Aerodynamic Shape Optimization based on the One-Shot Approach

Master’s Thesis
Tim-Adrian Albring

Supervisors:
Prof. Dr. Nicolas Gauger
Dipl.-Math. Max Sagebaum

Aachen, January 6, 2014
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Aachen, January 6, 2014

Tim-Adrian Albring
# Contents

0 Introduction 1

1 Numerical Solution of the Aerodynamic Flow Equations 4

1.1 Flow Equations 5
  1.1.1 Navier-Stokes Equations 8
  1.1.2 Euler Equations 8

1.2 Aerodynamic Coefficients 9

1.3 Finite-Volume Discretization of Conservation Laws 9
  1.3.1 Space Integration 10
  1.3.2 Time Integration 13

2 Theory of One-Shot Shape Optimization 15

2.1 Problem Statement and Parametrization 15

2.2 Approximated Reduced Sequential Quadratic Programming 16
  2.2.1 Karush-Kuhn-Tucker System 16
  2.2.2 Null Space Ansatz and Variable Elimination Basis 18
  2.2.3 Approximate Null Space Iteration 20
  2.2.4 Notes on Convergence 21

2.3 Reduced Quasi-Newton Method 22
  2.3.1 Karush-Kuhn-Tucker System based on the Shifted Lagrangian 22
  2.3.2 Piggy-Back Iteration 23
  2.3.3 Conditions on the Design Preconditioner 23
  2.3.4 Descent on the Exact Penalty Function 24
  2.3.5 Explicit Choice of $B$ 25
  2.3.6 Notes on Convergence 26

3 Numerical Aspects in Shape Optimization 27

3.1 Hessian Approximation by the Laplace-Beltrami Operator 27

3.2 Additional Constraints 29

3.3 Mesh Deformation 30

4 Algorithmic Differentiation and Implementation 33

4.1 Introduction to Algorithmic Differentiation 33
  4.1.1 Forward Propagation 33
  4.1.2 Reverse Propagation 34
  4.1.3 Practical Implementation 37

4.2 Application to SU2 39
  4.2.1 Integration into the existing Framework 40
  4.2.2 Comment on Performance 41

5 Test Cases 43
Contents

5.1 Transonic NACA0012
   5.1.1 Physical and Numerical Definition
   5.1.2 Validation of the Reduced Gradient
   5.1.3 Direct and Adjoint Solution for the initial Design
   5.1.4 Optimization

5.2 Supersonic NACA0012
   5.2.1 Physical and Numerical Definition
   5.2.2 Optimization

6 Conclusion and Outlook

Bibliography
# List of Figures

0.1 Structural differences between the two design approaches .................................. 3

1.1 Relation between the different flow models ....................................................... 4
1.2 Aerodynamic Forces on the Airfoil defined through the surface $S_0$ ..................... 9
1.3 Cell vertex approach using a median-dual scheme ............................................. 10

3.1 Tangential shape coordinate $\xi$ ................................................................. 29
3.2 Infeasibility $|h(u_k)|$ for the Quadratic penalty method and the Penalty multiplier method using a volume constraint with $\mu = 100$. .................................................. 31
3.3 Initial (black) and deformed (red) mesh after optimization. ................................. 32

4.1 The class hierarchy of $SU^2$ (taken from Palacios et al. [26]). .......................... 41

5.1 Discretization of the physical domain. .......................................................... 44
5.2 Comparison of the components in $x_2$-direction of the reduced gradient with finite differences. ................................................................. 45
5.3 Convergence of the coupled iteration for the initial design. ............................... 46
5.4 Contour plot of the adjoint density. ............................................................. 46
5.5 Shape gradient $g$ for the initial design ......................................................... 47
5.6 Optimization history using different values of $\varepsilon$ for the transonic test case. .... 48
5.7 Pressure $p$ for the transonic test case. .......................................................... 49
5.8 Pressure Coefficient $C_p$ and design $u$ for the initial and optimized case. ............ 50
5.9 Optimization history using $\varepsilon = 0.1$ for the supersonic test case. .................. 51
5.10 Density $\rho$ for the supersonic test case. ...................................................... 52
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>General evaluation procedure.</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Forward propagation of gradients.</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Reverse propagation of gradients.</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Average time for one iteration and memory consumption.</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Free-stream conditions for the transonic test case.</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Final values of the drag coefficient, the 2-norm of the reduced gradient and the volume for the transonic test case.</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Free-stream conditions for the supersonic test case.</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>Final values of the drag coefficient, the 2-norm of the reduced gradient and the volume for the supersonic test case.</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Tables
The problem of finding the optimal shape of an airfoil was first studied by Lighthill in 1945, who solved it for the case of two-dimensional incompressible flow by conformally mapping the profile to a unit circle [21]. In the next 30 years this approach was further extended to compressible and three-dimensional flows. Finally, in 1978 Hicks and Henne [13] have explored the possibility of meeting desired design objectives by using constrained optimization. They specified a set of parameters defining the configuration, and any computer program for flow analysis is used to evaluate the aerodynamic characteristics. Then an optimization method selects values of these parameters that minimizes some criterion, like drag for example, subject so some other constraints like wing thickness and volume. In principle this method is rather flexible, allowing the designer to choose any reasonable design objectives. However, it becomes very expensive if the number of design parameters is increased because the design gradient is calculated using finite differences and requires a flow solution for each of the parameters.

The next major step are the adjoint methods for design that were introduced by Mohammadi and Pironneau [24] for fluid dynamics. But the application in aeronautical computational fluid dynamics was pioneered by Jameson [16]. He applied the theory of optimal control to the flow equations to get a formulation of the gradient that requires only the solution of the flow equation and the adjoint equation, thereby making it independent of the number of design parameters. Although the continuous version, based on the continuous integral formulation of the flow equations, is quiet efficient in terms of memory consumption and run-time, it is difficult to extend the approach to handle additional equations like turbulence or transition models, for example. Furthermore there is a slight inconsistency between the calculated gradient and the objective function because the method yields a discrete approximation of the gradient of the analytic objective function, rather than of the discretized objective function. As a result the optimization is likely to fail to converge further once we are near a local minimum. Nevertheless, this approach is used frequently nowadays for shape optimization problems. A possible remedy for the latter problem is the use of a discrete adjoint method where the optimal control theory is applied to the discretized flow equations. Still, it suffers the same drawback, namely the difficult extension to complex flow models if the traditional approach based on the transposed of the state Jacobian is deployed. More on the advantages and disadvantages of either of the methods can be found in the paper of Giles and Pierce [4].

Both adjoint methods mentioned above implicitly assume that the state equations, and hence the state variables, are eliminated at each design iteration by performing a complete analysis for a given design. Such methods are often referred to as Nested Analysis and Design (NAND) because the state and adjoint solver are nested inside the design loop (see figure 0.1a). Since the analysis in aerodynamic shape optimization is usually done by a solution of the flow equations, the requirement of completely resolving the state for a particular design is highly onerous. The alternative are the Simultaneous Analysis and Design (SAND)-methods. Here the state equations are retained as equality constraints in the optimization problem. The state variables are now regarded as optimization variables, and the state equations are satisfied asymptotically, as,
simultaneously, both the design variables converge to their optimal values and the state variables converge to values consistent with the optimal design (see figure 0.1b). Although known for more than twenty years, this approach is used quite infrequently in practice because it apparently introduces new difficulties. First, it requires that the flow solver be embedded in the optimization solver. Thus, the use of existing codes is prevented. Second, although the degrees of freedom are unchanged (the dimension of the reduced gradient remains the same as in the NAND case) the constraint Jacobian and Lagrangian Hessian matrices become very large and sparse and are often not easy to access. In the remainder of this work we will derive the One-Shot method along the lines of Ito et al. [15] and Hamdi and Griewank [9] based on the approximate solution of the reduced Karush-Kuhn-Tucker system in each iteration that do not demand Newton’s method for the solution of the constraint equation. Therefore, we can avoid the previously stated disadvantages of SAND methods. In combination with Algorithmic Differentiation for the calculation of the derivatives it is possibly to use the existing sophisticated open-source flow solver Stanford University Unstructured (SU2) for the solution of the state equation and thereby augment the already established framework to potentially enable shape optimization in flows described by complex models. Furthermore, because the resulting optimization environment will feature as a foundation for the development of efficient preconditioners for non-parametric design methods we will use a special preconditioner based on the Hessian approximating nature of the Laplace-Beltrami operator that shows great promise for further acceleration of the One-Shot method.

First, in chapter 1 a short introduction on the physical flow modeling using the Navier-Stokes and Euler equations will be given. Then the numerical solution of such problems with the Finite-Volume method is explained, with special emphasis on the main algorithms used in the SU2 code. Chapter 2 features the theoretical foundation of the One-Shot method and short comments on the convergence. Chapter 3 is devoted to some additional numerical aspects that are needed for the implementation, especially the definition of a practical design preconditioner. In chapter 4 an introduction to Algorithmic Differentiation is given and its application to the SU2 code will be explained shortly. Finally, in chapter 5 some results for the optimization in inviscid flows are presented.
Introduction

One Step of State Iteration

One Step of Adjoint Iteration

One Step of Design Iteration

Until Convergence

(a) Nested Analysis and Design

(b) Simultaneous Analysis and Design

Figure 0.1: Structural differences between the two design approaches
In aerodynamics the flow around an object is described by means of application of conservation laws to the three fundamental quantities, mass, momentum and energy. The resulting system of partial differential equations is known as the Navier-Stokes equations. Each of these equations is non-linear which has major consequences on the whole of fluid mechanics. It is responsible for the appearance of turbulence, which is a spontaneous instability of the flow, and it leads to the existence of shock-waves in supersonic flows. The numerical description of all flow phenomena is a formidable task as all time and length scales must be resolved. Therefore, there exist various levels of approximations and simplifications to the Navier-Stokes equations that reduce the high demands on computer resources. Figure 1.1 shows the relation between

![Diagram](image)

**Figure 1.1: Relation between the different flow models**

the most common models used in practice. Laminar flows can be accurately duplicated by computations without any additional information. However, to describe turbulent flows an extremely fine resolution is required which forms the basis of Direct Numerical Simulation. In Large Eddy Simulation the objective is again to directly describe the turbulent fluctuations, but restricted to larger scales with the small scales
being modeled. If the complete influence of turbulence is described by an appropriate model we arrive at the *Reynolds Averaged Navier-Stokes equations*. At high Reynolds numbers the influence of the viscous and turbulent shear stresses is confined in small regions close to the walls and outside these layers the flow behaves as inviscid. Hence, the *Euler equations* often allow a good approximation of the pressure field.

In this chapter we will introduce the Navier-Stokes equations and the Euler equations using the mathematical description of a conservation law. The derivation is primarily based on the introductory book from Hirsch [14]. Furthermore, the Finite Volume method will be introduced to discretize and finally solve the resulting numerical approximation.

### 1.1 Flow Equations

Many physical processes in nature, in particular the flow of fluids, can be described using a conservation principle. Consider for example the quantity $U(x,t)$ in a volume $\Omega \subset \mathbb{R}^d$, where $d$ is the dimension of the physical space. A conservation law describes, how the quantity $\int_{\Omega} U(x,t) dx$ changes with time. More specific conservation means that the variation of a conserved quantity within a given volume $\Omega$ is due to the amount of that quantity which is crossing the boundary surface. Furthermore there may be changes due to sources or sinks in the volume. A rigorous formulation of a scalar and vector conservation law is given by

**Definition 1.1 (Conservation law)**

Let $\Omega \subset \mathbb{R}^d$, $x \in \Omega$ and $t \in \mathbb{R}_+$. If $U = U(x,t) \in \mathbb{R}$, then $U$ satisfies a scalar conservation law, if there is a function $F = F(U) \in \mathbb{R}^d$ such that

$$\frac{\partial}{\partial t} \int_{\Omega} U \, dx = - \int_{\partial \Omega} F(U)^T \vec{n} dS + \int_{\Omega} Q_v dx + \int_{\partial \Omega} Q_s^T \vec{n} dS,$$

(1.1)

where $\vec{n} = \vec{n}(x) \in \mathbb{R}^d$ is the outward normal for $x \in \partial \Omega$. $U$ is then called the conserved quantity and $F$ flux or flux-function. $Q_v = Q_v(x,t) \in \mathbb{R}$ and $Q_s = Q_s(x,t) \in \mathbb{R}^d$ are volume and surface sources, respectively.

If $U = U(x,t) \in \mathbb{R}^d$, then $U$ satisfies a vector conservation law, if there is a function $F = F(U) \in \mathbb{R}^{d \times d}$ such that

$$\frac{\partial}{\partial t} \int_{\Omega} U \, dx = - \int_{\partial \Omega} F(U) \vec{n} dS + \int_{\Omega} Q_v dx + \int_{\partial \Omega} Q_s \vec{n} dS,$$

(1.2)

$U$ is then called the vector of conserved quantity and $F$ flux or flux-function. $Q_v = Q_v(x,t) \in \mathbb{R}^d$ and $Q_s = Q_s(x,t) \in \mathbb{R}^{d \times d}$ are volume and surface sources, respectively.

If we use integration by parts on equation (1.1) we get

$$\int_{\Omega} \frac{\partial U}{\partial t} + \text{div}(F(U)) dx - \int_{\Omega} Q_v dx - \int_{\partial \Omega} \text{div}(Q_s) dx = 0$$

(1.3)

and thereby the point-wise (or differential) form of the conservation law:

$$\frac{\partial U}{\partial t} + \text{div}(F(U)) - Q_v = 0$$

(1.4)

We get the same result if integration by parts is applied to the vector conservation law (1.2) and we define the divergence operator for a matrix $A(x) \in \mathbb{R}^3$ by

$$\text{div}(A(x)) = \begin{pmatrix} \text{div}(a_{11}) & a_{12} & a_{13} \\ \text{div}(a_{21}) & a_{22} & a_{23} \\ \text{div}(a_{31}) & a_{32} & a_{33} \end{pmatrix} \in \mathbb{R}^3.$$

(1.5)

Bear in mind that formulation (1.4) only makes sense if $U$, $F$, and $Q_s$ are differentiable, which is not the case in presence of shock waves that originate from non-linearities, for example. In contrast to that, the
1 Numerical Solution of the Aerodynamic Flow Equations

integral form (1.1) only needs integrability of the functions which is a much weaker requirement. Nevertheless both formulations can be used for the development of numerical methods.

Up to now we considered the abstract mathematical description of a conservation law and we have not yet provided any specific information about the fluxes. In the case of fluid flow the fluxes consist of contributions from two different physical processes, namely convection and diffusion. Convection describes the transport of a quantity by the fluids velocity whereas diffusion is defined as a flux that is present at fluids at rest, due to the macroscopic effect of the molecular thermal agitation. The convective flux is defined as the dyadic product of the conserved quantity and the fluid velocity \( \vec{v}(x,t) \in \mathbb{R}^d \):

\[
F_c(U) = U\vec{v}^T. \tag{1.6}
\]

It represents the amount of \( U \) that is transported by the flow. The diffusive flux can be described by Fick’s first law of diffusion which relates the flux to the concentration. It postulates that the flux goes from regions of high concentration to low concentration, with a magnitude that is proportional to the concentration gradient. It can be written as

\[
F_d(U) = -\rho D \nabla u, \tag{1.7}
\]

where \( D \in \mathbb{R} \) is the diffusivity coefficient and \( u = U/\rho \). The ratio between the diffusive flux and the convective flux can be characterized by the Peclet number \( Pe \):

\[
Pe = \frac{VL}{D} \tag{1.8}
\]

where \( V, L \in \mathbb{R} \) are a reference velocity and a reference length, respectively. Hence, if this ratio is larger than one, the evolution of the quantity \( U \) will be dominated by convection, while it will be dominated by diffusion when the Peclet number is lower than one.

The motion of a fluid is completely described by the conservation laws for the three basic properties mass, momentum and energy. The conservation of mass leads to the continuity equation, the conservation of momentum is a generalization of Newton’s second law and the energy equation represents the first principle of Thermodynamics.

**Conservation of Mass**

The conservation of mass describes the intuitive law, that mass can neither be created nor destroyed and thus is conserved over time. With the absence of chemical reactions, the specific mass or density \( \rho(x,t) \in \mathbb{R}^d \) can only change due to convection. Thus with \( U \equiv \rho \), the convective flux and without source terms we get the following integral mass conservation equation from the general form of a scalar conservation law (1.1):

\[
\frac{\partial}{\partial t} \int_{\Omega} \rho dx + \int_{\partial \Omega} (\rho \vec{v})^T \vec{n} dx = 0. \tag{1.9}
\]

The pointwise formulation of equation (1.9) can be written as the following partial differential equation:

\[
\frac{\partial \rho}{\partial t} + \text{div}(\rho \vec{v}) = 0 \text{ on } \Omega \times \mathbb{R}_+. \tag{1.10}
\]

Equation (1.10) is also known as continuity equation.

**Conservation of Momentum**

Momentum is defined as the product of the (specific) mass and the velocity vector. Thus we have \( U \equiv \rho \vec{v} \in \mathbb{R}^d \). As with mass conservation there is no diffusive contribution to the flux. From Newton’s second law in mechanics we know that the momentum changes due to the forces that apply. For fluid motion we can distinguish between external forces, like gravity, and an internal force. The latter is a friction force that a fluid exerts on itself and it acts on the boundary of the volume \( \Omega \) since they cancel two per two in every point inside the volume. Cauchy derived a law, also called Cauchy’s stress theorem, stating that this
internal force is a linear function of the normal vector \( \vec{n} \). Thus, if we neglect external forces, there is a matrix \( \sigma = \sigma(x, t) \in \mathbb{R}^{d \times d} \), called stress tensor, such that \( Q_s = \sigma \) in equation (1.2). Therefore with (1.6) we get the following equation for the conservation of momentum:

\[
\frac{\partial}{\partial t} \int_\Omega \rho \vec{v} dx + \int_{\partial \Omega} \vec{v} (\rho \vec{v})^T \vec{n} dS - \int_{\partial \Omega} \sigma \vec{n} dS = 0.
\]  (1.11)

For a Newtonian fluid, the total internal stress tensor \( \sigma \) is taken to be

\[
\sigma = -pI + \tau
\]  (1.12)

where \( p = p(x, t) \in \mathbb{R} \) is the isentropic pressure and \( \tau = \tau(x, t) \in \mathbb{R}^{d \times d} \) is the viscous shear stress tensor. With the assumption that the fluid is in local thermodynamic equilibrium \( \tau \) is defined by

\[
\tau_{ij} = \mu \left[ \left( \frac{\partial v_j}{\partial x_i} + \frac{\partial v_i}{\partial x_j} \right) - \frac{2}{3} \text{div}(\vec{v}) \delta_{ij} \right],
\]  (1.13)

with \( \mu \in \mathbb{R} \) as the dynamic viscosity of the fluid. When the stress tensor (1.12) is introduced in equation (1.11) and by using the point-wise formulation (1.4) we obtain

\[
\frac{\partial (\rho \vec{v})}{\partial t} + \text{div}((\rho \vec{v}) \vec{v}^T + pI - \tau) = 0, \text{ on } \Omega \times \mathbb{R}^+.\]  (1.14)

Note, that although we assumed that there is no diffusive flux, the viscous term \( \text{div} \tau \) contains second order derivatives of the velocity and acts as a diffusion with the kinematic viscosity \( \nu = \mu/\rho \) as diffusion coefficient. Therefore we can introduce a particular form of the Peclet number (1.8) that describes the ratio between momentum convection and diffusion. This ratio is known as the Reynolds number \( \text{Re} \):

\[
\text{Re} = \frac{VL}{\nu}.\]  (1.15)

In practical applications, coming from similitude theory, the Reynolds number is an important characterization of the structure of the flow.

