An Efficient CAD-Based Multidisciplinary Optimization Framework for Turbomachinery Design

Thesis approved by
the Department of Computer Science
Technische Universität Kaiserslautern
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Marc Schwalbach

Date of Defense: February 12, 2021
Dean: Prof. Dr. Jens Schmitt
Reviewer: Prof. Dr. Nicolas R. Gauger
Reviewer: Prof. Dr. Tom Verstraete

D 386

Zusammenfassung

Multidisziplinäre Optimierungen (MDOs) sind Optimierungsprobleme, die diverse Diszipline in einer einzigen Optimierung kombinieren, um gleichzeitig mehrere Designbeschränkungen zu erfüllen. Mann betrachtet z.B. aerodynamische und strukturelle Kriterien, um eine Form zu entwickeln, die nicht nur aerodynamische Effizienz besitzt, sondern auch strukturelle Beschränkungen beachtet. Kombiniert mit CAD-basierte Parametrisierungen, erzeugt die Optimierung eine verbesserte, herstellbare Form. Für Anwendungen auf Turbomaschinen, wurden MDOs erfolgreich mit gradientfreie Optimierungsmethoden wie z.B. genetische Algorithmen, Surrogatmodellierung und anderen umgesetzt. Obwohl solche Algorithmen einfach anzuwenden sind, da kein Zugang zum Quellcode benötigt wird, ist die notwendinge Anzahl der Iterationen zur Konvergenz abhängig von der Anzahl der Designparameter. Daraus folgen hohe Kosten und ein beschränkter Designraum. Eine konkurrenzfähige Alternative bieten gradientbasierte Algorithmen mit adjungierte Methoden an. Hierbei ist die Berechnungskomplexität nicht abhängig von der Anzahl der Designparameter, aber eher von der Anzahl der Ausgaben. Solche Methoden wurden mit dem Einsatz von adjungierte Strömungslöser und CAD-basierte Parametrisierungen ausführlich für einzeldisziplinäre Optimierungen der Aerodynamik angewandt. Allerdings kommen gerade erst CAD-basierte MDOs mit dem Einsatz von adjungierten Methoden hervor.

Die vorliegende Dissertation leistet ihren Beitrag zu diesem Forschungsgebiet durch die Entwicklung eines CAD-basierten, adjungierten MDO Frameworks für den Entwurf von Turbomaschinen unter Berücksichtigung der strömungs- und strukturmechanischen Diziplinen. Um dies zu erreichen, wird das CAD-basierte Optimierungsframework *cado* des von Kármán Instituts durch die Entwicklung eines FEM- Strukturlösers erweitert. Der Strukturlöser wird mit der Hilfe des algorithmischen Differenzierungstools *CoDiPack* der TU Kaiserslautern differenziert. Obwohl die Mehrheit des Codes als eine Black-Box differenziert werden kann, benötigen die iterativen Linear- und Eigenlöser eine Sonderbehandlung um die Präzision zu gewährleisten und den Speicherbedarf zu reduzieren. Daraus folgt, dass der Strukturlöser sowohl Spannungs- als auch Vibrationsgradienten mit einem Kosten unabhängig von der Anzahl der Designparameter berechnen kann. Für die vorgelegte Anwendung der Optimierung einer Radialturbine hat der benötigte Gradient ein Berechnungskosten von ungefähr 3,14-mal der des Ausgangscodes und einen ungefähr 2,76-fachen maximalen Speicherbedarf im Vergleich zu dem Ausgangscode.

Der FEM-Löser nutzt objektorientiertes Design aus, um dieselbe Struktur für verschiedene Anwendungen mit minimaler Neu-Differenzierung wieder zu verwenden. Dies wird durch die Betrachtung eines Testfalls mit Verbundwerkstoffen vorgeführt, wobei die Gradienten durch die Erweiterung des Designraums problemlos auch bzgl. Materialparametern berechnet wurden. Zusätzlich wurde der Strukturlöser für eine CAD-basierte Gitterverformung wiederverwendet, die die FEM-Gittergradienten zu den CAD-Parametern durchpropagiert. Hiermit wird der Link zwischen der CAD-Form und des FEM-Gitters geschlossen. Letzlich wurde das MDO-Framework für die Optimierung der aerodynamischen Effizienz einer Radialturbine unter strukturellen Beschränkungen angewandt.

Abstract

Multidisciplinary optimizations (MDOs) encompass optimization problems that combine different disciplines into a single optimization with the aim of converging towards a design that simultaneously fulfills multiple criteria. For example, considering both fluid and structural disciplines to obtain a shape that is not only aerodynamically efficient, but also respects structural constraints. Combined with CAD-based parametrizations, the optimization produces an improved, manufacturable shape. For turbomachinery applications, this method has been successfully applied using gradient-free optimization methods such as genetic algorithms, surrogate modeling, and others. While such algorithms can be easily applied without access to the source code, the number of iterations to converge is dependent on the number of design parameters. This results in high computational costs and limited design spaces. A competitive alternative is offered by gradient-based optimization algorithms combined with adjoint methods, where the computational complexity of the gradient calculation is no longer dependent on the number of design parameters, but rather on the number of outputs. Such methods have been extensively used in single-disciplinary aerodynamic optimizations using adjoint fluid solvers and CAD parametrizations. However, CAD-based MDOs leveraging adjoint methods are just beginning to emerge.

This thesis contributes to this field of research by setting up a CAD-based adjoint MDO framework for turbomachinery design using both fluid and structural disciplines. To achieve this, the von Kármán Institute's existing CAD-based optimization framework *cado* is augmented by the development of a FEM-based structural solver which has been differentiated using the algorithmic differentiation tool *CoDiPack* of TU Kaiserslautern. While most of the code could be differentiated in a black-box fashion, special treatment is required for the iterative linear and eigenvalue solvers to ensure accuracy and reduce memory consumption. As a result, the solver has the capability of computing both stress and vibration gradients at a cost independent on the number of design parameters. For the presented application case of a radial turbine optimization, the full gradient calculation has a computational cost of approximately 3.14 times the cost of a primal run and the peak memory usage of approximately 2.76 times that of a primal run.

The FEM code leverages object-oriented design such that the same base structure can be reused for different purposes with minimal re-differentiation. This is demonstrated by considering a composite material test case where the gradients could be easily calculated with respect to an extended design space that includes material properties. Additionally, the structural solver is reused within a CAD-based mesh deformation, which propagates the structural FEM mesh gradients through to the CAD parameters. This closes the link between the CAD shape and FEM mesh. Finally, the MDO framework is applied by optimizing the aerodynamic efficiency of a radial turbine while respecting structural constraints.

Acknowledgments

I would like to thank

Professors Tom Verstraete and Nicolas Gauger for their guidance, trust, and mentorship

Lasse, Christopher, Mohamed and the whole Turbomachinery Dept. at the von Kármán Institute for their proficient knowledge in the field

CoDiPack guru Max and the SciComp team at TU Kaiserslautern for their AD wisdom

the European Commission for its funding of the IODA project (grant no. 642959)

Queen Mary University of London and Rolls-Royce Deutschland for hosting me during my project secondments

all IODA fellows for their comradeship

my parents and wife for their love, support, and patience

Nomenclature

$\boldsymbol{b}$	load vector
$\boldsymbol{C}$	fourth-order stiffness tensor
$\boldsymbol{f}$	body forces
$\boldsymbol{I}$	identity matrix
$\boldsymbol{J}$	single-entry matrix
$\boldsymbol{N}$	shape functions
$\boldsymbol{P}$	CFD system matrix
$\boldsymbol{R}$	CFD residual
$\boldsymbol{S}$	strain operator
$\boldsymbol{t}$	external surface surface forces / traction forces
$\boldsymbol{u}$	FEM displacements
$\boldsymbol{x}$	unstructured mesh coordinates
$\dot{q}_h$	heat flux
$\hat{\boldsymbol{i}}$	unit vector
$\vec{F}_c$	convective fluxes
$\vec{f}_e$	body forces
$\vec{F}_v$	viscous fluxes
$\vec{n}$	normal vector
$\vec{Q}$	source terms
$\vec{r}$	vector from origin to point
$\vec{v}_a$	absolute velocity
$\vec{v}_e$	entrainment velocity
$\vec{v}_r$	relative velocity
$\vec{W}$	conservative state variables
A	stiffness matrix
C	B-spline curve
c_p	isobaric specific heat capacity
c_v	isochoric specific heat capacity
d	inequality constraint threshold

dS	surface element
E	total energy
e	internal energy
E_Y	Young's modulus
F	example function
H	total enthalpy
h	inequality constraints
J	objective function
K	stiffness matrix
k	thermal conductivity coefficient
M	mass matrix
m	number of FEM nodes
n	number of design parameters
n_h	number of inequality constraints
n_p	number of preceding variables
n_{int}	number of intermediate variables
P	point on mesh
p	pressure
P_λ	vibrational penalty term
Pr	Prandtl number
R	specific gas constant
RPM	rotations per minute
S	B-spline surface
s	auxiliary variable
Sig	sigmoid function
T	Temperature
t	time
u, v	parametric space coordinates
u, v, w	velocities
V	edge vertex, contravariant velocity

w	function variable
y	output variable(s)
α	Runge-Kutta coefficient
$\boldsymbol{\alpha}$	CAD design parameters
$\boldsymbol{\beta}$	material design parameters
$\boldsymbol{\epsilon}$	strain tensor
$\boldsymbol{\lambda}$	variable step sizes
$\boldsymbol{\psi}$	adjoint CFD variables
$\boldsymbol{\sigma}$	Cauchy stress tensor
$\boldsymbol{\xi}$	local FEM coordinates
Δ	step size
$\delta\boldsymbol{u}$	virtual displacement
η	efficiency
Γ	control surface
γ	heat capacity ratio
λ	Lamé's first parameter
λ_i	i-th eigenvalue
λ_{EO_N}	eigenvalue of engine order N
μ	viscosity coefficient / Lamé's second parameter
ν	Poisson's ratio
Ω	control volume
ω	frequency
ω_λ	vibrational constraint weight
ω_σ	stress constraint weight
ω_{EO_N}	frequency of engine order N
ϕ	arithmetic operation
ρ	density
σ	von Mises stress
τ	viscous stress
$\vec{\omega}$	angular velocity

AD	Algorithmic/automatic Differentiation
CAD	Computer-aided Design
CFD	Computational Fluid Dynamics
CSM	Computational Structural Mechanics
FEM	Finite Element Method
FFD	Free Form Deformation
MDO	Multidisciplinary Optimization

Contents

1 Introduction 1

2 Governing Equations 5
- 2.1 Navier-Stokes Equations . . . 5
  - 2.1.1 Ideal Gas Law . . . 6
  - 2.1.2 Rotating Frame of Reference . . . 7
- 2.2 Linear Elasticity . . . 8
- 2.3 Free Dynamic Vibration . . . 9

3 Optimization Framework 11
- 3.1 Optimization Methods . . . 11
  - 3.1.1 Gradient-Free Optimization Methods . . . 11
  - 3.1.2 Gradient-Based Optimization Methods . . . 12
  - 3.1.3 The Adjoint Method . . . 12
- 3.2 CAD-Based MDO Framework . . . 15
- 3.3 CAD-Based Parametrization . . . 16
- 3.4 Mesh Generation . . . 17
  - 3.4.1 Fluid Mesh Generation . . . 17
  - 3.4.2 Solid Mesh Generation . . . 17
  - 3.4.3 Linking Unstructured Mesh and CAD Geometry . . . 20
- 3.5 Computational Fluid Dynamics Solver . . . 24

4 Computational Structural Mechanics Solver 27
- 4.1 Finite Element Method . . . 27
  - 4.1.1 Linear Stress Analysis . . . 28
  - 4.1.2 Free Vibration Analysis . . . 30
- 4.2 Finite Element Code Structure . . . 31
  - 4.2.1 Shape Functions . . . 31
  - 4.2.2 Finite Elements . . . 31
  - 4.2.3 System Assembly . . . 32
  - 4.2.4 Linear Stress Analysis . . . 34
  - 4.2.5 Free Vibration Analysis . . . 35
  - 4.2.6 Mesh Deformation . . . 35
- 4.3 Validation of FEM Solver . . . 37
  - 4.3.1 Linear Stress Analysis Validation . . . 38
  - 4.3.2 Free Vibration Analysis Validation . . . 39

5 Efficient and Maintainable Code Differentiation 41
- 5.1 Algorithmic Differentiation . . . 41
  - 5.1.1 Tangent (Forward) Mode . . . 42
  - 5.1.2 Adjoint (Reverse) Mode . . . 44
  - 5.1.3 Source Code Transformation Methods . . . 46
  - 5.1.4 Operator Overloading Methods . . . 47
  - 5.1.5 Finite Differences & Complex Step Method . . . 50
- 5.2 Differentiating a Structural Sress Solver with AD . . . 50
  - 5.2.1 Gradient Comparison and Computational Cost . . . 52
- 5.3 Differentiating a Structural Vibration Solver with AD . . . 53

5.3.1 Tangent Model 54
5.3.2 Adjoint Model 55
5.3.3 Gradient Comparison and Computational Cost 58
5.4 Reducing Assembly Memory Consumption Using Checkpointing 59
5.5 Differentiating from CAD Kernel to Mesh with AD 64
5.5.1 Solid Mesh Sensitivities 64
5.5.2 Fluid Mesh Sensitivities 67
5.6 Reusability of Differentiated Framework - Design Space Extension 69
5.7 Computational Costs of MDO Framework 72

6 Validation and Application of MDO Framework 73

6.1 Gradient Comparison from CAD to Stress and Vibration Outputs 73
6.2 Structural Shape Optimization of a Radial Turbine 73
6.2.1 Numerical Setup 75
6.2.2 Optimization Results 75
6.3 Aerodynamic Shape Optimization of a Radial Turbine Under Stress Constraints 76
6.3.1 Numerical Setup 76
6.3.2 Optimization Results 78
6.4 Aerodynamic Shape Optimization of a Radial Turbine Under Stress and Vibration Constraints 80
6.4.1 Numerical Setup 81
6.4.2 Optimization Results 82

7 Conclusion and Outlook 83

References 85

Curriculum Vitae 93

1 Introduction

For decades, engineering intuition, experience, and experimentation were the de facto driving forces in the design cycles that improved machine efficiencies. However, these developments involved the time-consuming and expensive cycle of prototype development, experimentation, and assessment, which involved numerous iterations and errors. The dawn of numerical simulation and optimization methods perhaps began around 1922 with Richardson's proposals of weather predictions by numerical methods [53], when computations still had to be performed manuallly by human computers. It wasn't until the late 1950s that computational resources were adequate enough, resulting in a rapid rise in numerical modeling developments, especially in the field of computational fluid dynamics (CFD) [31]. Since its inception in the 1920s, nearly 100 years later, significant improvements in computational power and numerical methods has enabled us to perform more high-fidelity design work in the numerical laboratory, rather than the physical one, at a fraction of the cost.

Especially in the field of aerodynamics, the developments of CFD analysis and optimization methods have proven indispensable. Applications range from the exterior shapes of aircrafts, automobiles, wind turbines, watercrafts, and rockets to the shape of internally used components such as radial turbines, cooling channels, and even blood vessels. Some of the benefiting industries include aviation, space, defense, automotive, energy, and medical devices. Fluids are everywhere. For the majority of these applications, typically the development of an *optimal* shape that can deliver the highest efficiency is of utmost importance. To achieve this goal, numerous optimization techniques [67] have been developed over the years. Notably, the combination of multiple disciplines, such as fluid, solid, and thermal, has given rise to so-called *multi-disciplinary optimizations* (MDOs) [41] that strive to achieve, e.g., a maximum aerodynamic performance, while repecting structural and thermal constraints.

In the field of turbomachinery applications, the ubiquity of MDO methods is continuously growing. Gradient-free MDO frameworks, which include methods such as genetic algorithms [69, 74], response surfaces [15], and differential evolution [32, 46] have been successfully applied, often in combination with surrogate models [62] to accelerate the optimization. However, gradient-free methods have their limits, which will be discussed in chapter 3, and gradient-based methods using *adjoints* offer a competitive alternative for computationally efficient MDO frameworks as the adjoint method enables gradient calculations at a cost dependent on the number of outputs, rather than the number of inputs. This has a significant impact on typical optimization problems where a single or few objectives are dependent on a large number of design parameters.

Several advancements have been made in the field of adjoint-based MDOs, especially in the field of external aerodynamics, extensive work has been done on coupled aerostructural optimizations [1, 39, 37, 43]. From these contributions, it is evident that the self-adjoint property of the linear elastic equations can be used to compute the structural adjoint vector without the need of a differentiated solver. However, the partial derivatives of the structural solver's residual with respect to the design parameters are still required and need to be derived analytically or are approximated by finite differences [38, 40]. This differentiation procedure can be considerably improved by applying advanced source-code augmentation techniques such as *algorithmic differentiation* (AD) directly on the numerical solver's source code.

Concerning adjoint optimizations within the field of turbomachinery, a few recent developments include adjoint formulations for non-ideal compressible fluid dynamics [79] and reduced order models for unsteady problems [55], both utilizing a *free-form deformation* (FFD) parametrization for the optimization. The choice of an optimal parametrization is a continuous discussion and recent work [3] compares both FFD and CAD-based parametrizations. While FFD is highly flexible for shape deformations, CAD-based parametrizations offer more control over geometric, i.e., manufacturing constraints and is a field of ongoing research. Both open-source [48] and in-house [57] CAD kernels have been differentiated using algorithmic differentiation to perform CAD-based adjoint optimizations, while some parametrizations involve the use of B-Spline surfaces to maintain a link to the CAD model without having to differentiate a fully-fledged CAD kernel [75]. Other groups handle the integration of CAD-based parametrizations into the adjoint optimization chain via a design velocity approach, i.e., finite-differencing black-box CAD packages [71], which allows the use of commercial closed-source CAD packages. However, the majority of these developments remain focused solely on aerodynamic objectives and constraints. To compensate for the lack of an adjoint structural solver, some groups have included structural constraints without the use of gradients by generating a Kriging metamodel based on hundreds of Finite-Element Method (FEM) simulations [4]. Adjoint-based MDO frameworks using a CAD-based parametrization and adjoint structural analyses, are only now beginning to emerge [76, 60].

This thesis seeks to contribute to this continuously evolving community with the development of a CAD-based adjoint multidisciplinary optimization framework, using state of the art algorithmic differentiation techniques and a 3D adjoint structural solver. The main contributions of this work can be broken down into the following three components:

1. the development and algorithmic differentiation of an object-oriented 3D structural solver to compute stress and vibration gradients at a low computational cost,

2. linking the structural gradients with the CAD kernel via an unstructured mesh deformation to acquire gradients with respect to the CAD parameters,

3. and combining these modules with an adjoint CFD solver to set up CAD-based MDOs of turbomachinery components.

For the inclusion of the structural discipline in a gradient-based MDO, gradients of structural quantities of interest, such as the maximum von Mises stress or eigenfrequencies, are required. While it is possible to set up an adjoint structural solver using an analytical approach, this thesis proposes a more algorithmic approach to leverage object-oriented design. This leads to the development of an adjoint computational structural mechanics (CSM) solver based on the FEM method, which was fully differentiated using operator-overloading AD. With this solver, stress and vibration gradients can be computed at a low computational cost, making a gradient-based MDO computationally feasible and eliminating the need of approximations by finite differences. The resulting cost is one additional linear system solve for the maximum von Mises stress gradients and one additional outer product per eigenvalue for eigenvalue gradients. Additionally, the solver follows an object-oriented design, such that the differentiated base classes can be extended to implement new features that are automatically differentiated. For example, an unstructured mesh deformation based on the linear elastic analogy was implemented as a child class of the differentiated base solver. As a result, the gradients could directly be calculated without the need of further code differentiation. This software design aims to alleviate the problems associated with the code extensibility and maintainability of AD-differentiated solvers, speeding up code iterations and paving the way for more complex structural gradient calculations.

As CAD is the de facto standard for digital geometries in the engineering world, the optimization framework focuses on a CAD-based parametrization. This not only eliminates the approximation errors of fitting a CAD geometry to an optimized mesh as a post-processing step, but also enables the inclusion of geometric constraints to ensure manufacturability of the optimized shape. Finally, the examination of the optimized CAD parameters gives more insight to engineers, as values of, e.g., a thickness or angle distribution along a turbine blade are more comprehensible than the change of coordinates of an arbitrary FEM node. Within this work, a CAD kernel was differentiated using forward AD and a CAD-based unstructured mesh deformation was implemented to link the CAD geometry with the computational mesh, enabling gradient calculations with respect to CAD parameters. The unstructured mesh deformation profits from the differentiated structural solver base class to efficiently propagate gradients through the CAD kernel.

Similar to an integration test, the differentiated modules are ultimately coupled together to test their applicability as a whole. An optimization framework where the entire chain from CAD parameters to objective function has been differentiated with the aid of AD is set up. This demonstrates the relevance of the efficiently calculated gradients for gradient-based optimizations. The chain is first tested with a CAD-based structural optimization of a radial turbine. Subsequently, this is combined with an adjoint CFD solver to set up a CAD-based MDO framework involving objectives and constraints from multiple disciplines - aerodynamics, stress, and vibration.

This work is structured as follows. In chapter 2, this thesis will introduce the governing equations relevant for the multidisciplinary optimization (MDO), i.e., the fluid and solid disciplines. Afterwards, the optimization framework as a whole is described in chapter 3, including a discussion of optimization methods, the adjoint method for gradient calculations, CAD-based parametrization, mesh generation for the fluid and structural solvers, as well as the adjoint computational fluid dynamics (CFD) solver used in this work. Chapter 4 will then detail the design of the CSM solver that was developed within the scope of this thesis. The following chapter, chapter 5, will then go into further detail of the application of algorithmic differentiation (AD) on the developed CSM solver and CAD kernel to enable gradient calculations using the adjoint method in an efficient, algorithmic manner. This chapter also discusses how the differentiated base solver facilitates gradient calculations for child-class contributions. Finally, chapter 6 showcases applications of the developed CAD-based MDO framework with optimization test cases before concluding in chapter 7.

2 Governing Equations

This section introduces the governing fluid and structural equations considered in this thesis. The Navier-Stokes equations define the governing fluid equations, which are formulated in a rotating frame of reference for turbomachinery simulations. For the structural equations, this mainly includes the linear stress analysis and vibration analysis, which are described by linear elasticity and free dynamic vibration.

2.1 Navier-Stokes Equations

The Navier-Stokes [7] equations describe the physical conservation laws of a fluid given by the conservation of mass, momentum, and energy within a control volume Ω. These can be formulated as the rate of change of conservative state variables over time, their spatial flux, and source terms:

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{W}d\Omega + \oint_{\partial\Omega}\left(\vec{F}_c - \vec{F}_v\right)dS = \int_{\Omega}\vec{Q}d\Omega \tag{1}$$

The vector $\vec{W}$ contains the conservative state variables

$$\vec{W} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{pmatrix}, \tag{2}$$

where ρ denotes the density, u, v, w the velocities in x, y, and z directions, respectively, and E the total energy. The spatial fluxes represent the rate of change through the boundary $\partial\Omega$ of the control volume and are made up of two components $\vec{F}_c$ and $\vec{F}_v$. The first represents the convective fluxes

$$\vec{F}_c = \begin{pmatrix} \rho V \\ \rho u V + n_x p \\ \rho v V + n_y p \\ \rho w V + n_z p \\ \rho H V \end{pmatrix}, \tag{3}$$

whereby H defines the total enthalpy, p the pressure, and V the contravariant velocity

$$V = \vec{v}\cdot\vec{n} = n_x u + n_y v + n_z w. \tag{4}$$

The second term defines the viscous fluxes

$$\vec{F}_v = \begin{pmatrix} 0 \\ n_x\tau_{xx} + n_y\tau_{xy} + n_z\tau_{xz} \\ n_x\tau_{yx} + n_y\tau_{yy} + n_z\tau_{yz} \\ n_x\tau_{zx} + n_y\tau_{zy} + n_z\tau_{zz} \\ n_x\Theta_x + n_y\Theta_y + n_z\Theta_z \end{pmatrix}, \tag{5}$$

where the influence of heat conduction and viscous stress is categorized into the terms

$$\Theta_x = u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + k\frac{\partial T}{\partial x} \tag{6}$$

$$\Theta_y = u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + k\frac{\partial T}{\partial y} \tag{7}$$

$$\Theta_z = u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + k\frac{\partial T}{\partial z}. \tag{8}$$

Here T defines the temperature, k the conductivity coefficient, and τ the viscous stress tensor. The source term $\vec{Q}$ represents the effects of body forces and volumetric heating and is given as

$$\vec{Q} = \begin{pmatrix} 0 \\ \rho f_{e,x} \\ \rho f_{e,y} \\ \rho f_{e,z} \\ \rho \vec{f_e} \cdot \vec{v} + \dot{q}_h \end{pmatrix}. \tag{9}$$

The body forces are defined as $\vec{f_e}$, and $\dot{q}_h$ represents the heat flux.

2.1.1 Ideal Gas Law

The Navier-Stokes equations (1) comprise a system of five equations with seven unknowns: ρ, u, v, w, E, p, T. Two additional state equations are introduced to close the system of equations. The *ideal gas* law gives the relationship between pressure, density, and temperature as

$$p = \rho R T \tag{10}$$

with the specific gas constant defined by R. Additionally, the following relationship between the internal energy e and temperature T holds:

$$e = c_v T, \tag{11}$$

where c_v is the isochoric specific heat capacity and the internal energy is defined as

$$e = E - \frac{|\vec{v}|^2}{2}. \tag{12}$$

The temperature T can thus be directly computed via (11) and (12) using the conservative variables. To determine the pressure p from the conservative variables, the specific heat capacity definitions

$$R = c_p - c_v, \quad \gamma = \frac{c_p}{c_v} \tag{13}$$

can be used. From (11) and (10), one derives

$$
\begin{aligned}
p &= \rho R \frac{e}{c_v} \\
p &= \rho \frac{R}{c_v} e \\
p &= \rho \left(\gamma - 1\right) e \\
p &= \rho \left(\gamma - 1\right) \left(E - \frac{|\vec{v}|^2}{2}\right).
\end{aligned}
$$

Alternatively, the temperature T can be directly plugged into (10) if it has already been computed via (11).

Finally, the coefficients μ and k have to be computed, which represent the dynamic viscosity and thermal conductivity, respectively. The coefficients are computed using Sutherland's law [65]

$$\mu = \frac{1.45 T^{\frac{3}{2}}}{T + 110} \cdot 10^{-6} \tag{14}$$

and for thermal conductivity

$$k = c_p \frac{\mu}{Pr}, \tag{15}$$

where Pr represents the dimensionless Prandtl number. The Prandtl number represents the ratio between momentum and thermal diffusivity and is generally $Pr = 0.72$ for air, closing our system of equations for the fluid discipline.

2.1.2 Rotating Frame of Reference

For turbomachinery applications, which typically undergo a rotating motion, it is necessary to formulate the Navier-Stokes equations (1) for a rotating frame of reference. Assuming the considered system rotates with a constant angular velocity of $\vec{\omega}$ around the x-axis, such that $\vec{\omega} = (\omega\ 0\ 0)$, the absolute velocity is given as

$$\vec{v}_a = \vec{v}_r + \vec{v}_e = \vec{v}_r + \vec{\omega} \times \vec{r}. \tag{16}$$

Here $\vec{v}_r$ defines the relative velocity, $\vec{v}_e$ the entrainment velocity and $\vec{r}$ the vector from the coordinate system origin to the considered point. Consequently, additional forces, namely the *Coriolis* and *centrifugal* forces are introduced into the Navier-Stokes equations (1). The Coriolis force is given as

$$\vec{f}_{Cor} = -2 \left(\vec{\omega} \times \vec{v}_r\right) \tag{17}$$

and the centrifugal force as

$$\vec{f}_{cen} = -\vec{\omega} \times \left(\vec{\omega} \times \vec{r}\right) = \omega^2 \vec{r}_n, \tag{18}$$

where $\vec{r}_n$ defines the perpendicular vector from the axis of rotation to the considered point. Both forces affect the momentum equations, while only the centrifugal forces influence the energy equation. As a result, the Navier-Stokes equations generally take the same form as given in (1), but with some modified terms. Thus, the total energy is now given as

$$E = e + \frac{|\vec{v}_r|^2}{2} - \frac{|\vec{v}_e|^2}{2} = e + \frac{u^2 + v^2 + w^2}{2} - \frac{\omega^2 |\vec{r}_n|^2}{2}. \tag{19}$$

The convective fluxes change to

$$\vec{F}_c = \begin{pmatrix} \rho V \\ \rho u V + n_x p \\ \rho v V + n_y p \\ \rho w V + n_z p \\ \rho I V \end{pmatrix}, \tag{20}$$

where the variable I represents the *rothalpy*

$$I = h + \frac{|\vec{v}_r|^2}{2} - \frac{|\vec{v}_e|^2}{2} = H - \frac{\omega^2 |\vec{r}_n|^2}{2}. \tag{21}$$

The contravariant velocity V is defined as

$$V = n_x u + n_y v + n_z w, \tag{22}$$

and u, v, and w define the cartesian velocity vectors in the rotating frame, such that

$$\vec{v}_r = \begin{pmatrix} u \\ v \\ w \end{pmatrix}. \tag{23}$$

The Coriolis (17) and centrifugal (18) forces augment the source terms to

$$\vec{Q} = \begin{pmatrix} 0 \\ \rho f_{e,x} \\ \rho \omega \left(y\omega + 2w \right) + \rho f_{e,y} \\ \rho \omega \left(z\omega - 2v \right) + \rho f_{e,z} \\ \rho \vec{f}_e \cdot \vec{v}_r + \dot{q}_h \end{pmatrix}. \tag{24}$$

Finally, the pressure is now computed by

$$p = (\gamma - 1)\, \rho \left[E - \frac{u^2 + v^2 + w^2 - \omega^2 |\vec{r}_n|^2}{2} \right], \tag{25}$$

closing the Navier-Stokes equations for a rotating frame of reference.

