j(x)\,\mathrm{d}x - \varepsilon_2 \int_{T_j} \langle \nabla_x \phi_i(x), \nabla_x \phi_j(x) \rangle_2 \,\mathrm{d}x = \\ &= \varepsilon_1 \int_T \Phi_i(\sigma)\Phi_j(\sigma)\sqrt{\det(J^T J)}\,\mathrm{d}\sigma - \\ &\quad \varepsilon_2 \int_{T_j} \langle (J^\dagger)^T \nabla_\sigma \Phi_i(\sigma), (J^\dagger)^T \nabla_\sigma \Phi_j(\sigma) \rangle_2 \sqrt{\det(J^T J)}\,\mathrm{d}\sigma. \end{aligned} \tag{6.13}$$

The implementation of this transformation requires extending the corresponding classes, CElement and CGaussVariable, in SU2 to deal with such changes in coordinates. The new functions allow the user to hand $(n+1)$-dimensional vectors for the mesh node coordinates of an n-dimensional element to the classes. They also provide an implementation to transform the gradients on the element, by the Moore-Penrose inverse, and they can evaluate the integrals in Equation (6.13).

In combination, the extensions to the linear finite element solver, described in this subsection, enable the new gradient smoothing solver to assemble a discrete representation of the Laplace-Beltrami operator with finite elements on the design surface or in the volume.

AD evaluation of the parameterization

After providing an extended finite element solver, to calculate a discrete representation of the parameterized Laplace-Beltrami operator, the next task in the implementation of the gradient smoothing solver class is to provide an evaluation routine for the Jacobian of the parameterization $\mathrm{D}_p M(p)$. Instead of evaluating and storing the whole matrix, some observations on the dimensions and structure of Equation (6.1) can help to determine the most efficient solution.

1. For the optimization algorithm, one needs the solution of a linear system of equations with the matrix $B := \mathrm{D}_p M(p)^T A \mathrm{D}_p M(p)$, where A is the stiffness matrix seen previously. The individual components of this matrix product are never explicitly used on their own.
2. The stiffness matrix $A \in \mathbb{R}^{n_x \times n_x}$ has the same dimensions as the vector of discrete mesh coordinates x. The Jacobian $\mathrm{D}_p M(p) \in \mathbb{R}^{n_x \times n_p}$ is of size n_x in one dimension, while being of size n_p, i.e., the size of p, in the other dimension. It is therefore a highly non-square matrix.
3. For aerodynamic shape optimization, the number of design parameters is usually much smaller than the number of mesh points, i.e., $n_p \ll n_x$. This implies that B is a relatively small matrix composed of a product of three relatively large matrices.

These three points demonstrate the inefficiency of evaluating the matrix multiplication directly. Instead, one should adopt the approach to only compute products of B with arbitrary vectors. This allows the user to either run such a routine n_p times to get the full matrix B or hand the routine to an iterative linear equation solver.

Three consecutive matrix vector products must be evaluated to implement such a function. One with the Jacobian $\mathrm{D}_p M(p)$, the stiffness matrix A, and the transposed Jacobian $\mathrm{D}_p M(p)^T$ each. Section 3.3 presented the forward and reverse mode of AD and explained how to calculate arbitrary matrix vector products with a Jacobian matrix in Equations (3.39) and (3.41) respectively. With this in mind, it is easy to see how applying a powerful AD tool can be genuinely beneficial here.

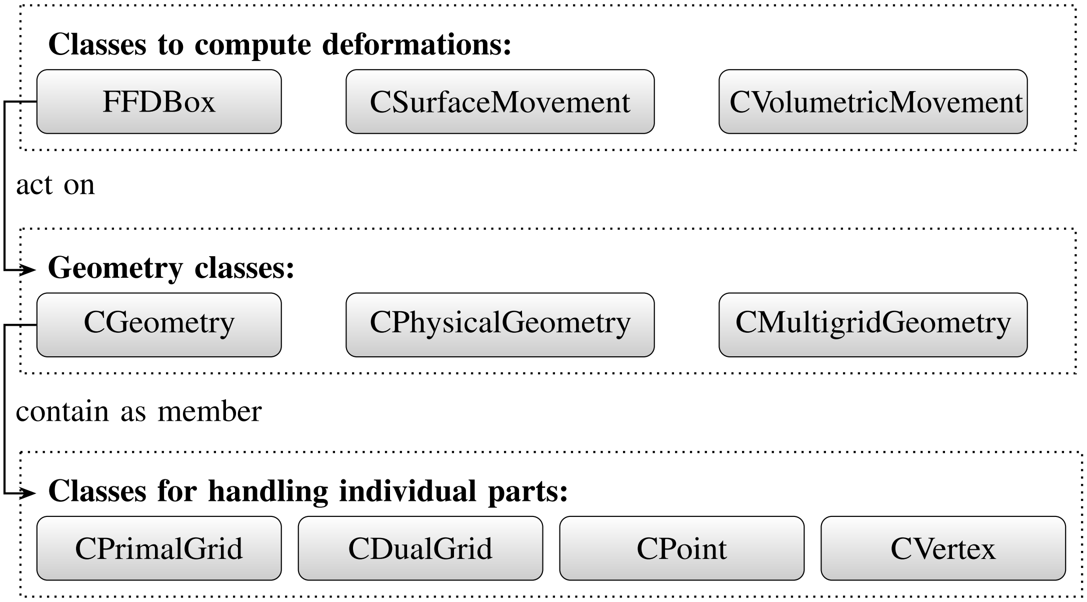


Figure 6.7: Different classes for deformation in SU2.

Some derivatives of the parameterization are already available within the SU2 adjoint framework, namely the right hand side of Equation (6.1), i.e., $-\mathrm{D}_p\tilde{L}(p) = -\mathrm{D}_x L(u,\lambda,x)\mathrm{D}_p M(p)$, can be computed. For this, the structure of the parameterization has to be taken into account, with the relevant class hierarchy shown in Figure 6.7. Instances of the ‘CVolumetricMovement’ and ‘CSurfaceMovement’ classes will act on the geometry classes, which store the individual grid cells, to calculate a deformation. There are two implementations to evaluate a derivative of the mesh deformation process. Both are part of the adjoint solution chain and can be found within the SU2_DOT and SU2_DOT_AD executables. The first one is based on finite differences, namely for each design variable the SU2_DOT executable calls the deformation routines and computes the difference quotients in the mesh coordinates. The second approach, located in SU2_DOT_AD, uses AD and records one evaluation of the deformation $M(p)$ onto a tape. This tape is then initialized with the sensitivities in the mesh coordinates, i.e., $\mathrm{D}_x L(u,\lambda,x)$, as seeding and evaluated.

The second implementation is a good template for the gradient smoothing solver, but some limitations require further code implementation. In this work, several steps are taken to deploy a general, flexible AD-based evaluation of the parameterization Jacobian. Since the CGradientSmoothingSolver class provides an evaluation of the matrix in Equation (6.1), it makes no sense to distribute this across several executables. Instead, everything should exist entirely within the SU2_CFD_AD executable. As a first step, the source code for the evaluation of $\mathrm{D}_p M(p)$ is transferred into SU2_CFD_AD.

To enable the code to accept arbitrary input vectors, in multiple evaluations, and to compute products with both $\mathrm{D}_p M(p)$ and $\mathrm{D}_p M(p)^T$, the process is split into three functions.

First, record one complete evaluation of the mesh deformation. Following the AD logic, the function ‘RecordParameterizationJacobian’ shown in Listing 6.2.1 stores the computational graph of evaluating the parameterization onto a tape. The pseudocode in Algorithm 6.2.1 shows the major steps computed by the function ‘RecordParameterizationJacobian’. Multiple derivative evaluations can be computed once all steps are stored on the AD tape.

```
void CGradientSmoothingSolver::RecordParameterizationJacobian(
    CGeometry *geometry,
    CSurfaceMovement *surface_movement,
    CSysVector<su2double>& registeredCoord,
    CConfig *config);
```

Listing 6.2.1: Function call for recording the parameterization to the tape.

Algorithm 6.2.1 Recording of the parameterization

```
start recording of the AD tape ↻
register input variables $p$
compute $x = M(p)$
register output variables $x$
stop recording
return tape ↻
```

This is done by the function shown in Listing 6.2.2, i.e., ‘ProjectMeshToDV’. This function can use the recorded tape, provided by the AD tool, to calculate arbitrary matrix vector products $\bar{p} = \mathrm{D}_p M(p)^T \bar{x}$, with a seeding $\bar{x}$, by deploying reverse mode AD. The major steps are outlined in Algorithm 6.2.2.

```
void CGradientSmoothingSolver:ProjectMeshToDV(
    CGeometry *geometry,
    CSysVector<su2double>& sensitivity,
    std::vector<su2double>& output,
    CSysVector<su2double>& registeredCoord,
    CConfig *config);
```

Listing 6.2.2: Function call for reverse evaluation of the parameterization tape.

The lacking third function ‘ProjectDVtoMesh’ should calculate the missing matrix vector product $\dot{x} = \mathrm{D}_p M(p)\dot{p}$, for arbitrary seedings $\dot{p}$. Ideally, this would be computed with the forward mode of AD, see Equation (3.39). However, there is a major problem with this based on integrating the AD tool CoDiPack into SU2, where the floating-point type in the source code is exchanged by the typename ‘su2double’, which is replaced at compile time with the specified AD data type. This means that there are separate, compiled executables, each using either regular double (SU2_CFD), the forward mode type codi:RealForward (SU2_CFD_DIRECTDIFF), or the reverse mode type codi::RealReverse (SU2_CFD_AD). Implementing a solver class mixing forward and reverse mode AD inside the SU2_CFD_AD executable would break this structure, require an overhead of copy operations for type conversion, and provide a significant source of error potential. While theoretically possible, it is highly impracticable from a developer’s perspective.

CoDiPack provides a solution for such situations in the form of tape forward evaluation. This feature enables the user to evaluate the AD tape in the same order as it was recorded, thus emulating the forward mode of AD. Applying the tape forward evaluation in the ‘ProjectDVtoMesh’ function

Algorithm 6.2.2 Reverse mode evaluation of the parameterization tape

```
set seeding for the output variables $\bar{x}$
evaluate the tape $\circlearrowleft$
get sensitivities from the input variables $\bar{p}$
reset the tape $\circlearrowleft$
return $\bar{p}$
```

in Listing 6.2.3 allows the CGradientSmoothingSolver to compute the matrix vector product $\dot{x} = \mathrm{D}_p M(p)\dot{p}$, while using the reverse mode floating point type codi::RealReverse. While this advanced feature is a bit slower than pure forward mode evaluation, it still retains a competitive performance, which is especially beneficial since the tape for the parameterization evaluation is already recorded by the function in Listing 6.2.1. It allows the evaluation of arbitrary matrix vector products with $\mathrm{D}_p M(p)$, as shown in Algorithm 6.2.3.

```
void CGradientSmoothingSolver::ProjectDVtoMesh(
     CGeometry *geometry,
     std::vector<su2double>& seeding,
     CSysVector<su2double>& result,
     CSysVector<su2double>& registeredCoord,
     CConfig *config);
```

Listing 6.2.3: Function call for forward evaluation of the parameterization tape.

Algorithm 6.2.3 Forward evaluation of the parameterization tape

```
set seeding for the input variables $\dot{p}$
call forward tape $\circlearrowleft$ evaluation
get sensitivities from the output variables $\dot{x}$
reset the tape $\circlearrowleft$
return $\dot{x}$
```

By combining the three described routines with the calculation of the stiffness matrix by the finite elements approach from above, it is possible to evaluate all terms of Equation (6.1) inside the gradient smoothing solver class. Thus allowing the reinterpretation of gradients by Sobolev smoothing for the optimization.

6.2.2 One Shot Driver

New implementations are necessary to implement the One Shot optimization from Chapter 4, especially the constrained multistep One Shot Algorithm 4.3.1, in the SU2 framework. These have to extend the discrete adjoint capabilities and combine them with the optimization process.
So far, the current SU2 adjoint framework was constructed around the reverse accumulation Algorithm 3.4.3, since it provides advantages in runtime and memory requirements, as discussed in Subsection 3.4. However, this neglects One Shot's key concept of driving flow solutions, adjoints,

and design optimization to convergence simultaneously.
From the mathematical formulation of One Shot, it is apparent to start by implementing a piggyback iteration and then extend it with design updates. The changes to the adjoint executable include changing the overall data flow of the problem. Referring to the class hierarchy of SU2, shown in Figure 6.2, this is best done in the driver class. In this subsection, the newly implemented COneShotSinglezoneDriver driver is introduced and its relationship with the existing CDiscAdjSinglezoneDriver class is discussed.

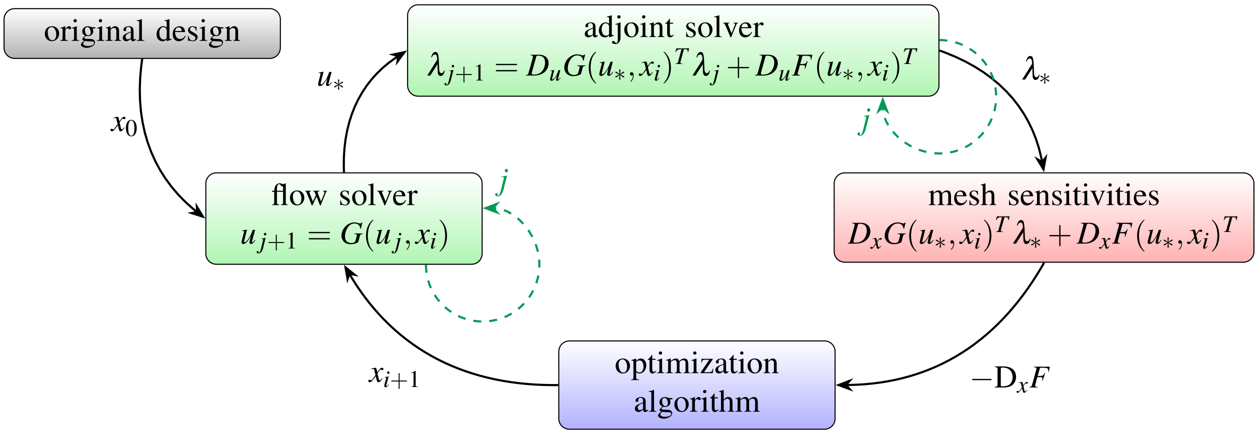


Figure 6.8: Optimization process with reverse accumulation.

To understand how the driver has to be changed to use a piggyback iteration, consider how the old SU2 implementation works. The computational flow during the existing optimization with reverse accumulation is shown in Figure 6.8. Arrows symbolize successive steps and dashed arrows symbolize loops. First, the physical flow problem is solved using a corresponding flow driver, then the adjoint problem is solved by an adjoint driver, iterating the adjoint solver. This is equivalent to the reverse accumulation Algorithm 3.4.3. The introduction of the Sobolev smoothing solver already discussed the role of different executables. In Figure 6.8, the flow solver would be executed with a call to SU2_CFD using a flow driver class and then a call to SU2_CFD_AD would execute the adjoint solver by utilizing the CDiscAdjSinglezoneDriver class. To implement the One Shot optimization process, this is now replaced by a single call to SU2_CFD_AD using a new driver class.
From a mathematical viewpoint, the piggyback iteration in Equation (3.48) does a coupled iteration for the flow and adjoint equations. As driver classes within SU2 are designed to manage different solvers working together, the new One Shot driver has to combine the flow and adjoint solvers. This means a flow solver class, to solve the flow physics, is coupled with an instance of the discrete adjoint solver class, to evaluate the adjoint equation, in each pseudo timestep. It is also beneficial to include an instance of the CGradientSmoothingSolver class from Subsection 6.2.1, since the additional functionality is later needed for the optimizer anyway. See Figure 6.3 for an overview of these different solvers for flow and adjoint within SU2. The new COneShotSinglezoneDriver driver class implements the piggyback Algorithm 3.4.4, by combining these solvers in a coupled iteration. As piggyback uses the current state solution u_i in each iteration, it also changes the order in which the solvers are called and consequently the data flow. The new process is outlined in Figure 6.9.
The new class COneShotSinglezoneDriver is implemented as an overload of the already existing

CDiscAdjSinglezoneDriver class, since the two share much common functionality. AD is applied to compute derivatives for the adjoint, so the AD logic has to take the changed data flow into account. The key points can be summarized as follows.

- For the CDiscAdjSinglezoneDriver, it is only necessary to record the flow iteration onto an AD tape twice. In Algorithm 3.4.3, two matrix vector products with Jacobians appear, $\mathrm{D}_u G(u_*,x)^T\lambda$ and $\mathrm{D}_x G(u_*,x)^T\lambda$, both of which can be stored onto AD tapes. Repeated evaluations of matrix vector products are done by calling the tape multiple times. Note that a single recording of the whole Jacobian $[\mathrm{D}_u,\mathrm{D}_x]\,G(u_*,x)^T$ is also possible, but would require more memory overhead.
  Therefore, the CDiscAdjSinglezoneDriver class does two recordings. First, $\mathrm{D}_u G(u_*,x)^T$ is recorded in a preprocess routine, with the converged conservative state variables u_* as inputs. The function activates the AD recording and stores one step of the flow solver, and possibly turbulence solver, on the AD tape. In the main run routine of CDiscAdjSinglezoneDriver, a pseudo time-stepping loop evaluates this tape and iterates the procedure by setting the tape back, initializing the seeding with the previous adjoint state, and reevaluating the tape. Once this iteration converges to the adjoint solution λ_*, the second recording can be done as a post-process to evaluate $\mathrm{D}_x G(u_*,x)^T\lambda_*$ for the design equation once.

- On the contrary the COneShotSinglezoneDriver has to calculate matrix vector products with different Jacobians $\mathrm{D}_u G(u_j,x)^T\lambda_j$ in each iteration. Notice the change in the flow state $u_* \to u_j$. This implies that the pseudo time-stepping loop must be extended to record one flow solver step and then immediately evaluate the tape. Therefore, each pseudo timestep becomes more expensive, requiring the AD overhead from recording each step anew. Once convergence to an adjoint solution λ_* is reached, or after a prescribed number of piggyback steps J, the time-stepping stops. The remaining task is to evaluate the design equation similar to the post-process of CDiscAdjSinglezoneDriver. Mathematically, this piggyback logic evaluates the sensitivity of the current Lagrangian with respect to the mesh coordinates $\mathrm{D}_x L(u_J,\lambda_J,x)^T$.

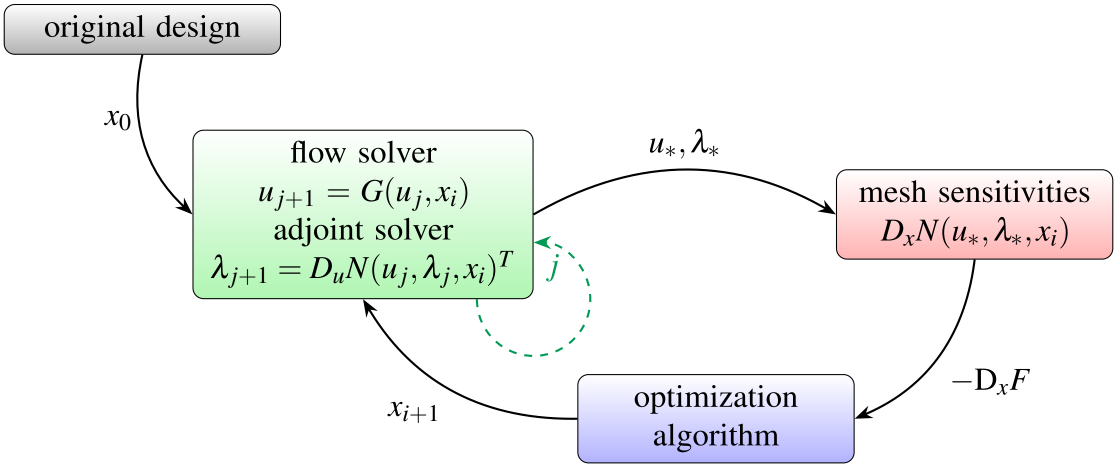


Figure 6.9: Optimization process with piggyback.

It is important to note that this piggyback implementation alone immediately allows for applying unconstrained One Shot methods out of the box. The new driver can compute the flow and adjoint states, the objective function, and the gradient of the Lagrangian in one evaluation. Function values and derivatives can then be used in Algorithm 4.2.2. More detailed optimization strategies are discussed in Section 6.3. Overall, by calling SU2_CFD_AD with the piggyback driver and controlling the number of iterations, a sliding range of methods from classical optimization, with the converged flow and adjoint solutions, over multistep One Shot, up to single-step One Shot is possible.

6.3 Incorporation into the Python Optimizer

In Subsection 6.1.3, the features of FADO that make it an excellent choice for the optimization interface were listed. Now, the reduced SQP Algorithm 3.5.2 and One Shot Algorithm 4.3.1 from previous chapters are reexamined and special attention is given on how to represent them within the framework efficiently. In this thesis, several new C++ solver classes and a new driver class are added to SU2, as discussed in Section 6.2, and the first task is to make these available to an optimization algorithm. To run these simulations, many new configuration options are added and hence they produce new types of output files, which contain information relevant to the optimization algorithm. The additional flexibility of FADO in setting configuration files, via the label replacement mechanism, and handling file I/O is convenient for this task.