**Conservation of Energy**

In thermodynamics the energy content of a system is measured by its internal energy per unit mass \( e = e(x, t) \in \mathbb{R} \). The conserved quantity in a fluid is \( U = \rho E \) where \( E = E(x, t) \in \mathbb{R} \) is the total energy defined as the sum of its internal energy and its kinetic energy per unit mass \( ||\vec{v}||^2/2 \):

\[
E = e + \frac{||\vec{v}||^2}{2}.
\]  (1.16)

In addition to a convective flux there is now also a diffusive flux. By using (1.7) and noting that there is no diffusive flux associated with motion it can be written as

\[
F_d = -\rho \gamma k \nabla e = -k \nabla T
\]  (1.17)

where \( T = T(x, t) \in \mathbb{R} \) is the absolute temperature. The last equality can easily be derived from basic thermodynamic relations. \( \gamma \in \mathbb{R} \) is the ratio of specific heat coefficients under constant pressure and constant volume \( \gamma = c_p/c_v \) and \( k, \kappa \in \mathbb{R} \) are the thermal diffusivity and thermal conductivity coefficients, respectively. If we assume an adiabatic system (i.e. no heat fluxes through the surfaces) the first law of thermodynamics states that the variation of the total energy is due to the work of the forces that are acting on it. In the derivation of the conservation of momentum we identified the stress tensor as the cause of an internal force. From classical mechanics we know that the rate of work done by a force is defined as the scalar product of the force, \( \sigma \vec{n} \) in this case, and the velocity vector \( \vec{v} \). Using the generic scalar conservation law (1.1) and the definition of the stress tensor we then get the following surface source \( Q_s \):

\[
Q_s = \sigma \vec{v} = -\rho \vec{v} + \tau \vec{v}
\]  (1.18)
Summarizing all the contributions the integral form of the equation for the conservation of energy becomes
\[
\frac{\partial}{\partial t} \int_{\Omega} \rho E \, dx + \int_{\partial \Omega} (\rho E) \vec{v}^T \vec{n} \, dS - \int_{\partial \Omega} k (\nabla T)^T \vec{n} \, dS + \int_{\partial \Omega} (\sigma \vec{v})^T \vec{n} \, dS = 0.
\] (1.19)

And, again, after transformation of the surface integrals we get for the differential form
\[
\frac{\partial (\rho E)}{\partial t} + \text{div}(\rho \vec{v} E - k \nabla T - p \vec{v} + \tau \vec{v}) = 0, \quad \text{on } \Omega \times \mathbb{R}_+.
\] (1.20)

Additionally, from thermodynamics we get a relation between the internal energy \(e\) and the other thermodynamic variables. For a perfect gas we have for example
\[
e = \frac{1}{\gamma - 1} \frac{p}{\rho}. \tag{1.21}
\]

Furthermore the viscosity coefficient \(\mu\) is strongly influenced by temperature. For gases a commonly used relation is Sutherland’s law:
\[
\mu = \frac{1.45 T^\frac{2}{3}}{T + 110} \times 10^{-6}. \tag{1.22}
\]

### 1.1.1 Navier-Stokes Equations

The Navier-Stokes equations are the most complete description of fluid flows. They consist of the conservation equations for the three flow quantities \(U = (\rho, \rho \vec{v}, \rho E)^T\) derived in the previous section. They form a system of five (in three-dimensional space) fully coupled time-dependent partial differential equations for the five unknowns, velocity vector and two thermodynamic quantities. But since we will use a Finite-Volume approach for the discretization later on, it is more convenient to use the integral form by combining the equations (1.9), (1.11) and (1.19):
\[
\frac{\partial}{\partial t} \int_{\Omega} U \, dx + \int_{\partial \Omega} (F^{\text{inv}}(U) + F^{\text{vis}}(U)) \vec{n} \, dS = 0 \tag{1.23}
\]

where
\[
F^{\text{inv}}(U) := \begin{pmatrix}
(p \vec{v})^T \\
\rho \vec{v}^T \vec{v}^T - pI \\
(\rho E) \vec{v}^T - (\rho \vec{v})^T
\end{pmatrix}, \quad F^{\text{vis}}(U) := \begin{pmatrix}
0 \\
0 \\
(\tau \vec{v})^T - k (\nabla T)^T
\end{pmatrix} \tag{1.24}
\]

are the inviscid (convective plus surface source) and viscous (diffusive) fluxes, respectively.

### 1.1.2 Euler Equations

By neglecting all shear stresses and heat conducting terms in the Navier-Stokes equations we get the Euler equations. They describe non-viscous, non-heat conducting flows and are especially valid for flows at high Reynolds numbers outside of viscous regions developing near solid surfaces. If we drop \(F^{\text{visc}}\) in equation (1.23) we get for the Euler equations
\[
\frac{\partial}{\partial t} \int_{\Omega} U \, dx + \int_{\partial \Omega} F^{\text{inv}}(U) \vec{n} \, dS = 0, \tag{1.25}
\]

where \(F^{\text{inv}}(U)\) is still defined as in (1.24).
1 Numerical Solution of the Aerodynamic Flow Equations

1.2 Aerodynamic Coefficients

In external flows an important task is the prediction of the forces acting on a body. Especially in aerodynamics lift and drag coefficients usually determine the performance of an airfoil. If the volume $\Omega$ contains a solid body then an additional external force $-R \in \mathbb{R}^d$ has to be added to the right-hand side of equation (1.11). Since $\Omega$ can be arbitrary we choose it in such a way that $\partial \Omega$ coincides with the surface $S_b$ representing the solid body. Hence, for stationary flows and using the no-slip condition for viscous flows we get

$$ - \int_{S_b} p n dS + \int_{S_b} \tau n dS = R = L + D \quad (1.26) $$

where $L \in \mathbb{R}^d$ and $D \in \mathbb{R}^d$ are the lift and drag forces, respectively. For inviscid flows the second term on the left-hand side vanishes. In aerodynamics $L$ represents the component of $R$ that is normal to the far-field velocity vector $\vec{v}_{\infty}$ whereas $D$ is the component parallel to that vector (see figure 1.2). If we take $\vec{v}_{\infty} = (\cos(\alpha), \sin(\alpha))^T \|\vec{v}_{\infty}\|_2$, where $\alpha$ is the angle between $\vec{v}_{\infty}$ and the $x$-axis, we get by scalar projection of $R$ onto the corresponding directions for two dimensions

$$ F_D := R^T \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \end{pmatrix} = - \int_{S_b} p(\cos(\alpha)n_x + \sin(\alpha)n_y) dS + \int_{S_b} (\cos(\alpha)\tau_{wx} + \sin(\alpha)\tau_{wy}) dS \quad (1.27) $$

and

$$ F_L := R^T \begin{pmatrix} -\sin(\alpha) \\ \cos(\alpha) \end{pmatrix} = - \int_{S_b} p(\cos(\alpha)n_y - \sin(\alpha)n_x) dS + \int_{S_b} (\cos(\alpha)\tau_{wy} - \sin(\alpha)\tau_{wx}) dS \quad (1.28) $$

with $\tau_w := \tau n$. The drag and lift coefficients are then defined as

$$ C_D := \frac{F_D}{\frac{1}{2} \rho_{\infty} \|\vec{v}_{\infty}\|_2^2 A} \quad (1.29) $$

$$ C_L := \frac{F_L}{\frac{1}{2} \rho_{\infty} \|\vec{v}_{\infty}\|_2^2 A} \quad (1.30) $$

where $A$ is the length or planform area of the airfoil and the index $\infty$ denotes far-field values. Furthermore, the pressure coefficient $C_p$ is defined as

$$ C_p = \frac{p - p_{\infty}}{\frac{1}{2} \rho_{\infty} \|\vec{v}_{\infty}\|_2^2}. \quad (1.31) $$

Figure 1.2: Aerodynamic Forces on the Airfoil defined through the surface $S_b$.

1.3 Finite-Volume Discretization of Conservation Laws

For the numerical solution of conservation laws the Finite Volume-Method (FVM) has become the most widely applied method today. The reasons for this are its generality, the conceptual simplicity and its ease
1 Numerical Solution of the Aerodynamic Flow Equations

of implementation on arbitrary grids. The method is based on the integral formulation of the conservation law that is directly discretized in physical space. In contrast to Finite-Difference-Methods or Finite-Element-Methods, where the main numerical quantities are the local function values, for the FVM the evolution of cell-averaged values is considered. Once a grid has been generated we associate a finite volume, or control volume, to each mesh point and apply the integral conservation law to this local volume. This immediately gives the essential advantage of the FVM, namely that the conservation property is automatically satisfied. In the following a description of the FVM method as well as the discretization scheme that is used in the Stanford University Unstructured (SU^2) code [26] will be given.

1.3.1 Space Integration

Before we perform the actual space integration on \( \Omega \) we have to define how we relate the control volumes to the (primal) grid. Here, a so-called median-dual, vertex-based scheme shown in figure 1.3 is used. The control volumes are formed by connecting the centroids, face and edge-midpoints of all cells sharing the particular node. This leads to the dual grid with volumes \( \Omega_i \) (dashed red cells) such that \( \bigcup_{i=0}^{N} \Omega_i \approx \Omega \).

The vector integral conservation law (1.1) is applied to each control volume \( \Omega_i \), associated to mesh point \( i \) defining the discretized equation for the unknown \( U_i \) attached to that same vertex or cell:

\[
\frac{\partial}{\partial t} \int_{\Omega_i} U dx + \int_{\partial \Omega_i} F(U) \vec{n} dS = \int_{\Omega_i} Q_v dx \tag{1.32}
\]

Equation (1.32) can be integrated from \( t^n := n \Delta t \) to \( t^{n+1} := (n + 1) \Delta t \) for a control volume \( \Omega_i \) to obtain

\[
\int_{\Omega_i} U^{n+1} dx = \int_{\Omega_i} U^n dx - \int_{t^n}^{t^{n+1}} \int_{\Omega_i} F(U) \vec{n} dS dt + \int_{t^n}^{t^{n+1}} \int_{\Omega} Q_v dx dt \tag{1.33}
\]

where \( U^n := U(t^n, x) \). We can replace the surface integrals by a sum over all the bounding faces of the considered volume \( \Omega_i \):

\[
\int_{\Omega_i} U^{n+1} dx = \int_{\Omega_i} U^n dx - \sum_{j \in N(i)} \int_{t^n}^{t^{n+1}} \int_{f_{ij}} F(U) \vec{n}_{ij} dtdS + \int_{t^n}^{t^{n+1}} \int_{\Omega} Q_v dx dt \tag{1.34}
\]

Note, the two integrals over \( F(U) \vec{n}_{ij} \) can be interchanged because this function is continuous, even in the case of discontinuous \( U \) (Rankine-Hugoniot condition). \( N(i) \subset \mathbb{N} \) is the set of indices of the neighboring
nodes to node $i$, $f_{ij}$ is the face associated with the edge connecting nodes $i$ and $j$ and $\vec{n}_{ij}$ is the outward normal of the face $f_{ij}$. Using this equation we can define the cell-averaged conservative variable $\bar{U}_i^n$ at time $t^n$ and the cell- and time-averaged sources $\bar{Q}_v$, and a time-averaged flux $\bar{F}$:

$$\bar{U}_i^n := \frac{1}{|\Omega_i|} \int_{\Omega_i} U^n dx, \quad \bar{Q}_v := \frac{1}{|\Omega_i|} \Delta t \int_{t^n}^{t^{n+1}} \int_{\Omega_i} Q_v dx dt, \quad \bar{F}(U) := \frac{1}{\Delta t} \int_{t^n}^{t^{n+1}} F(U) dt$$  \hspace{1cm} (1.35)

with the cell volume $|\Omega_i| := \int_{\Omega_i} dx$. Thus, the conservative discretization of (1.32) takes the form:

$$\bar{U}_i^{n+1} = \bar{U}_i^n - \frac{\Delta t}{|\Omega_i|} \sum_{j \in N(i)} \int_{f_{ij}} \bar{F}(U)\vec{n}_{ij} dS + \Delta t \bar{Q}_v$$  \hspace{1cm} (1.36)

Note, that equation (1.36) is still the exact relation for the time evolution of the space averaged conservative variables $\bar{U}_i^n$ over cell $i$ if the flux $\bar{F}(U)\vec{n}_{ij}$ through the faces $f_{ij}$ is known. Since $U$ is in general not known on the faces, the integral over the faces of $\bar{F}(U)\vec{n}_{ij}$ has to be reconstructed by either using the cell-averaged values $\bar{U}_i^n$ and $\bar{U}_j^n$ or by using $\bar{U}_i^{n+1}$ and $\bar{U}_j^{n+1}$. The way how this reconstruction is accomplished defines different numerical schemes. For a general (projected) numerical flux $\tilde{F}_{ij}$ then holds

$$\tilde{F}_{ij} \Delta S_{ij} := \tilde{F}(\bar{U}_i, \bar{U}_j) \Delta S_{ij} \approx \int_{f_{ij}} \tilde{F}(U)\vec{n}_{ij} dS$$  \hspace{1cm} (1.37)

where $\Delta S_{ij}$ is the area of the face $f_{ij}$. Equation (1.37) follows immediately if we consider $\tilde{F}_{ij}$ as the approximation of $\bar{F}\vec{n}_{ij}$ evaluated at the midpoint of the face $f_{ij}$ and by using the midpoint-rule for numerical integration. If we furthermore allow an arbitrary time discretization we get the semi-discretized finite volume scheme that can be written as an ordinary differential equation:

$$|\Omega_i| \frac{d\bar{U}_i}{dt} = - \sum_{j \in N(i)} \tilde{F}_{ij} \Delta S_{ij} + \bar{Q}_v =: -R_i(\bar{U})$$ \hspace{1cm} (1.38)

with $R_i(U)$ called residual.

For the definition of numerical fluxes it is convenient to differentiate between the discretization of the inviscid and viscous fluxes ($F(U) = F^{\text{inv}}(U) + F^{\text{vis}}(U)$). Because of the physical nature of the viscous fluxes, the only reasonable way is to employ central differences (central averaging) and the derivatives of the primitive variables can be approximated using the Green-Gauss method [14, 26]. Then the behavior of the scheme depends heavily on the employed discretization for the inviscid fluxes. Therefore we will neglect the viscous fluxes in the following and use the hyperbolic Euler equations as the model problem.

An important property that is used in the proof of convergence of a finite volume scheme for hyperbolic conservation laws is the following:

**Definition 1.2 (Consistent Finite-Volume Scheme)**

A finite volume scheme is consistent with equation (1.32) if

- it can be written as (1.38) (conservative form),
- the numerical flux $\tilde{F}(\bar{U}_i, \bar{U}_j)$ is Lipschitz-continuous in all its arguments and
- if $\bar{U}_i = \bar{U}_j = U$, we should have

$$\tilde{F}(U, U) \Delta S_{ij} = \int_{f_{ij}} \tilde{F}(U)\vec{n}_{ij} dS.$$
Using this definition we can now give the well known Lax-Wendroff Theorem [19]:

**Theorem 1.1 (Lax-Wendroff)**

If the sequence of solutions \( \{\overline{U}_n^n\} \) to a consistent finite-volume scheme converges (as \( \Delta t, |\Omega| \to 0 \)) bounded a.e. to a function \( \overline{U}(x,t) \) then \( \overline{U} \) is a weak solution of (1.1). In particular discontinuities have the correct shock speed given by the Rankine-Hugoniot condition [20].

The Lax-Wendroff-Theorem essentially states, that if we have convergence the limit is a weak solution of (1.1). Nevertheless it does not give any information on convergence itself nor if it is the physically correct solution (since there are in general infinitely many weak solutions). For non-linear systems of equations, like in the case of the Euler equations, there are, except for some special cases, no statements regarding those points. Thus one has to rely on the results from scalar equations where an extensive analysis is available. In this case it is known that the physical correct weak solution must be the vanishing-viscosity solution to the proper viscous equation. This can be ensured by demanding some numerical entropy condition on the numerical flux \( \tilde{F} \). Furthermore, non-linear stability of the scheme can be achieved by a monotonicity requirement, which leads to the concept of Total-Variation-Diminishing (TVD) [20].

