Imperial College of Science, Technology and Medicine
Department of Aeronautics

A High-Performance Open-Source Framework for Multiphysics Simulation and Adjoint-based Shape and Topology Optimization

Pedro Carrusca Gomes

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Aeronautical Engineering of Imperial College London.

June 2021

Statement

I hereby declare that I am the author of this thesis, and that it is the product of my own work. This dissertation has not been submitted for consideration towards any other degree or award. Where appropriate, I have fully acknowledged the ideas and results from the work of other people.

Copyright Notice

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under an Attribution-NonCommercial-NoDerivatives 4.0 International license (CC BY-NC-ND 4.0). Under this licence, you may copy and redistribute the material in any medium or format. You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. You may not use the material for commercial purposes. If you remix, transform, or build upon the material, you may not distribute the modified material. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

The first part of this thesis presents the advances made in the Open-Source software SU2, towards transforming it into a high-performance framework for design and optimization of multiphysics problems. Through this work, and in collaboration with other authors, a tenfold performance improvement was achieved for some problems. More importantly, problems that had previously been impossible to solve in SU2, can now be used in numerical optimization with shape or topology variables. Furthermore, it is now exponentially simpler to study new multiphysics applications, and to develop new numerical schemes taking advantage of modern high-performance-computing systems.

In the second part of this thesis, these capabilities allowed the application of topology optimization to medium scale fluid-structure interaction problems, using high-fidelity models (nonlinear elasticity and Reynolds-averaged Navier-Stokes equations), which had not been done before in the literature. This showed that topology optimization can be used to target aerodynamic objectives, by tailoring the interaction between fluid and structure. However, it also made evident the limitations of density-based methods for this type of problem, in particular, reliably converging to discrete solutions. This was overcome with new strategies to both guarantee and accelerate (i.e. reduce the overall computational cost) the convergence to discrete solutions in fluid-structure interaction problems.

Acknowledgements

I thank my supervisor, Prof. Rafael Palacios, for guiding me through the PhD without ever setting a course, thereby giving me freedom to pursue less academically-profitable subjects, while keeping me from crashing against the rocks (great patience he has, grateful I am).

I thank my early and late stage assessors, Dr. Ajit Panesar and Dr. Chris Cantwell, respectively, for their helpful advice on topology optimization and high performance computing. I am also grateful to my examiners, Prof. Joaquim Martins and Prof. Peter Vincent for taking the time to read and discuss this work.

To the SU2 developers with whom I had the opportunity to work more closely, or whose work, or knowledge, helped me most, in chronological order, Dr. Rubén Sánchez Fernández, Dr. Charanya Venkatesan-Crome, Tim Albring, Dr. Thomas D. Economon, Prof. Edwin van der Weide, Ole Burghardt, and Tobias Kattmann. I thank you for the always-interesting discussions we had (and maybe will continue to), at times too lively, alas it takes practice to climb onto someone's shoulders without occasionally stepping on their toes. I also thank Tobias Kattmann for proof-reading chapters 3 and 4.

I thank Prof. Nicolas Gauger for welcoming me at TU Kaiserslautern for a brief, but productive, time of my PhD. I am also grateful to Dr. Max Sagebaum and Johannes Blühdorn for their help with many aspects of algorithmic differentiation, from theory to integration in SU2.

I am especially thankful for the computational resources gracefully provided by the UK Turbulence Consortium, in particular grant EP/R029326/1 and its principal investigator Dr. Sylvain Laizet. The work in this thesis would not have been possible otherwise. I am also grateful to the UK Materials and Molecular Modelling Hub for computational resources which are partially funded by EPSRC (EP/T022213/1), and to EPSRC (EP/R007470/1) for partly funding this research.

Finally, I thank my family for setting the example, and encouraging me to strive for what I believe in.

Contents

Statement **i**

Copyright Notice **iii**

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 Topology optimization with fluid-structure interaction 2

1.2 PDE-constrained optimization 4

1.3 Research goals 6

1.4 Thesis layout 7

1.5 Publications 7

2 Methodology **9**

2.1 Partitioned simulation of fluid-structure interaction 9

2.1.1 Fluid problem 10

2.1.2 Structural problem 13

2.1.3 Mesh deformation 14

2.1.4 Coupling conditions 16

2.1.5 Partitioned solution methods 17

2.2 Density-based topology optimization 19

2.2.1 Challenges and solutions 21

2.2.2 Solution by gradient-based optimization methods 23

2.2.3 Benchmark example 25

2.2.4 A strategy for discrete topologies in FSI 26

3 Discrete Adjoint Method **29**

3.1 Adjoint equations 29
3.1.1 Interpretation of adjoint variables 31
3.1.2 Solution methods 33
3.1.3 Adjoints of complex programs 43
3.1.4 Trade-off between performance and generality 44
3.2 Implementation aspects 45
3.2.1 Manual differentiation 45
3.2.2 Algorithmic differentiation 47
3.2.3 Matrix-based methods 51
3.2.4 Choice of parameters 52
3.3 Multiphysics adjoints in SU2 53
3.3.1 Earlier work on fluid-structure-interaction 54
3.3.2 General method for multizone problems 56

4 Efficient Implementation of Unstructured Solvers 63
4.1 Capabilities and limitations of hardware and software 64
4.1.1 Hardware 64
4.1.2 Programming languages and compilers 65
4.1.3 Effects of storage layout 78
4.1.4 Explicit and implicit vectorization 83
4.1.5 Algorithmic complexity 89
4.2 Algorithms and data structures of unstructured solvers 91
4.2.1 Composition and adjacency matrices 91
4.2.2 Types of algorithm 93
4.2.3 Improving temporal locality 95
4.3 Parallelization approaches 95
4.3.1 Domain decomposition 96
4.3.2 Shared data parallelism 98
4.3.3 Interaction between approaches 102
4.3.4 Loop carried dependencies 103
4.4 Case study, SU2 implementation improvements 105
4.4.1 General optimizations 107
4.4.2 Hybrid parallel strategy 109
4.4.3 Vectorization 111
4.4.4 Solution times on a large scale problem 116

4.4.5 Algorithmic improvements allowed by better performance 117
4.4.6 Summary of improvements 118

5 Shape and Topology Optimization with Fluid-Structure Interaction 119
5.1 A compliant airfoil 119
5.1.1 Baseline airfoil from aerostructural shape optimization 120
5.1.2 Fully coupled topology optimization step 121
5.1.3 Inverse design step 123
5.1.4 Comparison with conventional approaches to encourage solid-void topologies 125
5.1.5 Discussion 128
5.2 A flexible wing 129
5.2.1 Baseline rigid design 130
5.2.2 Topology optimization 133
5.2.3 Final remarks 141

6 Conclusions 143
6.1 Research outcomes 144
6.2 Future work 144

Bibliography 146

List of Tables

3.1 Cost for left and right preconditioned fixed-points, total BiCGSTAB iterations. . 38

3.2 Cost for different transposed linear solver options to obtain residuals of 10^{-3} for the NACA0012 problem. 39

3.3 Cost of fixed-point and Krylov approaches to obtain residuals of 10^{-3} for the NACA0012 problem. 41

3.4 Memory used by the SU2 discrete adjoint solver relative to primal, typical RANS problem. 49

3.5 Time per explicit iteration of the SU2 discrete adjoint flow solver relative to primal. 50

4.1 Effect of small loop optimizations on small matrix size. 68

4.2 Effect of small loop optimizations on large matrix size. 69

4.3 Effect of access patterns on different storage layouts. 79

4.4 Effect of prefetching on scattered access pattern. 80

4.5 Effect of vectorizing sequential accesses on column major layout. 81

4.6 Comparison of solution times for the simple benchmark, before and after general optimizations. 109

4.7 Effect of vectorization on the simple benchmark problem, 24 core machine with AVX2. 115

4.8 Effect of vectorization on the simple benchmark problem, 40 core machine with AVX512. 116

4.9 Lift and drag coefficients of the HLCRM at 8 deg AoA. 117

5.1 Shape optimization parametric study results ($()^l$: Ma 0.25, $()^h$: Ma 0.5). 121

5.2 Topology optimization summary. 124

5.3 Performance values at different optimization steps. 140

List of Figures

2.1 4:1 aspect ratio tip loaded cantilever. 25

2.2 Optimized topology for 60 kN with the *open* filter. 26

2.3 Optimized topology for 240 kN with the *close* filter. 26

2.4 4:1 tip loaded cantilever optimization histories for the two load values considered. 26

3.1 Magnitude and direction of the adjoint momentum of lift for an airfoil at 0 deg AoA. 33

3.2 Comparison of left and right preconditioned fixed-point iterations. 38

3.3 Effect of linear solver type and tolerance on the convergence of SU2's discrete adjoint solver for the NACA0012 problem. 39

3.4 Comparison of fixed-point (black) and Krylov (red) methods for the NACA0012 problem, lift objective function. 41

3.5 Results for the HL-CRM at 8 deg AoA and Mach 0.2, "B3" coarse mesh. 42

3.6 Schematic representation of the solution of two sets of variables by sequential quasi-Newton iterations. 43

3.7 Wing geometry and free-form-deformation box, viewed from the wing tip and the top. 59

3.8 Deformed configuration of the wing and pressure coefficient contours. 59

3.9 Absolute value of drag sensitivities, from the adjoint solver, and finite difference error bounds. 59

3.10 Absolute value of elastic energy sensitivities, from the adjoint solver, and finite difference error bounds. 59

4.1 Types of 2-D layout. a) Row-major or array of structures; b) Column-major or structure of arrays; c) Array of structure of arrays. 79

4.2 Strategies to gather data for multiple outer indices to then process it with SIMD instructions. 82

4.3 Definition of the elements of a mesh. 92

4.4 Definition of the connections between control volumes. 92

4.5 Gather-scatter pattern in edge loops. 94

4.6 Global matrix structure. 96
4.7 Local matrix structures and vectors with halo values. 97
4.8 Data race in an edge loop, caused by simultaneous access to one point. 99
4.9 Reduction strategy to avoid data races in edge loops. 100
4.10 Simple (single) and edge group coloring. 102
4.11 Difference between funneled and multiple communication modes. 103
4.12 Example coloring of the rows of a matrix to allow simultaneous Gauss-Seidel updates within each color. 105
4.13 Geometry, with pressure coefficient contours, of the simple benchmark problem. 106
4.14 Pressure coefficient contours on the high lift CRM at 8 deg AoA, medium grid. . 107
4.15 Observed speedup vs expected speedup with increasing number of compute nodes. 110
4.16 Average time per iteration against number of threads per rank (2-socket 24 core nodes), CRM Coarse mesh. 111
4.17 Percentage of time spent in different tasks of the flow solver. 116
4.18 Comparison of convergence and solution time between quasi-Newton and Newton-Krylov approaches for the high-lift CRM Coarse grid. 118

5.1 Shape optimization parameterization. 121
5.2 Baseline shape optimization results for 10 mm constraint, undeformed (black) and deformed at Mach 0.5 (red) configurations. 121
5.3 Baseline shape optimization results for 6 mm constraint, undeformed (black) and deformed at Mach 0.5 (red) configurations. 121
5.4 *Fully coupled* step convergence history. 123
5.5 *Fully coupled* step results, filtered structural density and Mach number contours. 123
5.6 Results of *inverse design* step, density contours, target (black) and actual (red) deformed surfaces. 124
5.7 Detail of *inverse design* step results w/o (left) and with (right) stability function. 125
5.8 *Weighted average* approach, convergence history. 126
5.9 *Weighted average* approach results, filtered structural density and Mach number contours, deformed configuration from *fully coupled* step in red for reference. . . 126
5.10 Density contours on the deformed geometry for the *weighted average* objective with SIMP exponent of 4. 127
5.11 Convergence history for *gradual mass minimization* approach. 127
5.12 Density contours after the *gradual mass minimization* approach (deformed geometry), *fully coupled* step deformed outline (in black) for reference. 128
5.13 *Gradual mass minimization* approach, contours of derivative of penalized objective function with respect to density. 128

5.14 Pressure coefficient distribution at Mach 0.5 for the baseline airfoil (6 mm constraint) and for the *fully coupled* step result. 129

5.15 Initial wing geometry and FFD box. 131

5.16 Structural mesh near wing tip. 131

5.17 Comparison of initial (bottom) and optimized (top) wing. 132

5.18 Pressure coefficient contours on initial (bottom) and optimized (top) designs. . . 132

5.19 Spanwise lift distribution on initial (light color) and optimized (dark color) designs. 133

5.20 Results of recovering lift. 134

5.21 Material distribution after minimizing stiff fraction. 135

5.22 Material distribution after minimizing compliance. 136

5.23 Material distribution after minimizing stiff fraction for a lower compliance constraint. 137

5.24 Material distribution after improving discreteness. 138

5.25 Material distribution after improving discreteness (different view). 139

5.26 Comparison of deformed configurations, between recovering lift and final design. 140

5.27 Spanwise lift distribution on initial (light color), shape-optimized (dark color), and final (dashed line) designs. 141

5.28 Von Mises stress normalized by local elasticity modulus. 141

Chapter 1

Introduction

The need for more efficient commercial aircraft has resulted in an increase of wing aspect ratio (AR) over the years [1], which has been greatly facilitated by the adoption of composite material-based designs. In the high altitude long endurance (HALE) aircraft space, the aspect ratios used are even higher ($\approx$35) [2]. Moreover, wind turbine blades can also be seen as high AR wings and in this field the tendency has also been to increase the power output of single units, by increasing their size in order to improve efficiency and harness the energy of higher altitude winds [3].

Slender aerodynamic structures designed for low weight and high aerodynamic efficiency will unavoidably be subjected to large displacements and strong fluid structure interaction (FSI) effects, i.e. the deformation of the structure due to the aerodynamic loads alters its aerodynamic characteristics thereby changing the loads applied initially and so on. The aeroelastic system may reach static equilibrium, exhibit limit cycle oscillations (e.g. flutter), or diverge (suffering catastrophic failure). In the presence of strong coupling effects, prediction and optimization of aerodynamic features and flight performance, is not possible with current tools. Therefore, simulation of this non-linear, tightly coupled behavior requires new robust computational methods capable of accurately modeling fluid flow, structural deformations accounting for geometric, and possibly material, non-linearities, and the interaction between domains.

The main purpose of this move towards larger and slender structures (and thus more flexible), is to increase the efficiency of the machines they are part of, therefore it is highly desirable that they be optimum in some sense. There are many examples in the literature of large scale aircraft optimization [4, 5]. However, the structural and aerodynamic design are often separated and done sequentially, or simplified when they are done simultaneously (by using medium fidelity models). While it has been shown that a sequential design strategy can produce sub-optimal results in the presence of tight coupling between disciplines [6], simplifications are nevertheless required to reduce the computational cost of numerical optimization of complex systems.

The conventional structural design of wings consists in sizing the individual components of the skeleton-like structure to which the wing leading and trailing edge fairings, fuel tanks, and control surface actuators are attached. These components are: the spars, large spanwise beam-like elements that confer bending stiffness to the wing; the ribs, that define the cross section of the wing and together with the spars form the wing-box; the stringers, which are spanwise elements that transfer the load from the outer panels to the spars and ribs. Considering only its load bearing internal components, a wing is mostly hollow. Topology optimization is a method of determining how to best distribute material within an available volume according to some measure of constrained optimality. Compared to the conventional design process, it represents a radical paradigm shift (from which those more flexible structures may benefit). However, reports of its application to wing design are few and far between. Arguably, structures resulting from applying topology optimization to the entire volume of a wing would be difficult to manufacture with current methods, unless this requirement is taken in consideration, e.g. by constraining the shape of the available volume. Nevertheless, the unconstrained approach may be viable for smaller structures, or specialized applications where additive manufacturing may be suitable, or when the benefits outweigh the costs. Therefore, it is worth exploring this design concept, to understand the potential benefits and limitations of current topology optimization methods applied to tighly coupled FSI problems.

There is growing interest in even richer combinations of physical disciplines (than just FSI). For example, supersonic flight and some turbomachinery applications may warrant aerothermoelastic considerations during their design [7], certainly other multiphysics applications await the design tool that will enable them. There is merit, therefore, in developing highly generic numerical methods and software for multiphysics simulation and optimization. Furthermore, it is reasonable to expect the importance of simultaneously optimizing all disciplines to increase as their number grows, for this reason, it is important for those methods (and their software implementation) to be nearly as efficient as purpose-built tools.

Some literature on these two research goals is reviewed next to further frame the motivation for this work. Method-specific literature is reviewed in later chapters as the chosen methods and their alternatives are discussed.

1.1 Topology optimization with fluid-structure interaction

Maute et al. ([8, 9]) tackled the optimum layout (minimum material) of reinforcement structures for an airfoil and a wing subject to constraints on deformation, lift, and drag. The Euler equations were solved on the fluid side while the wing was modeled by three surfaces of linear-elastic shell elements. Their work demonstrated the potential importance of considering the real FSI loads in an optimization.

Around the same time Krog et al. [10] reported on the optimization of the wing box ribs of the

Airbus A380. To reconcile the local and redundant nature of this component with the typical objectives of topology optimization (minimum compliance for given available material) a fully decoupled strategy was adopted, i.e. the fluid loads were assumed to be constant. Nevertheless, weight savings of approximately 1000kg were achieved. Recently in [11], also assuming constant loading, an example is given of large-scale (600k variables) topology optimization of compliance and linear bucking of the common research model (CRM) wing box. However, the largest published example of decoupled topology optimization is ref. [12], where a wing box was discretized by one billion structural elements. A review of similar decoupled applications is given in [13].

Later, in [14], and also with coupled FSI, topology optimization was applied to a morphing airfoil. The actuating mechanism, actuating points, and hinge locations were determined such that the lift to drag ratio is maximized subject to constraints on mass and pitching moment. The flow was modeled by the Euler equations and the structure discretized with hyper-elastic solid elements.

In [15] the distribution of composite-material reinforcement on the membrane wing of a micro air vehicle was optimized. Low-fidelity modeling of the fluid and membrane was adopted to reduce computational cost. The strong coupling and non-linear response of the structure was exploited to achieve load-augmentation or alleviation. Still using low-fidelity modeling, the same authors [16] optimized the aeroelastic stability characteristics of a plate-like wing showing that a maximum stiffness design is not always optimum. The same authors and Kobayashi [17] then optimized flapping wings and their actuation mechanisms using a cellular-division approach rather than the density approach. More recently, a level-set topology optimization method was used on an aeroelastic stability optimization problem [18] to avoid some of the disadvantages of the density approach.

In 2014 James, Kennedy, and Martins ([6]) performed concurrent shape and topology optimization of the CRM wing box, considering potential flow and linear elasticity. Optimizing both sets of variables simultaneously allowed them to achieve better results than taking a sequential approach. Shortly after, in [19], almost the same problem was considered, but using a level-set method instead of the density approach and not including any shape variables. The same problem has recently been solved with a finer structural mesh in [20], which proved to be important to allow the appearance of truss-like structures (this has been taken to the extreme in the one billion element example mentioned above). In fact, mesh-dependency effects are well known in topology optimization and have motivated adaptive mesh strategies.

In aerodynamic applications, where loads are distributed, authors have also resorted to approximate representations of the surface, e.g. by limiting the lower bound on the stiffness of surface elements, to avoid unreasonably large meshes while still ensuring the surface loads can be transferred effectively to the internal structural features. It will be shown in this thesis that this is one of the greatest challenges in applying topology optimization to FSI problems.

The issue of connecting the surface to the main load-bearing structures is eliminated if the entire structure is solid. This has been explored recently in [21], where two-material topology optimization was used to design wind tunnel models with specific flutter characteristics.

In summary, topology optimization has been used for classical minimum weight design problems and also to explore load alleviation or augmentation concepts (passive and active), or to improve stability characteristics. It has been used on conceptual 2D studies and on large-scale realistic 3D problems, mostly making use of the density approach. However, due to its computational cost, inviscid and often low-fidely fluid models are used, this is a disadvantage as a practical wing design should include viscous drag and additional aerodynamic constraints, such as buffet [22].

Since a direct comparison has yet to be presented, it is an open question if this more complex design space can significantly improve the state of the art in very flexible wing design. The work of Maute, Stanford and their associates does show that topology optimization can be used beyond minimum weight design to produce load alleviation or augmentation (passively or actively), and improved aeroelastic stability characteristics.

While consensus seems to exist around the importance of fully coupled FSI simulation and optimization, only James et al. [6] have so far optimized shape and topology simultaneously using a limited parameterization of the shape (spanwise twist distribution). Moreover, lower-fidelity models are often used to reduce the computational cost, and only inviscid fluid models have been used. Indeed one of the disadvantages of topology optimization is the larger number of iterations it requires, efficient optimization strategies are needed to reduce the cost.

This thesis will investigate means to perform concurrent shape and topology optimization of very flexible wings using high-fidelity FSI modeling, accounting for structural non-linearities, and will aim to apply the methods to relatively large-scale problems. This work will be carried within the SU2 open-source framework [23] building upon its FSI and discrete adjoint capabilities [24].

1.2 PDE-constrained optimization

Computational optimization problems are common in contexts such as optimal control, model calibration, engineering design, etc. due to the availability of mathematical models for the underlying physical phenomena. When the phenomena are modeled by partial differential equations (PDE), the optimization problem is said to be PDE-constrained. Two major approaches can be identified to solve this class of optimization problem. One only requires the PDE constraints to be satisfied for the optimal (final) solution, whereas the other requires them to be satisfied at every step of the optimization. The former may fall in either the AAO (all at once) or SAND (simultaneous analysis and design) categories, depending on how multidisci-

plinary coupling constraints are handled [25]. To a large extent, in this class of methods (called one-shot by some authors) the optimizer is also the PDE solver.
The second approach, where PDE's are satisfied in each iteration, is more common due to the availability of mature solvers for common primal problems (e.g. the Navier-Stokes equations) and general optimizers. Moreover, it does not require the optimizer to have any knowledge of the PDE constraints, which allows optimizers and solvers to be developed independently. Most mathematical programming methods used in this approach (which in multidisciplinary contexts falls in the broader "discipline feasible" category) require the gradient of the objective and constraint functions with respect to the design parameters. In effect, gradient based methods are the only effective alternative for general large scale optimization, though simpler alternatives do exist based on heuristic optimality criteria for some types of topology optimization problem.

Adjoint methods are a common strategy to obtain gradients of functions whose evaluation involves the solution of a PDE problem, especially when the number of design parameters is large relative to the number of functions. The discrete adjoint method is based on the discretized primal problem. Its continuous counterpart is based on the differentiation of the primal PDE's and subsequent discretization/solution of an adjoint PDE problem. The derivatives obtained with the discrete adjoint method do not suffer from discretization errors (in the sense that they operate directly on algebraic equations) and as will be shown, it is possible to derive generic discrete adjoint approaches for multiphysics problems. By definition, continuous adjoint methods need to target a specific primal problem and there are further restrictions on the type of function that can be differentiated. This is not to say that the discrete approach is free of challenges, nor that the continuous approach is without merit. For one a continuous adjoint solver is a PDE solver akin to the primal, from which efficient implementation strategies can be borrowed. It will be seen later that this is not always the case for discrete adjoints, and that the trade-off between generality and efficiency motivates different variations of the method.

Although the discrete adjoint approach promises generality, it is most often presented in a specific context by deriving solution methods for particular methods and types of multiphysics problem. For example, in [7, 26] the authors present a framework for simultaneous FSI and conjugate-heat-transfer (CHT), designed especially to couple a flow solver to a thermoelastic solver. Part of this specificity is carried from hand-coded implementation strategies (which are method-specific), which are used, for example, in FUN3D [27]. A similar example for pure CHT problems is presented in [28], there the authors also incorporate density-based topology optimization variables. Indeed, the differentiation of multiphysics solvers for topology optimization is common [19, 29, 30]. However, in such applications solvers tend to be of the monolithic type (for example so that locations of the domain can be either fluid or solid), or the discrete adjoint methodology is developed specifically for the primal methods used. Algorithmic (or automatic) differentiation lies on the other end of the spectrum of differentiation techniques, however, despite being generic, efficiency concerns require the code to be modified in ways that are specific to the methods. For example, in [31] the open-source software OpenFOAM [32] was differentiated using AD, applying it to the entire CHT solver and using a checkpointing strategy to reduce memory usage. In SU2 (and thus this work), AD is also applied to entire solvers, however the main strategy used to reduce memory usage is preaccumulation [33, 34].

As mentioned above, given the increasing interest in more complex problems, it is highly relevant to propose generic approaches that will allow easy extension as other combinations of physical disciplines become relevant (for example FSI combined with heat transfer). Although numerous examples of purpose-built discrete adjoint methods could be found in the literature, such a flexible approach, adaptable to multiple kinds of partitioned multiphysics problems, has not been developed so far.

1.3 Research goals

This project will investigate means to perform concurrent shape and topology optimization of very flexible wings using high-fidelity FSI modeling, accounting for structural non-linearities and viscous fluid effects, and will aim to apply it to relatively large-scale problems. Furthermore, the focus will be on aerodynamic-driven optimizations, where the structural topology will have a direct influence on the aerodynamic performance, instead of its more common use as a structural design tool. While there are examples of large-scale aerostructural shape optimization using high-fidelity models, this is yet to be done with topology optimization. One of the impediments is the computational cost, noting that topology optimization requires at least $O(10)$ more iterations, and larger structural meshes. Therefore, the performance of software and methods will be an important aspect of enabling this work. Also from a software implementation perspective, this work (which will be carried within the SU2 open-source framework [23]) will aim to develop foundations to facilitate further high performance developments. That is, simplify the process of introducing new physics, new parameters, new functions, etc., while taking advantage of modern high performance computing systems, with minimal programming effort. These goals can be summarized in the following questions:

1. Is there a potential benefit in using internal topology to influence aerodynamic properties?
2. What are the challenges in combining shape and topology optimization?
3. Are there aspects of aerostructural topology optimization for which high-fidelity models are indispensable?
4. Besides computational cost, what challenges does the application and models introduce?
5. Is it possible to mitigate this cost?

6. What implementation strategies should be used to allow extending this work to other multiphysics problems with relative ease?

7. What are the trade-offs associated with such general strategies?

1.4 Thesis layout

Having introduced the main research goals of this thesis (aerostructural topology optimization and differentiation of generic multiphysics problems), the relevant numerical methods for simulation of FSI are presented in chapter 2, along with the topology optimization strategies used for the main numerical examples in this work (which are presented in chapter 5). Chapter 3 is dedicated to discrete adjoint methods, while it reviews the most common solution methods and implementation strategies, the focus of the chapter is on generic approaches that can be easily extended to new physical phenomena, new design strategies, etc. These, of course, have some limitations which are described in that chapter, and for which this work has proposed solutions. Chapter 4 focuses on efficient software implementation aspects of low-order unstructured solvers (used in this work), whose application to SU2 was necessary to allow solving the numerical examples of chapter 5 (note how performace is an ever-present limiting factor in the topology optimization examples listed above).

1.5 Publications

The work from this thesis has resulted in the journal and conference publications listed below.

Journal publications

P. Gomes and R. Palacios, “Aerodynamic-driven topology optimization of compliant airfoils,” *Structural and Multidisciplinary Optimization*, vol. 62, 2020.

O. Burghardt, P. Gomes, T. Kattmann, T. D. Economon, N. R. Gauger, and R. Palacios, "Discrete adjoint methodology for general multiphysics problems - A modular and efficient algorithmic outline with implementation in an open-source simulation software," *Structural and Multidisciplinary Optimization*, 2021.

P. Gomes and R. Palacios, "Pitfalls of Discrete Adjoint Fixed-Points based on Algorithmic Differentiation," *AIAA Journal*, 2021.

[pending] P. Gomes and R. Palacios, “Aerostructural Topology Optimization using High Fidelity Modeling,” *Structural and Multidisciplinary Optimization*.

Conference publications

C. Venkatesan-Crome, P. Gomes and R. Palacios, "Optimal Compliant Airfoils Using Fully Non-Linear FSI Models," in *AIAA Scitech 2019 Forum*, January 2019

P. Gomes and R. Palacios, "Aerodynamic Driven Multidisciplinary Topology Optimization of Compliant Airfoils," in *AIAA Scitech 2020 Forum*, January 2020.

O. Burghardt, N. R. Gauger, P. Gomes, R. Palacios, T. Kattmann and T. D. Economon, "Coupled Discrete Adjoints for Multiphysics in SU2," in *AIAA Aviation 2020 Forum*, June 2020.

P. Gomes, T. D. Economon and R. Palacios, "Sustainable High-Performance Optimizations in SU2," in *AIAA Scitech 2021 Forum*, January 2021.

P. Gomes and R. Palacios, “Aerostructural Topology Optimization using High Fidelity Modeling,” in *AeroBest 2021*, July 2021.

Chapter 2

Methodology

This chapter starts with a description of the computational models and methods used for the fluid-structure interaction (FSI) problems in this work. By and large, these methods were already part of SU2, nevertheless, some adaptations were necessary, and they are usually discussed at the end of each subsection. Moreover, the software implementation of those methods benefited from substantial refactoring, which will be detailed in chapter 4. The second part of this chapter focuses on the structural topology optimization strategies implemented by the author in SU2, and used for the numerical examples of chapter 5.

2.1 Partitioned simulation of fluid-structure interaction

FSI problems arise when a fluid medium and a deformable (or moving) solid medium interact over a common interface. They are of special interest when the characteristics of the media provide for strong mutual influence. Simulation of FSI, involves the numerical solution of mathematical descriptions of the physics within each medium, subject to coupling conditions at their interface.

Two major solution approaches can be identified. In the monolithic approach, both sets of governing equations are jointly discretized, typically using a single scheme (i.e. finite element method), accounting for the coupling conditions, and the solution for fluid and structural variables is sought simultaneously. The partitioned approach makes use of independent solution methods for the individual problems, thereby transferring the onus of enforcing the coupling conditions to a coupling method.

Evidently the partitioned approach is attractive from the development point of view, as it can leverage existing technology (even in the form of black-box solvers) whereas the monolithic approach requires the development of specialized fluid-structure solvers. However, partitioned formulations may not ensure energy preservation and thus become unstable for time-domain problems [35], furthermore, the simulation of incompressible flows may require a special interface treatment if the boundary conditions are such that the volume of fluid must remain constant [36]. The monolithic approach is advantageous when problems exhibit these characteristics, hence it is commonly used in biomechanical applications (e.g. arterial flows [37]). For a review of the two approaches see e.g. [38]. For the steady-state, compressible, external flow problems of interest to this thesis, the partitioned approach (which is the one implemented in SU2) is by far the most common choice.

In this approach, as the solid domain undergoes deformation, so does the fluid domain, i.e. the interface moves. The typically-adopted Lagrangian specification of the structural displacement field, makes it straightforward to track the nodes of the structure as it deforms. Conversely, the Eulerian specification of the flow field creates two possible ways of accounting for the interface movement. In the arbitrary Lagrangian-Eulerian (ALE) formulation of the fluid flow equations, the fluid grid is allowed to move, and thus in time accurate simulations the grid velocities need to be considered in the discretization of convective fluxes. The grid is deformed based on the structural deformation, such that the two domains do not overlap. Some authors treat this deformation as a third field of the FSI problem (with its own governing equations), whereas others consider it part of the fluid problem. No significant changes are required to wall boundary conditions.

Alternatively, in immersed boundary (IB) methods (initially developed by Charles Peskin [39]), the fluid grid is fixed and the presence of the structure within it is modeled by source terms in the momentum equations. The use of a fixed fluid grid avoids the problem of maintaining good grid quality as it is deformed (a concern also in non FSI shape optimization problems), and may permit the use of very efficient structured solvers. Moreover, this way of modeling solids within fluids is notably suitable for topology optimization problems, see e.g. the work of [40], [41], or [42]. However, accurately modelling near wall flow at high Reynolds number (an indispensable capability for the prediction of viscous drag) is a non trivial task in IB methods (see e.g. [43]). In [44] overlapping fluid grids were used, one fixed the other deforming, to combine the advantages of both approaches, this comes at the expense of an additional layer of coupling between the fluid domains. The three-field ALE formulation adopted in SU2 (see [45]) is analyzed in the remainder of this section.

2.1.1 Fluid problem

In the fluid domain the flow is governed by the continuity, Navier-Stokes, and energy conservation equations, which in the ALE formulation may be written as

$$\frac{\partial \mathbf{w}}{\partial t} + \nabla \cdot \mathbf{F^c}_{\mathbf{ALE}}(\mathbf{w}) = \nabla \cdot \mathbf{F^d}(\mathbf{w}) + \mathbf{Q}(\mathbf{w}), \tag{2.1}$$

where $\mathbf{w} = (\rho, \rho\mathbf{v}, \rho E)$ is the vector of conservative variables. The convective and diffusive fluxes, and the volumetric sources are defined, respectively, as

$$\mathbf{F^c}_{\mathbf{ALE}}(\mathbf{w}) = \begin{pmatrix} \rho(\mathbf{v} - \dot{\mathbf{z}}) \\ \rho\mathbf{v} \otimes (\mathbf{v} - \dot{\mathbf{z}}) \\ \rho E(\mathbf{v} - \dot{\mathbf{z}}) \end{pmatrix}, \tag{2.2}$$

$$\mathbf{F^d}(\mathbf{w}) = \begin{pmatrix} \mathbf{0} \\ \overline{\overline{\boldsymbol{\tau}}} \\ \kappa\nabla T \end{pmatrix}, \tag{2.3}$$

$$\mathbf{Q}(\mathbf{w}) = \begin{pmatrix} 0 \\ -\nabla p \\ \nabla \cdot (\overline{\overline{\boldsymbol{\tau}}} \cdot \mathbf{v} - p\mathbf{v}) \end{pmatrix}, \tag{2.4}$$

where ρ is the density, $\mathbf{v}$ and $\dot{\mathbf{z}}$, the flow and grid velocities respectively (in a Cartesian coordinate system), p the pressure, E the total energy per unit mass, $\overline{\overline{\boldsymbol{\tau}}}$ the stress tensor, κ the thermal conductivity and T the temperature (all material properties refer to the fluid).