2.2 Linear Elasticity

For the structural constraints of a turbomachinery optimization, the stress and vibration analyses play a vital role. To derive the system of equations for the static linear stress analysis, one starts with the equilibrium equation

$$\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{f} = \boldsymbol{0}, \tag{26}$$

the strain-displacement equations

$$\boldsymbol{\epsilon} = \frac{1}{2} \left[\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T \right], \tag{27}$$

and the constitutive equations based on Hooke's law

$$\boldsymbol{\sigma} = \boldsymbol{C} : \boldsymbol{\epsilon}, \tag{28}$$

where $\boldsymbol{C} : \boldsymbol{\epsilon}$ defines the inner product $\boldsymbol{C}_{ij}\boldsymbol{\epsilon}_{ij}$ [63], and $\boldsymbol{C}$ defines the stiffness tensor. $\boldsymbol{f} := \begin{pmatrix} f_x & f_y & f_z \end{pmatrix}^T$ denotes the body forces and $\boldsymbol{\sigma}$ the stress tensor

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{pmatrix}. \tag{29}$$

The *principle of virtual work* is commonly used to derive the discrete, algebraic form of the linear elastic equations solved by the FEM method. Consider a virtual displacement field $\delta\boldsymbol{u}$, which vanishes on the boundary $\delta\boldsymbol{u} = \boldsymbol{0}|_{S_u}$. As a result of the virtual displacement, a virtual strain follows through (27):

$$\delta\boldsymbol{\epsilon} = \frac{1}{2}\left[\nabla\delta\boldsymbol{u} + (\nabla\delta\boldsymbol{u})^T\right] = \boldsymbol{S}\delta\boldsymbol{u}, \tag{30}$$

where $\boldsymbol{S}$ is defined as a linear differential operator. The *internal virtual work* is defined as

$$\int_\Omega \delta\boldsymbol{\epsilon}^T\boldsymbol{\sigma}d\Omega \tag{31}$$

and *external virtual work* as

$$\int_\Omega \delta\boldsymbol{u}^T\boldsymbol{f}d\Omega + \int_\Gamma \delta\boldsymbol{u}^T\boldsymbol{t}d\Gamma, \tag{32}$$

where $\boldsymbol{f}$ are external body forces and $\boldsymbol{t}$ the external surface or traction forces of the form

$$\boldsymbol{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = \begin{pmatrix} n_x\sigma_x + n_y\tau_{xy} + n_z\tau_{xz} \\ n_x\tau_{xy} + n_y\sigma_y + n_z\tau_{yz} \\ n_x\tau_{xz} + n_y\tau_{yz} + n_z\sigma_z \end{pmatrix}. \tag{33}$$

Equating the internal virtual work with the external virtual work leads to

$$\int_\Omega \delta\boldsymbol{\epsilon}^T\boldsymbol{\sigma}d\Omega = \int_\Omega \delta\boldsymbol{u}^T\boldsymbol{f}d\Omega + \int_\Gamma \delta\boldsymbol{u}^T\boldsymbol{t}d\Gamma, \tag{34}$$

which is essentially the weak form of the equilibrium equations [82], considering a body with the domain Ω and boundary Γ. Applying the Galerkin method on (34) leads to the finite element formulation, which is described in more detail in section 4.1.1.

2.3 Free Dynamic Vibration

When taking vibrational effects into account, mainly two additional forces are considered [82]. The first is the *inertial force* $-\rho\ddot{\boldsymbol{u}}$, with the density ρ and acceleration $\ddot{\boldsymbol{u}}$. The second force is the *frictional resistance*, which, in a simple linear case is $-\boldsymbol{\mu}\dot{\boldsymbol{u}}$, with a given viscosity $\boldsymbol{\mu}$. In a static problem, this results in the body force $\boldsymbol{f}$ replaced by

$$\boldsymbol{f} = \bar{\boldsymbol{f}} - \rho\ddot{\boldsymbol{u}} - \boldsymbol{\mu}\dot{\boldsymbol{u}}, \tag{35}$$

whereby the external body forces are now defined in $\bar{\boldsymbol{f}}$. In the case of a free vibration problem, the *damping* term is dropped, resulting in

$$\boldsymbol{f} = \bar{\boldsymbol{f}} - \rho \ddot{\boldsymbol{u}}. \tag{36}$$

This is then plugged into the weak form of the equilibrium equations (34). The finite element formulation for the free vibration analysis is discussed in section 4.1.2.

3 Optimization Framework

In this section, the building blocks for a CAD-based adjoint multidisciplinary optimization framework are introduced. First, gradient-free and gradient-based optimization methods are discussed, followed by an introduction of the adjoint method for computing gradients efficiently. The CAD-based MDO framework is then layed out, which includes the CAD-based parametrization, mesh generation for the fluid and solid domains, mesh deformation for the solid domain, and finally the adjoint CFD solver. The adjoint structural solver is thoroughly discussed in the follower chapter 4.

3.1 Optimization Methods

A general optimization problem is formulated by first defining an objective $J(\boldsymbol{\alpha})$ that is dependent on the design variables $\boldsymbol{\alpha} \in \mathbb{R}^n$, where n is the number of design variables. The goal is to minimize the objective J by finding the optimal design

$$\begin{gathered} \boldsymbol{\alpha}^* = \arg\min_{\boldsymbol{\alpha}} J(\boldsymbol{\alpha}) \\ \text{such that } h_i(\boldsymbol{\alpha}^*) \leq d_i(\boldsymbol{\alpha}^*),\ i = 0, ..., n_h - 1 \end{gathered} \tag{37}$$

where $h_i(\boldsymbol{\alpha}^*) \leq d_i(\boldsymbol{\alpha}^*)$ are the inequality constraints that need to be satisfied by the optimum and n_h is the number of constraints. The methods for solving this optimization problem are divided into two main branches: gradient-free and gradient-based methods, which are discussed in the next two sections.

3.1.1 Gradient-Free Optimization Methods

Gradient-free optimization methods are typically non-deterministic optimization methods, which do not require gradient information. These methods are based on heuristics or inspired by natural processes to find potentially multiple local optima. Popular methods include evolutionary algorithms, such as *genetic algorithms* [26] and *differential evolution* [64], *simulated annealing* [33], and *partical swarm optimization* [16].

Such methods are rather straightforward to implement and non-invasive, meaning that the source code of, e.g., the CFD solver is not required and can be regarded as a *black-box*. This is advantageous from the user's perspective, as the choice of numerical solvers, CAD, and mesh generation tools is not limited by the source code requirement, allowing users to opt for commercial black-box solutions. Gradients are not required, and thus the optimization problem does not have to be differentiable and could even be noisy. This is practical in cases where gradients are difficult or not possible to obtain. However, the number of iterations, and thus evaluations of J, required to reach an optimal design generally increases with the number of design variables. This *curse of dimensionality* [52] can easily lead to significantly high, or even prohibitive, computational costs, depending on the size of the design space [10].

3.1.2 Gradient-Based Optimization Methods

Gradient-based optimization methods use gradient information to converge towards a local optimum with less iterations. The simplest canon example of a gradient-based optimization method is the *steepest descent* method

$$\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i - \Delta \frac{\partial J}{\partial \boldsymbol{\alpha}}, \tag{38}$$

where Δ represents the step size. Recall that the gradient provides the information of how much the objective J changes with respect to changes in the design variable $\boldsymbol{\alpha}$. The basic idea behind steepest descent is then to update the variable $\boldsymbol{\alpha}$ in the direction of the gradient such that the value of J is reduced.

The downside of gradient-based methods is that the gradient $\frac{\partial J}{\partial \boldsymbol{\alpha}} \in \mathbb{R}^n$ is required. Not only does this require the objective to be differentiable, but computing the gradient can be computationally expensive. Using a non-invasive approach, such as finite-difference (FD) perturbations, leads to a computational cost proportional to the number of design parameters n. For first-order FD approximations, $n + 1$ evaluations of J would be required and $2n$ evaluations for second-order FD. Thus, the number of design parameters, and in effect the richness of the design space, is also restricted due to the linearly increasing computational costs associated with the gradient calculation.

3.1.3 The Adjoint Method

The adjoint method, which was first applied in aerodynamic optimizations by Pironneau [51] and Jameson [28, 30] using continuous adjoints, offers a remedy to the high computational costs associated with gradient calculations. In short, the cost of computing gradients using the adjoint method is proportional to the dimension of the output J, rather than the input $\boldsymbol{\alpha}$. This offers a considerable cost advantage for optimization problems where the dimension of the objective function is typically far less than the number of design parameters. Additionally, a much richer design space can be used, since the cost of computing the gradient is independent of the size of the design space. As opposed to finite differences, which is a gradient approximation method that introduces truncation errors, the adjoint method is numerically stable.

To derive the *adjoint model* that this method requires, there are two general possibilities. One can analytically derive the adjoint equations from the *primal* equations considered, e.g., the adjoint model of the Navier-Stokes equations [29], and then discretize the adjoint equations to solve them numerically. This is commonly referred to as the *continuous adjoint* approach. The other approach inverts the order - first discretize your equations, e.g. Navier-Stokes, and then derive the adjoint model of the discretized equations. This is called the *discrete adjoint* approach.

The continuous adjoint approach offers more physical insight into the adjoint model and can be suitably discretized to be solved efficiently. However, deriving the continuous adjoint equations from the primal equations is a meticulous and arduous task. Additionally, the task of deriving the continuous adjoint equations would need to be repeated every time different primal equations are considered. The discrete adjoint approach, on the other hand, is dependent on the discretization of the primal equations and the quality of the numerical code. Using a technique called *algorithmic differentiation* (AD), an adjoint model of the discretized equations can be derived by differentiating the numerical code directly. The advantage here is that differentiation process is semi-automated once set up, meaning that new additions to the primal code automatically lead to a differentiated code for the gradient calculation. Moreover, the discrete adjoint method does not necessarily require anlaytical differentiability of the underlying equations, as long as the discretized equations are algorithmically differentiable [44, 9].

In this thesis, the discrete adjoint approach is considered, using an algorithmic differentiation tool to derive the adjoint code. This is discussed in more detail in chapter 5.

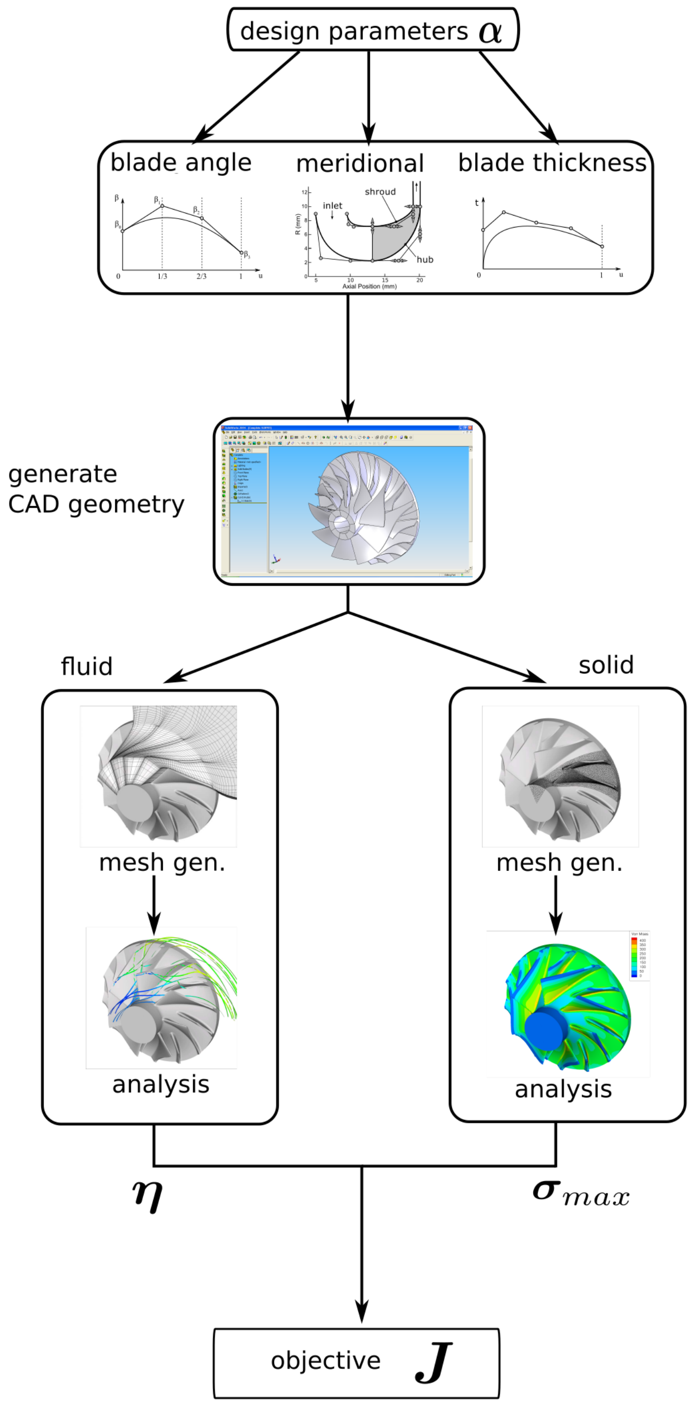


Figure 1: Flowchart of the CAD-based MDO framework

3.2 CAD-Based MDO Framework

A multidisciplinary optimization (MDO) is an optimization problem that takes more than one discipline into consideration. For example, a typical setting for shape optimizations in aerospace or automotive applications consists of an aerodynamic objective and a structural constraint. To perform an efficient MDO with adjoints, adjoint models of both disciplines are required, as well as an appropriate parametrization. This thesis now introduces the components that constitute the MDO framework, including the parametrization (sec. 3.3), mesh generation (sec. 3.4), and the adjoint solvers for the fluid (sec. 3.5) and solid (chap. 4) disciplines.

The typical goal of a turbomachinery optimization would be to maximize the efficiency η, while respecting structural constraints, such as keeping the maximum von Mises stress σ_{max} below a defined threshold to avoid structural failure. Consider the workflow as shown in figure 1, which begins with CAD parameters $\boldsymbol{\alpha}$, which serve as the *design parameters* for the optimization. The CAD parameters $\boldsymbol{\alpha}$ serve as the inputs for the CAD kernel, which generates the CAD geometry. At this point, the generated geometry can be exported into the STEP file format, which is an ISO standard for CAD models. In section 3.3, this parametrization is discussed in more detail.

Next, the fluid structured mesh and solid unstructured mesh are generated based on the internal representation of the CAD geometry. During the mesh generation process, a conformity between the outer mesh faces and the CAD surfaces is ensured, minimizing errors between the computational meshes and the geometry. An in-house CFD solver [45] and CSM solver [58] are then used to perform the CFD and CSM analyses, respectively. The resulting quantities of interest, such as the efficiency η and maximum von Mises stress σ_{max}, can be combined into a single objective function $J(\eta, \sigma_{max})$.

The optimization problem is now formulated as solving for the optimal CAD parameters $\boldsymbol{\alpha}^*$ that minimize the objective function J (37). Using a gradient-based optimization method, this requires the gradients of the objective function with respect to the design parameters $\frac{dJ}{d\boldsymbol{\alpha}}$, which involves components from the entire MDO chain of figure 1. While the adjoint CFD and CSM solvers deliver sensitivities with respect to the computational meshes, i.e. $\frac{\partial \eta}{\partial \boldsymbol{x}_{fluid}}$, $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}}$, these still have to be multiplied with the sensitivities of the mesh with respect to the CAD parameters via the chain rule:

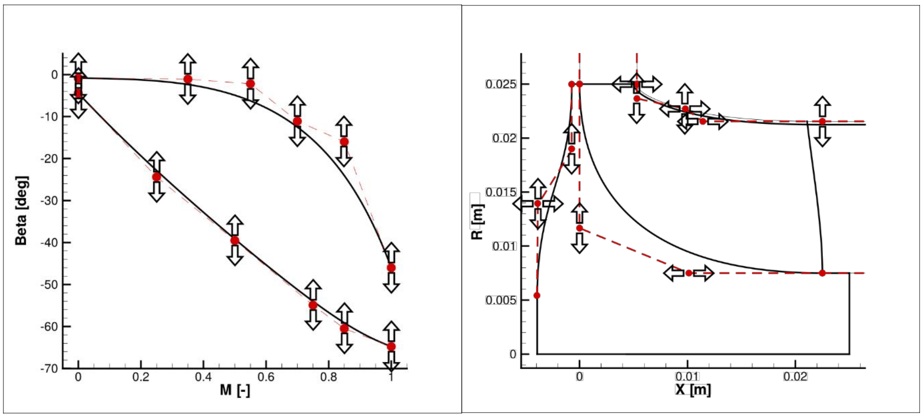
$$\begin{aligned} \frac{d\eta}{d\boldsymbol{\alpha}} &= \frac{\partial \eta}{\partial \boldsymbol{x}_{fluid}} \frac{\partial \boldsymbol{x}_{fluid}}{\partial \boldsymbol{\alpha}} \\ \frac{d\sigma_{max}}{d\boldsymbol{\alpha}} &= \frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}} \frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{\alpha}} \end{aligned} \tag{39}$$

The primal and differentiated models used to compute the gradients required by (39) are introduced in the following sections.

3.3 CAD-Based Parametrization

For the majority of engineering applications, and turbomachinery design and production in particular, CAD models are ubiquitous. Not only are they used to design 3D models, but also to exchange designs between departments, and are ultimately passed on to the production dep

arment for manufacturing. Maintaining a link to the master CAD geometry during an optimization has several advantages. For instance, CAD-free shape optimizations would typically output a point cloud or parametrized B-Spline as the resulting shape. Afterwards, a CAD model would still need to be fitted to the optimized shape, resulting in fitting errors between the optimized shape and the CAD geometry. Working with the master CAD geometry directly during an optimization eliminates this fitting error. Additionally, manufacturing constraints are more easily applied to a CAD geometry to, e.g., maintain curvature or straight edges where needed, while respecting a minimal thickness. Last but not least, a CAD parametrization is more intelligible for engineers and provides more insight into how the optimizer is manipulating the shape to minimize the objective.



(a) Blade angle parametrization

(b) Meridional shape parametrization

Figure 2: CAD-based parametrizations

CADO [73], the von Kármán Institute's (VKI) CAD-based optimization framework, provides the benefits noted above and is extended to support multidisciplinary optimizations within this work. Internally, the geometries are described using BSpline and NURBS to define the curves and surfaces, which are generated based on the user's inputs, i.e., the CAD parameters $\boldsymbol{\alpha}$. For a radial turbine, for example, typically the shape of the blade can be determined by user-defined blade angle and thickness distributions. This can also be used to create the shape of the wheel hub and shroud, while other components such as the back plate can be described using BSplines. An example of a CAD parametrization for a radial turbine can be seen in figure 2. Here the red dots represent the control points which are essentially the design parameters $\boldsymbol{\alpha}$ for the optimization. The arrows on these control points represent the degrees of freedom, which can be used to define geometrical constraints. During an optimization, these control points are moved to change the shape of the geometry. For different applications, e.g., for axial compressors, radial turbines or radial compressors, corresponding application-specific algorithms have been developed that generate CAD geometries for each geometry, using the same underlying CAD kernel of CADO.

3.4 Mesh Generation

3.4.1 Fluid Mesh Generation

For the CFD analysis, a 3D multi-block structured mesh is utilized, which is generated using a CAD-based approach within CADO. This is achieved by first creating a topology layer based on the CAD geometry (figure 3). In contrast to the geometry, which defines the *shape*, the topology describes the *connectivity* among the vertices, edges, faces, and surfaces of the geometry. For example, the toplogy information of a face would include the bordering edges and the order in which they form a closed loop. Using the CAD geometry and topology as a starting point ensures that the resulting mesh follows the geometry's shape with high accuracy (figure 4). The assembly and types of grids used, i.e. C-grids, O-grids, and H-grids, depends on the application, e.g. a radial turbine or axial fan blade CAD geometry.

The size and quality of the mesh is set by the user by specifying, e.g., the number of mesh points and the cell size along given boundaries. Based on these inputs, an initial mesh is generated, which is then refined using an elliptic smoother technique [68], whereby the user can specify the number of smoothing iterations performed. The node count and topology of the fluid mesh remains constant if the mesh quality settings are unchanged, allowing a direct link between the CAD geometry $\boldsymbol{\alpha}$ and the fluid mesh $\boldsymbol{x}_{fluid}$ through the mesh generation.

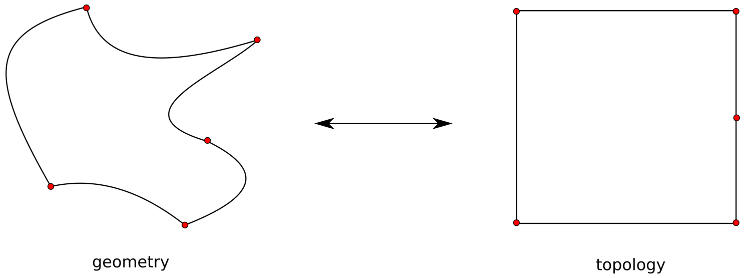


Figure 3: Illustrative example of connection between shape geometry (left) and topology layer (right). The geometry defines the shape, while the topology defines the connectivity.

3.4.2 Solid Mesh Generation

The CSM analysis uses an unstructured mesh which is generated based on the famous TetGen [61] mesh generation algorithm. Analogously to the fluid mesh generation, the CAD geometry and topology information is used as the starting point of the unstructured mesh generation. This information is passed on to

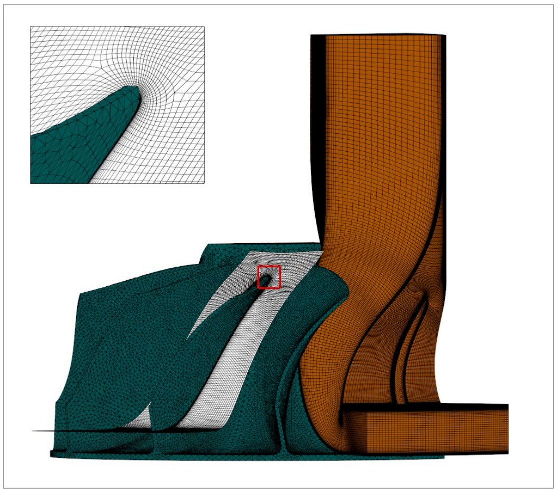


Figure 4: Solid unstructured mesh (green) and fluid structured mesh (white, orange) of a radial turbine

the mesh generation algorithm, which takes a hierarchical approach, going through the vertices, edges, faces, and finally generating the 3D solid mesh $\boldsymbol{x}_{solid}$ as portrayed in figure 5.

First, the vertices are simply meshed to be in the same position as their corresponding points according to the topology. The edges formed by the vertices are then meshed based on user-specific quality metrics and the curvature. After the edges have been meshed, the surrounded faces are initially meshed in parametric space using a Delaunay triangulation. The internal domain is then meshed based on the the grid size and quality defined by the user. Finally, the meshed faces serve as boundaries surrounding an internal volume, which is meshed using a 3D Delaunay triangulation. The resulting conformity with the CAD geometry is shown in figure 4, which shows the generated solid and fluid meshes.

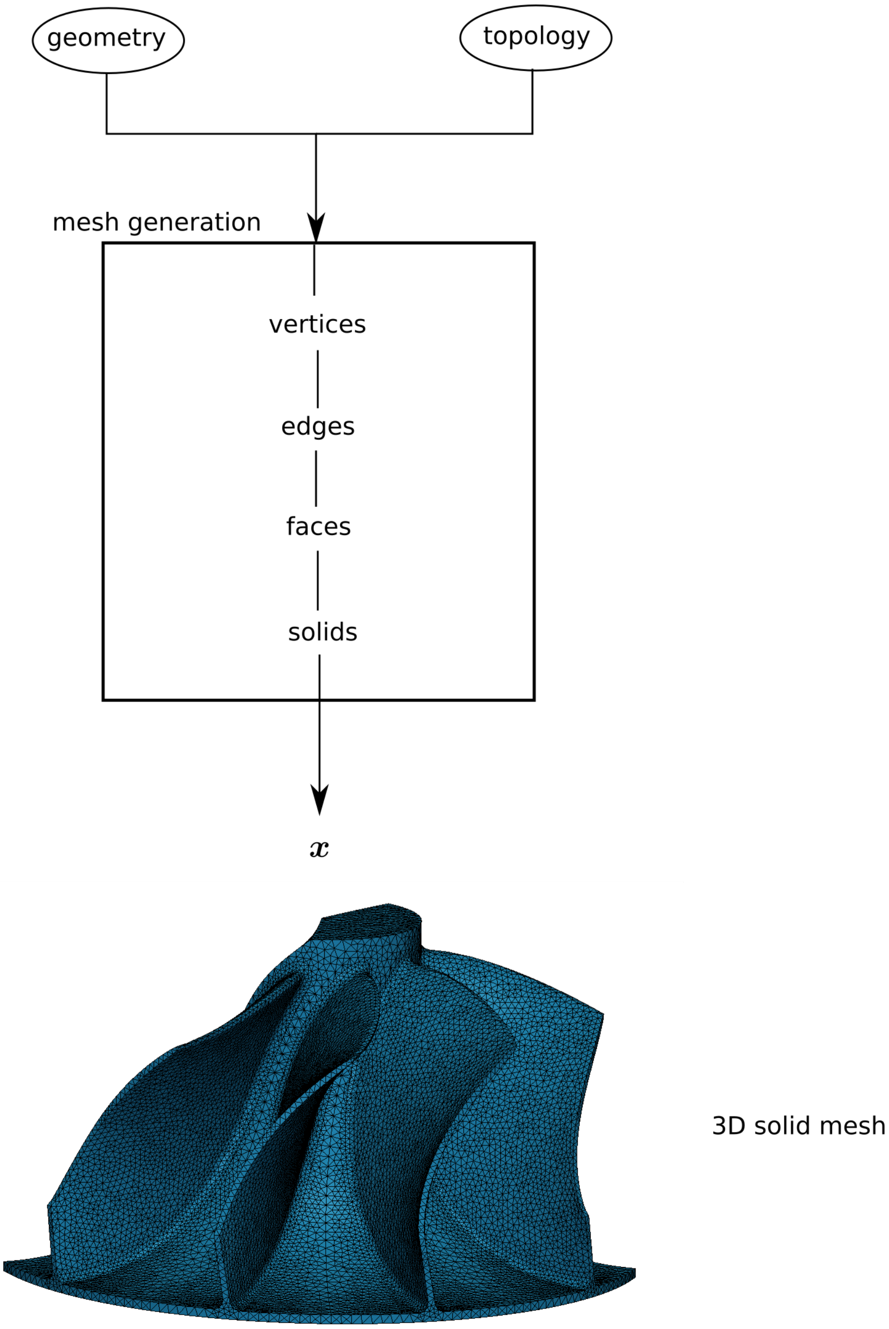
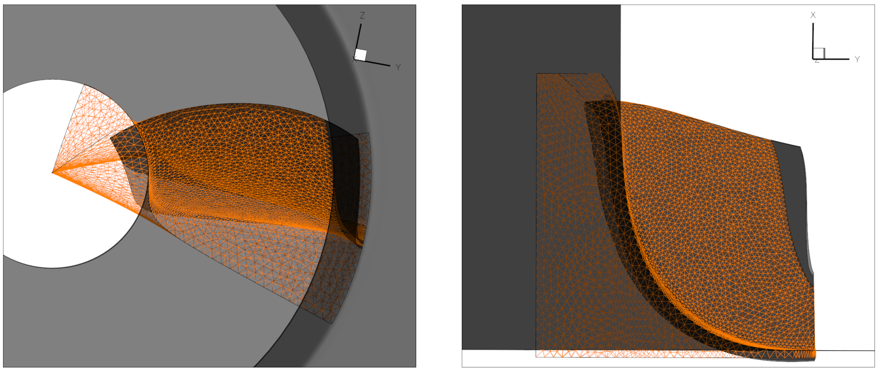


Figure 5: Flowchart of unstructured meshing procedure



(a) axial view (b) radial view

Figure 6: CAD geometry (gray shade) and unstructured mesh (orange)

3.4.3 Linking Unstructured Mesh and CAD Geometry

As the CAD geometry is updated during the optimization, the unstructured mesh must be updated as well. Recall that the CAD surface serves as the interface between the geometry and the mesh (figure 6). Using this interface, a CAD-based mesh deformation method is implemented to ensure the link between FEM mesh and CAD geometry remains intact.

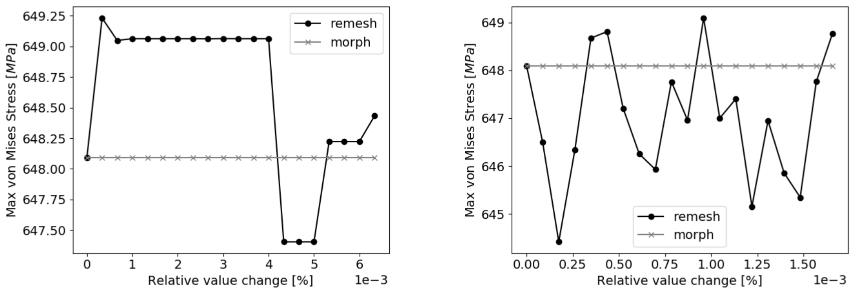
The optimization algorithm requires the gradients with respect to the CAD parameters $\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}}$, which can be acquired via the chain rule

$$\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}} = \frac{\partial \sigma_{max}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}. \tag{40}$$

The first term $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}}$ is computed by the adjoint structural solver and represents the stress gradients with respect to the FEM mesh nodes. The second term $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}$ could be computed by differentiating either the unstructured mesh generation or a corresponding mesh deformation.

For the fluid discipline, remeshing the structured mesh at each design iteration keeps the mesh topology in tact, such that a differentiated mesh generation can be easily used. However, for the solid discipline, rerunning the unstructured mesh generation for each design step could potentially change the mesh topology and mesh node count, resulting in discontinuities in the objective function.

Take for example the plots shown in figure 7, which show the change in maximum von Mises stress based on the value changes of two different CAD parameters. The initial value $\alpha_{i,0}$ is increasingly perturbed by a value Δ. For one CAD parameter (figure 7a, $\Delta = 10^{-6}$), the maximum von Mises stress value remains constant for certain regions of parameter values and the discontinuities can be associated with mesh topology changes. Other CAD parameters, on the other hand, are even more sensitive and have different maximum von Mises stress values for each parameter update (figure 7b, $\Delta = 10^{-8}$). This can be avoided by remeshing as little as possible and morphing the mesh instead,



(a) $\alpha_{i,0} = 0.3,\ \Delta = 10^{-6}$
`Blade.TESweep.Point.3.X`

(b) $\alpha_{i,0} = 0.01146789,\ \Delta = 10^{-8}$
`Meridional.Patch2.HubPoint.2.Y`

Figure 7: Comparison of σ_{max} over 20 iterations of updating parameters after remeshing (black) or deforming the mesh (grey). The initial CAD parameter value $\alpha_{i,0}$ is perturbed by Δ over 20 iterations. The relative value change is with respect to the initial value $\alpha_{i,0}$.

which, as seen in both plots of figure 7, has no significant effect on the quantity of interest. Thus, a CAD-based mesh deformation that maintains the mesh topology is preferred.