**Jameson-Schmidt-Turkel Scheme**

One of the widely used numerical fluxes is the Jameson-Schmidt-Turkel (JST) scheme [17]. In contrast to Roe’s scheme, where the dissipative term is an implicit consequence of how the Riemann problem is solved at the cell interface, the JST scheme is based on a central discretization plus an explicit modeled dissipation term and can be written as

\[
\tilde{F}_{ij} = F(\overline{U}_i, \overline{U}_j) = F \left( \frac{\overline{U}_i + \overline{U}_j}{2} \right) \tilde{u} - d_{ij},
\]

(1.39)

The artificial dissipation \( d_{ij} \) on structured grids is a blending of 2nd- and 4th-order differences that correspond to low and high order dissipation scaled by the maximum eigenvalue of the inviscid flux Jacobian. The generalization on unstructured grids is a combination of an undivided Laplacian and a biharmonic operator. The two levels of dissipation are blended using a pressure switch for triggering low-order dissipation in the vicinity of shock waves. On the edge connecting nodes \( i \) and \( j \) \( d_{ij} \) can be expressed as

\[
d_{ij} = \left( \varepsilon_i^{(2)}(\overline{U}_j - \overline{U}_i) - \varepsilon_i^{(4)}(\nabla^2 \overline{U}_j - \nabla^2 \overline{U}_i) \right) \varphi_{ij} \lambda_{ij}
\]

(1.40)

where the undivided Laplacian \( \nabla^2 \overline{U}_i \) is defined as

\[
\nabla^2 \overline{U}_i := \sum_{k \in N(i)} (\overline{U}_k - \overline{U}_i).
\]

(1.41)

Like in Roe-type schemes, where the dissipation is scaled by the eigenvectors of the inviscid flux Jacobian, we take \( \lambda_{ij} \) as an approximation of the local spectral radius of the inviscid flux Jacobian:

\[
\lambda_{ij} := ((\|\tilde{v}_{ij} \cdot \tilde{n}\| + c_{ij}) \Delta S_{ij}, \quad \tilde{v}_{ij} = \frac{\tilde{v}_i + \tilde{v}_j}{2}, \quad c_{ij} = \sqrt{\frac{\gamma R(T_i + T_j)}{2}}
\]

(1.42)

The \( \varphi_{ij} \) occurring in equation (1.40) is an additional scaling parameter. In the implement version it is taken to be

\[
\varphi_{ij} = \frac{\varphi_i \varphi_j}{\varphi_i + \varphi_j}, \quad \varphi_i = \left( \frac{\sum_{k \in N(i)} \lambda_{ik}}{4 \lambda_{ij}} \right)^p
\]

(1.43)

An important part of the scheme is the handling of the switching parameters \( \varepsilon_i^{(2)} \) and \( \varepsilon_i^{(4)} \). The task for \( \varepsilon_i^{(2)} \) is to achieve less low-order dissipation in smooth regions and the right amount near shock waves to make the scheme first order accurate. Therefore a pressure switch is used:

\[
\varepsilon_i^{(2)} := \kappa^{(2)} s_2 \frac{\sum_{k \in N(i)} (p_k - p_i)}{\sum_{k \in N(i)} (p_k + p_i)}
\]

(1.44)
The fourth order dissipation, blended in by \( \varepsilon_{ij}^{(4)} \), damps high frequencies which helps to reach steady state quickly. Though it has been found that it produces overshoots near shock waves and therefore it needs to be switched off then by using

\[
\varepsilon_{ij}^{(4)} := s_4 \max(0, \kappa^{(4)} - \varepsilon_{ij}^{(2)})
\]

(1.45)

\( s_2 \) and \( s_4 \) are again some stretching parameters. Typical values for \( \kappa^{(2)} \) and \( \kappa^{(4)} \) are usually

\[
\kappa^{(2)} = \frac{1}{4}, \quad \kappa^{(4)} = \frac{1}{256}
\]

(1.46)

It follows that we have in smooth regions \( \varepsilon_{ij}^{(2)} \sim O(h^2) \) and \( \varepsilon_{ij}^{(4)} \sim O(1) \), where \( h \) is the maximal diameter of the cells in the grid. The added term \( d_{ij} \) is then of order \( h^3 \). Near a shock wave we have \( \varepsilon_{ij}^{(2)} \sim O(1) \) and the scheme behaves locally like a first order scheme. Note, the JST scheme, like it is introduced here, is not TVD, hence it still may produce overshoots at shocks waves if \( \kappa^{(4)} \) is not chosen properly.

### 1.3.2 Time Integration

In this work we only consider steady state simulations, i.e. we are interested in the solution of the conservation laws for \( t \to \infty \). As a result we do not have to care about the accuracy of the time discretization and we can make use of convergence acceleration techniques like local time-stepping. Consider the semi-discretized finite volume scheme (1.38). Since it has to be valid over the whole time interval, one has to make the choice of evaluating \( R_i(U) \) either at time \( t^n \) or \( t^{n+1} \). The former choice leads to explicit schemes and the latter one to implicit schemes. We focus here on the in theory unconditionally stable implicit schemes where the easiest way is the use of an implicit Euler scheme. It can be written as

\[
\frac{|\Omega_i|}{\Delta t_i} \Delta U^n_i = -R_i(U^{n+1}).
\]

(1.47)

where \( \Delta U^n_i := \tilde{U}^{n+1}_i - \tilde{U}^n_i \). The indices at \( \Delta t^n_i \) indicate the possibility that it may vary between cells and in time. Since the residual at time \( t^{n+1} \) is unknown we have to linearize it around \( t^n \):

\[
R_i(U^{n+1}) = R_i(U^n) + \frac{\partial R_i(U^n)}{\partial t} \Delta t^n_i + O(\Delta t^2)
\]

(1.48)

\[
= R_i(U^n) + \sum_{j \in \mathcal{N}(i)} \frac{\partial R_i(U^n)}{\partial U_j} \Delta U^n_j + O(\Delta t^2).
\]

(1.49)

For the last equality the chain rule and the implicit Euler discretization for the time derivative have been used. Thus we have to solve the following linear system for the updates \( \Delta U^n_j, j = 1, \ldots, N \):

\[
\sum_{j \in \mathcal{N}(i)} \left( \frac{|\Omega_i|}{\Delta t_i} \delta_{ij} \frac{\partial R_i(U^n)}{\partial U_j} \right) \Delta U^n_j = -R_i(U^n)
\]

(1.50)

or in matrix-vector form

\[
(D^n)_{ij} := \frac{|\Omega_i|}{\Delta t^n_i} \delta_{ij}, \quad \left( \frac{\partial R_i(U^n)}{\partial U^n} \right)_{ij} := \frac{\partial R_i(U^n)}{\partial U_j}
\]

(1.52)

An important fact is that if \( D^n \) would be zero \( (\Delta t^n_i \to \infty) \), then (1.51) would be identical to a step of Newttons method applied to the nonlinear system \( R(U) = 0 \). But despite implicit schemes being unconditionally stable in theory, a specific value of \( \Delta t^n_i \) is still needed to relax the problem due to its high nonlinearity. In that sense, the implicit Euler scheme can be seen as a damped variant of Newttons method. However, local
time-stepping can be used to accelerate convergence to a steady state which allows each cell in the mesh to advance at a different local time step. The local time step calculation is based on the estimated values of the eigenvalues $\lambda_{\text{conv}}^i$ and $\lambda_{\text{visc}}^i$ of the inviscid and viscous Jacobian at every node $i$:

$$\Delta t_n^i = N_{\text{CFL}}|\Omega_i| \min \left( \frac{1}{\lambda_{\text{conv}}^i}, \frac{1}{\lambda_{\text{visc}}^i} \right)$$

where $N_{\text{CFL}}$ is the prescribed Courant-Friedrichs-Levy (CFL) number. Further acceleration can be achieved by using a Full Approximation Scheme multi-grid method that is implemented in the SU$^2$ code. Since the depiction of this method exceeds the scope of this work, the interested reader may be referred to Palacios et al. [26] or to general multi-grid literature. The same holds for the solution of the arising linear systems. SU$^2$ offers several well-known iterative solver like Gmres or Symmetric-Gauss-Seidel variants as well as suitable preconditioners.
The term One-Shot was first used in the work of TA’ASAN [30] where he discussed the efficient numerical treatment of optimal control problems governed by elliptic partial differential equations. A new method was designed to solve the full optimization problem directly, rather than accelerating a descent method (NAND method), by an efficient multi-grid solver for the equations involved. The extension of this approach to other optimal control problems followed, especially to shape optimization in potential flows [18]. It was demonstrated that the solution of the control problem can be achieved with the cost of solving the constraint equations about two to three times. This is the one of the notable advantages of the One-Shot method, namely bringing the cost of design and optimization to the same order as that of a single analysis. The fact that the cost of the optimization is bounded by a reasonable factor times the cost of a single analysis is commonly denoted by bounded retardation. Reduced Sequential Quadratic Programming (rSQP) methods [1] became quiet popular and are known for keeping this factor quiet small. They have been successfully applied to shape optimization problems [3, 11], but they essentially require the access to the exact state Jacobian or rather they demand Newton’s method for the solution of the constraint equation. For problems where the flow is governed by complex models typically slow converging variants, like the implicit Euler method (or damped Newton’s method), are used to achieve convergence at all (ref. section 1.3.2). While it may be possible to still construct the state Jacobian for this cases, if furthermore multi-grid methods or other methods for convergence acceleration are deployed, this will hardly be the case anymore. Consequently, the state update in each iteration will only be an approximation of the exact Newton step that will likely cause a wrong stationary point of the optimization. In this case for example inexact rSQP methods can be used [10], but they still require that the solution accuracy is increased with the closeness to the optimal solution. In the remainder of this chapter we will introduce two methods that can instead be applied for the solution of the optimization problem. The first one is derived from the SQP method for general optimization problems, whereas the second one is based on the reformulation of the problem more suitable for constraints solved by fixed-point iterations. Furthermore, the latter allows the direct computation of the needed derivatives by using Algorithmic Differentiation without the explicit knowledge of the constraint Jacobian.

### 2.1 Problem Statement and Parametrization

Aerodynamic shape optimization is closely related to optimal control of distributed systems. Such problems are distinguished from general nonlinear programming problems by the fact that the vector of variables can be partitioned into the state vector \( y \in Y \) and the control vector \( u \in U \). Furthermore we have an equation
Theory of One-Shot Shape Optimization

c(y, u) = 0 defining the state. The optimization problem can then be formulated as

\[ \min_{y, u} f(y, u) \]
\[ \text{subject to } c(y, u) = 0 \]

where \( f : Y \times U \to \mathbb{R} \) is the cost function. We assume that \( U, Y \) and their Cartesian product \( X = Y \times U \) are suitable Hilbert spaces. In aerodynamic shape optimization we want to find a shape of an airfoil that has the best performance properties measured by the cost function \( f \). For example we want to decrease the drag force exerted on the airfoil by the surrounding flow or to increase the lift. In this cases the cost function may be the drag \((1.29)\) or lift coefficient \((1.30)\). The equation \( c(y, u) = 0 \) can then be identified by the flow equations from section 1.1 and consequently \( y \) is identical to the vector of conserved variables \( \bar{U} \).

In the following we interpret the problem as a-priori discretized using the Finite Volume Method considered in section 1.3. Thus, \( U \) and \( Y \) have finite dimensions \( n = \text{dim}(Y) \) and \( m = \text{dim}(U) \) and their elements can be identified from \( \mathbb{R}^n \) and \( \mathbb{R}^m \), respectively. This convention allows us to write duals as transposed vectors and inner products as the usual scalar products in Euclidean space. Furthermore we assume that for each design \( u \in U \) there exists a unique state \( y \in Y \) such that \( \frac{\partial c}{\partial y} \) is invertible.

The computation of the (reduced) gradient in the adjoint methods as well as in the One-Shot method, as we will see, is in principle independent of the number of design variables used to represent the design. Therefore it is quiet natural to use the coordinates of the mesh nodes itself as the parametrization in order to use the maximum degrees of freedom for the optimization. This parametrization is usually denoted by COD-free or Free-Node parametrization [24]. However, mostly in practice a much smaller design space is used by representing the design with the help of splines or other smooth functions. Today, often the so called Hicks-Henne functions [13] are deployed. This is primarily because mesh sensitivities are required, i.e. the Jacobian \( \frac{\partial c}{\partial u} \). They are typically calculated using Finite Differences and hence pose as an additional burden especially if the dimension of \( u \) is high. In this work the explicit handling of mesh sensitivities is avoided by the use of Algorithmic Differentiation and therefore a Free-Node parametrization is realizable without additional computational effort and consequently \( u \) contains the coordinates of the surface mesh nodes. Though special geometric constraints have to be applied in this case in order get a physically meaningful solution. For example by allowing only a movement in the direction of the normal at a surface mesh node. This method can essentially be thought of as using shape calculus on a discrete level to compute the gradient. Indeed, we will later define a suitable design preconditioner that is based on the theory of shape calculus.

2.2 Approximated Reduced Sequential Quadratic Programming

The method described in this section is well suited for problems where we have access to the constraint Jacobian, but the solution of the occurring linear systems is rather inaccurate. Before we apply it to aerodynamic shape optimization we will first give an introduction on the solution of general optimization problems. This is helpful in the sense that this method can be seen as a modified or approximated reduced Sequential Quadratic Programming (rSQP) method. Hence the basic idea behind the SQP method is explained in the following.

2.2.1 Karush-Kuhn-Tucker System

Consider the equality constrained optimization problem

\[ \min_x f(x) \text{ subject to } c(x) = 0 \]

where \( x \in \mathbb{R}^{n+m} \), \( f : \mathbb{R}^{n+m} \to \mathbb{R} \) and \( c : \mathbb{R}^{n+m} \to \mathbb{R}^n \). Most of the theory regarding the solution of constrained optimization relies on the concept of the Lagrangian multiplier method where the original problem is replaced by an unconstrained problem. This can be achieved by introducing the Lagrangian function \( L : \mathbb{R}^{n+m} \times \mathbb{R}^n \to \mathbb{R} \) for problem (2.3):

\[ L(x, \lambda) := f(x) - \lambda^T c(x), \]
where $\lambda \in \mathbb{R}^n$ is the Lagrange multiplier. The conditions that qualify a point to be a minimizer can then be summarized by the following theorem:

**Theorem 2.1 (First-Order Necessary Conditions)**

If $x^* \in \mathbb{R}^{n+m}$ is a (local) solution of (2.3) and the functions $f$ and $c$ are continuously differentiable, then there exists a Lagrange multiplier $\lambda^* \in \mathbb{R}^n$ such that the following conditions are satisfied at $(x^*, \lambda^*)$:

\[
\nabla_x L(x^*, \lambda^*) = 0 \quad (2.5)
\]
\[
c(x^*) = 0 \quad (2.6)
\]

A proof of theorem 2.1 can for example be found in [25]. The optimality conditions (2.5) and (2.6) are also known as Karush-Kuhn-Tucker (KKT) conditions. They essentially form a nonlinear system in the unknowns $x$ and $\lambda$. The conceptually simplest approach to solve this system is the application of Newton’s method. If we assume we are at an iterate $(x_k, \lambda_k)$ the Newton step is given by

\[
\begin{pmatrix}
x_{k+1} \\
\lambda_{k+1}
\end{pmatrix} = \begin{pmatrix}
x_k \\
\lambda_k
\end{pmatrix} + \begin{pmatrix}
\Delta x_k \\
\Delta \lambda_k
\end{pmatrix} \quad (2.7)
\]

where $\Delta x_k$ and $\Delta \lambda_k$ solve the following linear system:

\[
\begin{pmatrix}
W_k & -A_k^T \\
-A_k & 0
\end{pmatrix}
\begin{pmatrix}
\Delta x_k \\
\Delta \lambda_k
\end{pmatrix} = \begin{pmatrix}
-\nabla f_k + \lambda_k^T \nabla c_k \\
c_k
\end{pmatrix} \quad (2.8)
\]

with

\[
A_k := \nabla c(x_k) \in \mathbb{R}^{n \times (n+m)}, \quad W_k := \nabla^2_{xx} L(x_k, \lambda_k) \in \mathbb{R}^{(n+m) \times (n+m)}. \quad (2.9)
\]

The subscript $k$ means evaluation at $x_k$ and $\lambda_k$. Provided the matrix $A_k$ has full rank and the matrix $W_k$ is positive definite on the tangent space of constraints, i.e. $d^T W_k d > 0$ for $A_k d = 0$, the Newton iteration can be shown to be locally quadratically convergent. Besides, at the solution $x^*$ the following theorem holds:

**Theorem 2.2 (Second-Order Necessary Conditions)**

Suppose that $x^*$ is a local solution of (2.3) and the KKT conditions (2.5) - (2.6) are satisfied. Then

\[
d^T \nabla^2_{xx} L(x^*, \lambda^*) d \geq 0, \quad (2.10)
\]

for all $d$ that satisfy $\nabla c(x^*) d = 0$.

Nevertheless, for the development of practical algorithms it is more convenient to use an alternative way of view on the Newton iteration (2.8). Therefore consider the following quadratic sub-program at the iterate $(x_k, \lambda_k)$ with linearized constraints to find a search direction $p_k$:

\[
\min_{p_k} \frac{1}{2} p_k^T W_k p_k + \nabla f(x_k)^T p_k \quad (2.11)
\]

subject to $A_k^T p_k + c(x_k) = 0 \quad (2.12)$

It is easy to verify that if a solution exists, it can be expressed as the following linear system:

\[
\begin{pmatrix}
W_k & -A_k^T \\
-A_k & 0
\end{pmatrix}
\begin{pmatrix}
p_k \\
\mu_k
\end{pmatrix} = \begin{pmatrix}
-\nabla f_k \\
c_k
\end{pmatrix} \quad (2.13)
\]

$\mu_k$ is the Lagrange multiplier for the linearized constraint. A key observation is that we can identify $\mu_k$ with $\lambda_{k+1}$ from (2.7). If we subtract $\lambda_k^T \nabla c_k$ on both sides in the first equation of (2.8) we get

\[
\begin{pmatrix}
W_k & -A_k^T \\
-A_k & 0
\end{pmatrix}
\begin{pmatrix}
\Delta x_k \\
\lambda_{k+1}
\end{pmatrix} = \begin{pmatrix}
-\nabla f_k \\
c_k
\end{pmatrix}, \quad (2.14)
\]
which is nothing but equation (2.13) with \( \mu_k = \lambda_{k+1} \) and \( p_k = \Delta x_k \). Thus we have a one-to-one correspondence between Newton’s method applied to the KKT conditions and solving a quadratic sub-problem in each iteration \( k \). If the Hessian \( W_k \) is available, a straightforward approach to solve the system (2.14) is to perform a sparse factorization of the block matrix or we may solve the Schur complement system derived from (2.14).