For a Newtonian fluid, and ignoring bulk viscosity effects, the stress tensor is given by

$$\overline{\overline{\boldsymbol{\tau}}} = \mu \left(\nabla\mathbf{v} + \nabla\mathbf{v}^T - \frac{2}{3}\overline{\overline{\mathbf{I}}}(\nabla \cdot \mathbf{v}) \right). \tag{2.5}$$

Furthermore, under the Boussinesq hypothesis, turbulence is modeled as increased viscosity, that is,

$$\mu = \mu_d + \mu_t, \tag{2.6}$$

with μ_d and μ_t the dynamic and turbulent viscosities, respectively. The latter is determined by a suitable turbulence model (e.g. Menter's Shear Stress Transport (SST) model [46]). Using the definition of Prandtl number (Pr), the total thermal conductivity is given as a function of viscosity and specific heat at constant pressure (C_p) as

$$\kappa = C_p \left(\frac{\mu_d}{\mathrm{Pr}_d} + \frac{\mu_t}{\mathrm{Pr}_t} \right). \tag{2.7}$$

Pressure and temperature are related with the conservative variables via a suitable equation of state, e.g. for an ideal gas with gas constant R and ratio of specific heats γ it is

$$p = (\gamma - 1)\rho \left(E - \frac{1}{2}\mathbf{v} \cdot \mathbf{v} \right), \quad T = \frac{p}{\rho R}.$$

In SU2, equation (2.1) is integrated in space, using a vertex-based finite volume method (FVM), on a median-dual grid, which results in the following semi-discrete integral equation for each volume [23]

$$\int_{\Omega_i} \frac{\partial \mathbf{w}}{\partial t} d\Omega + \sum_{j \in \mathcal{N}(i)} (\tilde{\mathbf{F}}^{\mathbf{c}}_{ij} + \tilde{\mathbf{F}}^{\mathbf{d}}_{ij}) A_{ij} - \mathbf{Q}|\Omega_i| = \int_{\Omega_i} \frac{\partial \mathbf{w}}{\partial t} d\Omega + \mathbf{R}_i(\mathbf{w}) = \mathbf{0}, \tag{2.8}$$

where $\tilde{\mathbf{F}}_{\mathbf{ij}}$ are the discretized fluxes projected onto the normal of the face (of area A_{ij}) associated with edge ij (which connects nodes i and j), $|\Omega_i|$ is the volume of the dual control volume i, and $\mathcal{N}$ represents the set of its neighbours (i.e. the nodes directly connected to it). The convective fluxes are computed using some numeric scheme, typically either Jameson-Schmidt-Turkel (JST, with scalar or matrix dissipation) or an approximate Riemann solver such as Roe's method. The viscous fluxes are computed from the average of nodal gradients, with support of the directional derivative to avoid decoupling [47]. These gradients, which are also used for MUSCL reconstruction of the variables (for second-order upwind convective methods), are computed using either the weighted least squares method, or making use of the Green-Gauss (divergence) theorem in discrete form. The spatially integrated terms in (2.8) are commonly denoted the residual, $\mathbf{R}_i$.

Equation (2.8) can be integrated in time either explicitly (typically using a Runge-Kutta method) or implicitly, by approximating the time derivative by a backward difference formula (usually second order). For dual-time integration, or when a steady state solution is sought, the pseudo-time (τ) is first introduced, and the solution is then marched in this variable to the end of the time step, or to steady state, using first-order implicit-Euler integration. That is,

$$\frac{|\Omega_i|}{\Delta\tau}(\mathbf{w} - \mathbf{w}^{n-1}) + \mathbf{R}^*_i(\mathbf{w}) = \mathbf{0}, \tag{2.9}$$

where $\mathbf{R}^*_i$ is the (possibly) modified residual if accurate time integration is desired. For simplicity, the modification has been omitted as it can be readily found in the aforementioned literature. Linearizing equation (2.9) about $\mathbf{w}$ results in a quasi-Newton solution process, requiring, at each iteration, the solution of a linear system of equations with the Jacobian of the residual. Namely,

$$\left(\frac{|\Omega_i|}{\Delta\tau_i}\delta_{ij} + \frac{\partial \mathbf{R}^*_i}{\partial \mathbf{w}}\right)(\mathbf{w}^{n+1} - \mathbf{w}) = -\mathbf{R}^*_i. \tag{2.10}$$

It is common for the solution of this equation to be approximate in two ways. First, and more obvious, the linear system is solved to a relatively large tolerance ($O(0.1)$), i.e. it is smoothed. Second, when the linearization (Jacobian) is stored (as a sparse matrix) it is only approximate. For example, by considering only the derivatives with respect to immediate neighbors, thereby increasing sparsity. Furthermore, for central schemes it is common to artificially increase the Jacobian of numerical dissipation terms to improve the diagonal dominance of the matrix (and thus its condition number). If products with the Jacobian are obtained in a matrix free manner, sparsity is no longer a concern. This requires a Krylov linear solver (typically GMRES), and thus forms the basis for Newton-Krylov methods. Nevertheless, the condition number may still need to be improved to reduce the cost of solving the linear systems. Although the pseudo-time increment can be set arbitrarily low, to improve the numerical properties of the matrix, for the purpose of obtaining a steady-state solution it is more efficient, up to a point, to act on the linearization of the residual. One common acceleration technique is to determine a local, for each volume, pseudo-time step based on a target Courant-Friedrichs-Lewy (CFL) number. This local CFL may also be defined locally to improve stability, for example based on the local rate of residual reduction, or on an automatic limiting of the rate of change of the solution [48]. In SU2, a geometric multigrid method may also be used to accelerate the solution, the solution of the linear system then becomes the multigrid smoothing operator. In what follows, the fluid solution process is represented by the fixed-point iteration

$$\mathbf{w} = \mathcal{F}(\mathbf{w}, \mathbf{z}). \tag{2.11}$$

The turbulence model equations are solved in the same manner but, in SU2, they are lagged. Nevertheless, for the purposes of the fixed-point representation, the turbulence variables can be considered part of $\mathbf{w}$.

The compressible flow solver in SU2 has been verified and validated for a number of external flow applications [4, 23, 48].

2.1.2 Structural problem

The deformed state of a solid domain, Ω, is governed by the point-wise equilibrium of linear momentum and of tractions on its surface (Γ), that is,

$$\rho(\ddot{\mathbf{u}} - \mathbf{f}) - \nabla \cdot \overline{\overline{\boldsymbol{\sigma}}} = \mathbf{0} \quad \text{in } \Omega, \tag{2.12}$$

$$\overline{\overline{\boldsymbol{\sigma}}}\mathbf{n} - \boldsymbol{\lambda} = \mathbf{0} \quad \text{on } \Gamma, \tag{2.13}$$

where ρ is the density, $\ddot{\mathbf{u}}$ the acceleration with respect to an inertial frame, $\mathbf{f}$ the body forces, $\overline{\overline{\boldsymbol{\sigma}}}$ the Cauchy stress tensor, $\mathbf{n}$ the outward surface unit normal, and $\boldsymbol{\lambda}$ the external tractions. To solve equation (2.12), for the structural displacements via the finite element method (FEM), its weak form is first established and the stresses related to the displacements via kinematics and material constitutive models, the resulting equations are then discretized.

Applying the principle of virtual work, the problem can be stated as finding the structural displacements $\mathbf{u}$ that respect the essential boundary conditions, such that, for an arbitrary virtual displacement, $\delta\mathbf{u}$, vanishing at essential boundaries, the total virtual work is zero [49], that is,

$$\int_{\Omega_c} \delta\mathbf{u} \cdot \rho(\ddot{\mathbf{u}} - \mathbf{f}) d\Omega + \int_{\Omega_r} \delta\overline{\overline{\boldsymbol{E}}} : \overline{\overline{\boldsymbol{S}}} d\Omega - \int_{\Gamma_c} \delta\mathbf{u} \cdot \boldsymbol{\lambda} d\Gamma = 0, \tag{2.14}$$

were $\delta\overline{\overline{\boldsymbol{E}}}$ is the variation of the Green-Lagrange strain tensor with respect to $\delta\mathbf{u}$, $\overline{\overline{\boldsymbol{S}}}$ is the second Piola-Kirchhoff stress tensor, Ω_r refers to the reference (undeformed) configuration of the structure, and Ω_c and Γ_c to the current configuration of the structure and its surface. The Green-Lagrange strain tensor is a Lagrangian measure of finite strain given by

$$\overline{\overline{\boldsymbol{E}}} = \frac{1}{2}(\overline{\overline{\boldsymbol{C}}} - \overline{\overline{\boldsymbol{I}}}), \tag{2.15}$$

where $\overline{\overline{\boldsymbol{C}}}$ is the right Cauchy-Green deformation tensor, computed from the deformation gradient $\overline{\overline{\boldsymbol{F}}}$ as

$$\overline{\overline{\boldsymbol{C}}} = \overline{\overline{\boldsymbol{F}}}^T \overline{\overline{\boldsymbol{F}}} \tag{2.16}$$

,

$$\overline{\overline{\boldsymbol{F}}} = \overline{\overline{\boldsymbol{I}}} + \nabla_X \mathbf{u}, \tag{2.17}$$

where X are the coordinates in the reference configuration. For hyperelastic materials the relation between stress and strain is given by the strain energy density function Ψ as

$$\overline{\overline{\boldsymbol{S}}} = \frac{\partial \Psi}{\partial \overline{\overline{\boldsymbol{E}}}}. \tag{2.18}$$

For example, for a neo-Hookean material with Lamé constants μ and λ, and defining $J = \det\overline{\overline{\boldsymbol{F}}}$,

$$\Psi = \frac{\mu}{2}\left(\mathrm{tr}\overline{\overline{\boldsymbol{C}}} - 3\right) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2. \tag{2.19}$$

Equation (2.14) is linearized around the current state of deformation before being discretized. The implementation in SU2 (described and verified in [45]) uses isoparametric linear solid elements and it is largely based on [50]. The solution is then found iteratively via the Newton-Raphson method (the Jacobian matrix of this problem is called the tangent stiffness matrix), which again can be formulated as a fixed-point iteration, namely

$$\mathbf{u} = \mathcal{S}(\mathbf{u}, \boldsymbol{\lambda}(\mathbf{w}, \mathbf{z})). \tag{2.20}$$

2.1.3 Mesh deformation

The deformation of the fluid-structure interface must be transferred to the fluid grid, in a way that maintains its quality. Mesh deformation strategies are usually preferred to re-meshing due to their simpler implementation, higher computational efficiency, and differentiability (which is important in optimization contexts). Several strategies have been proposed:

- *Interpolation techniques*, for example, based on radial basis functions (RBF), can be used to smoothly spread the interface displacements to the interior nodes ([51]). When basis functions with compact support are used, the approach is attractive from a computational cost perspective. However, it lacks fine control over how certain areas of the domain may be differently affected by the boundary displacements. Moreover, the author's experience with using RBF's for mesh deformation, is that to maintain acceptable quality, the radius should be proportional to the deformation, which can negate the effect of compact support.

- *Diffusion analogies* (e.g. [52]) solve a Poisson equation for the displacements with Dirichlet boundary conditions at the interface, other types of boundary condition may be used for other fluid boundaries, which allows, for example, sliding planes to be implemented. A spatially varying "diffusion" coefficient may be used to avoid excessive deformation of small control volumes, Helenbrook [53] explores the use of the biharmonic operator to improve mesh quality near boundaries (since it allows Dirichlet conditions on the gradients). The approach is relatively expensive but incompressible flow solvers will in general have fast solvers for this type of equation.
- The *spring analogy*, in which the edges of the fluid grid are considered linear springs. Torsional springs may also be added to help preserve orthogonality close to solid walls. Variable spring stiffness (e.g. inversely proportional to edge lenght) is commonly used to keep edges from collapsing [54]. High computational cost can be avoided by using the method in an explicit manner.
- The *solid elasticity analogy*, consists in solving the equations of solid mechanics for the entire fluid domain. This approach also permits different boundary conditions to be used, and the control volumes (elements in this context) may also be given a local stiffness, for example, inversely proportional to their volume or distance to deforming boundaries. This approach carries the highest computational cost. Consider, for example, that the required level of refinement near viscous walls for RANS problems is unnecessary for an elasticity problem. Furthermore, elasticity problems are notorious for being numerically stiff, and thus difficult to scale. Note that strong linear preconditioners and solvers lose some efficiency with increased parallelization, which can make this deformation approach impractical for large meshes. Nevertheless, in an FSI context, a solver for this type of equation is likely to be available, which simplifies the implementation. This is the approach used in SU2.

A common strategy to all these approaches, when they are used outside the FSI context, i.e. for shape optimization, is to apply the displacement in increments as a way to achieve better mesh quality (thereby making the mesh deformation operation nonlinear). For FSI problems, this application in increments is inherently present due to the iterative nature of the coupling algorithms. However, this may very easily result in a final fluid mesh that depends on convergence settings. For example, if the solid mechanics analogy is used, and a stiffness distribution inversely proportional to element volume is computed every iteration, the final mesh will depend on the convergence history, for once an element is compressed it will not recover its initial shape. Therefore, a desirable property of the deformation operation, is for the deformed grid to be defined exclusively by the initial node coordinates and the boundary deformation $\mathbf{u}_\Gamma$ (in keeping with the elasticity analogy, the mesh deformation should be elastic rather than plastic). That is,

$$\mathbf{z} = \mathcal{M}(\mathbf{u}_\Gamma). \tag{2.21}$$

Moreover, as will be seen, not depending on the deformed coordinates (at any iteration) is important when deriving a steady-state adjoint FSI solver. Finally, it is worth noting that no single deformation strategy is best, while physical analogies become expensive for large scale problems, explicit approaches struggle with maintaining mesh quality when deformations are large. Therefore, multiscale approaches have been proposed, where an expensive strategy is applied to a coarse representation of the mesh, and an algebraic approach is then used within each coarse unit [55]. However, such strategies are difficult to implement for unstructured meshes, and not fundamental for the size of the problems studied in this work, they will, however, have to be pursued for full aircraft RANS simulations. Of course, tuning the elasticity analogy was still crucial for this work. In particular, ensuring the linear systems are initialized to reduce their cost as the problem converges, and also reducing their overall cost by improving the numerical properties of the stiffness matrix. The simplest way to achieve this is by lowering the Poison's ratio, more intricate strategies involve clamping all inner nodes that are far away from the deforming surfaces, and avoiding excessively high local stiffness as wall distance or element volume approach zero.

2.1.4 Coupling conditions

At the interface between domains (Γ), the fluid grid velocity ($\dot{\mathbf{z}}$) is equal to the velocity of the structure's surface ($\dot{\mathbf{u}}$), and the solid and fluid stresses are in equilibrium, that is

$$\begin{aligned} \dot{\mathbf{z}}_\Gamma &= \dot{\mathbf{u}}_\Gamma, \\ \overline{\overline{\boldsymbol{\sigma}}}\mathbf{n}_s + \left(-p\overline{\overline{\boldsymbol{I}}} + \overline{\overline{\boldsymbol{\tau}}}\right)\mathbf{n}_f &= \mathbf{0} \quad \text{on } \Gamma. \end{aligned} \tag{2.22}$$

From the fluid's perspective the interface is a wall, for which the no-slip condition dictates $\mathbf{v}_\Gamma = \dot{\mathbf{z}}_\Gamma$. For time accurate FSI simulations the grid velocities at the internal nodes can be computed via a finite difference formula, or using the fluid grid deformation operation if it is linear (since otherwise the grid velocity would not be consistent with the deformation). Grid velocities are ignored for steady state simulations.

The coupling conditions simply state that the interface moves according to the structural deformation, and that the structure is subjected to the fluid forces. As the grid size requirements are usually not the same for both problems, the grids will usually not match at the interface. Therefore, suitable interpolation strategies must be used to transfer the displacements onto the fluid side, and the stresses onto the structural side.

The interpolation of displacements and stresses between discrete locations can be generalized by introducing interpolation matrices, $\mathbf{H}$, such that

$$\begin{aligned}\Delta \mathbf{z}_\Gamma &= \mathbf{H}_{sf}\mathbf{u}_\Gamma,\\ \boldsymbol{\lambda}_s &= \mathbf{H}_{fs}\boldsymbol{\lambda}_f \quad \text{on } \Gamma.\end{aligned} \tag{2.23}$$

The interpolation of displacements should be smooth, which bars the use of nearest neighbour interpolation, instead, RBF (e.g. [56]) or isoparametric (e.g. [57]) interpolation are suitable techniques. For tractions, nearest neighbour interpolation may also be used as the response of the structure will in general be smooth. A common alternative, however, is the use of conservative interpolation, whereby setting $\mathbf{H}_{fs} = \mathbf{H}_{sf}^T$, and transferring discrete forces rather than tractions, results in equal work on both sides of the interface. That is,

$$\boldsymbol{\lambda}_s \cdot \mathbf{u}_\Gamma = \boldsymbol{\lambda}_f \cdot \Delta \mathbf{z}_\Gamma. \tag{2.24}$$

Although this is an attractive property, in areas where the structural mesh is finer that the fluid (which is common for topology optimization), the conservative approach creates a non-physical representation of the loads, where only some structural nodes are loaded. In turn, this may result in non-physical deformations of the surface, which may be exacerbated due to the coupling with the fluid (e.g. due to shocks). Therefore, in these cases, the author has found it better to use an interpolation technique consistent with the one used for displacements. Note that unless the matrices are square, it is not possible for the interpolation to be both consistent and conservative, since a rectangular matrix cannot have both its columns and its rows adding to one.

It is worth noting a particularly pernicious implementation aspect of these interpolation techniques, which had to be addressed as part of this work. Namely, it is necessary to ensure that the interpolation is not sensitive to domain decomposition, and to node renumbering strategies. For example, in nearest neighbor interpolation, a structural node should be mapped to the same fluid node, regardless of how these are ordered. This means they should be ranked not only by distance, but also by some invariant that can be used to break ties between equidistant candidates. One such invariant is the global index of the node, i.e. prior to partitioning and renumbering. Similarly, some radial-basis functions lead to matrices with high condition number, hence sensitive to floating-point errors. It is advisable, therefore, to give an invariant order to the nodes before preparing the RBF matrices.

2.1.5 Partitioned solution methods

The role of a coupling method is to ensure conditions (2.22) at convergence. Based on a Dirichlet-Neuman (D-N) decomposition, the coupling problem can be stated as finding interface displacements, such that the interface tractions are in equilibrium. Two operators can be defined from the fluid and structural fixed-points, by noting that under a suitable level of convergence, the fluid solver can be considered the D-N operator $\mathbf{F}$ (which maps displacements to tractions), and the structural solver the N-D operator $\mathbf{S}$ (which does the opposite), i.e. $\boldsymbol{\lambda}_f = \mathbf{F}(\mathbf{u}_\Gamma)$ and $\mathbf{u}_\Gamma = \mathbf{S}(\boldsymbol{\lambda}_f)$. In terms of the operations described in this section, the fluid operator consists in first transferring the structural displacements to the fluid side, then deforming the fluid grid, and finally solving the fluid flow equation, thereby obtaining the fluid tractions. Whereas the structural operator consists in first transferring the fluid tractions to the structural side, and then solving the structural problem to obtain the displacements. These operators can be applied sequentially to produce a fixed-point iteration for the interface displacements, namely,

$$\mathbf{u}_\Gamma = \mathbf{S} \circ \mathbf{F}(\mathbf{u}_\Gamma), \tag{2.25}$$

which in practice is equivalent to a block-Gauss-Seidel (BGS) approach to solve the coupled problem. It is common, however, for this approach to require significant under-relaxation for strongly coupled problems, that is

$$\mathbf{u}_\Gamma^{n+1} = \omega \mathbf{S} \circ \mathbf{F}(\mathbf{u}_\Gamma) + (1-\omega)\mathbf{u}_\Gamma. \tag{2.26}$$

The relaxation factor, ω, is often determined by Aitken's relaxation (see e.g. [58]). Two major factors make the BGS approach popular despite its limitations, first it is well suited for coupling black-box solvers, and second, it allows amortizing the cost of deforming the mesh over multiple fluid and structural iterations. Nevertheless, other partitioned methods have been proposed to improve its convergence rate (and thus stability) or to allow the structural and fluid solvers to be executed simultaneously (see for example [59], [36]). The interface-quasi-Newton family of methods is common. These typically start with a manipulation of (2.25) to derive an interface residual, namely

$$\mathbf{R}_\Gamma(\mathbf{u}_\Gamma) = \mathbf{S} \circ \mathbf{F}(\mathbf{u}_\Gamma) - \mathbf{u}_\Gamma \equiv \mathbf{r} = \hat{\mathbf{u}}_\Gamma - \mathbf{u}_\Gamma = \mathbf{0}, \tag{2.27}$$

which is then solved in a quasi-Newton manner. That is,

$$\mathbf{u}_\Gamma^{n+1} = \mathbf{u}_\Gamma + \tilde{\mathbf{r}}_u^{-1}(-\mathbf{r}), \tag{2.28}$$

where $\tilde{\mathbf{r}}_u^{-1}$ is an approximate inverse Jacobian. Different strategies to obtain this matrix, or its action on the residual, result in different methods within the quasi-Newton family. For example, the IQN-ILS method [60] (used for some problems in this work) applies linear least squares to a window of inputs ($\mathbf{u}_\Gamma$) and outputs ($\mathbf{r}$) of the residual function to accelerate its convergence. A process not unlike those used for solution [61], or stabilization [62], of general fixed-point equations. One important practical aspect of using these methods, is that the fluid and structural problems should be solved to reasonable accuracy within each coupling iteration. Otherwise, the assumption that the interface residual is only a function of the interface variables is incorrect. Note that the BGS approach is not affected by this issue.

To allow the fluid and structural solvers to be executed simultaneously, a block-Jacobi approach can be used instead of BGS (which is sequential). That is,

$$\begin{pmatrix} \boldsymbol{\lambda}_f \\ \mathbf{u}_\Gamma \end{pmatrix} = \begin{pmatrix} \mathbf{F}(\mathbf{u}_\Gamma) \\ \mathbf{S}(\boldsymbol{\lambda}_f) \end{pmatrix}, \tag{2.29}$$

to which quasi-Newton methods can also be applied. Simultaneous, and possibly asynchronous, execution is important to make optimal use of high performance computing systems, for example the structural problem may not scale to the same extent as the fluid (which is typically larger). Moreover, the two solvers may run on different architectures, for example, the flow solver on GPU and the structural solver on CPU. This type of approach was not explored in this work, since topology optimization results in structural and fluid problems of similar cost.

2.2 Density-based topology optimization

Topology optimization is a well established technique for determining the optimal distribution of material (or materials) within an available volume. It was initially applied to structural problems ([63]) but has since been used across a broad range of disciplines. The continuous form of the problem is commonly stated as

$$\begin{aligned} \min_{\rho} \quad & f(\mathbf{u},\rho) \\ \text{subject to}: \ & \mathcal{S}(\mathbf{u},\rho) = 0 \quad \text{in } \Omega \\ & g_0(\rho) = \int_\Omega \rho \; d\Omega - V_0 \leq 0 \\ & g_i(\mathbf{u},\rho) \leq 0, \quad \text{i=1,}\cdots\text{,M} \\ & \rho \in \mathbb{Z} \cap [0,1], \quad \forall \mathbf{x} \in \Omega \end{aligned} \tag{2.30}$$

where the equality constraint represents the governing equations, g_0 is the optional constraint on the volume of solid material ($\rho = 1$) used by the design, g_i are M inequality constrains (included for generality), and ρ may either be 0 or 1 anywhere in the design domain Ω.

This formulation suggests a density-based approach, i.e. finding the function $\rho(\mathbf{x})$ that minimizes f subject to the constraints. Notwithstanding, shape optimization of the boundaries separating solid and void regions, can be a valid alternative if new boundaries (holes) can be created during the optimization (see the bubble method of [64]). The density and boundary approaches may also be viewed respectively as Eulerian and Lagrangian descriptions of the domain [65]. In the latter the governing equations only need to be respected (solved) inside the solid region, moreover, density influences them through the shape of the boundary, conversely, the Eulerian approach requires a mechanism to be included in the governing equations whereby density can signal the absence or presence of material. This is typically achieved through penalization of relevant material properties, such as the elasticity modulus in structural applications, or the thermal conductivity in heat conduction applications. One clear advantage of the Lagrangian description is the lower computational burden of solving the governing equations only where necessary. However, the Eulerian description is more common due to its simplicity, in fact, most level-set methods adopt it (e.g. [66]), despite being boundary approaches, to avoid dealing with adaptive mesh strategies.

This project aims to explore the use of topology optimization in aerostructural settings, and not to propose new topology optimization approaches. With this in mind, the density approach is the logical candidate as there is no dearth of examples of its successful application and of how to manage its shortcomings (see e.g. [67, 68]) albeit in purely structural contexts. For an in-depth review of other approaches see e.g. [65].

The discretization of the domain naturally discretizes the density function into a vector of design variables, an optimization method can then be used to determine the optimum topology. Discrete methods have been applied to directly handle the binary nature of the variables (e.g. genetic algorithms [69]), however, the number of function evaluations required by such methods makes them too expensive for large scale problems. A much more common approach is the use of continuous variables and gradient-based optimization methods (the next chapter deals with how to obtain gradients). Then, a penalization method is used to discourage intermediate values. For example, in Bendsoe's SIMP approach (Simplified Isotropic Material with Penalization [70]) a power law is used, e.g. for the elasticity modulus (E)

$$E(\rho) = E_{min} + (E_{ref} - E_{min})\rho^p \tag{2.31}$$

where E_{min} is introduced for stability of the structural solver and typically set to 0.1% of E_{ref}, the reference value of the property. It will be discussed later that this lower bound on stiffness also relaxes the optimization problem. When the exponent (p) is greater than 1, the ratio of stiffness to mass at intermediate densities is unfavorable, and thus, if this ratio is valuable under the objective and constraints of the problem, the optimum topology should be discrete. A similar approach is the rational function of [71]. In most published applications of SIMP authors use $p = 3$, higher values may be required, for example if inertial loads are significant, but should be avoided as they make convergence more difficult and increase the propensity to converge to local minima. For this reason, many authors also recommend the exponent to be ramped to achieve better results and reproducibility (e.g. when testing a different optimization algorithm). However, the author's experience is that this may sometimes drive the solution into a topology that is not optimal for the final exponent, and from which the algorithm repeatedly tries to escape, always failing. Formally, however, the penalization exponent should be based on realizability of intermediate densities via a two-phase microstructured-material, which is itself completely discrete, and obtained from an arrangement of the lower and upper bound material properties (e.g. E_{min} and E_{ref}). It is then said that the penalized material (2.31) respects the Hashin-Shtrikman bounds for two-phase materials [67], which in 2-D requires $p \geq 3$ for Poisson ratio of 1/3.

2.2.1 Challenges and solutions

In general, topology optimization problems are non-convex, which makes the results obtained via mathematical programming methods sensitive to optimization parameters such as initial values, continuation strategies, the methods themselves, etc. Many engineering problems suffer from this and, in practice, little can be done about it apart from empirically testing the sensitivity of a given problem to the optimization parameters.

A more important issue (arguably the most) is the mesh dependency created by penalizing intermediate densities. As the mesh is refined, finer and finer checkerboard patches of unpenalized intermediate stiffness appear, the condition is exacerbated by numerical issues such as the overestimated stiffness of corner contacts. Several ways of tackling the problem have been proposed which fall in two broad categories:

- *Constraints* can be added to the problem to prevent sharp oscillations of material distribution, in [72] local gradient constraints are used to avoid checkerboards, the large number of constrains makes the optimization problem extremely costly, global constraints (often implemented as penalties) are therefore proposed by several authors, see e.g. [73, 74]. Functions that directly detect corner contacts have also been proposed by [75], however, this methodology is not easily extended to unstructured grids.

- *Smoothing strategies*, inspired by image processing techniques, filter sharp variations by either pre-processing the densities (see [76]) or post-processing their derivatives (see [77]). The latter predates the former and is appealing when the adjoint code is hand built, for it dispenses differentiation of one more operation (i.e. the filter). However, density filtering poses no challenge when algorithmic differentiations is used. Moreover, it is more versatile as it can also control feature size see e.g. [78, 79]. The simplest form of the technique consists in replacing the derivative/density by a weighted average of neighbouring values. In general, filtering also ameliorates the non-convex nature of the optimization problem.

The ability of the density filtering approach to control feature size is fundamental in engineering and is a necessary condition for mesh independence (it ensures a minimum number of elements is used to discretize the smallest feature of the topology). Moreover, contrary to constraint-based approaches, it does not increase the complexity of the optimization problem, hence it was the selected approach for this project.

Simple filter kernels create a "gray" halo that is undesirable when perfectly discrete topologies are sought, in the sense that it creates an inconsistency between the model and the real world. Moreover, this increases the sensitivity to the SIMP exponent, since higher values favor topologies with smaller "wetted" area (and thus less material in halo regions). To avoid this, in [78] an approximation of the Heaviside (step) function is used in a projection step after the application of a conical-weight filter, namely

$$y(x) = 1 - e^{-\beta x} + x\ e^{-\beta}. \tag{2.32}$$

For β approaching infinity, small values of x are projected to 1. As large values of β make the gradients poorly behaved, a continuation strategy needs to be adopted, and therein lies the greatest problem of the approach, for the filtering and projection step becomes less volume preserving as β increases, consequently, the design-density field must become sparser to respect the volume constraint, slender features that developed early in the optimization may therefore become poorly represented. To avoid this, function (2.32) can be modified ($y^* = 1 - y(1-x)$) to project values towards 0 instead of 1 thereby making the design field denser, in so doing, control over minimum thickness is traded for control over minimum hole size.

To avoid these shortcomings, [79] proposed morphology-based filtering operations that can: produce solid-void designs; be nearly volume preserving; and impose both minimum thickness and minimum hole size. This is done by sequencing *dilation* (material deposition around a solid point) and *erosion* operations (material removal around a void point) whose continuous formulation is, respectively,

$$\tilde{y}_i(x) = \log\left(\frac{1}{N}\sum_{j\in\mathcal{N}(i)} e^{\beta x_j}\right)/\beta, \tag{2.33}$$

$$\overline{y}_i(x) = 1 - \tilde{y}_i(1-x), \tag{2.34}$$

where $\mathcal{N}$ is the set of N points inside the reference radius. As $\beta \to 0$ both operations reduce to the arithmetic average. Following dilation by erosion **closes** holes smaller than the reference radius, the opposite **opens** holes of at least that size. The close/open operations are similar to the dilate/erode or the heaviside "up"/"down" ones, but they are almost volume preserving which improves the convergence of the optimization (a continuation strategy for β is required nevertheless). To control minimum hole size and minimum thickness simultaneously, Sigmund suggests the close-open or open-close filters, however, the author's experience is that these worsen convergence, possibly due to the initial large effective filter radius (when β is small 4 averages are performed).

Although simple in nature, discrete filters are challenging to implement for partitioned unstructured domains with re-entrant features. If the neighbour search is based on distance alone those features may be bridged. Moreover, the filter operation will in general require values from interior elements of adjacent partitions which are not trivial to communicate (domain decomposition is discussed in a later chapter). Finally, the memory required to differentiate discrete filters with algorithmic differentiation, can be significant when the element size is small compared to the filter radius. For example, with a ratio of 1:3.5 in a 3D Cartesian grid each element has 178 neighbours (per filtering stage). It may be necessary, therefore, to analytically differentiate the filter operation. For these reasons, some authors propose the use of PDE-based filters, [80] used the Helmholtz equation, although computationally efficient, this approach does not allow the same level of control that morphology-based filters do.

It is worth noting that a major challenge of topology optimization is incorporating stress constraints, which are a staple of mechanical engineering design, but notoriously difficult to handle in gradient-based optimizations. While treating them like local constraints substantially increases the cost of the optimization, converting them to global constraints (via aggregation strategies [81]) reduces the performance of optimized designs. Topology optimization makes matters worse by creating a singularity at zero density. In particular, void material is not stressed (by definition) but as density tends to zero stress tends to a finite value. To address this, in [82] the stress constraints were replaced by quality functions, whereas [83] proposes a penalization of stresses consistent with that of stiffness and a relaxation of the stress constraints near zero density. Stress constraints will not be explicitly included in this work.

2.2.2 Solution by gradient-based optimization methods

Although the method for computing gradients will only be discussed in the next chapter, it is now worth discussing the strategy used in this work to solve topology optimization problems.

Topology optimization problems are characterized by a large number of design variables and relatively few constraints (excluding simple bound constraints), that from the physics point of view, do not need to be imposed strictly. Therefore, the exterior penalty method is herein used to impose constraints, that is, problems of the form

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & f(\boldsymbol{\alpha}) \\ \text{subject to :} \quad & g_i(\boldsymbol{\alpha}) = 0 \\ & h_j(\boldsymbol{\alpha}) \leq 0 \end{aligned}$$

become $\min\limits_{\boldsymbol{\alpha}} \hat{f}(\boldsymbol{\alpha})$ with

$$\hat{f}(\boldsymbol{\alpha}) = f(\boldsymbol{\alpha}) + \sum_{i=1}^{N_g} a_i g_i(\boldsymbol{\alpha})^2 + \sum_{j=1}^{N_h} b_j h_j^+(\boldsymbol{\alpha})^2, \tag{2.35}$$

where $h^+ = \max(0, h)$. The penalized objective function can then be minimized using an unconstrained (but bounded) optimization method. The penalty parameters (a_i and b_j) need to be gradually increased (usually by multiplying the previous value by a fixed factor r) until a predetermined small constraint tolerance is met. This creates the need for outer iterations since updating the parameters within unconstrained (inner) iterations leads to bad approximations of the Hessian matrix. Although these outer iterations force an undesired reversion to steepest descent, they are also needed (and commonly used) to update topology optimization parameters. The continuation strategy for these parameters, which was found to be adequate for both structural and FSI problems, is given in algorithm 1.

```
Data: optimization problem, $p$
Result: solved problem
set loose convergence criteria for optimizer;
while $true$ do
    UnconstrainedMinimization($p$);
    // update the penalty parameters $(a_i, b_i)$
    foreach $c$ in $p.constraints$ do
        if $c.value > tolerance$ then
            $c.penalty \leftarrow c.penalty \times r$
        end
    end
    // conditional update of other parameters (e.g. $\beta$)
    if $p.constraints.satisfied()$ then
        $p.parameters.nextvalue()$;
        if $p.parameters.finalvalue()$ then
            restore tight criteria for optimizer;
            UnconstrainedMinimization($p$);
            return $p$
        end
    end
end
```

Algorithm 1: Exterior penalty method with parameter updates.

This was implemented by the author in a new optimization framework[1], which is more versatile than what was available in SU2 (regarding definition of design variables and parameters). Note that penalty parameters are increased geometrically, whereas problem parameters are simply a sequence of values through which the algorithm advances if all constraints are currently satisfied. Before the target values of the parameters are reached, loose convergence criteria are used for the unconstrained optimizer (e.g. 40 inner iterations). The objective function is shifted and scaled by representative minimum value and range respectively, the constraints are shifted by their bounds and scaled by a reference value (the reciprocal of the bound unless otherwise specified). Doing so allows the same constraint tolerance (≈ 0.01) to be used, the penalty parameters to be initialized equally ($a_i^0, b_i^0 \in [1, 10]$), and also updated with the same factor ($r \in [1.4, 4]$).

[1] https://github.com/su2code/FADO

The unconstrained optimizer used in this work is the L-BFGS-B [84] implementation available with SciPy [85]. This choice was made during the early research stage, based on comparisons with the well know MMA method [86]. Later, the constrained optimizer IPOPT [87] was also tested, based on promising results in the literature [88], however, it performed worse than L-BFGS-B. Recently, other researchers have also found that current versions of IPOPT have difficulties with topology optimization problems [89]. One computationally advantageous aspect of the exterior penalty approach (which was not explored in this work) is that it may reduce the cost of evaluating gradients, for example, it may be possible to differentiate the penalized function at the same cost of differentiating one of the constraints or objectives.

2.2.3 Benchmark example

The strategy described above, and the implementation of the SIMP scheme in SU2, were tested by reproducing published results for the classic tip loaded cantilever problem, shown in Figure 2.1.

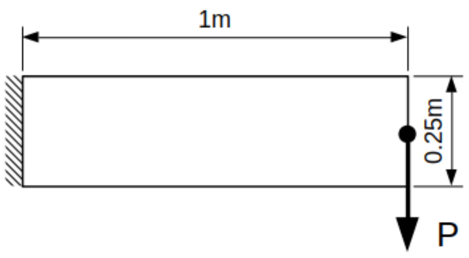


Figure 2.1: 4:1 aspect ratio tip loaded cantilever.