For the new implementation, recall the constrained reduced SQP Algorithm 3.5.2. Different programs are used to compute different parts of the procedure when implementing it. In Algorithm 6.3.1, the process is depicted again, with the distribution of individual tasks marked by color. This thesis implements the reduced SQP optimization as a new optimizer available to the FADO framework. The overarching procedure and certain tasks are directly implemented in Python, shown by the blue overlay. They are provided by a series of Python functions and do not require additional calls to external tools. For function and gradient calculations, different CFD executables have to be called through the FADO interfaces. These are the parts depicted by a green overlay and use multiple SU2 executables, SU2_DEF for mesh deformation, SU2_CFD for flow computations, and SU2_CFD_AD for the adjoints. To calculate the design update, the matrix B must be computed by the new solver implementation from Section 6.2 too. Finally, the quadratic subproblem in each step, in the red overlay, is solved using the common quadratic solver package 'cvxopt'[4], a widely used Python package for solving convex optimization problems [122, 7]. The quadratic subproblems in the reduced SQP algorithm are convex since the Sobolev smoothing system matrix is symmetric positive definite by design, so this is a valid choice.

The green parts of the algorithm can be computed by existing SU2 capabilities and the new routines implemented in Subsection 6.2. An external package solves the quadratic problem without needing a new code implementation. Nevertheless, the blue parts in the algorithm must be computed by the new Python optimizer. They can be broadly distributed into three separate tasks.

1. Provide the main iterative optimization loop. This includes driving the optimization process, controlling the data flow and dependencies, and checking for convergence criteria.
2. Compute an adequate step size. In theory, SQP algorithms can use the unaltered solution v of the quadratic problem (3.64) as a design update. However, this is rarely done in practice.

---

[4] CVXOPT: Python Software for Convex Optimization, https://cvxopt.org

Algorithm 6.3.1 Distribution of the reduced SQP algorithm

```
input Initial design variables $p_0$, set iteration counter $i = 0$
while err > tol do
    $x_i = M(p_i)$                                              ▷ compute a deformed mesh
    for $j = 0, 1, \dots, J_u$ do                               ▷ solve the flow equation
        $u_{j+1} = G(u_j, x_i)$
    $u_i = u_{J_u};\ \ y = F(u_i, x_i)$
    for $j = 0, 1, \dots, J_\lambda$ do                         ▷ solve the adjoint equations
        $\lambda_j = \mathrm{D}_u G(u_i, x_i)^T \lambda_j + \mathrm{D}_u F(u_i, x_i)^T$
    $\lambda_i = \lambda_{J_\lambda}$
    for $k = 0, 1, \dots, n_E$ do
        for $j = 0, 1, \dots, J_k$ do
            $\lambda_j^{E_k} = \mathrm{D}_u G(u_i, x_i)^T \lambda_j^{E_k} + \mathrm{D}_u F(u_i, x_i)^T$
        $\lambda_i^{E_k} = \lambda_{J_k}^{E_k}$
    for $l = 0, 1, \dots, n_C$ do
        for $j = 0, 1, \dots, J_l$ do
            $\lambda_j^{C_l} = \mathrm{D}_u G(u_i, x_i)^T \lambda_j^{C_l} + \mathrm{D}_u C_l(u_i, x_i)^T$
        $\lambda_i^{C_l} = \lambda_{J_l}^{C_l}$
    $\tilde{D}_p F = \left(\lambda_i^T \mathrm{D}_x G(u_i, x_i) + \mathrm{D}_x F(u_i, x_i)\right) \mathrm{D}_p M(p_i)$    ▷ evaluate the design equations
    for $k = 0, 1, \dots, n_E$ do
        $\tilde{D}_p E_k = \left((\lambda_i^{E_k})^T \mathrm{D}_x G(u_i, x_i) + \mathrm{D}_x E_k(u_i, x_i)\right) \mathrm{D}_p M(p_i)$
    for $l = 0, 1, \dots, n_C$ do
        $\tilde{D}_p C_l = \left((\lambda_i^{C_l})^T \mathrm{D}_x G(u_i, x_i) + \mathrm{D}_x C_l(u_i, x_i)\right) \mathrm{D}_p M(p_i)$
    solve the quadratic problem: $\min_{v \in S_v} \frac{1}{2} v^T B_i v + \tilde{D}_p F v$    ▷ compute design update
                                 $s.t.\ \ \tilde{D}_p E v + E = 0$
                                 $\tilde{D}_p C v + C \geq 0$
    $p_{i+1} = p_i + v$                                         ▷ update the design
    $i = i + 1$
return $p_i$
```

The reasoning behind this is that the quadratic approximation does not take higher order nonlinearity into account and therefore choosing v itself might not lead to good convergence. This is similar to gradient descent methods, not using the full length gradient, but adapting the step size.

An adapted step is calculated using a line search function $l(v)$. The simplest example is just multiplication by some constant factor $\alpha_l < 1$, while more advanced methods might perform backtracking based on the Lagrangian function value or check for descent criteria.

3. Keeping track of intermediate designs, solutions, and the optimization history. Of course, a user wants to keep track of the optimization's progress. For this, it is essential to store data on the intermediate values of functions, constraint values, gradients, and Lagrangian multipliers on the disc.

```
def SQPconstrained(x0, func, f_eqcons, f_ieqcons,
                   fprime, fprime_eqcons, fprime_ieqcons,
                   fdotdot, iter, acc, lsmode,
                   config, xb=None, driver=None):
  ...
  Initialization and Preprocessing
  ...
  # main optimizer loop
  while (err > acc and step <= iter):
    # evaluate the functions
    F = func(p)
    E = f_eqcons(p)
    C = f_ieqcons(p)
    D_F = fprime(p)
    D_E = fprime_eqcons(p)
    D_C = fprime_ieqcons(p)
    # Hessian computation
    H_F = fdotdot(p)
    if config.hybrid_sobolev:
      if (np.size(config.epsilon3) > 1):
        H_F = H_F + config.epsilon3[step]*np.identity(len(p))
      else:
        H_F = H_F + config.epsilon3*np.identity(len(p))
    ...
    Pack variables for the cvxopt quadratic solver
    ...
    # solve the interior quadratic problem
    if np.size(E) > 0:
      sol = cvxopt.solvers.qp(P, q, G, h, A, b)
    else:
      sol = cvxopt.solvers.qp(P, q, G, h)
    ...
    # line search
    delta_p = linesearch(p, delta_p, F, Lagrang, D_F, D_E, D_C,
                         func, f_eqcons, f_ieqcons,
                         lm_eqcons, lm_ieqcons,
                         acc, lsmode, step, config)
```

```
    # update the Lagrangian for linesearch in the next iteration
    L = Lagrangian(p, func, f_eqcons, f_ieqcons,
                        lm_eqcons, lm_ieqcons)
    gradL = D_F + lm_eqcons @ D_E + lm_ieqcons @ D_C
    # update the design
    p = p + delta_p
    err = np.linalg.norm(delta_p, 2)
    ...
    Code for convergence control and output
    ...
    # increase counter at the end of the loop
    step += 1
  return 0
```

Listing 6.3.1: Main implementation of the optimization algorithm

In Listings 6.3.1, some major steps of the optimization implementation are shown. After initialization, the main loop begins by evaluating the necessary aerodynamic functions and derivatives. This is done in the code via function calls to the wrapped flow and adjoint executables through the FADO toolbox. For example, the wrapper named ‘fdotdot’ evaluates the parameterized Laplace-Beltrami operator by calling SU2_CFD_AD. A common interface called ‘ExternalRunWithPreAndPostProcess’ provides the capability to define such external runs. The user can specify parameters for handling file I/O and subdirectories in a Python control script and describe pre- and post-processes specific to the given test case. The additional layer these wrappers provide allows an easy exchange of the SU2 routines. Utilizing this, function and gradient computations with piggyback can be inserted here to then call a One Shot optimization if requested.

Once all values are assembled, they are packed into cvxopt matrix classes, required by the cvxopt package to call its quadratic solver. The solver call itself is executed by the line
‘sol = cvxopt.solvers.qp(P, q, G, h, A, b)’.

This returns an instance of a solution class ‘sol’, packing different information about the quadratic solution. Afterwards, the proposed design step and the computed Lagrangian multipliers can be extracted from ‘sol’. Using those results, a line search algorithm is called to test descent conditions and compute an adequate step length to update the design properly. Finally, the optimizer checks for convergence, outputs the optimization history data, and terminates the process if an endpoint is reached.

This implementation allows for the application of Algorithm 3.5.2 and it can also be readily extended for the One Shot Optimization Algorithm 4.3.1. The Python optimizer itself can stay unchanged and all the user has to do is alter the configuration for the function and adjoint evaluations of a given test case. Instead of calls to SU2_CFD and SU2_CFD_AD, to compute converged flow and adjoint solutions, one has to wrap a call to SU2_CFD_AD and set the configuration file to use the new piggyback driver. The external run will evaluate a prescribed number of piggyback steps and return derivatives of the reduced Lagrangian. They can then be plugged directly into the SQP framework resulting in Algorithm 4.3.1.

Chapter 7

Numerical Results

In this chapter, the design optimization techniques presented in this thesis are applied to relevant, standard test cases. Special focus is given to the new methodology for Sobolev smoothing of parameterized designs and how it compares as a Hessian approximation to other established techniques in different optimization settings. To get a good comparison, two different test cases are examined, both well-known in the CFD community: the NACA 0012 airfoil and the ONERA M6 wing.

The NACA 0012 airfoil has been used extensively in the past and in Section 7.1, the optimization of this airfoil together with an Euler flow is examined. Tests are done using fully converged flow and gradient values in the reduced SQP Algorithm 3.5.1 and using piggyback approximations in the One Shot Algorithm 4.3.1, showing how the new methodology offers excellent performance in both settings. The ONERA M6 wing is optimized in Section 7.2 to conduct a more challenging test. This 3D geometry, together with RANS equations and a turbulence model, better resembles relevant engineering test cases. Both optimization algorithms, reduced SQP and One Shot are compared with the results of other Quasi-Newton methods, demonstrating the good performance of the new techniques in an analysis and design framework. Additionally, the computational costs are measured and set into perspective, showing the excellent competitive performance of the new parameterized Sobolev smoothing algorithm.

The numerical studies presented here and a report on the results have been published in parts in the paper by Dick, Schmidt, and Gauger [33].

7.1 NACA 0012 Test Case

The first test case in this thesis is the famous NACA 0012 airfoil. This geometry has been one of the most widely used reference tests in the CFD community for decades. Since its properties and dynamics are well understood by now, it is a great starting point for the investigations in this thesis and a staple test case to apply the ideas introduced in this work.

The original airfoil description stems from the first half of the 20th century and the efforts of the National Advisory Committee for Aeronautics (NACA), a predecessor organization of NASA, to standardize and wind tunnel test airfoils [60]. The airfoil is symmetric, with the upper and lower surface curve having the same reflected shape. While many different flow conditions have been studied for this airfoil, a steady Euler flow is considered here. The standard conditions are a Mach number of $M = 0.8$, an angle of attack $\alpha = 1.25^\circ$, and standard air. This results in the appearance

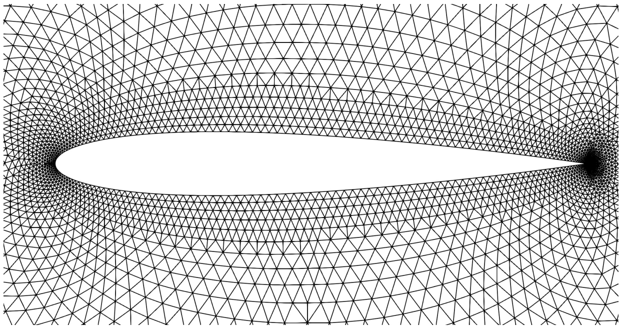


Figure 7.1: The NACA 0012 airfoil and part of the surrounding mesh.

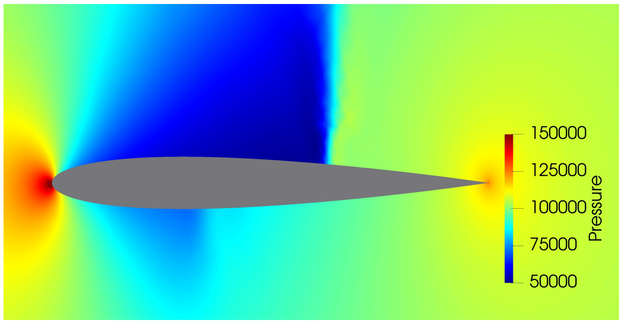


Figure 7.2: Flow field pressure distribution around the NACA 0012 airfoil.

of a typical shock above the airfoil. In this Euler simulation, the drag is induced by this shock. A successful strategy to minimize drag should therefore aim to remove the shock.

The computational mesh for this case is partially depicted in Figure 7.1. It is a two-dimensional, unstructured mesh with triangular cells containing 5233 points and 10216 elements. The mesh also has two boundary markers, one for the airfoil and one for the farfield boundary. For optimization, 38 Hicks-Henne bump functions are utilized as a parameterization of the airfoil. Of these functions, 19 are applied to the upper surface curve and 19 to the lower surface curve to achieve the desired deformation. As described in Section 2.3.3, the amplitudes of these functions are the design parameters, with their peaks distributed equally at relative chord lengths of $0.05, 0.10, \dots, 0.95$.

The Euler simulation is run using a Jameson-Schmidt-Turkel (JST) scheme [63] for the spatial discretization and an implicit Euler time-stepping. The pressure distribution for the flow solution can be seen in Figure 7.2, where the mentioned shock on the upper side is clearly visible.

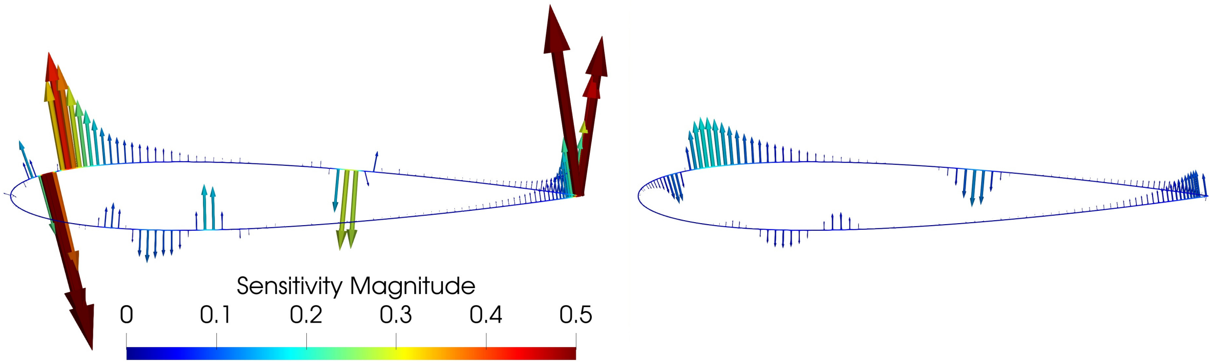


Figure 7.3: Normal surface sensitivities for the NACA 0012 airfoil. The adjoint surface sensitivities are shown on the left and the H^1-gradient on the right.

Before an actual optimization is performed, the finite element solver is validated by a test on the surface mesh sensitivities. Mathematically, the surface sensitivities of the drag coefficient, calculated by the adjoint solver, are reinterpreted according to Corollary 5.2.14 to get the H^1-gradient. In Figure 7.3, the original surface sensitivities $\mathrm{D}_x c_D$ are depicted on the left and the reinterpreted gradient $\nabla_{\mathrm{H}^1} c_D$ on the right. The surface Laplace-Beltrami operator $(\mathrm{I} - \Delta_\Gamma)$ should have typical, strong dissipation properties. Therefore, a characteristic behavior is expected when applied to the surface sensitivities. Here, this becomes visible as high sensitivities are smoothed out, e.g., around the trailing edge, while the identity part preserves the major contour lines of the sensitivity distribution.

7.1.1 Results for Reduced SQP Optimization

After introducing the test case and validating the finite element solver, the combination of the Laplace-Beltrami operator and the Hicks-Henne parameterization is used as the Hessian approximation in a shape optimization test. The aim is to minimize the airfoil's drag coefficient c_D as the objective function. For a convenient notation, the function values are are given in drag counts (1 count $= 10^{-4}$) for the drag coefficient and lift counts (1 count $= 10^{-2}$) for the lift coefficient. An unconstrained formulation for the drag minimization problem would reduce the thickness of the airfoil to zero, therefore additional constraints have to be enforced to ensure a nontrivial solution. In this case, keeping the original lift coefficient of $c_L = 32.69$ lift counts, together with the implicit limitations from the Hicks-Henne parameterization, is enough to obtain a well-defined optimization problem

$$\begin{aligned} \min_{u,x,p}\ & c_D(u,x) \\ s.t.\ & M(p) = x \\ & G(u,x) = u \\ & c_L(u,x) = 32.69. \end{aligned} \tag{7.1}$$

In this first optimization comparison, the aim is to investigate the influence of gradient preconditioning in the parameterized formulation from Equation (5.59). How do the change in the scalar product and the associated change to the H^1-gradient affect the performance of an optimizer?

To test this, Algorithm 3.5.1 is used with different choices for B. Choosing appropriate values for the smoothing parameters $\varepsilon_1, \varepsilon_2$ in Equation (5.59) can be an involved task and different sources suggest different theoretical values, e.g., [70]. In general, the values are test case dependent and for this work, they were determined by a parameter study to find a good fit for the Hessian in the optimum. With this, B is constructed according to Equation (5.59), with two settings of weights, first $\varepsilon_1 = 1.0, \varepsilon_2 = 0.0625$ and second $\varepsilon_1 = 1.0, \varepsilon_2 = 0.625$. The different ε_2 values are used to investigate further the effect of increased Laplacian smoothing on the optimization process. The finite elements stiffness matrix is assembled on the airfoil's surface, representing an H^1 scalar product there, and projected using the AD differentiated Hicks-Henne parameterization. Overall, this first approach results in a parameter formulation of the Laplace-Beltrami operator as a Hessian approximation. The second algorithm uses classical constrained gradient descent with the adjoint mesh sensitivities for comparison. Mathematically, this is equivalent to setting B to the identity matrix, in place of the Hessian, in the SQP Algorithm 3.5.1. For all three optimization runs, the step sizes are determined by a comparable backtracking heuristic, checking for descent in the optimization problem. It is also worth noting, that in this thesis the lift constraint is explicitly treated as a constraint according to Algorithm 3.5.1, while many other studies will adjust the angle of attack to keep the desired target lift.

Results for the different optimization methods are shown in Figure 7.4. The values for c_D and c_L are plotted throughout the optimization. All methods can effectively reduce the drag coefficient to a comparable degree. The original value of ca. 210 drag counts is reduced down to an optimal drag coefficient of ca. 13 drag counts for all three optimizations. In direct comparison, the parameterized

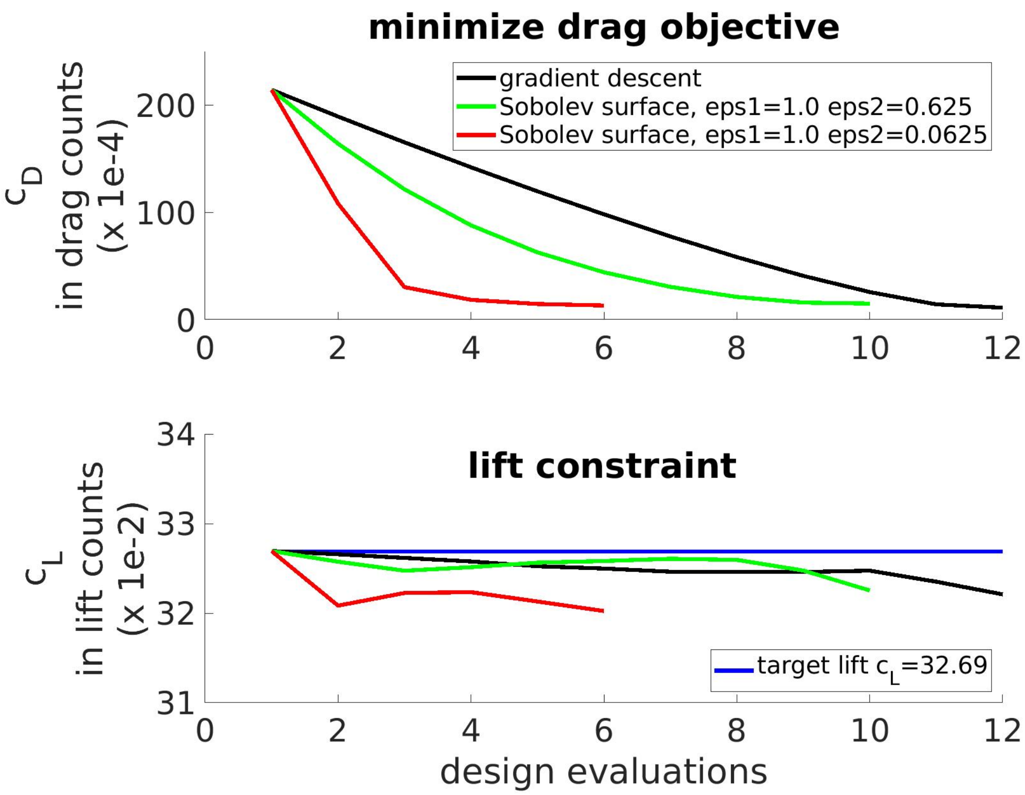


Figure 7.4: Comparison of different shape optimization methods for the NACA 0012 test case. [33, Figure 4]

Sobolev smoothing achieves a considerably faster convergence than gradient descent. At the same time, the algorithms keep the lift constraint within close bounds of 0.6 lift counts from the target lift coefficient. Here, the lift deviation is slightly higher for the Sobolev method. In a direct comparison of the smoothing parameter settings, the first choice of $\varepsilon_1 = 1.0, \varepsilon_2 = 0.0625$ decreases the objective function faster, than the second choice of $\varepsilon_1 = 1.0, \varepsilon_2 = 0.625$. Overall, the preconditioned H^1-gradient yields a faster rate of convergence for the optimization algorithm than traditional gradient descent methods do, despite the parameterization.