### 2.2.2 Null Space Ansatz and Variable Elimination Basis

A more viable approach for shape optimization problems is the use of a reduced SQP (rSQP) method that allows one to exploit the specific structure of such problems. The method is based on the null space ansatz [25]. To this end let us decompose the search direction \( p_k \) into two components:

\[
p_k = Z_k p_Z + Y_k p_Y
\]

in which \( Z_k \in \mathbb{R}^{(n+m) \times m} \) is a matrix whose columns form a basis of the null space of \( A_k \), i.e. \( A_k Z_k = 0 \), and \( Y_k \in \mathbb{R}^{(n+m) \times n} \) is chosen so that the matrix

\[
Q_k = \begin{pmatrix} Z_k & Y_k \end{pmatrix}
\]

is nonsingular. Hence, \( Z_k \) and \( Y_k \) form a basis for \( \mathbb{R}^{n+m} \). In the following \( p_Z \in \mathbb{R}^m \) is called the null space component and \( p_Y \in \mathbb{R}^n \) the range space component even though strictly speaking the columns of \( Y_k \) need not span the range space of \( A_k^T \). If we insert (2.15) into (2.14) we obtain the following systems to be solved for \( p_Y \) and \( p_Z \):

\[
A_k Y_k p_Y = -c_k
\]

\[
Z_k^T W_k Z_k p_Z = -Z_k^T W_k Y_k p_Y - Z_k^T \nabla f_k
\]

The new Lagrange multiplier can then be obtained by solving

\[
(A_k Y_k)^T \lambda_{k+1} = Y_k^T (\nabla f_k + W_k p_k),
\]

which follows immediately from the first equation in (2.14). The term \( H_k := Z_k^T W_k Z_k \) is called reduced Hessian and is of dimension \( m \times m \), similarly \( q_k := Z_k^T \nabla f_k \) is the reduced Gradient. Thus we have reduced the SQP system (2.14) to the three smaller systems (2.17), (2.18) and (2.19). Like the Hessian of the Lagrangian, according to theorem 2.2 the reduced Hessian must be positive definite at a local minimizer. This method has been proved to be very effective in many areas of applications because it requires only the computation or approximation of \( H_k \) instead of the full \( (n+m) \times (n+m) \) Hessian \( W_k \). We can simplify the problem by ignoring the term involving \( p_k \) in equation (2.19) and at the same time decoupling the computation of \( \lambda_{k+1} \) and \( p_k \). This can be justified by the fact that \( p_k \) goes to zero if we approach the solution while \( \nabla f_k \) typically will not. A simple approach to handling the problematic matrix \( Z_k^T W_k Y_k \) is to ignore it. Since \( p_Y \) converges usually to zero faster than \( p_Z \) the resulting equation gives a good approximation. If we furthermore use an approximation \( B_k \) of the reduced Hessian \( H_k \) such that \( B_k \approx H_k \), it can be shown that the method exhibits two-step Q-superlinear convergence, i.e.

\[
\frac{\|x_{k+2} - x^*\|}{\|x_k - x^*\|} \to 0 \text{ as } k \to 0,
\]

using appropriate range and null space bases [25]. However, the main problem is typically the computation of a suitable null space \( Z_k \). A QR factorization of \( A_k^T \) would be possible, but unfortunately, for problems in which \( A_k \) is large and sparse, a sparse QR factorization is very costly to compute. In shape optimization, or more general in optimal control, we can explicitly state \( Z_k \) by using a variable elimination approach. Therefore we use the fact that shape optimization problems differ from general optimization problems in that we can split \( x \) into a state \( y \in \mathbb{R}^n \) and a design component \( u \in \mathbb{R}^m \) such that

\[
x^T := (y^T, u^T).
\]
Then $A_k$ and $W_k$ have the following block structure:

$$A_k = \begin{pmatrix} A_y & A_u \end{pmatrix}, \quad W_k = \begin{pmatrix} W_{yy} & W_{yu} \\ W_{uy} & W_{uu} \end{pmatrix}$$

(2.22)

where the subscripts $u$ and $y$ indicate the respective derivative. $A_y$ is called the state Jacobian and is very sparse, whereas $A_u$ can be sparse or dense, depending on the influence of the design variables. For free-node parametrization the perturbation of a node is typically felt by all nodes, so we can assume that $A_u$ is dense. Overall, since $n \gg m$, $W$ is also very sparse. Due to the partition of $A_k$ a simple way to get a suitable null space is to perform a block-wise Gauss Elimination of the null space system $A_k Z_k = 0$ resulting in

$$(A_y \quad A_u) \begin{pmatrix} Z_1 \\ Z_2 \end{pmatrix} = 0 \Rightarrow Z_1 = -A_y^{-1} A_u Z_2.$$  

(2.23)

The particular choice of $Z_2 \in \mathbb{R}^{m \times m}$ is arbitrary so that $Z_2 \equiv I$ is reasonable. The approach

$$Z_k = \begin{pmatrix} -A_y^{-1} A_u \cr I \end{pmatrix}$$

(2.24)

is referred to as Direct Variable Elimination [25]. $Y_k$ can then be chosen to be

$$Y_k = \begin{pmatrix} I \\ 0 \end{pmatrix}.$$  

(2.25)

Clearly, $Q_k$ in (2.16) is non-singular provided $A_y$ is non-singular, and hence $Z_k$ and $Y_k$ defined in (2.24) and (2.25) form a basis of $\mathbb{R}^{n+m}$. Because of its simplicity this basis is a quiet common choice, although it is not as numerically stable as for example a orthogonal basis. Especially if the matrix $A_y$ is ill-conditioned, the step computation may be inaccurate. Substituting the definition of the basis into (2.15) yields

$$\Delta x_k = p_k = \begin{pmatrix} \Delta y_k \\ \Delta u_k \end{pmatrix} = \begin{pmatrix} -A_y^{-1} A_u p_Z + p_Y \\ p_Z \end{pmatrix}$$

(2.26)

and the range step equation (2.17) becomes

$$A_y \Delta y_k + A_u \Delta u_k = -c_k.$$  

(2.27)

Using the null space definition (2.24) in (2.18) gives the equation to be solved for the null space step $p_Z = \Delta u_k$:

$$B_k \Delta u_k = -\nabla u f_k + A^T u A_y^{-1} \nabla y f_k = -g_k.$$  

(2.28)

where $B_k$ is again some approximation of the exact reduced Hessian $H_k$ given by

$$H_k = Z_k^T W_{yy} Z_k - Z_k^T W_{yu} - W_{uy} Z_k + W_{uu} = A_y^T A_y^{-1} W_{yy} A_y - A_y^T A_y^{-1} W_{yu} - W_{uy} A_y^{-1} A_u + W_{uu}.$$  

(2.29)

(2.30)

The new Lagrange multiplier estimate can be obtained from

$$A_y^T \lambda_{k+1} = \nabla_y f_k \Leftrightarrow A_y^T \Delta \lambda_k = \nabla_y f_k - A_y^T \lambda_k.$$  

(2.31)

Finally, the new values $y_{k+1}$, $u_{k+1}$ and $\lambda_{k+1}$ are defined by

$$\begin{pmatrix} y_{k+1} \\ u_{k+1} \\ \lambda_{k+1} \end{pmatrix} = \begin{pmatrix} y_k \\ u_k \\ \lambda_k \end{pmatrix} + \begin{pmatrix} \Delta y_k \\ \Delta u_k \\ \Delta \lambda_k \end{pmatrix}.$$  

(2.32)

As for the general SQP (2.8) we can write the rSQP method in matrix form. If we insert (2.31) into (2.28) we get

$$B_k \Delta u_k - A_y^T A_y^{-1} \lambda_k = - (\nabla u f - A_y^T \lambda_k),$$  

(2.33)
and hence, the rSQP system becomes

\[
\begin{pmatrix}
0 & 0 & -A_T^T \\
0 & B_k & -A_T^T \\
-A_y & -A_u & 0
\end{pmatrix}
\begin{pmatrix}
\Delta y_k \\
\Delta u_k \\
\Delta \lambda_k
\end{pmatrix}
= \begin{pmatrix}
-(\nabla_y f_k - A_T^T \lambda_k) \\
-(\nabla_y f_k - A_T^T \lambda_k) \\
\nabla_y L_k
\end{pmatrix} + \begin{pmatrix}
-(\nabla_u f_k - A_T^T \lambda_k) \\
-(\nabla_u f_k - A_T^T \lambda_k) \\
\nabla_u L_k
\end{pmatrix}.
\] (2.34)

It should be noted that due to equation (2.27) the calculation of the state step \(\Delta y_k\) and the design step \(\Delta u_k\) is still coupled. This would require indeed the (approximated) inverse of the coefficient matrix in equation (2.34) to actually solve the system. To overcome this problem we can use the Taylor expansion of \(c(y_k, u_{k+1})\) around \(u_k\):

\[
c(y_k, u_{k+1}) \approx c(y_k, u_k) + A_u \Delta u_k
\] (2.35)

Using this approximation (2.27) reduces to

\[
A_u \Delta y_k = -c(y_k, u_{k+1}),
\] (2.36)

and the system (2.34) can be solved using a block-wise approach that requires only the acting of \(A_y^{-1}, A_y^{-T}\) and \(B_y^{-1}\) on some vectors. This is a valid approach since we essentially undo a part of the linearization of the constraint in the quadratic program (2.12).

### 2.2.3 Approximate Null Space Iteration

Let us assume we already have an established iterative solver for the state equation \(c(y, u) = 0\) for given design \(u\) of the form

\[
y_{k+1} = y_k - \tau \hat{A}_y^{-1} c(y_k, u),
\] (2.37)

where \(\tau \in \mathbb{R}\) is a damping factor. In our case this would be the solution method presented in section 1.3.2. Note, that it is neither possible to give an explicit representation of \(\hat{A}_y\) nor the value of \(\tau\) due to the use of different convergence acceleration methods. If we would have a Newton-based solver, i.e. \(\hat{A}_y = A_y\) and \(\alpha = 1\), we could simply hook the optimization into the solution of the state equation. But even in this case, since the resulting linear system is often calculated using a rather inaccurate iterative solver itself, we have in general \(\hat{A}_y^{-1} \neq A_y^{-1}\). Especially the inexact solution of the equation for the Lagrange multiplier (2.31) will then lead to errors in the reduced gradient, which in turn results in a wrong stationary point of the algorithm which is not optimal. We will ignore the fact that \(\tau\) is not known and assume that we can incorporate its existence by introducing a appropriately chosen step size \(\tau > 0\) for the updates in (2.32) such that

\[
\begin{pmatrix}
y_{k+1} \\
u_{k+1} \\
\lambda_{k+1}
\end{pmatrix}
= \begin{pmatrix}
y_k \\
u_k \\
\lambda_k
\end{pmatrix} + \tau \begin{pmatrix}
\Delta y_k \\
\Delta u_k \\
\Delta \lambda_k
\end{pmatrix}.
\] (2.38)

In Ito et al. [15] the authors proposed the approximation \(\hat{A}_y \approx A_y\) in the coefficient matrix in (2.34) which results in

\[
\begin{pmatrix}
0 & 0 & -\hat{A}_y^T \\
0 & B_k & -\hat{A}_y^T \\
-\hat{A}_y & -A_u & 0
\end{pmatrix}
\begin{pmatrix}
\Delta y_k \\
\Delta u_k \\
\Delta \lambda_k
\end{pmatrix}
= \begin{pmatrix}
-(\nabla_y L_k) \\
-(\nabla_u L_k) \\
\nabla_u L_k
\end{pmatrix}.
\] (2.39)

This iteration is referred to as Approximate Null Space Iteration since it is based on the approximate null space representation

\[
Z_k \approx \bar{Z}_k := \left(-\hat{A}_y^{-1} A_u\right)
\] (2.40)

If we have closer look on the resulting equation for the Lagrangian multiplier

\[
\hat{A}_y^T \Delta \lambda_k = \nabla_y f_k - A_y^T \lambda_k,
\] (2.41)
we note that the $\lambda_k$ now have a true residual history, rather than being calculated from scratch in each iteration like in (2.31). Thus, equation (2.41) can be seen as a defect-correcting version of (2.31). Relating to $B_k$ it has been found that the method works better if it is an approximation of the reduced Hessian consistent with (2.40), i.e.

$$B_k \approx \tilde{H}_k := \tilde{Z}_k^T W_k \tilde{Z}_k,$$

(2.42)

rather than of $H_k$.

### 2.2.4 Notes on Convergence

Convergence of iteration (2.39) for the general nonlinear case is not yet guaranteed theoretically and is part of ongoing research. However, in [15] theoretical convergence results for a simpler problem are given. Therefore the linear-quadratic problem

$$\min_{y,u} \frac{1}{2} (y^T W_y y + y^T W_{yu} u + u^T W_{uy} y + u^T W_u u) + f_y^T y + f_u^T u$$

subject to $A_y y + A_u u + c = 0$

(2.43)

(2.44)

is considered, where the matrices $W_y \in \mathbb{R}^{n \times n}$ and $W_u \in \mathbb{R}^{m \times m}$ are supposed to be symmetric and $A_y$ is supposed to be invertible. Note that the indices do not refer to any derivatives, but rather to their specific position in the target functional and constraints, so that the occurring matrices and vectors are somewhat arbitrary. This problem may stand alone or it may arise as a quadratic sub-program in a nonlinear optimization problem like (2.11) - (2.12). If we define the Lagrangian for this problem as

$$L(y, u, \lambda) := \frac{1}{2} (y^T W_y y + y^T W_{yu} u + u^T W_{uy} y + u^T W_u u) + f_y^T y + f_u^T u - \lambda^T (A_y y + A_u u + c)$$

(2.45)

we can use the method depicted in the previous sections for the solution. Trivially, the SQP method should give the solution in one step, whereas the rSQP method, since some terms are omitted, consequently should not. Hence, we get the following iteration

$$\begin{pmatrix} y_{k+1} \\ u_{k+1} \\ \lambda_{k+1} \end{pmatrix} = \begin{pmatrix} y_k \\ u_k \\ \lambda_k \end{pmatrix} - \begin{pmatrix} 0 & 0 & -\tilde{A}_y^T \\ 0 & B & -A_u^T \\ -\tilde{A}_y & -A_u & 0 \end{pmatrix}^{-1} \begin{pmatrix} W_y & W_{yu} & -A_y^T \\ W_{uy} & W_u & -A_u^T \\ -A_y & -A_u & 0 \end{pmatrix} \begin{pmatrix} y_k \\ u_k \\ \lambda_k \end{pmatrix} + \begin{pmatrix} f_y \\ f_u \\ c_k \end{pmatrix},$$

(2.46)

where $A_y$ and $B$ are approximations of $A_y$ and $\tilde{H}$, respectively. Then the following theorem gives information as to convergence:

**Theorem 2.3 (Convergence of the Approximate Null Space Iteration)**

Define

$$\rho_A := \rho(I - \tilde{A}_y^{-1} A_y) \quad \text{and} \quad \rho_H := \rho(I - B^{-1} \tilde{H}),$$

(2.47)

where $\rho$ denotes the spectral radius. Then there exists an $\eta > 0$, such that iteration (2.46) converges to the solution of the linear-quadratic problem (2.43), provided

$$\max(\rho_A, \rho_H) < \eta.$$

(2.48)

The proof of this theorem is essentially based on the nilpotency of degree 3 of the iteration matrix and perturbation analysis. It shows that the convergence behavior of iteration (2.46) is limited both by the quality of the solution of the state equation and by the approximation quality of the consistent reduced Hessian $\tilde{H}$. The former is fixed by the particular solver for $c(y, u) = 0$. Most of the possible choices for $B_k$ are based on just using $W_u$, or rather $W_{uu}$ for the general problem, since it is the easiest accessible part in (2.29). Though it is still to question whether this choice is useful.
2.3 Reduced Quasi-Newton Method

In this section we will use another approach for the derivation of a solution method for the shape optimization problem (2.1) - (2.2). It is closely related to the method described in the previous section as the resulting iteration is similar to (2.39) and differs only in the usage of the available information. Based on the principles of Algorithmic Differentiation we can directly augment the state solver to generate the Lagrangian multiplier and the reduced gradient without the need for an explicit representation of the state preconditioner $\tilde{A}_y$. The theory of rSQP methods in turn allows a first idea on how to choose a suitable preconditioner for the design update, although this choices do not necessarily lead to convergence in theory. Therefore, it will be shown how to derive a design preconditioner that ensures even global convergence under reasonable assumptions.