In particular the nonlinear topology optimization results of [90], where a material with elasticity modulus of 3 GPa and Poisson ratio of 0.4 was considered. For this test the morphology based filters were used to obtain solid-void topologies. The objective is to minimize end compliance (W_{tip}) subject to an equivalent mass constraint of 0.5, that is

$$\begin{aligned} \min_{\rho} \quad & W_{\text{tip}} \equiv \mathrm{P}\delta_y \\ \text{subject to :} \quad & \frac{1}{V}\int_V \rho \mathrm{d}\Omega \leq 0.5 \end{aligned} \tag{2.36}$$

where δ_y is the vertical displacement of the point where the load is applied. Since geometric non-linearities are considered, different topologies are expected for different values of the load (P). Referring to algorithm 1, the objective is normalized by initial value, $b_1^0 = 8$ and $r = 2$, the constraint tolerance is 0.005. The value sequence for the filter parameter β (equation (2.33)) is $\{0.01, 1, 4, 16, 64, 200\}$. The initial small value makes both morphology filters equivalent to the simpler conical filter. The loose convergence criteria for L-BFGS-B are 40 iterations or a variation of objective function value less than 10^{-5}. For the tight criteria the latter is reduced to 10^{-7} and the number of iterations unlimited. The domain is discretized with 10 000 square isoparametric elements, the filter radius (defining $\mathcal{N}(i)$ in (2.33)) is twice the element size, however note that the *close* and *open* filters are applied in two stages which effectively doubles the radius of the neighborhood. Ramping the SIMP exponent was not found to be advantageous for these simple cases, instead, a constant value of 3 was used. Because loads are large in these examples, it is not practical to start the optimization with a uniform density distribution that also respects the mass constraint, instead, the optimal density distribution for SIMP exponent of 1 and linear elasticity was used (which was obtained with the same process described above, but without increasing the filter parameter). Figures 2.2 and 2.3 show the topologies obtained for P=60 kN with the *open* filter and for P=240 kN with the *close* filter, respectively.

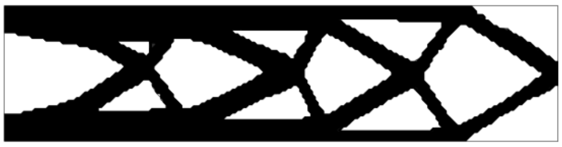


Figure 2.2: Optimized topology for 60 kN with the *open* filter.

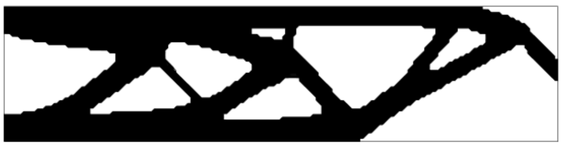


Figure 2.3: Optimized topology for 240 kN with the *close* filter.

Figure 2.4 shows the convergence history for both cases, both the obtained topologies and compliance values ($W_{\text{tip}}(60\,\text{kN}) = 4.36\,\text{kJ}$ and $W_{\text{tip}}(240\,\text{kN}) = 67.0\,\text{kJ}$) compare favorably with the reference results (4.65 kJ and 66.5 kJ respectively). The optimization process requires on average 350 function evaluations, which again compares well with other sources (e.g. [79]), and the obtained topologies after filtering are nearly perfectly discrete.

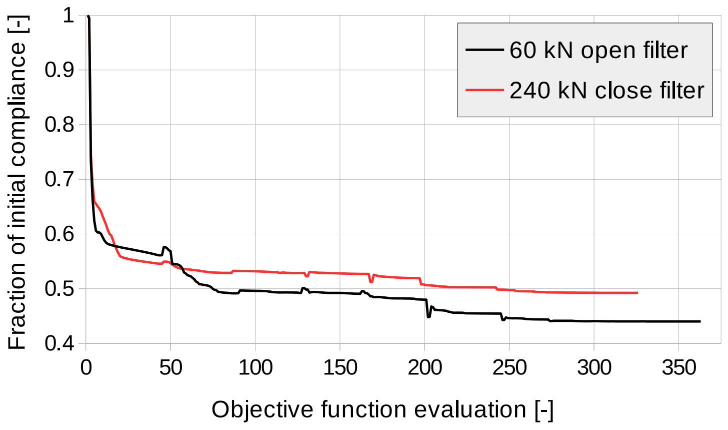


Figure 2.4: 4:1 tip loaded cantilever optimization histories for the two load values considered.

2.2.4 A strategy for discrete topologies in FSI

To obtain discrete topologies, density-based topology optimization methods (such as the SIMP formulation) rely on the penalization of intermediate densities. In the case of structural problems, the local ratio of stiffness to mass is lower for intermediate densities than for solid or void material. Certain objective functions, such as compliance, benefit from higher values of this ratio anywhere in the domain, that is, compliance decreases regardless of where new material is added. Then, by constraining the available amount of material, the optimizer tends to place it where it is most effective at reducing compliance.

On the other hand, aerodynamic optimization objectives generally do not benefit from an optimum stiffness to mass ratio, and thus intermediate densities may not be naturally avoided using the SIMP formulation. This is especially true when seeking passive load alleviation, as deformation is necessary, and in 2D where an increase in lift due to higher mass does not penalize drag as severely as it does in 3D due to higher induced drag. Therefore, an explicit mass objective or constraint may be required. However, here it was found that this tends to make the optimization process less robust, since a strong-enough incentive to remove material often conduces to critically stable structures, that is, close to buckling, for which it is difficult to fully converge the primal and adjoint problems. Furthermore, it was also observed that these strategies increase the sensitivity to initial design values. For example, starting from an initial configuration that produces more lift than required, will cause material to be quickly removed from high strain energy areas. Therefore, it is not trivial to determine to what value the mass should be constrained, or how it should be weighted before combining it with an aerodynamic objective function, such that a discrete topology is obtained (this is discussed further in chapter 5).

To avoid these issues, a two-step optimization process has been investigated to reduce the interference between the goals of improving aerodynamic performance and producing a realizable (i.e. discrete) topology. The proposed two-step strategy consists of first solving the natural aerostructural optimization problem, for example minimizing drag subject to a lift constraint, without additional penalties or manipulation of the objectives. This first step is conducted with *fully coupled* FSI modelling (so called multidisciplinary-feasible approach) and generally results in a non-discrete topology, but one defined by feasible FSI solutions and realizable since the SIMP exponent is still set according to the two-phase material bounds. The second step then aims to produce a discrete topology that replicates the response of the former, i.e. one that under the fluid loads known from the coupled FSI simulation results in the same deformed surface. This *inverse design* step is formulated similarly to a compliant mechanism design problem, for example the force inverter (see [67] or [79]), but instead of focusing on the response of a single node, an error metric is defined for the entire interface as

$$\epsilon_\Gamma = \frac{1}{N_\Gamma} \sum_{j \in \Gamma} (\mathbf{u}_j - \mathbf{u}_j^*)^2 \tag{2.37}$$

and used as a constraint in a mass minimization problem. Here both steps are solved with algorithm 1. While it would be possible to perform the *inverse design* step simulating only the structure, for highly flexible structures it has been found that doing so can easily result in an unstable structure, that buckles under small variation of FSI loads caused by small discrepancy between the target response (obtained from the *fully coupled* step) and the response of the discrete-topology structure. To mitigate this issue, a one-way FSI simulation has been used instead, then, based on the variation of the fluid loads (due to the discrepancy in target and current surface coordinates), a stability metric is defined based on points where the variation of work is positive, that is

$$W_\Gamma^+ = \sum_{j\in\Gamma} \max(0,\, \mathrm{d}\Gamma_j \cdot (\boldsymbol{\lambda}_j - \boldsymbol{\lambda}_j^*) \cdot (\mathbf{u}_j - \mathbf{u}_j^*))^2 \tag{2.38}$$

The one-way simulation is relatively inexpensive as good initial conditions for the flow field are available from the coupled FSI simulation, especially if the initial topology is almost feasible in error metric (2.37) (the topology obtained in the *fully coupled* step should not be used as the starting point to avoid convergence to a poor local optimum, i.e. not discrete). The upper bounds for constraints based on equations (2.37) and (2.38) are set based on a reference value obtained by applying a small ($\approx 1\%$) perturbation to the elasticity modulus of solid material.

For clarity it is worth considering a different view of the *fully coupled* step, which could be stated as finding surface displacements $\mathbf{u}_\Gamma^*$ that minimize the objective function, subject to the optimization constraints, and to the additional one that $\exists \boldsymbol{\rho} : \mathbf{u}_\Gamma^* = \mathbf{S}(\boldsymbol{\rho}, \boldsymbol{\lambda}_\Gamma^*)$. In other words, the outputs of the *fully coupled* step are (feasible) displacements and tractions (not the material distribution) which then become inputs of the *inverse design* step, tasked with producing a completely new material distribution that is discrete. An iterative procedure with the two steps could be proposed, in which case explicit penalization of non discreteness would likely be required to maintain the discreteness achieved by the *inverse design* step. It is worth noting that this does not decouple the multidisciplinary nature of the problem, in a way, in the *inverse design* step the FSI coupling conditions become optimization constraints, whereas in the *fully coupled* step they are inherently satisfied.

Although the optimization problem can be posed in an alternative way that avoids the discreteness and stability issues. For example, by converting it to a compliance minimization problem with aerodynamic constraints while prescribing the topology of certain key areas, such as the trailing edge. Here, it was deemed important to keep the design space as unconstrained as possible, and to deal with the aforementioned issues in a way that allowed assessing the capability of topology optimization to improve aerodynamic performance across multiple operating points.

Chapter 3

Discrete Adjoint Method

As mentioned in chapter 1, the discrete adjoint method is the approach used in this work to obtain the derivatives of functions computed from the solution of PDE problems. These arise, for example, in the context of the topology optimization problems described in chapter 2. The present chapter, is principally focused on how different discrete adjoint methods and implementation strategies contribute to the trade-off between performance and generality. The framework developed in SU2, for solving generic adjoint problems, is presented at the end of the chapter after a discussion on solution methods and implementation strategies.

3.1 Adjoint equations

There is more than one way to derive discrete adjoint solvers, by which is meant algorithms that solve discrete adjoint equations. These equations can be different depending on the chosen approach, and different approaches offer different insights into the method, its characteristics, and indeed suggest variations to the implementation strategy.

To begin, take the typical PDE constrained optimization statement

$$\begin{aligned} \min_{\alpha} \quad & J(\mathbf{x}, \boldsymbol{\alpha}) \\ \text{subject to} \quad & \mathcal{R}(\mathbf{x}, \boldsymbol{\alpha}) = \mathbf{0}, \end{aligned} \tag{3.1}$$

where J, the function of interest, depends on the optimization variables $\boldsymbol{\alpha}$ (also known as parameters) and the solution $\mathbf{x}$ of the discretized primal problem (also known as state). Primal convergence is implied by the constraints $\mathcal{R} = \mathbf{0}$, where $\mathcal{R}$ is a residual arising from either the discretization or the solution of the problem. This distinction will be clarified shortly.

Given this implicit relation between state and parameters, application of the chain rule to J

yields

$$\mathrm{d}_\alpha J = J_\alpha + J_x \ \mathbf{x}_\alpha, \tag{3.2}$$

where subscripts denote partial derivatives. Note that vectors (e.g. $\mathbf{x}$) are column vectors but the shape of derivative terms "m_n" is m by n.
The $\mathbf{x}_\alpha$ term represents the sensitivity of the state with respect to the parameters, that is, the linearization of the primal solver around the converged solution. To compute it recall that $\mathcal{R} = \mathbf{0} \ \forall \ \boldsymbol{\alpha}$, therefore

$$\mathrm{d}_\alpha \mathcal{R} = \mathcal{R}_\alpha + \mathcal{R}_x \ \mathbf{x}_\alpha = \mathbf{0} \rightarrow \mathcal{R}_x \ \mathbf{x}_\alpha = -\mathcal{R}_\alpha, \tag{3.3}$$

which is a linear system with one right-hand-side per parameter. Solving these equations and using the result to evaluate (3.2) constitutes the direct differentiation of J. To obtain the adjoint equations, (3.3) is substituted into (3.2) yielding

$$\mathrm{d}_\alpha J = J_\alpha + \boldsymbol{\lambda}^\mathrm{T} \mathcal{R}_\alpha, \tag{3.4}$$

which is an alternative way to evaluate the gradient of J, where the adjoint variables $\boldsymbol{\lambda}$ are given by the adjoint equations

$$\mathcal{R}_x^\mathrm{T} \ \boldsymbol{\lambda} = -J_x^\mathrm{T}, \tag{3.5}$$

which are also a linear system, but with one right-hand-side per function. It is important to note that this result is based on the identity $u^\mathrm{T}(Av) = (A^\mathrm{T}u)^\mathrm{T}v$, and that this identity provides a way to verify adjoint ($\boldsymbol{\lambda}$) and direct ($\mathbf{x}_\alpha$) solutions.

In regards to computational effort (time complexity), the choice between direct or adjoint approaches depends on the number of parameters versus the number of functions. However, depending on what the residual of the problem represents, storage (space complexity) may also become a decisive criteria. For example, for time domain problems solved by time-marching algorithms, the Jacobian $\mathcal{R}_x$ is a lower triangular, block-n-diagonal matrix, with one block per time step. Solving the direct equations is equivalent to forward substitution, whereas the adjoint equations must be solved by backwards substitution (the Jacobian is transposed). The direct problem can be solved alongside the primal, whereas the adjoint can only be solved once all primal results are known. Therefore, the space complexity of the adjoint approach becomes linear in the number of time steps, and reducing it requires the time complexity to be increased, e.g. by *checkpointing* (recomputing primal results during the adjoint solution, based on a downsampled version of the converged state).

3.1.1 Interpretation of adjoint variables

While the previous discussion gave some insight into the differences between direct and adjoint approaches, it revealed very little about the nature and physical meaning of adjoint variables. They are indeed more than a simple byproduct of algebraic manipulation.
Returning to the optimization statement (3.1), let

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\lambda}) = J + \boldsymbol{\lambda}^{\mathrm{T}} \mathcal{R}, \tag{3.6}$$

be the Lagrangian function, where the adjoint variables $\boldsymbol{\lambda}$ are the Lagrange multipliers (the interpretation of which is well known in optimization problems) whose value can be chosen arbitrarily since $\mathcal{R} = \mathbf{0}$ for any value of $\boldsymbol{\alpha}$. This implies $\mathcal{L} = J$ and simplifies differentiating it with respect to the parameters to

$$\mathrm{d}_\alpha J = \mathrm{d}_\alpha \mathcal{L} = \mathcal{L}_\alpha + \mathcal{L}_x \, \mathbf{x}_\alpha. \tag{3.7}$$

Formally, the above results from $\mathcal{L}_\lambda = \mathbf{0}$, due to $\mathcal{R} = \mathbf{0}$. Consider now the additional condition $\mathcal{L}_x = \mathbf{0}$, which simplifies (3.7) to be

$$\mathrm{d}_\alpha J = \mathcal{L}_\alpha = J_\alpha + \boldsymbol{\lambda}^{\mathrm{T}} \, \mathcal{R}_\alpha, \tag{3.8}$$

where $\mathbf{x}_\alpha$ no longer needs to be evaluated. The equations that result from this requirement,

$$\mathcal{R}_x^{\mathrm{T}} \, \boldsymbol{\lambda} = -J_x^{\mathrm{T}},$$

are identical to the adjoint equations obtained previously.

Lagrange multipliers, and thus adjoint variables, represent the marginal effect of the constraints on the optimum value of the function [91], or, in other words, the cost of respecting the physical laws associated with $\mathcal{R}$. To demonstrate this, note that the constraints can be assumed to contain an arbitrary constant part $\mathbf{c}$, that is

$$\mathcal{R} = \mathbf{c} - \hat{\mathcal{R}}(\mathbf{x}, \boldsymbol{\alpha}) = \mathbf{0}. \tag{3.9}$$

Now consider the parametric optimization problem given by

$$V(\mathbf{c}) = J^*(\mathbf{c}) = J(\mathbf{x}^*(\boldsymbol{\alpha}^*(\mathbf{c})), \boldsymbol{\alpha}^*(\mathbf{c})), \tag{3.10}$$

where the superscript $*$ indicates optimality, i.e. the solution of problem (3.1), whose first order optimality conditions are obtained by differentiating the Lagrangian with respect to each of its variables

$$\begin{aligned} J_\alpha^{\mathrm{T}} + \mathcal{R}_\alpha^{\mathrm{T}}\ \boldsymbol{\lambda} &= \mathbf{0}, \\ J_x^{\mathrm{T}} + \mathcal{R}_x^{\mathrm{T}}\ \boldsymbol{\lambda} &= \mathbf{0}, \\ \mathcal{R} &= \mathbf{0}. \end{aligned} \tag{3.11}$$

Note how these conditions state, respectively, the convergence of the optimization, the adjoint equations, and the primal equations. Differentiating both sides of (3.10) with respect to $\mathbf{c}$ yields

$$\mathrm{d}_c V = J_x\ \mathbf{x}_c + J_\alpha\ \boldsymbol{\alpha}_c, \tag{3.12}$$

where the superscripts have been omitted for simplicity. Substituting the first order conditions into (3.12),

$$\mathrm{d}_c V = -\boldsymbol{\lambda}^{\mathrm{T}}(\mathcal{R}_x\ \mathbf{x}_c + \mathcal{R}_\alpha\ \boldsymbol{\alpha}_c), \tag{3.13}$$

and differentiating $\mathcal{R}$, again using the fact that it is $\mathbf{0}$ for any value of $\mathbf{c}$,

$$\mathrm{d}_c \mathcal{R} = \mathcal{R}_x\ \mathbf{x}_c + \mathcal{R}_\alpha\ \boldsymbol{\lambda}_c + \mathbf{I} = \mathbf{0}. \tag{3.14}$$

Finally, substitution of (3.14) into (3.13) yields

$$\mathrm{d}_c V = \boldsymbol{\lambda}^{\mathrm{T}}. \tag{3.15}$$

This result can be easily demonstrated for non optimum values of the parameters, for example by considering $\mathbf{c}$ as the parameters (i.e. $\boldsymbol{\alpha} \equiv \mathbf{c}$) and performing the required substitutions. However, the implication for the solution of (3.1) is more meaningful.

Evidently, the physical meaning of adjoint variables depends only on that of the constants $\mathbf{c}$. For example, many physical processes are modeled by the convection-diffusion (or transport) equation, fluid flow, heat transfer, turbulence, to name a few. The equation states that, at a given location the rate of change of the transported quantity ϕ (momentum, energy, etc.) with time is equal to its volumetric rate of production ($\dot{q}$) minus the divergence of the convective and diffusive fluxes.

$$\frac{\partial \phi}{\partial t} = \dot{q} - \nabla \cdot (\mathbf{U}\phi - \mu \nabla \phi) \tag{3.16}$$

The source term may be a function of the other variables, to model for example the production of turbulent kinetic energy, or the production of thermal energy due to friction, it may be a penalization to model an immersed boundary, the heat produced due to Joule heating or radioactive decay, the result of a body acceleration, or something completely arbitrary such as the constants introduced in (3.9). Therefore, an adjoint variable associated with the imbalance of a convection-diffusion problem, represents the sensitivity of the objective function to a change in the rate of production of the transported quantity at the corresponding location. Consider

Fig. 3.1 where the magnitude and direction of the adjoint momentum of lift are shown for a NACA0012 airfoil at 0 deg AoA. Unsurprisingly, inducing circulation around the airfoil increases lift, except close to the surface where it would be more effective to produce an updraft. Note that the upstream-travelling wake simply shows that in the primal problem information travels predominantly downwind.

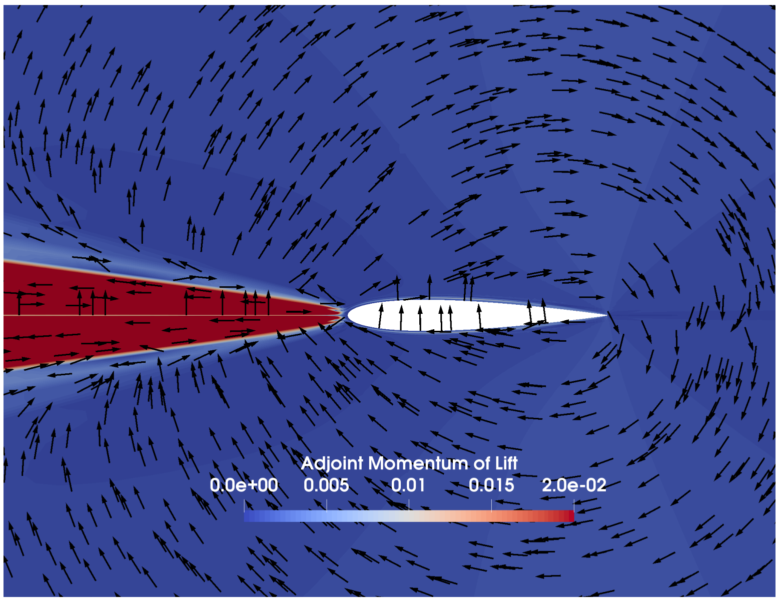


Figure 3.1: Magnitude and direction of the adjoint momentum of lift for an airfoil at 0 deg AoA.

3.1.2 Solution methods

The previous section has given valuable insight into the nature of adjoint variables, however, it did not suggest an algorithm to solve the adjoint problem. To some extent, adjoint equations (3.5) play the role of governing (linear) equations, which, similarly to the primal (typically nonlinear) equations, can be solved by multiple methods.

For classes of primal problems that are solved by an exact Newton method, and where the residual Jacobian is available, the adjoint solution method is evidently to solve a linear system. Furthermore, if those problems make use of a complete factorization of the Jacobian (e.g. elasticity problems) the adjoint variables may even be obtained at close to zero cost, more so if the problem is self-adjoint. For other classes of problems, fluid flow being the prime example, an accurate residual Jacobian is generally not available and its high condition number renders the linear system much harder to solve than expected. The fact that such problems are never solved by exact Newton methods is telling. The reasons and solutions for the lack of availability of Jacobians will be explored further on. For this class of problems it is necessary to either find a strong preconditioner for (3.5), or to derive the adjoint equations based on a definition of the residual $\mathcal{R}$ that has a well conditioned Jacobian.

Discretization residual and solution residual

To that end, the discussion returns to the distinction between "discretization residual" and "solution residual". The solution methods for all problems of interest for this work can be viewed as fixed-point iterations, where new values of $\mathbf{x}$ are given by some primal solution algorithm or program $\mathcal{P}$, that is

$$\mathbf{x}^{n+1} = \mathcal{P}(\mathbf{x}^n, \boldsymbol{\alpha}). \tag{3.17}$$

At convergence, when the fixed-point of (3.17) is obtained, the residual can be defined as

$$\mathcal{R} = \mathcal{P}(\mathbf{x}, \boldsymbol{\alpha}) - \mathbf{x} = \mathbf{0}. \tag{3.18}$$

Using this as the constraint of the optimization problem (3.1) to define the Lagrangian, and restating the requirement on its gradient with respect to the state,

$$\mathcal{L}_x = J_x + \hat{\boldsymbol{\lambda}}^{\mathrm{T}}(\mathcal{P}_x - \mathbf{I}) = \mathbf{0}, \tag{3.19}$$

results in a fixed-point iteration for a different set of adjoint variables $\hat{\boldsymbol{\lambda}}$, that is

$$\hat{\boldsymbol{\lambda}}^{\mathrm{T}} = J_x + \hat{\boldsymbol{\lambda}}^{\mathrm{T}}\mathcal{P}_x. \tag{3.20}$$

Similar manipulation of (3.8) yields

$$\mathrm{d}_\alpha J = J_\alpha + \hat{\boldsymbol{\lambda}}^{\mathrm{T}}\, \mathcal{P}_\alpha. \tag{3.21}$$

The properties of (3.20) can be studied by considering a concrete solution algorithm, take a quasi-Newton method where new iterates are given by

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \tilde{\mathbf{M}}^{-1}\hat{\mathcal{R}}, \tag{3.22}$$

where $\tilde{\mathbf{M}}^{-1}$ is an approximate inverse (usually a linear system solved approximately) of an approximate (and better conditioned) Jacobian of the discretization residual $\hat{\mathcal{R}}$. Differentiating the right-hand-side of (3.22) with respect to the state at convergence (i.e. $\hat{\mathcal{R}} = \mathbf{0}$) and substituting into (3.20) results in

$$\hat{\boldsymbol{\lambda}} = \hat{\boldsymbol{\lambda}} + \omega(J_x^{\mathrm{T}} + \hat{\mathcal{R}}_x^{\mathrm{T}}\tilde{\mathbf{M}}^{-\mathrm{T}}\hat{\boldsymbol{\lambda}}), \tag{3.23}$$

which is a Richardson iteration for the right-preconditioned adjoint equations (3.5), where the relaxation factor ω was introduced for completeness. It is known that this procedure converges if

$$||\mathbf{I} - \omega\hat{\mathcal{R}}_x^{\mathrm{T}}\tilde{\mathbf{M}}^{-\mathrm{T}}|| < 1. \tag{3.24}$$

The relationship between the adjoint variables obtained for the discretization and solution residual approaches is

$$\boldsymbol{\lambda} = \tilde{\mathbf{M}}^{-\mathrm{T}}\hat{\boldsymbol{\lambda}}, \tag{3.25}$$

and so the direct interpretation of these fixed-point adjoint variables requires knowledge about the solution method, e.g. how certain boundary conditions are enforced. Moreover, since the adjoint variables depend on the preconditioner, it is of little use to initialize the solution of (3.20) when changes are made to the solution methods within $\mathcal{P}$, e.g. changing the CFL number. These observations are fairly general since many methods can be written in the quasi-Newton form, either naturally or after some algebra, more examples are given later.
The convergence condition (3.24) offers some insight into what aspects of the primal program influence the convergence of the adjoint, namely under-relaxation and the quality of $\mathbf{M}$, which represents a trade-off between how well it approximates $\hat{\mathcal{R}}_x$ and its cost relative to one evaluation of the residual. Note that for 2D problems where $\mathbf{M}$ is a matrix, factorization is efficient since it only needs to be done once at relatively low cost (due to low bandwidth) and then the cost of the linear systems becomes independent of the condition number of $\mathbf{M}$. For large scale 3D problems this is not an option, moreover $\mathbf{M}$ might not be a matrix (e.g. it could be a multigrid method) and the fact that the $\tilde{\mathbf{M}}^{-1}$ operation often involves iterative processes makes guaranteeing (3.24) rather difficult and indeed raises other issues that will now be discussed.

Efficient treatment of linear solvers

Among the most common discrete adjoint solvers, the use of a fixed-point adjoint iteration based on the "solution residual" is unique to SU2 [33], although it was demonstrated earlier for a 2D code in [92]. The approach is based on the abstract application of algorithmic differentiation (AD) to the "whole iteration" (instead of "only" to the discretization residual) and is inspired by the broader topic of algorithmic differentiation of general fixed-point iterations [93, 94]. Earlier fixed-point adjoint solution methods in the literature were based on the duality-preserving concept [95]. They typically read [96, 97]

$$\boldsymbol{\lambda} = \boldsymbol{\lambda} + \omega\tilde{\mathbf{M}}^{-\mathrm{T}}(J_x^{\mathrm{T}} + \mathcal{R}_x^{\mathrm{T}}\boldsymbol{\lambda}), \tag{3.26}$$

which is a Richardson iteration for the left preconditioned adjoint equations. Condition (3.24) still holds since the eigenspectra of $\mathbf{AB}$ and $\mathbf{BA}$ are the same. The objective of preserving duality is to inherit the convergence properties of the primal solution method by routinely transposing (and reversing) the primal solution algorithm (note the similarity between (3.22) and (3.26)) as done in e.g. [95, 98, 99, 100]. This has the added effect of making the gradient of the objective function equal at every iteration of the tangent and adjoint problems (so long as both start from 0).

This left-right difference in the fixed-point form was pointed out in [33] but not explored further. In theory both types of iteration have the same convergence properties, however, the nature of the primal solution method, i.e. the preconditioner $\mathbf{M}$, may warrant some practical considerations. In particular, when $\tilde{\mathbf{M}}^{-1}$ involves some form of iterative linear solver, the operation is only linear with respect to the right-hand side if the linear solver is allowed to converge (making $\tilde{\mathbf{M}}^{-1} \equiv \mathbf{M}^{-1}$), which in general would be an inefficient way to solve the primal problem. The nonlinearity of approximate solutions is of little importance to the primal problem and the left fixed-point, since they are applied to a residual that converges to zero.

Conversely, with the right fixed-point that results from applying AD to the entire primal iteration, the operation is applied to a non zero vector even at convergence. The implication is that if iterative processes within the adjoint are not converged to reasonable accuracy, even the asymptotic convergence behavior will depend on J_x, the right-hand side of the adjoint equation (3.5). In partitioned multiphysics contexts this right-hand side includes contributions due to the coupling between different solvers[24, 101], which can be less smooth than the gradient of the objective function depending on the type of interface interpolation used. Empirically, this translates into normalized linear system residuals of $O(10^{-4})$ being required by this type of adjoint solver, whereas $O(10^{-1})$ often suffices for the primal solver. Moreover, there is also a dependence on the type of linear solver, since methods like GMRES produce smoother solutions than conjugate-gradient (CG) methods for the same residual. Furthermore, care is needed when evaluating $\hat{\boldsymbol{\lambda}}^{\mathrm{T}}\mathcal{P}_\alpha$ to ensure consistency with $\hat{\boldsymbol{\lambda}}^{\mathrm{T}}\mathcal{P}_x$ (e.g. the same starting point should be used) otherwise the gradient of the objective function will be incorrect.

Note that obtaining $\hat{\boldsymbol{\lambda}}^{\mathrm{T}}\mathcal{P}_x$ via AD achieves a consistent transposition of $\tilde{\mathbf{M}}^{-1}$ by definition. However, since in practice the solution of linear systems is not differentiated automatically (which would be extremely inefficient) but instead manually replaced by analytical formulas for matrix operations (specifically $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, see [102]) the resulting transposed linear systems should be solved accurately, that is, what in the primal solver is $\tilde{\mathbf{M}}^{-1}\mathcal{R}$ with $\mathcal{R} \rightarrow \mathbf{0}$ at convergence, should become "exactly" $\mathbf{M}^{-\mathrm{T}}\hat{\boldsymbol{\lambda}}$ as the adjoint solver converges. Note that these transposed linear systems can be initialized since the right-hand sides converge (e.g. to $\hat{\boldsymbol{\lambda}}$), whereas for the left fixed-point there is no benefit in doing so.

The limitations of approximately solved linear systems during reverse accumulation have recently been discussed in [103]. The problem addressed there is different and it arises from "whole program AD", where the objective function gradient is evaluated at the start of the reverse pass, used to seed the reverse-mode derivatives of the state and parameters, and then not handled explicitly anymore. The typical treatment given to linear solvers will effectively corrupt J_x if the initial assumption (exact linear operation) is not respected during reverse accumulation, as shown in [103].

However, this is not the issue that affects the right fixed-point. Since J_x is reintroduced in every iteration, if (3.23) converges, it does so to the solution of (3.5). For that to happen in a manner that is relatively independent of the adjoint right-hand side, $\tilde{\mathbf{M}}^{-\mathrm{T}}$ must be applied more accurately than in the left fixed-point. Intuitively, if in forming $\tilde{\mathbf{M}}^{-\mathrm{T}}\hat{\boldsymbol{\lambda}}$ the system is not solved with sufficient accuracy, some error components are not attenuated thereby making the fixed-point process sensitive to the adjoint right-hand side. To avoid "over solving" the transpose systems, it is then helpful to initialize them and to make their target tolerance ϵ proportional to the current residual of the adjoint equations r_{adj}, that is,

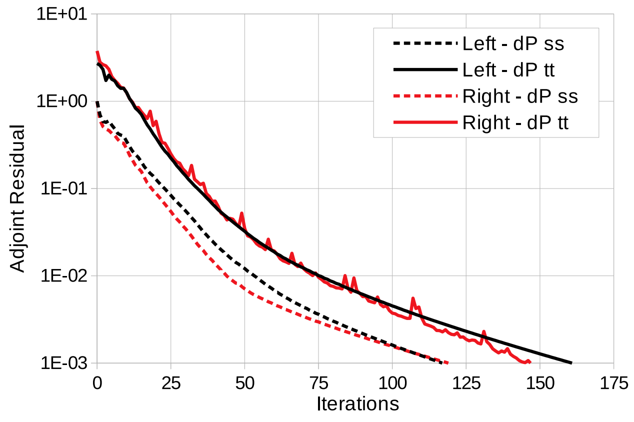
$$\epsilon = \max(\epsilon_L, \min(r_{adj}\beta, \epsilon_U)) \tag{3.27}$$

where $\epsilon_L \approx \epsilon_{adj}\beta$ and $\epsilon_U \approx 0.01$ are suitable lower and upper bounds respectively, β is $O(1)$ or $O(10^{-3})$ for the left and right fixed-point respectively, and ϵ_{adj} is the target for r_{adj}. Residuals are defined here as $r = ||\mathbf{b} - \mathbf{A}\mathbf{x}||(||\mathbf{b}||)^{-1}$.

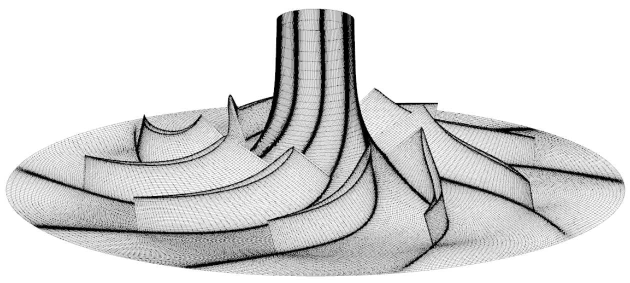
A simple adjoint solver, for which it was easy to switch between left and right preconditioning, is used to substantiate these claims with some empirical evidence. The primal solver is an unstructured pressure-based coupled solver, developed by the author [1], for the incompressible RANS equations, based on the SIMPLE algorithm with Rhie and Chow momentum interpolation. It also uses Menter's SST turbulence model, whose two equations are solved in a segregated manner. The adjoint solver uses the frozen turbulence and frozen limiter assumptions. This produces a representative adjoint problem, that is, an ill-conditioned residual Jacobian $\mathcal{R}_x$ and a more tractable approximate Jacobian $\mathbf{M}$ used in the primal solver to march to steady state (as per (3.22)) using BiCGSTAB, which is preconditioned with the additive Schwarz block ILU(0) of $\mathbf{M}$. Its transpose is then used in the adjoint solver to precondition the fixed-point, using either (3.26) or (3.23). Fig. 3.2a shows the convergence history of the adjoint fixed-point for two objective functions, with a fixed relaxation factor of 0.7 and the proportional tolerance strategy as described above. The functions are total-to-total (tt) and static-to-static (ss) pressure rise across the centrifugal pump geometry shown in Fig. 3.2b. Table 3.1 lists the cost for each function and type of iteration, measured by the total number of BiCGSTAB (inner) iterations which are the largest contribution to solution time. The two types of fixed-point preconditioning perform similarly, both in number of outer iterations, i.e. convergence rate, and in number of inner iterations, i.e. computational cost. For this problem, not converging the right preconditioner to the much lower tolerance would result in divergence, whereas not initializing it would increase the computational cost by one order of magnitude. This is much more evident for this incompressible flow problem where due to the numerical stiffness of the Poisson equation for pressure, on average 50 linear iterations are performed per outer iteration.

The importance of adequate convergence and influence of the type of linear solver are now explored for SU2. The test problem is the turbulent flow around a NACA0012 airfoil at zero angle of attack, Mach 0.5 and Reynolds number of 20 million. The SA turbulence model is used

[1]https://github.com/pcarruscag/Flow-Solver-Experiments



(a) Convergence histories.