The last point in this test is to consider the computed optimized airfoil profiles and the surrounding flow fields. The optimal solution should be shock free and only have spurious drag stemming from numerical stabilization since this is an inviscid test case. Figure 7.5 shows the pressure distribution for the original airfoil (left), the Sobolev smoothing optimized airfoil with $\varepsilon_1 = 1.0, \varepsilon_2 = 0.0625$ (center), and the classical gradient descent optimized airfoil (right). In comparison with the original NACA 0012 profile, the shock above the airfoil is removed, thereby eliminating the shock-induced drag. For the gradient descent algorithm, a small scale remnant of a shock remains, but this is arguably in the range of spurious drag. Therefore, Figures 7.4 and 7.5 show the expected results for this test case, validating the underlying approach.

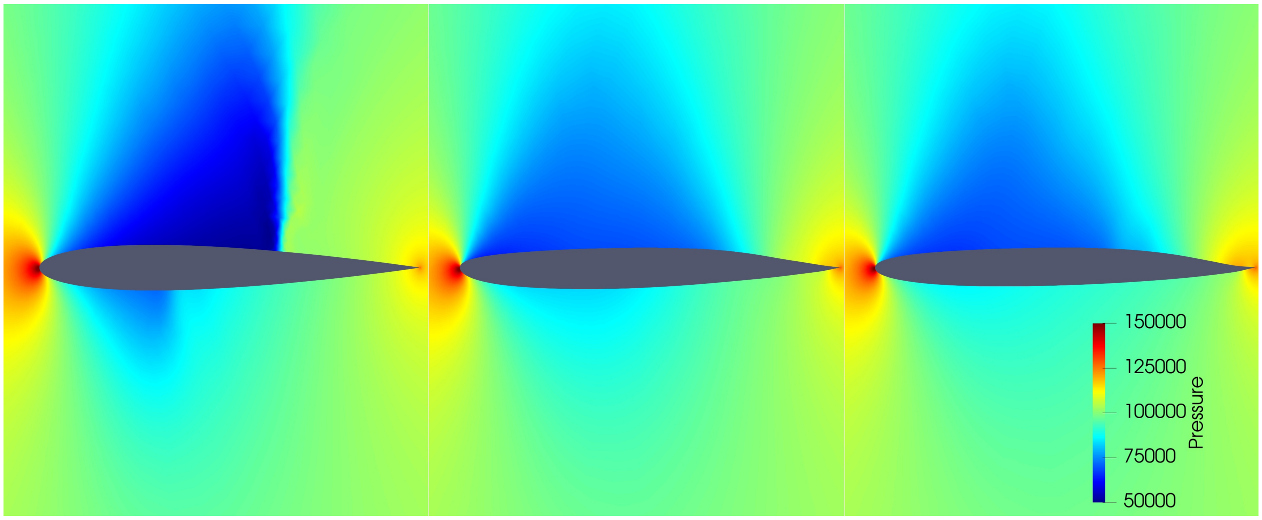


Figure 7.5: Flow field comparison for the original and optimized NACA 0012 airfoil. Starting airfoil (left), result Sobolev smoothing (center), result gradient descent (right).

7.1.2 Results for One Shot Optimization

The previous Subsection 7.1.1 demonstrated good performance of the Sobolev gradient treatment when converged flow and adjoint solutions are used in the process. However, application in a classical reduced SQP framework is not the only focus of this thesis. The Sobolev methodology is combined with a multistep One Shot optimization algorithm as well and therefore Algorithm 4.3.1 is applied to the NACA 0012 test case. Nonetheless, there are some points to consider when running such an optimization, which must be discussed first.

For this test, the Laplace-Beltrami operator is constructed with the setting $\varepsilon_1 = 1.0$ and $\varepsilon_2 = 0.625$ known from the reduced SQP test 7.1.1. The factor $\varepsilon_2 = 0.625$ for the Laplace part is chosen because One Shot can benefit from the smoother deformations associated with it. As a baseline comparison, a gradient descent type One Shot is done using untreated, approximated surface sensitivities computed from the multistep piggyback iteration. Mathematically, this can be achieved by using the identity matrix in place of B in Algorithm 4.3.1 and limiting the step size to compute the design update.

Another important factor for any multistep One Shot optimization is the number of piggyback steps for flow and adjoint iterations. For the NACA 0012 test case at hand, 10 coupled steps for flow and adjoint achieve a good balance between computational cost and accuracy. A more in-depth discussion of this issue will be presented in Subsection 7.2.2.

In Figure 7.6, optimizations with the different settings are plotted. Some observations can be drawn immediately from this picture.

- One Shot optimization can work successfully for both presented algorithms if the parameters are chosen carefully, e.g., the red and magenta lines. If the preconditioner is chosen positive definite enough and the maximum step size for a design update is small enough, then the iteration will ultimately converge. However, this can lead to small steps and subsequently slow convergence of the optimization. Here, the maximum step sizes are deliberately set close to the maximum values for which the One Shot optimization remains stable and does not diverge to test the limits of the presented methodology.

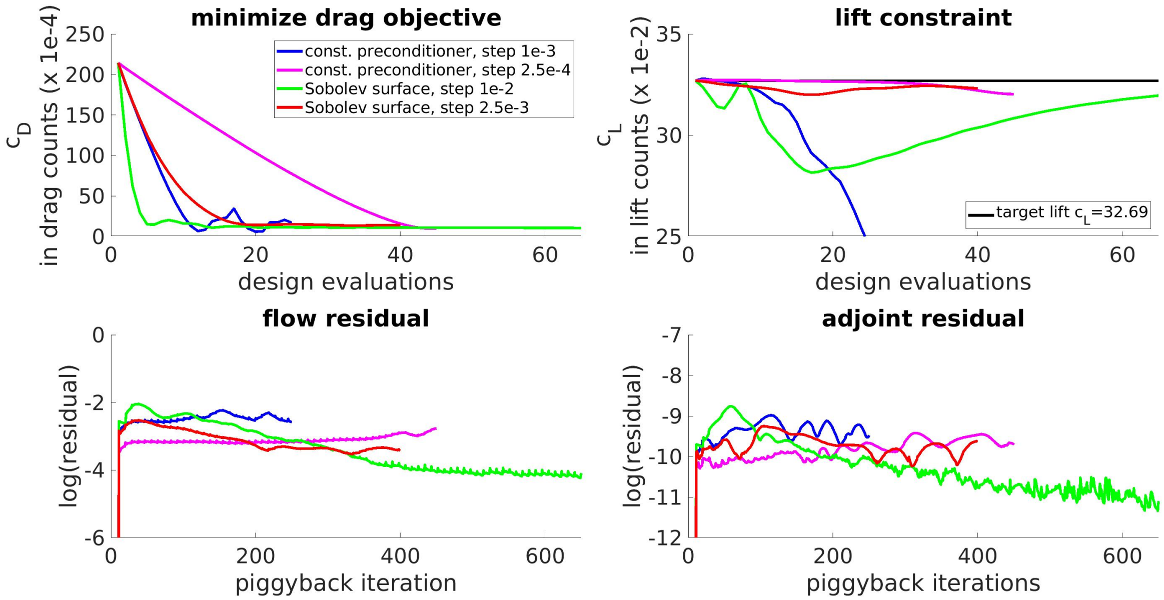


Figure 7.6: Comparison of different One Shot strategies for the NACA 0012 test case.

- Using parameterized Sobolev gradient smoothing in place of the Hessian approximation can result in a fast decrease in the objective function, as shown in green. However, this leads to violations in the lift constraint and additional iterations are necessary to get back to an acceptable violation of ca. 0.6 lift counts. Using a more restrictive maximum design update step size of $2.5e^{-3}$ is shown in red. This results in better adherence to the constraint, but limits the speed with which the objective function decreases. Nonetheless, it results in a net benefit.

- For gradient descent like One Shot, i.e., using $B = \mathrm{I}$ and limiting the step size, additional caution is necessary. A maximum allowed design update of $2.5e^{-4}$, as shown in magenta, leads to a slower but successful optimization. On the contrary, as shown in blue, choosing too large design updates can lead to stability issues and a violation in the lift constraint in the presented case.

- All algorithms depicted here can keep the residuals for flow and adjoint within a reasonable range. There are two factors at work here, the first is that the design updates have to be restricted to avoid divergence in the flow and adjoint solvers. This will happen if the proposed steps v from Algorithm 4.3.1 are chosen without further reduction. On the other hand, the decrease in residual is restricted by technical factors of the solver. Since the mesh coordinates are deformed and the MPI communication buffers are not stored for a restart of the adjoint executable, but initialized by zero again, the residuals will jump up after each design update when the SU2 piggyback method is restarted, see Section 6.2.2. This makes the theoretical convergence to machine precision impossible while the optimization is ongoing. Instead, the flow and adjoint solutions are converged again after the optimization is terminated, to verify the solution. This is not depicted here, but generally does not significantly change the objective function value.

In total, the objective function is reduced by the new One Shot algorithm with parameterized Sobolev smoothing to ca. $c_D = 10.2$ drag counts. By gradient descent, a similar value of ca. $c_D = 9.8$ drag counts is reached. Both values are in the range of spurious drag, as seen in the previous Subsection 7.1.1. In this test, the Sobolev preconditioned One Shot optimizer with $\varepsilon_1 = 1.0$ and $\varepsilon_2 = 0.625$, shown in green and red, helped to stabilize the method. This allowed for larger design updates than pure gradient descent, while keeping the constraints and avoiding divergence of the flow and adjoint solvers.

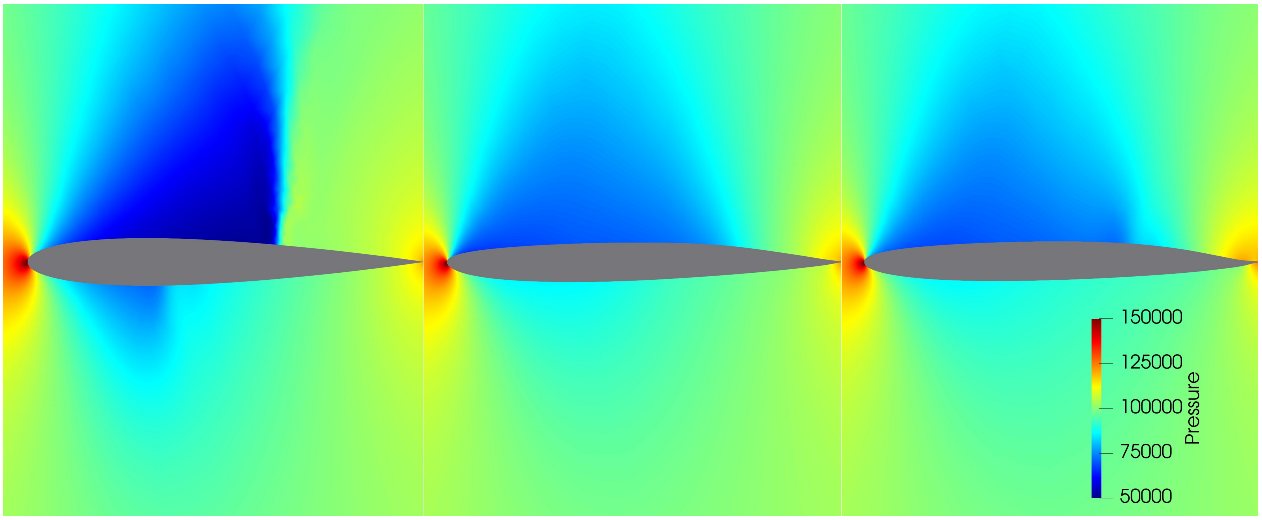


Figure 7.7: Pressure field around optimized NACA 0012 airfoils. Starting airfoil (left), One Shot with Sobolev smoothing (center), One Shot with gradient descent (right).

After investigating the optimization progress itself, it is also worth taking a look at the resulting airfoils. They are shown in Figure 7.7, where the results for Sobolev treated One Shot are in the middle and for the identity as preconditioner are on the right side. As mentioned in Subsection 7.1.1, the shock induced drag is the primary source of drag to be minimized in this test case. As shown in Figure 7.7, the One Shot optimization removes the shock on the upper side efficiently, resulting in flow fields similar to the results for reduced SQP optimization from Figure 7.5.

Overall, the new methodology is successfully applied to a 2D Euler test case, like the NACA 0012 airfoil. In the next section, the method is tested for a more involved 3D RANS test case to see how it performs for larger scale optimization problems.

7.2 ONERA M6 Test Case

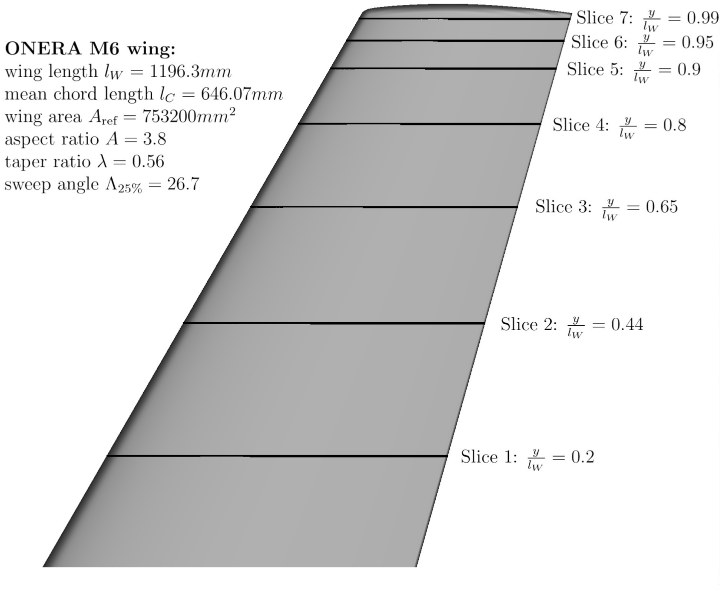


Figure 7.8: Description of the ONERA M6 geometry.

The ONERA M6 wing is a three-dimensional transonic test case that has been one of the most widely used reference cases throughout the CFD community. The original geometry dates back to a swept back wing model originally constructed at the ONERA Aerodynamics Department in 1972. Experimental data from wind tunnel measurements of this wing at different flow conditions were conducted by Schmitt and Charpin[1]. The results were introduced as a reference case for CFD simulations with the publication of a report by the NATO advisory group for aerospace research and development (AGARD) on the construction of a database for CFD simulations in 1979, see [1, Appendix B1]. This report provides an extensive collection of experimental wind tunnel data for different types of airfoil and wing geometries and many of today's reference tests date back to this database.

The transonic flow conditions feature a characteristic double shock on the upper wing surface, named a lambda shock because of its distinct shape. Since this was especially challenging for early CFD simulations to compute, it became a reference case in validating CFD codes. In particular, after Jameson published the results of an Euler simulation for this test case in 1982 [64]. For a more recent description of the geometry, see the work of Mayeur, Dumont, et al. [82, 81].

The wing description is depicted in Figure 7.8. While several different flow conditions were published originally, one standard setting became dominant throughout the aerodynamic community. It uses a Mach number of $M = 0.8395$, though this is rounded to $M = 0.84$ in many applications. The angle of attack is $\alpha = 3.06^\circ$ and the wing has a mean chord length of $l_C = 0.64607m$ and wing area of $A_{ref} = 0.7532m^2$. Overall, this results in a Reynolds number of $\text{Re} = 11.72 \times 10^6$.

Measurement data from wind tunnel tests for the pressure coefficient is given along the seven spanwise cross sections depicted in Figure 7.8. Their locations are given as percentages of the total

---

[1] Pressure Distributions on the ONERA-M6-Wing at Transonic Mach Numbers, in [1, Appendix B1]

length and are listed in Table 7.1. For each section, 34 pressure values at different locations along the cross section are stated in the database.

section	1	2	3	4	5	6	7
$\frac{y}{l_W}$	0.20	0.44	0.65	0.80	0.90	0.96	0.99

Table 7.1: Cross sections for the pressure measurements in the ONERA M6 wind tunnel database.

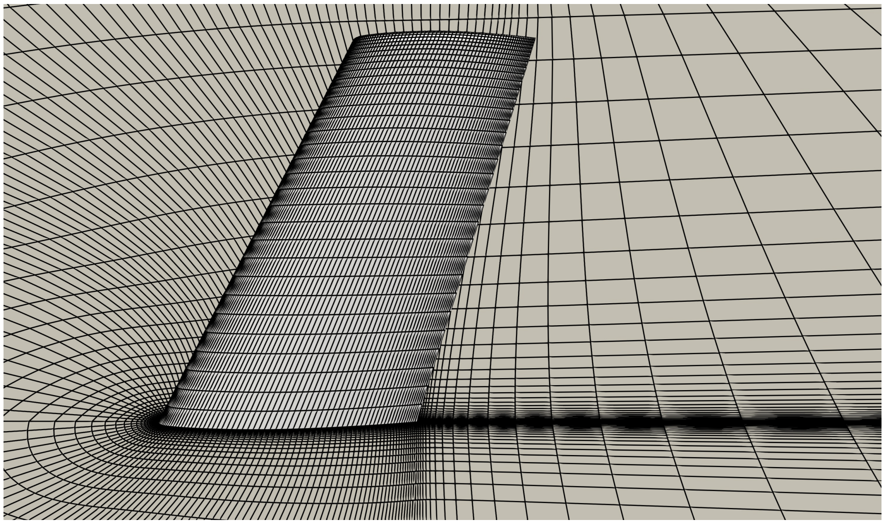


Figure 7.9: The ONERA M6 wing and part of the surrounding mesh, used in the CFD simulation for this test case.

Flow computations on the ONERA M6 wing are performed using a structured C-type computational mesh. Since SU2 is an unstructured code, the structured mesh is stored in the native SU2 unstructured format[2]. The mesh consists of 306577 points and 294912 hexahedral elements describing the shape and has a logarithmic progression of the cell thickness towards the wing surface for boundary layer resolution. For a visual impression, see Figure 7.9. It features three distinct boundaries with different boundary conditions, one Navier-Stokes boundary for the wing surface, one symmetry boundary on the wall, and one freestream boundary in the farfield.

The design is parameterized by an FFD box with 10 cells in the x-direction, 8 cells in the y-direction, and 1 cell in the z-direction. The layout of this box around the wing is shown in Figure 7.10. Only deformations of the control points in the z-direction are considered for shape updates to keep the wing's overall length and chord span constant. This results in a total of 198 design parameters. As discussed in Subsection 2.3.3, this FFD box is used to deform the surface nodes and the volume mesh is then adjusted using a linear elasticity approach based on the wall distance.

The flow simulation is performed using RANS equations, as described in Equation (2.25). Two turbulence models, an SA and an SST model, are used for validation. Although, only the SST model will be used for the shape optimization tests. As finite volume solver a Jameson-Schmidt-Turkel (JST) scheme is deployed, with artificial dissipation coefficients $d^{(2)} = \frac{1}{2}$ and $d^{(4)} = \frac{1}{128}$. For time integration, an implicit Euler scheme is used, where the time steps are chosen by an adaptive

---

[2] See the SU2 documentation, `https://su2code.github.io/docs_v7/Mesh-File/`

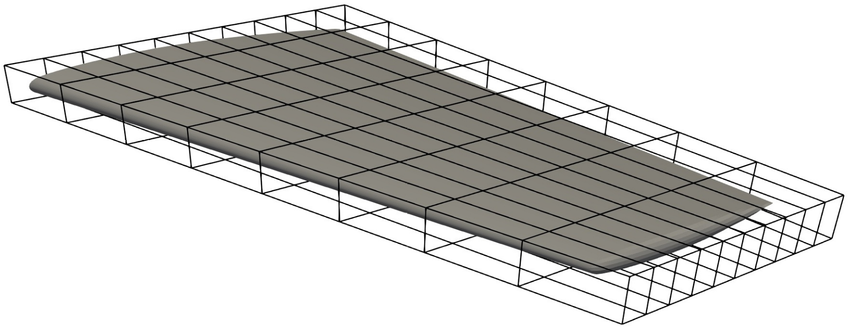


Figure 7.10: FFD box parameterization of the ONERA M6 wing.