2.3.1 Karush-Kuhn-Tucker System based on the Shifted Lagrangian

Unlike in rSQP methods, we incorporate the fact that the state equation is solved by a slow converging iterative solver even before we formulate the first order necessary conditions for optimality. Therefore we will use another approach for the derivation of a solution method for the shape optimization problem (2.1) - (2.2). It is closely related to the method described in the previous section as the resulting iteration is similar to (2.39) and differs only in the usage of the available information. Based on the principles of Algorithmic Differentiation we can directly augment the state solver to generate the Lagrangian multiplier. Then, according to the first order necessary conditions (Theorem 2.1), a KKT point $(y^*, \lambda^*, u^*)$ must satisfy

$$y^* = G(y^*, u^*),$$

$$\lambda^* = N_y(y^*, \lambda^*, u^*) = f_y(y^*, u^*) + G^T_y(y^*, u^*)\lambda^*, $$

$$0 = N_u(y^*, \lambda^*, u^*) = f_u(y^*, u^*) + G^T_u(y^*, u^*)\lambda^*.$$  

Note, in theory $\lambda$ is a row vector since it is an element of the dual space of $\mathbb{R}^n$. However, for the application it is more convenient to treat it as a column vector like it is done here. The shifted Lagrangian $N$ is related to the standard Lagrangian (2.4) by the shift $\lambda^T y$ and hence the second order necessary conditions (theorem 2.2) applied to $N$ and $L$ are equivalent. We note that the second equation is, like the first one, a fixed-point equation and can be solved similarly. Then in order to drive $L_u = N_u$ to zero we can apply a quasi-Newton method. Hence, to reach the KKT point the following coupled iteration can be used [9]:

$$
\begin{pmatrix}
y_{k+1} \\
\lambda_{k+1} \\
u_{k+1}
\end{pmatrix}
=
\begin{pmatrix}
G(y_k, u_k) \\
N_y(y_k, \lambda_k, u_k) \\
u_k - B_k^{-1} N_u(y_k, \lambda_k, u_k)
\end{pmatrix}
$$

where $B_k$ is the design preconditioner which must be selected as a symmetric positive definite $m \times m$ matrix.
2.3.2 Piggy-Back Iteration

For a complex flow solver it is not unusual that thousands of iterations are needed to obtain a high accuracy solution of the state equation. Especially at the beginning of iteration (2.49) where we are quite far away from \((y^*, u)\), we cannot ensure that \(G\) will be contractive meaning that the spectral radius \(\rho_G := \rho(G_y)\) will be smaller than 1 for all arguments of \((y, u)\). However, for the remainder of this chapter we assume that this will be the case and we have

\[ \|G_y(y, u)\| \leq \rho_G < 1 \Rightarrow \|G(y, u) - G(z, u)\| \leq \rho_G\|y - z\|. \quad (2.59) \]

The implication follows from the mean value theorem on any convex subdomain of \(Y\). Moreover no eigenvalue of \(G_y(y, u)\) can be in modulus greater than \(\rho_G\) at any point \((y, u)\). The simultaneous iteration for fixed \(u\) of the state equation and the equation for the Lagrangian multiplier

\[
\begin{pmatrix}
(y_{k+1} \\
\lambda_{k+1}
\end{pmatrix}
= \begin{pmatrix}
G(y_k, u) \\
N_y(y_k, \lambda_k, u)
\end{pmatrix} \quad (2.60)
\]

is commonly denoted as Piggy-Back iteration \([9]\). Since \(G\) is a contractive function, also \(N_y\) is contractive as the iteration matrix is precisely \(G_y^*\) and \(\|G_y^*(y, u)\| = \|G_y(y, u)\| \leq \rho_G\). Hence (2.60) converges to a unique fixed-point \((y^*, \lambda^*)\). By the Banach fixed-point theorem both, the state variable and the Lagrangian multiplier, converge with the same asymptotic R-factor, i.e.

\[
\begin{align*}
\limsup_k \sqrt[2]{\|y_k - y^*\|} & \leq \rho_G, \quad (2.61) \\
\limsup_k \sqrt[2]{\|\lambda_k - \lambda^*\|} & \leq \rho_G. \quad (2.62)
\end{align*}
\]

But since the Lagrangian multiplier are heading for a moving target, namely the solution of an equation parametrized by \(y_k\), they somewhat lag behind the normal iterates \([5]\) in that

\[ \|\lambda_k - \lambda^*\| \sim k\|y_k - y^*\|. \quad (2.63) \]

2.3.3 Conditions on the Design Preconditioner

\(B_k\) should be defined such that the design update influences the spectral radius of the coupled iteration (2.58) to stay below 1 and as close as possible to \(\rho_G\). The rate of convergence to a limit point \((y^*, \lambda^*, u^*)\) is determined by the associated Jacobian

\[
J^* := \frac{\partial(y_{k+1}, \lambda_{k+1}, u_{k+1})}{\partial(y_k, \lambda_k, u_k)}_{(y^*, \lambda^*, u^*)} = \begin{pmatrix}
G_y & 0 & -G_{uv}
N_{yy} & G_y^T & N_{yu}
-B^{-1}_{uu}N_{yu} & -B^{-1}_{uu}G_y^T & I - B^{-1}_{uu}N_{uu}
\end{pmatrix} \quad (2.64)
\]

where it assumed that \(B_*\) is the limit of the \(B_k\). Convergence is ensured by the contraction property \(\rho(J^*) < 1\). Interestingly, GRIEWANK \([5]\) proved that unless they happen to coincide with those of \(G_y\), the eigenvalues \(\gamma\) of \(J^*\) solve the following nonlinear eigenvalue problem:

\[
\det((\gamma - 1)B_* + H(\gamma)) = 0 \quad (2.65)
\]

where

\[
H(\gamma) = Z^T(\gamma) \begin{pmatrix}
N_{yy} & N_{yu} \\
N_{yu} & N_{uu}
\end{pmatrix} Z(\gamma) \quad \text{and} \quad Z(\gamma) = \begin{pmatrix}
(\gamma I - G_y)^{-1}G_u \\
I
\end{pmatrix} \quad (2.66)
\]

Note, in view of an rSQP method (ref. section 2.2.2) \(N_u\) can be identified as the reduced gradient based on the null space basis

\[ Z(1) = Z = \left((I - G_y)^{-1}G_u\right). \quad (2.67) \]
and $H(1)$ is the corresponding reduced Hessian (2.29). Therefore, the positive definiteness property of $H(1)$, i.e. $H(1) > 0$, corresponds to the usual second order necessary condition at constrained minima. Now one has to look for conditions that exclude the existence of eigenvalues with modulus greater or equal to 1. In [5] it is shown that it is necessary that $B_k$ is symmetric positive definite and $B_k > \frac{1}{2} H(-1)$ to ensure that eigenvalues stay less than 1, though these conditions are not sufficient to exclude eigenvalues less than -1. If the optimization problem is strongly convex then $H(-1)$ is like all $H(\gamma)$ positive definite so it is reasonable to consider it as a candidate for a preconditioner $B_k$. It was shown in elliptic test problems that the choice $B_k \approx H(-1)$ leads to full-step convergence, whereas the obvious choice $B_k \approx H(1)$, like in rSQP methods, needs steps of the form $u_{k+1} = u_k - \tau B_k^{-1} N_u$ with a properly chosen $\tau \in \mathbb{R}$ in order to converge. Hence, deriving a preconditioner which ensures even local convergence seems to be quite difficult.

2.3.4 Descent on the Exact Penalty Function

In [8] the authors introduced a penalty function of doubly augmented Lagrangian type to solve the optimization problem (2.51) - (2.52). They are looking for descent on

$$L^\alpha(y, \lambda, u) := \frac{\alpha}{2} \|G(y, u) - y\|^2 + \frac{\beta}{2} \|N_y^T(y, \lambda, u) - \lambda\|^2 + N(y, \lambda, u) - \lambda^T y$$  (2.68)

where the weighting coefficients $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are strictly positive. Furthermore, in [8] they proved that $L^\alpha$ is an exact penalty function [25] provided the so-called correspondence condition

$$\alpha \beta \Delta G^T \Delta G_y > I + \beta N_{yy} \quad \text{with} \quad \Delta G_y = I - G_y$$  (2.69)

holds. This implies that all stationary points of the constrained problem (2.51) - (2.52) are also stationary points of the penalty function $L^\alpha$ and the Hessian of $L^\alpha$ is positive definite at a stationary point if and only if the reduced Hessian $H(1)$ introduced in the previous section is positive definite. If the step increment vector associated to iteration (2.58)

$$s(y, \lambda, u) := \begin{pmatrix} \Delta y \\ \Delta \lambda \\ \Delta u \end{pmatrix} = \begin{pmatrix} G(y, u) - y \\ N_y^T(y, \lambda, u) - \lambda \\ -B^{-1} N_u^T(y, \lambda, u) \end{pmatrix}$$  (2.70)

is introduced it can be shown (ref. [8]) that the latter yields descent for all “large” positive preconditioner $B$ if

$$\sqrt{\alpha \beta (1 - \rho_C)} > 1 + \frac{\beta}{2} \theta \quad \text{with} \quad \theta := N_{yy}$$  (2.71)

The term “large” is defined in the sense that $B > B_0$, where $B_0$ has yet to be defined. For obvious reasons inequality (2.71) is known as descent condition. It is seen from this condition that we essentially achieve descent if $\alpha$ is chosen sufficiently large. However, it is known for penalty functions that too large values will slow down the convergence speed. A particular choice of $\alpha$ and $\beta$ can be derived from (2.71) by trying to make $\alpha$ as small as possible while still satisfying this condition. Therefore we may minimize $\alpha$ as a function of $\beta$ such that

$$\min\limits_\beta \sqrt{\alpha} \equiv \frac{1 + \frac{\beta}{2} \theta}{(1 - \rho_C)\sqrt{\beta}}$$  (2.72)

which leads to the following weighting coefficient values:

$$\beta = \frac{2}{\theta} \quad \text{and} \quad \alpha = \frac{2\theta}{(1 - \rho_C)^2}.$$  (2.73)

We still have to define $B_0$ in order to be able to construct a suitable $B$. To this end we note that the gradient of $L^\alpha$ is given by

$$\nabla L^\alpha(y, \lambda, u) = \begin{pmatrix} \nabla_y L^\alpha \\ \nabla_\lambda L^\alpha \\ \nabla_u L^\alpha \end{pmatrix} = \begin{pmatrix} \alpha \Delta y^T (G_y - I) + \lambda^T (I + \beta N_{yy}) \\ \beta \Delta \lambda^T (G_y - I) + \Delta y^T N_{yy} + N_u \\ \alpha \Delta y^T G_u + \beta \lambda^T N_{uu} + N_u \end{pmatrix}$$  (2.74)
or in different notation
\[ \nabla L^a(y, \lambda, u) = -Ms(y, \lambda, u), \quad \text{where} \quad M = \begin{pmatrix} \alpha \Delta G_y^T & -I - \beta N_{yy} & 0 \\ -I & \beta \Delta G_y & 0 \\ -\alpha G_u^T - \beta N_{yu}^T & B \end{pmatrix}. \] (2.75)

The step \( s \) will give descent on (2.68) if it satisfies the condition
\[ s^T \nabla L^a(y, \lambda, u) = s^T Ms < 0. \] (2.76)

Hence, \( B \) needs to be defined in such a way that the matrix \( M \) will be positive definite. The difficulty in deriving a suitable preconditioner lies in the goal to define \( B \) such that \( B^{-1} \) is as large as possible, in order to make significant design corrections using \( \Delta u = -B^{-1}N_u^T \), while still guaranteeing that \( s \) yields descent on the doubly augmented Lagrangian. Therefore, in [8] they used the symmetric part
\[ M_s := \frac{1}{2} (M^T + M) \] (2.77)
to essentially come up with an lower bound for \( B \):

**Theorem 2.4 (Descent on \( L^a \))**

If \( \alpha \) and \( \beta \) satisfy the descent condition (2.71) then any preconditioner fulfilling
\[ B \geq B_0 := \frac{1}{\sigma} \left( \alpha G_u G_u + \beta N_{yu}^T N_{yu} \right) \] (2.78)
yields descent on \( L^a \) defined in (2.68), where \( \sigma \in \mathbb{R} \) is defined as
\[ \sigma = 1 - \rho_G - \frac{(1 + \frac{\beta}{2} \theta)^2}{\alpha \beta (1 - \rho_G)}. \] (2.79)

### 2.3.5 Explicit Choice of \( B \)

Using the lower bound for the design preconditioner \( B \) we can now derive an explicit form. Furthermore it is possible to establish a connection to the Hessian of the augmented Lagrangian \( L^a \) with respect to the design. The following derivation is based on the paper of Hamdi and Griewank [9]. Since we seek for a lower value of \( L^a \) in each iteration \( k \) it is natural to consider a \( \Delta u \) such that
\[ \min_{\Delta u} L^a(y_k + \Delta y, \lambda_k + \Delta \lambda, u_k + \Delta u). \] (2.80)

Using the quadratic approximation of \( L^a \) and the step increment vector \( s \) introduced in (2.70) we can rewrite (2.80) as
\[ \min_{\Delta u} s^T \nabla L^a(y_k, \lambda_k, u_k) + \frac{1}{2} s^T \nabla^2 L^a(y_k, \lambda_k, u_k)s. \] (2.81)

or, equivalently,
\[ \min_{\Delta u} E(\Delta u), \] (2.82)

where \( E(\Delta u) \) is the quadratic form written as
\[ E(\Delta u) = \Delta u^T (\nabla_u L^a + \nabla_{uy} L^a \Delta y + \nabla_{u\lambda} L^a \Delta \lambda) + \frac{1}{2} \Delta u^T \nabla_{uu} L^a \Delta u \]
\[ \approx \Delta u^T \nabla_u L^a(y_k + \Delta y, \lambda_k + \Delta \lambda, u_k) + \frac{1}{2} \Delta u^T \nabla_{uu} L^a \Delta u \] (2.83)
Consequently, if we assume that $\nabla L^a_{uu}$ is positive definite, the minimizer of $E$ is given by

$$\Delta u = -\nabla^{-1} L^a(y_k, \lambda_k, u_k) \nabla_u L^a(y_k + \Delta y, \lambda_k + \Delta \lambda, u_k).$$

(2.84)

If we are at a stationary point ($\Delta y = 0, \Delta \lambda = 0$) we have

$$\nabla_{uu} L^a = \alpha G_u^T G_u + \beta N_{yu}^T N_{yu} + N_{uu},$$

(2.85)

thus it is reasonable to use $B \approx \alpha G_u^T G_u + \beta N_{yu}^T N_{yu} + N_{uu}$. Indeed, in line with theorem 2.4 a suitable design preconditioner is given by

$$B := B_0 + \frac{1}{\sigma} N_{uu} = \frac{1}{\sigma}(\alpha G_u^T G_u + \beta N_{yu}^T N_{yu} + N_{uu}).$$

(2.86)

### 2.3.6 Notes on Convergence

In contrast to the approximate Null-space iteration introduced in the previous section, here even global convergence of the coupled iteration (2.58) can be proven for the general case. However, it is based on the assumption that we apply a line-search procedure to ensure monotonic decrease of the augmented Lagrangian such that all iterates during the optimization lie in the bounded level set $N_0$ of $L^a$ defined by

$$N_0 := \{(y, \lambda, u) \text{ such that } L^a(y, \lambda, u) \leq L^a(y_0, \lambda_0, u_0)\},$$

(2.87)

where $(y_0, \lambda_0, u_0)$ denotes the starting iterates. For global convergence using line-search methods it then needs to be shown that the angle between the steepest descent direction $-\nabla L^a$ and the search direction $s_k$, $\varphi_k$, is always bounded away from $\pi/2$ [25]. This is equivalent to the condition

$$\cos \varphi_k := -\frac{s^T(y_k, \lambda_k, u_k) \nabla L^a(y_k, \lambda_k, u_k)}{\|s(y_k, \lambda_k, u_k)\| \|\nabla L^a(y_k, \lambda_k, u_k)\|} > 0.$$ 

(2.88)

Using all of the assumptions from the previous subsections the following theorem can be shown to hold:

**Theorem 2.5**

If all level sets of $L^a$ are bounded and $N_{uu}$ is positive definite, then there exists a $C > 0$ such that

$$\cos \varphi_k \geq C > 0, \quad \forall k \in \mathbb{N}^+,$$

(2.89)

where $\varphi_k$ is defined through (2.88).

The proof can be found in [9] and is essentially based on the symmetric matrix $M_k$ defined in equation (2.77). In the same paper it is also shown that the level sets of $L^a$ are bounded provided $f$ fulfills reasonable conditions.
In the preceding chapter the theoretical foundations of One-Shot optimization were considered. For the application we need to discuss some aspects that will arise if one wants to actually implement the method. We begin with a summary of the necessary steps for the general approach based on the theory in section 2.3. First, we need a proper initialization of the variables. The state $y_0$ is typically initialized using the far field values specified at the boundaries of the domain. A reasonable value for $\lambda_0$ is for example zero. However, the final state and adjoint solution should not depend on this value. In contrast, the final design depends heavily on the chosen initial design $u_0$ as one can only expect to reach a local optimum, which is quite typical for gradient based methods. Then we do the following steps for $k = 0, 1, 2, \ldots$:

1. One step of the solver for the state equation: $y_{k+1} = G(y_k, u_k)$.
   This corresponds simply to one iteration of the given flow solver. As opposed to other rSQP methods the explicit knowledge of $G$ is not required for the next steps.

2. One step of the adjoint iteration: $\lambda_{k+1} = N_y(y_k, u_k)$.
   We assume that the necessary gradient of $N$ is evaluated along with the state iteration step by using Algorithmic Differentiation so that no further numerical difficulties will arise here for the time being.

3. Design update: $u_{k+1} = u_k - B_k^{-1} N_u(y_k, \lambda_k, u_k)$.
   Again, getting $N_u$ will not pose a problem at the moment. The main concern here is a suitable definition of the design preconditioner $B_k$. This will be addressed in the next section.

4. Deformation of the volumetric mesh.
   As a boundary of the computational domain was altered in the previous step it is necessary to adapt the volumetric mesh accordingly. Section 3.3 deals with this problem.

5. Convergence check. This essentially means to check whether $\Delta y$, $\Delta \lambda$ and $\Delta u$ are small enough.

Especially the definition of a practical design preconditioner that has decent computational complexity is of great importance. But also additional constraints complementing the optimization problem need to be discussed as they force the resulting design to be physically significant.

### 3.1 Hessian Approximation by the Laplace-Beltrami Operator

We have seen that by using the point of view of an rSQP method a preconditioner $B_k$ should approximate, at least asymptotically, the reduced Hessian $H(1)$. However, in case of an reduced Quasi-Newton method it is not clear whether this obvious choice is effective. In fact, global convergence can only be proved at the moment for a special preconditioner based on an augmented Lagrangian (ref. section 2.3). But in practice,
the presented preconditioner is not very efficient as we need second derivatives. Furthermore, it may not be possible to preserve the regularity of the shape. So there is a demand for new types of preconditioners that also work well in case of Free-Node parametrizations. Unlike in NAND methods, where a line search can compensate a moderate approximation, the preconditioner has to be well scaled in this case. Nevertheless, in this section we present a preconditioner that already has proven to be quite effective, although in theory it is based on an approximation of the reduced Hessian $H(1)$.