(b) Centrifugal pump geometry.

Figure 3.2: Comparison of left and right preconditioned fixed-point iterations.

Iteration type	ΔP_{ss}	ΔP_{tt}
Left	5967	7712
Right	7532	7398

Table 3.1: Cost for left and right preconditioned fixed-points, total BiCGSTAB iterations.

without wall functions and convective fluxes are evaluated using the JST scheme. The domain is discretized with an O-grid of 75 000 quadrilaterals and the solution method being linearized (operator $\mathcal{P}$ in (3.17)) is a segregated quasi-Newton approach for mean flow and turbulence variables using the first-order residual Jacobian with a local CFL number of 1000. Fig. 3.3 shows the influence of the type of linear solver and tolerance (used for the transposed linear systems) on the convergence of the right fixed-point for the adjoint variables of lift. The linear preconditioner for the transposed systems is again the additive Schwarz block ILU(0). The first four lines in Fig. 3.3 correspond to the default strategy of not initializing the transposed systems, whereas in the last (fine dotted) line GMRES was initialized and allowed to run up to 30 iterations per fixed-point or until a linear residual of 10^{-7}. Table 3.2 lists for the three successful configurations (i.e. adjoint residual is $O(10^{-3})$) the total cost associated with the transposed linear systems, measured by the total number of matrix-vector products with the first-order Jacobian and applications of the linear preconditioner.

Several aspects are worth noting. First, due to its smoother solutions GMRES can be used with a lower tolerance than BiCGSTAB, which results in a large difference in solution time. Second, initializing the transposed linear systems reduces their cost by approximately 50%. Finally, and more nuanced, when the linear systems are initialized a smaller tolerance is needed. By analogy with the primal problem it can be said that a certain amount of smoothing is required per adjoint iteration, as simply maintaining the 10^{-2} tolerance required by the zero-initialized strategy, would not be effective. Here the 30 iteration subspace size serves this purpose while also avoiding "over solving" the linear systems while the adjoint residual is high. For CG-type

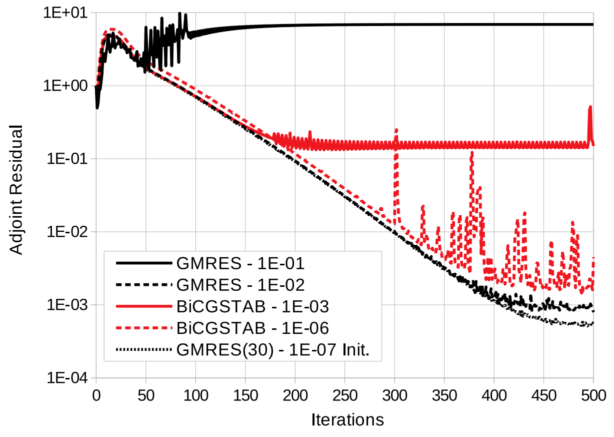


Figure 3.3: Effect of linear solver type and tolerance on the convergence of SU2's discrete adjoint solver for the NACA0012 problem.

Linear solver	Total cost
GMRES 10^{-2}	23888
BiCGSTAB 10^{-6} ($r_{adj} = 0.002$)	113904
GMRES(30) 10^{-7} Init.	12178

Table 3.2: Cost for different transposed linear solver options to obtain residuals of 10^{-3} for the NACA0012 problem.

methods limiting the number of iterations may fail catastrophically since these methods do not have the monotonic convergence of minimum residual methods. Consequently, the proportional tolerance strategy (3.27) is more appropriate for CG methods. This type of strategy is clearly important for efficient fixed-point implementations, note how the 10^{-6} tolerance for BiCGSTAB is not needed in the first 150 iterations, where 10^{-3} achieves the same reduction in adjoint residuals. However, implementing these strategies within a generic framework is challenging when the transposed linear systems are created and managed by AD. As an example, one must keep track of which initialization should be used depending on context (e.g. multigrid pre/post smoothing) and especially across operations with the two Jacobians ($\mathcal{P}_x$ and $\mathcal{P}_\alpha$).

Acceleration and stabilization by Krylov methods

In spite of inheriting the convergence properties of the primal problem by preserving duality, it is common for linearizations (tangent or adjoint) of industrial problems to have stability issues, even when the primal problem converges. This has been attributed to limit cycle behavior found in the primal problem either due to inherent unsteadiness or due to shortcomings of the primal methods [104, 96, 105, 97]. Proposed solutions have ranged from stabilization approaches using GMRES [104] or the Recursive Projection Method (RPM) [62, 106] to Newton-Krylov and Krylov methods for primal and adjoint equations respectively [107, 105, 97].

To stabilize the right preconditioned fixed-point (3.20) it is, of course, possible to use the RPM while still having in mind the earlier linear solver observations, lest the method be unnecessarily burdened with also stabilizing those processes. To use GMRES, or some other Krylov method, note first that manipulation of (3.20) yields the system

$$(\mathcal{P}_x^{\mathrm{T}} - \mathbf{I})\hat{\boldsymbol{\lambda}} = -J_x^{\mathrm{T}} \tag{3.28}$$

Here, matrix-free methods are the only alternative since $\mathcal{P}_x$ is dense. Again, the nature of the primal solution algorithm must be considered. If it contains only linear operations, any matrix-free method is applicable, as done in, e.g., [104]. However, the presence of incompletely solved linear systems is a challenge since they may not be easily extricable from the primal program (after all, this is what motivates the "whole iteration" approach (3.20)) to then be used as variable preconditioners within a flexible Krylov method, such as FGMRES, as it is routinely done for the left fixed-point [97]. Then, if the action of $\mathcal{P}_x^{\mathrm{T}}$ on a vector is not strictly linear, i.e. for all $\mathbf{u}$ and $\mathbf{v}$

$$\mathcal{P}_x^{\mathrm{T}}(\mathbf{u}+\mathbf{v}) = \mathcal{P}_x^{\mathrm{T}}(\mathbf{u}) + \mathcal{P}_x^{\mathrm{T}}(\mathbf{v}) \tag{3.29}$$

there is no guarantee that, for example, GMRES will minimize the residual. Therefore, iterative processes within the adjoint program that do not respect (3.29), need to be converged to much lower tolerances than that expected for the outer Krylov method. Furthermore, the outer solver should be restarted to avoid the build-up of numerical error.

Contrary to the right fixed-point approach, Krylov methods evaluate the vector-Jacobian product with different vectors in each iteration (e.g., the orthonormal basis built by GMRES). Therefore, although this makes the solution process more robust with respect to right-hand side sensitivity, it becomes extremely difficult to initialize the iterative processes within the adjoint program. To configure it such that it is inherently linear, for example by using a fixed number of implicit smoothing iterations in lieu of Krylov linear solvers, will almost always require a reduction of CFL number. Then, to compensate for the weaker preconditioning of the adjoint problem, a larger subspace size may need to be used to avoid the stagnation of GMRES. Therefore, it is possible that its direct application to (3.28) will stabilize the problem but not accelerate it substantially, and for the size of subspace to make it a significant contribution to memory usage. These aspects are shown for the NACA0012 problem, by comparing two Krylov approaches with their corresponding fixed-point iterations. In the first of those, GMRES is used as the outer solver and restarted every 50 iterations, the transposed linear systems are solved to a tolerance of 10^{-5} using GMRES (also restarted every 50 iterations). Whereas in the second the transposed linear systems are smoothed with 20 applications of the ILU(0) preconditioner with a relaxation factor of 0.7 (an iteration akin to (3.26)). To enable this, the subspace size is increased to 100 and the CFL number reduced from 1000 to 500. Fig. 3.4 shows the convergence histories and table 3.3 lists the cost of the different approaches, the adjoint residual plotted is the L2 norm over all variables (flow and turbulence).

As expected, GMRES improves the convergence rate for both types of fixed-point, however, applying it to one with inner Krylov methods results in higher computational cost. The stagnation of the Krylov approaches is due to the accumulation of numerical errors, which effectively causes a loss of linearity. Note in Fig. 3.4 how solving the transposed systems with a direct LU factorization mitigates this problem, which is also ameliorated by use of double precision for the linear algebra operations. Note that no operation is "truly" linear in floating point

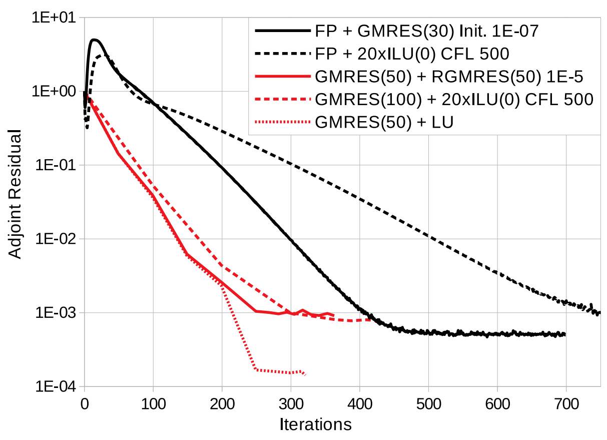


Figure 3.4: Comparison of fixed-point (black) and Krylov (red) methods for the NACA0012 problem, lift objective function.

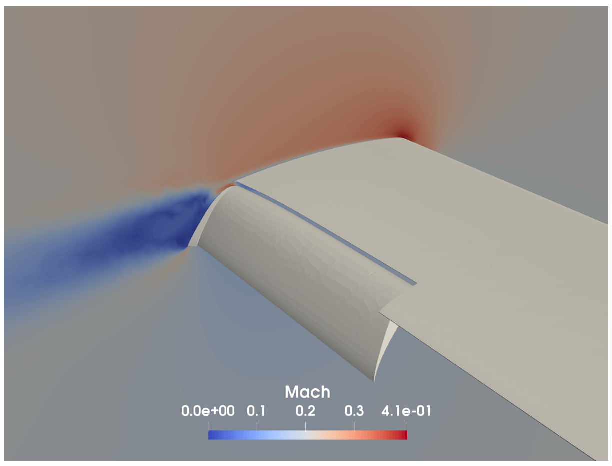
Method	Cost
FP + GMRES(30) 10^{-7} Init.	12178
FP + 20xILU CFL 500	14800
GMRES(50) + RGMRES(50) 10^{-5}	38759
GMRES(100) + 20xILU CFL 500	6000

Table 3.3: Cost of fixed-point and Krylov approaches to obtain residuals of 10^{-3} for the NACA0012 problem.

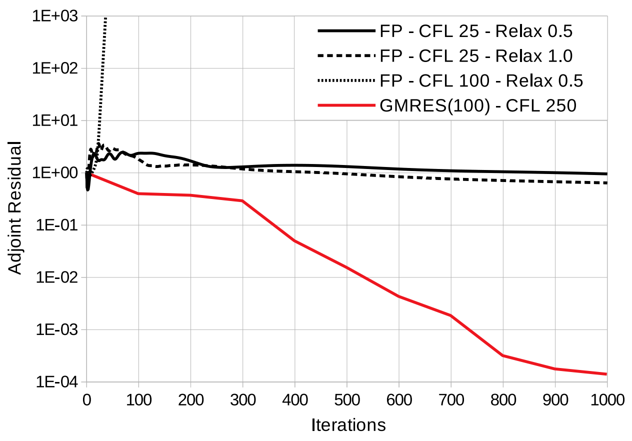
arithmetic (especially single precision) due to round-off error. Not being able to separate the preconditioner from the vector-Jacobian product aggravates this condition, and increases the importance of scaling all solution variables.

The original objective of applying GMRES to (or instead of) the fixed-point was to improve their stability, the importance of this argument is made more evident by a more challenging problem. Namely the high lift configuration of NASA's common research model (HL-CRM), which has significant regions of separated flow (see Fig. 3.5a) that complicate the task of obtaining completely converged primal results. The turbulence model and numerical schemes are identical to the NACA0012 example. Since in high-lift configuration the drag is mostly due to pressure forces, there is no noticeable downside in using a centered scheme. Previous numerical experiments in [48] showed the results to be in good agreement with other CFD codes for all reference meshes. The coarse reference hybrid mesh (B3-Coarse, see [108]) was used to study the convergence of the discrete adjoint solver. This mesh has approximately 8.3 million nodes and 18 million elements. The primal problem can be solved to residuals of 10^{-7} using either a quasi-Newton approach with CFL of 25 or a Newton-Krylov approach with CFL of 100, in both cases using GMRES+ILU(0) to approximately solve the linear systems. Fig. 3.5b is evidence that a converging primal iteration may not yield a converging adjoint fixed-point as shown by the stagnant curves for CFL of 25. With the quasi-Newton approach, a CFL of 100 results in limit-cycle oscillations, which in the adjoint fixed-point translates to quick divergence even with significant relaxation (fine dotted line).

To use GMRES to stabilize the fixed point iteration for this problem, the Krylov solvers were replaced by ILU smoothing. Specifically 30 iterations (as this gave a cost per iteration similar to the primal problem with CFL of 100) with relaxation factor of 0.3 (which was found to maximize smoothing), the CFL was then increased (to 250) until the average linear residual reduction from the 30 iterations was 1 to 2 orders of magnitude. With this setup and a GMRES



(a) Mach number contours at 35 ft from centerline.



(b) Comparison of fixed-point (black) and Krylov (red) methods, lift objective function.

Figure 3.5: Results for the HL-CRM at 8 deg AoA and Mach 0.2, "B3" coarse mesh.

subspace size of 100 iterations, the adjoint residuals for lift and drag across all variables (flow and turbulence) drop 4 orders of magnitude in 1000 iterations, as shown in Fig. 3.5b. The adjoint computed in this manner takes approximately 30% of the time needed by the primal problem.

Note that the application of GMRES to the right preconditioned adjoint system (3.28) still allows a generic approach without the challenges of managing the initialization of transposed linear systems. A subspace size of $O(100)$ iterations requires an amount of memory comparable to the primal problem, which may not be significant for the adjoint problem depending on the implementation strategy, nevertheless, the flexible variant of GMRES requires twice the storage and this could be an argument for linear preconditioners, even when they are easy to separate from the primal fixed-point.

It would be interesting to compare these results directly with FGMRES applied to the adjoint equations based on the discretization residuals (3.5), preconditioned by either ILU smoothing or GMRES unrestricted by tight tolerances. The former should be less affected by floating point issues, the latter could be faster by achieving the same smoothing effect with fewer iterations, or by supporting higher CFL number. However, for most codes this would require substantial changes to their architecture.

Finally, in both the NACA0012 and HL-CRM cases the numerical challenges are mostly due to the turbulence model, in the former it is the first adjoint residual to stagnate, and to diverge in the latter. This is not to suggest the use of the frozen turbulence assumption (which can be one of the most damaging simplifications for some optimization methods, see [109]) but rather that special care may be needed, for example different CFL number or using single precision linear algebra for the mean flow but not for turbulence.

3.1.3 Adjoints of complex programs

It is useful to show the applicability of the results from the previous sections to more intricate solution methods of the primal equations, as this will also begin to address how to obtain vector-Jacobian products. Consider the segregated solution of mean flow and turbulence equations, for which one iteration can be represented by the diagram in Fig. 3.6, where the state was split into $\mathbf{x}_1$ and $\mathbf{x}_2$ and the operations with $\mathbf{M}_i$ are depicted as multiplication by a constant, this is valid under the assumption of zero residuals but in practice these operations depend on the state. Circles represent a function, which may be trivial like addition, or complex like computing the discretization residuals ($\mathcal{R}$).

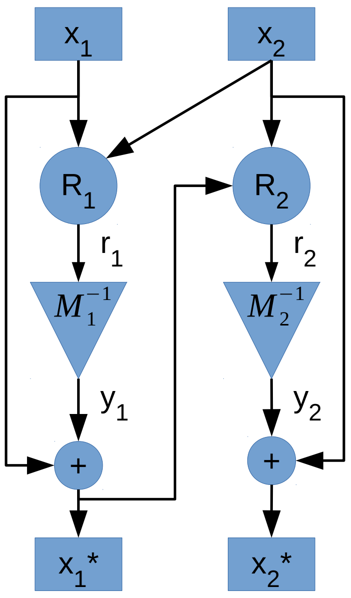


Figure 3.6: Schematic representation of the solution of two sets of variables by sequential quasi-Newton iterations.

The reverse mode of differentiation can be used to evaluate vector-Jacobian products. This mode of differentiation consists in traversing the chain rule in reverse, obtaining for each operation the derivative of the final output of the program with respect to the inputs of the operation. For completeness, the forward mode of differentiation yields for each operation the derivative of its outputs with respect to the inputs of the program. The reverse mode derivatives of some variable y are commonly represented as $\bar{y}$. There are known expressions to obtain them for matrix operations (see [102]), while for a general function $y = f(x)$ it is $\bar{x} = \bar{y}\ f_x$. The initial statement (regarding Jacobian-vector products) is simple to prove by applying this definition to the function $g(x) = \boldsymbol{\lambda} \cdot f(x)$ starting with $\bar{g} = 1$.

For the example of Fig. 3.6, splitting also the adjoint vector into $\boldsymbol{\lambda}_1$ and $\boldsymbol{\lambda}_2$ and setting (or seeding) $\bar{\mathbf{x}}_i^* = \boldsymbol{\lambda}_i^{\mathrm{T}}$ to evaluate the product with $\mathcal{P}_x$, the reverse mode derivatives at the intermediate steps are

$$\begin{aligned}
\overline{\mathbf{y}}_2 &= \boldsymbol{\lambda}_2^{\mathrm{T}}, \\
\overline{\mathbf{r}}_2 &= \overline{\mathbf{y}}_2 \mathbf{M}_2^{-1}, \\
\overline{\mathbf{y}}_1 &= \boldsymbol{\lambda}_1^{\mathrm{T}} + \overline{\mathbf{r}}_2 \; \partial_{x_1} \mathbf{R}_2, \\
\overline{\mathbf{r}}_1 &= \overline{\mathbf{y}}_1 \mathbf{M}_1^{-1}.
\end{aligned} \tag{3.30}$$

Collecting the results at the nodes for $\mathbf{x}_1$ and $\mathbf{x}_2$, the product $\overline{\mathbf{x}} = \boldsymbol{\lambda}^{\mathrm{T}} \mathcal{P}_x$ is obtained as

$$\begin{aligned}
\overline{\mathbf{x}}_1 &= \overline{\mathbf{y}}_1 + \overline{\mathbf{r}}_1 \; \partial_{x_1} \mathbf{R}_1, \\
\overline{\mathbf{x}}_2 &= \overline{\mathbf{y}}_2 + \overline{\mathbf{r}}_2 \; \partial_{x_2} \mathbf{R}_2 + \overline{\mathbf{r}}_1 \; \partial_{x_2} \mathbf{R}_1,
\end{aligned} \tag{3.31}$$

which with some algebra can be presented in the right-preconditioned form

$$\overline{\mathbf{x}}^{\mathrm{T}} = \begin{bmatrix} \partial_{x_1} \mathbf{R}_1 & \partial_{x_2} \mathbf{R}_1 \\ \partial_{x_1} \mathbf{R}_2 & \partial_{x_2} \mathbf{R}_2 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \mathbf{M}_1 & \mathbf{0} \\ -\partial_{x_1} \mathbf{R}_2 & \mathbf{M}_2 \end{bmatrix}^{-\mathrm{T}} \boldsymbol{\lambda}. \tag{3.32}$$

This approach of splitting the state and adjoint vectors according to the high-level structure of the primal program, can be recursively applied to different time instances, different solvers in a multi-physics problem, and different variables within weakly coupled solvers. With such a partitioned view of the problem it is relatively simple to suggest adjoint solution methods. For example, the aforementioned reverse time marching, or block Gauss-Seidel strategies for multi-physics (see e.g. [24]).

3.1.4 Trade-off between performance and generality

Notwithstanding the linear solver challenges, the "solution residual" derivation and subsequent analysis of what high-level operations are required to evaluate $\mathcal{P}_x^{\mathrm{T}} \hat{\boldsymbol{\lambda}}$, suggests solutions for how to precondition the adjoint equation (3.5) to allow its solution by iterative methods, and more importantly, it shows what aspects of the primal program $\mathcal{P}$ can be tuned to improve the convergence of the adjoint fixed-point approach. This type of analysis is an important tool because direct application of general linear preconditioners (e.g. ILU(n)) to the Jacobian $\mathcal{R}_x$ is not trivial due to its high condition number, there are reports [96] of up to ILU(4) being used, with one and two levels of fill being common [110]. Moreover, general linear preconditioners may pose a greater scalability challenge on modern (heterogeneous) parallel machines than that of transposing the primal solution method. Nevertheless, the linear solver aspects mentioned before are disadvantages of the right fixed-point approach compared with its left counterpart, which offers more flexibility and control over the solution methods when a discretization residual is readily available, and consequently better performance. In broadening the scope of application to any primal fixed-point, essentially any solver, the "whole iteration" strategy trades performance for generality.

Its advantage is its applicability when a discretization residual cannot be readily identified. For example, the SIMPLE scheme (and others like it) does not have a single equation like (3.22) for any of its variables (velocities and pressure) all of them are subject to one or more corrections over the course of one iteration. Then, to derive a traditional discrete adjoint equation (3.5) that is consistent with the primal solution method, authors [111, 110] augment the state and residual vectors beyond physical variables to include the Rhie and Chow interpolated fluxes, and use either general preconditioners [110] or ones based on the primal method [111].
As demonstrated above, in the whole iteration approach the preconditioning of the adjoint equations (3.5) is embedded in the vector-Jacobian product $\boldsymbol{\lambda}^{\mathrm{T}}\mathcal{P}_x$, thereby allowing a generic treatment of any type of solver. This ability for top-level handling has later been used to build a multiphysics adjoint solver within the SU2 framework [101]. This is paid for with additional care and cost, needed to ensure that the preconditioning is effective, efficient, and compatible with stabilization and acceleration techniques. This multiphysics framework is based on the partitioning strategy introduced in the previous subsection. Later in this chapter, it will be described in more detail after the following discussion on implementation strategies.

3.2 Implementation aspects

Regardless of the adjoint solution method (fixed-point, Krylov, etc.), vector-Jacobian (or Jacobian-transposed-vector) products are central to any implementation. In fact, in the "whole iteration" approach, the solution method *is* a vector-Jacobian product. Given that any discrete adjoint implementation represents a compromise between performance, maintainability, third-party compatibility, knowledge level for new developers, etc. the techniques to compute those products, described in this section, should be seen as complementary. Furthermore, this discussion is applicable even for self-adjoint problems where an exact Jacobian with respect to the state is available, since differentiating the residual with respect to the parameters may not be trivial.

3.2.1 Manual differentiation

The reverse mode of differentiation described in subsection 3.1.3 can be applied recursively to smaller and smaller components of a function or program, to obtain highly optimized (and problem-specific) functions to compute vector-Jacobian products in a matrix-free manner. This technique has the potential for highest performance and lowest memory usage, at the expense of higher maintenance costs. One example of its application is manually replacing large linear algebra operations (such as solving linear systems) by their adjoint counterparts, as done in the previous sections. For such operations there is no alternative if the resulting code is expected to be efficient.

As a lower level example consider the computation of spatial gradients by the Green-Gauss method, which for a median-dual discretization can be written as

$$\mathbf{G}_i = \frac{1}{\Omega_i} \sum_{(j,k)} s(i,j) \mathbf{A}_k \, \frac{1}{2}(\phi_i + \phi_j), \tag{3.33}$$

where boundary contributions are omitted for simplicity. The summation for each volume Ω is over its direct neighboring pairs of nodes j and edges k. The sign function s changes the direction of the normal vector $\mathbf{A}$ as needed, the common convention is for the normal to be directed from i to j if $i < j$. The gradient of the solution variable ϕ is denoted by $\mathbf{G}$ to avoid the potentially confusing notation of reverse mode derivatives of spatial derivatives. Recalling that for $y = f(x)$ it is $\bar{x} = \bar{y}\ f_x$, and applying it to (3.33) yields for each of the inputs

$$\begin{aligned}
\bar{\phi}_i &= \overline{\mathbf{G}}_i \cdot \frac{1}{2\Omega_i} \sum_{(j,k)} s(i,j) \mathbf{A}_k, \\
\bar{\phi}_j &= \overline{\mathbf{G}}_i \cdot \frac{1}{2\Omega_i} s(i,j) \mathbf{A}_k, \\
\bar{\Omega}_i &= -\overline{\mathbf{G}}_i \cdot \mathbf{G}_i \, \frac{1}{\Omega_i}, \\
\overline{\mathbf{A}}_k &= \overline{\mathbf{G}}_i \frac{1}{2\Omega_i} s(i,j)(\phi_i + \phi_j).
\end{aligned} \tag{3.34}$$

Collecting the contributions for each node and each edge over the entire domain, and noting that $s(j,i) = -s(i,j)$, results in the expressions

$$\begin{aligned}
\bar{\phi}_i &= \frac{1}{2} \sum_{(j,k)} s(i,j) \mathbf{A}_k \cdot \left(\frac{1}{\Omega_i} \overline{\mathbf{G}}_i - \frac{1}{\Omega_j} \overline{\mathbf{G}}_j \right), \\
\overline{\mathbf{A}}_k &= \frac{1}{2}(\phi_i + \phi_j) \left(\frac{1}{\Omega_i} \overline{\mathbf{G}}_i - \frac{1}{\Omega_j} \overline{\mathbf{G}}_j \right), \\
\bar{\Omega}_i &= -\overline{\mathbf{G}}_i \cdot \mathbf{G}_i \, \frac{1}{\Omega_i}.
\end{aligned} \tag{3.35}$$

Note that the only additional memory required by the hand-differentiated implementation is to store the reverse mode derivatives Moreover, under the assumption that the solution is converged, the same values of ϕ and $\mathbf{G}$ can be used to compute reverse mode derivatives in all calls to the gradient routine within the program (or iteration). Furthermore, the final access pattern to memory was not changed, initial naive application of the differentiation rule resulted in a *scattering* of contributions to all inputs of (3.33), however, manually collecting the terms restored the embarrassingly parallel *gather* pattern of the primal algorithm. This second step required domain knowledge, namely that each edge connects two nodes, consequently, the sparse pattern associated with the neighbor matrix is symmetric.

3.2.2 Algorithmic differentiation

Algorithmic differentiation (AD) (also known as *automatic* differentiation, although the accuracy of this designation depends on context) refers to the software implementation of forward or reverse-mode differentiation as described earlier. The objective of this section is not to provide an exhaustive description of AD techniques, which can be sought in [112], but rather to discuss the implications of using them to build discrete adjoint solvers. AD is commonly implemented either via source transformation or operator overloading.

Source transformation

Source transformation AD tools create new code based on the primal program. In a way these tools transform code like that of (3.33) into (3.34), in general such tools lack the domain knowledge required to obtain (3.35). Domain specific languages (DSL) can provide optimized implementations for certain families of operations, for example adjoint stencil operations which still perform *gather* operations and thus are thread-safe like their primal counterparts. The programmer can annotate the primal code to inform the source transformation tool, for example, that in the Green-Gauss example ϕ and $\mathbf{G}$ will not change and will still be available during the reverse pass, this allows the tool not to store a copy of those variables. Nevertheless, it is common for transformed sources to require manual tuning to remove this kind of store/restore instructions. In general, source transformation tools do not support all the features of C++, or to do so they need to borrow operator-overloading techniques. Thus they are most suitable for C and Fortran codes, notwithstanding, it is not unreasonable to use such tools to help create optimized reverse-mode code for specific functions of a C++ program.

Operator overloading

Operator overloading AD consists in the application of $\bar{x} = \bar{y} f_x$ to all unary and binary operations in the code (which are what C++ allows programmers to overload). To do this, an AD type that replaces the usual computation type (float or double) is defined and all relevant math functions and operators are overloaded, such that for each elementary function its Jacobian (f_x) can be computed. To allow the reverse mode derivatives to be computed without generating new code, a representation of the the primal computations needs to be created by storing enough information about them in a *tape* during a recording pass of the primal code. There are multiple options for what information to store (see e.g. [34]), that discussion is outside the scope of this work and as will be explained, its does not matter for the final adjoint implementation which to be efficient must make heavy use of higher level features of the AD tool. The main downside of this technique compared with source transformation is that it will use more memory, whereas the latter will encode a loop as a reverse loop in the adjoint code at compile-time, the former will unroll the entire loop into the tape at runtime. The main advantage is that in general a first version of an adjoint solver can be obtained in less time, even if inefficient, such a solver can guide the development of highly tuned hand-differentiated codes.

Three features are fundamental to make operator overloading efficient for HPC-class codes, namely expression templates, external function handling, and preaccumulation. Expression templates allow data to be stored in the tape per assignment rather than per statement, that is, a complex statement such as $y = cos(a) * \sqrt{a - b(c - d)}$ generates one tape entry rather than one for each unary/binary operation, this reduces both memory usage and tape interpretation time. External function handling allows the programmer to provide reverse-mode code for certain key areas, for example linear solvers, and for external functions that the AD tool cannot access. Finally, preaccumulation consists in converting sections of the tape into small Jacobians during the recording phase, again this reduces memory usage and interpretation time, but at the expense of longer recording times since the preaccumulated sections need to be evaluated a number of times to generate a Jacobian (typically the minimum between number of inputs and number of outputs, i.e. forward and reverse mode evaluation respectively). Preaccumulation requires the programmer to define what regions and variables it should be used on, and this can be a common source of errors when new developers add features to the primal code without being aware of the implications to adjoint solvers. Logically this technique is most effective for compute-dense regions of code. Therefore, choosing the preaccumulation scope to take advantage of locality is important, for example, capturing several calls of a function where some inputs or outputs are the same. To further reduce memory usage it is important to fuse loops, recompute certain quantities rather than storing them, and in the limit re-organize certain computations for the recording phase.

To illustrate the last point consider a finite element elasticity problem on a hexahedral grid. Typically the residual and stiffness matrix are computed by looping over elements gathering data from its nodes (8 in this case) and then scattering contributions to those nodes. With 3 state variables (displacements) and 3 parameters (coordinates) per node, and at least 2 parameters per element (Young's modulus and Poisson's ratio), more if other material properties are relevant. The size of the preaccumulated Jacobian is $(8(3 + 3) + 2)(8 \times 3) = 1200$ per element. Alternatively, the complete residual of each node could be evaluated by looping over the 8 elements that contribute to it (discarding the contributions to other nodes), with this strategy the size of the Jacobian is $(27(3 + 3) + 8 \times 2)(1 \times 3) = 534$ per node. Since there are approximately as many nodes as there are elements in a hexahedral grid, the second approach is approximately 2 times more efficient in terms of memory use. Moreover, it is not 8 times more computationally expensive since the section of the tape would be evaluated fewer times. Even if the primal code is structured in a way that allows this type of re-organization, its impact to the readability and maintainability of the code is high. The conclusion of this analysis is that excluding the use of external functions, operator overloading AD requires at least as much memory as storing the entire Jacobian of an operation. Note that for finite volume discretizations a similar re organization would not be beneficial, since looping over edges is already equivalent to looping over the neighbors of each node, and only one quantity (the flux) is computed per edge. It is, however, very beneficial to fuse all loops over edges that contribute to their fluxes, and all loops over nodes that contribute to their residuals.

Table 3.4 shows a realistic assessment of how much memory is used by the discrete adjoint flow solver in SU2 relative to the primal flow solver[2] for a practical RANS problem (SA model, MUSCL reconstruction, Roe flux, steady state Euler implicit without multigrid). For reference, for this type of problem the primal solver requires approximately 4 GB and 2 GB per 10^6 elements on the hexahedral and hybrid meshes respectively. The relative memory use is better for Euler or Navier-Stokes problems since gradients are only computed once per iteration. It is much worse, however, if a multigrid solution is differentiated, since currently multiple smoothing iterations are added to the tape. The Jacobian with respect to the parameters (mesh coordinates) uses more memory due to the computation of the geometric properties of the dual volumes. It would be possible to bring memory use down to approximately six times that of the primal solver by manually differentiating gradients and geometric properties, but thus far that was not deemed necessary.

Mesh type	Jacobian	Memory factor
Hexahedral	State ($\mathcal{P}_x$)	7.17
Hexahedral	Parameters ($\mathcal{P}_\alpha$)	10.6
Hybrid	State ($\mathcal{P}_x$)	7.72
Hybrid	Parameters ($\mathcal{P}_\alpha$)	10.2

Table 3.4: Memory used by the SU2 discrete adjoint solver relative to primal, typical RANS problem.

Typically the use of operator overloading involves replacing the native floating-point type by the AD type in all active regions of the code, such that any variable can easily be considered part of the state or parameters. In the context of typical unstructured solvers this has implications on the optimizability of the code, for example vectorization becomes difficult (note that a reverse-mode AD type is a real-integer pair, therefore, in an array of these types the real parts are not contiguous in memory). Even if external functions are used, they will either operate on a type with twice the size, or require primal and reverse-mode values to be extracted from the AD type into temporary variables, which may be unavoidable if the external function is "truly external", such as a call to a BLAS library. Note that in different contexts it may be possible to define matrix-AD types for which the real parts are contiguous in memory. Notwithstanding

[2]after the improvements made by the author, see chapter 4

these challenges, one interpretation of the tape may be faster than one residual evaluation of the primal code, it all depends on how much computation can be condensed into the preaccumulated Jacobians. Table 3.5 shows the time taken per explicit iteration of the adjoint solver relative to the primal. For Navier-Stokes problems the adjoint solver is at a lesser disadvantage as more computations are included in the same preaccumulation scope (with a similar number of inputs and outputs). Reducing memory usage would reduce tape interpretation time proportionally, however, since implicit solution methods are more common, and it is generally possible to use higher CFL for adjoint problems, the time disadvantage is not as important as the higher memory usage

Equations	Time factor
Euler	4.2
Navier-Stokes	3.4

Table 3.5: Time per explicit iteration of the SU2 discrete adjoint flow solver relative to primal.

There is yet hope for operator overloading *-based* discrete adjoint solvers that are nearly as efficient, in both compute and storage, as the primal solvers, but less so for their development being *automatic*. However, note that the discussion in this section was focused on the 10% of a code that is responsible for 90% of its runtime. Being able to immediately differentiate the other 90% of code (e.g. boundary conditions, fluid models) is an advantage of applying operator overloading AD to an entire code, especially one like SU2 that aims to be a generic framework open for anyone to develop.

Other considerations

There are two aspects of AD, that although not directly relevant for vector-Jacobian products, are relevant to the implementation of matrix-based strategies (i.e. where the Jacobian is obtained).

First, the forward (or tangent) mode of AD, which cannot be used directly to evaluate vector-Jacobian products (unless the Jacobian is symmetric). It is tape-less even if implemented via operator overloading, in which case the replacement type is a real-real pair. Derivatives are evaluated immediately without the need for recording the primal function, thus the reduced optimizability of the code is not offset by the potential savings of preaccumulating some regions. With the source transformation approach there is no impact on optimizability. Parallelism is not affected by either approach, since there is no reversal of the flow of information.

Second, vector mode, or multi directional forward mode, or multi objective reverse mode AD, which consists in propagating multiple derivatives simultaneously. It is equivalent to solving equations (3.3) or (3.5) for multiple right-hand-sides at once, which can reduce the cost per right-hand-side by increasing the FLOP-to-byte ratio of bandwidth-limited operations, but for that to happen sparse linear solvers must operate on a matrix right-hand-side (rather than on one column at a time). Furthermore, vector mode provides a naturally vectorizable dimension to all computations, even with operator overloading (the replacement type becomes a real-array pair), and this may, for example, tip the scale further towards direct differentiation of time domain problems (as mentioned earlier the adjoint of such problems may be more compute intensive if it is necessary to recompute states to save storage).