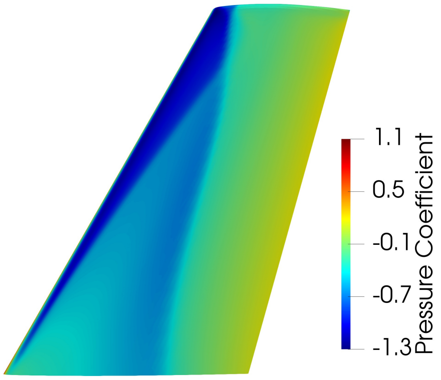


Figure 7.11: Pressure coefficient on the upper side of the ONERA M6 wing showing the typical lambda shock.

heuristic, keeping a maximal CFL number of ca. 20.

Before an optimization takes place, the mesh and test case are validated against the available physical pressure data from wind tunnel testing, as well as checking the accordance of the calculated drag and lift coefficients with other comparable simulation results. Figure 7.11 shows the pressure coefficient on the wing surface. The depicted merging double shock system is typical for the ONERA M6 test case. In Figure 7.12, the simulated pressure distribution at different cross sections is plotted for two different turbulence models against the experimental data.

The CFD simulation accurately resolves the shock position on the upper wing surface. In particular, the results of RANS computations with the SST and SA turbulence models are in close agreement with each other. For the cross sections at 20%, 44%, 65%, and 90%, the shock positions are in accordance with the available experimental data. This result is consistent with the findings of other studies on this test case done by NASA [100, 115]. Nonetheless, there are two sections where the flow solver struggles to resolve the physical solution exactly. The first is at the merging of the double shock system at 80% of the length and the second is on the wingtip at 99%. Here, a wingtip vortex appears, a situation that is notoriously complex to predict. The observed inaccuracies are due to limitations in the turbulence models used. The SST model can not resolve the behavior for this test case exactly, hence a more involved Reynolds stress model would be necessary. More details on this issue and a detailed explanation can be found in the work of Jakirlić, Eisfeld, et al. [61].

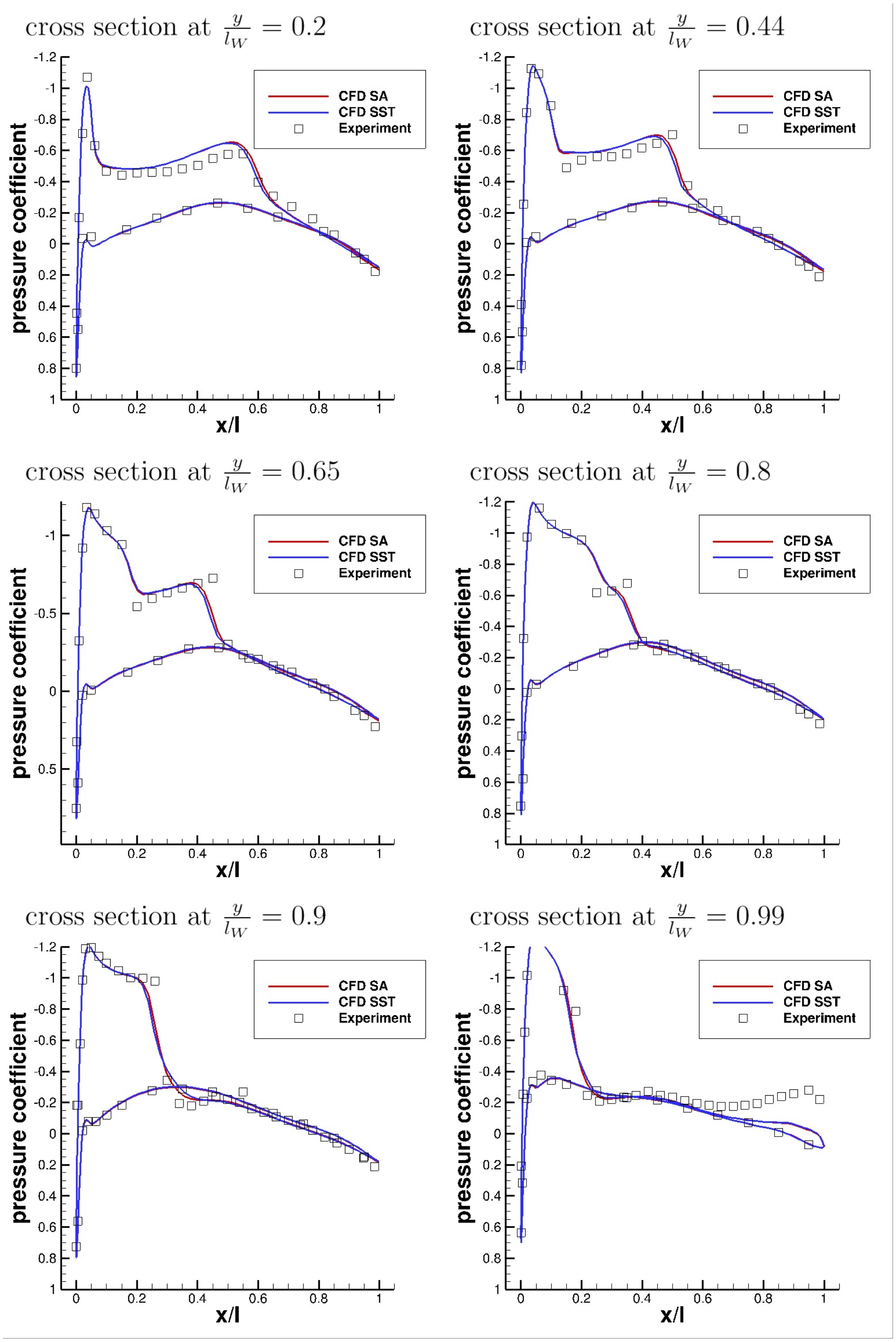


Figure 7.12: Validation of the ONERA M6 CFD simulation results against the experimental pressure values at different cross sections. [33, Figure 8]

While there are experimental data for the pressure coefficients, no precise experimental values for the drag and lift coefficients are available. However, the results from the simulation used in this thesis are compared to other simulation values from relevant reference papers in Table 7.2. From this, one can conclude that the predicted lift seems to be fine, while the predicted drag coefficient is at the lower end of the spectrum.

	own result	Rumsey [100]	Araya [9]	Nielsen & Anderson [89]
c_D (in drag counts)	143.9	172.3	189	168
c_L (in lift counts)	24.99	26.6	25.3	25.3

Table 7.2: CFD computed drag and lift values for the ONERA M6 wing.

Overall, one can conclude that this is a valid test case for optimization purposes. Especially, if the emphasis is on developing new optimization techniques.

7.2.1 Results for Reduced SQP Optimization

To begin with the numerical experiments for the ONERA M6 wing, the parameterized Sobolev gradient reinterpretation is used as a Hessian approximation in a reduced SQP algorithm as the first test. For this, the exact converged flow solutions and gradients from the consistent converged adjoint solutions are used. The optimization is run using the reduced SQP implementation for Algorithm 3.5.2 within SU2 and the FADO framework, as described in Section 6.3.

For the Hessian approximation, the hybrid Laplace-Beltrami operator from Equation (5.75) is used. Following the discussion in Section 5.4, the operator is assembled as a linear combination of three different terms. The respective weight values $\varepsilon_1, \varepsilon_2, \varepsilon_3$ are determined as the result of a parameter study, conducted to fit the hybrid Laplace-Beltrami operator to the Hessian matrix in the optimum. To classify the performance of the reduced SQP optimizer, it is compared with other optimization algorithms. Two algorithms are considered for the comparison, representing different classes of optimization methods.

1. The first comparative method is a projected gradient descent algorithm. It is well known from nonlinear optimization that insertion of the unmodified identity matrix into the SQP framework yields an optimization equivalent to classical gradient descent abiding constraints. This approach has already been used in Section 7.1. The method provides a baseline for performance, being a first order method. As comparable step size control and update procedures are used, any performance differences result from an alternative Hessian approximation resulting in a different search direction. Thus the difference in optimization when using parameterized Sobolev smoothing can be measured.

2. The second algorithm used for comparison is the SLSQP algorithm, particularly the implementation by Kraft [69]. This is the standard SQP optimizer included in the Python SciPy 'optimize' module. It uses an iterative BFGS update to approximate the Hessian and computes the step size using a merit function based upon the L^1-norm of the constraints. As a widespread superlinear SQP method, it is a representative example of a standard Quasi-Newton method with iterative Hessian approximations, which are typical within nonlinear numerical optimization.

The optimization problem is a minimization of the drag coefficient c_D under mixed equality and inequality constraints. The optimizer should keep the lift coefficient at a constant value of $c_L =$ 24.99 lift counts, i.e., the lift value of the original ONERA M6 wing. At the same time, a set of geometric inequality constraints is imposed as well. To prevent the wing profile from getting too thin, there are minimum thickness values t_{wing} prescribed along five different cross sections at different relative lengths shown in Table 7.3. Such thickness constraints could be imposed implicitly by the parameterization. Here, they are instead given explicitly to the optimizer to demonstrate the capability to handle additional geometry restrictions.

position $\frac{y}{l_W}$	0.0	0.2	0.4	0.6	0.8
minimum thickness	0.077	0.072	0.066	0.060	0.054

Table 7.3: Thickness constraint positions and values for the optimization problem.

Thus the complete mathematical minimization problem takes the form shown in Equation (7.2).

$$\begin{aligned} &\min_{u,x,p} c_D(u,x) \\ &s.t.\ \ M(p) = x \\ &\qquad G(u,x) = u \\ &\qquad c_L(u,x) = 24.99 \\ &\qquad t_{\text{wing}}(u,x)|_{\frac{y}{l_W}=0.0} \geq 0.77 \\ &\qquad t_{\text{wing}}(u,x)|_{\frac{y}{l_W}=0.2} \geq 0.72 \\ &\qquad t_{\text{wing}}(u,x)|_{\frac{y}{l_W}=0.4} \geq 0.66 \\ &\qquad t_{\text{wing}}(u,x)|_{\frac{y}{l_W}=0.6} \geq 0.60 \\ &\qquad t_{\text{wing}}(u,x)|_{\frac{y}{l_W}=0.8} \geq 0.54 \end{aligned} \tag{7.2}$$

This problem is solved using the reduced SQP algorithm for mixed constraints 3.5.2. Two different settings are used for the Hessian approximation, representing the different ways the Sobolev gradient reinterpretation can be implemented.

1. The first setting uses Sobolev smoothing on the design surface. This means that the discrete finite elements representation of the Laplace-Beltrami operator is assembled on the surface mesh cells forming the wing. It is then projected using the derivatives of the FFD box parameterization. The weights are set to $\varepsilon_1 = 56.9$, $\varepsilon_2 = 0.9$, and $\varepsilon_3 = 0.1$. These values were determined by a parameter study to find the best fit for the positive definite part of the Hessian matrix in the optimum.

2. The second setting assembles the finite element discretization of the Laplace-Beltrami operator in the whole volume mesh. Mathematically, this means approximating the Hessian in the flow domain instead of on the surface. The derivatives of the parameterization are calculated via the chain rule from the FFD box parameterization and the linear elasticity based mesh movement, see Equation (5.65). The weights for combining the different terms in this case were $\varepsilon_1 = 0.0$, $\varepsilon_2 = 7.2$, and $\varepsilon_3 = 0.1$.

When comparing the performance of different optimization methods, it is important to point out the role of line search procedures and step size control. Obviously, this will significantly influence the overall performance of different algorithms. For more complex methods, like SLSQP, one has to distinguish between the main steps of the optimization algorithm and the line search steps. In the main step, all aerodynamic values of the current design are evaluated, including flow and adjoint solutions, parameterization, and the reduced gradients. These values are handed to the line search, which evaluates the functions at different step lengths. For each proposed step length, the mesh deformation and a flow simulation are calculated to test for sufficient descent. If this is not achieved, then additional step lengths are tested.

The SLSQP optimizer uses an L^1 style merit function to determine a good step length, see [69, Section 2.2] for details. The reduced SQP implementation offers several different methods. Starting with a simple scale heuristic that will scale down the step length if the norm is bigger than a given threshold. More advanced settings include a backtracking strategy based on descent conditions. For this test, the proposed initial step size is reduced if the objective function does not show sufficient descent.

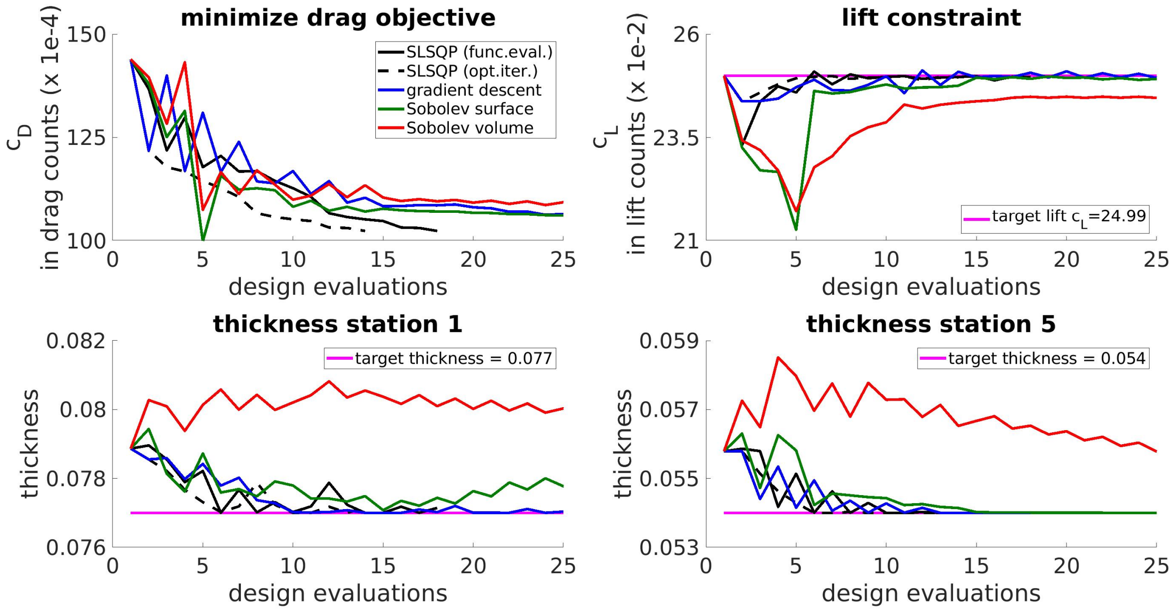


Figure 7.13: Comparison of different optimization algorithms, with converged flow and adjoint solutions, for the ONERA M6 test case.

	RSQP surface	RSQP volume	SLSQP	grad. desc.
c_D (in drag counts)	105.3	108.1	102.4	105.6
relative reduction	26.8%	24.89%	28.85%	26.6%

Table 7.4: Improvement in the c_D value for different optimization strategies.

A comparison between the performance of the different optimization algorithms can be seen in Figure 7.13. The relevant, resulting drag values are shown in Table 7.4. For analysis, the two smoothing approaches on the surface, as drawn in green, and in the volume mesh, as drawn in red, are compared first. Both offer a considerable overall reduction of the objective function, indicating that both Laplace-Beltrami approximations can be useful as Hessian approximations. The original drag coefficient of $c_D = 143.9$ drag counts is reduced to $c_D = 105.3$ drag counts for the surface smoothing and $c_D = 108.1$ drag counts for the volume smoothing procedure. This represents an higher improvement of ca. 26.8% for the surface setting compared to 24.89% for the volume setting. Regarding adherence to the constraints, both methods keep the constraints up to a certain degree, although the volume approach shows a variation of 0.56 lift counts in the end. In total, the surface smoothing approach seems to be the superior choice for this test case.

The next question is why there are observable oscillations in the drag and lift values when using the reduced SQP method? As SQP methods generally only guarantee convergence to a feasible solution, they are not strictly keeping the constraints at intermediate designs. Instead, they aim at optimizing the Lagrangian, which incorporates the constraints weighted by their current Lagrange multipliers. This leads to a behavior where the optimization process violates the equality constraint at intermediate points and then converges back towards a feasible solution, as can be observed for the reduced SQP algorithms in Figure 7.13. See Gherman [45] for more details on the properties of reduced SQP methods.

Next, the gradient descent algorithm shown in blue is used for comparison. Set against surface smoothing, one can observe slower optimization progress, taking longer to reduce the objective function. At the same time, the constraints are held within a smaller range during the optimization and after 25 steps, the algorithm reaches a reduction of the drag value by 26.6%, to a value of $c_D = 105.6$ drag counts. In total, Sobolev smoothing on the surface converges faster than the gradient descent method, while both reach comparable optima and both adhere well to the constraints.

For a final comparison, the SLSQP method can be seen in the same Figure 7.13, drawn in black. The figure distinguishes between the optimization plotted over the major optimizer steps, shown by the dashed black line, and plotted over the number of tested designs including line search steps, shown by the solid black line. The method shows a fast rate of convergence towards an optimal solution. It reduces the drag to $c_D = 102.4$ drag counts, which is a reduction by 28.85%. At the same time, it can be observed that the method keeps the equality constraint very well after a couple of iterations. This can be credited to two factors. First, the line search based on a merit function accepts only steps keeping the constraint violation minimal. Second is the fact that the iterative BFGS Hessian updates work well in the present situation. A further point worth noting is that the progress is slowed down considerably when counting all design evaluations, including line searches, instead of just the major optimization steps. Overall, the SLSQP method has the fastest convergence and reaches the lowest local minimum for the given setting. However, this will change when using the One Shot approach, as will be seen in Subsection 7.2.2.

Naturally, not only the resulting aerodynamic coefficients are interesting, but the resulting designs and their flow fields are relevant as well. Whether or not the lambda shock on the upper wing surface has been removed is especially important since the shock induced drag is one of the major contributors to the overall drag value. The relevant pressure coefficient distribution on the surface is depicted in the two Figures 7.14 and 7.15. The full-sized lambda shock is clearly visible in the original flow field. In contrast to this, the optimized profiles show a smooth pressure distribution on the surface and have a highly reduced shock in their flow fields. In particular, the results for Sobolev smoothing on the surface and gradient descent are very similar. There is an observable difference for the volume Sobolev smoothed SQP optimizer, with some of the shock remaining on the leading edge. This is in line with this method showing a slightly lower performance for the given test case,

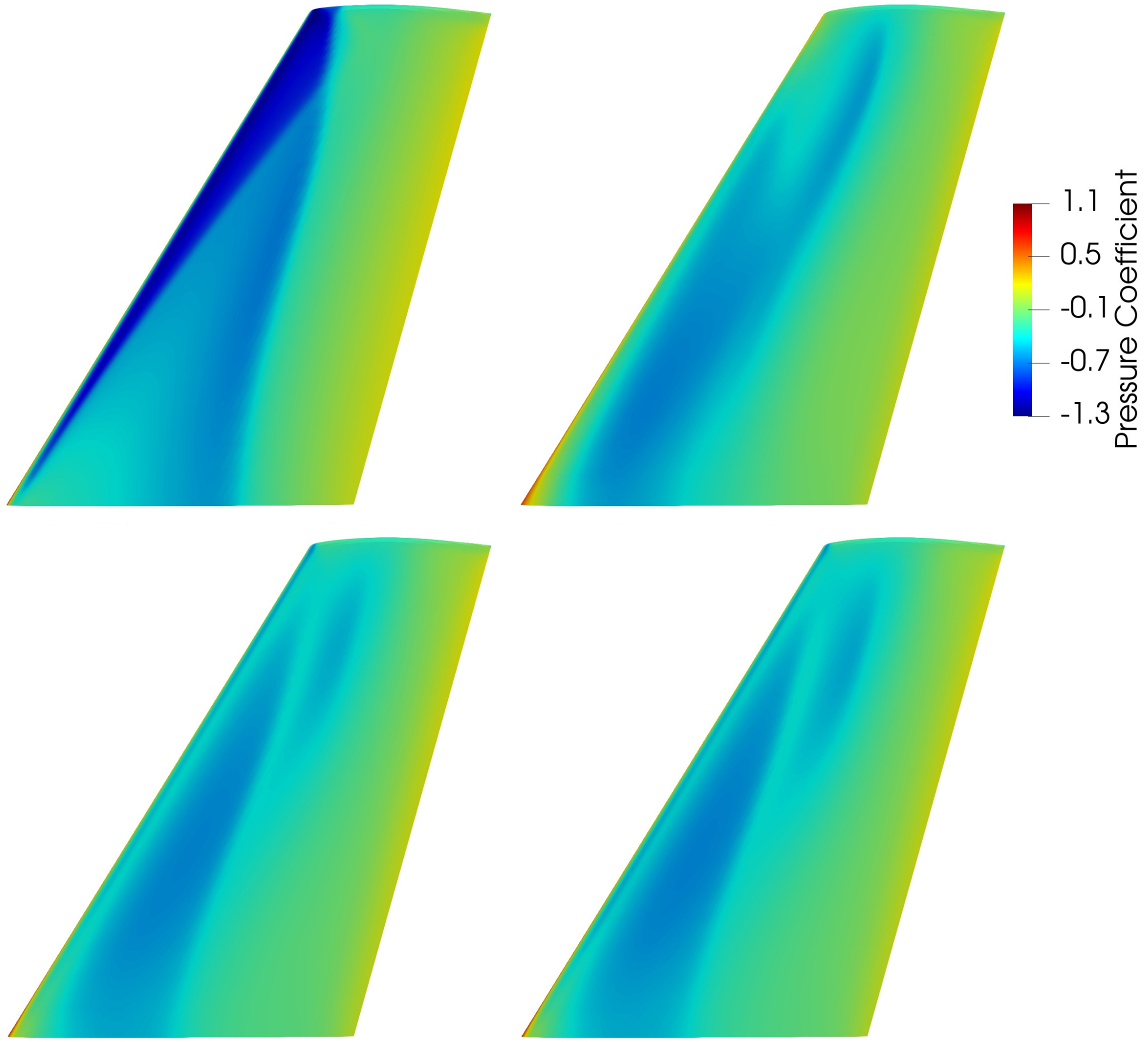


Figure 7.14: Pressure coefficient on the wing surface in comparison to the original wing. Starting wing (upper left), result SLSQP (upper right), result gradient descent (lower left), result Sobolev smoothing on the surface (lower right).