A natural choice for the preconditioner would be the (discrete) shape Hessian of the problem [24]. But even with Algorithmic Differentiation the complexity for the computation is still way too high to be feasible. In the thesis of SCHMIDT [28] it is shown that a good shape Hessian approximation must mimic a pseudodifferential operator. By an analysis of the operator symbol of the Hessian for Euler flow it was found that it is best to approximate the Hessian by an anisotropic operator in chord and span direction. Chord-wise, or in 2D, the Hessian resembles a diffusive operator like the Laplacian. Based on this fact, the Laplace-Beltrami operator is well suited because it can easily discretized using finite differences. This method is also commonly denoted as Sobolev-Smoothing and is often used just to smooth the reduced gradient in gradient-descent methods [16] instead of treating it as a shape Hessian approximation. Commonly, in shape calculus only variations in the direction of the normal of the surface are considered. Therefore we first of all need to project the gradient $N_u \in \mathbb{R}^m$ into this subspace. In two dimensions we can achieve this by

$$g := P(u_k)N_u(y_k, \lambda_k, u_k) \in \mathbb{R}^\frac{m}{2},$$

where $g$ is referred to as the shape gradient. The projection $P(u_k)$ is defined as

$$P(u_k) := \begin{pmatrix} n_1^{(0)} & n_2^{(0)} & \dots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & \cdots & n_1^{(m-1)} & n_2^{(m-1)} \end{pmatrix} \in \mathbb{R}^{\frac{m}{2} \times m}.$$  

$(n_1^{(i)}, n_2^{(i)})$ are the unit normal components in $x_1$ and $x_2$ direction at surface mesh node $i$, respectively. The normal vectors are obviously dependent on the current design, thus $P$ has a direct dependency on $u_k$. The shape Hessian $H_s$ can then be approximated as

$$H_s \approx (\varepsilon \Delta^{k}_{\Gamma} + I) \in \mathbb{R}^{\frac{m}{2} \times \frac{m}{2}}$$

where $I \in \mathbb{R}^{\frac{m}{2} \times \frac{m}{2}}$ is the identity for $\frac{m}{2}$ surface mesh points. The variable $\varepsilon \in \mathbb{R}$ is a smoothing parameter and allows some fine-tuning that is especially necessary since the use of the Laplace-Beltrami operator will likely result in an under-smoothing of low frequencies and an over-smoothing of high frequencies (see [28]). However, how to choose $\kappa$ to achieve the best possible approximation is still not quite clear. The matrix $\Delta^{k}_{\Gamma}$, representing second derivatives of the surface quantity $g_i$ at mesh node $i$, is given by the central difference stencil

$$g_i \frac{2}{(\xi_i - \xi_{i-1})(\xi_{i+1} - \xi_i)} - g_{i-1} \frac{2}{(\xi_i - \xi_{i-1})(\xi_{i+1} - \xi_{i-1})} - g_{i+1} \frac{2}{(\xi_{i+1} - \xi_i)(\xi_{i+1} - \xi_{i-1})},$$

where the $\xi \in \mathbb{R}$ is the tangential coordinate on the surface $\Gamma$ (see figure 3.1). Since the latter is already parametrized by linear segments it is reasonable to define the $\xi_i$ using arc lengths:

$$\xi_0 = 0,$$

$$\xi_i = \sum_{j=1}^{i} \sqrt{\left(x_1^{(j)} - x_1^{(j-1)}\right)^2 + \left(x_2^{(j)} - x_2^{(j-1)}\right)^2}, \quad i = 1, \ldots, \frac{m}{2} - 1$$

The $(x_1^{(i)}, x_2^{(i)})$ are the coordinates of the consecutively numbered surface mesh nodes $i$. Thus, if we solve the linear system

$$(I - \varepsilon \Delta^{k}_{\Gamma}) \tilde{g} = g,$$
we can update the design using
\[ u_{k+1} = u_k - \tau P^T(u_k)\tilde{g}, \] (3.8)
where \( \tau > 0 \) is a suitable scaling parameter. Note that this corresponds to the reduced Hessian approximation
\[ H(1) \approx N_{uu} \approx B_k \equiv \tau P^{T}(u_k)(\varepsilon \Delta_k^2 + I)P^{-1}(u_k). \] (3.9)
The linear system (3.7) features a tridiagonal matrix and hence a simple direct method like Gauss-Elimination provides sufficient performance. Despite the shape Hessian approximating nature of the Laplace-Beltrami operator that will lead to a convergence acceleration of the One-Shot method, this approach features another major advantage for Free-Node parametrizations. The reduced gradient is known to have a lower regularity than the shape (typically it will be in the space \( L^2(\Gamma) \) only [24]). If we use it directly to update the design (as for example in gradient descent methods) we can not ensure that the resulting shape will stay in the same regularity class as the initial shape and hence non-physical designs may occur [29]. On the contrary, equation (3.7) is the discretized version of the Laplace equation whose solutions are in the Sobolev space \( H^2 \) that is, they will be continuously differentiable up to order 2 and therefore the shape will also stay in that same regularity class in each iteration. Note, unlike in other Hessian approximations this is independent of the approximation quality.

### 3.2 Additional Constraints

If we apply the method described in the previous sections to problem (2.1) - (2.2) with \( f(y,u) \) being the drag coefficient the resulting shape will certainly have a lower drag count than the initial shape. However, to prevent a degeneration of the shape we have to add at least some geometrical constraints to the optimization problem. There are many approaches to include arbitrary inequality or equality constraints, but here we restrict ourselves to a constant volume constraint using the penalty multiplier method [25]. Of course, in practice also constraints involving the state variables, like a lift constraint, are necessary, but their handling is much more challenging in the One-Shot context since we only have approximations to the reduced gradients in each iteration.

In addition to the constraint \( c(y,u) = 0 \) we now also consider the scalar constraint \( h(u) = 0 \) with \( h : U \to \mathbb{R} \). The easiest way to incorporate this constraint is the use of a quadratic penalty term added to the cost function (2.1), i.e. we define a penalty function \( Q(y,u;\mu) \) with
\[ Q(y,u;\mu) := f(y,u) + \frac{\mu}{2} h^2(u). \] (3.10)
Then we use the theory from section 2.3 with \( Q \) instead of \( f \). By driving the penalty parameter \( \mu \in \mathbb{R} \) to infinity we penalize the constraint violations with increasing severity and thereby force the minimizer of the penalty function to the feasible region. It can be shown that the minimizers \( u^* \) of \( Q \) do not quite satisfy the condition \( h(u^*) = 0 \). Instead they are perturbed slightly to approximately satisfy
\[ h(u^*) = -\frac{1}{\mu} \eta^*, \] (3.11)
where \( \eta^* \in \mathbb{R} \) is the exact Lagrangian multiplier for the constraint \( h(u) \). Hence, this perturbation vanishes if \( \mu \to \infty \). However, the optimization becomes more and more difficult to perform when \( \mu \) becomes large.
A better approach is to extend the Lagrangian (2.53) using an estimate $\eta$ to $\eta^*$:

$$L(y, \lambda, u, \eta; \mu) := N(y, \lambda, u) - y^T \lambda - \eta h(u) + \frac{\mu}{2} h^2(u).$$

(3.12)

We can see from this definition that if $\eta$ is set to zero, we obtain the Lagrangian based on the quadratic penalty function $Q$. On the other hand, if we use the exact value $\eta = \eta^*$, it can be shown that we get the correct minimum of the original optimization problem for any positive value of $\mu$ [12]. Then there is no need to use a small value. Using the first order necessary conditions we get instead of (2.57)

$$N_u(y^*, \lambda^*, u^*) - (\eta - \mu h(u^*)) h_u(u^*) = 0,$$

(3.13)

and if we would use the exact multiplier

$$N_u(y^*, \lambda^*, u^*) - \eta^* h_u(u^*) = 0.$$  

(3.14)

Comparing equation (3.13) with (3.14) we expect that

$$\eta - \mu h(u^*) \to \eta^*$$

(3.15)

as the minimum is approached. Instead of an infeasibility proportional to $1/\mu$ as in (3.11) we now have

$$h(u^*) \approx (\eta - \eta^*) \frac{1}{\mu},$$

(3.16)

that is much smaller if $\eta$ is close to $\eta^*$. Hestenes [12] suggested to use an initial guess for $\eta$, solve the optimization problem and update $\eta$ using equation (3.15). However, in the One-Shot sense we perform the update in each iteration $k$. First, we update the design vector $u$ based on equation (3.13), i.e.

$$u_{k+1} = u_k - B_k^{-1} N_u(y_k, \lambda_k, u_k) - (\eta_k - \mu h(u_k)) B_k^{-1} h_u(u_k),$$

(3.17)

then, we update the Lagrangian multiplier estimate:

$$\eta_{k+1} = \eta_k - \mu h(u_k),$$

(3.18)

with $\mu$ being constant during the optimization. Figure 3.2 shows a typical developing of the infeasibility $|h(u_k)|$ for a volume constraint during the optimization. Here, $h(u_k)$ measures the deviation from a prescribed volume. The final infeasibility for the Penalty multiplier method is almost three orders of magnitude lower than for the Quadratic penalty method, which indicates that $\eta_k$ will be a good estimate to $\eta^*$ at the end of the optimization.

### 3.3 Mesh Deformation

Since a remeshing of the computational grid after each design step is extremely costly, grid deformation methods are commonly used. They move the mesh nodes according to a deformation of the boundary. Here will give a short introduction on the method that models the mesh as an elastic solid using the equations of linear elasticity. It is based on the work of Dwight [2] and is implemented in the SU2 framework. Note that the following is completely independent of the previous sections, therefore the variables used here are not to be confused with the ones used in the description of the One-Shot method.

The equations of linear elasticity govern small displacements $d(x) \in \mathbb{R}^3$ of an elastic solid subject to body forces and surface tractions. They may be written as

$$\text{div} \sigma = f \text{ on } \Omega,$$

(3.19)

where $f = f(x) \in \mathbb{R}^3$ is some body force and $\Omega \subset \mathbb{R}^3$ the computational domain. $\sigma = \sigma(x) \in \mathbb{R}^{3\times3}$ is the stress tensor given by the constitutive relation

$$\sigma = \lambda \text{Tr}(\epsilon) I + 2\mu \epsilon.$$  

(3.20)
### 3 Numerical Aspects in Shape Optimization

#### Figure 3.2: Infeasibility $|h(u_k)|$ for the Quadratic penalty method and the Penalty multiplier method using a volume constraint with $\mu = 100$.

| Iteration | $|h(u_k)|$ |
|-----------|------------|
| 0         | 1000       |
| 500       | 10         |
| 1000      | 100        |

$\mu \in \mathbb{R}$ and $\lambda \in \mathbb{R}$ are the so-called Lamé constants and are typically a property of the elastic material. $\epsilon = \epsilon(x) \in \mathbb{R}^{3 \times 3}$ is the rate of strain tensor defined as

$$
\epsilon = \frac{1}{2} (\nabla d + \nabla d^T).
$$

Finally, equation (3.19) is complemented by the Dirichlet boundary conditions

$$
d(x) = 0 \quad \text{on} \quad \partial \Omega_f,
$$

$$
d(x) = \Delta u(x) \quad \text{on} \quad \partial \Omega_s,
$$

with $\partial \Omega_f$ and $\partial \Omega_s$ denoting the far-field and surface boundary and consequently $\partial \Omega_f \cup \partial \Omega_s = \partial \Omega$. Thus, the far-field boundary is fixed and the surface displacement equals the design update calculated during the One-Shot iteration. In linear elasticity theory the Lamé constants are commonly defined in terms of Young’s modulus $E > 0$ and Poisson’s ratio that is a measure of how much a material shrinks in the lateral direction as it extends in the axial direction. However, for the application to mesh deformation it is known that by taking $E$ proportional to the inverse of the cell volumes the robustness of the method can be increased considerably. To still admit rigid body rotations in this case as solutions of (3.19) the Lamé constants are taken to be

$$
\lambda = -E, \quad \mu = E.
$$

As a result equation (3.19) can no longer be seen as a model of elasticity, but its behavior is still similar.

To solve the governing equations a finite element discretization using a Galerkin ansatz can be applied. The trial and test spaces are defined as

$$
D^h = \{d^h|d^h \in (H^h(\Omega))^3, d^h = 0 \text{ on } \partial \Omega_f, d^h = \Delta u \text{ on } \partial \Omega_s\},
$$

$$
\Phi^h = \{\phi^h|\phi^h \in (H^h(\Omega))^3, \phi^h = 0 \text{ on } \partial \Omega\}.
$$
where \((H^h(\Omega))^3 \subset (H(\Omega))^3\). By multiplying (3.19) with \(\phi^h\) and integration by parts the finite dimensional problem can then be stated as: find \(d^h \in D^h\) such that

\[
\int_{\Omega} \epsilon(\phi^h) \sigma(d^h) d\Omega = \int_{\Omega} \phi^h \cdot f d\Omega, \quad \forall \phi^h \in \Phi^h.
\]

(3.26)

\((H^h(\Omega))^3\) can be taken as the space of linear functions which simplifies the implementation considerably. The resulting linear system is then solved using a Biconjugate Gradient Stabilized Method (BCGSTAB) [31]. Furthermore it is assumed that the resolution of the already present mesh for the flow computation is sufficient to resolve the features of the elastic solution. Figure 3.3 shows the initial mesh and the deformed mesh after an optimization process conducted in chapter 5.
Algorithmic Differentiation and Implementation

In the preceding chapters we have always assumed that the calculation of the occurring gradients does not pose as a serious problem. In fact, the function \( G(y,u) \) itself is rather complex in reality and we can expect that the derivatives are even more challenging to compute. However, if \( G(y,u) \) is given as a function in a computer program, in this case by the SU\(^2\) flow solver, we can indeed get the Jacobian (or the transposed of it) times an arbitrary direction with a rather low computational complexity. In this chapter we will give a short introduction to Algorithmic Differentiation and show how the One-Shot method is actually implemented and integrated into the existing framework.

4.1 Introduction to Algorithmic Differentiation

Algorithmic Differentiation or also called Automatic Differentiation is a way to calculate the derivative of a function by means of the transformation of the underlying program which calculates the numerical values of this function. As distinguished from symbolic differentiation an explicit expression for the derivative is never formed. An advantage over FD is that no truncation errors are present, thus the numerical value can be determined up to machine accuracy.

4.1.1 Forward Propagation

Suppose we have a function \( f \in C^1 : U \subset \mathbb{R}^n \rightarrow \mathbb{R}^m \) and the expression
\[
y = f(x),
\]
that is, \( x \in U, y \in \mathbb{R}^m \). The evaluation of the function \( f \) can be represented as a sequence of \( l \) elementary functions \( \varphi_i \) and intermediate values \( v_i \) like it is shown in table 4.1. In computer programs those functions may be for example intrinsic functions provided by the compiler. The precedence relation \( j \prec i \) means that

\[
\begin{align*}
v_{i-n} &= x_i, & i &= 1\ldots n \\
v_i &= \varphi_i(v_j)_{j<i}, & i &= 1\ldots l \\
y_{m-i} &= v_{i-1}, & i &= m-1\ldots 0
\end{align*}
\]

Table 4.1: General evaluation procedure.

\( i \) depends directly on \( j \). Applying the chain rule to the \( v_i \) in table 4.1 results in
\[
\dot{v}_i = \sum_{j<i} \frac{\partial}{\partial v_j} \varphi_i(u_i) \dot{v}_j
\]

(4.2)
with the abbreviation \( u_i = (v_j)_{j<i} \in \mathbb{R}^n \), i.e. the vector \( u_i \) is the concatenation of the \( v_j \) on which \( \varphi_i \) depends. Note that for building the derivative of the general evaluation procedure shown in table 4.1 we need the derivatives of the values \( v_{i-n}, i = 1 \ldots n \) as an input, that is, values for \( \dot{x}_i \). By formally applying the chain rule to the expression (4.1) we get

\[
\dot{y} := \frac{df}{dx}(x) \cdot \dot{x} = \sum_{i=1}^{n} \frac{\partial f(x)}{\partial x_i} \dot{x}_i
\]  

(4.3)

Thus we can get the derivative \( \frac{df(x)}{dx} \in \mathbb{R}^m \) by setting \( \dot{x}_i = 1 \) and \( \dot{x}_j = 0, j \neq i, j = 1, \ldots, n \). If equations (4.2) is combined with the evaluation trace one gets the forward propagation of gradients given in table 4.2.

<table>
<thead>
<tr>
<th>( v_{i-n}, \dot{v}_{i-n} )</th>
<th>( [x_i, \dot{x}_i] )</th>
<th>( i = 1 \ldots n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [v_i, \dot{v}_i] )</td>
<td>( [\varphi_i(v_j)<em>{j&lt;i}, \sum</em>{j&lt;i} \frac{\partial \varphi_i(u_i)}{\partial v_j} \dot{v}_j] )</td>
<td>( i = 1 \ldots l )</td>
</tr>
<tr>
<td>( [y_{m-i}, \dot{y}_{m-i}] )</td>
<td>( [v_{m-i}, \dot{v}_{m-i}] )</td>
<td>( i = m-1 \ldots 0 )</td>
</tr>
</tbody>
</table>

Table 4.2: Forward propagation of gradients.

**Example 4.1**

Consider the function \( g : \mathbb{R}^2 \rightarrow \mathbb{R}^2 \) and the expression:

\[
y = g(x_1, x_2) := \left( \frac{\sin(x_1) + e^{x_2}}{\sqrt{x_1}} \right)
\]

(4.4)

The evaluation procedure for this function looks like the following:

\[
\begin{align*}
v_{-1} &= x_1, & \dot{v}_{-1} &= \dot{x}_1, \\
v_0 &= x_2, & \dot{v}_0 &= \dot{x}_2, \\
v_1 &= \sin(v_{-1}), & \dot{v}_1 &= \cos(v_{-1}) \dot{v}_{-1}, \\
v_2 &= e^{v_0}, & \dot{v}_2 &= e^{v_0} \dot{v}_0, \\
v_3 &= \sqrt{v_0}, & \dot{v}_3 &= \frac{1}{2 \sqrt{v_0}} \dot{v}_0, \\
v_4 &= \frac{1}{v_{-1}}, & \dot{v}_4 &= -\frac{1}{v_{-1}^2} \dot{v}_{-1}, \\
v_5 &= v_1 + v_2, & \dot{v}_5 &= \dot{v}_1 + \dot{v}_2, \\
v_6 &= v_3 v_4, & \dot{v}_6 &= \dot{v}_3 v_4 + v_3 \dot{v}_4, \\
y_1 &= v_5, & \dot{y}_1 &= \dot{v}_5, \\
y_2 &= v_6, & \dot{y}_2 &= \dot{v}_6.
\end{align*}
\]

In order to evaluate the Jacobian \( \frac{dg}{dx}(x_1, x_2) \in \mathbb{R}^2 \times \mathbb{R}^2 \) at a point \( (w_1, w_2)^T \) the evaluation trace has to be called with \( (x_1, x_2)^T = (w_1, w_2)^T \) and \( \dot{x} = e_i, i = 1, 2 \), where \( e_i \in \mathbb{R}^2 \) is the \( i \)-th Cartesian basis vector. Thus, in general \( n \) calls of the evaluation trace are necessary to accumulate the full Jacobian.

### 4.1.2 Reverse Propagation

For the forward mode the derivatives are propagated in the same direction as the corresponding elemental function values, i.e. we ask how an infinitesimal change in the input values propagates through the evaluation trace and affects the output. But we could also ask the other way round: How sensitive are the output values to a change in the input values? The latter is the basis for the reverse propagation. There are many ways in
which the reverse mode can be introduced, but in general it can be thought of as the backward application of the chain rule. A sophisticated derivation can be found in Griewank and Walther [6]. Here only the main results needed for the application are described.