3.2.3 Matrix-based methods

There are some arguments for instantiating the Jacobian of a discretization residual instead of directly obtaining its action on a vector in a matrix-free manner. As detailed in the previous section, for a given operation this would not use more storage than operator overloading AD with preaccumulation. The Jacobian can be stored in a format that allows better parallelization and optimization of the matrix-vector product than is possible with source transformation. It may be feasible to obtain the Jacobian without using AD, which may allow adjoint solvers to be developed even in cases where AD is difficult to apply, for example closed sources, external libraries, complex code bases (for example [113, 110]), or highly optimized ones, to name a few. The sledgehammer approach of solving equation (3.5) via a direct sparse solver becomes available.

One possible way of generating rows of the Jacobian is the aforementioned re-organization of routines into a high level function that computes the residual for a single solution location (node or cell). This function can be differentiated using either AD (of any kind), the complex step method, or finite differences (which allows opaque routines to be differentiated). This local residual function is natural for finite difference discretizations, it is relatively straightforward to create for finite element methods, but complex to define for unstructured finite volume methods. In general, the residual of a control volume (dual or not) depends on neighbors of neighbors (NoN) due to the way quantities such as gradients, limiters, laplacians, sensors, etc. are computed. To some extent the discretization is not local, as it involves multiple loops over the entire domain to compute these variables. It is possible to reconstruct, for each volume, the part of the mesh that influences its residual so that the normal solver routines can be applied locally, this is a complex task. Consider for example that in the context of domain decomposition some NoN are usually not available locally. Nevertheless, this approach results in an embarrassingly parallel way of generating the Jacobian, it was used for example in [114] and by the author in the simple adjoint solver mentioned in subsection 3.1.2.

Alternatively, since the Jacobian is sparse (and its graph is known) it is possible to perturb (or seed in case AD is used) multiple entries of the entire solution vector simultaneously to obtain multiple columns of the Jacobian (see [113, 115, 110]). For completeness, using the reverse mode would generate rows, but if efficient reverse mode routines were available this approach would

be redundant. Determining which locations can be perturbed simultaneously is a graph coloring problem, any two locations that are NoN to each other must have different colors. Therefore, the minimum number of colors is equal to the maximum number of NoN in the mesh, and the number of times the residual function needs to be evaluated is equal to the number of colors multiplied by the number of degrees of freedom per location divided by the AD vector length (if vector mode is used). Note that special care is still needed for domain decomposition as the perturbations will cross MPI boundaries. For structured grids efficient analytical colorings can be devised that allow the computation of the Jacobian in $O(10)$ to $O(100)$ time (relative to one residual evaluation) [115], greedy coloring can produce acceptable results for unstructured meshes where the number of NoN is small [113]. However, for polyhedral meshes (e.g. dual volumes of a hybrid mesh) the number of NoN can be $O(100)$ and 200 colors can easily be required, which makes this approach almost one order of magnitude more expensive for this type of discretization. This can still be acceptable for pressure-based incompressible solvers whose residuals are inexpensive to evaluate compared to a complete iteration. The analysis for the Jacobian with respect to mesh coordinates ($\mathcal{R}_\alpha$) is similar but must account for what geometric properties are used to compute the gradients at the direct neighbors, and how those properties depend on the mesh coordinates.
These two approaches can also be used in a matrix-free manner where the rows or columns are immediately used in the product, which can be especially important for the geometric Jacobian which is used less often, and in general is denser.

3.2.4 Choice of parameters

In the preceding sections it was mentioned in passing that the adjoint parameters ($\boldsymbol{\alpha}$) are typically the mesh coordinates. This is a common choice for shape optimization (and a requirement for goal oriented mesh adaptation) but logically other applications require other variables, for example, global model constants for model calibration, or local densities for density-based topology optimization. What makes mesh coordinates a common choice is that this allows a modular approach, whereby any type of differentiable shape parameterization method ($\mathcal{S}$) can use those sensitivities to complete the chain rule and obtain the derivatives with respect to the design parameters ($\boldsymbol{\beta}$) that are actually used in the optimization, that is,

$$\mathrm{d}_\beta J = \mathrm{d}_\alpha J \;\; \mathrm{d}_\beta \mathcal{S}. \tag{3.36}$$

Here, what is meant by choice of parameters is not the type of parameterization but rather the adjoint parameters, or, in other words, where the design and simulation pipeline (the chain rule) is split. Usually the computational cost of the discrete adjoint method does not depend strongly on the number of parameters, however, the storage requirements might depending on the implementation strategy. This is shown by the SU2 measurements of table 3.4, where the geometric sensitivities use 30 to 50% more memory. Similar issues arise for topology optimization where local densities are typically the result of a spatial filtering operation. When this type of pre-processing operation is differentiated using operator overloading AD, peak memory usage may be reduced by including them in the parameterization $\mathcal{S}$ rather than in the program $\mathcal{P}$. Furthermore, when the strategies of section 3.2.3 are used, computational cost may also become a concern since Jacobians with respect to mesh coordinates are denser and more expensive to compute. Then, depending on the number of design parameters ($\boldsymbol{\beta}$) it may become less expensive to differentiate the parameterization with a forward method. Note that for some types of problem, for example fluid structure interaction, the computation of geometric properties (areas and volumes) is part of the state Jacobian since the deformed mesh coordinates depend on the state variables.

3.3 Multiphysics adjoints in SU2

Having described general trade-offs between different solution approaches and implementation strategies, the case of SU2 can now be discussed in more detail. The current vision for SU2 is for it to become a framework for multiphysics simulation and optimization. This term, framework, can sometimes be used with too much liberty. A framework imposes a certain way, i.e. a flow of control, to perform a general task, e.g. computing adjoints. It then allows its users to extend, or specialize, its functionality, by allowing them to override certain key aspects (for example, what are the parameters of the problem). An even more ambitious goal of SU2, is for this extension process to be "discrete-adjoint-oblivious", that is, for new functionality to be differentiated with no additional effort. To achieve these goals, operator overloading AD is used to implement the "whole iteration" solution approach. In a multiphysics context, where there are time iterations, coupling iterations (across different physics), and inner iterations (for each physics), it is important to specify to which iteration AD is applied. Because operator overloading AD is used, the only cost-efficient solution is to record inner iterations, one for each physical domain, since otherwise the memory usage would be too high (due to multiple iterations being added to the tape). In doing so, the solution process is no longer a "simple" application of AD, it becomes necessary to orchestrate these recordings into a multiphysics (and possibly also unsteady) adjoint solution method.

It is worth noting that this description, in terms of coupling and inner iterations, is typical of partitioned solution methods, and thus not appropriate for monolithic solvers. In fact, *zone* is a more appropriate designation, and better software abstraction, than physics. A zone is a physical region, which may have multiple physics (i.e. solvers) associated. The solvers within a zone, are typically tightly coupled and exchange information over the entire domain. Whereas zones, exchange information across shared boundaries (although nothing precludes them from overlapping). Before deriving a solution method for general *multizone* problems, it is worth considering a specific multiphysics example.

3.3.1 Earlier work on fluid-structure-interaction

Earlier work by Sanchez et al.[24], followed a domain-specific strategy for FSI, analogous to that of Maute et al.[116] (based on the discretization residuals of the three-field problem, i.e. flow, structural, and mesh deformation), to obtain an AD-based discrete adjoint solver for coupled FSI problems. Their derivation is based on a global fixed-point iteration, obtained by concatenating the inner iterations of the three fields. Contrary to single zone problems, this global fixed-point is not the solution method of the primal problem, but rather a convenient manipulation used to derive the coupled adjoint equations. That is, this fixed-point is used as the equality constraints in (3.1). In particular,

$$\mathcal{R} = \mathcal{P}(\mathbf{x}, \boldsymbol{\alpha}) - \mathbf{x} = \begin{pmatrix} \mathcal{F}(\mathbf{w}, \mathbf{z}, \boldsymbol{\alpha}) \\ \mathcal{S}(\mathbf{u}, \mathbf{w}, \mathbf{z}, \boldsymbol{\alpha}) \\ \mathcal{M}(\mathbf{u}, \boldsymbol{\alpha}) \end{pmatrix} - \begin{pmatrix} \mathbf{w} \\ \mathbf{u} \\ \mathbf{z} \end{pmatrix} = \mathbf{0}, \tag{3.37}$$

where $\mathcal{F}, \mathcal{S}, \mathcal{M}$ and $\mathbf{w}, \mathbf{u}, \mathbf{z}$ are the fluid, structural, and mesh iterations and state variables, respectively. Recalling the adjoint fixed-point (3.20)

$$\hat{\boldsymbol{\lambda}}^{\mathrm{T}} = J_x + \hat{\boldsymbol{\lambda}}^{\mathrm{T}} \mathcal{P}_x,$$

but now using the definition of residual based on the three fields, results in

$$\begin{pmatrix} \overline{\mathbf{w}} \\ \overline{\mathbf{u}} \\ \overline{\mathbf{z}} \end{pmatrix} = \begin{pmatrix} J_w^{\mathrm{T}} \\ J_u^{\mathrm{T}} \\ J_z^{\mathrm{T}} \end{pmatrix} + \begin{bmatrix} \mathcal{F}_w & \mathbf{0} & \mathcal{F}_z \\ \mathcal{S}_w & \mathcal{S}_u & \mathcal{S}_z \\ \mathbf{0} & \mathcal{M}_u & \mathbf{0} \end{bmatrix}^{\mathrm{T}} \begin{pmatrix} \overline{\mathbf{w}} \\ \overline{\mathbf{u}} \\ \overline{\mathbf{z}} \end{pmatrix}, \tag{3.38}$$

where the over-bar denotes adjoint variables. This is not yet the complete coupled solution method. For that, and in analogy with the primal problem, inner iterations are introduced for the fluid and structural problems (mesh deformation is assumed to be explicit). Which results in a segregated solution method (also known as lagged-coupled-adjoint [117]), given by

$$\begin{aligned} \overline{\mathbf{w}}^* &= \overline{\mathbf{F}}(\overline{\mathbf{w}}, \mathcal{S}_w^T \overline{\mathbf{u}}) \\ \overline{\mathbf{z}}^* &= \overline{\mathbf{M}}(\overline{\mathbf{w}}^*, \overline{\mathbf{u}}) = \mathcal{F}_z^T \overline{\mathbf{w}}^* + \mathcal{S}_z^T \overline{\mathbf{u}} \\ \overline{\mathbf{u}}^* &= \overline{\mathbf{S}}(\overline{\mathbf{u}}, \mathcal{M}_u^T \overline{\mathbf{z}}^*) \end{aligned} \tag{3.39}$$

where * denotes new values after all inner iterations. Also in analogy with primal problems, the combined effect of these iterations is a *solver*, denoted here by $\overline{\mathbf{F}}, \overline{\mathbf{M}}$, and $\overline{\mathbf{S}}$, for each of the three fields. Note that it is possible to apply a Krylov method to (3.38), as it was shown for single zone problems, and as it has been done for approaches based on the discretization residuals (e.g. [5]). However, in the "whole iteration" context, this may not be efficient for some mesh deformation strategies. In particular, the solid elasticity analogy, which involves the solution of a linear system, that is much more expensive than one fluid or structural iteration. Doing so, with the accuracy required to ensure the linearity of the vector-Jacobian product, would not be cost-efficient. Furthermore, even if a cheaper deformation approach were used, the structural solver would eventually become the bottleneck. Indeed, each of the fields has very different convergence rates, one mesh iteration, a few structural iterations, tens of fluid iterations, and this suits the segregated strategy well. Note, however, that this issue would be less severe if flexible preconditioning were possible. Checkpointing is another aspect that makes the segregated strategy more adequate. As the previous section showed, the major challenge of operator overloading AD is memory usage. The segregated approach allows each field to be recorded independently, once per coupling iteration, thereby reducing peak memory usage. However, this is only efficient if the recording cost is amortized over a number of inner iterations.

The downside of this type of solution method, is the potential lack of stability for strongly nonlinear problems. Solving this by relaxing the update of cross terms (i.e. products with off-diagonal blocks of the Jacobian) results in slow convergence. A better alternative, is to use stronger coupling methods [36] which are common for primal simulations [59]. Typically, such methods determine the interface variables of the problem, since these define the solution of all fields. To obtain the interface problem for (3.39), note that with sufficient convergence of the adjoint solvers, it can be simplified to

$$\begin{aligned} \overline{\mathbf{u}} &= \overline{\mathbf{S}}(\mathcal{M}_u^T\overline{\mathbf{z}}) \\ \overline{\mathbf{w}} &= \overline{\mathbf{F}}(\mathcal{S}_w^T\overline{\mathbf{u}}) \\ \overline{\mathbf{z}} &= \overline{\mathbf{M}}(\overline{\mathbf{w}},\overline{\mathbf{u}}). \end{aligned} \tag{3.40}$$

Moreover, note that by the construction of $\mathcal{M}$, it is $\mathcal{M}_u = 0$ for any structural node that is not part of the interface (Γ). Therefore, the adjoint interface displacements are identified as

$$\mathcal{M}_u^T\overline{\mathbf{z}} = \begin{pmatrix} \mathbf{0} \\ \overline{\mathbf{u}}_\Gamma \end{pmatrix}, \tag{3.41}$$

which allows defining an adjoint interface residual, given by

$$\overline{\mathbf{R}}_\Gamma(\overline{\mathbf{u}}_\Gamma) = \overline{\mathbf{M}} \circ \overline{\mathbf{F}} \circ \overline{\mathbf{S}}(\overline{\mathbf{u}}_\Gamma) - \overline{\mathbf{u}}_\Gamma, \tag{3.42}$$

to which quasi-Newton methods, such as IQN-ILS [36], can be applied to determine $\overline{\mathbf{u}}_\Gamma$ and consequently $\boldsymbol{\lambda}$. This strategy was used by the author in [118].

This approach is not as general as it is desired, since its implementation requires domain-specific knowledge to evaluate the different cross terms in an efficient way (especially $\mathcal{M}_u$). However, at the time it was implemented, the AD tool did not allow the more general approach that is described next.

3.3.2 General method for multizone problems

The desired (for generality) partitioning of the adjoint problem by zones, rather than by fields, makes the three-field FSI approach inadequate, since the mesh deformation and fluid problem are part of the same zone. In other words, the mesh coordinates become dependent variables (of the structural displacements) and their adjoints are no longer handled explicitly. Then, the fluid and structural fixed points are written as

$$\begin{aligned} \mathbf{w} &= \mathcal{F}(\mathbf{w},\mathbf{u}) \\ \mathbf{u} &= \mathcal{S}(\mathbf{u},\mathbf{w}) \end{aligned}. \tag{3.43}$$

However, for the reasons already mentioned, a combined recording of the fluid iteration and mesh deformation would be inefficient to evaluate. Similarly, the checkpointing approach is not adequate for problems where the number of inner iterations is small, which is the case of conjugate heat transfer (CHT) problems. Instead, it is more efficient to record all iterations. This requires either multiple tapes, or for the AD tool to allow partial evaluation of its global tape. Furthermore, in the latter case, due to the fixed-point nature of the primal iterations (which override the state variables), a custom way of referring to their input and output adjoint values is needed. These were the insights by Burghardt [101], which made the general approach possible. It consists of identifying the existence of cross terms, programmatically, from the interfaces defined by users of the code. The sum of cross terms is combined with the objective function gradient, for each zone, to form the right-hand-side for the inner iterations (with the diagonal blocks of the Jacobian). For example, for an FSI problem with two zones this process is equivalent to

$$\begin{aligned} \overline{\mathbf{w}} &= (J_w^{\mathrm{T}} + \mathcal{S}_w^{\mathrm{T}}\overline{\mathbf{u}}) + \mathcal{F}_w^{\mathrm{T}}\overline{\mathbf{w}} \\ \overline{\mathbf{u}} &= (J_u^{\mathrm{T}} + \mathcal{F}_u^{\mathrm{T}}\overline{\mathbf{w}}) + \mathcal{S}_u^{\mathrm{T}}\overline{\mathbf{u}} \end{aligned}. \tag{3.44}$$

The combined recording is organized, by sections, in a manner that allows evaluating diagonal terms separately from cross-terms (where mesh deformation is included). The **first** section is used to register all state variables, and desired parameters, as inputs of the global iteration. The **second** section contains the *dependencies* of all zones, these are relatively simple (but crucial) orchestrations of pre- and post-processing operations within each zone, that capture the cyclic nature of iterative methods. For example, eddy viscosity is needed at the start of a RANS iteration, however, this quantity results from the post-processing of the turbulence solver, which in turn, requires some quantities from the pre-processing of the flow solver, namely the primitive variables and their gradients. Without *dependencies*, it would not be possible to record just one iteration. This is one area of the adjoint framework that requires modification when a new solver is added. The **third** section contains the objective function, this also requires transferring data across all interfaces and deforming the mesh for zones that receive displacements, the *dependencies* of such zones need to be re-evaluated. To evaluate the gradient of the objective function the recording is stopped here, otherwise the third section is empty. The **fourth** section contains the transfer of data. Finally, one section per zone is dedicated to the respective primal iteration and to register the state variables as outputs.

To evaluate the product of a vector with a diagonal term, the adjoints of the outputs are first seeded with the vector, then, the corresponding section of the tape is evaluated, skipping the transfer of data, but evaluating the dependencies. The result of the product is then read from the adjoints of the inputs. To be able to extract also the cross terms for which this zone is responsible (by reading the adjoints of the inputs of other zones), the section for the transfer of data (and mesh deformation) is not skipped. To obtain the gradient of the objective function, or to evaluate the derivatives with respect to the parameters, all the relevant output adjoints are seeded and the entire tape is evaluated.

Because all domain-specific knowledge is encapsulated in abstractions, such as *dependencies*, *solvers*, transfer of data, etc., the framework can be used for an arbitrary number of zones (and thus physics). The advantage of a general strategy becomes more evident when considering aerothermoelasticity problems, where domain-specific approaches have required authors to contend with 9×9 block Jacobians [7].

The main contributions of the author to this framework (full details in [101]) were: a strategy to manage cross terms such that they can be updated with relaxation and in a Gauss-Seidel manner (rather than Jacobi, i.e. only after all zones are iterated); the use of GMRES within inner iterations; general tuning for specific FSI aspects; and refactoring the transfer strategy to allow overlapping zones, which allows combined FSI and CHT simulations.

Interface methods are not used at the moment, as the solution method described above has (thus far) proved more stable than the FSI-specific approach [3]. A generic way to convert the multizone iteration into an interface problem, would be to consider the cross terms of the last zone as the interface variables. This would require a programmatic way to identify interface indices (i.e. a way of inferring the sparsity pattern of each cross term), which preferably would not need domain-specific knowledge. A further challenge for such methods, in a multizone context, is to scale the different terms appropriately to avoid numerical issues.

The combined recording strategy does raise memory usage issues. In particular, recording the mesh deformation operation requires approximately as much storage as a typical RANS fluid

[3]The reason for this is unknown, both approaches were verified, and to some extent are equivalent. Insofar as it is simpler, the current implementation is less likely to be at fault.

problem, which would make the framework impractical even for medium scale problems. Therefore, currently this operation is not accurately differentiated with respect to the unreformed mesh coordinates. Instead, the stiffness matrix of the mesh deformation problem is assumed to be constant. Although this does not affect the adjoint solution, it nevertheless introduces errors in the shape sensitivities of problems with large deformations.

Numerical example

To verify the shape sensitivities for FSI problems, the simulation of a flexible wing at low Mach number (0.6) and 4 degrees angle-of-attack (AoA) is considered. The wing geometry shown in Fig. 3.7, is generated by *lofting* two symmetric 4-digit NACA profiles. The root profile has 0.25 m chord and 9% thickness, whereas the tip has 0.175 m chord and 7.2% thickness. The wing span is 1 m, with the 25% chord-line swept back 5 degrees, and a linear twist distribution of -3 degrees to prevent static divergence. The wing structure is approximated as a neo-Hookean hyper-elastic solid, with elasticity modulus of 7.5 GPa and Poisson's ratio of 0.35 With these properties the vertical displacement of the wing tip is approximately 20% of chord (see Fig. 3.8). On the fluid side, air at standard temperature and pressure is considered. Both fluid and structural grids are composed mostly of hexahedra, the former having 832 000 nodes and the latter 174 000 nodes. Although the fluid grid is not sufficiently fine for detailed flow field analysis, since $y^+ \approx 5$ and Menter's SST turbulence model is used without wall functions (which at the time of writing were not fully implemented in SU2), it is nevertheless adequate for verification of discrete adjoint sensitivities (whose accuracy is not mesh dependant). Convective fluxes are computed with a second order Roe scheme, the gradients are computed via the Green-Gauss theorem, and the MUSCL reconstruction of flow variables is limited by the Venkatakrishnan and Wang's limiter.

The free-form-deformation (FFD) box shown in Fig. 3.7 has 98 control points, but instead of verifying derivatives for each point (and each Cartesian coordinate), stretch (acting on chord) and translation variables were defined for each spanwise section of FFD points. These 14 variables will be referred to as S_{0-6} and T_{0-6} respectively (starting the numbering at the root), moreover their values are normalized by 20% of root chord (0.05 m). The gradients of drag coefficient (C_d) and elastic energy (or compliance) were verified, these were chosen because the first is fluid oriented, whereas the second is mostly structural. The values of these functions are also normalized (by 0.01 and 75 J respectively) which leaves both gradients dimensionless and of comparable magnitudes.

A suitable finite difference step size was determined by conducting a convergence study, starting with a step size of 0.05 and halving it each time, to obtain second order central approximations to the gradients. At step sizes below 0.0125, the approximations start to diverge due to the typical limitations of finite differences, in particular, the variation of the functions approach the

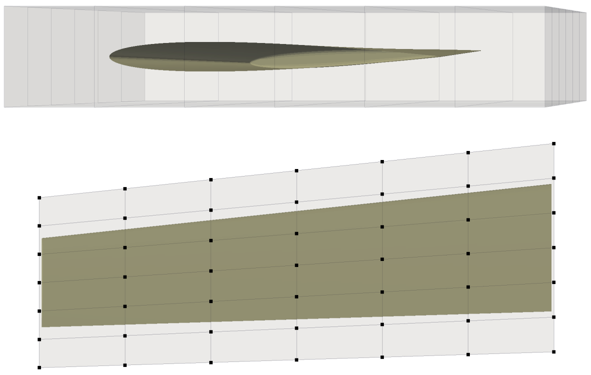


Figure 3.7: Wing geometry and free-form-deformation box, viewed from the wing tip and the top.

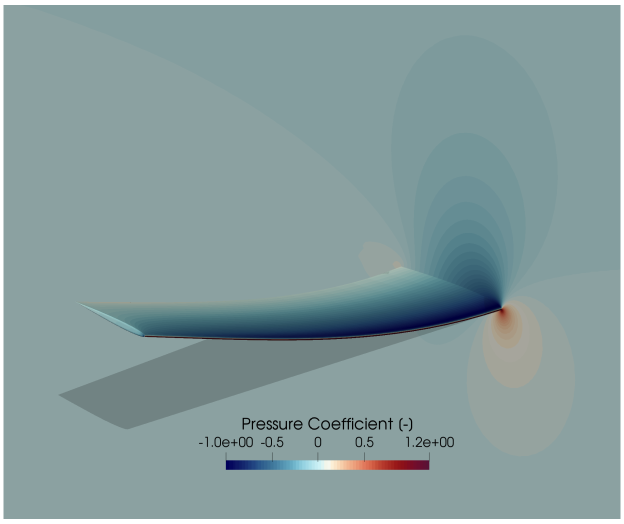


Figure 3.8: Deformed configuration of the wing and pressure coefficient contours.

accuracy to within which the primal problem can be converged ($\approx 10^{-6}$ change over FSI iterations). Therefore, gradients obtained with step size of 0.025, were considered for comparison with the discrete adjoint based derivatives. The error associated with the approximation, used to compute lower and upper confidence bounds, was estimated by taking the average plus three standard deviations of the differences between second and fourth order central approximations, across all variables (S_{0-6}, T_{0-6}). The fourth order values were computed by considering also the function values obtained at ± 0.05. The results for drag and elastic energy are plotted in Fig. 3.9 and Fig. 3.10, respectively.

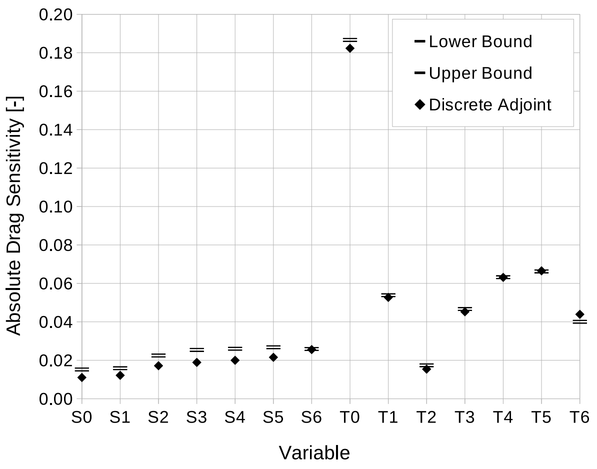


Figure 3.9: Absolute value of drag sensitivities, from the adjoint solver, and finite difference error bounds.

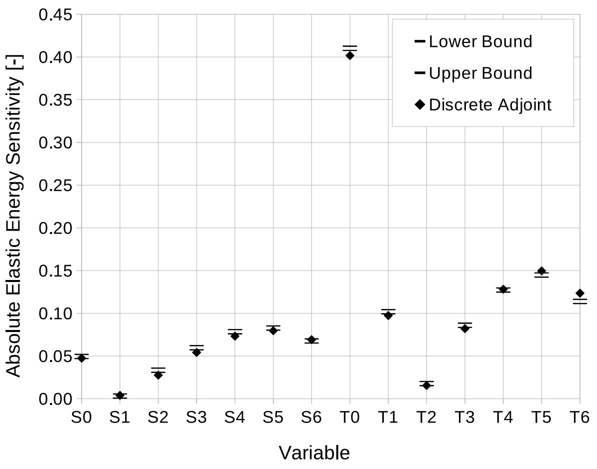


Figure 3.10: Absolute value of elastic energy sensitivities, from the adjoint solver, and finite difference error bounds.

The agreement between finite difference approximations, and adjoint sensitivities of drag, is better for the translation variables (T_{0-6}). To some extent this was expected due to the larger magnitude of those derivatives, and the way those variables are constructed from the FFD control points. Note that a translation variable moves all the points in a section by an equal amount, whereas a stretch variable, moves points away from the centerline of the FFD box (proportionally to their distance to this line). The overall agreement is also better for the elastic energy function, which was again expected, due to the strong dependence of this function on pressure and structural deformation (i.e. almost directly on the state variables) whereas drag at subsonic speed depends mostly on viscous effects (and therefore on gradients of the state variables).

Since it was not possible to obtain derivative approximations with a small-enough step size (the one used is 2.5% of the plausible range for use in optimization), the discrepancies cannot be attributed exclusively to the memory-saving approximations, and may be due in part to finite difference errors. Running this adjoint FSI problem requires 113 GB of memory, or approximately 4 times the memory required to run the primal simulation. Note, however, that while a multigrid method was used for the primal solution, it was not used for the adjoint solution, since that would roughly double the memory footprint of the fluid problem, which would increase the adjoint-to-primal memory ratio to ≈ 6. Notwithstanding not using multi-grid, the adjoint solution takes approximately the same time as the primal, i.e. ≈ 15 minutes running on four Intel Xeon E5-2650v4 CPU (48 cores in total).

Extension to unsteady problems

For completeness, it is worth discussing the extension of the general framework to unsteady problems (without this capability, the "general" designation would be inadequate). This has been an active area of research at Imperial and of SU2 development [119], but it is not an integral part of this work. Nevertheless, to obtain an unsteady adjoint method, it is convenient to consider the state, and adjoint variables, as the concatenation of all time iterations, that is

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}^0 \\ \vdots \\ \mathbf{x}^N \end{pmatrix},$$

where superscripts denote the time iteration. Using this definition in the adjoint fixed-point equations (3.20), and noting that in most time marching methods the current solution depends on one or two prior time iterations, results in

$$\begin{pmatrix} \boldsymbol{\lambda}^0 \\ \vdots \\ \boldsymbol{\lambda}^N \end{pmatrix} = \begin{pmatrix} J_{x^0}^{\mathrm{T}} \\ \vdots \\ J_{x^N}^{\mathrm{T}} \end{pmatrix} + \begin{bmatrix} \mathcal{P}_{x^0}^0 & & & & \\ & \ddots & & \mathbf{0} & \\ \mathcal{P}_{x^{i-2}}^i & \mathcal{P}_{x^{i-1}}^i & \mathcal{P}_{x^i}^i & & \\ & & & \ddots & \\ \mathbf{0} & & \mathcal{P}_{x^{N-2}}^N & \mathcal{P}_{x^{N-1}}^N & \mathcal{P}_{x^N}^N \end{bmatrix}^{\mathrm{T}} \begin{pmatrix} \boldsymbol{\lambda}^0 \\ \vdots \\ \boldsymbol{\lambda}^N \end{pmatrix}, \tag{3.45}$$

where the Jacobian is a block lower banded matrix. It follows, that the adjoint problem can be solved by backwards time marching, where the adjoint variables for a particular adjoint-time iteration, i, are given by

$$\boldsymbol{\lambda}^i = J_{x^i}^{\mathrm{T}} + (\mathcal{P}_{x^i}^{i+2})^{\mathrm{T}}\boldsymbol{\lambda}^{i+2} + (\mathcal{P}_{x^i}^{i+1})^{\mathrm{T}}\boldsymbol{\lambda}^{i+1} + (\mathcal{P}_{x^i}^{i})^{\mathrm{T}}\boldsymbol{\lambda}^{i}. \tag{3.46}$$

Note that the vector-Jacobian products at previous adjoint-time iterations ($i+2$ and $i+1$), are known from those iterations, and simply combined with the gradient of the objective function, to obtain the right-hand-side for the current iteration. The question, therefore, is what modifications, if any, are required to make the multizone strategy applicable. In other words, in what conditions can $\mathcal{P}^i$ be "simply" substituted by the steady state multizone adjoint method.

If it is assumed, with some loss of generality, that the primal iterations are of a form where each zone only uses its own previous state (despite depending on the current global state), that is

$$\mathbf{x}^i = \begin{pmatrix} {}^0\mathcal{P}(\mathbf{x}^i, {}^0\mathbf{x}^{i-1}, \dots) \\ {}^1\mathcal{P}(\mathbf{x}^i, {}^1\mathbf{x}^{i-1}, \dots) \end{pmatrix}, \tag{3.47}$$

where pre-superscripts denote the zone index. The off-diagonal blocks in (3.45) become block diagonal, and there are no multizone cross terms due to previous state values. For example,

$$\frac{\partial^0\mathcal{P}^i}{\partial^1\mathbf{x}^{i-1}} = \mathbf{0}. \tag{3.48}$$

Therefore, for this restricted form of iteration, the unsteady contributions to the adjoint right-hand-side, can be managed by each zone independently of the multizone method (for example, by combining them with the gradient of the objective function). Note that the vector-Jacobian products responsible for these contributions, are obtained by treating the previous state values as parameters of the current iteration. Moreover, they should be managed by the adjoint solvers in each zone, since they are domain-specific (i.e. the time integration method may not be the same in all zones). One example of loss of generality, is the approximation of grid velocities by finite-differences, which cannot be used, since this would cause the fluid iteration to depend on previous values of mesh coordinates, which in turn, are a function of previous structural displacements. However, this can be avoided by computing fluid grid velocities from structural velocities, which are part of the current state of the structural problem, and thus, contribute to multizone cross terms analogously to structural displacements in steady-state problems. Another example, for which a solution has yet to be proposed, is the geometric conservation law that depends on volumes at the previous time step (when time derivatives are computed using second order backward differences).

Finally, note that spectral time-domain methods, such as harmonic balance, are not solved by marching in time, notwithstanding, the multizone method is clearly applicable to each time instance.

Chapter 4

Efficient Implementation of Unstructured Solvers

The previous chapters presented the simulation and optimization methodology used herein. While their focus is on methods, considerations regarding the software implementation were also justified. For example, in chapter 3, it is rather evident how discrete adjoint implementation choices can limit the suitability of the resulting software to large-scale problems. Indeed, performance is itself a tool, more than an objective, in the sense that it enables *more*: more robust solution methods, able to deal with more challenging applications; larger and richer problems, resolved in more detail; robust optimizations with larger design spaces, targeting wider operating envelopes. The examples go on, but in summary, performance is not just a complement for methods, for it is also a requirement to make their use possible.

In principle, obtaining good performance for particular operations is not a hard problem, in the sense that the capabilities and limitations of modern hardware and programming languages are well known and have been described at length. However, these are not mentioned in the vast majority of mathematical modeling books, which are focused primarily on methods. To some extent, this separate exposition of algorithms and implementation aspects is logical (in fact, it could be seen as good software engineering). However, performance results from the interaction between the implementation of a method (subject to the constraints of a programming language) and the capabilities of the target hardware. In this sense performance is multidisciplinary, and therefore, in analogy with physical engineering systems, it may not be possible to predict it without considering both fields. Moreover, doing so is nearly impossible for a complex software due to the trade-offs between different algorithms. However, contrary to physical systems, it is inexpensive to measure the performance of a software. Therefore, a high performance research code, that hopes to be long-lasting, should be implemented in the most generic way possible. So that it may be reconfigured (rather than re-written) to better suit particular algorithms or hardware platforms.

This chapter deals with efficiency and *generic programming* aspects applied to the algorithms of unstructured solvers, whose characteristics allow only a small subset of software and hardware aspects to be considered to propose good implementations. It is then shown that data layouts are responsible for the trade-offs between different types of algorithm, and how *generic programming* allows the latter to be implemented independently of the former. This is covered in section 4.1, which sets the background to discuss unstructured solver algorithms in more detail in section 4.2. Then, section 4.3 is dedicated to parallelization strategies for those algorithms. Finally, section 4.4 shows how these widely applicable concepts were used to improve SU2, thereby enabling the computationally expensive optimization problems covered in other chapters. This was, in fact, the main motivation (or at least justification) for the numerous contributions (over 1300 commits[1]) the author has made to SU2. By and large, SU2 is known for its low-order finite volume solvers, which have, of course, heavily influenced the discussion in this chapter. Nevertheless, important differences with high-order finite element methods are noted when appropriate, even though their specific aspects are not discussed.

4.1 Capabilities and limitations of hardware and software

Before restricting the discussion to specific algorithms of unstructured solvers, it is necessary to discuss what modern hardware and programming languages (and their compilers) can do for, or against, each other.

4.1.1 Hardware

CPU's, and to a much larger extent GPU's, thrive on applying a small set of instructions (preferably one) to data held in high levels of the cache hierarchy (preferably in register). Any deviation from this will cause a loss of performance, and one deviates from it mainly by asking the processing unit to operate on cold data (not in cache) or by branching, i.e. making the next instructions conditional on the results of current computations. Branching hurts performance by interfering with the instruction pipeline. The complete execution of one instruction requires multiple steps that take place over a number of clock cycles (around 10) that define its latency. The throughput is increased by having multiple execution units for some instructions and by staggering (i.e. pipelining) their execution, such that all stages of the pipeline are active in every clock cycle, performing the different steps for the multiple instructions in flight. One of the key differences between CPU and GPU's, is that the former are designed to minimize the overall latency, therefore most of their resources are dedicated to caching data and attempting to predict the outcome of branches, to then speculatively execute the most probable execution

[1] https://github.com/su2code/SU2/graphs/contributors

path. Conversely, GPU's are designed for throughput, which is achieved by having many execution units controlled by comparatively simple scheduling units.