The mesh deformation is based on the method described in [77] and can be summarized as a three-step process:

1. morph edge nodes
2. morph face nodes
3. morph inner nodes

In contrast to the original work, the inner mesh nodes are displaced using a linear elastic analogy instead of an inverse distance interpolation. The first two steps of the mesh deformation are forward differentiated using AD, while the third step makes use of the adjoint linear elastic solver. This method makes use of the adjoint structural solver and has shown to require less remeshing than the inverse distance method with this geometry [59]. The remainder of this section is dedicated to explaining the three steps of the mesh deformation.

1. Morph Edge Nodes

The edges of the mesh as shown in figure 8 are deformed based on the displaced edges of the corresponding CAD geometry. The CAD edges are defined using a B-spline curve $C(u)$, where the first and last points $C(u_B), C(u_E)$ are identical to the first and last vertex points of the mesh edge. From the CAD update, the displacements of u_B, u_E are known and can be used to define the displacements of the first and last vertices of the corresponding mesh edge. The end vertices are then used to solve for the remaining interior edge mesh points in parametric space. Solving for these mesh points in parametric space reduces the dimension to 1 (u) and implicitly applies the constraint of keeping

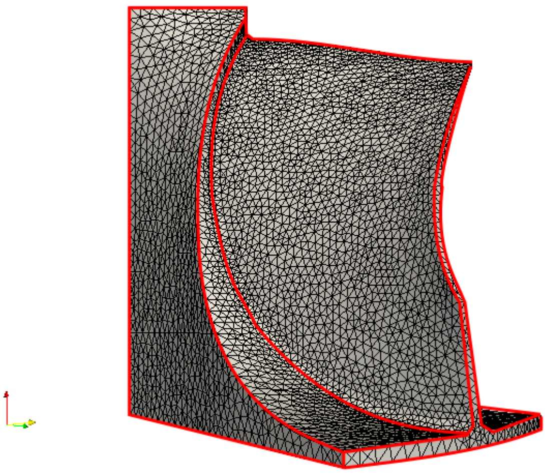


Figure 8: Highlighted edges of an unstructured mesh of a radial turbine

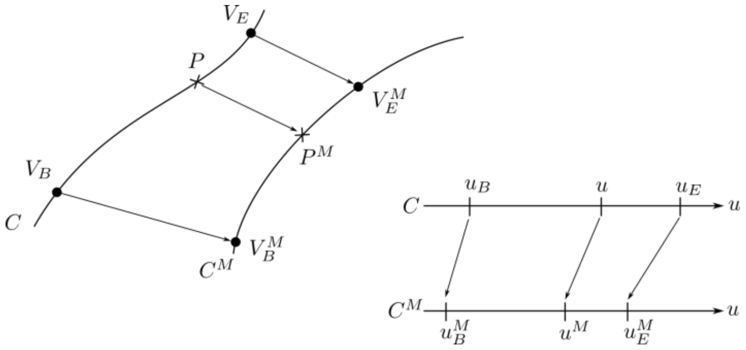


Figure 9: Deformation of mesh edge. Left: in 2D Cartesian space. Right: in 1D parametric CAD space

the mesh points along the B-spline curve.

Figure 9 illustrates this procedure. The original B-spline curve C with end vertices V_B and V_E is morphed to curve C^M. As a result, the end points are morphed to V_B^M and V_E^M, which can be directly computed from $C(u_B^M), C(u_E^M)$. The inner point P has to be morphed to $P^M = C(u^M)$. This is done in parametric space by solving

$$u^M = u_B^M + \frac{u_E^M - u_B^M}{u_E - u_B} \left(u - u_B\right), \tag{41}$$

followed by a relaxation using the linear spring analogy. Applying this procedure for each edge along the CAD faces results in the deformation of the mesh edge nodes.

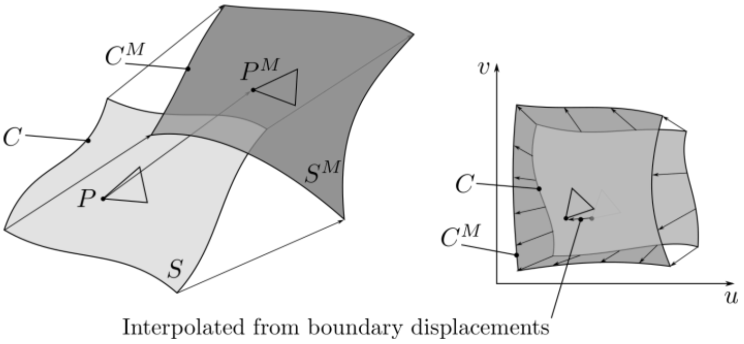


Figure 10: Morphing of face mesh nodes. Left: 3D Cartesian space. Right: 2D parametric CAD space.

2. Morph Face Nodes

Once the mesh nodes along all edges have been updated, the rest of the outer mesh nodes need to be updated as well. These nodes coincide with the CAD faces, which are defined by B-spline surfaces $S(u, v)$. Using the morphed edges from the previous step, the deformation of the inner nodes on the face can be computed. This is illustrated in figure 10, where the face S is updated to S^M. The edges of the face C^M are already known from the previous step and can be used as boundary conditions to solve for the inner mesh points P^M using an inverse distance interpolation. Analogously to the previous step, this is done in parametric (u, v) space to reduce the dimensionality to 2 and to constrain the deformation on the B-spline surface. This process is repeated for each face of the geometry, which completes the deformation of all outer mesh nodes.

3. Morph Inner Nodes

After the outer mesh nodes have been deformed by the previous two steps, a linear elastic problem can be set up to solve for the remaining inner mesh node deformations $\boldsymbol{u}_{inner}$. This is achieved by defining the previously computed outer node deformations $\boldsymbol{u}_{outer}$ as the boundary conditions of a linear elastic problem

$$A\boldsymbol{u} = \boldsymbol{b}, \tag{42}$$

which can be solved using the CSM solver discussed in chapter 4. The computed displacements $\boldsymbol{u}$ can then be used to update the entire mesh according to

$$\boldsymbol{x}^{i+1} = \boldsymbol{x}^{i} + \boldsymbol{u}, \tag{43}$$

keeping the mesh topology unchanged. The new mesh can then be used to perform CSM simulations for the next optimization iteration.

3.5 Computational Fluid Dynamics Solver

Primal

To numerically model the fluid dynamics that is typical for turbomachinery applications, the Navier-Stokes equations for a rotating frame of reference are augmented by a turbulence model. In this work, the widely-used compressible steady state *Reynolds-Averaged Navier Stokes* (RANS) formulation is used, where the Navier-Stokes equations are averaged using Reynolds time-averaging.

The in-house developed RANS solver of Müller [45, 47] is utilized, which uses a cell-centered finite volume method for its spatial discretization. The RANS formulation is closed using the negative Spalart-Allmaras turbulence model [2], assuming a fully turbulent inflow condition. For the temporal discretization, an implicit time integration scheme based on the JT-KIRK scheme [81] is used, utilizing a multistage Runge-Kutta time-stepping method of the form

$$\vec{W}^{(0)} = \vec{W}^{t} \tag{44}$$

$$\boldsymbol{P}\left[\vec{W}^{(1)} - \vec{W}^{(0)}\right] = -\alpha_1 \boldsymbol{R}\left(\vec{W}^{(0)}\right) \tag{45}$$

$$\vdots \tag{46}$$

$$\boldsymbol{P}\underbrace{\left[\vec{W}^{(m)} - \vec{W}^{(0)}\right]}_{\Delta\vec{W}^{(m)}} = -\alpha_m \boldsymbol{R}\left(\vec{W}^{(m-1)}\right) \tag{47}$$

$$\vec{W}^{t+1} = \vec{W}^{t} + \Delta\vec{W}^{(m)}, \tag{48}$$

with time step t, Runge-Kutta coefficients α_m, and m Runge-Kutta stages. Using an approximation of the Jacobian $\frac{\partial \boldsymbol{R}}{\partial \vec{W}}$ based on the first order residual $\tilde{\boldsymbol{R}}$, the system matrix $\boldsymbol{P}$ is defined as

$$\boldsymbol{P} = \frac{\Omega}{\Delta t}\boldsymbol{I} + \frac{\partial \tilde{\boldsymbol{R}}}{\partial \vec{W}}, \tag{49}$$

with the control volume Ω and identity matrix $\boldsymbol{I}$.

The solver is parallelized with Open MPI [17] and the time integration is accelerated by the use of local time-stepping and geometric multigrid methods [50]. The spatial discretization of the inviscid fluxes is implemented using a Roe flux-splitting scheme [54] with second-order accuracy due to the MUSCL reconstruction scheme [70]. A van-Albada flux limiting method [72] is used to handle shock oscillations and the numerical dissipation is treated using the entropy correction according to Harten and Hyman [23].

Adjoint

As discussed in section 3.1.3, the adjoint method is preferred in gradient calculations when the number of outputs is far less than the number of inputs. For CAD-based optimizations, one may typically have numerous CAD design variables $\boldsymbol{\alpha} \in \mathbb{R}^n$ and a one-dimensional objective $J \in \mathbb{R}$. The adjoint formulation of the CFD solver, which is discussed e.g. in [18], can be derived by first differentiating the steady state RANS solution

$$\boldsymbol{R}\left(\vec{W}\right) = 0 \tag{50}$$

by the design parameters $\boldsymbol{\alpha}$:

$$\frac{d\boldsymbol{R}}{d\boldsymbol{\alpha}} = \frac{\partial \boldsymbol{R}}{\partial \vec{W}}\frac{\partial \vec{W}}{\partial \boldsymbol{\alpha}} + \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{\alpha}} = 0. \tag{51}$$

This can then be reformulated as

$$\frac{\partial \vec{W}}{\partial \boldsymbol{\alpha}} = -\frac{\partial \boldsymbol{R}}{\partial \vec{W}}^{-1}\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{\alpha}}. \tag{52}$$

Next, the objective function $J(\vec{W}, \boldsymbol{\alpha})$ is differentiated as

$$\frac{dJ}{d\boldsymbol{\alpha}} = \frac{\partial J}{\partial \boldsymbol{\alpha}} + \frac{\partial J}{\partial \vec{W}}\frac{\partial \vec{W}}{\partial \boldsymbol{\alpha}} \tag{53}$$

and the derivative of the state variables (52) is inserted

$$\frac{dJ}{d\boldsymbol{\alpha}} = \frac{\partial J}{\partial \boldsymbol{\alpha}} + \underbrace{-\frac{\partial J}{\partial \vec{W}}\frac{\partial \boldsymbol{R}}{\partial \vec{W}}^{-1}}_{:=\boldsymbol{\psi}^T}\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{\alpha}} \tag{54}$$

$$\frac{dJ}{d\boldsymbol{\alpha}} = \frac{\partial J}{\partial \boldsymbol{\alpha}} + \boldsymbol{\psi}^T\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{\alpha}}, \tag{55}$$

where $\boldsymbol{\psi}$ defines the adjoint variable and the remaining partial derivatives $\frac{\partial J}{\partial \boldsymbol{\alpha}}$, $\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{\alpha}}$ can be computed with minimal computational effort. The adjoint variable $\boldsymbol{\psi}$ can be solved for via the adjoint equation

$$\frac{\partial \boldsymbol{R}}{\partial \vec{W}}^T \boldsymbol{\psi} = -\frac{\partial J}{\partial \vec{W}}^T \tag{56}$$

and the adjoint residual $\boldsymbol{R}_{\boldsymbol{\psi}}$ is defined as

$$\boldsymbol{R}_{\boldsymbol{\psi}} = \frac{\partial J}{\partial \vec{W}}^T + \frac{\partial \boldsymbol{R}}{\partial \vec{W}}^T \boldsymbol{\psi}. \tag{57}$$

$\frac{\partial \boldsymbol{R}}{\partial \vec{W}}^T$ is the transpose of the Jacobian of the primal $R(\vec{W}) = 0$. In principal, solving the linear system of equation of (56) would directly lead to the adjoint solution $\boldsymbol{\psi}$. However, in practice, this linear system is very stiff and requires a significant effort to solve. This is the reason for choosing the approximate Jacobian in equation (49) to solve the primal.

To solve equation (56) with the same logic as the primal, the equation is rewritten as (57) and the same time-marching method used to solve the primal flow equations is employed, except with the transpose of the system matrix $\boldsymbol{P}$:

$$\boldsymbol{\psi}^{(0)} = \boldsymbol{\psi}^{t} \tag{58}$$

$$\boldsymbol{P}^{T}\left[\boldsymbol{\psi}^{(1)} - \boldsymbol{\psi}^{(0)}\right] = -\alpha_{1}\boldsymbol{R}_{\boldsymbol{\psi}}\left(\vec{W}, \boldsymbol{\psi}^{(0)}\right) \tag{59}$$

$$\vdots \tag{60}$$

$$\boldsymbol{P}^{T}\underbrace{\left[\boldsymbol{\psi}^{(m)} - \boldsymbol{\psi}^{(0)}\right]}_{\Delta\boldsymbol{\psi}^{(m)}} = -\alpha_{m}\boldsymbol{R}_{\boldsymbol{\psi}}\left(\vec{W}, \boldsymbol{\psi}^{(m-1)}\right) \tag{61}$$

$$\boldsymbol{\psi}^{t+1} = \boldsymbol{\psi}^{t} + \Delta\boldsymbol{\psi}^{(m)} \tag{62}$$

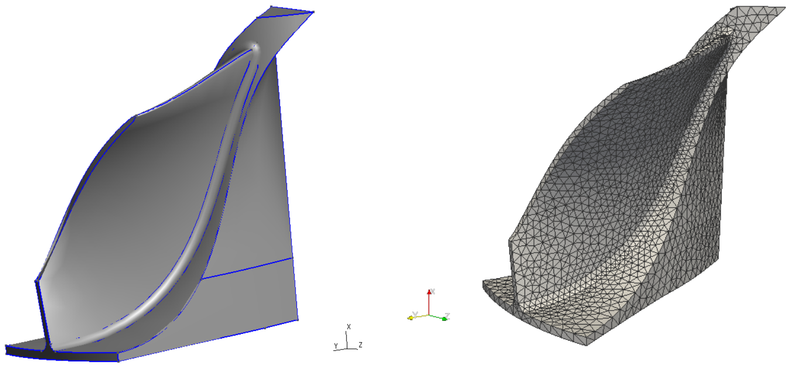
The system matrix $\boldsymbol{P}$ is given by the steady state solution of the primal. Because merely a transpose of the primal system matrix $\boldsymbol{P}^{T}$ is required, the adjoint solver has approximately the same memory requirements as the primal solver. In practice, its memory foot print is slightly higher, because $\frac{\partial \boldsymbol{R}}{\partial \vec{W}}$ is stored in memory for the convenience of evaluating (57) immediately. Additionally, the transposed system matrix $\boldsymbol{P}^{T}$ has the same eigenspectrum as $\boldsymbol{P}$, resulting in an adjoint convergence rate equivalent to that of the primal solver, where $\boldsymbol{P}$ is frozen during the last iterations and no longer updated. The partial derivatives in equation (56) are computed via hand-differentiated AD with aid of the source code transformation tool *Tapenade* [24]. A more detailed discussion of the CFD solver and its adjoint counterpart can be found in the doctoral dissertation of Müller [45].

4 Computational Structural Mechanics Solver

This chapter introduces the computational structural mechanics (CSM) solver which was developed and differentiated within this work. The corresponding governing equations are the linear elastic equations (72) and the free vibration equations (81). In order to solve these equations numerically, the finite element method (FEM) discretization is used. Additionally, structural gradients are required for the gradient-based optimization (39), which are preferrably computed at a low computational cost. This chapter introduces the FEM code's structure, which enables a straightforward discretization using algorithmic differentiation, discussed in chapter 5.

4.1 Finite Element Method

The finite element method (FEM) [83] is a widely used numerical discretization method for solving partial differential equations. The basic idea of the FEM method is to reduce the continuous problem, i.e. the domain, into multiple *finite* discrete problems (fig. 11), which can be solved using algebraic equations.



(a) CAD geoemetry of a radial turbine

(b) FEM mesh of a radial turbine

Figure 11: FEM divides a continuous (left) domain into a set of multiple discrete elements (right)

This is achieved by using the Galerkin method to approximate a continuous variable, e.g., the displacement u, by a linear combination of the form

$$
\begin{aligned}
u &\approx u_h \\
u_h &= \sum_i u_i N_i,
\end{aligned}
$$

where N_i are so-called *test* or *shape* functions. The shape functions are dependent on the type of finite elements chosen for the discretization, e.g., 2D linear squares (4 nodes), 3D linear hexahedral elements (8 nodes) and quadratic tetrahedral elements (10 nodes). Furthermore, the shape functions determine the polynomial form of the quantity over the element - e.g. whether the value of the displacement u changes linearly or quadratically within the element. The accuracy of the discretization is thus highly dependent on the elements chosen, as well as the number of elements used, i.e., how fine the FEM mesh is. In the context of a linear stress and free vibration analysis, the size of the resulting systems (72), (81) is directly dependent on the number of FEM nodes m.

A short introduction of the finite element formulations for the linear stress and vibration analyses is given here. For a thorough discussion of the basics of the FEM method, the interested reader is referred to the book of Zinkiewicz [83] and for its application in structural mechanics within turbomachinery, the book of Dhondt [12].

4.1.1 Linear Stress Analysis

For the derivation of the FEM formulation [82], we begin by rewriting the governing equations of linear elasticity introduced in section 2.2 into matrix form. The strain-displacement equation (27) can be written as

$$\boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{xz} \\ \gamma_{yz} \\ \gamma_{zx} \end{pmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \boldsymbol{S}\boldsymbol{u}, \tag{63}$$

with the differential operator $\boldsymbol{S}$ and the displacement field $\boldsymbol{u}$. The matrix form of the equilibrium equations (26) can be formulated as

$$\boldsymbol{S}^T\boldsymbol{\sigma} + \boldsymbol{b} = \boldsymbol{0} \tag{64}$$

and the constitutive equations (28), also known as the *stress-strain equations*, can be written as

$$\boldsymbol{\sigma} = \boldsymbol{D}(\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_0) + \boldsymbol{\sigma}_0, \tag{65}$$

with the *elasticity matrix* $\boldsymbol{D}$. $\boldsymbol{\epsilon}_0$ and $\boldsymbol{\sigma}_0$ denote the initial strains and initial stresses, respectively, which may be present at rest, i.e., before loads are applied. The elasticity matrix for isotropic materials is given as

$$\boldsymbol{D} = C_\sigma \begin{bmatrix} (1-\nu) & \nu & \nu & 0 & 0 & 0 \\ \nu & (1-\nu) & \nu & 0 & 0 & 0 \\ \nu & \nu & (1-\nu) & 0 & 0 & 0 \\ 0 & 0 & 0 & (1-2\nu)/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & (1-2\nu)/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & (1-2\nu)/2 \end{bmatrix}, \tag{66}$$

with Poisson's ratio ν, Young's modulus E, and

$$C_\sigma := \frac{E}{(1+\nu)(1-2\nu)}. \tag{67}$$

Plugging (65) into the principle of virtual work (34), the approximate weak form is given by

$$\int_\Omega \delta\boldsymbol{\epsilon}^T \left[\boldsymbol{\sigma}_0 + \boldsymbol{D}(\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_0)\right] d\Omega = \int_\Omega \delta\boldsymbol{u}^T \boldsymbol{f} d\Omega + \int_{\Gamma_t} \delta\boldsymbol{u}^T \bar{\boldsymbol{t}} d\Gamma. \tag{68}$$

Here the boundary Γ is divided into Γ_u and Γ_t, whereby the virtual displacements are zero $\delta \boldsymbol{u} = 0$ along the boundary Γ_u. The boundary conditions are given by the traction forces $\bar{\boldsymbol{t}}$ on the boundary Γ_t and known displacements $\bar{\boldsymbol{u}}$ on the boundary Γ_u where $\boldsymbol{u} = \bar{\boldsymbol{u}}$. Using the Galerkin method, the displacements are approximated as

$$\boldsymbol{u} \approx \hat{\boldsymbol{u}} = \sum_i N_i \tilde{\boldsymbol{u}}_i \tag{69}$$

and the strains are approximated as

$$\boldsymbol{\epsilon} \approx \hat{\boldsymbol{\epsilon}} = \sum_i \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_i}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_i}{\partial z} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial z} & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} & 0 & \frac{\partial N_i}{\partial x} \end{bmatrix} \tilde{\boldsymbol{u}}_i = \sum_i \boldsymbol{B}_i \tilde{\boldsymbol{u}}_i. \tag{70}$$

Introducing the approximations (69) and (70) into the weak form (68), leads to

$$\delta \tilde{\boldsymbol{u}}_i^T \left[\int_\Omega \boldsymbol{B}_i^T \left[\boldsymbol{\sigma}_0 + \boldsymbol{D} \left(\boldsymbol{B}_j \tilde{\boldsymbol{u}}_j - \boldsymbol{\epsilon}_0 \right) \right] d\Omega \right] = \delta \tilde{\boldsymbol{u}}_i^T \left[\int_\Omega N_i \boldsymbol{f} d\Omega + \int_{\Gamma_t} N_i \bar{\boldsymbol{t}} d\Gamma \right]. \tag{71}$$

This equation holds for any virtual displacement $\delta \tilde{\boldsymbol{u}}_i$, such that the multipliers must be equal and $\delta \tilde{\boldsymbol{u}}_i^T$ can be dropped. Rearranging the result finally leads to a linear system of equations of the form

$$K \tilde{\boldsymbol{u}} + \boldsymbol{b} = \boldsymbol{0}, \tag{72}$$

where $K \in \mathbb{R}^{3m \times 3m}$ is the stiffness matrix, $\boldsymbol{b} \in \mathbb{R}^{3m}$ the load vector, and $\tilde{\boldsymbol{u}} \in \mathbb{R}^{3m}$ the displacements. Splitting up the integrals into summations of integrals over a number of *finite elements*, the global matrix K and load vector $\boldsymbol{b}$ can be assembled together from elemental components

$$K_{ij} = \sum_e K_{ij}^e, \quad \boldsymbol{b}_i = \sum_e \boldsymbol{b}_i^e. \tag{73}$$

The elemental stiffness matrix K^e and load vector $\boldsymbol{b}^e$ are computed via

$$\begin{aligned} K_{ij}^e &= \int_{\Omega^e} \boldsymbol{B}_i^T \boldsymbol{D} \boldsymbol{B}_j d\Omega \\ \boldsymbol{b}_i^e &= \int_{\Omega^e} \left[\boldsymbol{B}_i^T \left(\boldsymbol{\sigma}_0 - \boldsymbol{D} \boldsymbol{\epsilon}_0 \right) - N_i \boldsymbol{f} \right] d\Omega - \int_{\Gamma_t^e} N_i \bar{\boldsymbol{t}} d\Gamma. \end{aligned}$$

The number of FEM nodes is given by m and the factor 3 in $\mathbb{R}^{3m}$ is due to this being a three-dimensional formulation. The linear system is typically solved using an iterative linear system solver provided by, e.g., the Eigen [21] or PETSc library [5], whereby smaller linear systems can also be solved using a direct linear solver.

The stiffness matrix and load vector are mainly dependent on the given material properties and the loads defined by the problem. For turbomachinery applications, the most significant loads are given by the centrifugal forces, which are generated by high rotational speeds. For structural constraints resulting from a stress analysis, the quantity of interest is the maximum von Mises stress $\sigma_{max} \in \mathbb{R}$, which can be evaluated as a post-processing step after the displacements $\boldsymbol{u}$ have been computed. This is achieved by first computing the strains via the strain-displacement equation (27, 70) and then determining the stress tensor according to the stress-strain equations (28, 65), which is typically done on the integration points of the FEM elements. The von Mises stress σ_{vm} for each integration point can then be computed via

$$\sigma_{vm}^2 = \frac{1}{2}\left[(\sigma_{0,0} - \sigma_{1,1})^2 + (\sigma_{1,1} - \sigma_{2,2})^2 + (\sigma_{2,2} - \sigma_{0,0})^2 + 6(\sigma_{1,2}^2 + \sigma_{2,0}^2 + \sigma_{0,1}^2)\right] \tag{74}$$

4.1.2 Free Vibration Analysis

The discrete form of the free vibration analysis results from plugging the body force vector (36), including inertial forces, into the Galerkin approximation of the principle of virtual work (71) [82]. This essentially results in the body force term being replaced by

$$-\int_{\Omega^e} \boldsymbol{N}^T \boldsymbol{f} d\Omega = -\int_{\Omega^e} \boldsymbol{N}^T \bar{\boldsymbol{f}} d\Omega + \int_{\Omega^e} \boldsymbol{N}^T \rho \ddot{\boldsymbol{u}} d\Omega, \tag{75}$$

whereby the body forces are now defined in $\bar{\boldsymbol{f}}$. In matrix form, this results in

$$K\boldsymbol{u} + M\ddot{\boldsymbol{u}} = \boldsymbol{b} \tag{76}$$

with the *mass matrix* $M \in \mathbb{R}^{3m \times 3m}$, where the elemental mass matrix M^e is computed by

$$M_{ij}^e = \int_{\Omega^e} \boldsymbol{N}_i^T \rho \boldsymbol{N}_j d\Omega. \tag{77}$$

Without external forcing terms, the load vector is simplified to $\boldsymbol{b} = \boldsymbol{0}$, such that the equation reduces to

$$K\boldsymbol{u} + M\ddot{\boldsymbol{u}} = \boldsymbol{0}. \tag{78}$$

The proposed solution of

$$\boldsymbol{u} = \boldsymbol{u}_k e^{i\omega_k t} \tag{79}$$

leads to a generalized eigenvalue system of the form

$$K\boldsymbol{u}_k - \omega_k^2 M\boldsymbol{u}_k = \boldsymbol{0}. \tag{80}$$

From the defintion $\lambda_k := \omega_k^2$ follows

$$\left(K(\boldsymbol{x}) - \lambda_k M(\boldsymbol{x})\right)\boldsymbol{u}_k = \boldsymbol{0}, \tag{81}$$

where λ_k is the k-th eigenvalue and $\boldsymbol{u}_k$ the k-th eigenvector. K is the same stiffness matrix used in the linear stress analysis (72) and can thus be recycled without additional costs. The generalized eigenvalue system (81) can be solved using an iterative eigenvalue solver provided, e.g., by the SLEPc library [25], resulting in the eigenvalues $\boldsymbol{\lambda}$ and the eigenvectors $\boldsymbol{u}$.

4.2 Finite Element Code Structure

Applied to structural mechanics, the FEM method essentially boils down to solving one global linear system (72) for the linear elastic analysis and one generalized eigenvalue problem (81) for the free vibration analysis. While the process of assembling and solving these global systems is generally the same, the building blocks, i.e., the shape of the individual elements or the stiffness matrix calculations can differ. For example, one could choose to discretize their FEM model using either linear 3D brick elements or quadratic 3D tetrahedral elements, both using different shape functions. The stiffness of the material could also be computed differently, e.g. either for *isotropic* materials or *composite* materials.

To take these different constellations into account and facilitate future code contributions, a modularized, object-oriented design is used. This section introduces the structure of the developed FEM code, followed by its differentiation in the following section, section 5.

4.2.1 Shape Functions

The most basic building block of the FEM method is the underlying shape function $\boldsymbol{N}$ used within the element. For all the different element shapes, a parent class `Shape` is defined that has the following pure virtual functions:

- `ShapeFunction` - defines the shape function $\boldsymbol{N}$ of the element
- `ShapeDerivative` - defines the derivative $\frac{\partial \boldsymbol{N}}{\partial \boldsymbol{\xi}}$ of the shape function with respect to the local coordinates
- `ExtrapolateValue` - extrapolates a value from the integration points to the FEM nodes

Each of the implemented child classes overload the virtual functions of the parent class `Shape`. Shape-specific features, such as the `extrapolation_matrix`, `integration_weights`, and `integration_points` are defined in the corresponding constructors of the child classes.

For the same kind of element, e.g., tetrahedral elements, one can have linear shape functions with four nodes or quadratic shape functions with ten nodes. This is visualized in figure 12, where e.g. the `FourNodeTetrahedra` class defines 1st order tetrahedras and the `TenNodeTetrahedra` class defines 2nd order tetrahedras. Both classes have their own shape functions and derivatives, but share the same extrapolation matrix. The FEM solver only deals with the parent class `Shape`, such that additional element shapes, e.g., `Brick` elements, can be added with minimal code refactoring.

4.2.2 Finite Elements

The `Element` class is used to define the structure of the element, i.e., the nodes that belong to it, as well as the physical side of the modelling, i.e., calculating the

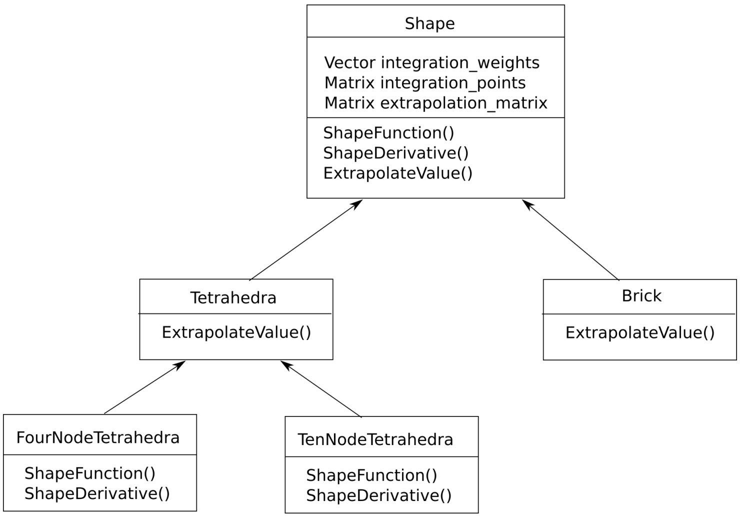


Figure 12: Class diagram of parent class `Shape` and child classes `Tetrahedra` and `Brick`, with the child classes of `Tetrahedra`: `FourNodeTetrahedra` (1st order elements) and `TenNodeTetrahedra` (2nd order elements)

local stiffness matrix, mass matrix, and load vector. These can differ e.g. when using an isotropic or composite material. This class also has a `Shape` object to define its shape as described above. The parent class `Element` contains functions for the assembly phase, which can be overloaded by child classes:

- `ComputeLocalStiffnessMatrix` - computes the element-local stiffness matrix K_e
- `ComputeLocalMassMatrix` - computes the element-local mass matrix M_e
- `ComputeLocalRHS` - computes the element-local load vector $\boldsymbol{b}_e$

A visualization is shown in figure 13, where two child classes `IsotropicElement` and `CompositeElement` are implemented with assembly functions that overload the parent class functions. With this design, alternative stiffness and mass matrix, as well as load vector calculations can be easily introduced by implementing a child class of `Element`. The algorithm for adding the element-local matrix contributions to the global matrix is independent of the local matrix calculation, as the assembly algorithm only requires the parent class `Element`.