For the application of the chain rule to the evaluation of the function \( f \) (table 4.1) it is useful to introduce the state transformation \( \Phi_i \) for each elemental function \( \varphi_i \):

\[
\mathbf{v}_i = \Phi_i(\mathbf{v}_{i-1}), \quad \Phi_i : \mathbb{R}^{n+l} \rightarrow \mathbb{R}^{n+l}
\]  

(4.5)

with

\[
\mathbf{v}_i := (v_{i-n}, \ldots, v_i, 0, \ldots, 0)^T \in \mathbb{R}^{n+l}
\]

In other words \( \Phi_i \) sets \( v_i \) to \( v_j \) for \( i \neq j \) unchanged. We can then write the expression (4.1) as the composition

\[
y = Q_m \Phi_i(\Phi_{i-1}(\ldots(\Phi_1(P^Tx))))
\]

(4.6)

where \( P_n \in \mathbb{R}^{n \times (n+l)} \) and \( Q_m \in \mathbb{R}^{m \times (n+l)} \) are the matrices that project an \((n+l)\)-vector onto its first \( n \) and last \( m \) components, respectively. The derivatives of the state transformations \( \nabla \Phi_i \) can be explicitly calculated and written as

\[
A_i := \nabla \Phi_i = I - e_{n+i}(\nabla \varphi_i(u_i) - e_{n+i})^T \in \mathbb{R}^{(n+i) \times (n+l)}
\]

(4.7)

Now equation (4.6) can be differentiated using the chain rule:

\[
\mathbf{y} = Q_m A_i A_{i-1} \ldots A_2 A_1 P_n^T \mathbf{x}
\]

(4.8)

Thus, the Jacobian of \( f \) can be written as

\[
\frac{df}{dx}(x) = Q_m A_i A_{i-1} \ldots A_2 A_1 P_n^T
\]

(4.9)

By transposing the product we obtain the adjoint relation

\[
\bar{x} = P_n A_1^T A_2^T \ldots A_{i-1}^T A_i^T Q_m^T \bar{y} = \left( \frac{df}{dx} \right)^T \bar{y}
\]

(4.10)

with \( \bar{x} \in \mathbb{R}^n, \bar{y} \in \mathbb{R}^m \) and the identity

\[
\bar{y} \bar{y} = \bar{x} \bar{x}.
\]

Let us look at equation (4.10) in more detail. Suppose we have a vector \( \mathbf{v}_i \) of adjoint quantities \( \mathbf{v}_j, 1 - n \leq j \leq l \) such that

\[
\mathbf{v}_i = A_1^T A_2^T \ldots A_{i-1}^T A_i^T Q_m^T \bar{y} = A_i^T \mathbf{v}_{i-1}.
\]

With definition (4.7) this becomes

\[
\bar{v}_i = \bar{v}_{i-1} + (\nabla \varphi_i(u_i) - e_{n+i}) e_{n+i}^T \bar{v}_{i-1}.
\]

Now we can analyze what happens to the adjoint quantity \( (\bar{v}_i)_j \) if we consider the \( i \)-th elemental function \( \varphi_i \) for \( i = l, \ldots, 1 \):

- Since \( (\nabla \varphi_i(u_i))_j = 0 \) and \( e_{n+i}^T \bar{v}_{i-1} = 0 \) for \( i \neq j \neq i \), \( \bar{v}_j \) is left unchanged if \( \varphi_i \) does not depend on \( v_j \).
- If \( i \neq j \) and \( j < i \) then \( (\nabla \varphi_i(u_i))_j = \frac{\partial \varphi_i}{\partial v_j} \) and \( e_{n+i}^T \bar{v}_{i-1} = \bar{v}_i \), thus, \( \bar{v}_j \) is incremented by \( \bar{v}_i \frac{\partial \varphi_i}{\partial v_j} \).
- \( \bar{v}_i \) is set to zero.

With this information it is possible to rewrite the adjoint relation (4.10) as an evaluation procedure like it was done for the forward evaluation. Since the matrix-vector products in equation (4.10) are calculated for \( i = l, l - 1, \ldots, 1 \) we have to go backward, or reverse, through the sequence of elementary functions in table 4.1. Since the intermediate values \( v_i \) are needed they have to be computed first by evaluating the sequence of elemental functions. Summarizing this yields the Reverse Propagation of Gradients shown in table 4.3 [see also 6].

As an input we need the vector \( \bar{y} \). At the end we have \( \bar{x} \), that is, the matrix-vector product \( \bar{x} = \nabla f(x)^T \bar{y} \).
Algorithmic Differentiation and Implementation

\[ v_{i-n} = x_i, \quad i = 1 \ldots n \]
\[ v_i = \varphi_i(v_j), \quad i = 1 \ldots l \]
\[ y_{m-i} = v_{l-i}, \quad i = m-1 \ldots 0 \]
\[ \bar{v}_{i} = \bar{y}_{m-i} \quad i = 0 \ldots m-1 \]
\[ \bar{v}_j = \bar{v}_j + \bar{v}_i \frac{\partial \varphi_i(u)}{\partial v_j}, \quad j \prec i, \quad i = l \ldots 1 \]
\[ \bar{x}_i = \bar{v}_{i-n}, \quad i = n \ldots 1 \]

Table 4.3: Reverse propagation of gradients.

Example 4.2

Consider again expression (4.4). Application of the Reverse Propagation of Gradients to this equation results in the following trace:

\[ v_{-1} = x_1, \]
\[ v_0 = x_2, \]
\[ v_1 = \sin(v_{-1}), \]
\[ v_2 = e^{v_0}, \]
\[ v_3 = \sqrt{v_0}, \]
\[ v_4 = \frac{1}{v_{-1}}, \]
\[ v_5 = v_1 + v_2, \]
\[ v_6 = v_3v_4, \]
\[ y_1 = v_5, \]
\[ y_2 = v_6, \]
\[ y_0 = y_2, \]
\[ \bar{v}_5 = \bar{y}_1, \]
\[ \bar{v}_4 = \bar{v}_6v_3, \]
\[ \bar{v}_3 = \bar{v}_6v_4, \]
\[ \bar{v}_1 = \bar{v}_5, \]
\[ \bar{v}_2 = \bar{v}_5, \]
\[ \bar{v}_{-1} = -\bar{v}_4 \frac{1}{v_{-1}}, \]
\[ \bar{v}_0 = \bar{v}_4 \frac{1}{2\sqrt{v_0}}, \]
\[ \bar{v}_0 = \bar{v}_2e^{v_0}, \]
\[ \bar{v}_{-1} = \bar{v}_1\cos(v_{-1}), \]
\[ \bar{x}_2 = \bar{v}_0, \]
\[ \bar{x}_1 = \bar{v}_{-1}. \]

It is assumed that each \( \bar{v}_i \) has been initialized with zero. Similar to the forward propagation we have to set \( \bar{y} = e_i \) for \( i = 1, 2 \) to accumulate the full Jacobian. But here in general \( m \) calls of the evaluation trace are necessary. Thus, if we have a function \( f: \mathbb{R}^n \rightarrow \mathbb{R}^m \) and \( m \ll n \), the reverse propagation is the method of choice for the efficient calculation of the Jacobian.
4.1.3 Practical Implementation

The generation of the recursions for the forward and reverse propagation, respectively, is of pure mechanical fashion provided that the derivatives of the elemental functions are known. To incorporate this fact into an actual application there exist two basic computer science concepts, namely Source Code Transformation and Operator Overloading. For the first method the source code of the function to be differentiated is augmented by derivative assignments according to section 4.1.1 or section 4.1.2. This can be done either by hand or with a preprocessor.

Implementation of a Forward Propagation Tool

Since for this work the second approach is used it will be explained in a slightly more detailed way. First, consider the forward propagation. In order to calculate the derivative of a function we define a new data type or class that contains the numerical value of \( v_i \) and \( \dot{v}_i \). A very basic implementation of an \texttt{fdouble}-class is shown in the C++ listing 4.1. Of course in a real implementation the data fields \texttt{value} and \texttt{dot_value} would be declared as \texttt{private}. Then overloaded versions of the arithmetic operations for scalars are implemented. These overloaded operations must manipulate \( \dot{v}_i \) according to the chain rule. As a last step any floating point program variable whose derivatives are needed is redeclared to be of the this new type of class. Then the gradient information of a function is generated along with the execution of this function if the derivatives of the inputs are properly initialized. In the C++ listing 4.2 the overloaded versions of the multiplication operator as well as of the sine function based on the \texttt{fdouble}-class are implemented exemplary.

One particular advantage of this approach is that all of the AD-specific functions can be concealed in suitable header files without modifying the original code. Though there are additionally some changes that cannot be avoided in order to initialize the independent variables and their derivatives and extracting the derivative values of the dependent variables. This is accomplished by calling functions that can set and return the \texttt{dot_value} data field of a \texttt{fdouble} variable.

```
class fdouble
{
    double value;
    double dot_value;
};
```

Listing 4.1: \texttt{fdouble}-class implementation for forward propagation.

```
const fdouble operator*(const fdouble &a, const fdouble &b)
{
    fdouble result;
    result.value = a.value * b.value;
    result.dot_value = a.dot_value * b.value + a.value * b.dot_value;
    return result;
}
fdouble sin(fdouble a)
{
    fdouble result;
    result.value = sin(a.value);
    result.dot_value = cos(a.value) * a.dot_value;
    return result;
}
```

Listing 4.2: Overloaded multiplication operator and sine function.
Implementation of a Reverse Propagation Tool

Now let us consider the reverse propagation. Like for the forward propagation we introduce a new data type. Though in this case it contains the numerical value and an identifier or index (see C++-listing 4.3 for `adouble-class` implementation). During the execution of the evaluation procedure we build up an internal representation of the computation, which we call `tape`. The tape is essentially an array consisting of tape entries `tape_entry` (Listing 4.4). A tape entry holds the operation code encoded as an integer (`oc`), a function value $v_i$ (`value`) and the corresponding value of $\bar{v}_i$ (`bar_value`). Furthermore it stores the values of the arguments of the specific operation (`arg1,arg2`). Additionally there is usually a numerical record to store the overwritten variables that is not shown here. Then, again, we define arithmetic operations of this new data type that correspond to the usual floating point operations. These operations calculate the floating point operations for $v_i$ as usual, but as a side effect they create a tape entry and thus record themselves and their arguments on the tape. We can then define a routine `interpret_tape()` that reverses through the tape and calculates the adjoint variables $\bar{v}_i$ correctly according to the second half of table 4.3. Just like in the forward implementation all floating point program variables must be redeclared to be of the new data type.

```
1 class adouble
2 {
3     double value;
4     int index;
5 };  
```

Listing 4.3: `adouble`-class implementation for reverse propagation.

```
1 class tape_entry
2 {
3     int oc;
4     double value;
5     double bar_value;
6     int arg1;
7     int arg2;
8 };  
```

Listing 4.4: `tape_entry`-class implementation.
4.2 Application to SU2

Now we turn to the actual application of AD to the flow solver SU$^2$. An important fact of the coupled iteration (2.58) is that the reverse propagation of gradients can be directly applied to calculate the gradients of $N(y, \lambda, u)$. In each iteration $k$ of the flow solver we evaluate

$$w = f(y_k, u_k) \in \mathbb{R},$$  \hspace{1cm} (4.11)

$$y_{k+1} = G(y_k, u_k) \in \mathbb{R}^n,$$  \hspace{1cm} (4.12)
where $f(y, u)$ is the target function. By application of the reverse propagation we get

$$\bar{y}_k = G_y^T(y_k, u_k)\bar{y}_{k+1} + f_y^T(y_k, u_k)\bar{w}, \quad (4.13)$$

$$\bar{u}_k = G_u^T(y_k, u_k)\bar{y}_{k+1} + f_u^T(y_k, u_k)\bar{w}. \quad (4.14)$$

Thus, if we set $\bar{y}_{k+1} \equiv \lambda_k^T$ and $\bar{w} \equiv 1$ we have the identities

$$N_y^T(y_k, \lambda_k, u_k) \equiv \bar{y}_k, \quad (4.15)$$

$$N_u^T(y_k, \lambda_k, u_k) \equiv \bar{u}_k. \quad (4.16)$$

Additional terms for the penalty multiplier method can easily be added to the evaluation of $w$. The flow solver can thereby be considered as a black-box that provides the values for $w$ and $\bar{y}_{k+1}$ and for the application it is not necessary to know the particular structure of the given functions $G(y, u)$ and $f(y, u)$. In [6] it is shown that the evaluation of $\bar{y}_k$ and $\bar{u}_k$ in theory does not exceed 4 times the operation count of evaluating (4.11) and (4.12). However, since we have to store all intermediate results of all arithmetic operations the memory requirement will be quite high.

### 4.2.1 Integration into the existing Framework

The Stanford University Unstructured (SU^2) code is not only a flow solver, but also a complete open-source software suite for the computational analysis and design to solve complex, multi-physics analysis and optimization tasks using arbitrary unstructured meshes. It is written in C++ and was build from scratch without the need of any external libraries which makes it an attractive target for the application of AD. Furthermore, since it leverages well-established object-oriented software architectures the implementation and extension of additional features is comparatively easy. Figure 4.1 gives an overview of the class hierarchy of the main component. The module SU2_CFD instantiates three basic classes, namely CConfig, COutput and CIntegration. CConfig reads the problem description and settings from the input file and COutput writes the output of the simulation in a user-specified format. The class CIntegration solves the particular governing equations using multi-grid or single-grid calculations. The core capabilities are embedded within CGeometry, CSolution and CNumerics that manage the geometry, the main solver functionality and the numerical methods, respectively. Each of the child class of CSolution represents a solver for particular set of governing equations, like for example for the Euler equations. For this work a new child class of CSolution was implemented that stores the adjoint solution $\lambda$ and the reduced gradient $N_u$. Although it does not need much functionality because the main calculations are done during the interpretation of the tape. However, this approach enables the use of the existing input and output structure. Additionally, a new class COneShot was introduced that combines for example the design preconditioner, the handling of constraints and the mesh deformation.

Differentiation of an existing code essentially means to augment the code to enable gradient calculations using AD. In this work the tool dco [22] was used for the differentiation of SU^2. It provides the necessary structures and procedures to build the computational graph according to the principles described in section 4.1.3 during the evaluation of (4.11) and (4.12). It is based on the operator overloading features and expression templates of the programming language C++. For the differentiation the used data types in SU^2 are then replaced by their overloaded counterparts provided by dco. A distinct interface enables the extraction and setting of the adjoint values of each variable inside the code. Furthermore, despite a new class, an iteration routine was implemented that coordinates the necessary steps for the One-Shot optimization. In listing 4.7 a slightly simplified version of this routine is shown. From line 3 to 9 the tape is set active, which means that all operations in between are stored on the tape provided that at least one of the involved variables is already registered on the tape, i.e. has a tape entry. Therefore, initial tape entries for $y_k$ and $u_k$ are created by calling the RegisterVariables function. Then, in line 7 the iteration of the flow solver is called or rather $w$ and $\bar{y}_{k+1}$ are evaluated (equations (4.11) and (4.12)). By setting the seedings, that is, the adjoint values of $y_{k+1}$ and $w$, we can then call the evaluateAdjoint routine of the AD interface that traverses through the tape and propagates the adjoints to the input variables using an algorithm similar to listing 4.6. Note, $y_k$ is seeded with the adjoint solution $\lambda_k$ from the previous iteration, for this reason the
function `InitializeSeeding` has the solution container as an argument. In line 15 the reduced gradient $N_u$ is extracted and stored in the adjoint solution structure (ADJFLOW_SOL index) by exploiting identity (4.16). Similarly, the adjoint solution $\lambda_k$ is updated by extracting $\bar{y}_k$ from the flow solution $y_k$ (at index FLOW_SOL in the solution container) by the routine `AdjointIteration` using (4.15). The design update can then be calculated by applying the preconditioner defined in section 3.1 (line 19). Finally, we can update the grid using the linear elasticity method described in section 3.3 by calling the `UpdateGrid` routine in line 21.

4.2.2 Comment on Performance

An interesting question is the performance of the differentiated version of $SU^2$. We have measured the average time of one iteration of the flow-solver for the non-differentiated version of the code and for one iteration of the coupled iteration (2.58) using the differentiated version. The average was taken over all iterations until the flow residual reached a certain threshold for the test case described in chapter 5. The values are shown in table 4.4. We essentially get a factor of 8. Although this might seem high at first glance,

<table>
<thead>
<tr>
<th></th>
<th>Flow Iteration</th>
<th>Coupled Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. Time</td>
<td>0.1559s (1.0)</td>
<td>1.3008s (8.34)</td>
</tr>
<tr>
<td>Memory</td>
<td>30.96MB (1.0)</td>
<td>1.159GB (38.33)</td>
</tr>
</tbody>
</table>

Table 4.4: Average time for one iteration and memory consumption.

keep in mind that we aim for a One-Shot optimization. Provided that the optimization has a low retardation factor $R$ and the mesh deformation is also reasonably fast, the overall costs of the optimization will be in the order of $8R$ times the run-time to get one flow solution. This can be a huge reduction compared to conventional NAND methods, especially in the case of a Free-Node parametrization. However, as expected, one drawback is the high memory overhead that is shown in table 4.4. For practical applications a factor of 38 is often unfeasible. But using for example advanced checkpointing techniques [6] and optimized linear solvers it is expected that this factor can greatly be reduced in the future.
void OneShotIteration (CGeometry *geometry_container, CConfig *config_container, CSolution **solution_container, CIntegration *integration_container, COneShot *oneshot_container)
{
    // Activate the tape.
    interfaceAD. activateTape();
    // Register solution y_k and design variables u_k on the tape.
    oneShot_container->RegisterVariables(solution_container, geometry_container);
    // Perform one iteration of the flow solver y_{k+1} = G(y_k,u_k) (note: the target function
    w = f(y_k,u_k) is also evaluated here).
    MeanFlowIteration(integration_container, geometry_container, solution_container, solver_container, config_container);
    // Deactivate the tape.
    interfaceAD. deactivateTape();
    // Set the seedings, i.e. the adjoint values of y_{k+1} and w.
    oneShot_container->InitializeSeeding(solution_container, geometry_container, config_container);
    // Interpretation of the tape.
    interfaceAD. evaluateAdjoint();
    // Extract and set the reduced gradient N_u.
    solution_container[ADJFLOW_SOL]->SetGradient(geometry_container, config_container);
    // Extract N_y and set lambda_{k+1}
    solution_container[ADJFLOW_SOL]->AdjustIteration(solution_container[FLOW_SOL], geometry_container, config_container);
    // Calculate the design update delta u using the design preconditioner.
    oneShot_container->SetDesignUpdate(solution_container, geometry_container, config_container);
    // Update the computational grid using the Linear Elasticity method and consequently set
    u_{k+1}
    oneShot_container->UpdateGrid(solution_container, geometry_container, config_container);
}

Listing 4.7: One-Shot Iteration in $SU^2$
CHAPTER 5
Test Cases

In this chapter we apply the One-Shot optimization to several test cases using the differentiated version of $SU^2$. They should not be considered as real world applications, but rather as a way to show the potential of One-Shot optimization combined with a complex flow solver and the performance of the consistent adjoint solver as well as of the design preconditioner based on the Laplace-Beltrami operator.