Memory subsystems are also much more efficient at providing continuous streams of data, than at satisfying scattered accesses to it. One reason for this is that memory is accessed by lines rather than one element of data at a time, that is, to access a single element (e.g. an integer) the processing unit loads the entire cache line (64 bytes) that contains it. Then, accessing the other elements in that line is much less expensive. This is a form of spatial locality. Another reason is that processing units, especially CPU's, are able to detect regular access patterns and exploit them by automatically prefetching data. In modern hardware the balance between compute and bandwidth is largely skewed towards the former, with processing units being capable of more floating point operations per second than what can be streamed out of memory, by a factor of $O(100)$. Thus, it is usually much more important to conserve bandwidth than computation, and to take advantage of the cache hierarchy to hide the latency of accessing main memory, which can be achieved via temporal locality, i.e. accessing recently used data that is likely to still exist in cache.

Similarly to having multiple cores and multiple execution units for some instructions, those units are generally capable of operating on wide data types. This single-instruction-multiple-data (SIMD) paradigm (commonly called vectorization) consists in applying the same instruction to multiple scalars (integers, floats, doubles) simultaneously, and with the same latency and throughput of processing a single element. However, this does not happen automatically, the compiler must issue this type of instruction rather than single element versions.

4.1.2 Programming languages and compilers

From the previous discussion it is evident that minimal indirection is desirable in the machine code generated by the compiler, i.e. the instructions, but also in how data is accessed. Then, the question is how to use the programming language to give the compiler the maximum amount of information, and how to store data to use the available bandwidth most efficiently. This discussion will be around C++ since it is used in SU2, and also because it is a language that allows close access to the hardware while providing language features that allow writing generic code.

The common cause of indirection is lack of compile-time information. Consider the example in listing 4.1. Besides the values of the data, the compiler does not know the size (n) of the arrays (x,y) and it does not known if they overlap (formally this is known as aliasing). Therefore, the compiler has to check that the size is not zero, check if the overlap is smaller than the vectorization width and if so handle one index at a time, otherwise it will vectorize, i.e. compute multiple indices with a single instruction, but only the part of the loop that is divisible by the vectorization width, handling the remainder with progressively narrower instructions.

```
void daxpy(int n, double a, const double* x, double* y) {
    for (int i=0; i<n; ++i)
        y[i] += a*x[i];
}
```

Listing 4.1: "AXPY" operation.

For large array sizes, the overhead of the extra checks is negligible, however, for small sizes it is significant. This is not only due to the checks themselves, but also because of the redundant instructions which pollute the instruction cache. Note that instructions are also data loaded by the processor. This issue is aggravated by more aggressive optimizations such as loop unrolling, to a point where it is possible for the code to perform worse with more optimization.

Small loop optimizations

Small loops are prevalent in numerical simulation, for example, over the number of degrees of freedom per node, or over the number of spatial dimensions. Returning to the axpy example, if the function is defined inline and the size is known at compile-time, e.g. it is a literal at the call site (daxpy$(3, a, x, y)$) the redundant instructions will not be generated. Likewise for the aliasing checks when the compiler can prove it will not occur, for example for two stack-allocated variables. To force the size to be a compile-time constant, it could be made a template parameter, as shown in listing 4.2.

```
template<int n, class Scalar, class Array>
void axpy(Scalar a, const Array& x, Array& y) {
    for (int i=0; i<n; ++i)
        y[i] += a*x[i];
}
```

Listing 4.2: Template "AXPY" operation.

Now, every time the function is called, the template is instantiated for a known size. However, templates are notorious for spreading, that is, the client of the axpy template would very likely also become a template, itself instantiated for the possible values of the size. At some point the spreading needs to be stopped, usually when the cost of choosing the right instantiation has been amortized over many calls to functions that benefit from the known size. One common mechanism to achieve this for free functions is shown in listing 4.3.

```
// Templatized implementation of the function would go here.
template<int n> void f();

// This should be defined in a cpp, and not inline, so that templates are instantiated
   only once.
void f(int n) {
    switch(n) {
        case 2: f<2>(); break;
        case 3: f<3>(); break;
        default: assert(false); break;
    }
}
```

Listing 4.3: Using a wrapper function to stop templating from spreading.

A particularly non-intrusive way, in that it avoids this type of template boilerplate, to inform the compiler about a variable with only two possible values (e.g. 2 or 3 spatial dimensions), is to state it in the body of the function, as listing 4.4 suggests. Then, the compiler should take advantage of this information for the inline functions called after the statement. When the function is a method of a class, and the size is a member instead of an argument, it is still important to make this statement for each method. The templating alternative gives the additional flexibility of using the size for stack allocated arrays, for which it is easiest for the compiler to prove that there is no aliasing, and thus that vectorization is safe.

```
void f(int n) {
    // In practice this logic should be encapsulated in an inlinable type or function.
    n = n>2? 3:2;
    assert(n==2 || n==3); // requirements should be asserted
    ...
}
```

Listing 4.4: A portable compiler hint.

In fact, assuring the compiler that aliasing will not occur, and thus that it is safe to vectorize and possibly fuse different loops, is not trivial to a point where this last optimization capability is rare among compilers. One possibility is to copy the inputs into local (i.e. on the stack) arrays, of course, this is not practical for large sizes (and it is cumbersome for small ones). Another is to use compiler-specific attributes, to annotate either the variables or the loops with information about aliasing, alignment (e.g. the start of the array coincides with a cache line), etc. However, a more portable alternative (i.e. not compiler-specific) is to use the OpenMP SIMD directive, as listing 4.5 shows.

```
void daxpy(int n, double a, const double* x, double* y) {
    assert(max(x,y) - min(x,y) > n || x==y); // this becomes a requirement
    #pragma omp simd
    for (int i=0; i<n; ++i)
        y[i] += a*x[i];
}
```

Listing 4.5: "AXPY" operation, with SIMD directive.

Together with size hints, this already allows functions with small loops to be well optimized, because for those, loop fusion is not as relevant as for large loops, where temporal locality is an important performance factor. This type of low level performance claim requires some empirical evidence to support it. Consider the typical operation of computing the tangential projection ($\mathbf{u}_t$) of a vector ($\mathbf{u}$) onto a plane defined by a normal vector ($\mathbf{n}$), given by

$$\mathbf{u}_t = \mathbf{u} - \frac{\mathbf{u} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}} \mathbf{n}.$$

For simplicity it is assumed that the signature for the function that performs this action for a number of vectors is

```
template<class Matrix>
void tangentComponent(const Matrix& n, Matrix& ut);
```

where the argument "ut" contains $\mathbf{u}$ on entry and $\mathbf{u}_t$ on exit. For reasons that will be discussed later, the Matrix type is assumed to use row-major storage, that is, the 2 or 3 components of each vector are contiguous in memory. The implementation uses two dot products and one axpy operation per pair of vectors. Three versions of the code were tested, on version A the number of components is not known and SIMD directives are not used, on version B the number is known to be either 2 or 3 and SIMD directives are used, version C additionally restricts the number of components to 3. The code was compiled with g++ 7.5 for the Skylake architecture with varying level of compiler optimization, namely O2, O2+unrolling, and O3+unrolling. The code was run in parallel (4 cores) and sequentially, on a machine with 2 memory channels, for two matrix sizes, one that just fits in L3 cache and another 100 times larger. The program was run 10000 and 100 times, respectively, so that the total number of computations was the same for the two sizes. Tables 4.1 and 4.2 show the execution time for the small and large sizes respectively. The execution time is normalized by the best over all combinations.

Version	Parallel	O2	O2+Unroll	O3+Unroll
A	No	7.7	12	13
B	No	6.0	4.0	7.3
C	No	8.8	3.2	4.0
A	Yes	2.2	3.5	3.8
B	Yes	1.7	1.2	2.2
C	Yes	2.6	**1**	1.3

Table 4.1: Effect of small loop optimizations on small matrix size.

This simple example shows many of the aspects addressed so far. All versions of the code eventually lose performance, once the optimization level becomes "excessive" for the level of information the compiler is given about the algorithm. Once the bandwidth limit is reached, all versions of the code perform similarly at all levels of optimization. This limit is reached for large matrices in parallel, note how the speedup gained from running the code in parallel is much less than expected. The variation in performance is small because even the superfluous computations (branches, etc.) take place while the processing unit waits for data. This clearly

Version	Parallel	O2	O2+Unroll	O3+Unroll
A	No	8.1	13	13
B	No	6.5	5.4	7.9
C	No	9.3	5.1	5.8
A	Yes	4.5	4.7	4.6
B	Yes	4.5	4.5	4.6
C	Yes	4.6	4.5	4.6

Table 4.2: Effect of small loop optimizations on large matrix size.

shows the importance of prioritizing bandwidth over computations. The situation is very different when the processing unit operates on cached data, in that case, the code is compute-bound and the implementation and compiler optimizations have a significant effect. As a first approximation, an algorithm can be judged to be bound by compute or bandwidth by comparing its FLOP-to-byte ratio against what the hardware is capable of. Essentially this consists of dividing the total number of operations by the amount of data retrieved from main memory. This analysis can be difficult if the effects of spatial and temporal locality are considered, in fact, given the simplicity of these small loop optimizations, it takes less time to simply apply them than to accurately predict their effect.

Large loop optimizations, advanced language features

Loop fusion

The optimizations described earlier do not address loop fusion, which is important for large loops due to temporal locality effects, and they do not address a few code quality issues in the examples. First, arrays are being passed by pointer, this is sub optimal because memory is a resource, and as such, it should be contained in a class that manages it during the lifetime of the object. By doing so, the requirements on aliasing, alignment, padding (rounding up the size to a convenient multiple), and any other implementation detail, are passed alongside the data and centralized in the class, which is responsible for guaranteeing them. This encapsulation is especially important for higher dimension arrays (e.g. matrices), as it allows hiding (from the user) the most important detail, namely how the different dimensions are flattened into linear addressing space (e.g. $row * columns + column$). The importance of this encapsulation will be even clearer in later sections. Second, the interface begs users to implement the loops themselves. Consider, for example, the steps required to compute $\mathbf{u} = 2\mathbf{x} + \mathbf{y} - \mathbf{z}$ using only copy and axpy operations. In either case the resulting code is not very readable, and the proliferation of custom loops becomes a maintenance problem (for example, consider the effort of refactoring them to include the SIMD directive).

The solution to obtain loop fusion and improve code quality, is to define a vector type and to use meta programming (namely expression templates) to automatically convert natural syntax ($\mathbf{u} = 2\mathbf{x} + \mathbf{y} - \mathbf{z}$) into specialized vector operations. Expressions are symbolic representations of operations (e.g. $\mathbf{x} + \mathbf{y}$) rather than their result, they can be assembled cheaply (to represent complex operations) and evaluated as a whole only when the final result is needed, i.e. without creating temporaries. The starting point is a base class for all expressions, with a mechanism to convert it to its derived type, namely the curiously recurring template pattern (CRTP), as shown below.

```cpp
template<class Derived, class Scalar>
struct VecExpr {
  // Cast the expression to Derived, usually to allow evaluation via operator[].
  const Derived& derived() const { return static_cast<const Derived&>(*this); }
};
```

Then, the vector type can derive from this base and implement the assignment operator (=), responsible for populating the vector based on any expression.

```cpp
template<class Scalar>
class Vector : public VecExpr<Vector<Scalar>,Scalar> { // CRTP
private:
    Scalar* data_ = nullptr;
    size_t size_ = 0;
    size_t paddedSize_ = 0;
public:
    // Align should be the SIMD width divided by sizeof(Scalar)
    static constexpr size_t Align = 4;
    static constexpr bool StoreAsRef = true; // explained later

    // aligned allocation of data, padding it to a multiple of "Align"
    void resize(size_t n);

    // operators to access data
    Scalar& operator[] (size_t i) { return data_[i]; }
    const Scalar& operator[] (size_t i) const { return data_[i]; }

    // the vector populates itself from any (T) expression of the same Scalar type
    template<class T>
    Vector& operator= (const VecExpr<T,Scalar>& expr) {
        // the size is guaranteed to be a multiple of Align
        for (size_t i=0; i < paddedSize_/Align; ++i)
            // vectors cannot overlap partially
            #pragma omp simd
            for (size_t j=0; j<Align; ++j)
                // the expression is evaluated for each index via [], as if it were a
    vector
                data_[i*Align+j] = expr.derived()[i*Align+j];
        return *this;
    }

    // quirks of C++
    Vector& operator= (const Vector& other) {
        return *this = static_cast<const VecExpr<Vector,Scalar>&>(other);
    };

    // other methods, including constructors and destructor
    ...
};
```

A reference to this vector type is convertible to a reference to its base class, due to inheritance, and vice versa due to the CRTP. An expression to add any two expressions, is also derived from the base expression class, and can be implemented as follows.

```
template<class U, class V, class Scalar>
struct VecAdd : VecExpr<VecAdd<U,V,Scalar>, Scalar> {
    store_t<const U> u;
    store_t<const V> v;
    static constexpr bool StoreAsRef = false;
    // the expression is contructed with left and right hand sides
    VecAdd(const U& u_, const V& v_) : u(u_), v(v_) {}
    // the implementation of the operation is trivial
    Scalar operator[] (size_t i) const { return u[i] + v[i]; }
};
```

Note that the left and right hand sides need to be stored either by reference if they are vectors (to avoid copying data), or by value if they are expressions. This is necessary because the expressions themselves are temporaries (formally r-values) which do not have an address and thus cannot be stored by reference. When the final expression is evaluated, the intermediate ones no longer exist since they have gone out of scope. One way to control the storage mode is via a trait ("StoreAsRef" in the examples) which informs the expression how to store a given value. For the vector type this is true, while for expressions it is false. This makes the (fair) assumption that vectors will not be created as r-values, nevertheless, an alternative will be discussed shortly. A possible meta programming mechanism to conditionally add an l-reference to a type based on the "StoreAsRef" trait is

```
// Base case, a struct template where reference is not added to the template type (T)
template<class T, bool> struct add_lref_if { using type = T; };
// Partial (boolean parameter) template specialization that adds the reference to T
template<class T> struct add_lref_if<T,true> { using type = T &; };

// Convenient way to simplify the syntax
template<class T> using store_t = typename add_lref_if<T,T::StoreAsRef>::type;
```

Finally, to convert natural syntax to an expression, the operator for addition needs to be overloaded for expression types, that is

```
template<class U, class V, class S>
VecAdd<U,V,S> operator+ (const VecExpr<U,S>& u, const VecExpr<V,S>& v) {
  return VecAdd<U,V,S>(u.derived(), v.derived());
}
```

The alternative to the "StoreAsRef" mechanism, is to overload the operator for all combinations of l-value and r-value references, and explicitly force the latter to be copied into the expression. However, this becomes rather verbose, especially if one considers also the overloads required for operations between vectors and scalars (e.g. $2\mathbf{u}$).

Given the simple and repetitive nature of the expression classes and operator overloads, code generation techniques are used to generate them, e.g. from a master macro, and code is not repeated. Another important detail is that in the example, the evaluation operator ([ ]) for the addition expression forces a conversion to "Scalar". However, if that type has its own expression templates (for example it is an AD type), they are defeated by this conversion. The implementation by the author in SU2[2] considers these aspects. Indeed, allowing this nesting of expressions is what justified a custom implementation, rather than adopting an existing library. It should be noted, however, that the code complexity grows rather quickly to implement level 2 and 3 BLAS operations, for example it is important to exploit associativity to reduce costs, and to convert some expressions into calls to standard BLAS functions to maximize performance.

To summarise, what was achieved by the vector class and the expression templates, is loop fusion and vectorization in a compiler-independent way. More importantly, the public interface for these operations is highly readable, and from a maintenance point of view, all implementation details are encapsulated in a class. This means, for example, that to explore non-temporal store instructions only one loop needs to be modified (the one in the assignment operator), likewise to add OpenMP work-sharing directives to parallelize operations with vectors. It is not necessary to benchmark this example, since it provides an algorithmic improvement over multiple calls to axpy-type functions to modify one vector, that is, the compute intensity is increased by traversing the target vector only once.

Loop tiling

While loop fusion is a way to optimize multiple loops, loop tiling is a way to optimize nested loops. Both improve performance by reducing the frequency with which main memory is accessed, thereby increasing the effective FLOP-to-byte ratio.

Consider the multiplication of a narrow matrix, or a list of vectors, by its transpose on the left, which occurs for example in linear least squares fitting, that is

[2] https://github.com/su2code/SU2/blob/v7.1.1/Common/include/linear_algebra/vector_expressions.hpp

$$\mathbf{M} = \mathbf{A}^{\mathrm{T}}\mathbf{A}. \tag{4.1}$$

Naive implementation of this multiplication would involve three nested loops to compute $n(n+1)/2$ dot products (n is number of columns). This requires each column (i.e. the entire matrix) to be traversed n times. If columns do not fit in cache, they need to be retrieved from main memory every time. Loop tiling avoids this by splitting the innermost loop (the dot products) into blocks that fit in cache. Then, a new outer loop over all blocks performs multiple updates of the result ($\mathbf{M}$ in this case) as shown is listing 4.6.

```
for (int begin = 0; begin < end; begin += tileSize) {
    for (int i = 0; i < cols; ++i)
        for (int j = 0; j <= i; ++j)
            for (int row = begin; row < min(rows,begin+tileSize); ++row)
                M(i,j) += A(row,i) * A(row,j);
}
```

Listing 4.6: Tiled multiplication of two narrow matrices.

This loop transformation allows the matrix to be retrieved from main memory only once (rather than n times), by moving one block at a time into cache. However, note that the output is now accessed multiple times, this is not an issue so far as $\mathbf{M}$ fits in cache (hence the choice of narrow $\mathbf{A}$ for this example). Conversely, when $\mathbf{M}$ is large, more tiling loops are needed to reduce also the frequency of writes to main memory.

Loop tiling is what allows level 3 BLAS operations to be bound by compute rather than bandwidth. There the technique is taken further to also take advantage of the differences between cache levels. For operations and data formats that are (or can be manipulated into being) compatible with BLAS routines it is wise to use a BLAS library. However, this type of nested loop occurs in other contexts, for example radial-basis-function interpolation (without compact support) where each interpolated value depends on all sample values. Another example is the GMRES method where operations akin to (4.1) are used to generate the vectors of the basis. Furthermore, it is sometimes more efficient to represent a matrix as a list of vectors (i.e. multiple blocks of memory, rather than one) which is not compatible with BLAS but allows swapping columns or rows quickly, without copying data, simply by swapping the corresponding pointers. An example of loop tiling applied to the block-structured CFD code ADFlow has been recently given in [120].

Encapsulation and insulation

The previous vector class is an example of encapsulation, that is, a number of implementation details were centralized in a few key places, in a way that is transparent to the compiler and thus fully optimizable. On the other hand, insulation refers to mechanisms whereby details are hidden from the compiler. Therefore, such mechanisms can occur at runtime. One example of this is the mechanism of listing 4.3, where the implementation of the dispatch function does not need to be available at the call site. This is why that type of strategy can be used to stop templates from spreading. However, a more classical example is polymorphism, which consists of using virtual functions to create abstract interfaces to objects whose concrete type is only known at runtime. One common application of polymorphism, is to allow the user of a library to extend its functionally by implementing only some details. Because polymorphism is a runtime mechanism, the library does not need to be recompiled, which allows it to be closed source software. This type of runtime flexibility is also important for very large code-bases to allow continuous deployment, where individual elements (libraries) of the application can be compiled in isolation without affecting all others. Typical scientific codes do not fit this description, nevertheless, providing some dedicated way for users to add custom functionality is not unreasonable. Moreover, insulation is clearly important to manage templates (and thus compilation times) and good abstractions (i.e. expressive) can make the high-level architecture easier to understand.

As an example, consider allowing users of the vector type defined earlier to create custom types compatible with the vector expressions. These could be used to represent functions, implement out-of-core objects (whose data does not reside in main memory), etc. The abstract interface for this is clearly the square bracket operator, that is,

```cpp
struct AbstractVector : VecExpr<AbstractVector,double> {
    static constexpr bool StoreAsRef = true;
    // Derived classes will have to override this method
    virtual double operator[] (size_t) const = 0;
};
```

Now, any function of the library that accepts an AbstractVector (by reference or pointer) can be passed a user defined object at runtime, for example,

```cpp
class UserDefinedVector final : public AbstractVector {
public:
    double operator[] (size_t i) const override { return i; }
};
```

Although this example is somewhat contrived, it clearly illustrates this feature of the language and allows discussing its implications. The main one is that even a simple function like

```cpp
void copy(const AbstractVector& src, Vector<double>& dst) { dst = src; }
```

will no longer be vectorized. This happens because when the template assignment operator of Vector is instantiated, the implementation of the user defined type is not known, hence it cannot be inlined and optimized. Instead, the compiler needs to retrieve the correct function pointer from the virtual table of the object and call it. This costs three levels of indirection versus one level for a direct function call. Furthermore, even if the compiler were aware of the implementation of UserDefinedVector, the only optimization it could legally perform, is to inline it speculatively. Then, at runtime, the concrete type of the object is checked, to decide between the speculative inline or the virtual call procedure. At that point it is up to the processor to notice (i.e. predict) which branch is being taken repeatedly. Indeed, the only way a virtual method can be perfectly inlined, is when it is known to be "final" for the type it is called on. That is, it would not be possible for a more derived class to override its implementation. In the present example this happens for UserDefinedVector since it is marked final (however, note that the base expression is only aware of AbstractVector, hence even a reference to UserDefinedVector would be upcast to AbstractVector by the expression templates). Evidently, using virtual functions for point-wise access to data, incurs significant overhead and reduces the optimizability of the code. Therefore, their use should be reserved for large methods (e.g. that perform significant computation), or amortized by accessing large data sets, or made optimizable. That is, similarly to the small loop optimization strategies, it may be important (for performance) to specialize some functions to operate on final classes rather than on the abstract base.

In summary, encapsulation and insulation are about creating compile-time and runtime interfaces, respectively. Moreover, insofar as they avoid code repetition, these are the main tools to write code that is extendable, maintainable, expressive, and generic (in the particular case of encapsulation). Any code repetition is difficult to modify. If complex conditional expressions are repeated, some of the occurrences either have bugs, or will have them after the first modification. The comments on such code snippets are out of date, by definition, documentation only remains actual if made enforceable by the compiler, i.e. a class name, function signature, concept, etc. For example, if a comment expresses an action, there is a good chance it is either redundant or what comes after it should be a function. It is particularly important for portability to encapsulate third party libraries (i.e. wrap them) to allow disabling or customizing those interfaces, this also applies to OpenMP directives (or any other pragma) which can be wrapped using macros. Furthermore, encapsulation provides a way to automatically profile the code for subsequent optimization, in functions with multiple "steps" (which are an anti-pattern) it may not be obvious at first which "step" is the most intensive. In the long run everything is made easier by using good abstractions to compartmentalize key operations, however small.

Generic programming

"Generic programming is an approach to programming that focuses on designing algorithms and data structures so that they work in the most general setting without loss of efficiency." [121]

The classic generic programming example is the standard template library (STL). This library provides a number of algorithms (sort, min_element, partition, etc.), and data structures (vector, list, map, etc.) which are made compatible with each other via the iterator concept.

Iterators are a way to encapsulate the details of how each data structure organizes its data in memory, they form a transparent interface, rather than abstract and opaque. Therefore, when a generic (template) algorithm is instantiated for a specific data structure, it is optimized to take advantage of how it stores its data in memory. Those algorithms can be extended by proving user-defined details via function objects at compile time, thereby allowing both to be optimized together.

Returning to the earlier example of the function

```
void tangentComponent(const Matrix& n, Matrix& ut);
```

it was mentioned that it could be implemented via dot products and axpy operations, which can be implemented (naively) as

```
double ddot(int n, const double* u, const double* v) {
    double r = 0;
    for (int i=0; i<n; ++i) r += u[i]*v[i];
    return r;
}

void daxpy(int n, double a, const double* x, double* y) {
    #pragma omp simd
    for (int i=0; i<n; ++i) y[i] += a*x[i];
}
```

Then, assuming that Matrix stores doubles in row-major order, a possible implementation of tangentComponent is

```
void tangentComponent(const Matrix& n, Matrix& ut) {
    for (int i = 0; i < n.rows(); ++i) {
        const double un = ddot(n.cols(), &n(i,0), &ut(i,0));
        const double nn = ddot(n.cols(), &n(i,0), &n(i,0));
        daxpy(n.cols(), -un/nn, &n(i,0), &ut(i,0));
    }
}
```

This is not generic because the algorithms are tied to a particular data layout, if the storage order of Matrix were changed to column-major, the elements in each row would no longer be adjacent in memory. The code would compile and run without crashing, but the results would be incorrect. What is needed are algorithms that are only bound to an interface, for example

```
template<class Interface>
auto dot(const Interface& u, const Interface& v) {
    typename Interface::Scalar r = 0;
    for (int i=0; i<u.size(); ++i) r += u[i]*v[i];
    return r;
}

```

```
template<class Interface, class Scalar>
void axpy(Scalar a, const Interface& x, Interface&& y) {
    #pragma omp simd
    for (int i=0; i<x.size(); ++i) y[i] += a*x[i];
}
```

and for Matrix to provide a lightweight object (the interface) that represents its rows, for example

```
template<class T>
struct RowInterfaceColMajor {
    using Scalar = T;
    Scalar* data;
    int cols, rows;
    RowInterfaceColMajor(Scalar* d, int c, int r) : data(d), cols(c), rows(r) {}
    int size() const { return cols; }
    Scalar& operator[] (int i) { return data[i*rows]; }
    const Scalar& operator[] (int i) const { return data[i*rows]; }
};
```

Then, tangentComponent can be implemented in a more generic way

```
void tangentComponent(const Matrix& n, Matrix& ut) {
    for (int i = 0; i < n.rows(); ++i) {
        const auto un = dot(n.row(i), ut.row(i));
        const auto nn = dot(n.row(i), n.row(i));
        axpy(-un/nn, n.row(i), ut.row(i));
    }
}
```

Furthermore, if the interface provides expression templates, the axpy function can be replaced by

```
ut.row(i) -= un/nn * n.row(i);
```

Any size hint would now be part of the row interface, for example, the "row" method of Matrix could have different template overloads to allow specifying a fixed size, two possible sizes, or no known size. This encapsulation also allows the algorithm to be tailored to some characteristics (formally traits) of the interface. For example, when data is not contiguous, and vectorization is forced via the SIMD directive, the performance may be worse than without vectorization. Similarly, dot products can be vectorized by carrying multiple partial sums over the SIMD width, and reducing them to a single scalar at the end. The overhead of this final reduction makes vectorization counterproductive for small sizes. The SIMD directive accepts an "if" clause, since version 5.0 of the OpenMP standard, which the axpy function could exploit by adding a "Contiguous" trait to the interface, that is,

```
template<class Interface, class Scalar>
```

```cpp
void axpy(Scalar a, const Interface x, Interface y) {
    #pragma omp simd if(Interface::Contiguous)
    for (int i=0; i<x.size(); ++i) y[i] += a*x[i];
}
```

Note that, in practice, traits should be defined in dedicated classes. This allows a base case to be defined with suitable defaults, and it allows specializations for native types (e.g. pointers) or from third party libraries (e.g. std::vector).

The encapsulation of details required for generic programming does not introduce runtime overhead. In fact, it allows generic algorithms to be better optimized for specific applications, by providing the compiler with more information. It should be evident from the discussion thus far, that informing the compiler is the only way to obtain performance. General abstract code cannot be optimized to the same extent. Furthermore, by replacing assumptions with expressive interfaces, the code is made more readable through the concepts expressed by them. The wrong preconception that high performance code must be hard to read, can only result from exercises in manual inlining of implementation details. Of course, diving into the machinery of generic programming will require progressively richer C++ vocabulary (which has been made simpler by newer versions of the standard) and some knowledge of the software architecture and idioms, lest the concepts be missed for the language details.

4.1.3 Effects of storage layout

The examples used so far already alluded to the importance of how data is stored in memory, for example, vectorization is most effective if data is contiguous. Indeed, it is for this topic that the interaction between software and hardware is more tightly coupled. Whereas for previous examples it was relatively straightforward to predict the effect of changes to the code, the effects of changing the layout depend on how the data structure is used, which makes it the principal contributor to the aforementioned trade-offs between algorithms.

Storage layout refers to how a multi dimensional address space (e.g. of a matrix) is flattened into the one dimensional virtual address space. Because of this characteristic of computer systems, only one dimension can be contiguous in memory, thus, advancing along non contiguous indices requires longer strides. The assumption that this type of n-D to 1-D mapping is known at compile time, usually implies that the data structure allocates a single block of memory. For data structures that do not meet this characteristic, i.e. allocate multiple blocks, the discussion that follows only applies within each block. Moreover, the focus will be on the narrow data structures that are characteristic of CFD solvers, which store a fixed amount of data per solution location, much smaller than the total number of locations. Fig. 4.1 shows the typical ways to map a pair of indices (i, j) to memory, this visual aid will also be helpful when discussing sparse storage formats or higher dimension arrays.

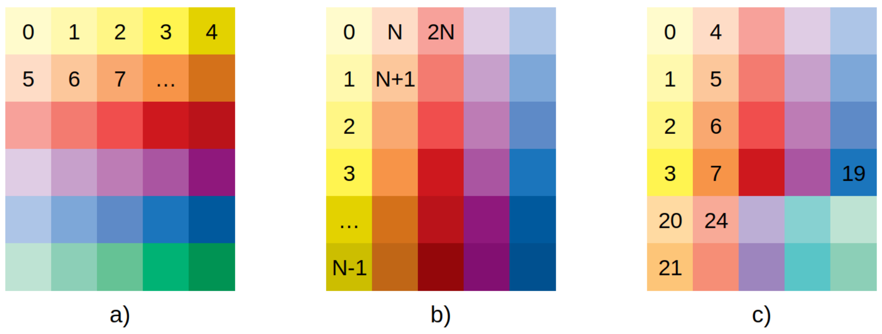


Figure 4.1: Types of 2-D layout. a) Row-major or array of structures; b) Column-major or structure of arrays; c) Array of structure of arrays.

Sequential and scattered access patterns

Sequential access consists in traversing the data structure along its outer index, using most of the values associated with inner indices in some computation. The function used in the examples of section 4.1.2 is an example of this.

Scattered access patterns arise when the order in which the outer indices are accessed, is given by some map or permutation of the natural, sequential, order. Insofar as they imply indirection, scattered accesses are slower, it is worth showing how much. To that end, the "tangentComponent" function is modified to read the target row index for each loop iteration from a permutation array.

```
void tangentComponent(const std::vector<int>& idx, const Matrix& n, Matrix& ut);
```

The permutation is given by $(i \mod S)(N \div S) + i \div S$ (in integer arithmetic) where N is the number of rows and $S \approx \sqrt{N}$ is the stride, i.e. the distance between consecutive loop iterations. This type of access was compared with sequential access, for row- and column-major storage, using the best version of the code (version C compiled with O2+unrolling) running in parallel. Table 4.3 shows the results for the small (L3 cache) and large (RAM) matrix sizes.

Matrix size	Access	Row-major	Col-major
Small	Sequential	**1**	1.1
Small	Scattered	4.1	8.5
Large	Sequential	4.5	4.5
Large	Scattered	12	23

Table 4.3: Effect of access patterns on different storage layouts.

When data is accessed sequentially the transition from row-major to column-major is of little consequence. For small sizes there is a 10% loss of performance because partial vectorization of the axpy operation is no longer possible. However, for large sizes where memory bandwidth is the limiting factor, both storage layouts perform similarly. Switching to scattered accesses causes significant loss of performance, as expected, due to the less efficient use of the available bandwidth and automatic prefetching capabilities. Note that the effect is more pronounced for the column-major layout, this is explained by spacial locality. Recall that memory is accessed by cache lines, each line contains on average 2.7 (64/24) rows of a row-major matrix (with three doubles per row) on the other hand, it only contains one third of each row of a column-major matrix. In this example temporal locality does not play a strong roll since each row is only accessed once.

Software prefetching

The main drawback of scattered accesses on column-major layouts is the increase in cache misses. The programmer can ameliorate this by explicitly instructing the processor to move data into cache. However, this introduces some overhead. By themselves, the prefetching intructions have a latency and throughput similar to addition or multiplication, however, to look ahead in the iteration space, requires an extra index of the permutation to be loaded and corresponding memory addresses to be computed. Moreover, the iteration space may not be defined at compile-time, for example if it is distributed dynamically over multiple threads. Therefore, it is necessary to tune the prefetching distance so that data becomes available as it is needed. This tuning is even more important when temporal locality effects are important, otherwise the prefetched data may actively pollute the cache. The effect of prefetching is shown in table 4.4.

Matrix size	Prefetch	Row-major	Col-major
Small	No	4.1	8.5
Small	Yes	4.0	8.2
Large	No	12	23
Large	Yes	9.4	19

Table 4.4: Effect of prefetching on scattered access pattern.

Note how the speedup is more pronounced for large matrix sizes, this is logical since the small size fits in cache. Although cache misses are more severe with column-major storage, the prefetching overhead is also higher since one instruction per column is needed to prefetch one row, whereas the row-major layout only requires one instruction per row (since they are small in this example). Moreover, there is not enough computation per loop iteration to hide the cost of the extra instructions. This explains why the speedup for large sizes is similar for both layouts.

Vectorization

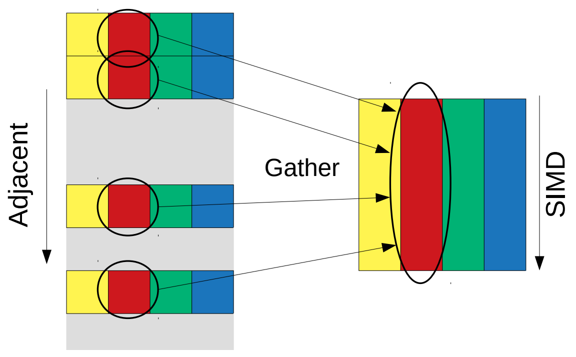
The advantage of column-major storage, is that it is more amenable to vectorization of algorithms that access data sequentially. While some vectorization is possible in the "tangentComponent" example with row-major storage (of the axpy operation), this makes poor use of the hardware, consider that modern CPU are capable of processing 4 to 8 doubles simultaneously (AVX and AVX512 respectively). This is even more relevant for GPU's where work is dispatched for large groups of threads (32 or 64), and allowing all of them to read from adjacent memory locations (so called coalesced memory access) is fundamental to fully utilize the available bandwidth. In cases where the inner dimension does not offer enough "vectorizability", processing multiple outer indices simultaneously may be an alternative. Especially if data is adjacent along the outer dimension, which is a requirement to access it at peak bandwidth.