4.2.3 System Assembly

The global stiffness matrix K, mass matrix M, and right-hand side $\boldsymbol{b}$ are constructed by looping over all discrete elements and adding their local contributions K_e, M_e, and $\boldsymbol{b}_e$ to the global system. For a three dimensional problem, the dimension of the global matrices is $3m \times 3m$ and $3m$ for the load vector, where m is the total number of FEM nodes. The size of the local matrices is dependent on the choice of shape functions, i.e., the number FEM nodes in each element. The local matrix and vector contributions are computed using element-level functions as described in the previous section.

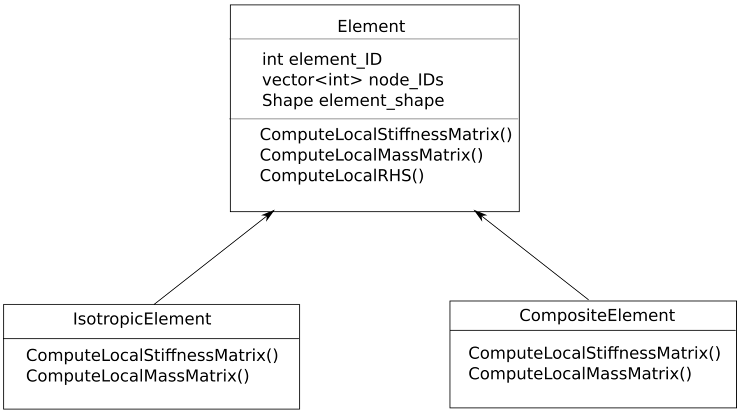


Figure 13: Class diagram of parent class `Element` and child classes `IsotropicElement` and `CompositeElement`

When the local contributions are added to the global system, the indices of the contributions depend on the unique global index of the corresponding FEM nodes. A visualization of the assembly procedure is shown in figure 14.

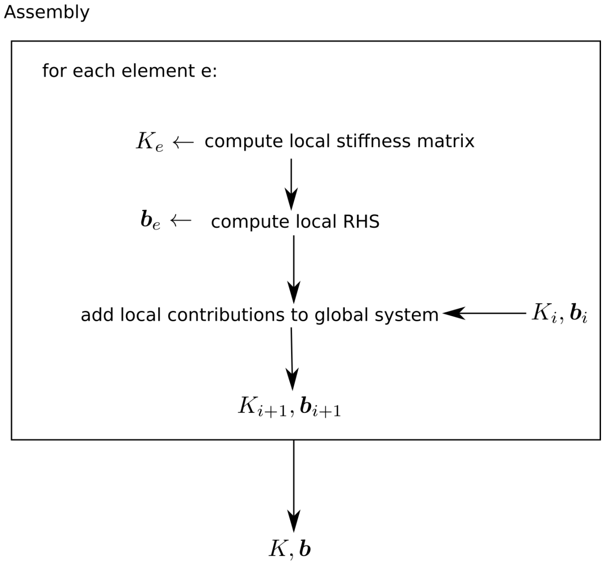

Assembly
for each element e:
$K_e \leftarrow$ compute local stiffness matrix
$\boldsymbol{b}_e \leftarrow$ compute local RHS
add local contributions to global system
$K_i, \boldsymbol{b}_i$
$K_{i+1}, \boldsymbol{b}_{i+1}$
$K, \boldsymbol{b}$

Figure 14: Visualization of system assembly. The global system matrix K and load vector $\boldsymbol{b}$ are made up of element-level local contributions.

Due to the sparse form of the resulting global matrices K, M and vector $\boldsymbol{b}$, `SparseMatrix` and `SparseVector` types of the *Eigen* [21] library are used. To speed up the assembly, the sparsity pattern is computed before actually computing the local element contributions. This is done by looping over all elements once and storing the number of global indices of the local contributions per row in a `sparsity` vector. The required size per row is then pre-allocated using the `reserve(sparsity)` function for Eigen's sparse matrices, resulting in an efficient system assembly.

4.2.4 Linear Stress Analysis

As discussed in section 3.2, the goal of differentiating the linear stress analysis is to be able to compute the gradient

$$\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}} \in \mathbb{R}^{3m} \tag{82}$$

efficiently using AD. To allow a more straightforward differentiation using AD, the stress analysis was logically divided into three major steps:

1. Assembly of K and $\boldsymbol{b}$
2. Solution of linear system $K\boldsymbol{u} = \boldsymbol{b}$
3. Calculation of maximum von Mises stress $\sigma_{max}(\boldsymbol{u})$

These steps are visualized in figure 15, where the node coordinates $\boldsymbol{x}$ are considered as the input parameters to the chain of functions. One of the main reasons for segregating the code into these three distinct steps is to enable the specific handling of differentiating a linear solver (sec. 5.2) and reducing the memory consumption of the assembly phase (sec. 5.4). Additionally, segragating the linear solver in this manner allows one to select a suitable iterative linear solver from different third party libraries. The three major steps are handled by the `Solver` class, where the inputs and outputs of the corresponding functions are clearly defined as in figure 15.

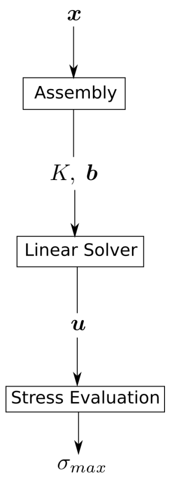


Figure 15: Outline and data flow of linear stress analysis algorithm

The implemented code of the three basic steps outlined above and shown in figure 15 uses only the parent class `Element`, allowing an easy extension of the code with different element types, as well as their implicit AD differentiation, as discussed in section 5.6.

4.2.5 Free Vibration Analysis

Analogously to the linear stress analysis implementation discussed in section 4.2.4, the free vibration analysis code is broken down into modular steps to ease the application of AD. The two main steps are visualized in figure 16, where the node coordinates $\boldsymbol{x}$ are considered the input variables and the eigenvalues $\boldsymbol{\lambda}$ as the output variables. Mainly, these steps are given as:

1. Assembly of K, M
2. Iterative solution of generalized eigenvalue problem (81)

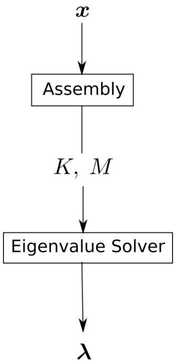


Figure 16: Outline and data flow of free vibration analysis algorithm

The first step, which is the assembly, is an extension of the same assembly function called in the linear stress analysis, since the stiffness matrix K is required for both analyses. A boolean flag `withVibration` is set to `true` when the mass matrix M is required. This way, while the elements are being looped over to compute the local stiffness matrix, the local mass matrix (`ComputeLocalMassMatrix`) is correspondingly computed and directly added into the global mass matrix.

For the second step, as was also done with the linear stress analysis, the iterative solver of the generalized eigenvalue system is separated from the rest of the code as it requires special treatment in differentiation, which is discussed in section 5.3. The iterative solver used in this thesis is the generalized eigenvalue solver using the Krylov-Schur method provided by the SLEPc library [25].

These steps are also handled by the `Solver` class, where a boolean flag from the input file triggers the use of the vibration analysis. In this case, the *assembly* also constructs the mass matrix M and calls the eigenvalue solver for the vibration analysis in addition to the linear solver for the stress analysis.

4.2.6 Mesh Deformation

The CSM solver, especially the linear elastic solver (sec. 4.2.4), discussed in this section mainly solves physical, structural problems. However, the same solver can be reused to tackle the mesh morphing of the unstructured mesh during

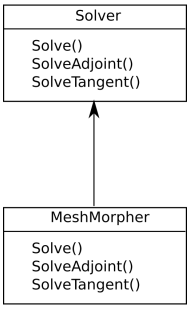


Figure 17: Class diagram of parent class `Solver` and child class `MeshMorpher`

the optimization by using a linear elastic analogy. A `MeshMorpher` class that inherits from the `Solver` class is defined to deal with any deformation-specific algorithms (fig. 17). These are mainly present in the tangent and adjoint version of the morpher, where different inputs and outputs are used compared to the adjoint linear stress analysis as described in section 5.5.1. The mesh deformation method is discussed in more detail in section 3.4.3 and its role in the gradient calculation is discussed in section 5.5.1.

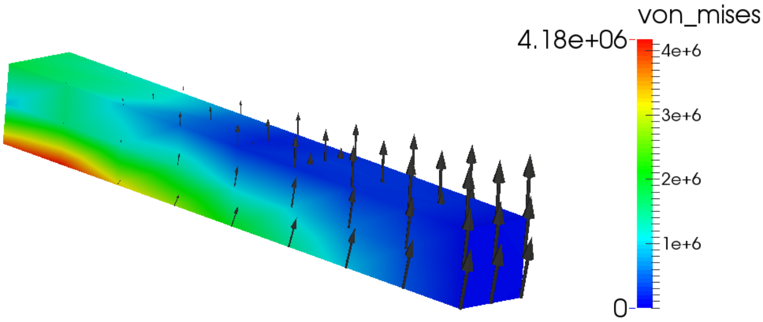


Figure 18: Rotating cantilever beam test case with von Mises stress σ_{VM} contours and displacement $\boldsymbol{u}$ vectors

4.3 Validation of FEM Solver

To validate the FEM solver implemented in this work, the well-known *Calculix* FEM solver [14, 12] is used as a reference. Two test cases are carried out:

1. A rotating beam test case consisting of a total of 90 FEM nodes (fig. 18)
2. A rotating axial fan blade test case consisting of a total of 11,382 FEM nodes (fig. 19)

Both test cases use quadratic, ten-node tetrahedral elements. The *cado* FEM solver can work with *Calculix* input files, hence identical input files are used for both FEM solvers in the following tests.

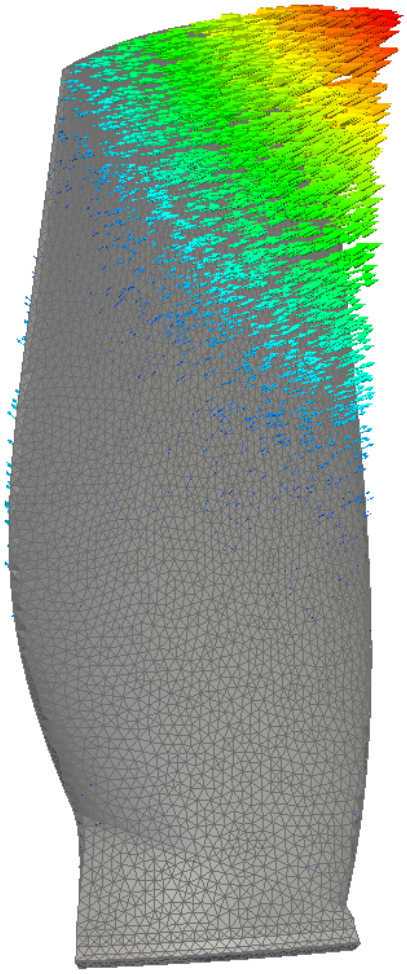
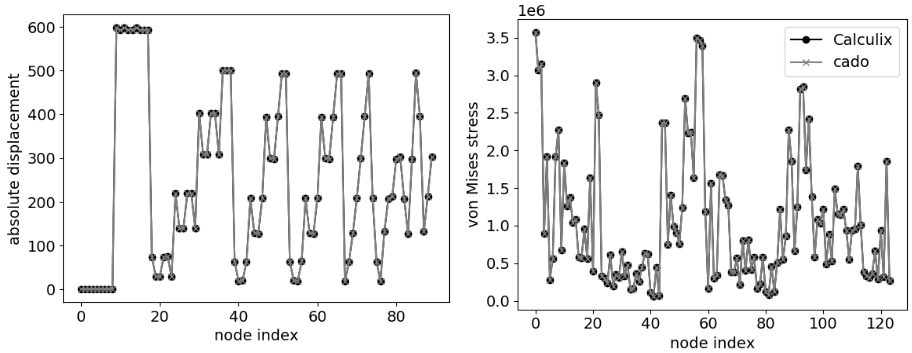


Figure 19: Mesh of axial fan blade test case with displacement vectors

4.3.1 Linear Stress Analysis Validation

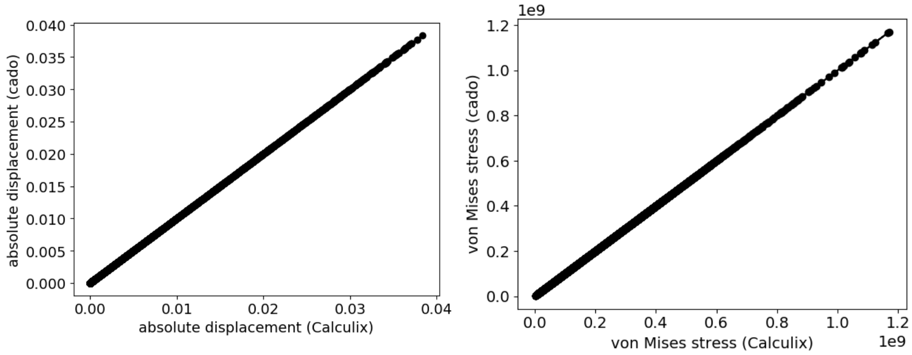
To validate the linear stress solver, a static linear stress analysis was performed for both the beam (fig. 18) and axial fan (fig. 19) test cases. The resulting displacements $\boldsymbol{u}$ and von Mises stresses σ_{VM} computed by both *cado* and *Calculix* were compared with each other. The results for the beam are shown in figure 20 and the comparison of the resulting axial fan displacements and von Mises stresses is shown in figure 21. Both test cases show an excellent agreement between the two solvers.



(a) Comparison of displacements $|\boldsymbol{u}|$

(b) Comparison of von Mises stress σ_{VM}

Figure 20: *Calculix* (black) and *cado* (gray) comparison using beam test case

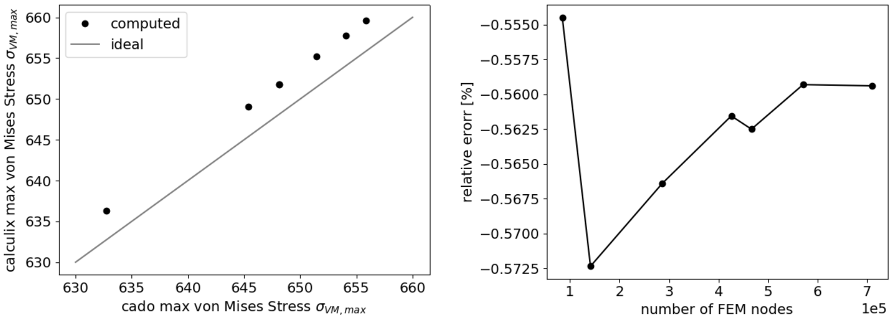


(a) Comparison of displacements $|\boldsymbol{u}|$

(b) Comparison of von Mises stress σ_{VM}

Figure 21: *Calculix* (x-axis) and *cado* (y-axis) comparison using axial fan test case

In figure 22a, a comparison of the maximum von Mises stress of the radial turbine test case used in the MDO applications is shown (figure 47, chapter 6). Tested with various numbers of FEM nodes, figure 22b shows that the relative error of cado's structural solver remains less than 1% when compared with the maximum von Mises stress calculated by Calculix. However, as is also exhibited in figure 22a, there is a slight negative bias of the maximum von Mises stress calculated by cado when compared to Calculix. This may be attributed to the use of different linear solver libraries which do not converge to exactly the same tolerance.

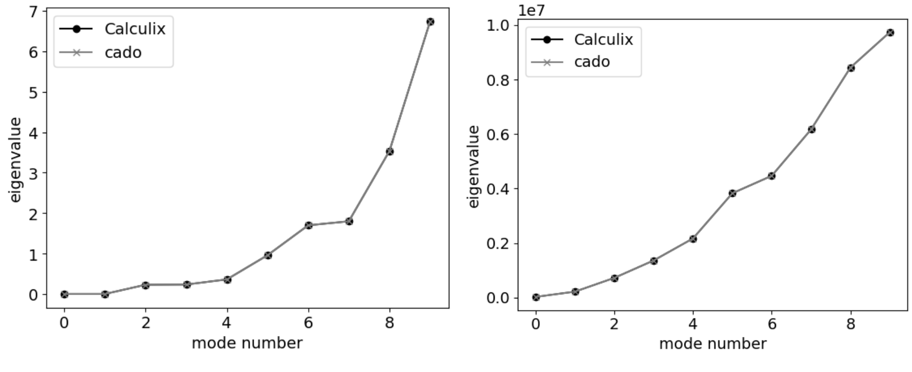


(a) *cado* vs *Calculix* values (b) Relative error

Figure 22: *Calculix* and *cado* maximum von Mises stress comparisons

4.3.2 Free Vibration Analysis Validation

To validate the vibration solver implemented in *cado*, a free vibration analysis of both beam and axial fan test cases is performed, where the 10 lowest eigenvalues are to be computed. The resulting eigenvalues λ_k computed by both *cado* and *Calculix* were compared and are also in good agreement with one another, as shown in figure 23a for the beam test case and figure 23b for the axial fan test case.



(a) Comparison using beam test case (b) Comparison using axial fan test case

Figure 23: *Calculix* (black) and *cado* (gray) comparison of eigenvalues λ_k

5 Efficient and Maintainable Code Differentiation

This section introduces the application of Algorithmic Differentiation, the core method used in this thesis to differentiate the structural solver's source code to obtain the structural sensitivities required for the adjoint-based MDO framework. The introduction of AD encompasses the two differentiation methods of AD - tangent (forward) mode and adjoint (reverse) mode, as well as typical gradient approximations used to assess their accuracy. Then the application of AD on the structural solver is discussed, in particular the differentiation of the stress and vibration analyses, as well as the CAD kernel and mesh generation, i.e., deformation. Finally, the extensibility and maintainability of the differentiated code is demonstrated by augmenting the primal code with new features that are automatically differentiated due to polymorphism.

5.1 Algorithmic Differentiation

Algorithmic differentiation (AD), also known as *automatic differentiation* or *computational differentiation*, is a source code augmentation method for computing exact derivatives of coded algorithms [20]. One of the main ideas behind AD is that every source code, no matter how complex, is ultimately a long list of elementary unary or binary operations such as

$$\begin{aligned} w_1 &= x_0 \\ w_2 &= sin(w_1) \\ w_3 &= w_1 + w_2. \\ y_0 &= w_3 \end{aligned}$$

This can be denoted as

$$w_j = \phi_j(w_{i_0}, ..., w_{i_{n_p-1}})_{i \prec j}, \quad \texttt{for } j = n, ..., n + n_{int} + m - 1, \tag{83}$$

where n denotes the number of independent inputs, n_{int} the number of intermediate variables, and m the number of resulting outputs. The subscript $i \prec j$ denotes that the previously computed n_p variables w_i precede the variable w_j in the source code, where i is within the index set $\{0, 1, 2, ..., j-1\}$ and ϕ_j represents the elementary function that maps the variables w_i to w_j. For unary operations $n_p = 1$ and for binary operations $n_p = 2$. This notation of listing elementary operations is also known as single-assignment code (SAC), where the value of a variable w_j is assigned only once [49].

The basic task of AD can be summarized as interpreting a complex code as SAC and performing the known differentiations of each basic operation ϕ_j. This operation can either be performed manually by hand or by using AD tools which can be found on the community website `www.autodiff.org` [8]. Note that while this section discusses AD using SAC examples, the actual code does not need to be written in SAC - this is handled by most AD tools automatically.

This section will introduce two short examples to illustrate the usage of AD to compute first-order derivatives using the tangent, also called *forward*, mode and using the adjoint, also called *reverse*, mode. Afterwards, the two common AD tool branches will be discussed - source code transformation and operator overloading methods. Gradient approximation methods are also briefly discussed, as these will be used to assess the accuracy of the AD implementation in this thesis.

5.1.1 Tangent (Forward) Mode

For the computation of first-order derivatives, consider the following function F, that takes an input x and outputs a variable y:

$$y = F(x),\ x, y \in \mathbb{R} \tag{84}$$

The first-order derivative $\nabla F(x)$ is a linear mapping

$$\dot{x} \rightarrow \nabla F \cdot \dot{x},\ \nabla F : \mathbb{R} \rightarrow \mathbb{R}, \tag{85}$$

where the dot notation $\dot{x}$ is used to represent a first-order derivative with respect to an auxiliary variable s, such that

$$\dot{x} = \frac{dx}{ds}. \tag{86}$$

With this definition, applying the chain rule of differentiation on (84) delivers

$$\dot{y} = \frac{dy}{ds} = \frac{dF}{dx}\frac{dx}{ds} = \nabla F(x) \cdot \dot{x}. \tag{87}$$

This is also known as the tangent model of $F(x)$. Setting $s = x$ and thereby $\dot{x} = 1$ *seeds* the tangent model to compute the derivative with respect to x as

$$\dot{y} = \nabla F(x) \cdot 1. \tag{88}$$

Analogously, this procedure can be applied on an arbitrarily complex function F by deriving the tangent model of the single-assignment code (83) of the function F:

$$\dot{w}_j = \sum_{i \prec j} \frac{\partial \phi_j}{\partial w_i} \cdot \dot{w}_i \tag{89}$$

While this can be coded manually by hand, it is a time-intensive and error-prone procedure. AD tools algorithmically break down complex functions F into SAC and derive the tangent (also known as forward) model (89) by applying the chain rule algorithmically.

This is a one-dimesional example with $x \in \mathbb{R}$, which can easily be extended to higher dimensions of $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^m$, where the dot notation $\dot{x}$ now represents a first-order partial derivative with respect to an auxiliary variable s

$$\dot{\boldsymbol{x}} = \frac{\partial \boldsymbol{x}}{\partial s}. \tag{90}$$

The Jacobian now induces a linear mapping of

$$\dot{\boldsymbol{x}} \rightarrow \nabla F \cdot \dot{\boldsymbol{x}},\ \nabla F : \mathbb{R}^n \rightarrow \mathbb{R}^m, \tag{91}$$

where n would be, e.g., the number of design parameters and m would be the number of objectives, which is often reduced to $m = 1$. This is a common setup of an optimization problem, where n design parameters serve as inputs and a single-dimensional objective function, e.g., $y \in \mathbb{R}$, has to be minimized. In this case, the tangent model takes the form

$$\dot{y} = \nabla F(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}. \tag{92}$$

Seeding the tangent model (92) with the unit vector

$$\dot{\boldsymbol{x}} = \hat{\boldsymbol{\imath}} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \tag{93}$$

where the i-th entry is equal to 1, would compute the i-th derivative $\frac{\partial F}{\partial x_i}$. Thus, to compute the entire gradient $\nabla F(\boldsymbol{x})$, n evaluations of the tangent model (92) are required, each seeded with the respective unit vector to compute the i-th derivative. To summarize, the gradient computation using the tangent model requires a computational cost of the order $\mathcal{O}(n) \cdot \mathit{cost}(F)$, where n is number of input variables and $\mathit{cost}(F)$ is the computational cost of a single evaluation of F. Note that $\mathit{cost}(F)$ is not necessarily equal to $\mathit{cost}(\nabla F)$, but differs by a small constant factor such that $\mathit{cost}(\nabla F)$ has a computational complexity of $\mathcal{O}(1) \cdot \mathit{cost}(F)$. Thus, the gradient calculation using the tangent model has a complexity of $\mathcal{O}(n) \cdot \mathit{cost}(F)$.

As a short example, this procedure is visualized in the directed acyclic graph (DAG) in figure 24, where the simple calculation of a function

$$y = \sin((x_1 \cdot x_2)^2) \tag{94}$$

is considered. For the sake of brevity, the single assignments of the input variables to intermediate variables and intermediate variables to output variables is omitted. The function takes two inputs, $(x_1,\ x_2)$, and produces one output y. In SAC, this algorithm can be broken down into three elementary arithmetic operations

$$\begin{aligned} w_1 &= x_1 \cdot x_2 \\ w_2 &= w_1^2 \\ y &= \sin(w_2) \end{aligned} \tag{95}$$

and the corresponding intermediate variables $w_1,\ w_2$. A forward differentiated version of this SAC essentially starts with the seeds $\dot{x}_1, \dot{x}_2$ and is given as

$$\begin{aligned} \dot{w}_1 &= \dot{x}_1 \cdot x_2 + x_1 \cdot \dot{x}_2 \\ \dot{w}_2 &= 2w_1 \cdot \dot{w}_1 \\ \dot{y} &= \cos(w_2) \cdot \dot{w}_2. \end{aligned}$$

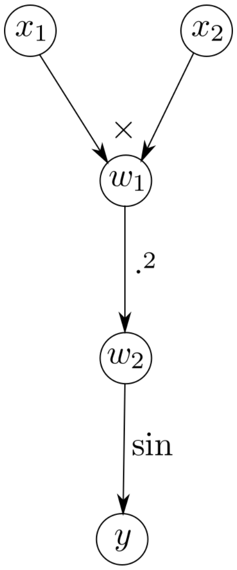


Figure 24: Directed Acyclic Graph of primal evaluation of $y = sin((x_1 \cdot x_2)^2)$ (95). Node labels represent primal variables. Edge labels represent arithmetic operations performed on the incoming data.

Seeding this procedure with $\dot{x}_1 = 1, \dot{x}_2 = 0$ would generate the gradient $\dot{y} = \frac{\partial y}{\partial x_1}$, while the seed $\dot{x}_1 = 0$, $\dot{x}_2 = 1$ produces the gradient $\dot{y} = \frac{\partial y}{\partial x_2}$. The full gradient $\frac{\partial y}{\partial \boldsymbol{x}}$ thus requires two evaluations of the tangent model. This is also visualized in figure 25, which shows that the inputs $\dot{x}_1$, $\dot{x}_2$ share the same path to the output, thus one evaluation for each input is required. In some cases, where inputs do not share the same computational path towards an output, these values can both be seeded and the respective gradients can be computed simultaneously.

5.1.2 Adjoint (Reverse) Mode

While the tangent model introduced in the previous section 5.1.1 enables the algorithmic calculation of gradients, the cost is dependent on the number of design parameters n. As a richer design space is explored and n increases, this method directly suffers from an increasing computational cost, resulting in prohibitively expensive gradient calculations. The adjoint method, which was discussed in section 3.1.3, offers a remedy to this situation, where the gradients can be evaluated at a computational cost proportional to the number of outputs, i.e., the dimensionality of the objective function. This is typically far less than the number of design parameters. Additionally, being able to compute gradients at a cost independent of the number of design parameters enables the exploration of richer design spaces to perform more complex optimizations. This section gives a background on how AD tools derive the required adjoint models.

The adjoint model of a function F with input variable x and output y (84)

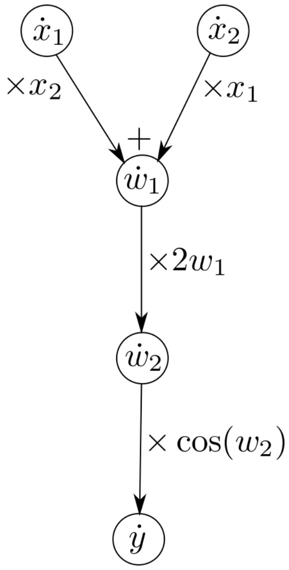


Figure 25: Directed Acyclic Graph of tangent evaluation of $y = sin((x_1 \cdot x_2)^2)$ (95). Node labels represent tangent variables. Edge labels represent arithmetic operations performed on the incoming data.

is given as

$$\bar{x} = \nabla F(x)^T \cdot \bar{y}, \tag{96}$$

where the bar notation $\bar{x}$ denotes adjoint variables. In contrast to the tangent variables, the *adjoints* $\bar{y}$ and $\bar{x}$ are defined as the partial derivatives of an auxiliary variable s with respect to y or x:

$$\bar{y} = \frac{\partial s}{\partial y}, \ \bar{x} = \frac{\partial s}{\partial x} \tag{97}$$

Equivalently, for higher dimensions

$$\bar{\boldsymbol{y}} = \frac{\partial s}{\partial \boldsymbol{y}}^T, \ \bar{\boldsymbol{x}} = \frac{\partial s}{\partial \boldsymbol{x}}^T. \tag{98}$$

With the chain rule, this results in

$$\bar{\boldsymbol{x}} = \frac{\partial s}{\partial \boldsymbol{x}}^T = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}^T \cdot \frac{\partial s}{\partial \boldsymbol{y}}^T = \nabla F(\boldsymbol{x})^T \cdot \bar{\boldsymbol{y}}. \tag{99}$$

Analogously to the tangent model, the adjoint model is *seeded* with $\bar{y} = 1$ to compute the gradient for the scalar case

$$\bar{x} = \nabla F(x)^T \cdot 1. \tag{100}$$

For higher dimensions where $\boldsymbol{x} \in \mathbb{R}^n$, the adjoint model takes the form

$$\bar{\boldsymbol{x}} = \nabla F(\boldsymbol{x})^T \cdot \bar{y}, \tag{101}$$

whereby a single run with $\bar{y} = 1$ would compute the entire gradient

$$\bar{\boldsymbol{x}} = \nabla F(\boldsymbol{x})^T \cdot 1. \tag{102}$$

AD tools interpret the function F in the form of single-assignment code to derive an adjoint model. In SAC representation, the adjoint model has the following form:

$$\bar{w}_i = \sum_{j:i \prec j} \frac{\partial \phi_j}{\partial w_i} \cdot \bar{w}_j \tag{103}$$

Note in (103) that the adjoint variable $\bar{w}_i$ consists of a sum of gradients of arithmetic operations ϕ_j, for all j that are preceded by i. This means that the adjoint of the SAC code is essentially a reverse accumulation of gradients, which is why this technique is often referred to as *reverse mode AD*. The reverse accumulation of gradients through the example DAG (fig 24), is shown in figure 26. The adjoint, or *reverse*, run of this order of operations, which is the application of (103), is given as

$$\begin{aligned}
\bar{w}_2 &= \frac{\partial y}{\partial w_2} \cdot \bar{y} = \cos(w_2) \cdot \bar{y} \\
\bar{w}_1 &= \frac{\partial w_2}{\partial w_1} \cdot \bar{w}_2 = 2w_1 \cdot \bar{w}_2 \\
\bar{x}_1 &= \frac{\partial w_1}{\partial x_1} \cdot \bar{w}_1 = x_2 \cdot \bar{w}_1 \\
\bar{x}_2 &= \frac{\partial w_1}{\partial x_2} \cdot \bar{w}_1 = x_1 \cdot \bar{w}_1.
\end{aligned} \tag{104}$$

Seeding this series of operations with $\bar{y} = 1$ would result in the derivatives

$$\bar{x}_1 = \frac{\partial y}{\partial x_1}, \; \bar{x}_2 = \frac{\partial y}{\partial x_2}. \tag{105}$$

Note that in the adjoint SAC (104), the intermediate variables w_1 and w_2 are required, which are first computed in a forward *primal* evaluation of (95).