5.1 Transonic NACA0012

5.1.1 Physical and Numerical Definition

The first test case is based on the NACA0012 airfoil as initial design and Euler equations (1.25) as the governing model. The flow regime is transonic such that a strong shock exists on the upper side of the profile leading to a high drag coefficient solely due to the wave drag. The computational domain $\Omega$ is discretized using 10216 triangles leading to 5233 points whereof 200 points represent the airfoil. Thus, we have $\dim(u) = m = 400$ and $\dim(y) = n = 20932$. Figure 5.1 shows the whole mesh and a close-up view of the airfoil. The outer boundary is 20 times the profile length away from the airfoil such that free stream boundary conditions can be applied everywhere without having to worry about unphysical reflections that might influence the solution. The prescribed values are listed in table 5.1. All other quantities, like the velocity vector or density, are then computed using these values. On the airfoil the component of the velocity normal to the surface is set to zero.

| Table 5.1: Free-stream conditions for the transonic test case. |
| --- | --- | --- | --- |
| Mach Number ($Ma$) | Angle of Attack ($\alpha$)$^{[\circ]}$ | Pressure ($p_{\infty}$) $[N/m^2]$ | Temperature ($T_{\infty}$) $[K]$ |
| 0.8 | 1.25 | 101235 | 273.15 |

The convective terms in the Euler equations are discretized using the JST scheme described in section 1.3.1. The second and fourth order dissipation coefficients are chosen as $\kappa^{(2)} = 0.25$ and $\kappa^{(4)} = 0.01$. Time discretization follows along the lines of section 1.3.2. A multi-grid $W$-cycle with 3 levels is used for convergence acceleration. The CFL number for the local time stepping is set to $N_{CFL} = 4$ on the finest mesh and is reduced by an factor of 0.9 on each subsequent multi-grid level. A Lower-Upper-Symmetric-Gauss-Seidel (LU-SGS) [32] method is then applied for the solution of the linear system. It has been found that this solver features the best compromise between memory consumption and run-time in the AD setting.
5.1.2 Validation of the Reduced Gradient

A crucial part of the optimization algorithm is of course the correctness of the reduced gradient $N_u$. For the validation we consider a converged flow and adjoint solution $(y^*, \lambda^*)$ such that we can assume $\nabla_u f(y^*, u) = N_u(y^*, \lambda^*, u)$ for a fixed design $u$ (here the initial NACA0012). Another way to calculate $\nabla_u f(y^*, u)$ is a direct differentiation approach approximated by finite differences. Therefore we eliminate the dependency of $f(y, u)$ on $y$ by considering it as a particular solution of $c(y, u) = 0$, i.e. $y = y(u)$. Then we can approximate the components of $\nabla_u f(y(u), u))$ by using forward differences:

$$
(\nabla_u f(y(u), u))_i \approx \frac{f(y(u + h e_i), u + h e_i) - f(y(u), u)}{h}, \quad i = 0, \ldots, m - 1,
$$

(5.1)

where $e_i \in \mathbb{R}^m$ is the $i$-th Cartesian basis vector of $\mathbb{R}^m$ and $h > 0$ a suitable step size. This approach can be very defective as cancellation errors may deteriorate the result, thus different values of $h$ should be considered. However, it is also extremely costly as we need $m+1$ flow solutions for each evaluation. Therefore we compare only the $x_2$ components of the gradient for $h = 10^{-4}$ and $h = 10^{-5}$. This comparison is plotted in figure 5.2. The design variables are consecutively numbered from index 0 being the trailing edge to the leading edge at index 99 comprising the lower side and from index 100 to index 199 comprising the upper side of the airfoil. Despite some part on the lower side of the airfoil, the finite difference approximations qualitatively match the reduced gradient. The spikes, for example at index 30, exist because the influence of the ghost cells on the gradient is missing. These ghost cells extend the computational domain in the parallel computation and contain additional variables. The finite difference approximation is calculated using a script outside of the flow solver and hence does not know about the ghost cells. Nevertheless, this is equivalent to not communicating the reduced gradient in the ghost cells back to the domain, thus we get the same representation error in this case.

5.1.3 Direct and Adjoint Solution for the initial Design

In continuous adjoint methods the adjoint variables are often used as a sanity check of the results, although they do not have a physical meaning and this checks are often based on intuition. For discrete adjoint methods people tend to use the corresponding variables as well. For that purpose $\lambda$ can be partitioned
identical to $y$ to get the presumed adjoints of the conservative variables. However, this might be misleading as was noticed during the development of the present adjoint solver.

Since the flow solver determines the convergence in terms of the density residuals it would make sense to additionally consider the adjoint density residual for the convergence of the coupled iteration. So we will have a closer look on the convergence of iteration (2.60) using the initial design. In figure 5.3a the root-mean-square (RMS) residuals of the density and the corresponding adjoint density are plotted. As expected they both are very similar as the iterations have the same contraction factor in theory, although the adjoint density reaches the desired target residual of $10^{-10}$ a little bit earlier. The drag coefficient $C_D$, plotted in figure 5.3b along with the 2-norm of the reduced gradient, shows rapid convergence to the final value $C_D = 0.0213546301$. In contrast to that, the reduced gradient needs at least 300 iterations until it settles. This corresponds to density and adjoint density residuals of approximately $5 \cdot 10^{-6}$. This makes sense, since the reduced gradient approximation can only be as good as the corresponding approximation of the adjoint. Finally, we have $\|N_u\|_2 = 10.3730133199$. Unlike in other adjoint methods, relying on the residuals of the flow and adjoint variables without considering the convergence of the reduced gradient is therefore not feasible in this case.

In fact, depending on the numerical methods used in the flow solver, the resulting adjoint variables have entirely different values, whereas the reduced gradient stays almost the same. A drastic example is the use of the multi-grid method. Figure 5.4 shows the density adjoint for enabled (right) and disabled (left) multi-grid acceleration. Without the multi-grid method there is a clear structure visible that is similar to the adjoint in the continuous case. With enabled multi-grid acceleration, which is the standard setting for all computations in this chapter, there is almost no structure visible anymore in this data range, especially near the airfoil. Nevertheless, the resulting shape gradients (ref. (3.1)) plotted in figure 5.5 are indistinguishable. Hence, it even does not make much sense to consult the adjoint variables itself for any further analysis.

![Figure 5.2: Comparison of the components in $x_2$-direction of the reduced gradient with finite differences.](image-url)
(a) Density and adjoint density residuals

(b) Drag coefficient $C_D$ and 2-norm of the reduced gradient

Figure 5.3: Convergence of the coupled iteration for the initial design.

(a) Without multi-grid acceleration

(b) With multi-grid acceleration

Figure 5.4: Contour plot of the adjoint density.
5.1.4 Optimization

For the optimization we consider now the coupled One-Shot iteration (2.58) accompanied by the additional numerical methods described in chapter 3. We constrain the problem (2.1) - (2.2) by fixing the position of the trailing edge using homogeneous Dirichlet boundary conditions for the solution of (3.7). This eliminates the handling of the problematic singularity of the gradient at this point that is for example visible in figure 5.5. Furthermore we enforce a constant volume of \( V_{\text{init}} = 0.0816925 \) of the airfoil using the penalty multiplier method (ref. section 3.2), i.e.

\[
h(u) := V_{\text{init}} - V(u),
\]

(5.2)

where \( V : U \rightarrow \mathbb{R} \) is the current volume. The initial estimate of the Lagrangian multiplier and the penalty parameter are \( \eta_0 = 0.5 \) and \( \mu = 10^2 \), respectively.

### Choice of the Smoothing Parameter \( \varepsilon \)

As was already mentioned in section 3.1, it is not clear how to choose the optimal smoothing parameter \( \varepsilon \) that occurs in the design preconditioner (3.9). A value that was found to work quiet well in practice in other optimization frameworks is \( \varepsilon = 5.0 \). So we first of all conduct a small parameter study to find a suitable value for our case. Therefore we take \( \varepsilon \in \{1.0, 3.0, 5.0, 7.0\} \) and perform the optimization using each value. The step size parameter is set to \( \tau = 1.0 \), i.e. the Laplace-Beltrami operator should feature a suitable scaling to eliminate the existence of a further parameter. The optimization was started using 10 iterations without a design update to smooth out possible oscillations during the initialization of the flow. Likewise, the design updates are stopped after 1000 iterations since no further decrease in the drag coefficient as well as in the reduced gradient was noticed. In figure 5.6a the RMS residuals of the density during the optimization are plotted. It is clearly visible that for \( \varepsilon = 7.0 \) we essentially only achieve a residual value of \( 10^{-3} \). This means the design updates strongly influence the spectral radius of the associated Jacobian (2.64) such that \( G \) looses its contraction property. In fact, in line with the observations made in subsection 5.1.3 we can not expect that the reduced gradient has reached an acceptable level of convergence yet in this case. This is confirmed by looking at figure 5.6b where \( C_D \) and \( \|N_u\| \) are shown. As soon as the design updates are stopped, they both jump immediately to higher values. A similar behaviour holds for \( \varepsilon = 5.0 \), although not that drastic as we reach a slightly lower density residual of \( 10^{-4} \). Using \( \varepsilon = 3.0 \) we get an even lower residual, such that the norm of the reduced gradient seems stable after the optimization ends. The best performance, is achieved with \( \varepsilon = 1.0 \). The density residual as well as the drag coefficient and the reduced gradient show relatively...
Figure 5.6: Optimization history using different values of $\varepsilon$ for the transonic test case.

The initial NACA0012 airfoil is known to be clearly unsuitable for the present flow conditions. In fact, the pressure contour plot in figure 5.7a shows a strong shock appearing on the upper side of the profile, almost at half of the cord. After optimization with $\varepsilon = 1.0$ we have essentially a shock-free airfoil (see figure 5.7b). The pressure coefficient $C_p$, plotted in figure 5.8 along with the resulting design, now shows a very smooth distribution over the whole surface. If we consider the resulting design we notice a reduced thickness distribution at the front, that leads to a somewhat decelerated flow compared to the initial design which in turn leads to a reduction of the static pressure on the upper side. By comparing the resulting value of the drag coefficient in table 5.2 with the value of the initial design from subsection 5.1.3, we have effectively a reduction of 200 drag counts (from $\Delta C_D = 213$ to $\Delta C_D = 2$). Note, the drag count is a dimensionless
5 Test Cases

(a) Initial Design  
(b) Optimized Design ($\varepsilon = 1.0$)

Figure 5.7: Pressure $p$ for the transonic test case.

unit used by aerospace engineers with $\Delta C_D := 10^{-4} C_D$ and is a more user-friendly measurement as the coefficient of drag is usually less than 1. In theory we could have expected a much smaller drag coefficient as the only physical contribution is the wave-drag. However, due to the artificial viscosity introduced in the JST scheme there is also some spurious numerical drag contained in the calculated drag force. Furthermore, in table 5.2 we see that the volume equals almost the initial value of $V_{\text{init}} = 0.0816925$, more precisely we have an infeasibility of $|h(u^*)| \approx 10^{-7}$.

As far as the performance is concerned, if we compare the number of iterations to reach an residual of $10^{-10}$ for the flow iteration and the One-Shot optimization, we get a retardation factor of

$$ R := \frac{\# \text{One-Shot Iterations}}{\# \text{Flow Iterations}} = \frac{1500}{530} \approx 2.8. \quad (5.3) $$

In combination with the increased run-time for one iteration due to the differentiation, as shown in section 4.2, the overall effort for the optimization is in the order of $8 \cdot 2.8 \approx 22$ flow iterations in this case. But it is expected that the AD overhead factor can greatly be reduced in the near future.

5.2 Supersonic NACA0012

5.2.1 Physical and Numerical Definition

As a second test case we again consider the NACA0012 profile as initial design, but now with supersonic free-stream conditions and zero angle of attack. The detailed conditions are shown in figure 5.3. In this case a strong detached bow shock is visible for the initial airfoil, as shown in 5.10a. A supersonic optimization is a good test case for the shape parametrization and for the geometric constraints. Both play a crucial role in supersonic optimization because, in order to reduce the drag, the airfoil nose is expected to become sharper, causing the bow shock to be almost attached to it and oblique almost everywhere. Shock-wave drag increases with increasing velocity normal to the shock wave. Consequently, oblique shocks feature smaller drag than normal shock waves for the same upstream Mach number.

Due to the stronger shocks compared to the transonic case, the dissipation coefficients in the JST scheme are increased to $\kappa^{(2)} = 0.5$ and $\kappa^{(4)} = 0.02$. Furthermore, the CFL number is decreased to $N_{\text{CFL}} = 2.0$ on the finest mesh.
5.2.2 Optimization

The settings for the optimization are essentially identical to the transonic case, except that in addition to fixing the trailing edge, the leading edge is also held fixed using an additional Dirichlet boundary condition during the solution of (3.7). This prevents surface inversions that are likely to occur in this case. By conducting the same qualitative study to find a suitable smoothing parameter as in subsection 5.1.4 it was found that $\varepsilon = 0.1$ offers the best performance. Again, smaller values were not possible in order to achieve full-step convergence (i.e. with $\tau = 1.0$). Since the overall convergence is faster than in the transonic case, we stopped the optimization after 500 One-Shot iterations.

In figure 5.9a the density residuals using only the flow and adjoint iteration, as well as for the coupled One-Shot iteration are shown. We immediately notice, by using the definition 5.3, that we have a relatively low retardation factor of $R \approx 1.91$ in this case. The drag coefficient and the reduced gradient, plotted in figure 5.9b, indicate a well converged solution. However, we see the same behaviour as in the transonic case, namely that the reduced gradient massively lags behind in terms of convergence compared to $C_D$ and the flow residual. The final values for the optimized and initial design are shown in table 5.4. The drag is drastically reduced by approximately 480 drag counts and the norm of the reduced gradient by two orders of magnitude. Again, the constant volume constraint is maintained well.

Table 5.4: Final values of the drag coefficient, the 2-norm of the reduced gradient and the volume for the supersonic test case.

<table>
<thead>
<tr>
<th></th>
<th>$C_D$</th>
<th>$|N_u|_2$</th>
<th>$V(u^*)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Design</td>
<td>0.0924693968</td>
<td>1.0307050670</td>
<td>0.0816925</td>
</tr>
<tr>
<td>Optimized Design ($\varepsilon = 0.1$)</td>
<td>0.0439877070</td>
<td>0.0773145068</td>
<td>0.081693</td>
</tr>
</tbody>
</table>
Figure 5.9: Optimization history using $\varepsilon = 0.1$ for the supersonic test case.

If we have a look on the resulting final design in figure 5.10 and the corresponding density contours, the shock structure coincides with what was expected at the beginning, namely that the initial bow shock is now oblique everywhere. In fact, designs like this where analytically found to be optimal regarding wave-drag in supersonic flows and are known as Sears-Haack bodies [7]. It should be noted that by using finitely many smooth functions for the parametrization of the design it is hardly possible to get a perfectly sharp nose like in this case with a Free-Node parametrization.
Figure 5.10: Density $\rho$ for the supersonic test case.
Conclusion and Outlook

In this work a framework for the aerodynamic shape optimization based on the Simultaneous Analysis and Design approach was derived and implemented. By using the theory of One-Shot optimization accompanied with Algorithmic Differentiation for the calculation of the gradients it was possible to augment the open-source flow solver Stanford University Unstructured to generate the optimal solution of the optimization problem along with the flow solution. The parametrization of the geometry using the coordinates of the surface mesh nodes allows a maximal degree of freedom, but new types of design preconditioners are necessary in this case to produce feasible designs and to ensure bounded retardation. For that reason we applied the Laplace-Beltrami operator as an approximation of the reduced Hessian. Though in theory it is not quite clear whether the standard reduced Hessian is really the best candidate for a preconditioner, as the similarity to rSQP methods might suggest. Nevertheless, the results for inviscid flows show that with a properly chosen smoothing parameter we can not only achieve qualitative significant designs, but we can also get a solution within a fixed number of iterations that is a small multiple of that for a single flow solution. Although there are more complicated approaches to incorporate additional constraints, the penalty multiplier method proved as a suitable and robust method, at least in the case of geometric equality constraints.

Overall, the foundations for robust and fast shape optimization are laid, but still, a lot of things require further attention and additional work:

- Since most of the flows in nature and technology are of turbulent kind the framework should obviously be extended to handle viscous flows as well. In the flow solver provided by SU2 there are already some turbulence models implemented, such that the additional effort consists only of the proper differentiation of the corresponding code lines.

- In the discussion of the results in chapter 5 we ignored the fact that despite a lower drag count the airfoils also have lost a significant amount of lift. This is not surprising as we did not include an appropriate constraint in the formulation of the optimization problem. How to efficiently handle constraints that depend on the state variables is still an open question, because for example projection methods would require a reasonable converged reduced gradient.

- At the moment the implemented version of the preconditioner works only for 2D problems. But the used approach together with a Free-node parametrization really proves its strength if there are thousands of design variables, that is, surface mesh nodes, like in the case of 3D problems. For unstructured grids, however, a finite difference discretization will not work, so that for example a finite element method must be applied then.

- To make the approach even more viable and easy to apply it is important to know how to choose the smoothing parameter $\kappa$ in the design preconditioner for different flow configurations. Therefore a
Conclusion and Outlook

A thorough understanding of the approximation property of the Laplace-Beltrami operator is necessary. Since analytic investigations are in general fairly complicated due to the specific structure of shape Hessians we suggest a numerical investigation based on the exact calculation of the reduced Hessian using AD. In combination with an inverse design approach it might be possible to determine the dependencies of $\varepsilon$ on the various flow parameters.

- Finally, the high memory requirement is currently not feasible. But by application of advanced AD techniques, like for example checkpointing schemes, one should be able to reduce this factor considerably. Furthermore, the used numerical methods in the flow solver should be analyzed regarding their memory and run-time behavior in the AD setting, such that they can be optimized to gain further improvement in efficiency.

If all of the aspects that were mentioned above are properly tackled, the resulting optimization framework should be a reasonable and efficient alternative to the conventional NAND methods that are used in practice.