Note that this is an expensive type of compiler optimization (O3 for most) that effectively involves moving blocks of the outer into the inner loops (e.g. the dot product loops in the "tangentComponent" example). Thus, not all compilers may perform it, especially if the body of the loop is too complex or contains unvectorizable operations (which is generally the case of transcendental math functions, unless some fast-math optimizations are allowed). Moreover, it was seen earlier that this level of optimization may actually introduce overhead, which the gains from vectorization may not offset, especially in bandwidth-limited algorithms. This is shown by the results in table 4.5, where scalar code is compared with implicitly (by the compiler) and explicitly (by the programmer) vectorized versions of the code (explicit vectorization is discussed later). For small sizes the advantage of vectorized code is evident, on the other hand, there is no noticeable benefit for large sizes. However, note that the vectorized version is able to use all the memory bandwidth with a single core.

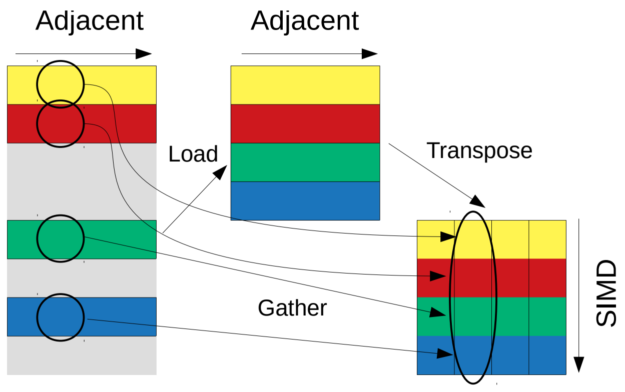
Matrix size	Scalar	Implicit Vec.	Explicit Vec.
Small	1.1	1.1	0.62
Large	4.5	4.6	4.5

Table 4.5: Effect of vectorizing sequential accesses on column major layout.

For scattered access patterns, vectorization requires *gathering* data into SIMD registers and then *scattering* the results. In other words, data needs to be made contiguous before vectorized computations can take place. While there are dedicated hardware instructions for this purpose (*gather/scatter*), their performance and availability depends on the architecture. Namely, scatter is only part of the AVX512 instruction set, and gather is approximately 50% faster on the Intel Skylake and AMD Zen3 architectures than on their respective predecessors. Notwithstanding this major performance leap, *gather* still has 10 times lower throughput than *load* [122]. Fig. 4.2a shows a diagram of this operation. For this type of access pattern, row-major layouts offer an alternative to gather and scatter data without using those instructions. That alternative consists in loading data along the inner dimension (using SIMD loads since data is adjacent) and then transposing it to make it contiguous along the outer indices that are being accessed. This is shown in Fig. 4.2b, in comparison with the native gather operation.



(a) Gather instructions on column-major layout.



(b) Gather or load-transpose on row-major layout.

Figure 4.2: Strategies to gather data for multiple outer indices to then process it with SIMD instructions.

Of course, this transposition adds overhead, however, it is vectorizable via *swizzle* instructions (shuffle, permute, etc.) which have high throughput on most architectures. Moreover, this is a common operation that compilers can identify and optimize well, especially when the number of elements per row is a multiple of the SIMD width. Nevertheless, making the associated loop counts available at compile time is crucial for performance, more so than in the optimization of small scalar loops. For clarity, this strategy of vectorizing the accesses along the inner dimension does not go against the earlier comment that this could make poor use of the hardware. Note that even when the number of elements per row is a multiple of the SIMD width, the operations performed for each element may be different, in which case they would not fit the single-instruction model without the grouping of multiple outer indices (which this operation makes possible).

A final overhead factor, that affects both strategies to fetch data (gather and load-transpose) is the increased probability of cache misses, which is a consequence of accessing data that is far apart. This has made row-major storage the recommended layout for cached machines [123], and may explain why the load-transpose strategy was also found useful on architectures with efficient *gather* instructionst [124]. Regardless of loading strategy, it is important that the amount of computation performed with the data justifies the added cost of accessing it in this manner.

Hybrid layouts

As mentioned in the introduction to this section, no single layout is optimal for all applications. The *array of structures of arrays* layout in Fig. 4.1 attempts to balance spatial locality with the vectorizability of sequential accesses on column-major storage. It was used in [124] for some of the geometric quantities used by an unstructured CFD solver, to reduce the pressure on the hardware prefetchers (i.e. the number of arrays from which data is streamed). While evaluating that layout for this work, it was noticed that the arithmetic needed to convert pairs of indices (i, j) to 1-D memory locations, namely

$$(i \div P)PN + Pj + (i \mod P)$$

where P is the length of the short (inner) arrays and N the number of columns, adds significant overhead to lightweight loops (even when P is a power of two). This happens when the compiler does not move the terms in i to the outside of a loop over j. To help the compiler it was helpful to define an iterator for inner indices, then the programmer can obtain the iterator outside of the loop (evaluating the terms in i) and simply increment it in the loop over j.

Hybrid layouts also find application in sparse storage formats. For example, matrix-vector multiplication of typical compressed sparse formats, either by rows (CSR) or columns (CSC), does not take full advantage of GPU (or CPU with wide vector units) unless the algorithm is modified to access memory in a coalesced manner [125]. Even for block-compressed formats (where each non-zero is a small dense matrix) which occur e.g. when there are multiple degrees of freedom per node, good memory bandwidth utilization is only possible for large block sizes [126]. The list-of-lists format (also known as ELLPACK format) allows coalesced accessed by storing the non-zero coefficients and column indices in column-major matrices. However, this requires the sparse matrix to have either a constant number of non-zeros per row, or for the rows be padded (with zero coefficients). Padding reduces the efficiency of this storage format when the number of non-zeros per row varies significantly, as it happens in dual discretizations of hybrid meshes. Earlier solutions consisted in storing the most common size in ELLPACK format and the remainder coefficients in a compressed format or as triplets [125]. More recently, it has been proposed to divide the matrix into multiple ELLPACK slices [127], thereby creating an array of structures of arrays, where each group of rows (slice) is allowed to have different number of columns. In fact, balancing different aspects of classical layouts (e.g. performance, versatility, storage efficiency) has resulted in the proposal of many hybrid storage formats [128].

4.1.4 Explicit and implicit vectorization

The results for the vectorized version of the "tangentComponent" function, in table 4.5, show that compiler-generated (implicit) vectorization may be a far cry from hand-tuned (explicit) code. Based on inspecting the assembly code generated for a number of examples, ranging from the simple "tangentComponent" to complex parts of SU2, the main shortcoming of implicit vectorization is register spillage, that is, data is moved in and out of memory more times than necessary. Moreover, implicit vectorization is not always automatic, as it was mentioned before, more complex code may require changes to help the compiler. One common change is to separate data accesses from computations, that is, moving data into local variables first, instead of referring to the original memory locations in the computations. Examples of this are found in [129, 124] in the context of unstructured solvers. One consequence of this approach, is that new loops over the SIMD width need to be introduced to annotate which areas the compiler should target for vectorization. Listing 4.7 shows what this type of change could look like for the simple example.

```cpp
void tangentComponent(const Matrix& n, Matrix& ut) {

  constexpr size_t nDim = 3;
  constexpr size_t simdLen = 4;

  for (size_t i0 = 0; i0 < n.rows(); i0 += simdLen) {

    double dot[simdLen] = {0}, n2[simdLen] = {0};

    for (size_t j = 0; j < nDim; ++j) {
        #pragma omp simd
        for (size_t k = 0; k < simdLen; ++k) {
            const auto i = i0 + k;
            dot[k] += n(i,j) * ut(i,j);
            n2[k] += n(i,j) * n(i,j);
        }
    }

    #pragma omp simd
    for (size_t k = 0; k < simdLen; ++k) dot[k] /= n2[k];

    for (size_t j = 0; j < nDim; ++j) {
        #pragma omp simd
        for (size_t k = 0; k < simdLen; ++k)
            ut(i0+k, j) -= dot[k] * n(i0+k, j);
    }
  }
}
```

Listing 4.7: Helping the compiler with vectorization.

On the other hand, explicit vectorization makes use of either inline assembly, or intrinsic types and functions [130] (which are essentially a lightweight wrapper for assembly). The latter are demonstrated in listing 4.8

```cpp
void tangentComponent(const Matrix& n, Matrix& ut) {

  constexpr size_t simdLen = 4;
  for (size_t i = 0; i < n.rows(); i += simdLen) {
    // load
    auto nx = _mm256_load_pd(&n(i,0));
    auto ny = _mm256_load_pd(&n(i,1));
    auto nz = _mm256_load_pd(&n(i,2));
    auto ux = _mm256_load_pd(&ut(i,0));
    auto uy = _mm256_load_pd(&ut(i,1));
    auto uz = _mm256_load_pd(&ut(i,2));

    // compute
    auto nn = _mm256_mul_pd(nx,nx);
    nn = _mm256_fmadd_pd(ny,ny,nn);
    nn = _mm256_fmadd_pd(nz,nz,nn);
    auto un = _mm256_mul_pd(nx,ux);
    un = _mm256_fmadd_pd(ny,uy,un);
    un = _mm256_fmadd_pd(nz,uz,un);
    auto un_nn = _mm256_div_pd(un,nn);
    ux = _mm256_fnmsub_pd(un_nn,nx,ux);
    uy = _mm256_fnmsub_pd(un_nn,ny,uy);
    uz = _mm256_fnmsub_pd(un_nn,nz,uz);

    // store
    _mm256_store_pd(&ut(i,0),ux);
    _mm256_store_pd(&ut(i,1),uy);
    _mm256_store_pd(&ut(i,2),uz);
  }
}
```

Listing 4.8: Explicit vectorization using x86 intrisics.

SIMD types

The two vectorization options described above are not very readable or generic. Moreover, using intrinsics not only ties the algorithm to a particular data layout, it locks it to one datatype and architecture. Notwithstanding, using this type of function allows portable performance, in the sense that it is predictably high across different compilers (old and new). The solution to the generality issues is, of course, encapsulation, namely, by defining SIMD types to wrap the type- and architecture-specific intrinsics with natural syntax, allows existing algorithms (e.g. axpy) to function with minimal changes. Furthermore, moving data into or out of the SIMD type should be implemented by the container (Matrix in the example) such that any assumption regarding the data layout stays within the class responsible for it. It is no coincidence that these considerations are reminiscent of the earlier "Vector" type, after all, a SIMD type is in essence a short vector. Therefore, the starting point for a portable SIMD type is a Vector of known size, e.g.

```
template<class Scalar, size_t Size = preferredSimdLen<Scalar>()>
class Array : public VecExpr<Array<Scalar,Size>, Scalar> {
    alignas(Size*sizeof(Scalar)) Scalar x_[Size];
    ...
```

The constructors of this type should reflect the normal ways to fetch data, e.g.

```
public:
    Array(Scalar x) { broadcast(x); }
    Array(const Scalar* ptr) { load(ptr); }
    template<class T> Array(const Scalar* beg, const T& offset) { gather(beg,offset); }
    ...
```

Math operations are provided by the expression templates, and so the complete implementation of the portable SIMD array has less than 100 lines[3]. Now, the type- and architecture-specific optimizations (via intrisics) are obtained by specialization of this class template. For example

```
template<>
class Array<double,4> {
public:
  // allows accessing individual elements, but requires compiler support as it is not
    strict C++
  union {
    __m256d reg; // this array specialization wraps an intrinsic type
    double x_[4];
  };

  Array(__m256d other) { reg = other; } // allow construction from wrapped type
  ...
```

Note that the specialization no longer uses expression templates, there is no benefit in doing so since each SIMD operation is by definition a single instruction. Instead, operators are overloaded to return arrays directly, for example

```
inline Array<double,4> operator+ (const Array<double,4>& a, const Array<double,4>& b) {
    return _mm256_add_pd(a.reg, b.reg);
}
```

For transcendental operations, for which there are no native instructions, the overload needs to loop over individual elements, e.g.

```
inline Array<double,4> cos (const Array<double,4>& a) {
    Array<double,4> r;
    #pragma omp simd
    for (size_t k=0; k<4; ++k) r[k] = cos(a[k]);
    return r;
}
```

[3]https://github.com/su2code/SU2/blob/v7.1.1/Common/include/parallelization/vectorization.hpp

Then, it is up to the compiler to call specialized libraries that may vectorize this type of operation (via some series expansion). To call such functions directly, it is important to note that different compilers use different libraries, thus one more layer of encapsulation is needed.

Similarly to what is done with expression templates, the specializations for different types and SIMD widths are obtained with code generation. This can be done using only the C preprocessor by defining a generic specialization in a header file (a "template" specialization as it were) where the names of the types, the size, and other properties are represented as macros. That is

```
template<>
class ARRAY_T {
public:
  using Scalar = SCALAR_T;
  using Register = REGISTER_T;
  enum : size_t {Size = sizeof(Register) / sizeof(Scalar)};
  union {
    Register reg;
    Scalar x_[Size];
  };
  ...
```

The same could be done for the implementation of the math operators, however, it is more elegant to define a common interface and then rely on overload resolution. For example

```
inline ARRAY_T operator+ (const ARRAY_T& a, const ARRAY_T& b) {
    return simd_add(a.reg, b.reg);
}
```

where "simd_add" is defined (elsewhere) as

```
inline __m256d simd_add(__m256d a, __m256d b) { return _mm256_add_pd(a,b); }
inline __m512d simd_add(__m512d a, __m512d b) { return _mm512_add_pd(a,b); }
// etc.
```

Specializations are created by defining the symbols and overloads, and then including the header with the generic specialization.

```
#ifdef __AVX__
// Create specialization for array of 4 doubles.
// symbols
#define ARRAY_T Array<double,4>
#define SCALAR_T double
#define REGISTER_T __m256d
// overloads
inline __m256d simd_add(__m256d a, __m256d b) { return _mm256_add_pd(a,b); }
// generate
#include "generator.hpp"
```

```
// undefine symbols
// #undef ...
#endif
```

The code generation mechanism is as concise as the base SIMD type[4] and it is not specific to any architecture. Its only requirement is the concept of a "native" SIMD type, which is available for a number of architectures, such as x86, ARM, and POWER, to name a few.

Listing 4.9 shows the explicitly vectorized version of "tangentComponent" written using the SIMD type. The algorithm is unchanged, hence it is as expressive as before. Nevertheless, it performs as if intrisics were used directly. The only intrusion is in how the loop is advanced and how data is retrieved and stored. Note how the interface for accessing data operates on entire rows of the matrix type, this allows the different modes of accessing data (explained in the previous section) to be selected automatically based on the storage layout.

```
void tangentComponent(const Matrix& n, Matrix& ut) {

  constexpr size_t nDim = 3;
  using Double = Array<double,4>;

  for (size_t i = 0; i < n.rows(); i += Double::Size) {
    auto n_i = n.getRows<std::array<Double,nDim> >(i);
    auto ut_i = ut.getRows<std::array<Double,nDim> >(i);

    auto un_nn = dot(ut_i,n_i) / dot(n_i,n_i);
    axpy(-un_nn, n_i, ut_i);

    ut.setRows(i, ut_i);
  }
}
```

Listing 4.9: Explicit vectorization using a SIMD type.

Branch-less code

For code more complex than the example above, the use of a SIMD type may have a stronger implication. Namely that it is not possible to branch (in the traditional sense) based on comparisons involving SIMD types, for example

```
Double fcn(Double a, Double b) {
    if (a > 0) return a+b;
    else return 2*b-a;
}
```

[4]https://github.com/su2code/SU2/blob/v7.1.1/Common/include/parallelization/special_vectorization.hpp

is not valid code, which poses an implementation challenge for piece-wise functions. The way to vectorize conditional expressions is to evaluate all branches and combine the results via masking operations. This masking can be either logical

```
    auto mask = a > 0;
    return mask & (a+b) | ~mask & (2*b-a);
```

or arithmetic.

```
    auto mask = a > 0;
    return mask * (a+b) + (1-mask) * (2*b-a);
```

In the former the relational operators (>,<,==,etc.) are overloaded to return SIMD types where all bits of each lane are either 1 or 0, masking is then done via bitwise operations. All these operations have SIMD variants, and this is the strategy used by compilers (when they succeed). For arithmetic masking, the results of the vectorized comparisons need to be converted to 0.0 or 1.0. Thus this approach has lower performance and care is needed if any branch may produce a non number (since $0 \times \text{Nan} \neq 0\text{b}\&\text{Nan}$). However, it is compatible with algorithmic differentiation types and natural for some functions. Consider for example the upwind flux of a scalar variable

```
Double flux(Double mdot, Double a, Double b) {
    if (mdot > 0) return mdot*a;
    else return mdot*b;
}
```

which can easily be made branch-less by clamping "mdot" using min and max (which have native SIMD implementations).

```
    return max(mdot,0)*a + min(mdot,0)*b;
```

Although branch-less code is not required by GPU, their SIMT (single-instruction-multiple-thread) execution model also benefits from it. Even scalar code executed on CPU may benefit with the increased predictability of the operations.

4.1.5 Algorithmic complexity

So far, the discussion was centered on techniques and aspects of improving particular algorithms. It is important, however, to first choose an appropriate algorithm (or to question existing choices), lest any lack of performance be due to scalability rather than lower-level performance aspects. This amounts to evaluating the complexity of the algorithm, that is, how the computational effort and storage requirements grow with the size of the problem. It is highly desirable for a simulation code to scale linearly. For that to happen, its components also need to scale linearly, otherwise they may unexpectedly become responsible for large fractions of the total runtime. The unexpectedness of this has two sources, one is the hierarchical nature of some elements of the hardware, the other is the "shape" of the problem.

Consider for example that loop tiling has no effect if the data already fits naturally in cache. Similarly, the performance of an algorithm that uses linear searches will degrade significantly each time the size of the data forces it to reside in slower memory. In fact, it is common for high-complexity algorithms to be faster than low-complexity alternatives on small problem sizes due to locality effects (which may give the programmer the wrong impressions during early testing), but then, of course, perform poorly at scale.

Regarding "shape", in two and three dimensions the number of boundary elements is proportional to $\sqrt{N}$ and $N^{2/3}$ of the total number of elements (N), respectively. These sub linear relations allow some algorithms, that operate on boundaries, to have more than linear complexity without becoming a bottleneck. However, for a very slender 3-D domain (in the limit, a slice) such algorithms may "suddenly" become a bottleneck since the number of boundary elements is directly proportional to the total. Two relevant examples, based on the experience of the author with SU2, are presented next to illustrate how algorithm complexity can guide the implementation of one algorithm and one data structure.

Transposing a sparse matrix

In the context of interpolation methods for fluid-structure-interaction interfaces, described in a previous chapter, the conservative approach involves transposing the interpolation matrix for structural displacements. This matrix is usually stored by rows, that is, for each fluid node, the indices and weights of the structural donor nodes are known. Transposing this information answers which fluid nodes does each structural node contribute to. Solving this by performing multiple linear searches has quadratic complexity in the number of boundary points. A better alternative is to first convert the original matrix into triplets (row,column,value) and then sort them by column (with $N \log N$ complexity), since this allows them to be traversed only once.

An augmented queue

Consider a queue (a structure that allows items to be added to the back of the queue and retrieved from the front, efficiently) with the additional characteristic that it should allow arbitrary items in the middle to be found and removed. Referring to STL types, a vector provides efficient *push back*, however, both searching and erasing elements is linear in the size of the vector. A list allows any item to be modified with constant complexity but searches are again linear (and much more expensive due to locality effects). Finally, maps (i.e. hash tables or binary trees) provide either constant or logarithmic search times, respectively, however the insertion order is not known like in a queue. One way to keep track of this order is to combine a map with a vector, where the former maps the value of the items to their position in the vector. Then, to remove one item from the queue, it should not be erased from the vector (since that has linear complexity), instead, a sentinel value should be left in its place. Once a number of elements has been marked for erasure via this process, the vector is compressed and the map is updated. Making this number proportional to the number of elements gives the operation constant complexity on average (also known as amortised constant time). This is also the strategy used in the STL vector to provide efficient *push back*.

A hybrid solution like this should only provide an interface for the operations it implements efficiently, a queue with "find and erase" in this case, and not the interface of either of the underlying types (it is generally bad practice to make expensive operations convenient to use).

4.2 Algorithms and data structures of unstructured solvers

Before proceeding to apply the general concepts of section 4.1 to unstructured solvers, first, it is helpful to describe their algorithms in terms of general characteristics. Here, what is meant by algorithm is not a specific method, for example, least-squares gradient computation, but rather a more general classification with respect to computational intensity and data accesses. Note how these two factors dictate which implementation aspects (e.g. data layout) and optimizations (e.g. vectorization) are relevant to obtain high performance. Furthermore, they have a major influence on parallelization strategies, as will be discussed in section 4.3.

In unstructured solvers there is no inherent structure (e.g. a grid) that defines how elements (triangles, hexahedra, etc.) are composed, and how nodes (and elements) are connected to each other. Instead, this type of information, which is divided in those two major types, needs to be stored explicitly in some structure that enables algorithms to be implemented.

4.2.1 Composition and adjacency matrices

Composition expresses how certain geometric entities are defined in terms of others, for example, which nodes form an element (that is which node *indices*, not their coordinates) as Fig. 4.3 shows. This is how most mesh formats are defined, thus it is the starting point to compute the geometric properties of the mesh, and to obtain the second type of information.

Adjacency expresses direct connections between the same type of geometric entity, for example, the lists of neighbors of each element or of each node. Clearly both types of information can be stored as sparse matrices. However, due to its symmetry, adjacency can also be represented as a graph, that is, a collection of connected pairs ("edges") of geometric entities. A graph uses approximately the same storage space as a compressed sparse format, but it has a regular structure (e.g. a matrix with 2 columns) which is important for the implementation of some

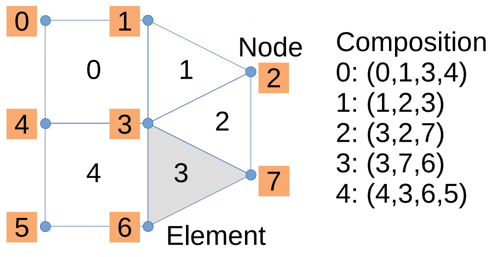


Figure 4.3: Definition of the elements of a mesh.

algorithms. Adjacency is used in the implementation of finite volume methods to integrate the divergence of fluxes for the control volumes. In cell centered codes the edges of the graph correspond to the faces of the elements. Whereas in node-based codes they correspond to the edges of the elements, which in turn constitute the faces of the dual control volumes (see Fig. 4.4). Note that in median-dual discretizations, the dual volumes are obtained by connecting various medians around each edge, however, this is not clear in the figure. For example, in 3-D one "dual face" is defined by the set of triangles obtained by connecting the median of the edge with the medians of the primal faces and elements that surround it. Nevertheless, once the areas of these construction faces are summed, there is only one noticeable difference between cell- and node-based discretizations. Namely that (median) dual faces are located half way between nodes (by construction), which is not the general case for primal faces and volumes.

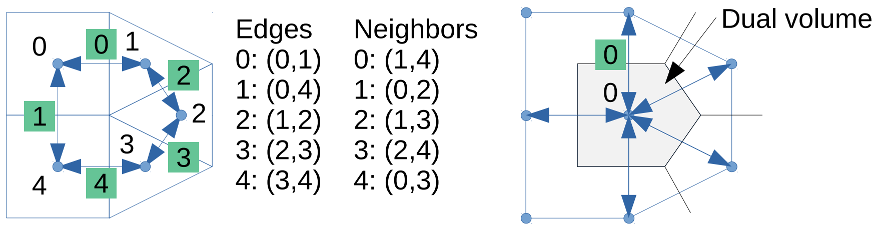


Figure 4.4: Definition of the connections between control volumes.

This type of information is generally not needed for finite element methods. Only the composition of the elements is required when the equations are integrated for each element based on nodal values. Note that adjacency indices do not need to follow any particular order, conversely, to properly define an element requires its nodes to be specified according to some convention.

During the discretization of the equations, these composition and adjacency structures are used to reference geometric and solution variables, which are logically associated with the various geometric entities. For clarity, logical association means that some integer index can be used to refer to both the geometric entity and the associated variables. It does not mean that data is packed into various geometric objects, in an object-oriented way. Using the available bandwidth in the most effective way is crucial for performance, as discussed in the previous section. To that end, an efficient solver implementation is mostly data-oriented. That is, akin to a database whose tables are the various matrices of variables. Then, composition and adjacency matrices are the means by which algorithms can query that data efficiently (i.e. with minimal indirection).

4.2.2 Types of algorithm

Point loops

The simplest algorithms are implemented by looping over solution locations, in any order, without needing to access the data of neighbors. One example is the computation of source terms. In terms of data dependencies, the reference for this type of operation is the "axpy" operation. Therefore, sequential access is possible and easily vectorizable if column-major or hybrid layouts are used to store data. Otherwise, vectorization is only profitable for compute-intensive tasks.

Stencil operations

This type of operation is akin to a point loop where information from direct neighbors is needed. The simplest example of this is a sparse matrix-vector multiplication (SpMv) (assuming the matrix is stored by rows). Since data is only read from neighbors (and not written to them) the outer indices can also be processed in any order. However, the number of inner indices is unlikely to be the same for every outer index. Therefore, vectorizing the processing of multiple outer indices is not trivial, and may require the use of the hybrid layouts described previously, inclusive for the adjacency matrix. However, note that this type of data access is typically found in low compute intensity tasks (truly akin to SpMv), hence that only benefit significantly from vectorization on some architectures. This is because for symmetric or anti-symmetric algorithms (i.e. $f(i,j) = \pm f(j,i)$) looping over edges is two times more computationally efficient since the connection between locations i and j is only visited once (thus f is computed once). However, that may not be the case for bandwidth, consider the computation of gradients by the Green-Gauss method implemented either as a stencil operation or a loop over edges. The latter option requires the gradient at each node to be modified many more times, namely one to initialize it to zero, several to update it with contributions from various edges, and a final to divide the surface integral by the volume. The stencil operation accesses the area of each face two times, nevertheless, it still accesses less data, especially when the number of degrees of freedom per node is large. Of course, temporal locality effects make this analysis less linear, notwithstanding, experience in SU2 shows the stencil version is generally faster.

Edge loops

In the context of finite volume methods, edge loops are associated with expressing a discretized conservation principle. As discussed above, looping over edges is the right choice for compute-intensive calculations whose results affect the two nodes of each edge. The traditional example is the computation of convective and viscous fluxes. By definition edges are defined by two nodes. Therefore, computations can be vectorized over multiple edges without having to rely on hybrid layouts. However, this type of *gather-scatter* data access (shown in Fig. 4.5) introduces a parallelization challenge, which will be addressed in section 4.3.

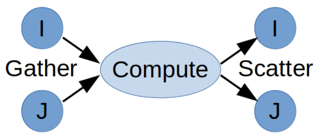


Figure 4.5: Gather-scatter pattern in edge loops.

Element loops

Loops over elements are common in finite element discretizations and are also found in the preprocessing of finite volume meshes. With respect to data access, they are similar to edge loops and thus pose the same parallelization challenge. However, in an unstructured context, not all elements have the same number of nodes. Therefore, to process multiple elements in a vectorized manner requires then to be of the same type. This suggests either an ordering of elements by type, or a processing schedule. Note that although finite element discretizations are computationally intensive, in structural applications they are inexpensive compared with the solution of linear systems (unless these are solved in a matrix-free manner, which is rare). For this reason, it is less important to vectorize such computations, nevertheless, small loop optimizations are still relevant. Here, it is worth noting some key aspects of high-order methods. First, a large number of solution nodes is uniquely associated with each element, hence the corresponding data is accessed sequentially. Second, these methods rely heavily on matrix-matrix multiplication, which can be optimized via BLAS routines, rather than by explicit vectorization. These are the main computational aspects that make flux-reconstruction methods highly suited for future HPC systems [131].

Linear system updates

In implicit solution methods, the solver is responsible for preparing a linear system as part of the algorithms described above. This involves setting the non-zero values of a sparse matrix and computing a right-hand-side vector. For generality the sparse storage format is typically independent of the domain-specific structures. However, it may be convenient to augment it to make the matrix updates more efficient. For example, in finite volume methods each edge can be uniquely associated with two off-diagonal values (i, j and j, i). Doing so avoids having to find the location of those values in the storage layout of the matrix.

4.2.3 Improving temporal locality

Except in the simplest algorithms, which access data sequentially, data indices are accessed multiple times. For example, each node is common to multiple edges and multiple elements. Therefore, due to caching effects, it is important for multiple accesses to the same location to occur as close to each other as possible. This improves performance by the same mechanism as loop tilling, that is, by preventing data from being retrieved from main memory multiple times.

From a graph theory point of view, this is achieved by renumbering indices such that the bandwidth of the graph (and any associated sparse matrix) is minimized. One method that achieves this is the Cuthill–McKee algorithm [132]. Then, the main entities that those indices form (edges or elements) should be sorted such that traversing them in order implies using recently used data. Which can be achieved by sorting by minimum index, using the maximum to break ties [123]. For edges this is equivalent to sorting by first followed by second index [23]. This convention is useful since it encodes the direction of any vector associated with the edge relative to its nodes (i.e. "points" from i to j). It should be noted that to use native *scatter* instructions, the sorting strategy needs to be modified so that indices do not repeat within each SIMD group. Nevertheless, on current architectures the load-transpose (or transpose-store in this case) strategy described before is more efficient [124].

4.3 Parallelization approaches

High performance computing and parallelization are indissociable. Having discussed, in the previous sections, general implementation aspects and algorithms (in an abstract sense) it will now be clearer what different parallelization approaches require of the algorithms, and what implications they may have to their performance. Parallelization approaches vary mostly with respect to the granularity with which work is distributed over multiple *workers* (machines, processors, cores, threads, SIMD lanes, execution units, etc.). The usual way to extract parallelism from an unstructured solver is through domain decomposition. This approach is compatible with distributed memory systems, where individual machines do not have direct access to the memory of others, hence it provides coarse grained parallelism. On shared memory systems, where multiple workers can access the same memory pool, it becomes possible to exploit finer-grained division of work. To some extend, vectorization is an example of this, since multiple data values are processed simultaneously. The heterogeneity of modern high performance computing (HPC) systems (e.g. CPU+GPU) has contributed to the increased interest and development of task-based parallelism. This approach consists in expressing the program as individual tasks with certain data dependencies, which are interpreted, by a runtime scheduler, as a directed acyclic graph (tasks as nodes, dependencies as edges). This scheduler is then responsible for dynamically mapping this graph to the available hardware resources [133]. This approach promises portability and reduced maintenance, as HPC systems progress into the exascale era. However, it requires a rewrite of the code (typically using a domain specific language), and the magnitude of such a task is clearly outside the scope of this work. Furthermore, even the algorithmic differentiation of shared-memory API's, such as OpenMP, is not widely available in open-source AD tools [134].

4.3.1 Domain decomposition

Although the geometric aspect of domain decomposition is easy to visualize, its impact on algorithms can be less evident. In fact this approach does not require the problem to be geometric in any way. Sparse matrix-vector multiplication is a good case study that demonstrates domain decomposition in a more abstract sense. The implications to the algorithms of unstructured solvers should then be evident, since SpMv is akin to stencil operations (or their equivalent edge loops).

General implementation aspects

Consider a sparse matrix with the sparsity pattern shown in Fig. 4.6, distributed over two workers, i.e. each responsible for half of the rows.

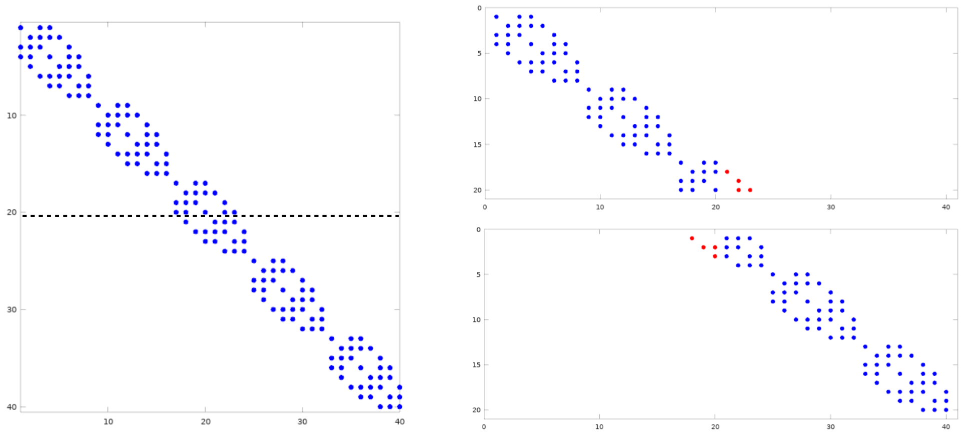


Figure 4.6: Global matrix structure.

Note how to perform a matrix-vector product, some coefficients (colored in red) of the two half matrices need to be multiplied by values of the vector that belong to the other domain. In a distributed-memory context, it is not efficient to communicate those values every time they are needed. Instead, the partitioned vectors of each domain are extended with so-called halo values. Furthermore, it is common for the data structures of each partition to be defined in terms of local indices, and for halos to be implicitly defined by their local index. For example, the tail of the local vectors contains all halos. Then, these sections of the vectors need to be kept consistent across all partitions. In the SpMv example, assuming that values are consistent at the start, only the result of the product needs to be synchronized, or communicated. This involves setting each halo value equal to the value computed by the partition responsible for that global index. These aspects are shown in Fig. 4.7.

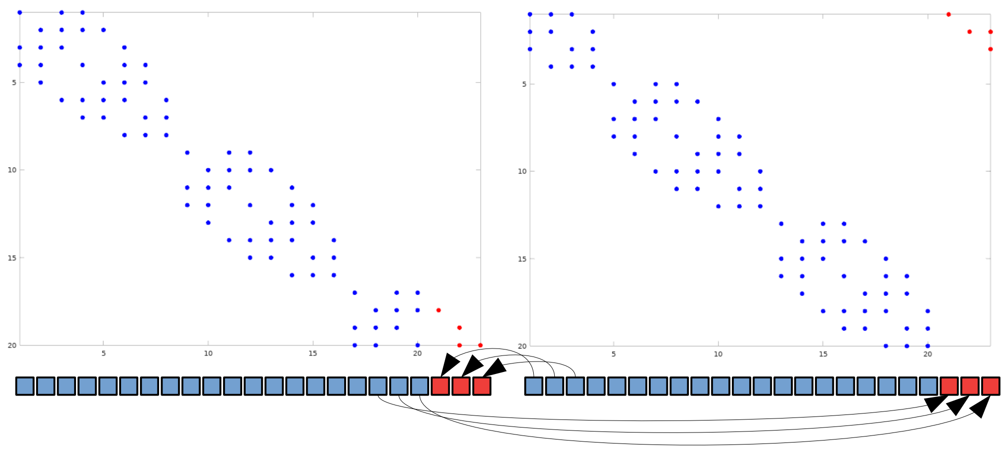


Figure 4.7: Local matrix structures and vectors with halo values.

In a shared-memory setting, communications are simple copy operations, whereas in distributed-memory the responsible partition needs to send messages to partitions that depend on its values. At the same time, receiving partitions need to listen for incoming messages to then update their halo values. Note that axpy-type operations do not require communications since each partition can directly compute its halo values. Communications are typically implemented with the Message Passing Interface (MPI) standard, the details of which are outside the scope of this work. However, it is important to note that MPI defines various communication modes, and the non-blocking type should be used to allow the application to scale to large number of partitions. Furthermore, communications are limited by bandwidth and thus benefit from all optimizations mentioned so far for that type operation.