To summarize, the evaluation of an adjoint model first requires an evaluation of the primal model, as well as certain intermediate variables. These are commonly referred to as the *forward* and *reverse* runs of an adjoint gradient evaluation. Additionally, the intermediate variables are stored in memory, which is referred to as the *tape*. A manually differentiated code requires extra care from the user to manage this storage, while AD tools take care of the tape automatically and, most likely, more efficiently. While this procedure works on complex codes in a semi-automatic black-box fashion, the resulting code may be inefficient. These issues are discussed further in sections 5.2, 5.3, and 5.4 for the particular application in this thesis.

5.1.3 Source Code Transformation Methods

Source code transformation tools, such as the widely used *Tapenade* [24], generate a new differentiated source code based on the primal source code supplied by the user. The original source code is not modified, but rather a new differentiated source code is generated, which can then be compiled and executed to compute derivatives.

The advantage with this method is that the differentiated source code can profit from compiler optimizations as any other piece of source code does, and

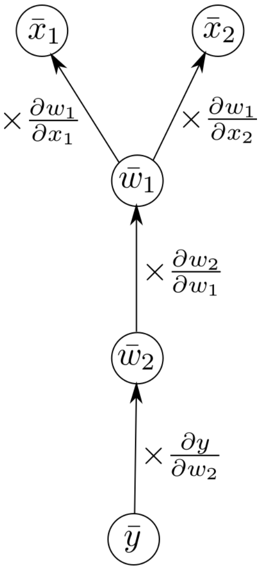

$\bar{x}_1$
$\bar{x}_2$
$\times \frac{\partial w_1}{\partial x_1}$
$\times \frac{\partial w_1}{\partial x_2}$
$\bar{w}_1$
$\times \frac{\partial w_2}{\partial w_1}$
$\bar{w}_2$
$\times \frac{\partial y}{\partial w_2}$
$\bar{y}$

Figure 26: Directed Acyclic Graph of adjoint evaluation of $y = sin((x_1 \cdot x_2)^2)$ (104). Node labels represent adjoint variables. Edge labels represent arithmetic operations performed on the incoming data.

it is applicable on the Fortran programming language, which is widely used in scientific computing applications. Additionally, the differentiated code can be scrutinized by an experienced user to double-check the differentiation and to perform manual code optimizatons. However, any change in the primal code requires an explicit re-differentiation to generate updated tangent or adjoint codes.

5.1.4 Operator Overloading Methods

On the other side of the spectrum are operator overloading (OO) tools such as *ADOL-C* [80], *dco/c++* [35], and *CoDiPack* [56]. These tools cater mostly to object-oriented languages such as C++. In this thesis, the tool *CoDiPack* is utilized, which offers low tape memory consumption and competitive tape evaluation performance due to the use of expression templates.

The OO AD tools are applied by using the AD tool-specific variable type, e.g. `codi::RealForward` and `codi::RealReverse`, instead of the `double` type. The general idea is that the AD types contain the gradient information in addition to the primal value, which can be set and retrieved using tool-specific functions as shown in algorithm 1. Once an input variable is registered, the execution of the program would store the required primal and gradient values of intermediate variables in memory commonly referred to as the *tape. Evaluating* the tape effectively performs the elementary adjoint operations, i.e., the *reverse* run, to compute the derivative of the registered input variable. The advantage of OO AD methods is that changes in the primal code are automatically reflected in the differentiated code and an explicit re-differentiation is usually not required.

```
// AD-type variable $x$ contains the primal and gradient values
$x$.PrimalValue;
$x$.GradientValue;

// This would set PrimalValue = 5
$x = 5$;

// This would return PrimalValue
$x$.GetValue();

// This would set GradientValue = 1
$x$.SetGradient(1);

// This would return GradientValue
$x$.GetGradient();
```

Algorithm 1: Basic data structure of AD type

For a basic usage example of forward AD with two inputs $x1$ and $x2$, see algorithm 2. Because we have two inputs, two evaluations have to be peformed, one for each variable. The i-th input variable is seeded with the fuction `x_i.setGradient(1)`. After computing the output variable `y`, calling the `getGradient()` function would return the gradient $\frac{\partial y}{\partial x_i}$. The seed gradient is then reset to zero using a call to `x_i.setGradient(0)`, ensuring we have a clean unit vector for the next input variable x_{i+1}. This is repeated for each input variable until the entire graident $\frac{\partial y}{\partial \boldsymbol{x}}$ has been computed.

For the reverse AD counterpart, the corresponding pseudocode is given in algorithm 3. The inputs are first registered using `tape.registerInput(x)` and the tape is activated to start recording operations. The variable `y` is then computed. Afterwards, the recording of the tape is stopped and the outputs are registered. The adjoint model is seeded with $\bar{y} = 1$ and the tape is evaluated, computing both gradients $\frac{\partial y}{\partial x1}$, $\frac{\partial y}{\partial x2}$ with a single evaluation. The gradients can be read directly from the input variables using `x_i.getGradient()`.

This implementation is greatly simplified if the source code makes use of a custom `typedef` for `double` types, e.g., `typedef double REAL`. Adjusting the code to use different AD types, e.g., for forward or reverse differentiation, is a matter of changing the declaration to `typedef AD_TYPE REAL`. For more experienced users, some AD tools, such as CoDiPack, have different AD types that offer trade-offs between memory consumption and runtime. For a more in-depth tutorial on the usage of CoDiPack, the interested reader is referred to the CoDiPack website `www.scicomp.uni-kl.de/codi`.

```
for i=0; i<2; i++ do
    // seed input variable $\dot{x_i} = 1$
    $x_i$.setGradient(1);
    $x1 = 3$;
    $x2 = 4$;

    // compute $y$
    $v1 = 5x2 + 7x1$;
    $v2 = v1 \cdot x2 + \cos(\frac{v1}{x1})$;
    $y = v1 \cdot \sqrt{v2}$;

    // returns gradient $\frac{\partial y}{\partial x_i}$
    $\frac{\partial y}{\partial x_i} = y$.getGradient();

    // seed is reset to zero to prepare for next loop iteration
    $x_i$.setGradient(0);
end
```

Algorithm 2: Basic usage of operator-overloading forward AD

```
// register $x1, x2$ as inputs and start recording tape
tape.registerInput($x1$);
tape.registerInput($x2$);
tape.setActive();
$x1 = 3$;
$x2 = 4$;

// evaluate function
$v1 = 5x2 + 7x1$;
$v2 = v1 \cdot x2 + \cos(\frac{v1}{x1})$;
$y = v1 \cdot \sqrt{v2}$;

// stop recording tape and register outputs
tape.setPassive();
tape.registerOutput($y$);

// seed adjoint model and evaluate tape (run adjoint model)
$y$.setGradient(1);
tape.evaluate();

// get gradients $\frac{\partial y}{\partial x1}, \frac{\partial y}{\partial x2}$
$\frac{\partial y}{\partial x1} = x1$.getGradient();
$\frac{\partial y}{\partial x2} = x2$.getGradient();
```

Algorithm 3: Basic usage of operator-overloading reverse AD

5.1.5 Finite Differences & Complex Step Method

To assess the accuracy of gradients computed by an AD-differentiated code, alternative gradient approximation methods are used as a comparison. These are typically the finite difference (FD) and complex step methods, which are briefly introduced in this section.

The finite difference approximation can be easily applied on code in a black-box fashion or on software that does not offer any source code access at all. It is the least intrusive, but also least accurate, method for approximating derivatives. The approximation error can be derived from the Taylor series expansion [11]. First-order gradient approximations, also called *forward difference*, are thus given as

$$\frac{\partial f}{\partial x} \approx \frac{f\left(x+\Delta\right) - f\left(x\right)}{\Delta} + \mathcal{O}\left(\Delta\right), \tag{106}$$

where Δ is the step size, and second order approximations, also known as *central difference*, as

$$\frac{\partial f}{\partial x} \approx \frac{f\left(x+\Delta\right) - f\left(x-\Delta\right)}{\Delta} + \mathcal{O}\left(\Delta^2\right). \tag{107}$$

Note that for second-order accurate gradient approximations, two function evaluations per input are required. The truncation error is dependent on the step size Δ, which is theoretically minimized as the step size decreases. However, due to computer round-off errors, and cancellation errors when applying the differencing operator on numbers that are nearly the same, the *condition error* of the FD method increases as the step size decreases [27].

Another method used for computing derivatives is the *complex step* method [42, 36], which is similar to forward AD mode. It boils down to the gradient approximation

$$\frac{\partial f}{\partial x} \approx \frac{\operatorname{Im}\left[f\left(x+i\Delta\right)\right]}{\Delta}, \tag{108}$$

which, as opposed to finite differences, is independent of a differencing operator. As a result, this eliminates the cancellation errors which plague the finite differences method when selecting a step size Δ. The approximation error is of the order $\mathcal{O}(\Delta^2)$ and the step size Δ can be set to extremely small values without any adverse effects [42]. To implement the complex step method, a `typedef` approach can also be used by using a `complex` type and evaluating the derivative according to (108) after performing a forward evaluation with the `complex` types. While this method is more accurate and efficient than finite differences, its overall cost is still dependent on the number of inputs.

5.2 Differentiating a Structural Sress Solver with AD

For structural stress constraints, the main quantity of interest is the maximum von Mises stress $\sigma_{max} \in \mathbb{R}$, which is typically required to be under a given threshold value σ_{req}, such that $\sigma_{max} < \sigma_{req}$. To include the structural constraints within an adjoint-based MDO framework, the gradient $\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}}$ is required, as shown in

$$\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}} = \frac{\partial \sigma_{max}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}. \tag{109}$$

The second term $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}$ is supplied by the differentiated mesh deformation as discussed in section 3.4.3. In this section, the computation of the first term, $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}}$, is discussed.

The maximum von Mises stress σ_{max} can be computed from the displacements $\boldsymbol{u}$ that result from a linear elastic problem (sec. 4.1.1). The finite element formulation [12] of the linear elastic equations is given as

$$K\boldsymbol{u} + M\ddot{\boldsymbol{u}} = \boldsymbol{b}, \tag{110}$$

where $\ddot{\boldsymbol{u}} = 0$ for a static linear stress analysis. This reduces the static linear stress analysis to a linear system of the form

$$K(\boldsymbol{x})\boldsymbol{u} = \boldsymbol{b}(\boldsymbol{x}), \tag{111}$$

where $K \in \mathbb{R}^{3m \times 3m}$ represents the stiffness matrix and $b \in \mathbb{R}^{3m}$ represents the load vector. This system can be solved for $\boldsymbol{u}$ using an iterative linear solver provided by, e.g., the Eigen [21] or PETSc [5] library.

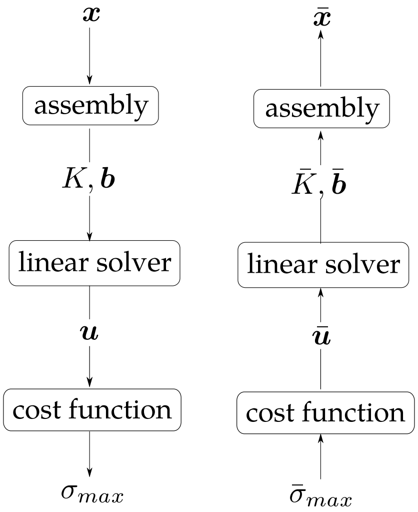


Figure 27: Flowchart of linear stress analysis with inputs and outputs.
Left: Primal problem. Right: Adjoint problem

In terms of *active* inputs and outputs, the mesh coordinates $\boldsymbol{x}$ serve as inputs to the linear system (111), which outputs $\boldsymbol{u}$ (fig. 27). This is then passed on as an input to the stress calculation to finally output the maximum von Mises stress σ_{max}. The inverse holds for the adjoint problem, where the seed $\bar{\sigma}_{max} = 1$ serves as the input and the final output is

$$\bar{\boldsymbol{x}} = \frac{\partial \boldsymbol{x}}{\partial \sigma_{max}} \in \mathbb{R}^{3m}. \tag{112}$$

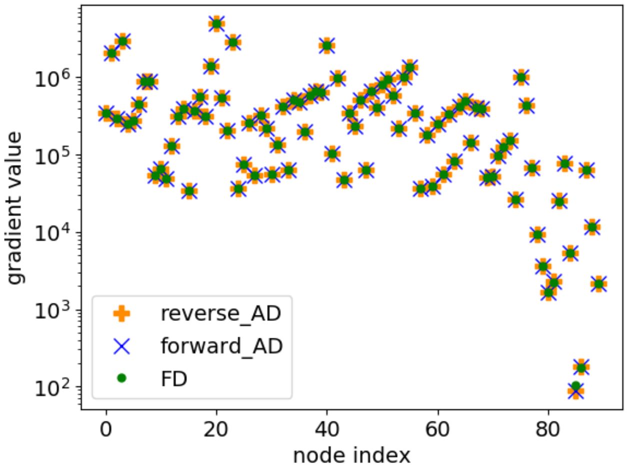


Figure 28: maximum von Mises stress sensitivities $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}}$ computed using reverse AD, forward AD, and FD

While a black-box AD differentiation of the assembly and stress calculation functions can be completed without a problem, differentiating through an iterative linear solver is generally not recommended. Not only would taping the iterative solver result in a high memory consumption, but the reverse run would use the same amount of iterations and the same Krylov basis vectors used by the primal solver. This could result in unconverged derivatives in the reversed iterative solver, which would propagate through the rest of the reverse run, resulting in inaccurate derivatives. To circumvent this problem, the adjoint of a linear system is typically reformulated into the well-known form [66] of

$$\bar{\boldsymbol{b}} = K^{-T}\bar{\boldsymbol{u}}, \quad \bar{K}_{i,j} = -u_j\bar{b}_i. \tag{113}$$

Thanks to the self-adjoint properties of the matrix K for linear elastic problems, this system can be further simplified to

$$K\bar{\boldsymbol{b}} = \bar{\boldsymbol{u}}, \quad \bar{K}_{i,j} = -u_j\bar{b}_i, \tag{114}$$

allowing one to recycle the same system matrix K as in the primal problem to solve the linear system for $\bar{\boldsymbol{b}}$ once $\bar{\boldsymbol{u}}$ has been computed from the reverse run of the stress calculation. $\bar{K}$ can then be directly evaluated. Finally, $\bar{\boldsymbol{b}}$ and $\bar{K}$ can be used to seed the reverse assembly run to compute $\bar{\boldsymbol{x}}$.

5.2.1 Gradient Comparison and Computational Cost

To assess the accuracy of the gradients calculated via the adjoint linear stress analysis, a cantilever beam test case with 90 degrees of freedom was used (figure 18). The sensitivities $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}}$ computed using reversed AD, forward AD, and finite differences (FD) were compared (figure 28). The comparison shows a good agreement among the computed gradients.

For the computational cost comparison, an axial fan blade test case with approximately 200,000 degrees of freedom was used (figure 19). The linear systems

	Primal P	Gradient Evaluation GE	Ratio GE/P
Peak Memory [GB]	0.85	24.55	28.88
Wall-clock time [s]	341.37	1296.09	3.79

Table 1: Memory and wall-clock run time comparison between primal linear stress analysis and a full gradient evaluation of $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}}$ using a forward and reverse run

are solved in parallel with 8 cores using Eigen's conjugate gradient solver, while the rest of the code is serial. In table 1, the performance results of an adjoint linear stress analysis are summarized. In the first column, the primal code has a wall-clock run time of approximately 5 minutes and 40 seconds and a peak memory consumption of 0.85 GB. For the gradient computation, a forward and reverse run is required, which amounts to two linear system solves. Compared to a primal run, a full gradient evaluation requires approximately $3.79 \cdot time(P)$ wall-clock time and a peak memory of approximately $28.88 \cdot memory(P)$. While the wall-clock time is within a competitive range, the memory consumption ratio is still relatively high and potentially infeasible for larger problems. The reduction of this memory consumption is detailed in section 5.4.

5.3 Differentiating a Structural Vibration Solver with AD

Besides structural stress constraints, vibration constraints are essential in turbomachinery design. In this section, the differentiation of a structural vibration solver using AD is introduced, which enables the calculation of gradients pertaining to vibration constraints.

A free vibration analysis departs from the linear elastic equations in unloaded form:

$$K\boldsymbol{u} + M\ddot{\boldsymbol{u}} = 0. \tag{115}$$

Using a proposed solution of

$$\boldsymbol{u} = \boldsymbol{u}_k e^{i\omega_k t}, \tag{116}$$

a generalized eigenvalue problem of the form

$$K\boldsymbol{u}_k = \omega_k^2 M\boldsymbol{u}_k \tag{117}$$

is obtained. Let $\lambda_k \in \mathbb{R}$ define the k-th eigenvalue with $\lambda_k := \omega_k^2$ and $\boldsymbol{u}_k \in \mathbb{R}^{3m}$ define the k-th eigenvector. Reformulated, the system can be written as

$$\left(K(\boldsymbol{x}) - \lambda_k M(\boldsymbol{x})\right)\boldsymbol{u}_k = 0, \tag{118}$$

where the same stiffness matrix K from the linear stress analysis (111) can be recycled. An additional matrix, the mass matrix $M \in \mathbb{R}^{3m\times 3m}$, needs to be computed prior to solving this system. Note that both matrices K and M are symmetric, such that

$$K = K^T, \quad M = M^T, \tag{119}$$

and the eigenvectors are mass-normalized by

$$\boldsymbol{u}_k^T M\boldsymbol{u}_k = 1. \tag{120}$$

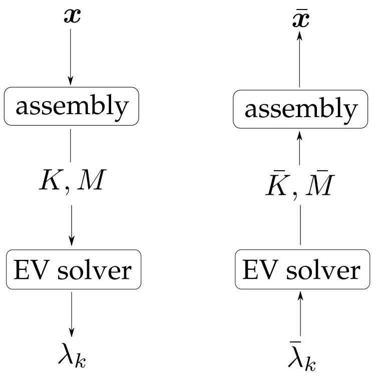


Figure 29: Flowchart of vibration analysis with inputs and outputs. Left: Primal problem. Right: Adjoint problem

The system (118) can be solved using an iterative generalized eigenvalue solver provided by, e.g., the SLEPc library [25]. Analogously to the linear stress analysis, the vibration analysis can be visualized in terms of inputs and outputs (figure 29), with the gradient of interest being the eigenvalue sensitivities with respect to the mesh nodes

$$\frac{\partial \lambda_k}{\partial \boldsymbol{x}} \in \mathbb{R}^{3m}. \tag{121}$$

The assembly phase can be AD differentiated in a black-box manner, but the iterative eigenvalue solver has to be treated separately. The tangent model of the vibration analysis will be introduced next, as this will be used to derive the adjoint model.

5.3.1 Tangent Model

To derive the tangent model of the vibration analysis, one begins by differentiating (118) by x_i:

$$\frac{\partial \left(K - \lambda_k M\right)}{\partial x_i} \boldsymbol{u}_k + \left(K - \lambda_k M\right) \frac{\partial \boldsymbol{u}_k}{\partial x_i} = 0 \tag{122}$$

$$\frac{\partial K}{\partial x_i} \boldsymbol{u}_k - \frac{\partial \lambda_k}{\partial x_i} M \boldsymbol{u}_k - \lambda_k \frac{\partial M}{\partial x_i} \boldsymbol{u}_k + \left(K - \lambda_k M\right) \frac{\partial \boldsymbol{u}_k}{\partial x_i} = 0 \tag{123}$$

This is followed by a left multiplication by $\boldsymbol{u}_k^T$:

$$\boldsymbol{u}_k^T \frac{\partial K}{\partial x_i} \boldsymbol{u}_k - \boldsymbol{u}_k^T \frac{\partial \lambda_k}{\partial x_i} M \boldsymbol{u}_k - \boldsymbol{u}_k^T \lambda_k \frac{\partial M}{\partial x_i} \boldsymbol{u}_k + \boldsymbol{u}_k^T \left(K - \lambda_k M\right) \frac{\partial \boldsymbol{u}_k}{\partial x_i} = 0 \tag{124}$$

Due to the definition of the eigenvalue problem (118) and the symmetry of matrices K and M (119), it follows that

$$\begin{aligned} \left(K - \lambda_k M\right) \boldsymbol{u}_k &= 0 \\ \left[\left(K - \lambda_k M\right) \boldsymbol{u}_k\right]^T &= 0^T \\ \boldsymbol{u}_k^T \left(K^T - \lambda_k M^T\right) &= 0 \\ \boldsymbol{u}_k^T \left(K - \lambda_k M\right) &= 0. \end{aligned} \tag{125}$$

With this, as well as the mass normalization (120), the system can be further simplified to

$$\boldsymbol{u}_k^T \frac{\partial K}{\partial x_i} \boldsymbol{u}_k - \frac{\partial \lambda_k}{\partial x_i} \underbrace{\boldsymbol{u}_k^T M \boldsymbol{u}_k}_{=1} - \boldsymbol{u}_k^T \lambda_k \frac{\partial M}{\partial x_i} \boldsymbol{u}_k + \underbrace{\boldsymbol{u}_k^T (K - \lambda_k M)}_{=0} \frac{\partial \boldsymbol{u}_k}{\partial x_i} = 0 \tag{126}$$

$$\boldsymbol{u}_k^T \frac{\partial K}{\partial x_i} \boldsymbol{u}_k - \frac{\partial \lambda_k}{\partial x_i} - \lambda_k \boldsymbol{u}_k^T \frac{\partial M}{\partial x_i} \boldsymbol{u}_k = 0. \tag{127}$$

Finally, this is rearranged to yield

$$\frac{\partial \lambda_k}{\partial x_i} = \boldsymbol{u}_k^T \left(\frac{\partial K}{\partial x_i} - \lambda_k \frac{\partial M}{\partial x_i} \right) \boldsymbol{u}_k \tag{128}$$

$$\dot{\lambda}_k = \boldsymbol{u}_k^T \left(\dot{K} - \lambda_k \dot{M} \right) \boldsymbol{u}_k \tag{129}$$

as the tangent model of (118). Given the right-hand side of (129) has been computed, this can be used to evaluate the sensitivities $\frac{\partial \lambda_k}{\partial x_i}$.

5.3.2 Adjoint Model

For generalized eigenvalue problems, previous work done by Lee [34] provides an adjoint formulation constructed using an augmented response function. The method leads to solving a linear system for the adjoint variables which are then used as inputs to a differentiated response function to compute the eigenvalue sensitivities (121). However, this formulation requires the gradients of the mass and stiffness matrices $\frac{\partial M}{\partial \boldsymbol{x}}$, $\frac{\partial K}{\partial \boldsymbol{x}}$, which may not be readily available. Dhondt et al. [13] used a symbolic approach to derive the adjoint model of the generalized eigenvalue problem, but used a perturbation approach to compute the mass and stiffness matrix gradients, which led to numerically unstable results.

In contrast, the method outlined in this thesis focuses on an adjoint formulation based on algorithmic differentiation, where the mass and stiffness matrix sensitivities are not required beforehand. Analogously to the iterative linear solver in the adjoint stress analysis, the iterative generalized eigenvalue solver of the vibration analysis has to be treated explicitly. For linear solvers, the differentiation technique is well understood [66], but has not yet been thoroughly explored for generalized eigenvalue solvers. The differentiation method is explained in this section.

The adjoint model for the vibration analysis can be viewed as

$$\bar{\boldsymbol{x}} = \frac{\partial \lambda_k}{\partial \boldsymbol{x}}^T \bar{\lambda}_k, \tag{130}$$

where $\bar{\lambda}_k = 1$ is seeded to compute the gradients of the k-th eigenvalue. Looking at the adjoint implementation in figure 29, the adjoint problem involves an adjoint generalized eigenvalue solver. To avoid the problematic black-box differentiation of an iterative eigenvalue solver, an adjoint model is derived to compute $\bar{K}, \bar{M}$, which are then passed on as inputs to the AD-differentiated assembly function.

Consider the generalized eigenvalue solver as a function

$$\lambda_k = \lambda_k (K, M), \tag{131}$$

which takes the matrices K and M as inputs and outputs the eigenvalue λ_k. The adjoint of K_{ij} is defined as

$$\begin{aligned}
\bar{K}_{ij} &= \frac{\partial s}{\partial K_{ij}} \\
\bar{K}_{ij} &= \frac{\partial s}{\partial \lambda_k} \frac{\partial \lambda_k}{\partial K_{ij}} \\
\bar{K}_{ij} &= \bar{\lambda}_k \frac{\partial \lambda_k}{\partial K_{ij}}.
\end{aligned} \tag{132}$$

To derive an equation for $\frac{\partial \lambda_k}{\partial K_{ij}}$, one starts by differentiating the eigenvalue problem (118) by K_{ij}:

$$\begin{aligned}
\frac{\partial}{\partial K_{ij}} \left[(K - \lambda_k M) \boldsymbol{u}_k \right] &= 0 \\
\left(\frac{\partial K}{\partial K_{ij}} - \frac{\partial \lambda_k}{\partial K_{ij}} M - \lambda_k \underbrace{\frac{\partial M}{\partial K_{ij}}}_{=0} \right) \boldsymbol{u}_k + (K - \lambda_k M) \frac{\partial \boldsymbol{u}_k}{\partial K_{ij}} &= 0 \\
\left(\frac{\partial K}{\partial K_{ij}} - \frac{\partial \lambda_k}{\partial K_{ij}} M \right) \boldsymbol{u}_k + (K - \lambda_k M) \frac{\partial \boldsymbol{u}_k}{\partial K_{ij}} &= 0
\end{aligned}$$

Left multiplying by $\boldsymbol{u}^T$ and using the relation (125) leads to

$$\begin{aligned}
\boldsymbol{u}_k^T \left(\frac{\partial K}{\partial K_{ij}} - \frac{\partial \lambda_k}{\partial K_{ij}} M \right) \boldsymbol{u}_k + \boldsymbol{u}_k^T (K - \lambda_k M) \frac{\partial \boldsymbol{u}_k}{\partial K_{ij}} &= 0 \\
\boldsymbol{u}_k^T \left(\frac{\partial K}{\partial K_{ij}} - \frac{\partial \lambda_k}{\partial K_{ij}} M \right) \boldsymbol{u}_k + \underbrace{\boldsymbol{u}_k^T (K - \lambda_k M)}_{=0} \frac{\partial \boldsymbol{u}_k}{\partial K_{ij}} &= 0 \\
\boldsymbol{u}_k^T \left(\frac{\partial K}{\partial K_{ij}} - \frac{\partial \lambda_k}{\partial K_{ij}} M \right) \boldsymbol{u}_k &= 0
\end{aligned}$$

Finally, the mass normalization (120) and the fact that $\frac{\partial K}{\partial K_{ij}} = \boldsymbol{J}_{ij}$ is a *single-entry matrix*, is used to further simplify to

$$\begin{aligned}
\boldsymbol{u}_k^T \left(\frac{\partial K}{\partial K_{ij}} - \frac{\partial \lambda_k}{\partial K_{ij}} M \right) \boldsymbol{u}_k &= 0 \\
\boldsymbol{u}_k^T \underbrace{\frac{\partial K}{\partial K_{ij}}}_{=\boldsymbol{J}_{ij}} \boldsymbol{u}_k - \frac{\partial \lambda_k}{\partial K_{ij}} \underbrace{\boldsymbol{u}_k^T M \boldsymbol{u}_k}_{=1} &= 0 \\
u_{k,i} u_{k,j} - \frac{\partial \lambda_k}{\partial K_{ij}} &= 0 \\
\implies \frac{\partial \lambda_k}{\partial K_{ij}} &= u_{k,i} u_{k,j}
\end{aligned} \tag{133}$$

The result of (133) is then plugged into (132) to give the adjoint of the stiffness matrix

$$\bar{K}_{ij} = \bar{\lambda}_k u_{k,i} u_{k,j}. \tag{134}$$

This results in a computational cost of one outer product to compute the entire matrix:

$$\bar{K} = \bar{\lambda}_k \left[\boldsymbol{u}_k \otimes \boldsymbol{u}_k\right] \tag{135}$$

The result (134) is now used to derive the adjoint of the mass matrix $\bar{M}$. It is known that the following dot product relationship [20, 49] between the tangent and adjoint models holds:

$$\left\langle \dot{K}, \bar{K} \right\rangle + \left\langle \dot{M}, \bar{M} \right\rangle = \left\langle \dot{\lambda}_k, \bar{\lambda}_k \right\rangle \tag{136}$$

The tangent model (129) is plugged into (136), leading to

$$\left\langle \dot{K}, \bar{K} \right\rangle + \left\langle \dot{M}, \bar{M} \right\rangle = \left\langle \boldsymbol{u}_k^T \left(\dot{K} - \lambda_k \dot{M} \right) \boldsymbol{u}_k, \bar{\lambda}_k \right\rangle \tag{137}$$

$$\left\langle \dot{K}, \bar{K} \right\rangle + \left\langle \dot{M}, \bar{M} \right\rangle = \left\langle \boldsymbol{u}_k^T \dot{K} \boldsymbol{u}_k, \bar{\lambda}_k \right\rangle + \left\langle -\lambda_k \boldsymbol{u}_k^T \dot{M} \boldsymbol{u}_k, \bar{\lambda}_k \right\rangle. \tag{138}$$

This can also be expressed in index notation as

$$\sum_{i,j} \dot{K}_{ij} \bar{K}_{ij} + \sum_{i,j} \dot{M}_{ij} \bar{M}_{ij} = \bar{\lambda}_k \sum_{i,j} \dot{K}_{ij} u_{k,i} u_{k,j} - \bar{\lambda}_k \lambda_k \sum_{i,j} \dot{M}_{ij} u_{k,i} u_{k,j}. \tag{139}$$

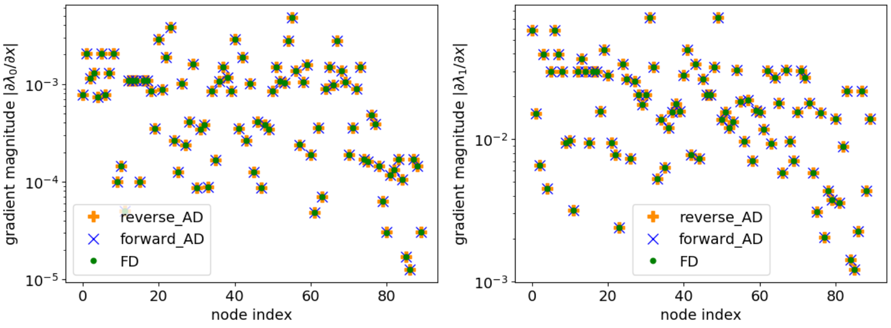
Plugging in the adjoint stiffness matrix (134) and simplifying leads to

$$\begin{aligned}
\bar{\lambda}_k \sum_{i,j} \dot{K}_{ij} u_{k,i} u_{k,j} + \sum_{i,j} \dot{M}_{ij} \bar{M}_{ij} &= \bar{\lambda}_k \sum_{i,j} \dot{K}_{ij} u_{k,i} u_{k,j} - \bar{\lambda}_k \lambda_k \sum_{i,j} \dot{M}_{ij} u_{k,i} u_{k,j} \\
\sum_{i,j} \dot{M}_{ij} \bar{M}_{ij} &= -\bar{\lambda}_k \lambda_k \sum_{i,j} \dot{M}_{ij} u_{k,i} u_{k,j} \\
\bar{M}_{ij} &= -\bar{\lambda}_k \lambda_k u_{k,i} u_{k,j} \\
\bar{M}_{ij} &= -\lambda_k \underbrace{\bar{\lambda}_k u_{k,i} u_{k,j}}_{=\bar{K}_{ij}},
\end{aligned}$$

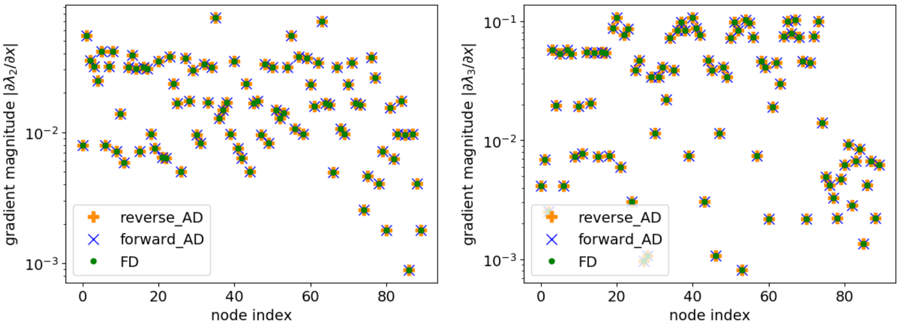
which gives the adjoint mass matrix as a function of the adjoint stiffness matrix and the k-th eigenvalue

$$\begin{aligned}
\bar{M}_{ij} &= -\lambda_k \bar{K}_{ij} \\
\bar{M} &= -\lambda_k \bar{K}.
\end{aligned} \tag{140}$$

Once the eigenvalue λ_k and eigenvector $\boldsymbol{u}_k$ have been computed, the adjoint stiffness matrix can be directly evaluated via equation (134) and a chosen seed $\bar{\lambda}_k$. Afterwards, the adjoint mass matrix can be directly computed using equation (140). The adjoint matrices $\bar{K}, \bar{M}$ can now be plugged into the reverse run of the AD-differentiated assembly function (figure 29) to finally arrive at the required gradient $\bar{\boldsymbol{x}} = \frac{\partial \lambda_k}{\partial \boldsymbol{x}}$.