albeit the drag is still considerably reduced. Again this shows that assembling the Laplace-Beltrami operator on the surface seems to be better than doing so in the volume mesh.
Next, consider Picture 7.16, where the cross sections at 65% of the wing length are depicted. As stated before, all optimization algorithms reduce the shock on the upper side. In the cross section plots, it can be seen how this is achieved by making the wing thinner near the leading edge and giving the lower side a distinct curvature. The effect is more pronounced for the SLSQP optimized shape, where only some residue of the original double shock remains. At the same time, the two other optimizers shown have very similar smooth pressure distributions in the cross section. This observation confirms how Sobolev reinterpretation of the derivatives can increase the convergence rate over gradient descent, while resulting in the same local optimum.
When considering the performance of optimization algorithms, there are multiple measurements relevant for a detailed analysis. After investigating the achieved improvements in the objective

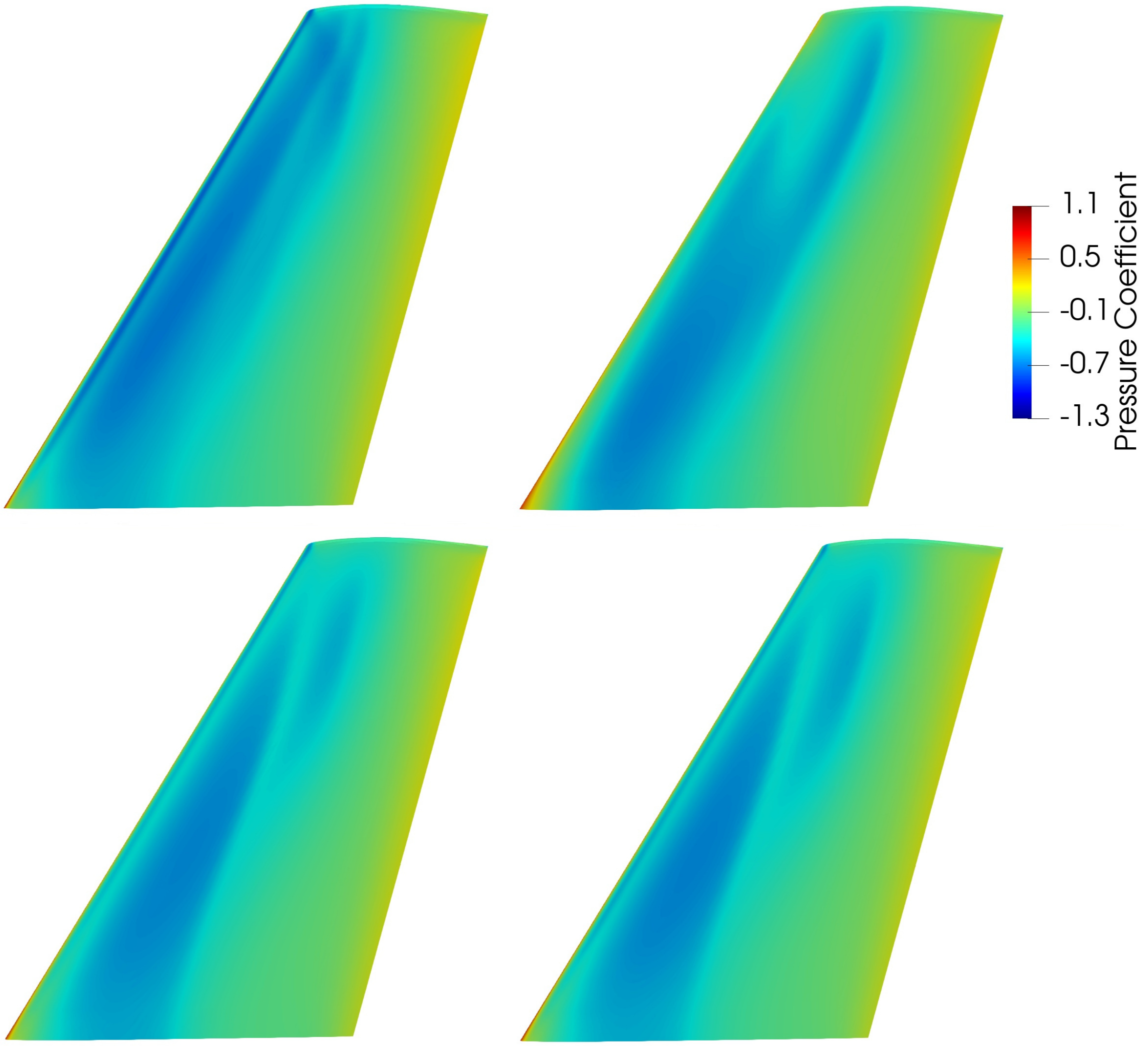


Figure 7.15: Pressure coefficient on the wing surface for optimized solutions. Result Sobolev smoothing in the volume (upper left), result SLSQP (upper right), result gradient descent (lower left), result Sobolev smoothing on the surface (lower right).

function and the optimized flow fields, one should also consider the necessary computational costs to achieve those results. The standard cost measures for any software implementation in this context are runtime and memory requirements.

To compare them, all optimizations are run on the high performance cluster ‘Elwetritsch’ at the ‘Regionales Hochschulrechenzentrum (RHRK), Technische Universität Kaiserslautern’[3]. Each run performs 25 optimization steps, or less if an algorithm converges before. Executables for the mesh deformation, flow solver, and adjoint solver are run in parallel using the MPI parallelization provided in SU2 and two different processor type settings are used. The first setting uses two Intel skylake XEON SP 6126 nodes with 12 cores and 24 threads, both having 96 GB of attached RAM. The second setting uses two Intel sandybridge XEON E5 2670 nodes with 8 cores, 16 threads, and 64 GB of RAM per processor. The test is conducted multiple times to ensure standardized results,

---

[3] Regionales Hochschulrechenzentrum (RHRK), `https://www.rhrk.uni-kl.de`

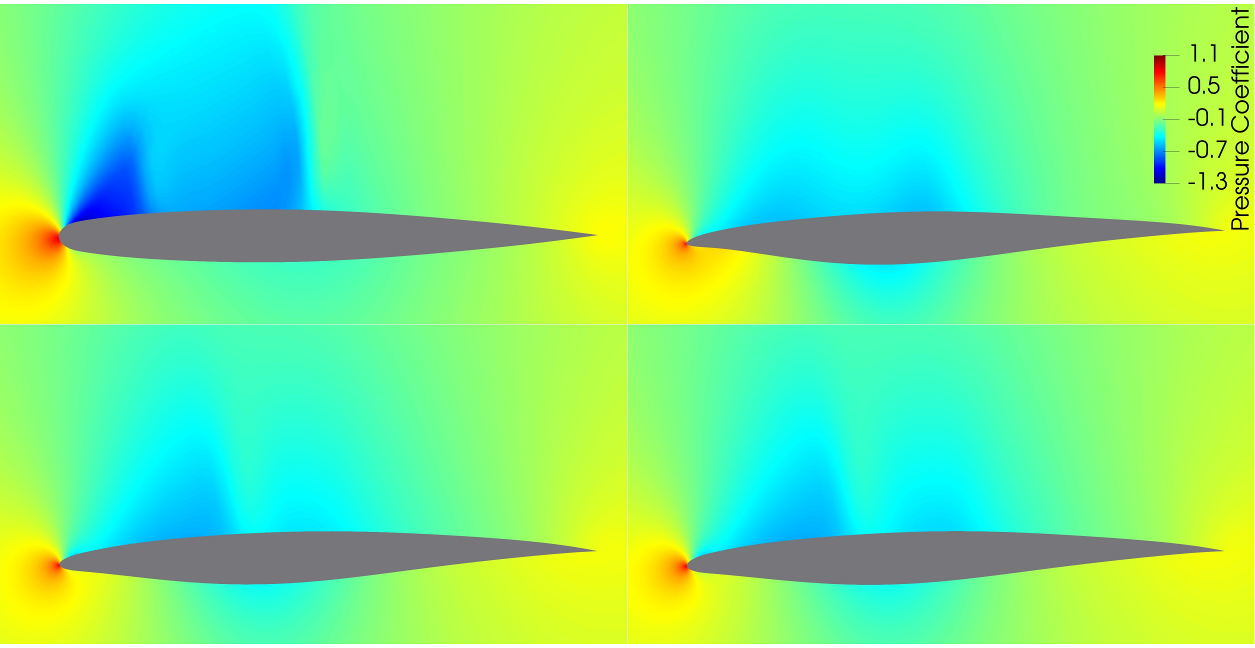


Figure 7.16: Cross sections and pressure values, at $\frac{y}{l} = 0.65$ on the wing, for the original and the optimized wing profiles. Starting profile (upper left), result SLSQP (upper right), result gradient descent (lower left), result Sobolev smoothing on the surface (lower right).

resulting in averaged times and memory. The complete processor nodes are blocked during the execution of the jobs to avoid interference from other processes on the cluster.

The resulting values are displayed in Table 7.5. Reduced SQP optimization, with parameterized Sobolev smoothing, has a comparable runtime to the gradient descent algorithm. This might be surprising at first since the computation of the Laplace-Beltrami operator and the AD evaluation of the parameterization need time. Nevertheless, keep in mind that the flow and adjoint solutions are used as restart values in the next optimization step. This means the increased cost for computing the Hessian approximation in each step can be entirely offset by achieving a smoother deformation and thereby faster convergence in the next iteration. In fact, the evaluation of Equation (5.75) is computational cheap in comparison to the flow solver, which will be important later on for the One Shot method, where it has to be evaluated after a small number of piggyback steps. With regard to memory, there is an observable overhead for Sobolev smoothing. Extra memory has to be allocated to compute the hybrid Laplace-Beltrami operator, via the implementations described in Chapter 6. Finally, the SLSQP optimizer has the lowest total runtime, mainly because it converges after fewer steps than the other methods, while the runtimes per optimization step are comparable.

In an overall comparison, it is possible to draw a series of observations concerning the performance of the Laplace-Beltrami Hessian approximation in the reduced SQP algorithm 3.5.2.

1. The combined matrix from Equation (5.75) can be used in optimization algorithms as an approximation of the real Hessian matrix. This results in a convergent behavior of the optimizer and allows for an effective minimization of the objective function for 3D RANS test cases.

2. Both ways of assembling the Laplace-Beltrami operator, on the surface and in the volume, may be used. Nonetheless, assembling the Laplace-Beltrami operator on the surface seems to be a

	RSQP surface	RSQP volume	SLSQP	grad. desc.
time skylake (sec)	270166.91	263644.04	173046.68	283626.55
time per step skylake (sec)	10806.68	10545.76	9613.7	11345.06
max. RAM skylake (MB)	12378.73	14573.2	9254.96	11322.31
time sandybridge (sec)	396332.82	410679.58	274475.48	395901.84
time per step sandybridge (sec)	15853.31	16427.18	15428.64	15836.07
max. RAM sandybridge (MB)	16852.33	13555.29	9195.27	9534.08

Table 7.5: Averaged time and memory consumption for different optimizers.

better choice than the volume approach and results in improved optimization performance.

3. In comparison to the gradient descent method, one can observe a faster convergence for the surface-based Sobolev smoothing, as expected for an approximated Newton method. This can improve the performance of the shape optimization.

4. The reduced SQP optimization struggles to achieve the same performance as other Quasi-Newton methods with iterative Hessian updates. This can be seen by the SLSQP method showing a better performance in terms of optimizer iterations and runtime. As can be seen from the comparison with the number of design evaluations, SLSQP needs some additional evaluations of the objective and constraint functions to perform line searches and to assemble its internal BFGS Hessian approximation.

5. Applying the newly developed Sobolev smoothing methodology in a reduced SQP algorithm does not increase the wall clock time of the optimization steps compared to pure gradient descent. This means that computing the parameterized Laplace-Beltrami operator is computationally cheap.

6. In terms of memory consumption, the algorithms can vary, but using the presented Hessian approximation technique will generally require more maximal RAM allocation since the adjoint solver is extended. These extensions include the finite element solver for the Laplace-Beltrami operator and AD functionality for the derivatives of the parameterization, as discussed in Chapter 6.

These observations are some of the key points to keep in mind from this test case. They leave two open questions to be answered by the following test case. First, will the computation of the parameterized Laplace-Beltrami operator still be computationally cheap in a One Shot context? Second, can Sobolev smoothing perform better in comparison with iterative Hessian approximations when using approximated gradients in a One Shot optimization? As the following subsection will show, the answer to both questions is yes.

7.2.2 Results for One Shot Optimization

In this section, the new Hessian approximation and gradient reinterpretation strategies introduced in this work are evaluated for their performance with respect to One Shot optimization. Subsection 7.1.2 already showed promising One Shot results for the NACA 0012 test case and now the algorithm is tested in a 3D RANS setting. For the test, the approximated matrix B from Equation (5.75) is used as the preconditioner in Algorithm 4.3.1. From theoretical considerations, it is clear that the exact Hessian matrix of the Lagrangian would be the ideal preconditioner for this system, so it is possible to evaluate the quality of an approximation by applying it in this context.

For the One Shot test, the hybrid Laplace-Beltrami operator is assembled on the surface mesh. This approach offers superior performance with converged gradients in the previous Subsection 7.2.1 and is computationally easier as well. For the weights in the surface setting the same values as in the previous test in Subsection 7.2.1 are used, i.e., $\varepsilon_1 = 56.9$, $\varepsilon_2 = 0.9$, and $\varepsilon_3 = 0.1$. The preconditioning approach is compared to a gradient descent style method, where the identity matrix is used as a constant preconditioner and the step size is restricted. Such a restriction is necessary, as One Shot would diverge for regular-sized update steps based on the approximated gradient. As a second comparison, the SLSQP implementation is tested as well when given One Shot like functions and gradients.

In Subsection 7.1.2, other factors to consider when using One Shot algorithms were mentioned, which will have a significant impact on the performance of the optimization algorithm. They are reexamined here, beginning first with the number of piggyback steps for the flow and adjoint solvers in between design updates and second the maximal step size for design updates. These two values are closely linked and must therefore be chosen in accordance. For the reduced SQP algorithms from the previous Subsection 7.2.1, larger design updates are no problem since all flow and adjoint solvers are fully converged for the new design before computing the next optimization step. For a multistep One Shot algorithm, this is not true. Here, a fixed number of piggyback steps is done using the previous flow and adjoint states as a restart point. If the design updates are too large, the chosen number of steps might not be sufficient to recover good flow and adjoint solutions. In fact, it is expected behavior for One Shot algorithms to diverge under such conditions and it is also questionable whether one should do large design steps based on inexact function and gradient values at all.

All of these considerations are linked together, as can be seen by examining Algorithms 4.3.1. Making the design updates small enough will stabilize the convergence of the optimization, while simultaneously stalling its progress. This can be easily achieved by choosing a positive definite preconditioning matrix with a large enough operator norm. Therefore, a good preconditioner must keep the flow and adjoint simulations stable, while at the same time allowing for design update steps as large as possible.

In Figures 7.17 and 7.18, the results for several different One Shot optimizations can be observed. All optimization algorithms perform 10 piggyback steps in between design updates to ensure equal conditions. Here, they are analyzed individually at first, before comparing them later on.

1. At first, look at One Shot with a constant preconditioner in Figure 7.17, which is equivalent to gradient descent with a limited maximal step size, shown in blue and magenta. As can be observed from the blue line, this results in a descent of the objective function, however for a maximal allowed step size of $2.5e^{-3}$, the optimizer shows an oscillating behavior. Increasing the maximal allowed step size to $5e^{-3}$ leads to the computation shown by the magenta line. This results in a faster reduction in the objective function at first, but also in increased oscillation, to the point where the whole One Shot process becomes unstable and diverges in the end. To solve this issue of oscillatory behavior, one might suggest increasing the number of piggyback iterations between design updates to achieve better function and gradient values.

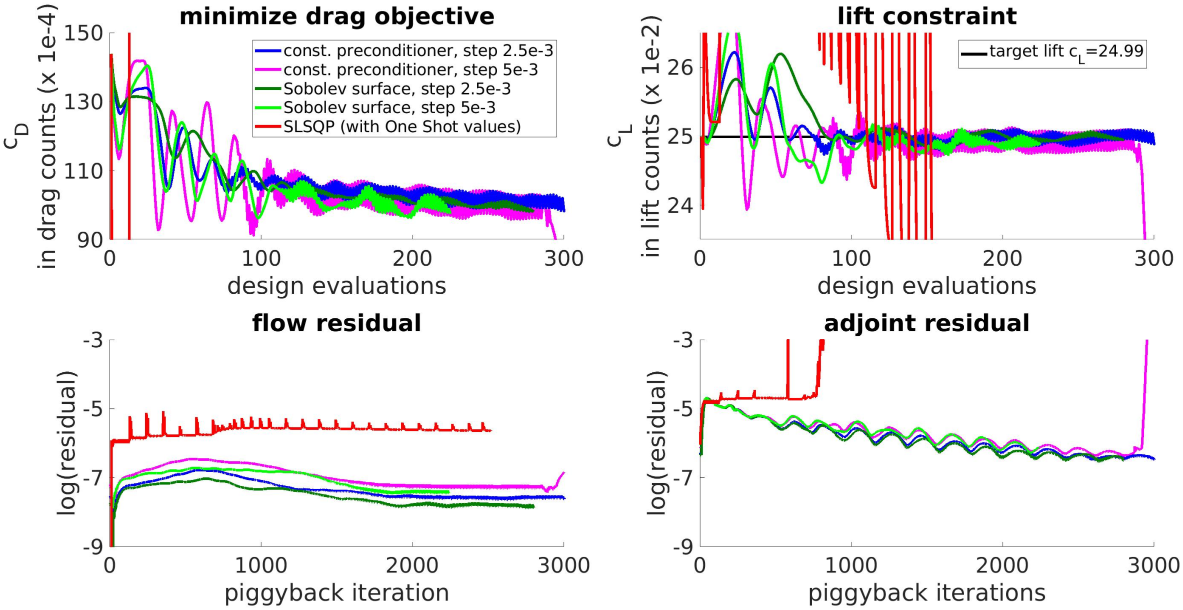


Figure 7.17: Comparison of different One Shot algorithms for the ONERA M6 test case.

However, this is counterproductive, as such additional costs would slow down the optimization progress. In Figure 7.18, the constant preconditioner is shown again, this time with a heuristic that decreases the allowed step size from a maximum of $5e^{-3}$ as the optimization progresses. This helps to limit the oscillations, albeit some remain, and results in an optimized drag of 101.3 drag counts. All the while, keeping the lift constraint at 24.7 lift counts, close to the target lift, and retaining stability in the flow and adjoint simulations.

2. Next, the One Shot optimization is run using the Hessian approximation via hybrid, parameterized Sobolev smoothing as a preconditioner. The Hessian approximation B was constructed using the weights of $\varepsilon_1 = 56.9$, $\varepsilon_2 = 0.9$, and $\varepsilon_3 = 0.1$, as stated above. Optimizations utilizing the surface Sobolev smoothing approach are run for different maximal step sizes, as shown by the light and dark green lines in Figure 7.17. Considering the course of the optimizations plotted there, it can be observed that preconditioning the search direction with the hybrid Laplace-Beltrami operator dampens the oscillations of the optimizer and helps the process to converge faster and with larger design updates without becoming unstable. This can be improved even further if a variable maximal step size is used again, as shown in Figure 7.18. This results in a fast decrease of the drag to 100.9 drag counts, while keeping the lift at 25.1 lift counts, within 0.65% from the target. Overall, the Sobolev based preconditioner appears to be an excellent choice for this One Shot optimization test case.