(a) Gradient comparison of $|\frac{\partial\lambda_0}{\partial \boldsymbol{x}}|$ (b) Gradient comparison of $|\frac{\partial\lambda_1}{\partial \boldsymbol{x}}|$



(c) Gradient comparison of $|\frac{\partial\lambda_2}{\partial \boldsymbol{x}}|$ (d) Gradient comparison of $|\frac{\partial\lambda_3}{\partial \boldsymbol{x}}|$

Figure 30: Comparison of eigenvalue gradients $|\frac{\partial\lambda_k}{\partial \boldsymbol{x}}|$ of a rotating cantilever beam computed using finite differences, forward AD, and reverse AD for $k = 0, 1, 2, 3$

5.3.3 Gradient Comparison and Computational Cost

To assess the accuracy of the adjoint vibration analysis, the eigenvalue gradients $\frac{\partial\lambda_k}{\partial \boldsymbol{x}}$ of four modes are compared with the gradients computed using finite differences and forward AD (figure 30) using a rotating cantilever beam test case (figure 18). The figure shows an excellent agreement among the computed sensitivities.

The computational performance of the gradient evaluation (GE) using the adjoint is assessed by comparing the wall-clock time required versus the wall-clock time of a primal vibration analysis. Additionally, the peak memory consumption is considered, as this can often be relatively high for AD-differentiated codes. For the performance comparison, an axial fan blade test case (figure 31) is used with apprixmately 200,000 degrees of freedom and 10 eigenvalues are evaluated, such that $\boldsymbol{\lambda} \in \mathbb{R}^{10}$ and the entire gradient $\frac{\partial\lambda_k}{\partial \boldsymbol{x}} \in \mathbb{R}^{10\times 3m}$ is evaluated.

The axial fan blade test case is used for the computational performance measurements (figure 19). The results are summed up in table 2, where one can see that a full gradient evaluation for this test case uses up to 12.41 times as much memory as the primal vibration analysis does. The wall clock time for

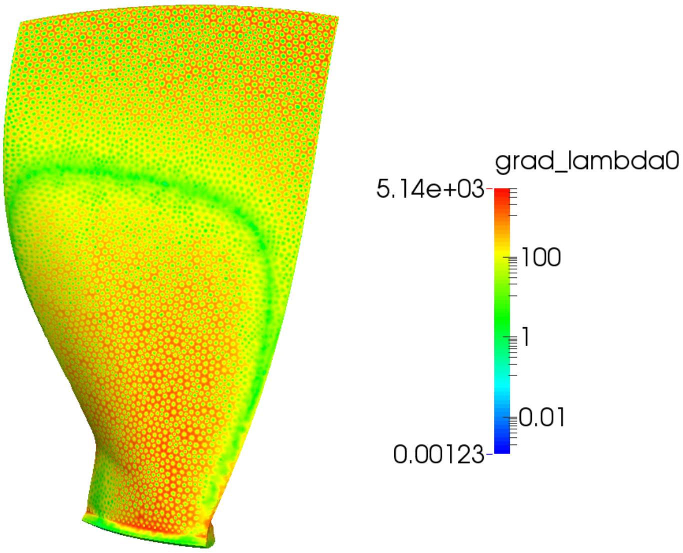


Figure 31: Eigenvalue sensitivities $\frac{\partial \lambda_0}{\partial \boldsymbol{x}}$ of an axial fan blade

	Primal P	Gradient Evaluation GE	Ratio GE/P
Peak memory [GB]	2.04	25.31	12.41
Wall-clock time [min]	156.28	171.25	1.09

Table 2: Peak memory and wall-clock run time comparison between primal vibration analysis and a full gradient evaluation of $\frac{\partial \boldsymbol{\lambda}}{\partial \boldsymbol{x}} \in \mathbb{R}^{10\times 3m}$ using a forward and reverse run, including adjoint stress analysis of an axial fan blade.

computing the entire gradient for all 10 eigenvalues is only 1.09 times as long as a primal vibration analysis. This is due to the fact that adjoint vibration analysis described in section 5.3.2 does not involve any additonal linear or eigenvalue problems to solve, only a direct evaluation of the adjoint mass and stiffness matrices $\bar{M}, \bar{K}$.

5.4 Reducing Assembly Memory Consumption Using Checkpointing

The assembly of the matrices K and M used in the linear stress and vibration analyses can be easily differentiated in a black-box fashion using AD. While the computed gradients are accurate and can be used for an optimization, the memory consumption can become considerably high for problems with a greater mesh node count. This is because the dimension of the matrices K and M is proportional to the number of mesh nodes m. Additionally, the matrices are assembled by looping over each element and computing local stiffness K_l and mass M_l matrices, which are then added into the global matrices K and M at each iteration (algorithm 4), by converting the local (element-wise) node indices to global node indices. A black-box differentiaton of this algorithm will cause the tape to grow proportional to the number of elements. To rectify this problem, a technique commonly known as *checkpointing* [20] is used.

```
Input  : mesh coordinates $\boldsymbol{x}$
Output: stiffness matrix $K$, mass matrix $M$
// Initialize global matrices to zero
$K$.SetZero();
$M$.SetZero();
foreach element $l$ do
    // Compute local matrices
    $K_l$ = ComputeLocalStiffnessMatrix($l$,$\boldsymbol{x}$);
    $M_l$ = ComputeLocalMassMatrix($l$,$\boldsymbol{x}$);
    // Add local matrices to global matrix
    AddLocalContribution($K_l$,$K$);
    AddLocalContribution($M_l$,$M$);
end
```

Algorithm 4: Pseudocode of assembly phase

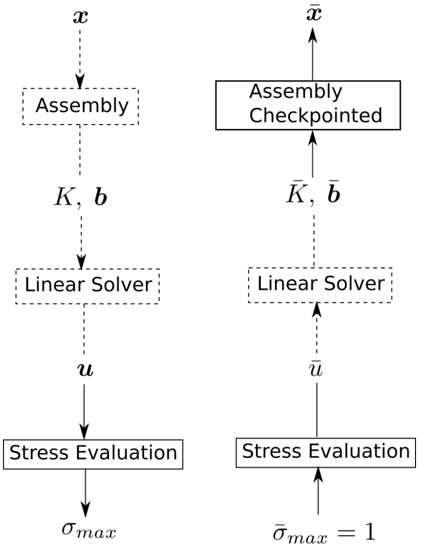


Figure 32: Flow chart of forward (left) and reverse (right) run of linear stress analysis using checkpointing in the assembly phase. Boxes with dashed lines represent that no taping is required.

The general idea behind checkpointing is to cut a section of code out of the taping procedure to save memory and to re-evaluate this portion of code during the reverse accumulation. The inputs necessary to perform the re-evaluation are typically stored in memory. The trade-off here is tape memory reduction in exchange for an increase in run-time due to the required re-evaluation. Because the assembly phase is computationally cheap compared to the linear system

```
Input  : $\bar{A}, \bar{M}$
Output: $\bar{\boldsymbol{x}}$
foreach element $l$ do
    // Start taping, store position
    start = tape.GetPosition();
    tape.SetActive();
    tape.RegisterInput($\boldsymbol{x}$);
    // Compute local matrices
    $A_l$ = ComputeLocalStiffnessMatrix($l$,$\boldsymbol{x}$);
    $M_l$ = ComputeLocalMassMatrix($l$,$\boldsymbol{x}$);
    // Add local matrices to global matrix
    AddLocalContribution($A_l$,$A$);
    AddLocalContribution($M_l$,$M$);
    // Register new matrix contributions as outputs and
       evaluate tape
    end = tape.GetPosition();
    tape.SetPassive();
    tape.RegisterOutput($A_l$,$M_l$);
    tape.Evaluate(start,end);
    // Add local adjoint contributions
    $\bar{\boldsymbol{x}}_l$ = tape.GetGradient();
    $\bar{\boldsymbol{x}}$ += $\bar{\boldsymbol{x}}_l$;
end
```

Algorithm 5: Pseudocode of element-level taping and evaluation during the reverse run of the assembly phase.

solve, the cost of re-running the assembly in exchange for the memory savings is warranted. Additionally, the inputs for the assembly routine do not need to be explicitly stored for the re-evaluation, as these do not change during the forward run.

The objective is to accumulate the adjoint contribution $\bar{\boldsymbol{x}}_l$ of each loop in algorithm 4 directly into the global $\bar{\boldsymbol{x}}$ element-wise. This is achieved by taping only the element-level code and performing local reverse runs for each element to obtain $\bar{\boldsymbol{x}}_l$ (algorithm 5), as opposed to taping the entire loop and performing a single reverse run to compute $\bar{\boldsymbol{x}}$ directly. As a result, in the forward run the assembly routine does not need to be taped (fig. 32). However, in the reverse run, a forward run of the assembly is repeated with the tape storage and evaluation limited to the element-level assembly routines (fig. 33). At this point in time the adjoint stiffness and mass matrices $\bar{K}, \bar{M}$ have already been computed according to the methods described in sections 5.2 and 5.3.2, which are required to seed the local reverse runs of the assembly phase.

Registering the entire matrix K, M and vector $\boldsymbol{b}$ as inputs for the local assembly would generate unnecessary memory and runtime overhead. Thus, a mapping of used matrix indices for each element

$$\texttt{indicesMap[element\_index]} = \texttt{vector(K\_row\_indices}, \texttt{K\_column\_indices)}$$

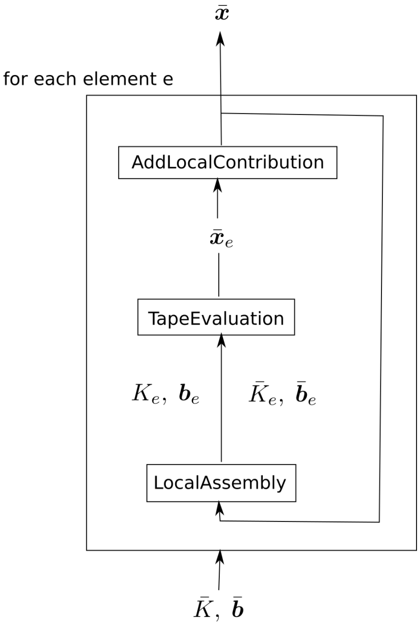


Figure 33: Flowchart of `Assembly Checkpointed` function. Element-local forward runs of the assembly are taped and directly evaluated. The gradients are then accumulated into $\bar{\boldsymbol{x}}$ at each iteration.

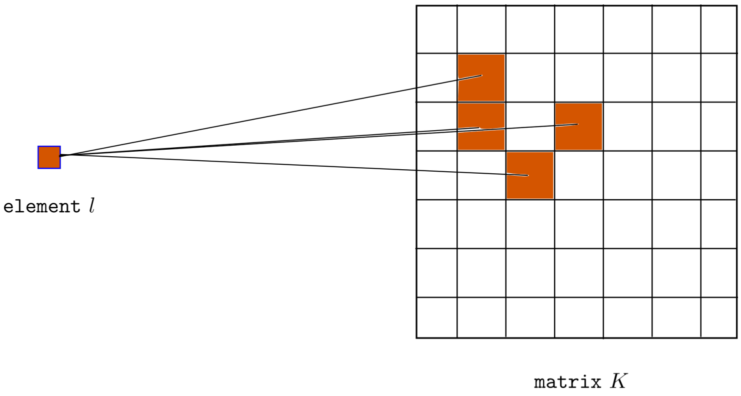
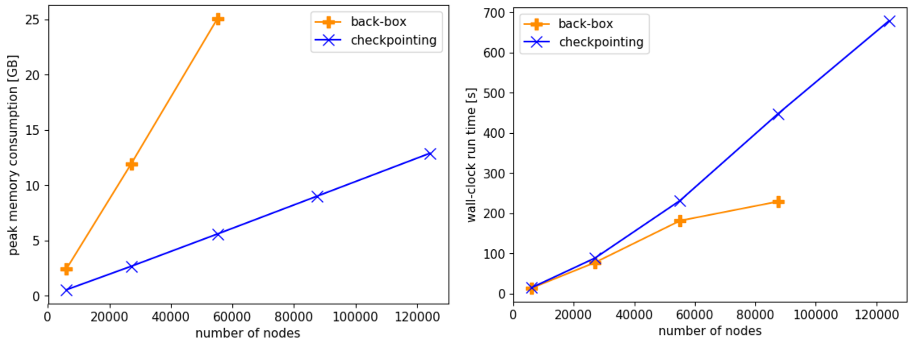


Figure 34: Illustrative example of index mapping of `element` l to indices within `matrix` K

is used to register the required matrix elements at each iteration (illustrated in figure 34) - analogously for vector indices. These indices are tracked and stored during the forward assembly phase. During the reverse run, at each element iteration, only the mapped matrix and vector indices are registered as inputs using `tape.registerInput()`. After the tape evaluation of each element-local assembly contribution, the gradients are accumulated into $\bar{\boldsymbol{x}}$, such that the entire gradient $\bar{\boldsymbol{x}} = \frac{\partial \sigma_{max}}{\partial \boldsymbol{x}}$ has been computed after the reverse assembly phase.



(a) Peak memory consumption [GB]

(b) Wall-clock run time [s]

Figure 35: Performance comparison between a black-box differentiation and a checkpointed differentiation of the matrix assembly

FEM Nodes	Black-box [GB]	With Checkpointing [GB]	Primal [GB]
5990	2.41	0.55	0.33
27097	11.94	2.68	0.98
55131	25.07	5.6	2.03
87527	Killed	9.02	3.23
124201	Killed	12.9	4.68

Table 3: Comparison of Peak Memory Consumption [GB] of Adjoint Linear Stress Analysis Using a Black-box Differentiated Assembly vs Using Checkpointing vs Primal Run

As a result, the size of the tape for the assembly is dependent on the number of operations performed per element, rather than dependent on the number of elements. Ultimately, this results in a drastic total memory reduction of approximately 77% (table 3) in exchange for an acceptable run-time cost increase of approximately 27% (table 4). Looking at the additional memory generated by the adjoint code with respect to the primal code, this has reduced by approximately 84%. The performance tests were made using a radial turbine test case.

A comparison of the peak memory consumption and wall-clock run time as a function of the number of elements is shown in figure 35. For the peak memory

FEM Nodes	Black-box [mm:ss]	With Checkpointing [mm:ss]
5990	0:13.58	0:14.53
27097	1:17.61	1:28.73
55131	3:01.37	3:51.5
87527	Killed	7:27.58
124201	Killed	11:18.70

Table 4: Comparison of Wall-clock Runtime [mm:ss] of Adjoint Linear Stress Analysis Using a Black-box Differentiated Assembly vs Using Checkpointing

	Primal P	Gradient Evaluation GE	Ratio GE/P
Peak Memory [GB]	4.68	12.9	2.76
Run-time [s]	215.97	678.7	3.14

Table 5: Performance comparisons between primal run and forward + reverse run of a radial turbine test case with 124,201 nodes

consumption, the black-box measurements show a much stronger dependence on the number of elements compared to the memory consumption when using checkpointing for the assembly (figure 35a). On the other hand, the trade-off is apparent in figure 35b, where a longer run time was measured when using the checkpointing technique. For meshes with more than 80,000 nodes, the simulations of the black-box differentiated adjoint were killed by the operating system due to the excessive memory usage. Ultimately, this leaves us with a run-time ratio of $3.14 \cdot time(P)$ and peak memory ratio of $2.76 \cdot memory(P)$ for a full gradient evaluation compared to a single primal run (table 5) when using the largest test case of 124,201 nodes.

5.5 Differentiating from CAD Kernel to Mesh with AD

The adjoint models of the CFD solver (section 3.5) and the CSM solver (sections 5.2, 5.3.2), are used to compute gradients with respect to the computational mesh ($\frac{\partial J}{\partial \boldsymbol{x}_{fluid}}$, $\frac{\partial J}{\partial \boldsymbol{x}_{solid}}$). However, the gradients of the computational mesh with respect to the CAD parameters are also required for both the structural and fluid meshes in order to compute the gradients required by the optimizer:

$$\frac{\partial J}{\partial \boldsymbol{\alpha}} = \frac{\partial \boldsymbol{x}_{fluid}}{\partial \boldsymbol{\alpha}} \frac{\partial J}{\partial \boldsymbol{x}_{fluid}} + \frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{\alpha}} \frac{\partial J}{\partial \boldsymbol{x}_{solid}} \tag{141}$$

This section discusses the use of forward AD to compute the fluid and solid mesh sensitivities $\frac{\partial \boldsymbol{x}_{fluid}}{\partial \boldsymbol{\alpha}}$ and $\frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{\alpha}}$, respectively.

5.5.1 Solid Mesh Sensitivities

For the solid side, the mesh deformation method described in section 3.4.3 is used, which boils down to three steps:

1. morph edge nodes
2. morph face nodes
3. morph inner nodes

This essentially boils down to a dependency of

$$\boldsymbol{x}_{solid} = \boldsymbol{x}_{solid}\left(\boldsymbol{x}_{face}\left(\boldsymbol{x}_{edge}\right)\right), \tag{142}$$

where the first two steps are used to compute the outer mesh nodes, $\boldsymbol{x}_{face}$ and $\boldsymbol{x}_{edge}$, and the last step computes the remaining mesh nodes $\boldsymbol{x}_{solid}$ using the

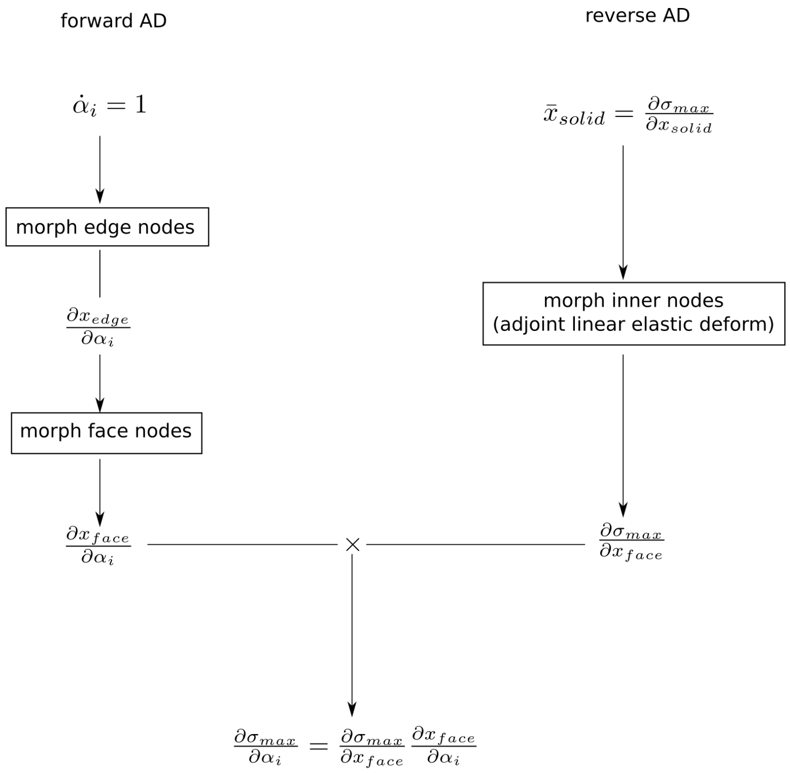

forward AD
reverse AD
$\dot{\alpha}_i = 1$
$\bar{x}_{solid} = \frac{\partial \sigma_{max}}{\partial x_{solid}}$
morph edge nodes
$\frac{\partial x_{edge}}{\partial \alpha_i}$
morph inner nodes (adjoint linear elastic deform)
morph face nodes
$\frac{\partial x_{face}}{\partial \alpha_i}$
×
$\frac{\partial \sigma_{max}}{\partial x_{face}}$
$\frac{\partial \sigma_{max}}{\partial \alpha_i} = \frac{\partial \sigma_{max}}{\partial x_{face}} \frac{\partial x_{face}}{\partial \alpha_i}$

Figure 36: Linking structural gradients to CAD. The left-hand side shows the forward-differentiation of the CAD-based mesh deform up to the outer mesh nodes $\frac{\partial x_{face}}{\partial \alpha_i}$. The right-hand side shows the reverse-differentiated inner mesh deformation based on the linear elastic analogy. Seeding the adjoint mesh deformation with the previously computed structural gradients $\frac{\partial \sigma_{max}}{\partial x_{solid}}$ propagates these gradients through to the CAD parameters.

linear elastic mesh deformation. This results in a chain rule of the form

$$\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}} = \frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}} \frac{\partial \boldsymbol{x}_{solid}}{\boldsymbol{\alpha}}$$

$$\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}} = \underbrace{\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}} \frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{x}_{face}}}_{\text{reverse AD}} \underbrace{\frac{\partial \boldsymbol{x}_{face}}{\partial \boldsymbol{x}_{edge}} \frac{\partial \boldsymbol{x}_{edge}}{\partial \boldsymbol{\alpha}}}_{\text{forward AD}},$$

whereby the first two terms are computed using the adjoint linear solver and the last two terms are computed using a black-box forward AD differentiation of the CAD-based mesh deformation code. This procedure is also illustrated in figure 36.

For the adjoint linear solver terms, one must take care that the inputs and outputs are properly defined. The adjoint linear stress analysis defines $\boldsymbol{x}_{solid}$ as inputs and σ_{max} as the output, while the mesh deformation defines $\boldsymbol{x}_{face}$ as the inputs and $\boldsymbol{x}_{solid}$ as the outputs. For the adjoint linear stress solver, it is clear that $\bar{\sigma}_{max} = 1$ and a single run is required. The adjoint mesh deformation,

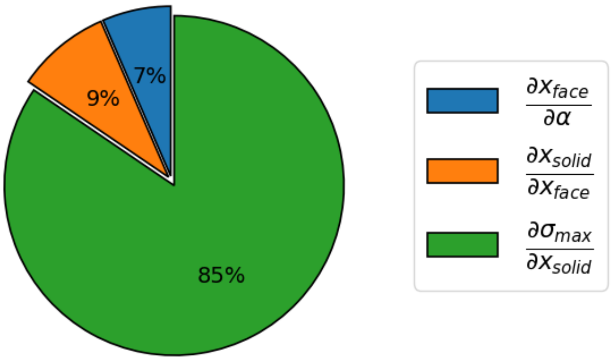


Figure 37: Total runtime breakdown for components of gradient evaluation $\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}} = \frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}} \frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{x}_{face}} \frac{\partial \boldsymbol{x}_{face}}{\partial \boldsymbol{\alpha}}$

however, has a high dimensional $\bar{\boldsymbol{x}}_{solid}$ seed input

$$\bar{\boldsymbol{x}}_{face} = \left(\frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{x}_{face}} \right)^T \bar{\boldsymbol{x}}_{solid}. \tag{143}$$

As discussed in section 5.1.2, the computational cost of computing a gradient via reverse AD is proportional to the number of outputs, i.e., in this case the dimension of $\boldsymbol{x}_{solid}$. However, because the gradient product $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}} \frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{x}_{face}}$ has to be calculated, rather than just the gradient $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}}$, the adjoint mesh deformation can be seeded with

$$\bar{\boldsymbol{x}}_{solid} = \frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}} \tag{144}$$

as an input, which would then directly compute the Jacobian-vector product $\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}} \frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{x}_{face}}$ with a single evaluation.

As a test case, a radial turbine geometry (figure 47) based on 24 CAD parameters $\boldsymbol{\alpha}$ was used. The first two steps of the mesh deformation are differentiated using forward AD and can be evaluated efficiently, costing approximately 9 seconds per CAD parameter. Furthermore, the forward AD evaluations can also be parallelized. With 24 CAD parameters and 8 CPU cores for parallelizations, time measurements have shown a runtime of 34.8 seconds for computing the gradients of the first two steps and approximately 47.8 seconds for the last step, the linear elastic deformation. Compared to the calculation of the stress gradients using the adjoint CSM solver, the mesh gradient calculation is significantly faster. A breakdown of the gradient calculation steps on the solid side is shown in figure 37.

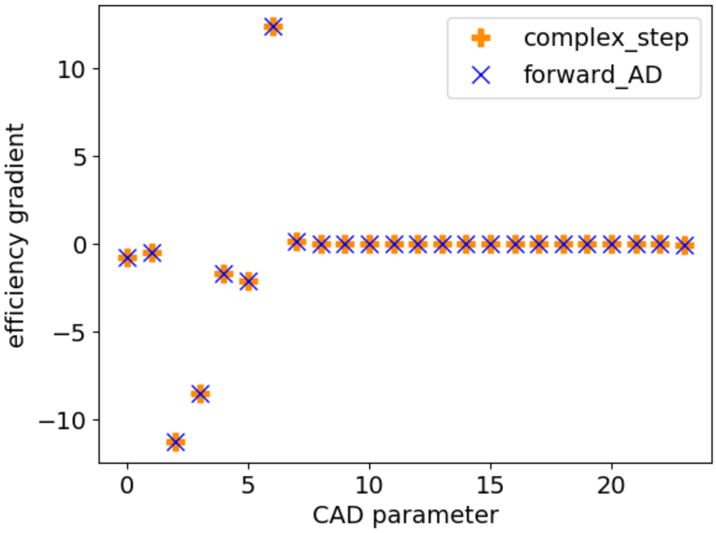


Figure 38: Accuracy comparison of gradient $\frac{\partial \eta_{TS}}{\partial \boldsymbol{\alpha}}$ between complex step (orange) and forward AD (blue) differentiation

5.5.2 Fluid Mesh Sensitivities

The fluid mesh sensitivities $\frac{\partial \boldsymbol{x}_{fluid}}{\partial \boldsymbol{\alpha}}$ were originally calculated using the complex step method. In this work, the complex step differentiation was replaced by a forward AD differentiation with CoDiPack, which encompasses the CAD kernel and structured fluid mesh generation. This is achieved by replacing the `complex` type by the forward AD type `codi::Real Forward` and extracting the sensitivities from each mesh node using the `getGradient()` function.

To assess the accuracy and performance of the forward AD differentiated CAD kernel and fluid mesh generation, a comparison is made with a complex step differentiation (section 5.1.5). As a test case, the CAD-based fluid mesh generation of a radial turbine was used, using 24 CAD parameters $\boldsymbol{\alpha}$ and approximately 750,000 fluid mesh points.

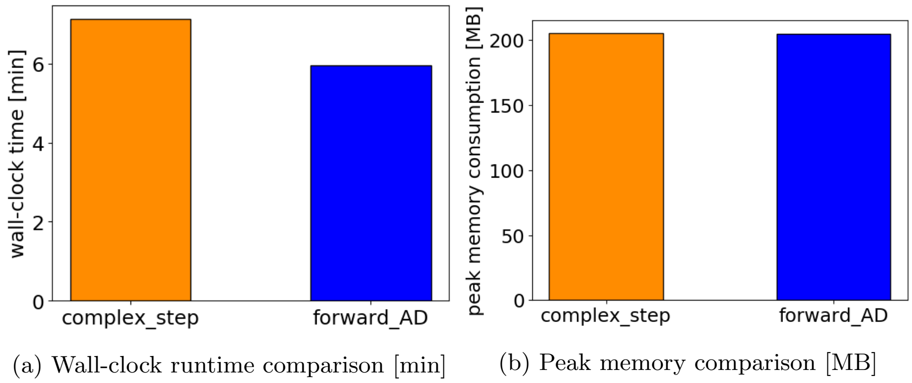
To compare the accuracy, the gradient of the total-to-static efficiency η_{TS} with respect to the CAD parameters $\boldsymbol{\alpha}$ is computed, whereby the full gradient is calculated via

$$\frac{\partial \eta_{TS}}{\partial \boldsymbol{\alpha}} = \frac{\partial \eta_{TS}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}. \tag{145}$$

The first component $\frac{\partial \eta_{TS}}{\partial \boldsymbol{x}}$ is provided by the adjoint CFD solver, while the second term $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}$ is provided by the forward differentiated CAD kernel and mesh generation.

.

For both the complex step and forward AD methods, this requires one evaluation per CAD parameter, i.e., 24 evaluations. The accuracy comparison (figure 38) shows a good agreement between the calculated gradients and both meth-



(a) Wall-clock runtime comparison [min] (b) Peak memory comparison [MB]

Figure 39: Performance comparison for the forward differentiation of the CAD kernel and fluid mesh generation between complex step (orange) and forward AD (blue) differentiation

	Wall-clock time [min]	Peak working memory [MB]
complex step	7.23	205.34
forward AD	5.96	205.1

Table 6: Performance comparison between a single forward AD evaluation and a single complex step evaluation of the fluid CAD-based mesh generation

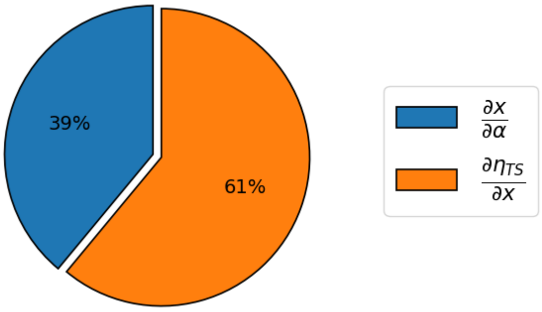


Figure 40: Total runtime breakdown for components of gradient evaluation $\frac{\partial \eta_{TS}}{\partial \boldsymbol{\alpha}} = \frac{\partial \eta_{TS}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}$

ods use approximately the same amount of peak working memory (figure 39b, table 6). The run time comparison for a single evaluation is shown in figure 39a, where a performance advantage using forward AD is evident. A single forward AD evaluation was measured to be 17.57% faster than a complex step differentiation. A full gradient calculation of $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}$ requires 24 evaluations. However, this can be easily parallelized, allowing an efficient gradient calculation. A runtime component breakdown is shown in figure 40, which shows that the parallel (8 cores) forward AD evaluations for $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}$ take up 39% of the total aerdoynamic gradient evaluation time. The remaining 61% of runtime are dedicated to the primal and adjoint CFD evaluations to compute the gradient $\frac{\partial \eta_{TS}}{\partial \boldsymbol{x}}$.

5.6 Reusability of Differentiated Framework - Design Space Extension

One of the main advantages of using the adjoint method for gradient computations is its cost independence of the design space. Not only does this allow one to use a rich design space for shape optimization, but it enables the extension of the design space to other possible design variables that not only influence the shape. Additionally, with an operator-overloading AD approach, an object oriented framework can be used to easily introduce primal code contributions without any additional AD implementations. In this section, the idea of extending the design variables from shape variables to include material variables with minimal code changes is introduced.

In previous sections, the gradients of quantities of interest such as the maximum von Mises stress σ_{max} or the eigenvalues λ_k were computed with respect to the unstructured mesh nodes $\boldsymbol{x}$. These mesh sensitivities would then be multiplied with the CAD kernel sensitivities $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\alpha}}$ to compute gradients with respect to CAD parameters $\boldsymbol{\alpha}$, enabling a CAD-based shape optimization. It seems like it is the default behavior of the adjoint CSM solver to compute gradients with respect to the mesh nodes $\boldsymbol{x}$, but actually the mesh nodes are registered as inputs using the AD tool. This is done by calling the `register_input(x)` function of CoDiPack, whereby `x` is an input variable (algorithm 5). In order to compute gradients with respect to additional input variables, one would need to register additional input variables in the same manner.

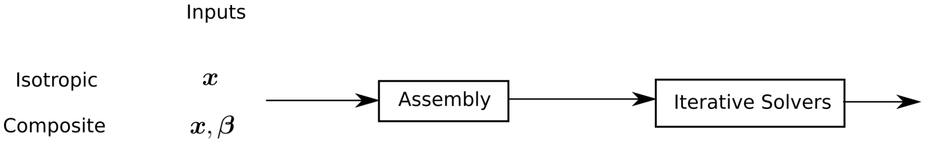


Figure 41: Isotropic elements are dependent on the shape $\boldsymbol{x}$. Composite elements are dependent on both the shape $\boldsymbol{x}$ and the material properties $\boldsymbol{\beta}$.

For instance, to introduce material design parameters $\boldsymbol{\beta}$, the design space could be extended as $(\boldsymbol{\alpha}\ \boldsymbol{\beta})^T$. The parameters $\boldsymbol{\beta}$ could, e.g., describe the lamination parameters that are used in the definition of composite materials [6, 22]. To include the computation of gradients with respect to $\boldsymbol{\beta}$, the additional

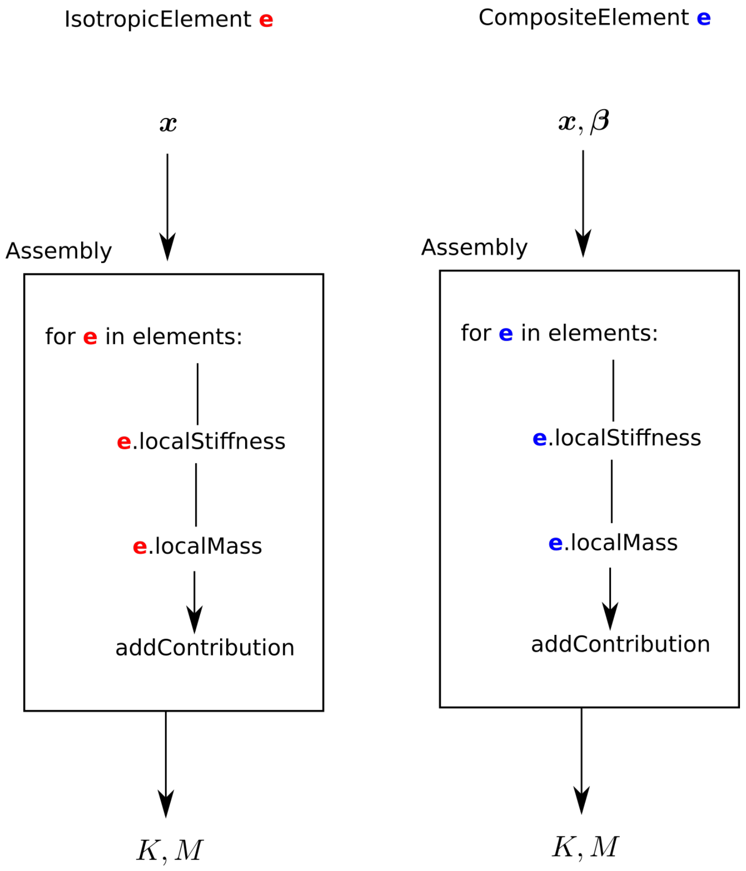


Figure 42: Use of polymorphism in assembly function. `IsotropicElement` (red) and `CompositeElement` (blue) both inherent from `Element` class. Using the `CompositeElement` class instead of the `IsotropicElement` class results in different local matrix calculations. Composite elements result in an additional input $\boldsymbol{\beta}$.

variables simply need to be registered as inputs using the `register_input(x)` function, resulting in gradients with respect to $\boldsymbol{x}$ and $\boldsymbol{\beta}$ when using the adjoint CSM solver (figure 41).

Additionally, for composite material applications, the stiffness matrix K is computed differently and is dependent on both the mesh nodes $\boldsymbol{x}$ and material parameters $\boldsymbol{\beta}$, such that $K(\boldsymbol{x}, \boldsymbol{\beta})$. Because the assembly function has previously been differentiated and CoDiPack is an operator-overloading tool, no additional AD code has to be implemented to handle the composite material stiffness matrix. This is achieved by implementing composite material capabilities using an object-oriented design approach. The algorithm flowcharts as shown in figures 32 and 33 remain the same.

In algorithms 4 and 5, the `ComputeLocalStiffnessMatrix(l,x)` and `ComputeLocalMassMatrix(l,x)` functions are called on an element-level basis,

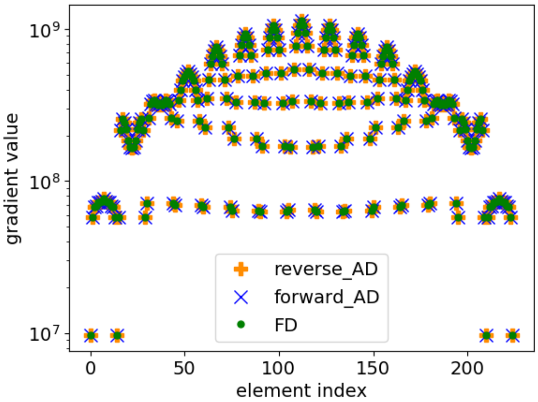


Figure 43: eigenvalue sensitivities $\frac{\partial \lambda_0}{\partial \beta_{10}}$ w.r.t. the lamination parameter β_{10}

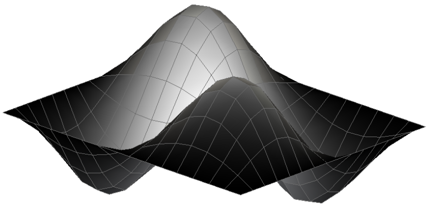


Figure 44: composite flat plate free vibration test case with perturbations of the fourth eigenmode

i.e., these local stiffness and mass matrix functions are members of the `element` class. A child class `CompositeElement` (fig. 13) is implemented with its own `ComputeLocalStiffnessMatrix(x)` function specifically for composite material applications. As a result, the AD tool will automatically differentiate through the local stiffness matrix computation implemented for composite materials as shown in figure 42.

To assess the accuracy of this approach, the gradients computed using AD and FD are compared in figure 43 using a flat-plate free vibration test case (fig. 44). The results show a positive agreement between the AD computed (both in forward and reverse AD) gradients and the gradients computed using FD. The gradients with respect to material properties $\boldsymbol{\beta}$ could potentially be used to perform material optimizations simultaneously with shape optimizations.

Using an object-oriented design approach to structure an AD-differentiated framework promotes code reusability and maintainability. The core functions, such as the assembly, iterative solver, and postprocessing only have to be differentiated once using the parent classes. This allows developers to focus on further development of the primal code by implementing subclasses of the already differentiated parent classes, enabling gradient calculations of additional features with minimal code maintenance from an AD perspective.

5.7 Computational Costs of MDO Framework

To assess the overall computational costs of the MDO framework and its individual components, a single optimization step is measured. The results are visualized in figure 45, where the MDO chain is broken down into the different components of the required gradients to assemble $\frac{\partial J}{\partial \boldsymbol{\alpha}}$ as shown in equation (141).

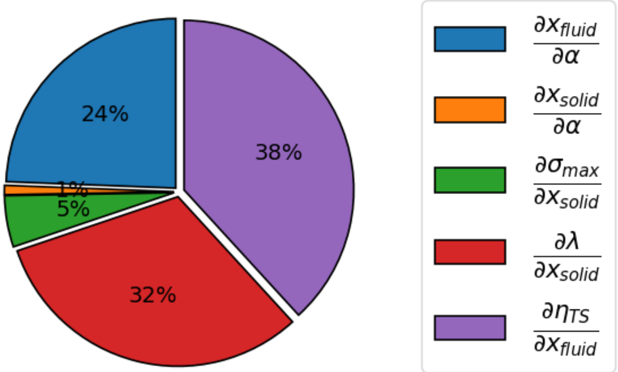


Figure 45: Run time breakdown including mesh generation and deformation, CFD, stress, and vibration calculations (primal + adjoint) with a total runtime of 153.38 minutes using 24 CAD parameters $\boldsymbol{\alpha}$, a CFD mesh of $\approx 750,000$ nodes and a CSM mesh of $\approx 90,000$ nodes.

6 Validation and Application of MDO Framework

Throughout the development of the adjoint structural solver and its integration into the MDO framework, several applications were tested with each achieved milestone. In this section, the different optimization applications of the developed work are introduced. First, the structural gradients with respect to CAD parameters calculated via the adjoint method are compared against finite difference approximations. The utility of the adjoint stress analysis is portrayed by performing an adjoint structural shape optimization of a radial turbine in (section 6.2). The adjoint stress analysis is then combined with the adjoint CFD solver to perform a MDO of a radial turbine (section 6.3). Finally, the adjoint vibration analysis is introduced to perform a MDO through optimizing efficiency, while adhering to mechanical stress and vibration constraints (section 6.4). All test cases use a CAD-based parametrization, where the CAD kernel and unstructured solid mesh are coupled via a mesh deformation method.

6.1 Gradient Comparison from CAD to Stress and Vibration Outputs

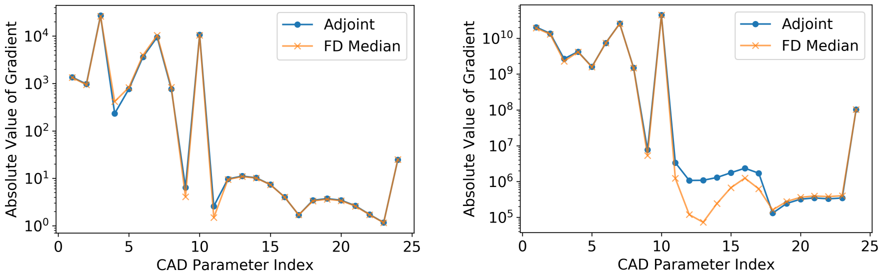
Within the previous chapter (ch. 5), the components required to compute structural gradients algorithmically were introduced and validated. One more validation of the whole chain, from CAD to maximum von Mises stress and eigenvalues, is warranted to demonstrate its applicability in a MDO framework. The adjoint gradients were calculated as described in section 5.5.1:

$$\begin{aligned}\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}} &= \underbrace{\frac{\partial \sigma_{max}}{\partial \boldsymbol{x}_{solid}} \frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{x}_{face}}}_{\text{reverse AD}} \underbrace{\frac{\partial \boldsymbol{x}_{face}}{\partial \boldsymbol{\alpha}}}_{\text{forward AD}} \\ \frac{\partial \lambda_i}{\partial \boldsymbol{\alpha}} &= \underbrace{\frac{\partial \lambda_i}{\partial \boldsymbol{x}_{solid}} \frac{\partial \boldsymbol{x}_{solid}}{\partial \boldsymbol{x}_{face}}}_{\text{reverse AD}} \underbrace{\frac{\partial \boldsymbol{x}_{face}}{\partial \boldsymbol{\alpha}}}_{\text{forward AD}} ,\end{aligned}$$

which is how they are computed in practice when using the differentiated solver. For comparison, second-order finite difference approximations of the same gradients were calculated using a range of step sizes $\Delta = 10^{-3}, 10^{-4}, ..., 10^{-7}$. The median result of the FD approximations is compared against the adjoint results as shown in figure 46, which generally show a good agreement with one other.

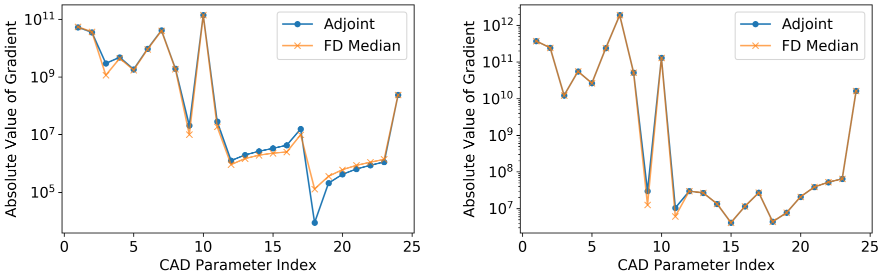
6.2 Structural Shape Optimization of a Radial Turbine

In this test case, a CAD-based shape optimization of a radial turbine is performed with the objective of minimizing the maximum von Mises stress σ_{max}. This application showcases that the sensitivities obtained from the adjoint structural solver can be used to successfully perform a structural shape optimization. Additionally, the CAD kernel, which defines the geometry, is coupled with the FEM mesh via a mesh deformation method as described in section 3.4.3.



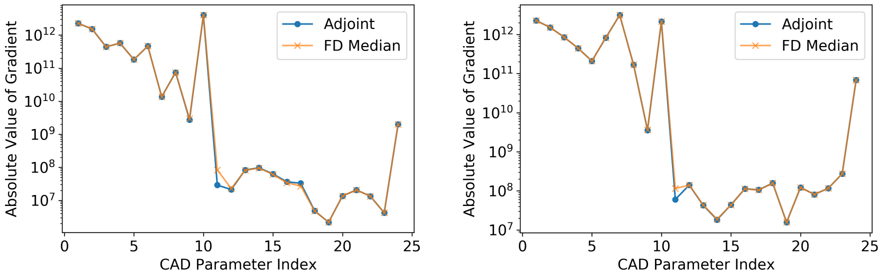
(a) Comparison of $|\frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}}|$

(b) Comparison of $|\frac{\partial \lambda_0}{\partial \boldsymbol{\alpha}}|$



(c) Comparison of $|\frac{\partial \lambda_1}{\partial \boldsymbol{\alpha}}|$

(d) Comparison of $|\frac{\partial \lambda_2}{\partial \boldsymbol{\alpha}}|$



(e) Comparison of $|\frac{\partial \lambda_3}{\partial \boldsymbol{\alpha}}|$

(f) Comparison of $|\frac{\partial \lambda_4}{\partial \boldsymbol{\alpha}}|$

Figure 46: Comparison of structural gradients with respect to CAD parameters $\boldsymbol{\alpha}$ calculated using the adjoint implementation and black-box finite differences.

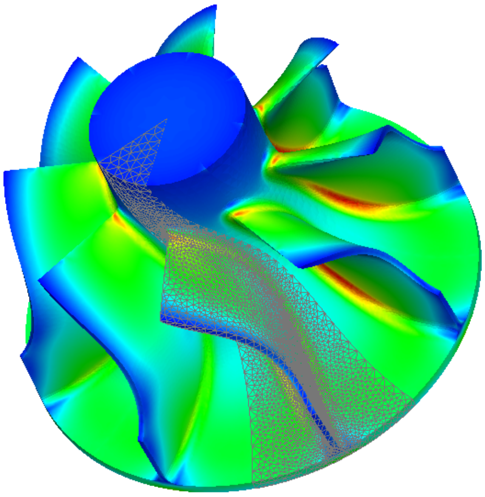


Figure 47: Unstructured mesh of radial turbine with von Mises stress contours

6.2.1 Numerical Setup

The unstructured mesh generated from the CAD geometry is discretized using quadratic tetrahedral elements, containing a total of approximately 85,000 nodes (figure 47). The objective of the optimization is to minimize the maximum von Mises stress σ_{max}. To ensure continuity of the objective, the maximum stress is approximated using a p-norm of the form

$$\sigma_{max} = \left(\sum_{i=0}^{m-1} \sigma_i^p \right)^{\frac{1}{p}} , \tag{146}$$

instead of using the discontinuous `max` function. A total of 7 CAD parameters $\boldsymbol{\alpha}$ are used in this test case. For the optimization, a steepest descent algorithm of the form

$$\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i - \Delta \frac{\partial \sigma_{max}}{\partial \boldsymbol{\alpha}_i} \tag{147}$$

is used with a constant step size of $\Delta = 10^{-8}$.

6.2.2 Optimization Results

The initial geometry is shown in figure 49 with contours of the von Mises stress σ distribution. Especially from the side view in figure 49b, the high stress area is observed to be at the fillet region connecting the blade to the hub. By extending the back plate near near the radial center, the optimizer is able to significantly reduce the overall stresses as shown in figure 50. A total reduction of 9.88% in the maximum von Mises stress σ_{max} is achieved within 49 iterations. The convergence history of the optimization is shown in figure 48. At iteration 28, when the objective was no longer significantly updated, a new structural mesh was generated, causing a kink in the convergence plot (recall the unstructured mesh generation vs deformation discussion in section 3.4.3). Aside from this iteration, all other design updates used a mesh deformation to update the mesh.

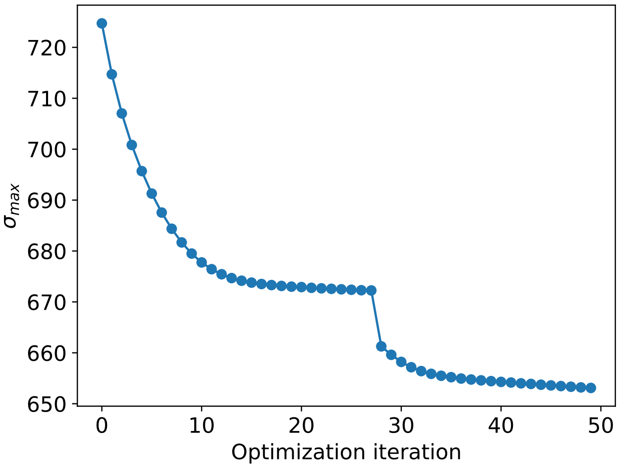
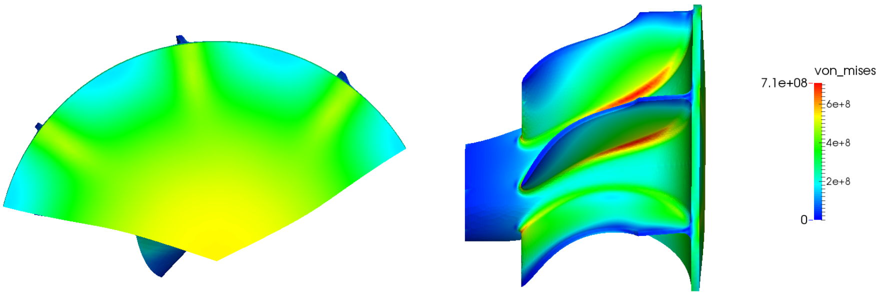


Figure 48: Convergence history of radial turbine structural optimization



(a) Bottom view

(b) Side view

Figure 49: Baseline geometry of radial turbine with von Mises stress σ distribution

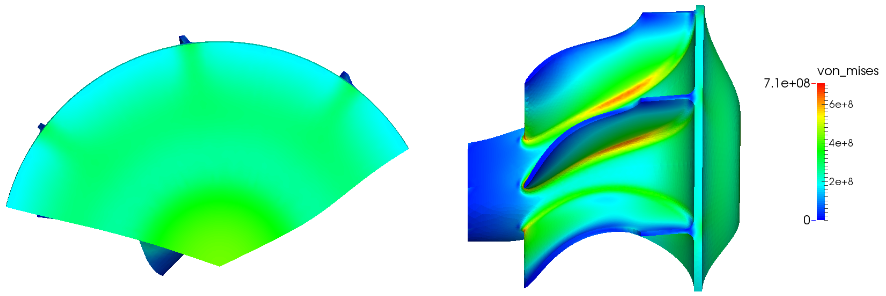
A total runtime of approximately 535 minutes was required to complete this optimization.

6.3 Aerodynamic Shape Optimization of a Radial Turbine Under Stress Constraints

After performing a structural CAD-based optimization to verify the utility of the adjoint structural solver (section 6.2), the structural and fluid disciplines are combined to perform an adjoint multidisciplinary optimization. The goal is to maximize the total-to-static efficiency η_{TS} while keeping the maximum von Mises stress below a defined threshold $\sigma_{max} < \sigma_{req}$.

6.3.1 Numerical Setup

A radial turbine geometry with 24 CAD parameters $\boldsymbol{\alpha}$ is used for this test case. The quantities of interest η_{TS} and σ_{max} from the fluid and solid disciplines are



(a) Bottom view (b) Side view

Figure 50: Optimized geometry of radial turbine with von Mises stress σ distribution

combined into a single objective function

$$J = (1 - \eta_{TS}) + \omega_\sigma \cdot Sig\left(\sigma_{max}\right) \cdot \left(\frac{\sigma_{max} - \sigma_{req}}{\sigma_{req}}\right)^2, \tag{148}$$

where the total-to-static effiency η_{TS} is defined as

$$\eta_{TS} = \frac{\frac{T_{0,2}}{T_{0,1}} - 1}{\left(\frac{p_2}{p_{0,1}}\right)^{\frac{\gamma-1}{\gamma}} - 1}. \tag{149}$$

The maximum von Mises stress σ_{max} is approximated using the p-norm (146) with $p = 50$. The sigmoid function

$$Sig(\sigma_{max}) = \frac{1}{1 + e^{-(\sigma_{max} - \sigma_{req})}} \tag{150}$$

is used to provide a continuous penalty term to the objective function, where ω_σ is the penalty weight. This sums up the optimization problem to minimizing J, which is solved using a steepest descent algorithm

$$\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i - \boldsymbol{\lambda} \circ \frac{\partial J}{\partial \boldsymbol{\alpha}}, \tag{151}$$

where the $\circ$ operator represents the Hadamard product

$$\boldsymbol{\lambda} \circ \frac{\partial J}{\partial \boldsymbol{\alpha}} = \left(\lambda_0 \cdot \frac{\partial J}{\partial \alpha_0}, \lambda_1 \cdot \frac{\partial J}{\partial \alpha_1}, \ldots, \lambda_{n-1} \cdot \frac{\partial J}{\partial \alpha_{n-1}}\right)^T. \tag{152}$$

Different fixed step sizes are defined in the vector $\boldsymbol{\lambda}$ due to the different orders of magnitude and units used in the CAD parameters $\boldsymbol{\alpha}$.

The optimization is performed in two steps. First, using a coarser CFD mesh of around 750,000 nodes to achieve an improvement at a cheaper computational cost within the fist 24 iterations. Starting at the 25th iteration, a finer CFD mesh with around 1.4 million nodes is used to go through the second half of the total optimization iterations.

nodes	iterations	runtime [days]	$\Delta\eta_{TS}$
$\approx$ 750k	24	3.5	2.77%
$\approx$ 1.4m	15	4	3.97%
$\approx$ 1.4m	10	2.5	4.03%

Table 7: Summary of optimization computational costs and efficiency improvements

	Baseline	Optimized	Δ [%]
Isentropic Efficiency (TS) [-]	0.7464	0.7765	4.0327
Isotropic Efficiency (TT) [-]	0.8516	0.8482	-0.3946
Power [kW]	1.2535	1.3854	10.5255
Max von Mises Stress [MPa]	681.801	662.159	-2.8809

Table 8: Aerodynamic and structural quantities of interest at baseline design and optimized design

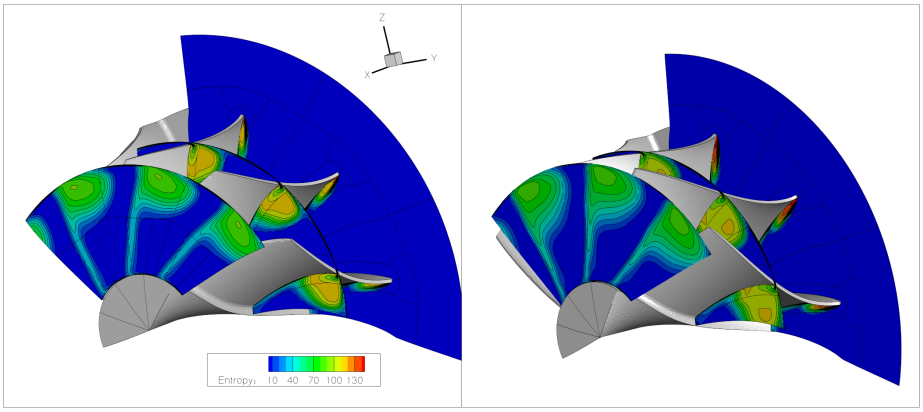
6.3.2 Optimization Results

Figure 52 shows a convergence history of the optimization. It can be seen that the optimizer is able to continuously reduce the objective J and increase the efficiency η_{TS}. The stress constraint is initially violated by the baseline design, but immediately corrected by the optimizer in the first iteration. As the optimization runs its course, the stress constraint $\sigma_{max} < \sigma_{req}$ is successfully obeyed. At iteration 25, a kink can be seen in the plots. This is due to the finer CFD mesh being used, resulting in a higher accuracy.

As shown in table 7, the first 24 iterations were performed within 3.5 days, resulting in an improvement in efficiency by 2.77%. With 15 more iterations using the finer mesh, a total efficiency improvement of 3.97% was achieved with an additional 4 days of runtime. The remaining iterations until iteration 49, using the finer mesh, took an additional 2.5 days to complete. The optimization was performed on an 8-core Intel i7-4790K machine. The primal and adjoint CFD runs were performed with 6 cores for optimal load-balancing. The forward differentiated CFD mesh generation and the adjoint CSM solver utilized all 8 cores in parallel.

Overall, this led to an efficiency improvement of 4.03% and a power improvement of 10.53% (table 8). As a result, the blades of the turbine are more heavily loaded and an increase in entropy generation at the leading edge of the blades can be observed (figure 51). A mass flow boundary condition was used to ensure the inlet and outlet mass flows remain unchanged (table 9). As a result, the total inlet pressure increased by around 4.5%.