3. The test for the SLSQP optimization algorithm is shown by the red line in Figure 7.17. Here, the SciPy library is given function handles to call a piggyback function and adjoint evaluation with a limited number of 10 inner iterations. As can be seen from the plot, the SLSQP algorithm does not work in a One Shot setting. The algorithm has lost the ability to compute reasonable descent steps, leading to a fast divergence of the whole process. This change in performance becomes especially noticeable compared to the excellent performance SLSQP

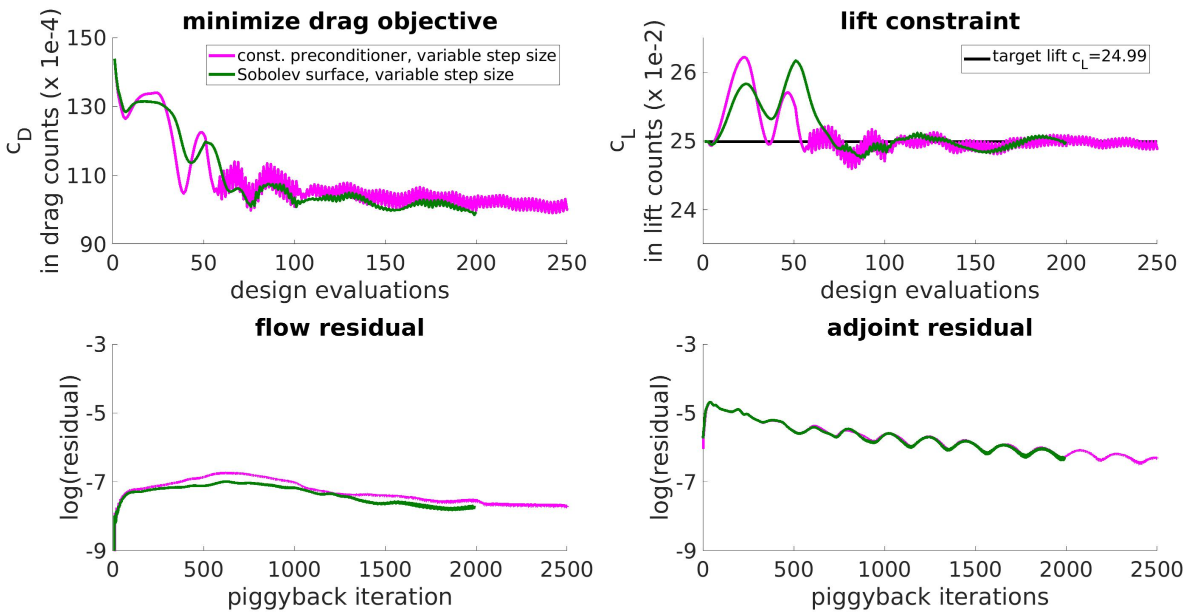


Figure 7.18: Comparison of One Shot optimization, with varying maximal step size, for Sobolev smoothing on the surface against constant preconditioning.

demonstrated in Subsection 7.2.1.

The explanation for this behavior involves the same factors which made SLSQP perform so well with exact functions and gradients. The internal Hessian approximation is built iteratively using the BFGS update formula. If given inexact derivatives, the numerical errors in the Hessian approximation will accumulate, leading to poor quality design updates. The second problem is that the line search is based on the descent of a merit function. If the function values are not exact, applying descent conditions to them will result in incorrect step size choices.

As is typical for One Shot optimizations, after the last design update, the flow simulations are converged to machine precision, which is not depicted in the plots. The resulting drag and lift values are stated in the text above and are generally in good accordance with the intermediate results depicted in the figures, i.e., the optima computed by One Shot were valid solutions.

The next point of interest are the optimal shapes resulting from One Shot optimization. The resulting flow fields for the optimized wing geometries are shown in Figure 7.19. Here, the pressure coefficients on the upper wing surface are shown since this is the area where the lambda shock is located. Both converged One Shot preconditioning approaches can efficiently remove the lambda shock on the upper side of the wing. This means they can eliminate the primary source of drag in this test case, while simultaneously keeping the lift constraint within close bounds and abiding by the geometric constraints. As a result, the double shock system is removed and only remnants of a small shock in the middle of the wing remain afterwards. Although, one can observe that Sobolev smoothing results in slightly smoother pressure distributions and a smaller residue shock.

The cross sections at 65% of the wing length can be seen in Figure 7.20. The different One Shot runs both result in smooth flow fields with a reduced double shock, similar to the results seen in Figure 7.19. For Sobolev smoothing on the surface, a smaller residue shock remains and unsurprisingly, it

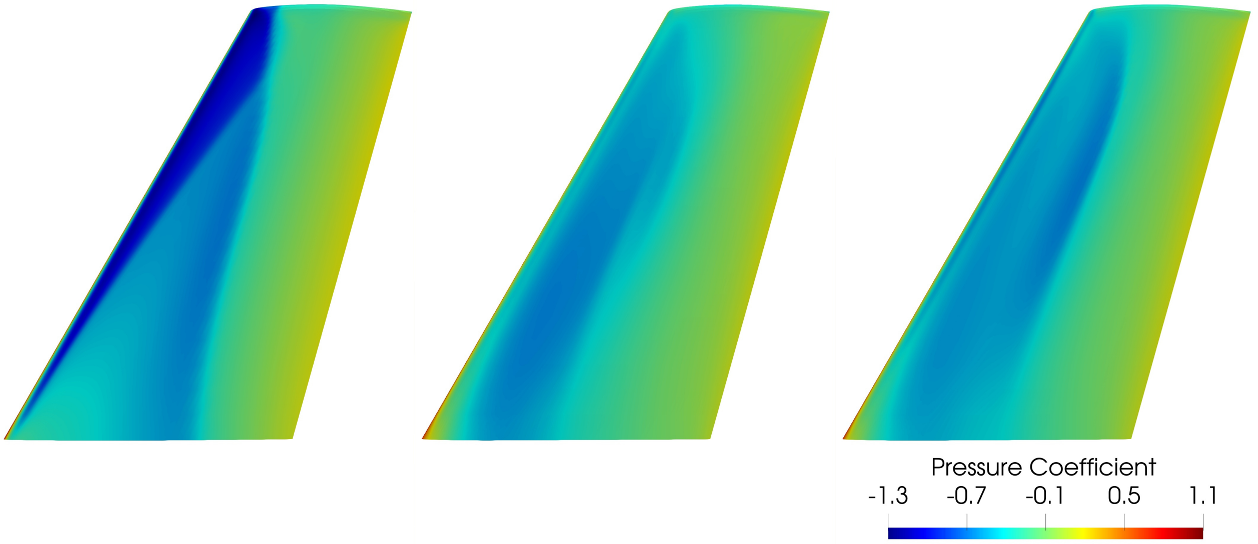


Figure 7.19: Pressure coefficient on the wing surface for the original wing and the designs optimized by One Shot. Starting wing (left), result Sobolev smoothing (center), result gradient descent (right).

also results in a slightly lower drag coefficient.
Considering the promising flow results for One Shot with parameterized Sobolev smoothing, the next question is how the method compares in terms of the computational cost. The key concept is to speed up the optimization by simultaneously driving flow, adjoint, and design to convergence. Table 7.6 shows the runtime and memory for the One Shot optimization algorithm in multiple settings. A number of 250 optimization steps are performed for both settings. Computations are once again executed on the same two processor architectures as investigated in Subsection 7.2.1. As the first setting, two Intel skylake XEON SP 6126 nodes with 12 cores, 24 threads, and 96 GB of RAM each are used, where the piggyback solver, the computation of the Hessian approximation, and the mesh deformation are run on 48 parallel MPI processes. Second, two Intel sandybridge XEON E5 2670 processors with 8 cores, 16 threads, and 64 GB of attached RAM are used, which results in an execution on 32 MPI processes.

	One Shot surface Sobolev smoothing	One Shot constant preconditioner
time skylake (sec)	41280.76	39346.5
memory skylake (MB)	19932.02	19287.57
time sandybridge (sec)	69830.6	89549.92
memory sandybridge (MB)	19670.87	34036.22

Table 7.6: Averaged time and memory consumption for different One Shot optimizers.

Taking a look at the times recorded in Table 7.6, a significant improvement in runtime can be observed when using One Shot algorithms as opposed to the times measured previously in Table 7.5. Comparing the One Shot measurements for Sobolev smoothing on the surface to Table 7.5, a speedup of roughly $3.98 - 4.19$ against SLSQP can be observed. In terms of memory usage, an

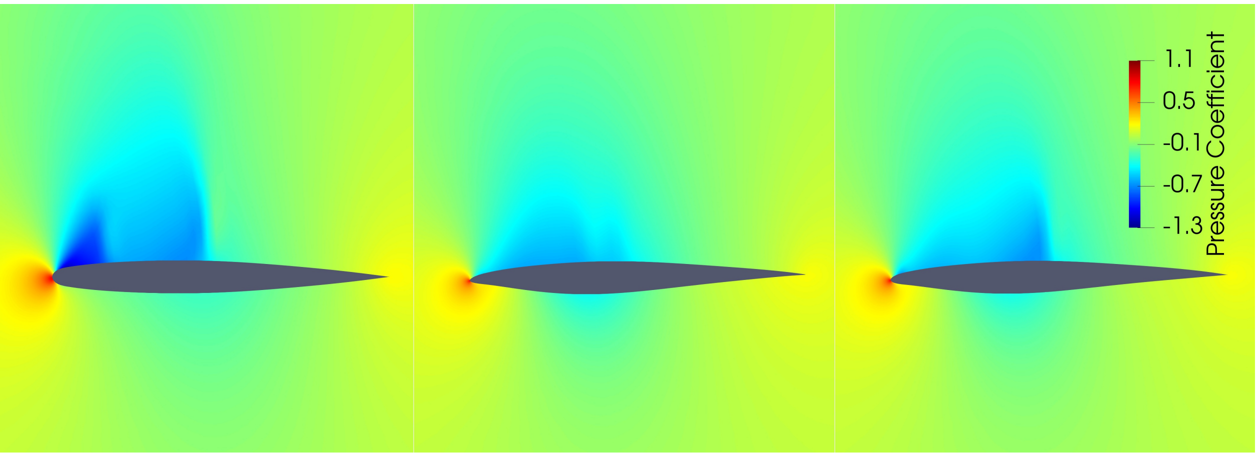


Figure 7.20: Cross section and pressure values, at $\frac{y}{l} = 0.65$ on the wing, for the original and the One Shot optimized wing profiles. Starting profile (left), result Sobolev smoothing (center), result gradient descent (right).

increase in RAM allocation against the reduced SQP optimization tests in Table 7.5 can be seen. This is due to the One Shot driver described in Subsection 6.2.2. As explained, the piggyback logic requires a new AD recording after each flow iteration, leading to increased memory consumption. The retardation factor is frequently used throughout the literature to put the time consumption of optimization methods into perspective. This quotient is computed by dividing the time required to do a complete optimization by the time required to converge the flow simulation for one design up to machine precision. For the given test case, the runtime of one flow simulation is 2224.1 sec. on the skylake and 4584.53 sec. on the sandybridge architecture. The resulting retardation factors are shown in Table 7.7.

	time retardation (skylake)	time retardation (sandybridge)	iteration retardation -
RSQP, Sobolev surface	121.47	86.45	-
RSQP, Sobolev volume	118.54	89.58	-
SLSQP	77.81	59.87	-
One Shot, Sobolev surface	18.56	17.69	4.08
One Shot, const. precond.	19.53	15.23	4.1

Table 7.7: Retardation factors for optimization.

Investigating the retardation factors presented here, one can see that the use of One Shot optimization results in a significant improvement. It is especially worth noting that time retardation factors view the process purely in terms of flow simulation cycles. This neglects that each design update will incorporate multiple adjoint simulations for the objective function and the flow constraint. Therefore, when investigating retardation factors, one needs to keep the performance of the adjoint solver in mind. Fortunately, the SU2 framework is used in this work, which has a very fast and efficient adjoint solver. Nonetheless, the reader should keep in mind that the piggyback solver implemented for this thesis, as explained in Section 6.2.2, has to do a new recording of the AD tape in each flow step. This overhead will result in a slower execution of the flow simulation than the pure flow solver executable without any AD features. Overall, the computed retardation factors between 17.69 and 18.56 are still very competitive for any optimization algorithm.

For example, the SLSQP optimization in Subsection 7.2.1 needs to perform 18 full flow simulations alone, without taking adjoint computations, etc. into account. Using the pure wall clock time for the full optimization results in a retardation factor between 59.87 and 77.81, depending on the architecture. A comparison of the different optimizers for the ONERA M6 wing test case reveals that the surface Sobolev smoothed One Shot optimization on the skylake nodes needs only 23.86% of the runtime SLSQP needs on the same architecture. Overall, this represents the stated $3.98 - 4.19$ speedup that the new algorithm has over established Quasi-Newton methods.

Also, paradoxically the retardation factors seem smaller on the older sandybridge chips. Due to the flow simulation better utilizing parallelization and vectorization on the newer skylake architecture, the baseline time is significantly smaller here. At the same time, the involved adjoint and other solvers cannot exploit these advantages in vectorization and other parallelization features to the same degree, leading to larger quotients when computing the retardation factors.

The numerical results presented above yield a series of important observations.

1. The hybrid Laplace-Beltrami operator introduced in this thesis can be used successfully as a preconditioner for One Shot optimization. Inserting the approximated Hessian matrix B from Equation (5.75) into this role leads to a stable and convergent One Shot optimization.

2. Using the new preconditioning methodology allows for larger design updates than a constant factor as a preconditioner. In addition, the parameterized Sobolev preconditioning helps reduce the oscillations in the optimizer, which would otherwise occur due to inexact function and gradient values. In turn, this enables a faster convergence of the optimization algorithm to an optimal solution.

3. The poor performance demonstrated by iterative Hessian update formulas, e.g., BFGS, in contrast, shows that traditional Quasi-Newton methods cannot be trivially adapted to work in One Shot optimization. Iterative Hessian update formulas struggle to compute a good approximation from inexact, approximated gradients.

4. Overall, the parameterized Sobolev preconditioned, multistep One Shot algorithm demonstrates significant advantages in terms of runtime compared to established Quasi-Newton methods, like SLSQP.

In conclusion, during the presented tests with the ONERA M6 wing, the combination of Sobolev smoothing and parameterization yielded significant benefits for optimization. Special focus is given to the application in a One Shot optimization algorithm, where the method can outperform established, classical Quasi-Newton algorithms.

To finish this chapter, the new methodology introduced in this thesis has been successfully tested for the different reference test cases for design optimization presented here. The parameterized Sobolev smoothing demonstrated competitive performance in aspects such as objective function reduction, retardation factors, runtime, etc. A more in-depth discussion and resulting conclusions will be drawn in Chapter 8.

Chapter 8

Conclusion and Outlook

8.1 Conclusion

Discussion

As a starting point, recall all the different backgrounds in design optimization from where the new methodology is derived. Two main lines of development drive it, beginning with the extensive research done on discrete adjoint optimization, based on algorithmic differentiation, discussed in Chapter 3. Such adjoint approaches offer a great way to compute derivatives for optimization independent of the number of design parameters. Here, using a fixed point formulation is especially advantageous since it enables the stable computation of consistent gradients. In Section 3.2, the necessity of projecting derivatives onto the parameters becomes already clear. A point that gets more involved in later chapters when dealing with second order derivatives. The extension to One Shot optimization is especially noteworthy when using discrete adjoints, since it is naturally based on the fixed point formulation, as discussed in Chapter 4. Many authors have worked on this in the past, and formulating a comprehensive, generally constrained One Shot algorithm with guaranteed convergence is still an open question. In particular, the construction of a preconditioner which can guarantee the desired convergence while not being too strict and stalling optimization progress remains a complex, test case dependent task.

The second background, discussed in its various aspects in Chapter 5, lies in the field of shape calculus and the reinterpretation of gradients in different Hilbert spaces. Naturally, this kind of research focuses on computing these gradients, by solving a partial differential equation with an elliptic operator. Sobolev gradient smoothing particularly uses the Laplace-Beltrami operator for a reinterpretation, which can help to increase the regularity of search directions and dampen high-frequency noise in the derivatives. Such methods are further motivated by the connection with the analytic shape Hessian from continuous optimization. As seen in Section 5.1, a connection between the Hessian and elliptic differential operators, e.g., the Laplace-Beltrami operator, can be motivated in two ways. First, by approximating the Hessian operator symbol in terms of Fourier analysis and second, by investigating Dido's problem. Such approaches offer considerable potential to apply theoretical results from differential calculus but can get very involved in practical applications. Many authors have tried to find a good approximation for the shape Hessian operator, although doing this for general optimization problems with Navier-Stokes equations remains an open question.

Combining all of these developments, it becomes clear that One Shot optimization benefits from preconditioners derived from Sobolev smoothing. An approach that has been demonstrated for free node optimization in the past. However, there remains a challenge to be addressed. Nearly all real-world industrial applications will incorporate a mesh parameterization. Usually, such a formulation involves a CAD tool for designing an aerodynamic component and a mesh generation process to get the necessary grid for computations. Therefore, an attempt to apply gradient smoothing in such a scenario should be formulated for the gradient with respect to the design parameters. Furthermore, free node optimization can be unfeasible for large, complicated industrial optimization problems, due to a large number of design parameters, when all mesh nodes are considered, and the sensitive nature of the problem. That is, freely deforming a mesh will easily reduce its quality and subsequently lead to problems with flow and adjoint convergence. Historically, Sobolev smoothing was partially introduced to deal with the issue of loss of regularity. Also, transforming the optimized meshes back and expressing them in terms of the parameterization is nontrivial in itself.

For these reasons, this thesis investigates how introducing a design parameterization affects the optimization procedure with Sobolev smoothing for the derivatives. This involves transforming the formulation of an approximated Newton step from the mesh to the design parameters. In particular, a new Theorem 5.3.1 for the connection of the reduced shape Hessian, discretized on the mesh, and the Hessian with respect to the design parameters is introduced and proven. The result is based on the generalized Faà di Bruno formula and enables the application of Sobolev smoothing for the design parameters in Section 5.3. Special focus is given to linear parameterizations, allowing for a simplified formulation. This new approach is derived from function space and shape calculus considerations and is independent of the accuracy of the discrete adjoint gradient. As such, it can have significant benefits in situations where only an approximation of the gradient is computed, such as One Shot optimization algorithms, which work with intermediate, non-converged adjoint values. Here, iterative Hessian updates would suffer from an accumulation of numerical errors. Also, traditional iterative methods, like BFGS, are ill-conditioned for high-dimensional design spaces and the Hessian approximation deteriorates in quality with subsequent updates. This behavior further motivates the approach taken in this thesis.

With the results from this thesis, it is possible to apply Sobolev smoothing for arbitrary design parameterizations and fit it into a flexible reduced SQP framework in place of the Hessian approximation. The overall optimization algorithm includes treatment of additional equality and inequality constraints, depending on the mesh or flow state. It can be easily extended with a piggyback iteration for the flow and adjoint solvers to formulate a constrained multistep One Shot optimization. This generalized SQP framework is especially interesting as it allows for the straightforward inclusion of additional constraints into the One Shot setting.

Having all algorithms in place, key implementation ideas are pointed out in Chapter 6. While this is done in the SU2 framework, the discussed issues are relevant for any potential implementation. Thus, the presented ideas could be implemented into arbitrary finite volume CFD solvers and design frameworks. Two points are particularly worth mentioning in this context, as they need to be treated with additional care. One is the extension of linear finite elements to surfaces. Many libraries support higher order elements, but not necessarily their embedding into curved higher-dimensional spaces. Second, multiple matrix vector products with the Jacobian of the parameterization and its transposed matrix must be evaluated. If the source code is available and can be treated with an AD tool, it is possible to implement very efficient code for this computation using algorithmic differentiation. For closed source or commercial design tools, one could still use finite differences in this role, although a loss in performance has to be expected from such an approach.

The established methodology and the implemented algorithms are tested for relevant test cases in Chapter 7. Comprehensive results are shown for the two-dimensional NACA 0012 airfoil with Euler equations and the more complicated three-dimensional ONERA M6 wing with RANS equations. In solving drag minimization problems, the presented Sobolev smoothing methodology showed convergent behavior and good results for reduced SQP optimizers. When reinterpreting the gradient, the increased regularity can allow for better convergence of the algorithm compared to classical gradient descent methods. At the same time, the computational cost of calculating the parameterized hybrid Laplace-Beltrami matrix is small compared to converging a single flow solution, allowing for computationally cheap preconditioning. However, the method has issues achieving comparable performance to iterative Hessian updates, like BFGS formulas, for the three-dimensional RANS test case with exact function values and gradients.

This changes in the next series of tests, when the derived results are combined with a One Shot optimization, following Algorithm 4.3.1. In this setting, parameterized Sobolev smoothing achieves stable and efficient convergence to an optimal design for both the NACA 0012 and ONERA M6 test case. When compared to constant preconditioning schemes, the Sobolev preconditioner is able to show measurably better performance in stabilizing the iteration. Here, the advantage compared to iterative Hessian updates becomes significant. For example, BFGS formulas have serious problems in such situations, as seen for the ONERA M6 test case. As mentioned earlier, this is due to the inexact approximation of adjoint gradients, provided by only a handful of piggyback steps. In contrast, the presented Sobolev smoothing based method is independent of low quality gradient approximations and remains stable even for high numbers of design parameters. Furthermore, the use of One Shot optimization leads to a significant improvement in runtime over classical fully converged optimization algorithms. Due to the fast assembly of the Laplace-Beltrami operator on the design surface mesh and the efficient evaluation of parameterization derivatives with AD, the additional costs remain controllable, even for the high number of design updates in One Shot optimization. This leads to a very competitive performance of the new algorithm, as seen in the investigation of retardation factors in Section 7.2.2. The multistep One Shot algorithm with parameterized Sobolev smoothing can outperform established Quasi-Newton methods, like SLSQP, with a speedup factor between $3.98 - 4.19$. At the same time, it achieves better stability than other One Shot preconditioning methods. Together, this demonstrates significant advantages for the methodology derived in this thesis and the new algorithm.