Figure 53 shows the von Mises stress distribution of the baseline and optimal geometries. The maximum von Mises stress was reduced by 2.88% to 662.16 MPa, slightly violating the threshold of $\sigma_{req} = 660MPa$ by approximately 0.33%. In the baseline geometry, the von Mises stresses are concentrated around the fillet area, where the blade and hub connect. During the optimization, the trailing edge of the blade is straightened, resulting in a distribution of stresses



(a) Baseline design

(b) Optimized design

Figure 51: Contours of entropy generation of the basline and optimized design

	Baseline	Optimized	Δ [%]
Inlet Mass Flow $[\frac{g}{s}]$	10.1506	10.1506	0
Outlet Mass Flow $[\frac{g}{s}]$	10.1496	10.1496	0
Total Inlet Pressure $[kPa]$	185.002	193.7433	4.5118
Static Outlet Pressure $[kPa]$	101.8099	102.357	0.5374
Total Inlet Temperature $[K]$	1050.0379	1050.0447	0.0006

Table 9: Aerodynamic quantities of interest at baseline design and optimized design

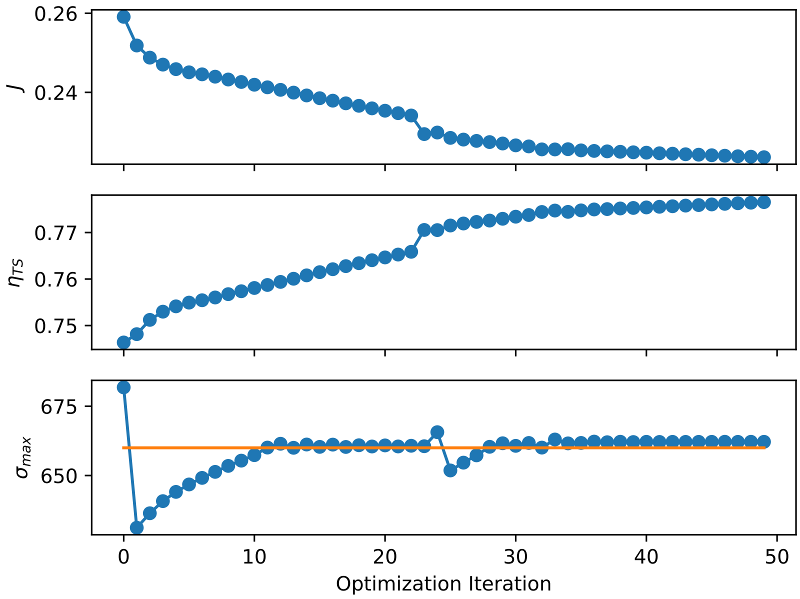
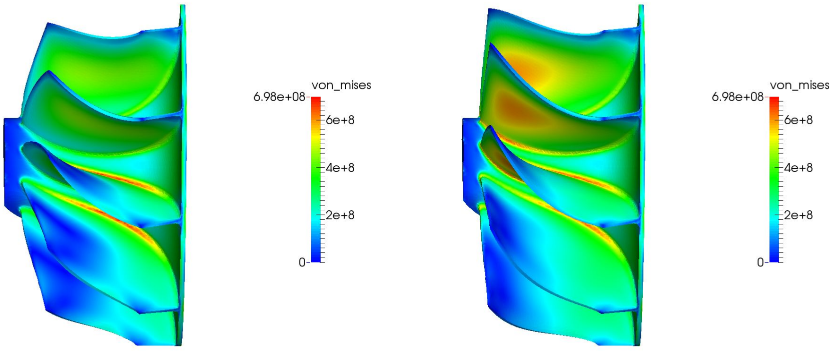
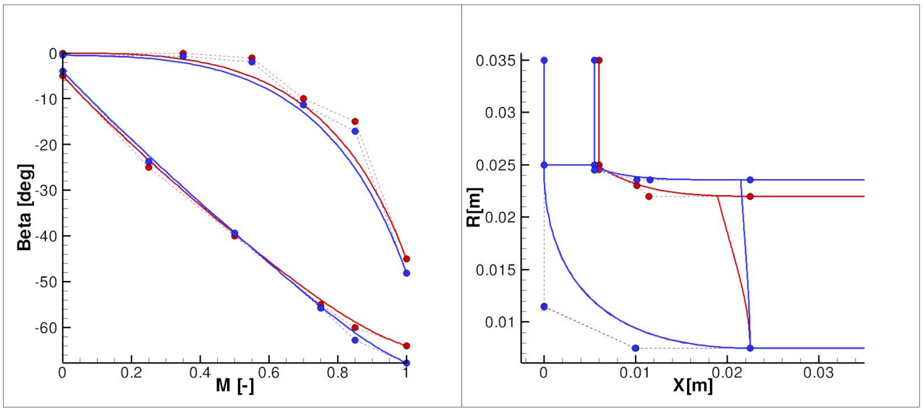


Figure 52: Plot of optimization history showing objective J, efficiency η_{TS}, and maximum von Mises stress σ_{max} over each optimization iteration. The orange line represents $\sigma_{req} = 660\,MPa$.



(a) Baseline geometry (b) Optimized geometry

Figure 53: Comparison of von Mises stress σ distributions for baseline and optimal designs



(a) Blade angle parametrization (b) Meridional parametrization

Figure 54: Comparison of CAD paremeters. Red: Baseline parameters. Blue: Optimized parameters

onto the blade. Figure 54 shows a comparison of CAD parameters from the baseline and optimal designs. In figure 54b, the CAD parameters along the shroud have moved considerably, leading to a straightening of the blade's trailing edge. As a result, the size of the outlet has increased and that of the inlet has slightly decreased.

6.4 Aerodynamic Shape Optimization of a Radial Turbine Under Stress and Vibration Constraints

Building on the adjoint-based MDO shown in section 6.3, the structural constraints can be further augmented. To achieve this, the adjoint free vibration analysis discussed in section 5.3 is used to compute eigenvalue sensitivities $\frac{\partial \lambda_0}{\partial \boldsymbol{x}}$.

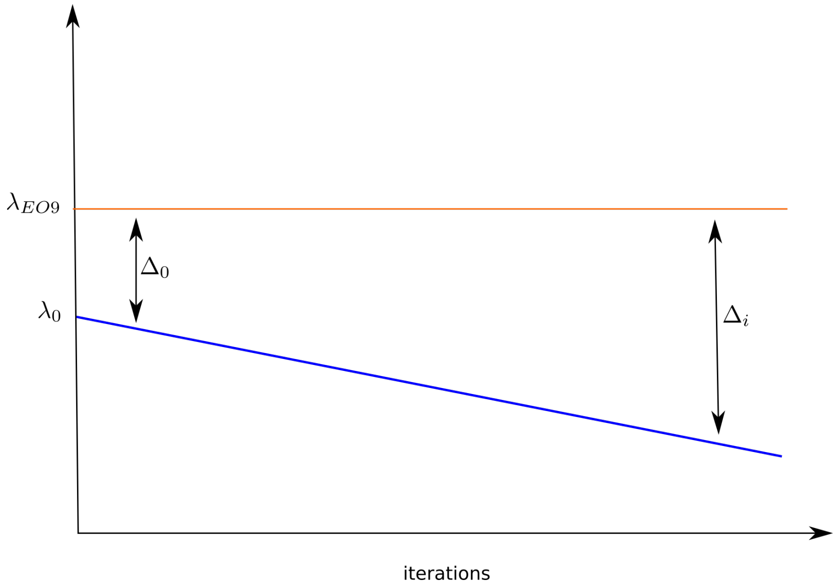


Figure 55: Visualization of the vibrational objective of maximizing the distance between the fundamental eigenvalue λ_0 and the ninth engine order λ_{EO9} by reducing λ_0

6.4.1 Numerical Setup

For the most part, the same numerical setup introduced in section 6.3 is used. In addition, the vibrational constraints are introduced. For turbomachinery applications, the vibration requirements mainly involve the avoidance of resonance with dominant engine frequencies. As a result, the lowest natural frequency, also known as the fundamental eigenfrequency, should not coincide with dominant engine order frequencies. For instance, assuming the radial turbine is surrounded by nine stator vanes, the turbine will be subjected to periodic disturbances nine times per revolution. Hence, the ninth engine order is significant. This goal is also visualized in figure 55, where the delta between the fundamental eigenvalue and the ninth engine order should be increased as the optimization progresses.

The engine frequency is computed as

$$\omega_{EO_N} = N \frac{RPM}{60}, \tag{153}$$

where N represents the engine order number and RPM the rotations per minute of the radial turbine. The engine order eigenvalue is thus computed as

$$\lambda_{EO_N} = \omega_{EO_N}^2. \tag{154}$$

The vibrational penalty term is then formulated as

$$P_\lambda = \omega_\lambda \cdot \frac{|\lambda_{EO_N} - \lambda_0|}{\lambda_{EO_N}}, \tag{155}$$

	Baseline	Optimized	Δ [%]
Isentropic Efficiency (TS) $[-]$	0.7464	0.7654	2.55%
Power $[kW]$	1.2535	1.3379	6.73%
Max VM Stress $[MPa]$	681.801	661.433	-2.99%
$\lvert\lambda_0 - \lambda_{EO9}\rvert$ $[s^{-2}]$	1.514e8	1.945e8	28.47%

Table 10: Aerodynamic and structural quantities of interest at baseline design and optimized design

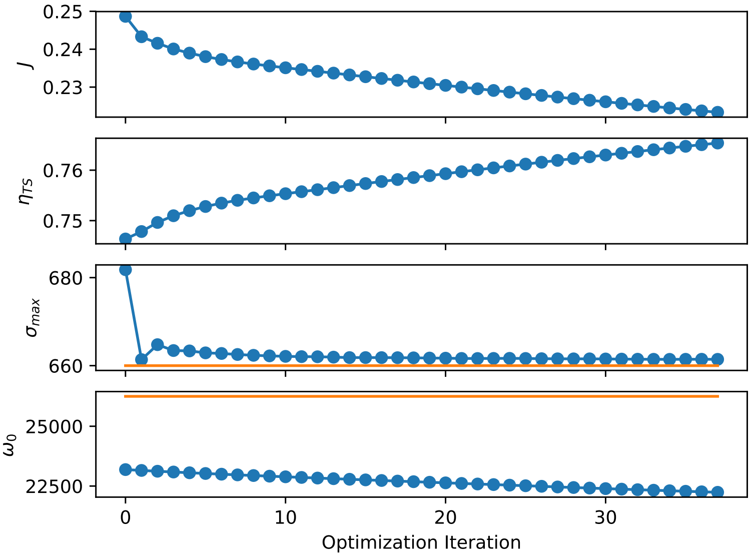


Figure 56: Plot of optimization history showing objective J, efficiency η_{TS}, maximum von Mises stress σ_{max}, where the orange line represents $\sigma_{req} = 660\,MPa$, and the fundamental eigenfrequency ω_0, where the orange line represents the ninth order engine frequency ω_{EO9}.

with a fixed coefficient ω_λ, which is then added to the objective function to finally receive

$$J = (1 - \eta_{TS}) + \omega_\sigma \cdot Sig(\sigma_{max}) \cdot \left(\frac{\sigma_{max} - \sigma_{req}}{\sigma_{req}}\right)^2 - P_\lambda. \qquad (156)$$

6.4.2 Optimization Results

The optimization history is shown in figure 56 and the results are tabulated in table 10. Similar results have been obtained as with the previous MDO (sec. 6.3). The objective has continuously decreased, leading to an overall efficiency increase of 2.55%, which in turn led to a power increase of 6.73%. The maximum von Mises stress has been reduced by 2.99%. While slightly violating the stress constraint, the final value is only 0.22% above the set threshold of $\sigma_{req} = 660\,MPa$. For the vibrational aspect, the difference between the fundamental eigenvalue and ninth engine order eigenvalue was successfully increased by 28.47%.

7 Conclusion and Outlook

This thesis has introduced a CAD-based multidisciplinary optimization framework using the adjoint method. In particular, the development and integration of an adjoint CSM solver, differentiated using AD, was discussed. The optimization framework was introduced in chapter 3, which highlighted how the entire optimization chain departs from the CAD design parameters. Based on these inputs, the CAD geometry and computational meshes are generated. The CAD surface acts as the interface that links the two disciplines, fluid and solid, together. The mesh generation for the fluid discipline was differentiated using forward AD to enable gradient calculations of the CFD mesh with respect to CAD parameters. For the solid discipline, on the other hand, a linear-elastic mesh deformation using the differentiated CSM solver was developed to compute the structural mesh gradient contributions. The CAD kernel was differentiated using forward AD.

For the gradients of the performance parameters, such as aerodynamic efficiency and maximum von Mises stresses, adjoint CFD (section 3.5) and CSM solvers were utilized. Within this thesis, the adjoint CSM solver was developed to perform a static linear stress analysis and free vibration analysis (chapter 4). The developed solver was validated against the widely-used open-source solver *Calculix* and both the stress and vibration analysis were differentiated using the AD tool *CoDiPack* (chapter 5). To obtain computationally efficient results for the gradient calculations, a tandem differentiation of black-box AD and special treatment of the iterative linear and eigenvalue solvers was carried out. As a result, the maximum von Mises stress gradients could be computed at an additional cost of one linear system solve using the same stiffness matrix and the eigenvalue gradients can be calculated at an additional cost of one outer product per eigenvalue. The computational cost of an entire gradient evaluation, including the forward and reverse run, has a peak memory usage of 2.76 times a primal run, including the calculation of both the stiffness and mass matrices. The run time ratio for a stress gradient calculation is approximately 3.14 times a primal run.

Additionally, this thesis explored how the object-oriented design of a solver, differentiated using operator-overloading AD, can be exploited to easily calculate gradients for different applications. The gradient calculation of the unstructured mesh deformation based on a linear elastic analogy reuses the differentiated base classes of the linear stress analysis (section 5.5.1). Gradients with respect to additional input parameters are also easily obtainable. This was discussed in section 5.6, where a different element class for composite elements was implemented to compute gradients with respect to material properties in addition to the usual shape parameters. Gradients for this case were directly available due to the differentiated parent class implementations, reducing AD-differentiation overhead that may occur otherwise.

Finally, the differentiated CAD kernel, adjoint CSM and CFD solvers, were integrated into a CAD-based adjoint MDO framework. Its application was then tested by performing a shape optimization of a radial turbine (chapter 6). First, a structural optimization was performed to validate the applicability of the computed stress gradients and differentiated unstructured mesh deformation. This test case was then further extended into a multidisciplinary optimization with the objective of maximizing isotropic efficiency, while maintaining maximum stress levels below a given threshold. Finally, the differentiated vibration analysis was included in the MDO framework to extend the test case's objective with the minimization of resonance.

With the optimization framework set up and the base structural solver differentiated, the road is paved for future developments. This thesis used gradient descent with a fixed step size, while future contributions could make use of more advanced techniques, e.g., the inclusion of step size adaptation methods or using a third party optimization library such as *SNOPT* [19] or *SciPy* [78]. The composite element component could be further extended by a corresponding CAD kernel, mesh generation and objective functions to peform an MDO not just for shape optimization, but also for simultaneous material optimization. Additionally, more physically-relevant effects such as thermal loads, centrifugal stiffening, up to a fully coupled fluid-structure interaction could be implemented in the structural solver and MDO framework. If possible, future code contributions should take advantage of the differentiated base classes to minimize the code differentiation efforts. The CAD kernel could be updated to a more object-oriented differentiation, similar to the CSM solver. Such a differentiated base CAD kernel would facilitate an accelerated development cycle for different CAD applications, such as radial turbines, axial fans, compressors, etc.

With continuous advances in this field, multidisciplinary optimizations will easily become multi-component optimizations, where not just multiple physical discplines are combined, but multiple mechanical components are combined and optimized simultaneously to achieve a well-defined common objective. Building on that, the objectives of future optimizations would eventually no longer be as low-level as *isotropic efficiency*, but rather high-level objectives such as maximizing *fuel economy*, overall *life-span* of components, or other industry objectives. In fact, the optimization framework introduced in this thesis could be generalized, i.e., modularized even further to provide an optimization *software suite* where the user could easily piece together an optimization problem of different fidelities.

References

[1] Mohammad Abu-Zurayk and Joël Brezillon. Shape optimization using the aerostructural coupled adjoint approach for viscous flows. *Evolutionary and deterministic methods for design, optimization and control, Capua*, 2011.

[2] Steven R. Allmaras and Forrester T. Johnson. Modifications and clarifications for the implementation of the Spalart-Allmaras turbulence model. In *Seventh international conference on computational fluid dynamics (ICCFD7)*, pages 1–11, 2012.

[3] Nitish Anand, Salvatore Vitale, Matteo Pini, and Piero Colonna. Assessment of FFD and CAD-based shape parametrization methods for adjoint-based turbomachinery shape optimization. *Proceedings of Montreal*, 7:9th, 2018.

[4] Jan Backhaus, Andreas Schmitz, Christian Frey, Max Sagebaum, Sebastian Mann, Marc Nagel, and Nicolas R. Gauger. Application of an algorithmically differentiated turbomachinery flow solver to the optimization of a fan stage. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, page 3997, 2017.

[5] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.13, Argonne National Laboratory, 2020.

[6] Ever J. Barbero. *Finite element analysis of composite materials using Abaqus*^{TM}. CRC press, 2013.

[7] Jiri Blazek. *Computational fluid dynamics: principles and applications*. Butterworth-Heinemann, 2015.

[8] Martin Bücker and Paul Hovland. Community portal for automatic differentiation. `www.autodiff.org`. Accessed July 28, 2020.

[9] Angelo Carnarius, Frank Thiele, Emre Özkaya, Anil Nemili, and Nicolas R. Gauger. Optimal control of unsteady flows using a discrete and a continuous adjoint approach. In *IFIP Conference on System Modeling and Optimization*, pages 318–327. Springer, 2011.

[10] Stephen Chen, James Montgomery, and Antonio Bolufé-Röhler. Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution. *Applied Intelligence*, 42(3):514–526, 2015.

[11] Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer-Verlag, 2006.

[12] Guido Dhondt. *The finite element method for three-dimensional thermomechanical applications*. John Wiley & Sons, 2004.

[13] Guido Dhondt, Marc Nagel, and Nicolas R. Gauger. Calculation of the eigenfrequeny sensitivity of aircraft engine structures. *Proceedings of EUROGEN 2005*, 2005.

[14] Guido Dhondt and Klaus Wittig. Calculix: a free software three-dimensional structural finite element program. *MTU Aero Engines GmbH, Munich*, 1998.

[15] Rolf Dornberger, Dirk Büche, and Peter Stoll. Multidisciplinary optimization in turbomachinery design. In *European Congress on Computational Methods in Applied Sciences and Engineering, Barcelona, Spain*, 2000.

[16] Russell Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pages 39–43. IEEE, 1995.

[17] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[18] Michael B. Giles and Niles A. Pierce. Adjoint equations in CFD - Duality, boundary conditions and solution behaviour. *13th Computational Fluid Dynamics Conference*, 1997.

[19] Philip E. Gill, Walter Murray, Michael A. Saunders, and Elizabeth Wong. User's guide for SNOPT 7.7: Software for large-scale nonlinear programming. Center for Computational Mathematics Report CCoM 18-1, Department of Mathematics, University of California, San Diego, La Jolla, CA, 2018.

[20] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[21] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[22] Zafer Gürdal, Raphael T. Haftka, and Prabhat Hajela. *Design and optimization of laminated composite materials.* John Wiley & Sons, 1999.

[23] Ami Harten and James M. Hyman. Self adjusting grid methods for one-dimensional hyperbolic conservation laws. *Journal of computational Physics*, 50(2):235–269, 1983.

[24] Laurent Hascoët and Valérie Pascual. *Tapenade 2.1 user's guide.* PhD thesis, INRIA, 2004.

[25] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software*, 31(3):351–362, 2005.

[26] John H. Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[27] Jocelyn Iott, Raphael T. Haftka, and Howard M. Adelman. Selecting step sizes in sensitivity analysis by finite differences. Technical Report Memorandum 86382, NASA, 1985.

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

[29] Antony Jameson, Luigi Martinelli, and Niles A. Pierce. Optimum aerodynamic design using the Navier-Stokes equations. *Theoretical and computational fluid dynamics*, 10(1-4):213–237, 1998.

[30] Antony Jameson and James Reuther. Control theory based airfoil design using the euler equations. In *5th Symposium on Multidisciplinary Analysis and Optimization*, page 4272, 1994.

[31] Norman L. Johnson. The legacy and future of CFD at Los Alamos. Technical report, Los Alamos National Lab., NM (United States), 1996.

[32] Michael M. Joly, Tom Verstraete, and Guillermo Paniagua. Integrated multifidelity, multidisciplinary evolutionary design optimization of counter-rotating compressors. *Integrated Computer-Aided Engineering*, 21(3):249–261, 2014.

[33] Scott Kirkpatrick, C. Daniel Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[34] Tae Hee Lee. Adjoint method for design sensitivity analysis of multiple eigenvalues and associated eigenvectors. *AIAA journal*, 45(8):1998–2004, 2007.

[35] Klaus Leppkes, Johannes Lotz, and Uwe Naumann. Derivative code by overloading in C++(dco/c++): Introduction and summary of features. Technical report, AIB-2016-08, RWTH Aachen University, 2016.

[36] James N. Lyness and Cleve B. Moler. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210, 1967.

[37] Charles A. Mader, Gaetan K. Kenway, Joaquim Martins, and Alejandra Uranga. Aerostructural optimization of the D8 wing with varying cruise Mach numbers. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, page 4436, 2017.

[38] Joaquim R. R. A. Martins. A coupled-adjoint method for high-fidelity aero-structural optimization. Technical report, Stanford University, Dept. of Aeronautics and Astronautics, 2002.

[39] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, 2004.

[40] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. *Optimization and Engineering*, 6(1):33–62, 2005.

[41] Joaquim R. R. A. Martins and Andrew B. Lambe. Multidisciplinary design optimization: a survey of architectures. *AIAA journal*, 51(9):2049–2075, 2013.

[42] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J. Alonso. The complex-step derivative approximation. *ACM Transactions on Mathematical Software (TOMS)*, 29(3):245–262, 2003.

[43] Asitav Mishra, Dimitri Mavriplis, and Jay Sitaraman. Time-dependent aeroelastic adjoint-based aerodynamic shape optimization of helicopter rotors in forward flight. *AIAA Journal*, pages 3813–3827, 2016.

[44] Bijan Mohammadi. Optimization of aerodynamic and acoustic performances of supersonic civil transports. *International Journal of Numerical Methods for Heat & Fluid Flow*, 14(7):893–909, 2004.

[45] Lasse Müller. *Multipoint Adjoint-Based Optimization of Turbomachinery with Application to Turbocharger Radial Turbines.* PhD thesis, Universite Libre de Bruxelles, 2018.

[46] Lasse Müller, Zuheyr Alsalihi, and Tom Verstraete. Multidisciplinary optimization of a turbocharger radial turbine. *Journal of Turbomachinery*, 135(2):021022, 2013.

[47] Lasse Müller and Tom Verstraete. CAD integrated multipoint adjoint-based optimization of a turbocharger radial turbine. *International Journal of Turbomachinery, Propulsion and Power*, 2(3):14, 2017.

[48] Orest Mykhaskiv, Mladen Banović, Salvatore Auriemma, Pavanakumar Mohanamuraly, Andrea Walther, Herve Legrand, and Jens-Dominik Müller. NURBS-based and parametric-based shape optimization with differentiated CAD kernel. *Computer-Aided Design and Applications*, 15(6):916–926, 2018.

[49] Uwe Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*, volume 24. SIAM, 2012.

[50] Niles A. Pierce and Michael B. Giles. Preconditioned multigrid methods for compressible flow calculations on stretched meshes. *Journal of Computational Physics*, 136(2):425–445, 1997.

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

[52] Bellman Richard. Dynamic programming. *Princeton University Press*, 89:92, 1957.

[53] Lewis Fry Richardson. *Weather prediction by numerical process.* Cambridge University Press, 2007.

[54] Philip L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[55] Antonio Rubino, Matteo Pini, Piero Colonna, Tim Albring, Sravya Nimmagadda, Thomas Economon, and Juan Alonso. Adjoint-based fluid dynamic design optimization in quasi-periodic unsteady flow problems using a harmonic balance method. *Journal of Computational Physics*, 372:220–235, 2018.

[56] Max Sagebaum, Tim Albring, and Nicolas R. Gauger. High-performance derivative computations using CoDiPack. *ACM Transactions on Mathematical Software (TOMS)*, 45(4), 2019.

[57] Ismael Sanchez Torreguitart, Tom Verstraete, and Lasse Müller. Optimization of the LS89 axial turbine profile using a CAD and adjoint based approach. *International Journal of Turbomachinery, Propulsion and Power*, 3(3):20, 2018.

[58] Marc Schwalbach, Tom Verstraete, and Nicolas R. Gauger. Adjoint optimization of turbomachinery components under mechanical constraints. In *The 8th VKI PhD Symposium*, 2017.

[59] Marc Schwalbach, Tom Verstraete, Jens-Dominik Müller, and Nicolas R. Gauger. A comparative study of two different CAD-based mesh deformation methods for structural shape optimization. In *Evolutionary and Deterministic Methods for Design Optimization and Control With Applications to Industrial and Societal Problems*, pages 47–60. Springer, 2019.

[60] Marc Schwalbach, Tom Verstraete, Lasse Müller, and Nicolas R. Gauger. CAD-based adjoint multidisciplinary optimization of a radial turbine under structural constraints. In *Conference Proceedings of the GPPS 2018 (GPPS-NA-2018-133)*, 2018.

[61] Hang Si. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software (TOMS)*, 41(2):11, 2015.

[62] Ulrich Siller, Christian Voß, and Eberhard Nicke. Automated multidisciplinary optimization of a transonic axial compressor. In *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, page 863, 2009.

[63] William S. Slaughter. The linearized theory of elasticity. *ISBN-10*, 817641173, 2002.

[64] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[65] William Sutherland. LII. The viscosity of gases and molecular force. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 36(223):507–531, 1893.

[66] Mohamed Tadjouddine, Shaun A. Forth, and Andy J. Keane. Adjoint differentiation of a structural dynamics solver. In *Automatic Differentiation: Applications, Theory, and Implementations*, pages 309–319. Springer, 2006.

[67] Dominique Thévenin and Gábor Janiga. *Optimization and computational fluid dynamics.* Springer Science & Business Media, 2008.

[68] Joe F. Thompson, Zahir U. A. Warsi, and C. Wayne Mastin. *Numerical grid generation: foundations and applications*, volume 45. North-holland Amsterdam, 1985.

[69] Mark G. Turner, Kevin Park, Kiran Siddappaji, Soumitr Dey, David P. Gutzwiller, Ali Merchant, and Dario Bruna. Framework for multidisciplinary optimization of turbomachinery. *ASME Paper Number GT2010-22228*, 2010.

[70] Bram Van Leer. Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *Journal of computational Physics*, 32(1):101–136, 1979.

[71] Ilias Vasilopoulos, Dheeraj Agarwal, Marcus Meyer, Trevor T. Robinson, and Cecil G. Armstrong. Linking parametric CAD with adjoint surface sensitivities. In *ECCOMAS Congress 2016—proceedings of the 7th European congress on computational methods in applied sciences and engineering*, volume 2, pages 3812–3827, 2016.

[72] Venkat Venkatakrishnan. On the accuracy of limiters and convergence to steady state solutions. In *31st Aerospace Sciences Meeting*, page 880, 1993.

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

[74] Tom Verstraete, Zuheyr Alsalihi, and René A. Van den Braembussche. Multidisciplinary optimization of a radial compressor for microgas turbine applications. *Journal of Turbomachinery*, 132(3):031004, 2010.

[75] Tom Verstraete, Lasse Müller, and Jens-Dominik Müller. Adjoint-based design optimisation of an internal cooling channel U-bend for minimised pressure losses. *International Journal of Turbomachinery, Propulsion and Power*, 2(2):10, 2017.

[76] Tom Verstraete, Lasse Müller, and Jens-Dominik Müller. Multidisciplinary adjoint optimization of trubomachinery components including aerodynamic and stress performance. In *35th AIAA Applied Aerodynamics Conference*, page 4083, 2017.

[77] Tom Verstraete, Lasse Müller, and Jens-Dominik Müller. CAD based adjoint optimization of the stresses in a radial turbine. In *Proceedings of ASME Turbo Expo 2017: Turbine Technical Conference and Exposition*, 2017.

[78] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert

Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[79] Salvatore Vitale, Tim A. Albring, Matteo Pini, Nicolas R. Gauger, and Piero Colonna. Fully turbulent discrete adjoint solver for non-ideal compressible flow applications. *Journal of the Global Power and Propulsion Society*, 1:252–270, 2017.

[80] Andrea Walther and Andreas Griewank. Getting started with ADOL-C. In *Combinatorial scientific computing*, pages 181–202, 2009.

[81] Shenren Xu, David Radford, Marcus Meyer, and Jens-Dominik Müller. Stabilisation of discrete steady adjoint solvers. *Journal of Computational Physics*, 299:175–195, 2015.

[82] Olek C. Zienkiewicz, Robert L. Taylor, and Jian Z. Zhu. *The finite element method: its basis and fundamentals*. Elsevier, 2005.

[83] Olgierd Cecil Zienkiewicz, Robert Leroy Taylor, Olgierd Cecil Zienkiewicz, and Robert Lee Taylor. *The finite element method*, volume 36. McGraw-hill London, 1977.

Curriculum Vitae

Marc Schwalbach

Education

2015 - 2021	PhD in Computer Science
Germany	Chair for Scientific Computing, TU Kaiserslautern
Belgium	von Kármán Institute for Fluid Dynamics
2013 - 2015	M.Sc. in Computational Engineering Science
Germany	RWTH Aachen University
	mit Auszeichnung
2009 - 2013	B.Sc. in Computational Engineering Science
Germany	RWTH Aachen University
2008	International Baccalaureate
Philippines	Cebu International School

Scholarships and Awards

2015 - 2018	Marie Curie Early Stage Researcher Fellowship
Belgium	von Kármán Institute for Fluid Dynamics
2016	Springorum Commemorative Coin
Germany	RWTH Aachen University
2011	DAAD RISE Worldwide Scholarship
Canada	The University of British Columbia

Employment

2018 - 2021	Engineer Research
Germany	GreenStone Energy GmbH
2015 - 2018	Marie Curie Early Stage Researcher
Belgium	von Kármán Institute for Fluid Dynamics