Summary

At last, this subsection gives a compact overview of the results from the previous discussion and compares them with the scientific objectives for this work, as formulated in Section 1.2.

To achieve this, recapitulate the title of this thesis, *‘Combining Parameterizations, Sobolev Methods and Shape Hessian Approximations for Aerodynamic Design Optimization’* and how it set the research goal of this work. First, the role of parameterization in design optimization is considered, particularly when working with an existing discrete adjoint optimization framework and the One Shot approach. Special focus is given to the incorporation of the chain rule to project derivatives onto the parameters for computing a design update. Next, an overview survey of shape Hessian approximation ideas is presented, motivating the use of the Laplace-Beltrami operator from Sobolev smoothing in this role. All of these results are combined into a novel approach, deriving a theorem based on the generalized Faà di Bruno formula, which connects the discretized reduced shape

Hessian on the mesh with the second order derivatives with respect to the parameterization, and using Sobolev smoothing in this formulation. This novel combination approach forms the main theoretical result of the presented thesis.

The introduction of parameterized Sobolev smoothing allows the inclusion of the Laplace-Beltrami operator into an optimization framework, leading to the formulation of reduced shape Hessian approximation techniques for reduced SQP and One Shot algorithms. Together, these two algorithms demonstrate how to apply shape Hessian calculations in an industrially relevant optimization setting. The formulated algorithms are successfully implemented into a modern CFD design framework and all the key features of the implementation process are documented and discussed. This discussion of the most important points and known relevant issues can serve as a template for future adaptations in other solvers.

By applying the implementation to numerical test cases, the full effects of the new formulation can be studied. The established results demonstrate how parameterized Sobolev smoothing can increase the regularity of an optimization algorithm. Additionally, due to the flexible formulation of the reduced SQP update, a complete set of mixed constraints could be included as well. Furthermore, these techniques prove highly effective as a preconditioner to constrained multistep One Shot optimization and such a One Shot algorithm is remarkably efficient in practice. By keeping computational costs low, the method remains competitive while resulting in a good reduction of the objective function and abiding by the constraints. It outperforms established Quasi-Newton methods based on iterative Hessian updates with exact gradients.

In conclusion, the novel approach in this thesis achieved considerable benefits in design optimization by applying Sobolev smoothing to parameterized shapes, even though the parameterization itself might already be smooth. In summary, this thesis has achieved its research objective of giving an efficient combination of parameterizations, Sobolev methods, and shape Hessians. This was successfully applied to aerodynamic design optimization and established a new, competitive algorithm for classical reduced SQP and One Shot optimization frameworks.

8.2 Outlook

Hopefully, this thesis gave answers to some interesting research questions. Although, follow-up ideas and open topics for further investigation remain. In this last section of the thesis, some of the possible continuations for this work are pointed out.

The overall objective throughout this thesis was to combine parameterizations with Sobolev smoothing methods and shape Hessian approximations. This goal has been achieved successfully. To do this, the mathematical background was explored and the incorporation of the parameterization into approximated Newton steps with a shape Hessian derived. Of course, using these theoretical results for Sobolev smoothing is only one possibility. Section 5.1 already listed literature with other results on Hessian approximation. The introduced formulation from Theorem 5.3.1 is quite general and offers enough flexibility to combine other shape Hessian approximations, e.g., different elliptic partial differential operators, with a parameterization into a preconditioning method for the design parameter update. With the multitude of shape calculus results available for various problems, much future research could be done to improve aerodynamic design optimization in many different areas. Such ideas are not limited to operator symbols derived by Fourier analysis. If it is possible to calculate an analytic expression for the complete Hessian matrix on the surface for a specific problem, it can be discretized on the mesh and then inserted into Equation (5.44).

On the other hand, one might not need to look at new techniques. The Sobolev smoothing method applied in this work is equivalent to a reinterpretation of the gradient in a different scalar product to prevent a loss of regularity. This means that other scalar products, or their respective hermitian operator, can also be used in this context. In a sense, the scalar product is already adapted when choosing a set of weights ε_1, ε_2, ε_3 in Equation (5.75), yet this can be further refined. When computing the Laplace-Beltrami operator via finite elements, a discrete representation of the operator is gradually assembled on the mesh cells. There is no reason why the optimal weights have to be constants on the whole mesh. Instead, a function could be defined to calculate good values for ε_1, ε_2 based on the properties of the current mesh cell. Kusch, Schmidt, and Gauger [71] already pointed in this direction when they derived a formulation based on a local coordinate system along the surface. The idea could also be expanded to adapt the ε values throughout the optimization, to keep them in line with the change in shape.

Corollary 5.3.2 simplifies the expression connecting the discrete reduced shape Hessian and the parameter Hessian. However, it only holds true for linear parameterizations, or if the optimization process is already close to the optimum. For nonlinear parameterizations and starting points further away from the optimum, the term $\sum_{k=1}^{n_x} \frac{\partial}{\partial x_k} L(u,\lambda,x)\, \mathrm{D}_{pp} M_k(p)$ might play a relevant role. This means that an accurate Hessian approximation on the design parameters has to take the second order derivatives of $M(p)$ into account. It is worth noting that such an approach is nontrivial, as many tools used in engineering and industrial application do not support the computation of those terms, and applying finite difference approximations can result in significant computational overheads. Even if the source code of the parameterization is available for AD applications, computing the full Hessian will still be expensive.

Another influence of the parameterization, which has not been fully investigated, is how the function $M(p)$ itself might act as a smoothing procedure. As pointed out in the motivation section, applying adjoint aerodynamic design optimization on a free node formulation will struggle with high-frequency noise in the derivatives with respect to the mesh and decreasing mesh quality after deformation. This is especially true for One Shot optimization, where adjoint solutions are not fully converged. For mesh deformation, a surface movement is frequently distributed to the volume mesh by a linear stiffness method or a similar approach. This has dampening properties for noise in the volume sensitivities when transferred back to the surface. Next, assume that there is a disturbance on the mesh coordinate derivatives $\delta(\mathrm{D}_x F)$. This is multiplied by application of the chain rule, i.e., $\delta(\mathrm{D}_p F) = \delta(\mathrm{D}_x F)\mathrm{D}_p M$. Experience may suggest that there are many surface nodes in the mesh and that adding a small, high-frequency disturbance to some of them will not affect the parameters too much. In this sense, the parameterization could act as a dampening in its own right. However, no thorough mathematical analysis has been conducted, and doing so could lead to interesting new results.

Another possible extension, which has been outside the scope of this work, is an application to different design optimization problems. For example, optimizing the shape of an object under a PDE constraint is not restricted to aerodynamic applications. Instead, other areas of engineering, e.g., structural mechanics or thermal regulation, are mathematically similar. The theoretical framework from this thesis could be expanded to precondition search directions for such problems, potentially expanding the formulation for coupled problems with flow equations.

Interesting questions remain for One Shot optimization as well. As stated in Chapter 4, many preconditioning strategies have been tried for different One Shot optimizations to achieve stable yet rapid convergence. As demonstrated by the numerical results of this thesis in Section 7.2, traditional iterative Hessian updates will struggle here. Therefore, new methods to derive computationally cheap preconditioning matrices B are greatly appreciated. Theoretical conditions for a suitable preconditioner are known from the convergence analysis of the One Shot methods. However, these properties are very hard to check in practice for each individual design update. Furthermore, they are trivially fulfilled if B is just positive definite enough. Yet, such a choice is highly undesirable since it slows down the optimization progress. In this thesis, a new efficient preconditioner was introduced that can help to ensure convergent and fast optimization. The remaining questions include a rigorous convergence analysis for this new method.

Finally, there is always room for performance improvements in the implementation and application of the method. The whole setting could be applied in an improved One Shot algorithm. While multistep One Shot implementations are very flexible in their own right, recent trends in HPC go beyond this. With the increased use of parallelization, it is no longer necessary to have different solvers run in a clear deterministic order. In particular, the idea of asynchronous One Shot could be very interesting in the future [21]. Here, one would have the flow, adjoint, and optimization iterations running independently from each other on multiple machines. Coupling can be done by defining interfaces to exchange deformed meshes, flow, and adjoint states. A further suggestion to increase the efficiency of such an approach is made here. The exchange of these large data files can be sped up significantly when using a modern memory layer API, e.g., the GASPI interface[1], instead of traditional file I/O. These techniques have been successfully applied to CFD [113] and could be adopted to allow for highly efficient One Shot optimization in industrial applications on modern HPC architectures. Such an implementation combined with the new shape Hessian approximation, based on parameterized Sobolev smoothing, would be a genuinely remarkable algorithm to tackle even some of the most challenging industrial aerodynamic design optimization problems.

---

[1] GPI-2 - Global Address Space Programming, `http://www.gpi-site.com`

References

[1] AGARD Advisory Report No. 138. *Experimental Database for Computer Program Assessment*. Technical Editing and Reproduction Ltd. North Atlantic Treaty Organization Advisory Group for Aerospace Research and Development (Organisation du Traite de l'Atlantique Nord). 1979.

[2] R. Adams and J. Fournier. *Sobolev Spaces*. Second Edition. Vol. 140. Pure and Applied Mathematics. Elsevier, 2003. ISBN: 978-0-1204-4143-3.

[3] T. Albring, T. Dick, and N. R. Gauger. "Assessment of the Recursive Projection Method for the Stabilization of Discrete Adjoint Solvers". In: *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA 2017-3664* (2017). DOI: `10.2514/6.2017-3664`.

[4] T. Albring, M. Sagebaum, and N. R. Gauger. "Development of a Consistent Discrete Adjoint Solver in an Evolving Aerodynamic Design Framework". In: *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA 2015-3240* (2015). DOI: `10.2514/6.2015-3240`.

[5] T. Albring, M. Sagebaum, and N. R. Gauger. "Efficient Aerodynamic Design using the Discrete Adjoint Method in SU2". In: *17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA 2016-3518* (2016). DOI: `10.2514/6.2016-3518`.

[6] G. Allaire, C. Dapogny, and F. Jouve. "Shape and topology optimization". In: *Geometric Partial Differential Equations - Part II*. Ed. by A. Bonito and R. Nochetto. Vol. 22. Handbook of Numerical Analysis. hal-02496063. Elsevier, 2021. Chap. 1, pp. 1–132. DOI: `10.1016/bs.hna.2020.10.004`.

[7] M. Andersen, J. Dahl, Z. Liu, and L. Vandenberghe. "Interior-Point Methods for Large-Scale Cone Programming". In: *Optimization for Machine Learning*. Ed. by S. Sra, S. Nowozin, and S. J. Wright. Neural Information Processing series. MIT Press, 2011. Chap. 3. DOI: `10.7551/mitpress/8996.003.0005`.

[8] J. Anderson. *Fundamentals of Aerodynamics*. Sixth Edition. McGraw-Hill Education, 2016. ISBN: 978-1-25912-991-9.

[9] G. Araya. "Turbulence Model Assessment in Compressible Flows around Complex Geometries with Unstructured Grids". In: *Fluids* 4 (2019), p. 81. DOI: `10.3390/fluids4020081`.

[10] E. Arian and S. Ta'asan. "Analysis of the Hessian for aerodynamic optimization: inviscid flow". In: *Computers & Fluids* 28.7 (1999), pp. 853–877. ISSN: 0045-7930. DOI: `10.1016/S0045-7930(98)00060-7`.

[11] E. Arian and V. N. Vatsa. "A Preconditioning Method for Shape Optimization Governed by the Euler Equations". In: *International Journal of Computational Fluid Dynamics* 12.1 (1999), pp. 17–27. DOI: 10.1080/10618569908940813.

[12] J. Backhaus, A. Schmitz, C. Frey, M. Sagebaum, et al. "Application of an Algorithmically Differentiated Turbomachinery Flow Solver to the Optimization of a Fan Stage". In: *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA 2017-3997* (2017). DOI: 10.2514/6.2017-3997.

[13] W. Bangerth, R. Hartmann, and G. Kanschat. "deal.II – A general-purpose object-oriented finite element library". In: *ACM Trans. Math. Softw.* 33.4 (2007), 24–es.

[14] A. Baydin, B. Pearlmutter, A. Radul, and J. Siskind. "Automatic Differentiation in Machine Learning: a Survey". In: *Journal of Machine Learning Research* 18 (2018), pp. 1–43.

[15] A. D. Belegundu and T. R. Chandrupatla. *Optimization Concepts and Applications in Engineering*. Third Edition. Cambridge University Press, 2019. ISBN: 978-1-10842-488-2.

[16] M. Bendsøe and O. Sigmund. *Topology Optimization: Theory, Methods and Applications*. Second Edition. Springer, 2003. ISBN: 3-540-42992-1.

[17] M. Bern and P. Plassmann. "Mesh Generation". In: *Handbook of Computational Geometry*. Ed. by J.-R. Sack and J. Urrutia. Elsevier Science, 2000. Chap. 6, pp. 291–332. ISBN: 978-0-444-82537-7. DOI: 10.1016/B978-044482537-7/50007-3.

[18] J. Blazek, ed. *Computational Fluid Dynamics: Principles and Applications*. Elsevier, 2001. ISBN: 0-080-43009-0.

[19] M. Blommaert. *Automated Magnetic Divertor Design for Optimal Power Exhaust*. Vol. 365. Energy & Environment. PhD thesis. RWTH Aachen. Schriften des Forschungszentrums Jülich, 2016. ISBN: 978-3-95806-216-0.

[20] J. Bonet and R. D. Wood. *Nonlinear Continuum Mechanics for Finite Element Analysis*. Cambridge university press, 1997. ISBN: 0-521-57272-X.

[21] T. Bosse. "An asynchronous Oneshot method with Load Balancing". In: *Proceedings of the 18 Euro AD workshop*. 2015. URL: http://www.autodiff.org/Docs/euroad/18th%20EuroAd%20Workshop%20-%20Torsten%20Bosse%20-%20A%20Blurred%20OneShot%20Method%20for%20Design%20Optimization.pdf.

[22] T. Bosse. "Augmenting the one-shot framework by additional constraints". In: *Optimization Methods and Software* 31.6 (2016), pp. 1132–1148. DOI: 10.1080/10556788.2016.1180692.

[23] T. Bosse, N. R. Gauger, A. Griewank, S. Günther, et al. "One-Shot Approaches to Design Optimzation". In: *Trends in PDE Constrained Optimization*. Vol. 165. International Series of Numerical Mathematics. Nov. 2014, pp. 43–66. ISBN: 978-3-319-05082-9. DOI: 10.1007/978-3-319-05083-6_5.

[24] T. Bosse, L. Lehmann, and A. Griewank. "Adaptive sequencing of primal, dual, and design steps in simulation based optimization". In: *Computational Optimization and Applications* 57 (2014), pp. 731–760. DOI: 10.1007/s10589-013-9606-z.

[25] D. Braess. *Finite Elements - Theory, Fast Solvers, and Applications in Elasticity Theory*. Third Edition. Cambridge University Press, 2007. ISBN: 978-0-511-27910-2.

[26] J. E. Castillo. *Mathematical Aspects of Numerical Grid Generation*. Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1991. ISBN: 978-0-89871-267-4. DOI: 10.1137/1.9781611971019.

[27] S. Chen and G. D. Doolen. “Lattice Boltzmann Method for Fluid Flows”. In: *Annual Review of Fluid Mechanics* 30.1 (1998), pp. 329–364. DOI: 10.1146/annurev.fluid.30.1.329.

[28] B. Christianson. “Reverse accumulation and attractive fixed points”. In: *Optimization Methods and Software* 3 (1994), pp. 311–326. DOI: 10.1080/10556789408805572.

[29] B. Christianson. “Reverse Accumulation and Implicit Functions”. In: *Optimization Methods and Software* 9.4 (1998), pp. 307–322. DOI: 10.1080/10556789808805697.

[30] B. Cockburn, G. E. Karniadakis, and C.-W. Shu, eds. *Discontinuous Galerkin Methods*. Vol. 11. Lecture Notes in Computational Science and Engineering. Springer, 2000. ISBN: 978-3-642-59721-3. DOI: 10.1007/978-3-642-59721-3.

[31] G. M. Constantine and T. H. Savits. “A Multivariate Faa di Bruno Formula with Applications”. In: *Transactions of the American Mathematical Society* 348.2 (1996), pp. 503–520. DOI: 10.1090/S0002-9947-96-01501-2.

[32] K. Deckelnick, P. J. Herbert, and M. Hinze. “A Novel $W^{1,\infty}$ Approach to Shape Optimisation with Lipschitz Domains”. Preprint SPP1962-162. Mar. 2021. URL: https://spp1962.wias-berlin.de/preprints/162.pdf.

[33] T. Dick, S. Schmidt, and N. R. Gauger. “Combining Sobolev Smoothing with Parameterized Shape Optimization”. In: *Computers & Fluids* 244 (2022). ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2022.105568.

[34] S. L. Dixon. *Fluid Mechanics and Thermodynamics of Turbomachinery*. Fourth Edition. Butterworth-Heinemann, 1998. ISBN: 0-7506-7059-2.

[35] T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, et al. “SU2: An Open-Source Suite for Multiphysics Simulation and Design”. In: *AIAA Journal* 54.3 (2016), pp. 828–846. DOI: 10.2514/1.J053813.

[36] B. Eisfeld. *Die Reynolds-Spannungsgleichungen für kompressible Strömung - Herleitung und Zusammenhänge*. Tech. rep. DLR, 2002. URL: https://elib.dlr.de/19743/.

[37] B. Eisfeld, ed. *Differential Reynolds Stress Modeling for Separating Flows in Industrial Aerodynamics*. Springer Tracts in Mechanical Engineering. Springer, 2015. ISBN: 978-3-319-15639-2. DOI: 10.1007/978-3-319-15639-2.

[38] L. Hernández Encinas and J. Muños Masqué. “A Short Proof of the Generalized Faà di Bruno’s Formula”. In: *Applied Mathematics Letters* 16.6 (2003), pp. 975–979. ISSN: 0893-9659. DOI: 10.1016/S0893-9659(03)90026-7.

[39] I. Faragó and J. Karátson. *Numerical Solution of Nonlinear Elliptic Problems via Preconditioning Operators. Theory and Applications*. Vol. 11. Advances in Computation: Theory and Practice. 2002. ISBN: 978-1-590-33376-1.

[40] C. L. Fefferman. *Existence and smoothness of the Navier-Stokes equation*. Princeton; NJ 08544-1000. 2000. URL: http://www.claymath.org/sites/default/files/navierstokes.pdf.

[41] R. Sánchez Fernández. “A Coupled Adjoint Method for Optimal Design in Fluid-Structure Interaction Problems with Large Displacements”. PhD thesis. Imperial College London, Sept. 2017. DOI: 10.25560/58882.

[42] O. Forster. *Analysis 2*. Vol. 11. Grundkurs Mathematik. Springer Spektrum, Wiesbaden, 1971. ISBN: 978-3-658-19411-6.

[43] N. R. Gauger, A. Griewank, A. Hamdi, C. Kratzenstein, et al. “Automated Extension of Fixed Point PDE Solvers for Optimal Design with Bounded Retardation”. In: *Constrained Optimization and Optimal Control for Partial Differential Equations*. Vol. 160. International Series of Numerical Mathematics. Springer, 2012, pp. 99–122. ISBN: 978-3-0348-0133-1.

[44] C. Geiger and C. Kanzow. *Theorie und Numerik restringierter Optimierungsaufgaben*. Springer-Verlag Berlin Heidelberg, 2002. ISBN: 978-3-642-56004-0.

[45] I. Gherman. “Approximate Partially Reduced SQP Approaches for Aerodynamic Shape Optimization Problems”. PhD thesis. Universität Trier, 2007.

[46] M. Giles and N. Pierce. “An Introduction to the Adjoint Approach to Design”. In: *Flow, Turbulence and Combustion* 65 (2000), pp. 393–415. DOI: 10.1023/A:1011430410075.

[47] A. Griewank. “Who Invented the Reverse Mode of Differentiation”. In: *Documenta Mathematica, Journal der Deutschen Mathematiker-Vereinigung* Extra Volume: Optimization Stories (2012), pp. 301–315. ISSN: 1431-0643. URL: https://www.math.uni-bielefeld.de/documenta/vol-ismp/52_griewank-andreas-b.pdf.

[48] A. Griewank and C. Faure. “Reduced functions, gradients and Hessians from fixed-point iterations for state equations”. In: *Numerical Algorithms* 30 (2002), pp. 113–139.

[49] A. Griewank and A. Walther. *Evaluating Derivatives*. Second Edition. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2008. ISBN: 978-0-898716-59-7. DOI: 10.1137/1.9780898717761.

[50] S. Günther. “Simultaneous Optimization with Unsteady Partial Differential Equations”. PhD thesis. RWTH Aachen, July 2017. DOI: 10.18154/RWTH-2017-06795.

[51] A. Hamdi and A. Griewank. “Properties of an augmented Lagrangian for design optimization”. In: *Optimization Methods and Software* 25 (2010). ISSN: 645-664. DOI: 10.1080/10556780903270910.

[52] A. Hamdi and A. Griewank. “Reduced Quasi-Newton Method for Simultaneous Design and Optimization”. In: *Computational Optimization and Applications* 49 (2011). ISSN: 1573-2894. DOI: 10.1007/s10589-009-9306-x.

[53] L. Hascoët and V. Pascual. “The Tapenade Automatic Differentiation tool: Principles, Model, and Specification”. In: *ACM Transactions on Mathematical Software* 39.3 (2013), 20:1–20:43. ISSN: 0098-3500. DOI: 10.1145/2450153.2450158.

[54] S. B. Hazra and V. Schulz. “Simultaneous Pseudo-Timestepping for Aerodynamic Shape Optimization Problems with State Constraints”. In: *SIAM Journal on Scientific Computing* 28.3 (2006), pp. 1078–1099. DOI: 10.1137/05062442X.

[55] S. B. Hazra, V. Schulz, J. Brezillon, and N. R. Gauger. “Aerodynamic shape optimization using simultaneous pseudo-timestepping”. In: *Journal of Computational Physics* 204.1 (2005), pp. 46–64. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2004.10.007.

[56] H. Heuser. *Funktionalanalysis*. Third Edition. Mathematische Leitfäden. B. G. Teubner Stuttgart, 1992. ISBN: 978-3-519-22206-4.

[57] R. M. Hicks and P. A. Henne. "Wing Design by Numerical Optimization". In: *Journal of Aircraft* 15.7 (1978), pp. 407–412. DOI: `10.2514/3.58379`.

[58] Ø. Hjelle and M. Dæhlen. *Triangulations and Applications*. Mathematics and Visualization. Springer, 2006. ISBN: 978-3-540-33261-9. DOI: `10.1007/3-540-33261-8`.

[59] T. Hytönen, J. van Neerven, M. Veraar, and L. Weis. *Analysis in Banach Spaces. Volume I: Martingales and Littlewood-Paley Theory*. Vol. 63. Ergebnisse der Mathematik und ihrer Grenzgebiete. 3. Folge / A Series of Modern Surveys in Mathematics. 2016. ISBN: 978-3-319-48520-1. DOI: `10.1007/978-3-319-48520-1`.

[60] E.N. Jacobs, K.E. Ward, and R.M. Pinkerton. *The characteristics of 78 related airfoil sections from tests in the variable-density windtunnel*. Tech. rep. National Advisory Committee for Aeronautics, Report No. 460, 1933. URL: `https://ntrs.nasa.gov/citations/19930091108`.

[61] S. Jakirlić, B. Eisfeld, R. Jester-Zürker, and N. Kroll. "Near-wall, Reynolds-stress model calculations of transonic flow configurations relevant to aircraft aerodynamics". In: *International Journal of Heat and Fluid Flow* (2007). DOI: `10.1016/j.ijheatfluidflow.2007.04.001`.

[62] A. Jameson. "Aerodynamic Design via Control Theory". In: *Journal of Scientific Computing* 3 (1988), pp. 233–260. DOI: `10.1007/BF01061285`.

[63] A. Jameson. "Origins and Further Development of the Jameson–Schmidt–Turkel Scheme". In: *AIAA Journal* 55.5 (2017), pp. 1487–1510. DOI: `10.2514/1.J055493`.

[64] A. Jameson. "Transonic airfoil calculations using the euler equations". In: *Numerical Methods in Aeronautical Fluid Dynamics*. Ed. by P. L. Roe. Academic Press, 1982.

[65] A. Jameson, T. J. Baker, and N. P. Weatherhill. "Calculation of Inviscid Transonic Flow over a Complete Aircraft". In: *AIAA 24th Aerospace Sciences Meeting, AIAA 1986-0103* (1986). DOI: `10.2514/6.1986-103`.

[66] A. Jameson, W. Schmidt, and E. Turkel. "Numerical solution of the Euler equations by finite volume methods using Runge-Kutta time-stepping schemes". In: *AIAA Journal* 1259 (1981). DOI: `10.2514/6.1981-1259`.

[67] A. Jameson and E. Turkel. "Implicit schemes and LU-decompositions". In: *Mathematics of Computation* 37 (1981), pp. 385–397. DOI: `10.1090/S0025-5718-1981-0628702-9`.

[68] R. N. King, K. Dykes, P. Graf, and P. E. Hamlington. "Optimization of wind plant layouts using an adjoint approach". In: *Wind Energy Science* 2.1 (2017), pp. 115–131. DOI: `10.5194/wes-2-115-2017`.

[69] D. Kraft. *A software package for sequential quadratic programming*. Tech. rep. Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht, Wiss. Berichtswesen d. DFVLR, 1988.

[70] J. Kusch, N. R. Gauger, S. Müller, and S. Schmidt. "Frequency based preconditioning and smoothing for shape optimization". In: *PAMM* 10 (2016), pp. 701–702. DOI: `10.1002/pamm.201610339`.

[71] J. Kusch, S. Schmidt, and N. R. Gauger. *An Approximate Newton Smoothing Method for Shape Optimization*. July 2018. arXiv: `1807.11232 [math.OC]`.

[72] L. Kusch, T. Albring, A. Walther, and N. R. Gauger. “A one-shot optimization framework with additional equality constraints applied to multi-objective aerodynamic shape optimization”. In: *Optimization Methods and Software* 33.4-6 (2018), pp. 694–707. DOI: `10.1080/10556788.2018.1437158`.

[73] L. D. Landau and E. M. Lifshitz. *Fluid Mechanics*. Second Edition. Vol. 6. Course of Theoretical Physics. 1987. ISBN: 0-08-033933-6.

[74] A. Lintermann. “Computational Meshing for CFD Simulations”. In: *Clinical and Biomedical Engineering in the Human Nose*. Ed. by K. Inthavong, N. Singh, E. Wong, and J. Tu. Biological and Medical Physics, Biomedical Engineering. Springer, 2021. Chap. 6, pp. 85–115. ISBN: 978-981-15-6716-2. DOI: `10.1007/978-981-15-6716-2`.

[75] M. B. Liu, G. R. Liu, and Z. Zong. “An Overview on Smoothed Particle Hydrodynamics”. In: *International Journal of Computational Methods* 5.1 (2008), pp. 135–188. DOI: `10.1142/S021987620800142X`.

[76] H. Lomax, T. H. Pulliam, and D. W. Zingg. *Fundamentals of Computational Fluid Dynamics*. Scientific Computation. Springer, 1999. ISBN: 978-3-662-04654-8.

[77] D. Luenberger and Y. Ye. *Linear and Nonlinear Programming*. Third Edition. International Series in Operations Research and Management Science. Springer, 2008. ISBN: 978-0-387-74502-2.

[78] W. T. Maier, J. T. Needels, C. Garbacz, F. Morgado, et al. “SU2-NEMO: An Open-Source Framework for High-Mach Nonequilibrium Multi-Species Flows”. In: *Aerospace* 8.7 (2021). ISSN: 2226-4310. DOI: `10.3390/aerospace8070193`.

[79] J. H. Manton. *Differential Calculus, Tensor Products and the Importance of Notation*. 2013. arXiv: `1208.0197 [math.HO]`.

[80] J. Martins, P. Sturdza, and J. J. Alonso. “The connection between the complex-step derivative approximation and algorithmic differentiation”. In: *39th Aerospace Sciences Meeting and Exhibit, AIAA 2001-0921* (2002). DOI: `10.2514/6.2001-921`.

[81] J. Mayeur, A. Dumont, D. Destarac, and V. Gleize. “RANS simulations on TMR 3D test cases with the Onera elsA flow solver”. In: *54th AIAA Aerospace Sciences Meeting, AIAA 2016-1357* (2016). DOI: `10.2514/6.2016-1357`.

[82] J. Mayeur, A. Dumont, D. Destarac, and V. Gleize. “RANS simulations on TMR test cases and M6 wing with the Onera elsA flow solver”. In: *53rd AIAA Aerospace Sciences Meeting, AIAA 2015-1745* (2015). DOI: `10.2514/6.2015-1745`.

[83] F. R. Menter. “Two-equation eddy-viscosity turbulence models for engineering applications”. In: *AIAA Journal* 32.8 (1994), pp. 1598–1605. DOI: `10.2514/3.12149`.

[84] B. Mohammadi and O. Pironneau. *Applied Shape Optimization for Fluids*. Second Edition. Numerical Mathematics and Scientific Computation. Oxford University press, 2010. ISBN: 978-0-199-54690-9.

[85] P. M. Müller, N. Kühl, M. Siebenborn, K. Deckelnick, et al. "A Novel p-Harmonic Descent Approach Applied to Fluid Dynamic Shape Optimization". Preprint SPP1962-165. Mar. 2021. URL: `https://spp1962.wias-berlin.de/preprints/165.pdf`.

[86] S. Nadarajah and A. Jameson. "A Comparison of the Continuous and Discrete Adjoint Approach to Automatic Aerodynamic Optimization". In: *38th Aerospace Sciences Meeting and Exhibit, AIAA 2000-0667* (2000). DOI: `10.2514/6.2000-667`.

[87] U. Naumann. *The Art of Differentiating Computer Programs*. Software, Environments and Tools. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2012. ISBN: 978-1-61197-206-1. DOI: `10.1137/1.9781611972078`.

[88] J. W. Neuberger. *Sobolev Gradients and Differential Equations*. Vol. 1670. Lecture Notes in Mathematics. Springer, 2010. ISBN: 978-3-642-04040-5. DOI: `10.1007/978-3-642-04041-2`.

[89] E. J. Nielsen and W. K. Anderson. "Recent Improvements in Aerodynamic Design Optimization on Unstructured Meshes". In: *AIAA Journal* 40.6 (2002). DOI: `10.2514/2.1765`.

[90] J. Nocedal and S. Wright. *Numerical Optimization*. First Edition. Springer Series in Operations Research. Springer New York, 1999. ISBN: 0-387-98793-2.

[91] E. Noether. "Invariante Variationsprobleme". In: *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse* 1918 (1918), pp. 235–257. URL: `https://gdz.sub.uni-goettingen.de/id/PPN252457811_1918`.

[92] E. Özkaya. "One-Shot Methods for Aerodynamic Shape Optimization". PhD thesis. RWTH Aachen, Aug. 2014. URL: `http://publications.rwth-aachen.de/record/444660`.

[93] F. Palacios, M. R. Colonno, A. C. Aranake, A. Campos, et al. "Stanford University Unstructured (SU2): An open-source integrated computational environment for multi-physics simulation and design". In: *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, AIAA 2013-0287* (2013). DOI: `10.2514/6.2013-287`.

[94] F. Palacios, T. D. Economon, A. C. Aranake, S. R. Copeland, et al. "Stanford University Unstructured (SU2): Analysis and Design Technology for Turbulent Flows". In: *52nd Aerospace Sciences Meeting, AIAA 2014-0243* (2014). DOI: `10.2514/6.2014-0243`.

[95] R. Pochampalli, E. Oezkaya, B. Y. Zhou, G. Suarez Martinez, et al. "Machine learning enhancement of Spalart-Allmaras Turbulence Model using Convolutional Neural Network". In: *AIAA Scitech 2021 Forum, AIAA 2021-1017* (2021). DOI: `10.2514/6.2021-1017`.

[96] S. Pope. *Turbulent Flows*. Cambridge University Press, 2000. ISBN: 978-0-511-84053-1. DOI: `10.1017/CBO9780511840531`.

[97] J. Procházková. "Free Form Deformation Methods – The Theory and Practice". In: *Proceedings of the Aplimat conference 2017, Bratislava*. 2017. URL: `https://www.researchgate.net/publication/314261522`.

[98] M. Reed and B. Simon. *Functional Analysis*. Second Edition. Vol. 1. Methods of Modern Mathematical Physics. Academic Press, 1981. ISBN: 978-0-125-85050-6.

[99] M. E. Rognes, D. A. Ham, C. J. Cotter, and A. T. T. McRae. "Automating the solution of PDEs on the sphere and other manifolds in FEniCS 1.2". In: *Geoscientific Model Development* 6.6 (2013), pp. 2099–2119. DOI: `10.5194/gmd-6-2099-2013`.

[100] C. Rumsey. *NASA CFL3D validation test data*. 2014. URL: `https://cfl3d.larc.nasa.gov/Cfl3dv6/cfl3dv6_testcases.html`.

[101] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Second Edition. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2003. ISBN: 0-89871-534-2.

[102] Y. Saad and M. H. Schultz. "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems". In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (1986), pp. 856–869. DOI: `10.1137/0907058`.

[103] M. Sagebaum, T. Albring, and N. R. Gauger. "High-Performance Derivative Computations using CoDiPack". In: *ACM Trans. Math. Softw.* 45.4 (2019). DOI: `10.1145/3356900`.

[104] S. Schmidt. "Efficient Large Scale Aerodynamic Design Based on Shape Calculus". PhD thesis. Universität Trier, Jan. 2010. DOI: `10.25353/ubtr-xxxx-e661-9d13/`.

[105] S. Schmidt. "Weak and Strong Form Shape Hessians and Their Automatic Generation". In: *Journal of Scientific Computing* 40.2 (2018), pp. C210–C233. DOI: `10.1137/16M1099972`.

[106] S. Schmidt, C. Ilic, V. Schulz, and N. R. Gauger. "Three-Dimensional Large-Scale Aerodynamic Shape Optimization Based on Shape Calculus". In: *AIAA journal* 51.11 (2013), pp. 2615–2627. DOI: `10.2514/1.J052245`.

[107] W. Schröder. *Fluidmechanik*. Vol. 7. Aachener Beiträge zur Strömungsmechanik. Wissenschaftsverlag Mainz in Aachen, 2004. ISBN: 3-86130-371-X.

[108] V. Schulz. "Reduced SQP Methods for Large-Scale Optimal Control Problems in DAE with Application to Path Planning Problems for Satellite Mounted Robots". PhD thesis. Universität Heidelberg, Feb. 1996.

[109] V. Schulz and I. Gherman. "One-Shot Methods for Aerodynamic Shape Optimization". In: *MEGADESIGN and MegaOpt – German Initiatives for Aerodynamic Simulation and Optimization in Aircraft Design*. Ed. by N. Kroll, D. Schwamborn, K. Becker, H. Rieger, et al. Vol. 107. Notes on Numerical Fluid Mechanics and Multidisciplinary Design. Springer, 2007, pp. 207–220.

[110] V. Schulz and M. Siebenborn. "Computational Comparison of Surface Metrics for PDE Constrained Shape Optimization". In: *Computational Methods in Applied Mathematics* 16.3 (2016), pp. 485–496. DOI: `10.1515/cmam-2016-0009`.

[111] T. W. Sederberg and S. R. Parry. "Free-Form Deformation of Solid Geometric Models". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. Vol. 20. ACM Press., 1986, pp. 151–160.

[112] J. Shah and M. Mäntylä. *Parametric and Feature-Based CAD/CAM: Concepts, Techniques, and Applications*. John Wiley and Sons, Inc., 1995. ISBN: 978-0-471-00214-7.

[113] C. Simmendinger, J. Jägersküpper, R. Machado, and C. Lojewski. "A PGAS-based Implementation for the Unstructured CFD Solver TAU". In: *5th Conference on Partitioned Global Address Space Programming Models*. Jan. 2011. URL: `https://elib.dlr.de/72100/`.

[114] M. Skorski. *Chain Rules for Hessian and Higher Derivatives Made Easy by Tensor Calculus*. University of Luxembourg. 2019. arXiv: `1911.13292 [cs.SC]`.

[115] J. W. Slater. *ONERA M6 Wing: Study no. 1*. 2002. URL: https://www.grc.nasa.gov/WWW/wind/valid/m6wing/m6wing01/m6wing01.html.

[116] J. Sokolowski and J.-P. Zolesio. *Introduction to Shape Optimization*. Springer Series in Computational Mathematics. Springer Verlag Berlin Heidelberg GmbH, 1992. ISBN: 978-3-642-63471-0.

[117] P. Spalart and S. Allmaras. "A one-equation turbulence model for aerodynamic flows". In: *30th Aerospace Sciences Meeting and Exhibit*. AIAA, 1992. DOI: 10.2514/6.1992-439.

[118] A. H. Stroud. *Approximate Calculation of Multiple Integrals*. Prentice-Hall Inc., 1971.

[119] S. Ta'asan. *"One shot" methods for optimal control of distributed parameter systems 1: Finite dimensional control*. Tech. rep. ICASE 91-2. NASA Langley Research Center, 1991. URL: https://ntrs.nasa.gov/citations/19910008345.

[120] S. Ta'asan. *Pseudo-Time Methods for Constrained Optimization Problems Governed by PDE*. Tech. rep. ICASE 95-32. 1995. URL: https://ntrs.nasa.gov/citations/19950024699.

[121] R. Van den Braembussche. *Design and Analysis of Centrifugal Compressors*. ASME Press and John Wiley & Sons Ltd, 2019. ISBN: 978-1-119-42409-3.

[122] L. Vandenberghe. *The CVXOPT linear and quadratic cone program solvers*. 2010. URL: http://www.seas.ucla.edu/~vandenbe/publications/coneprog.pdf.

[123] P. Virtanen, R. Gommers, T. Oliphant, M. Haberland, et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2. URL: https://rdcu.be/b08Wh.

[124] A. Walther, N. R. Gauger, L. Kusch, and N. Richert. "On an extension of one-shot methods to incorporate additional constraints". In: *Optimization Methods and Software* 31.3 (2016), pp. 494–510. DOI: 10.1080/10556788.2016.1146268.

[125] D. Wilcox. *Turbulence Modeling for CFD*. Third Edition. DCW Industries, Inc., 2006. ISBN: 978-1-928-72908-2.

[126] B. Y. Zhou, T. Albring, N. R. Gauger, C. R. da Silva, et al. "Efficient Airframe Noise Reduction Framework via Adjoint-Based Shape Optimization". In: *AIAA Journal* 59.2 (2021), pp. 580–595. DOI: 10.2514/1.J058917.

List of Figures

List of Tables

Appendices

Curriculum Vitae

Personal Details

Name: Thomas Karl Oskar Dick

Education

1997 - 2001	Grundschule Ostschule, Neustadt an der Weinstraße, Germany
2001 - 2010	Käthe-Kollwitz-Gymnasium, Neustadt an der Weinstraße, Germany
2011 - 2014	B.Sc. studies Mathematics, Technische Universität Kaiserslautern, Kaiserslautern, Germany
2014 - 2016	M.Sc. studies Technomathematik, Technische Universität Kaiserslautern, Kaiserslautern, Germany
since 2017	Ph.D. studies, Technische Universität Kaiserslautern, Kaiserslautern, Germany

Work Experience

2010 - 2011	Zivildienst, Deutsches Rotes Kreuz, Kreisverband Vorderpfalz
2013 - 2016	Student assistant, Mathematics Department, Technische Universität Kaiserslautern, Kaiserslautern, Germany
2017 - 2021	Research assistant, Chair for Scientific Computing, Technische Universität Kaiserslautern, Kaiserslautern, Germany

Lebenslauf

Persönliche Details

Name:	Thomas Karl Oskar Dick

Ausbildung

1997 - 2001	Grundschule Ostschule, Neustadt an der Weinstraße, Deutschland
2001 - 2010	Käthe-Kollwitz-Gymnasium, Neustadt an der Weinstraße, Deutschland
2011 - 2014	Studiengang mit Abschluss B.Sc. Mathematik, Technische Universität Kaiserslautern, Kaiserslautern, Deutschland
2014 - 2016	Studiengang mit Abschluss M.Sc. Technomathematik, Technische Universität Kaiserslautern, Kaiserslautern, Deutschland
seit 2017	Promotionsstudium, Technische Universität Kaiserslautern, Kaiserslautern, Deutschland

Arbeitserfahrung

2010 - 2011	Zivildienst, Deutsches Rotes Kreuz, Kreisverband Vorderpfalz
2013 - 2016	Wissenschaftliche Hilfskraft, Fachbereich Mathematik, Technische Universität Kaiserslautern, Kaiserslautern, Deutschland
2017 - 2021	Wissenschaftlicher Mitarbeiter, AG Scientific Computing, Technische Universität Kaiserslautern, Kaiserslautern, Deutschland

List of Publications

Research Articles

T. Albring, T. Dick, and N. R. Gauger. “Assessment of the Recursive Projection Method for the Stabilization of Discrete Adjoint Solvers”. In: *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA 2017-3664* (2017). DOI: `10.2514/6.2017-3664`.

T. Dick, S. Schmidt, and N. R. Gauger. “Combining Sobolev Smoothing with Parameterized Shape Optimization”. In: *Computers & Fluids* 244 (2022). ISSN: 0045-7930. DOI: `10.1016/j.compfluid.2022.105568`.

Conference Presentations

T. Dick, N. R. Gauger, and S. Schmidt. *Development of a gradient smoothing method for shape optimization in SU2*. DMV Anual Meeting 2020 Chemnitz. Sept. 2020. URL: `https://www.tu-chemnitz.de/mathematik/dmv2020/index.php`.

T. Dick and M. Sagebaum. *Differentiation of OpenMP parallel programs with CoDiPack*. 21. Euro AD Workshop Jena. Nov. 2018. URL: `http://www.autodiff.org/?module=Workshops&submenu=EuroAD%2F21%2Fprogramme`.