Challenges

There are, however, efficiency aspects specific to domain decomposition. The main ones are load balancing and minimization of communication costs. Note how a different distribution of the example matrix (splitting it just above or below the middle) would not create halos, formally it would not cut edges of the graph. However, this would result in an unbalanced distribution, where one worker would be assigned more work, thereby becoming a bottleneck. Partitioning algorithms attempt to minimize edge cuts while keeping load balance within some tolerance. This requires a cost model to be defined for the application, to then assign a weight to each partitioned entity. For the SpMv example, the cost model is the number of non-zeros and thus the weight of each row is its number of coefficients. For a finite volume discretization the cost model is some combination of the number of edges and nodes per partition, then, the weight of each node can be defined as the number of connected edges plus a constant. Note that although it is possible to balance both edges and nodes simultaneously, this tends to significantly increase the number of edge cuts. For finite element discretizations the integration order of the elements should also be considered. Regardless of the accuracy of the cost model, some effects are either difficult to predict, or impossible to account for. For example, the cost of boundary conditions, temporal locality effects due to local renumbering, the cost of iterative processes (e.g. for thermodynamic properties or wall models), to name a few. In the end, there is always some unbalance.

Another relevant aspect of domain decomposition, is that in heterogeneous systems the communication costs are not uniform due to the network topology. For example, cores of the same processor can communicate much more efficiently than cores from different machines, even if they are connected by high performance networks. Distributing a graph accounting for heterogeneity in the network (and possibly in the capabilities of the workers) is known as graph mapping (rather than partitioning).

To some extent, the above-mentioned non-blocking communication mode ameliorates both issues. First, by not forcing the order in which messages are received, it allows faster messages to be processed first, thereby hiding the latency of slower ones. And second, by separating the send and receive phases, it allows computations to be performed between them, which hides communication costs, and moreover, may improve load balancing by allowing distinct tasks to execute asynchronously (i.e. without synchronization barriers in between, recalling that it may not be efficient to balance both point-loops and edge-loops). Of course, this requires the code to be designed to take advantage of this MPI capability. Another way to improve load balancing, and account for network heterogeneity, is to use different parallelization strategies within each partition.

4.3.2 Shared data parallelism

Shared data parallelism works at the level of individual algorithms (as described in subsection 4.2.2) by assigning workers to portions of the algorithms. Generally these are some form of loop over entities of the domain, thus a portion is a part of the loop's range (i.e. a chunk, using OpenMP terminology). Recalling that different loop indices require access to the same data, fine grained distribution of loop indices is only possible if different workers have direct access to data (i.e. explicit communication is not necessary).

Benefits and limitations

The fine grained distribution mentioned above, is only efficient if accessing shared data is cheap. For example, although two processors of a two-socket machine have access to the entire memory of the system, however, it is not efficient for both to simultaneously operate on a small region of shared memory due to the cache coherence overhead that this introduces. Similar reasoning applies to cores of the same processor modifying the same cache line, which is known as false sharing. Apart from this particular issue, shared data access is efficient within NUMA nodes (non-uniform-memory-access) due to the uniform access of workers in those groups. Therefore, domain decomposition is also applicable to shared-memory systems to distribute data over NUMA nodes, in a way that restricts shared access to specific steps of the program, i.e. the communications, which in this case do not have to be handled with MPI.

Insofar as data is uniformly shared, the distribution of work is not forced to be static (as in domain decomposition) and thus dynamic load balancing is possible. In the SpMv example, this means that small groups of rows could be assigned dynamically to each worker. A particularly useful way to achieve this is using OpenMP, which provides a portable interface to distribute loops and *tasks* (i.e. individual sections of code) over threads via work-sharing directives. It is worth noting some performance aspects particular to OpenMP. First, note that dynamic scheduling has a cost, and thus it is not appropriate for light-weight loops which have a constant cost per loop index, e.g. "axpy" operations. Furthermore, static scheduling allows a number of that type of operation to be executed without synchronization, since each thread does not depend on the results of others. Second, very fine division (i.e. scheduling) of large loops is inefficient (even if done statically) due to the cache aspects mentioned above. Instead, loops should be divided such that on average, an equal number (1 to 10) of chunks is available for each thread. A final source of overhead is the startup of each parallel section, which also forces a synchronization at the end. It is important, therefore, to amortize this cost over many operations, which are made ready to call in parallel (by means of work-sharing directives), but do not launch threads by themselves (i.e. request them from the global thread pool).

The major challenge, however, of shared data approaches, is dealing with potential data-races that occur when different workers try to modify the same variable, which can happen for example in edge loops, as Fig. 4.8 shows. The strategies of dealing with this issue are discussed next.

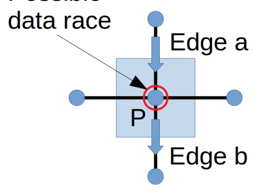


Figure 4.8: Data race in an edge loop, caused by simultaneous access to one point.

Locks **and** ***atomic*** **operations**

Atomic operations and locks are the typical software-based solutions that can be used to guard the access to potentially colliding memory locations, this has the advantage of requiring few changes to the code. However, for frequently used resources, these solutions have limited scalability, moreover atomic operations are only defined for native types. Higher performance solutions are needed for that type of resource. Nevertheless, locks and atomics are particularly useful for reduction operations, e.g. dot products of large vectors, finding the minimum or maximum value, etc. The typical implementation pattern, is for each thread to first work on local variables (within the loop), such that only the final update (outside of the loop) of the shared variable needs to be guarded.

Loop transformations

Note that some algorithms are inherently thread safe, namely point loops and stencil operations, since data is gathered into locations that are only modified by the worker assigned to them. In other words, data is not scattered. Any gather-scatter operation can be converted into a stencil operation, however, this may not be efficient for compute-intensive algorithms (as discussed in subsection 4.2.2).

Reduction

A reduction strategy consists in separating the compute and scatter components of loops, such that the former does not have to be duplicated. For example, edge quantities are stored (without scattering) during the normal edge loop, and then reduced for each node via a separate stencil operation, as Fig. 4.9 illustrates. Alternatively, the target locations (e.g. the residual vector) could be (partially) duplicated, reduction then becomes a simple sum at the expense of some storage overhead.

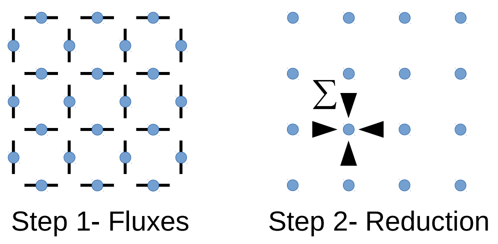


Figure 4.9: Reduction strategy to avoid data races in edge loops.

The reduction approach described above makes the original loop embarrassingly parallel. However, it has the downside of adding a low compute intensity reduction loop. In the context of finite volume discretizations and implicit methods, that loop is made more expensive by the updates to the residual Jacobian matrix (during the computation of edge fluxes). The reduction step then involves setting diagonal blocks of the matrix as the sum of the blocks in the same column, this is an operation with very poor memory access pattern when row-major storage is used for the matrix. This column sum warrants further explanation, the edge flux (F) contributes to the residual of two nodes (i and j), whose variables are used to computed it, therefore the flux Jacobians ($\partial_i F$ and $\partial_j F$) are used to update four locations of the residual Jacobian matrix ($A_{ii} \leftarrow A_{ii} + \partial_i F$, $A_{ij} \leftarrow \partial_j F$, $A_{jj} \leftarrow A_{jj} - \partial_j F$, $A_{ji} \leftarrow -\partial_i F$). Since the pair (i, j) is unique to each edge, it is always safe to write to the off-diagonal blocks. It is for the diagonal entries that race conditions may occur, note however that the data required to compute them is naturally stored in the columns, i.e. $A_{ii} = -\sum_{k \neq i} A_{ki}$, hence no storage overhead is incurred. Reducing the matrix diagonal is less expensive for scalar solvers, where the computation of fluxes is typically manipulated to ensure diagonal dominance, which results in $A_{ii} = -\sum_{k \neq i} A_{ik}$ (i.e. a row sum). This is also true for structural finite element solvers, on the other hand, reducing their residuals has a larger overhead since elements make distinct contributions to the residuals of each node. For this reason, the reduction strategy is not so well suited for this type of discretization.

Grid coloring

Edge coloring consists in identifying non intersecting groups of edges that can be freely processed in parallel, thereby avoiding the overhead of the reduction loop. However, recall that nodes are numbered to reduce the graph bandwidth and edges are sorted to improve temporal locality when traversing them. Therefore, looping over non intersecting edge groups has a negative impact on temporal locality (increased cache misses), so much so that with simple edge coloring the algorithm will perform worse than with the reduction strategy. To preserve some temporal locality it is possible to color groups of edges as depicted in Fig. 4.10. An adequate group size is 500 edges. This grouping introduces some parallel inefficiency due to the reduction in number of assignable work units within each color (a group must be processed by a single worker). A greater challenge in coloring the edges of an unstructured dual grid, is doing so with a reasonable number of colors. Note that the maximum number of intersecting edges can easily exceed 30 and a large number of colors also introduces parallel inefficiency by reducing the number of work units (colors cannot be processed simultaneously). Low parallel efficiency can be especially problematic on coarse multigrid levels due to the reduced mesh size.

A greedy algorithm is used in SU2 to color any entity defined by a set of points (edges, elements, or groupings thereof). The algorithm tries to assign the first available color to each group. A color is not available if any point in the group has already been associated with the color by another group. The interface with which this algorithm works is a sparse pattern.

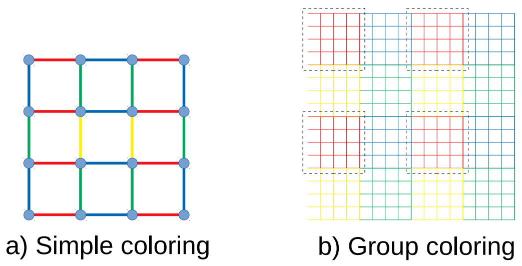


Figure 4.10: Simple (single) and edge group coloring.

Automatic fallback strategies

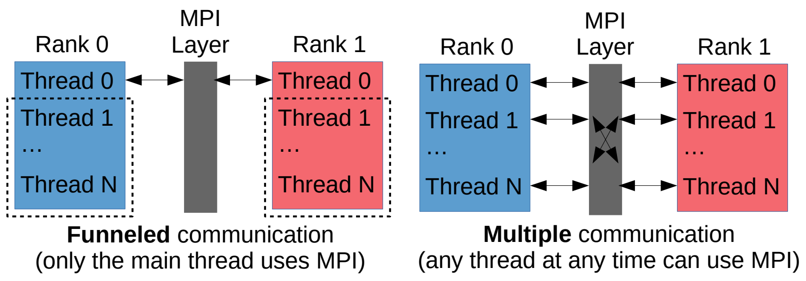
All strategies discussed above to deal with data-races have some downside. Group coloring generally has the best performance and it is relatively non-intrusive, as it involves a simple permutation of loop indices. However, it is possible for cache misses to become too frequent or parallel efficiency too low. Since the conditions for this depend on the specific problem, it is reasonable to combine different strategies. For example, if the coloring of some mesh is detected to be too inefficient, the reduction strategy can be used instead (this is used for edge loops in SU2). Another option is to use coloring while it is efficient, and to use locks (or atomics) for the remainder of the loop, thus avoiding the storage overhead of the reduction strategy. This is used for element loops in SU2, for which this overhead would be significant as mentioned before. Here however, good temporal locality is not as important for performance due to the work-split between discretization and linear solution. For that reason, simple element coloring can be used to maximize the number of assignable work units. In fact, if the solution process requires a direct factorization of the stiffness matrix, shared data parallelism is essential to avoid excessive memory usage (thus rendering the performance of the discretization phase a secondary concern).

4.3.3 Interaction between approaches

Combining different approaches allows the best overall performance. It is important, therefore, to discuss some details of their interaction.

MPI + Threads

The main aspect of a hybrid parallel implementation where multiple threads work within an MPI rank (responsible for a partition of the domain) is whether one or all threads make calls to MPI. In *thread funneled* mode only the main thread can make calls to MPI, in *serialized* mode all threads can use MPI but not simultaneously, which is only allowed in *multiple* mode. This is represented in Fig. 4.11.



Rank 0
MPI Layer
Rank 1
Thread 0
Thread 1
...
Thread N
Funneled communication (only the main thread uses MPI)
Rank 0
MPI Layer
Rank 1
Thread 0
Thread 1
...
Thread N
Multiple communication (any thread at any time can use MPI)

Figure 4.11: Difference between funneled and multiple communication modes.

For some algorithms (e.g. direct sparse solvers) multiple mode is fundamental for performance. On the other hand, unstructured solver algorithms require less often communications, furthermore, when threads are not rigidly assigned to parts of the data, due to dynamic scheduling, a synchronization point needs to be reached by the threads in a rank before communicating the results of an algorithm. For these reasons the funneled mode performs well, so long as messages are prepared and unpacked in parallel.

Threads and vectorization

It was mentioned that vectorization, to some extent, is an example of data parallelism. In fact, the limitation of scatter instructions not being allowed to refer to the same location multiple times, is similar to the data race problem described before, and thus can be solved with grid coloring. Nevertheless, some destructive interference is possible. For example, in edge/element loops the quantities associated with the outer indices are accessed sequentially, which potentially allows them to be *loaded/stored* with SIMD instructions. However, coloring implies a permutation of outer indices, it is important, therefore, to color groups of a size divisible by the SIMD width.

4.3.4 Loop carried dependencies

A common characteristic of the algorithms discussed so far is that different loop iterations are logically independent from each other. That is, they can be performed in any order, and simultaneously so long as race conditions are dealt with. This is representative of discretization loops, but not of any linear preconditioner that involves triangular solves (e.g. ILU, LU-SGS, etc.). Insofar as these preconditioners are an important part of implicit solution methods, a discussion on parallelization approaches is not complete without discussing them. To that end, consider the triangular solve with a lower-triangular matrix, that is

$$\mathbf{L}\mathbf{x} = \mathbf{b}. \tag{4.2}$$

The forward substitution procedure to solve this system is naturally sequential. That is, unless the structure of the matrix allows multiple starting points, the rows of $\mathbf{x}$ need to be processed in order. A typical strategy to parallelize this operation when $\mathbf{L}$ is dense, is to consider it a block matrix, that is

$$\begin{bmatrix} \mathbf{L}_{00} & \mathbf{0} \\ \mathbf{L}_{10} & \mathbf{L}_{11} \end{bmatrix} \begin{pmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_0 \\ \mathbf{b}_1 \end{pmatrix} \tag{4.3}$$

Then, after solving for $\mathbf{x}_0$, a smaller system is obtained for $\mathbf{x}_1$,

$$\mathbf{L}_{11}\mathbf{x}_1 = \mathbf{b}_1^* = \mathbf{b}_1 - \mathbf{L}_{10}\mathbf{x}_0, \tag{4.4}$$

whose right-hand-side can be computed in parallel. The size of $\mathbf{L}_{00}$ is chosen small compared with $\mathbf{L}$ so that computing $\mathbf{x}_0$ sequentially does not become a bottleneck. Therefore, this blocking of $\mathbf{L}$ is repeated many times, which also takes better advantage of the cache (see subsection 4.1.2).

When $\mathbf{L}$ is sparse, updating the right-hand-side in parallel is more challenging. If the matrix is stored by rows, it is necessary to check whether each column index belongs to the desired block (e.g. $\mathbf{L}_{10}$), likewise for the row indices of matrices stored by columns. Note that this information is implicit for dense storage. Row storage has an additional complexity challenge if all rows are traversed (to search for column indices), which would make the operation quadratic rather than linear. While this is not an issue for column storage, computing matrix vector products in parallel is more difficult due to data races. Solving either of these issues requires additional information about the sparsity of the matrix, for example bandwidth and coloring, respectively. Furthermore, note that this parallelization strategy is inefficient in a domain decomposition context, due to the restricted view each worker has of the matrix.

A totally different approach to solve (4.2) in parallel is to use an iterative method. For example Richardson iterations with a parallel preconditioner, such as Jacobi, Gauss-Seidel (with coloring, see Fig. 4.12), approximate sparse inverse [135], or an additive Schwarz decomposition of $\mathbf{L}$. The Schwarz alternative creates multiple starting points for forward substitution by ignoring some off diagonal coefficients. In fact this is the most evident, and common, strategy in domain decomposition contexts. However, it loses efficiency as the number of workers increases, and thus limits scalability when the condition number of $\mathbf{L}$ is high, as more iterations become necessary. Note that iterative approaches are less common when $\mathbf{L}$ is part of a preconditioner for a Krylov linear solver (which is already iterative).

To extend this discussion to factorizations, consider that $\mathbf{LU} = \mathbf{A}$ is a system of equations,

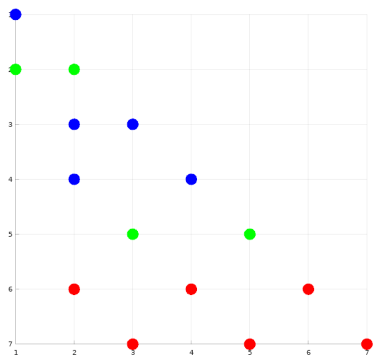


Figure 4.12: Example coloring of the rows of a matrix to allow simultaneous Gauss-Seidel updates within each color.

namely

$$\begin{aligned} l_{ii} &= \mathbf{1} \\ l_{ij} &= \left(a_{ij} - \sum_{k<j} l_{ik}u_{kj}\right) u_{jj}^{-1} \quad && j < i \\ u_{ij} &= a_{ij} - \sum_{k<i} l_{ik}u_{kj} && j \geq i, \end{aligned} \tag{4.5}$$

and thus can be solved in similar fashion to (4.2). Indeed obtaining the l and u coefficients in Gaussian elimination order is akin to forward substitution. To parallelize this process it is possible to block the matrices, with similar challenges when they are sparse. Although less evident, (4.5) can be solved iteratively [136] (note that it is a fixed-point iteration), this allows fine grained parallelism but has some drawbacks. Regarding storage, the algorithm can no longer work in place by overwriting **A**, furthermore a Jacobi solution strategy requires at least a partial copy of the factorization, whereas a Gauss-Seidel strategy requires coloring the coefficients to avoid data races. Regarding computation, each coefficient requires the product of a row with a column, note that accessing a column of a sparse matrix stored by rows is not efficient (and vice versa) at best the location of the coefficients is known but they are not adjacent in memory. Of course, if **A** is symmetric this is no longer an issue. These aspects make the additive decomposition of **A** more efficient when the number of workers is relatively small (which is generally the case in CPU clusters). Moreover this allows systems with the factors (**L** and **U**) to be solved in parallel, since the resulting structure provides multiple starting points.

4.4 Case study, SU2 implementation improvements

The concepts and strategies discussed so far have been applied to SU2 in three stages. First, some kernel areas have been refactored to improve locality and reduce overhead. Then, a hybrid parallel strategy has been implemented to improve the scalability of the code, and to improve the effectiveness of some algorithms (especially geometric multigrid and external direct sparse solvers). Finally, a vectorization library (based on a SIMD type) has been developed and applied to convective and viscous flux schemes. This not only benefited existing methods, but also made Newton-Krylov strategies cost-efficient.

The effect of these optimizations, was first analyzed for a simple subsonic flow problem, namely Mach 0.6 at 2 degrees angle of attack (AoA) over the wing geometry of Fig. 4.13, obtained by *lofting* two NACA0012 profiles. The chord is 0.25 m at the root and 0.175 m at the tip, the span is 1 m. The 0.25c line is swept back 5 degrees and the wing has no twist. Convective fluxes were computed with a second order Roe scheme, the gradients via the Green-Gauss theorem, and the flow variable reconstruction limited with Venkatakrishnan and Wang's limiter. Menter's SST turbulence model was used without wall functions and with first order upwind discretization of convective fluxes. A coarse mesh (approximately 530 000 hexahedrons) was used for the initial investigations as that allows the code to be easily profiled. Moreover, this problem allows a fair comparison of the different stages of implementation improvements, because it was not significantly affected by other changes, algorithmic in nature, that significantly benefited other problems. For example, the static load balancing strategies described in subsection 4.3.1 were implemented, by the author, later than the initial general optimizations. It would thus be unfair to compare the performance of the different stages of optimization for hybrid mesh problems.

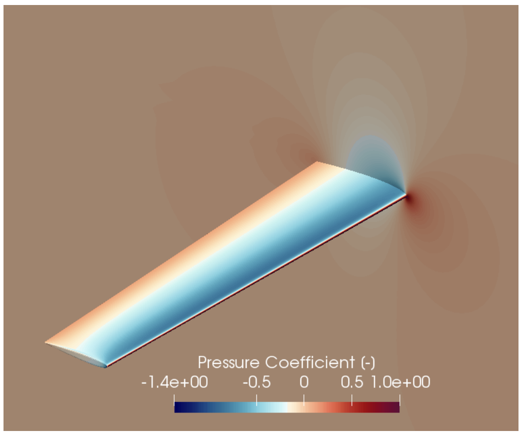


Figure 4.13: Geometry, with pressure coefficient contours, of the simple benchmark problem.

Nevertheless, larger scale problems such as NASA's Common Research Model (CRM, see Fig. 4.14) were used to study particular aspects of the final implementation, and to obtain reference execution times on some architectures[5]. In high-lift configuration the CRM has large regions of separated flow, which make it difficult to obtain converged results with upwind schemes. Instead, the JST scheme was used with second and forth order dissipation coefficients of 0.5 and 0.01, respectively. The Spalart-Allmaras turbulence model was used ("noft2" variant) with first order upwind discretization of convective fluxes, as it is typical for this problem [108]. Finally the "B3-HLCRM" reference grids were used, these are four mixed element grids

[5]A type of information that is rarely provided, but would be rather useful (for example, while choosing what software to use for a project).

(from tetrahedra to hexahedra) with 8, 27, 71, and 208 million nodes, henceforth designated as Coarse, Medium, Fine, and Extra Fine, respectively.

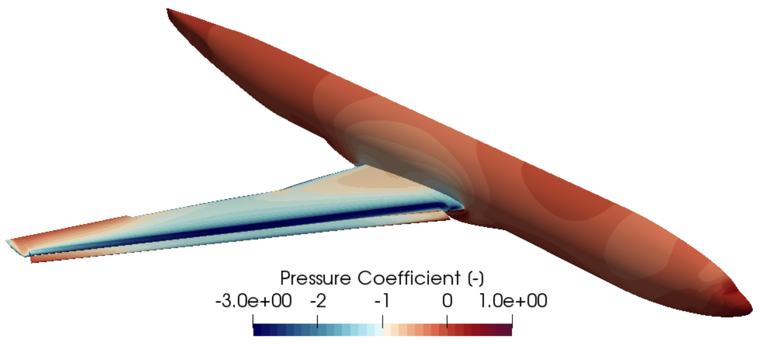


Figure 4.14: Pressure coefficient contours on the high lift CRM at 8 deg AoA, medium grid.

4.4.1 General optimizations

Data layout refactoring

Until version 7, the storage layout in SU2 could be described as *arrays of classes*, i.e. solvers would use arrays of *variables*, and all data associated with a node would be close together in memory. As the code grew, and acquired multiphysics capabilities, this architecture became a performance bottleneck due to the decreasing spatial locality and optimizability that ensues as more data members and virtual functions, respectively, were added to the base class. Either to allow access to particular variables of a new physics problem, or simply to allow common solution variables to be accessed by descriptive names (rather than by "magic number").

The storage layout introduced in version 7 can be described as *class of arrays of structures* (or row-major matrices), i.e. multiple arrays of structures (for example for the conservative and primitive variables) are encapsulated in *Variable* classes. This higher level encapsulation was kept to allow fine tuning of the layout. For example, if it proves beneficial for some method to store the conservative and primitive variables of each node contiguously, fewer changes should be required. The encapsulation also allows some variables to be stored only at particular nodes (e.g. boundaries). Row-major storage is used due to the spatial locality effects mentioned before, but also because the variables of a node (e.g. primitive variables) are passed by pointer to many routines throughout the code. It would thus be difficult to even experiment with a different layout.

The *Variable* classes that encapsulate the storage arrays remain polymorphic, this overhead was reduced in two ways. First, as there is a nearly one-to-one mapping between solvers and variables, the former access their own variables via references (or pointers) to derived class, instead of base, to allow compilers to de-virtualize. Second, the arrays of structures stored by the variable classes are of generic types with a common interface, important routines (e.g. computing gradients) were made generic (templated) and operate on the entire data sets (e.g. on the entire field of primitive variables), thereby making any virtual cost $O(1)$ instead of $O(N)$. Using this generic storage type for solution and geometric variables is also key to implement vectorization, as it centralizes aligned allocation, padding, and high-level vectorized accesses.

Loop fusion

Memory bandwidth is the main hardware bottleneck to low-order finite volume methods (FVM), when considering the typical ratio of compute-to-bandwidth of modern machines, and even before taking advantage of their SIMD capabilities. The single largest block of memory allocated by the solvers is the sparse matrix used to store the residual Jacobian for implicit solution methods. This matrix is populated during edge loops where the flux and its Jacobians are computed (with respect to the solution at the nodes of the edge). In previous versions of the code, two edge loops would be performed for a viscous problem, one to account for the convective fluxes, and another for the diffusive fluxes. This separation (between Euler and Navier-Stokes solvers and numerics) is physically logical but makes poor use of the available bandwidth. To avoid this overhead, the two loops were fused such that the viscous flux for each edge is computed immediately after the convective flux. Later, the vectorized versions of the classes used to compute fluxes, were designed such that both contributions are computed together. Note that this sacrifices the ability to (cleanly) implement hybrid multistage explicit time marching schemes.

Mixed floating point precision

As mentioned above, the sparse matrix needed for implicit solution methods is the largest block of memory allocated by the code. Albeit easy to vectorize, sparse matrix-vector multiplication is a bandwidth-bound operation. Since these linear systems only need to be smoothed, the matrix can be stored in single precision while all other operations in the code are carried in double precision. This improves the FLOP-to-byte ratio of sparse operations without impacting the overall accuracy of the code for problems that are solved iteratively, nevertheless, by default the code compiles for double precision linear algebra. The impact of this optimization is more significant for centered convective schemes and stretched grids, as both increase the condition number of the Jacobian matrix, and consequently the number of linear solver iterations.

Augmented sparse storage format

A block compressed row-major storage format (BCSR) is used for sparse matrices in SU2. In the aforementioned residual loops the matrix is populated by setting these blocks. By mapping off-diagonal entries to edges and diagonal entries to nodes, the location of blocks in 1-D storage space does not have to be searched. For viscous 3-D problems this increases the storage requirements of the matrices by approximately 7%. Storing the location of the diagonal entries is also beneficial when computing and applying linear preconditioners, like ILU or LU-SGS, that operate on the lower/upper parts of the matrix in different phases. Furthermore, operations on the blocks of the matrix are accelerated using Intel MKL's just-in-time code generation for GEMM-type operations (matrix-matrix or matrix-vector multiplication), which to some extent is an example of small loop optimization.

Numerical demonstration on benchmark problem

Total solution times (5 orders of magnitude residual reduction) were obtained for the simple benchmark problem, using a machine with 24 cores (two Intel Xeon E5-2650v4 CPU) with SU2 versions 6.2.0 and 7.0.6. Both versions were compiled with GCC 8.2.0 with optimization flags "-O2 -funroll-loops -ffast-math -march=broadwell -mtune=broadwell"[6], Intel MPI 2019.6 and MKL 2019.4 were used (the latter is not relevant to version 6.2.0). Single core results were not obtained as they are not representative of how CFD codes are used, nor of the compute-to-bandwidth ratio of current mainstream hardware. Table 4.6 shows wall-time, total number of iterations, and average time per iteration, version 7.0.6 is 3.2 times faster per iteration due to the general optimizations.

Version	Wall time	Iterations	Time per iteration
6.2.0	776 s	472	1.64 s
7.0.6	237 s	466	0.509 s

Table 4.6: Comparison of solution times for the simple benchmark, before and after general optimizations.

4.4.2 Hybrid parallel strategy

As mentioned before, the main advantages of hybrid parallel approaches are the improved load balance (static and dynamic), the reduction of communication costs (due to better awareness of the network topology), and finally the increased efficiency of some algorithms. The benchmark problem of Fig. 4.13 is used to demonstrate the strong scalability allowed by the hybrid strategy. Solution times (as previously described) were obtained on a cluster with 24 core nodes (two Intel Xeon E5-2650v4 CPU) connected by an InfiniBand network. Fig. 4.15 shows the observed

[6]The O3 level of optimization reduces the performance of the code, fast-math helps in the vectorization of transcendental operations and dot products, in practice not all optimizations included with it are required. Experience shows that accuracy is not compromised with GCC, moreover some of the fast-math optimizations are used by default by the Intel compilers.

speedup (relative to execution on one node) against the expected speedup (simply the number of nodes).

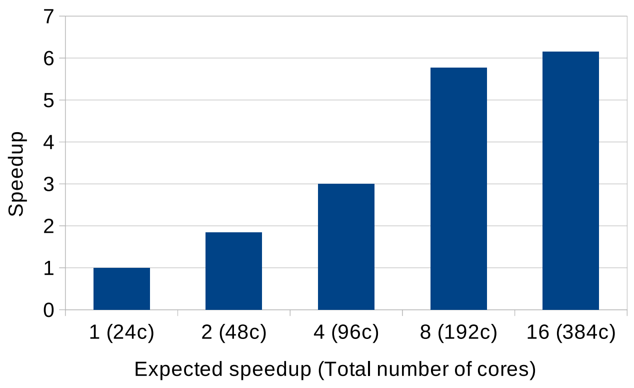


Figure 4.15: Observed speedup vs expected speedup with increasing number of compute nodes.

On the 24 and 48 core runs, 24 MPI ranks were used, 16 were used on the other three. In the transitions from 24 to 48 cores and 96 to 192 cores the relative speedup is approximately 1.85, which is close to the ideal value of 2. However, from 48 to 96 cores the relative speedup is 1.65, this is due to the automatic switch from coloring to the reduction strategy (which for this problem, on this kind of machine, performs noticeably worse). The final doubling of resources produces almost no speedup, note that on the 192 core sample, there are approximately 2750 nodes (16 500 edges) per core. Therefore, communication costs are dominant and the solution time is sensitive even to the position of the nodes in the network topology. Then, the final run with 384 cores also marks the point where one MPI rank spans two NUMA nodes, which has a negative impact on data parallel approaches (see subsection 4.3.2). Finally, note that for this problem the solution time would increase in MPI-only mode with more than 48 cores, due to reduced effectiveness of the geometric multigrid strategy. For this reason, this problem does not permit a fair comparison between the MPI-only and MPI+threads implementations. To that end, the CRM Coarse mesh was used to study the influence of the number of threads per MPI rank on the wall time per iteration, with a fixed number of cores. Specifically 192 cores (8 nodes) and 408 cores (17 nodes) as Fig. 4.16 shows.

With few threads per rank the domain decomposition overhead is highest (static load imbalance, communication costs, etc.), whereas with few MPI ranks per compute node, i.e. many threads, the shared memory overhead increases (coloring vs reduction, cache invalidation by *false sharing*, etc.). For this system and problem, the solution time is minimized with one MPI rank per socket (also a NUMA node in this system), this is expected anytime a significant number of compute nodes is used, especially with lower performance networks. Recent x86 CPU architectures have up to 64 cores per NUMA node, in which case 2 to 4 MPI ranks per NUMA node could be optimal, especially if that avoids falling back to the reduction strategy. As observed with the benchmark problem, there is a large penalty for covering 2 NUMA nodes with 1 MPI rank (24 threads in this case) therefore, the number of ranks should always be a

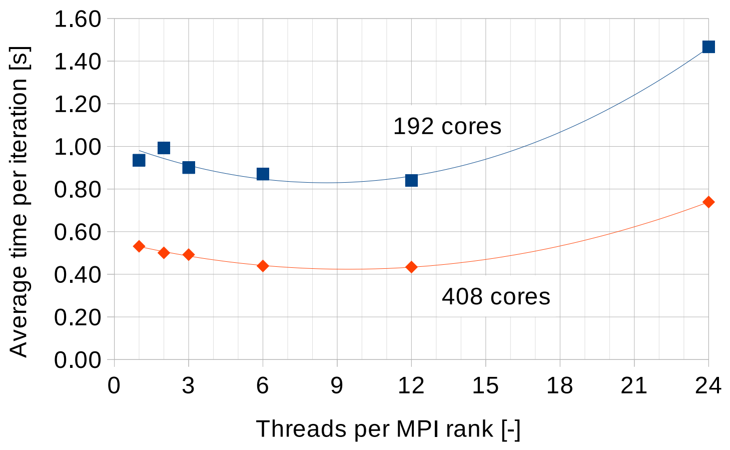


Figure 4.16: Average time per iteration against number of threads per rank (2-socket 24 core nodes), CRM Coarse mesh.

multiple of the number of NUMA nodes. From 192 to 408 cores at 12 threads per rank the parallel efficiency is 91.2%. At those core counts, the hybrid parallelization allows a speedup of 1.11 and 1.22 respectively, compared to plain MPI. This correlation between number of nodes and speedup is expected, and justified by the heterogeneity of communication times within nodes or over the network. However, within a single machine, even if multi-socket, plain MPI is expected to perform better insofar as the algorithms used remain effective with the number of partitions used. With 408 cores the solution time improves by using 2 threads rather than 1, whereas with 192 cores it worsens initially and only improves with 3 threads, this is explained by three factors. First, with 192 cores the communication cost is lower as all nodes are on the same network level, i.e. connected to the same switch; Second, load imbalance issues are less important as MPI domains are still relatively large; Third, and finally, for this problem the cost of switching to the hybrid approach is significant as the reduction strategy must be used immediately (2 threads) by the majority of MPI domains, due to the difficulty in coloring the edges of this mixed-element mesh.

4.4.3 Vectorization

The vectorization strategy is explicit and based on processing multiple edges simultaneously using a SIMD type as described in section 4.1.4. For this reason, the implementation of the convective and viscous schemes is as expressive as before, as listing 4.10 shows.

```
template<class PrimVarType, class ConsVarType, int nDim>
FORCEINLINE VectorDbl<nDim+2> inviscidProjFlux(const PrimVarType& V,
                                               const ConsVarType& U,
                                               const VectorDbl<nDim>& normal) {
  Double mdot = dot(U.momentum(), normal); // "Double" is the SIMD type
  VectorDbl<nDim+2> flux; // A static-size array of "Doubles"
  flux(0) = mdot;
  for (int iDim = 0; iDim < nDim; ++iDim) {
    flux(iDim+1) = mdot*V.velocity(iDim) + normal(iDim)*V.pressure();
  }
  flux(nDim+1) = mdot*V.enthalpy();
  return flux;
}
```

Listing 4.10: Snippet of SU2 code illustrating the vectorization approach.

Two key aspects must be considered to extract the full potential of a SIMD type. First, the number of variables is typically small (< 10), thus it is crucial for data sizes and loop counts to be known at compilation thereby avoiding dynamic allocation and allowing perfect unrolling of loops. Second, data accesses are more expensive due to the load-transpose strategy and the higher probability of cache misses (see subsection 4.1.3) thus they must be amortized.

Accounting for these aspects requires dedicated (i.e. specialized) versions of each spatial discretization scheme for 2-D and 3-D, with or without viscous fluxes, which results in 4 combinations in total. The 2-D/3-D versions allow known sizes, whereas the viscid/inviscid versions permit directly re-using (due to *inlining*) some of the data gathered for the convective flux in the viscous routines, and more importantly, scattering the results only once to the residual vector and sparse matrix.

Class and function templates are used to create the aforementioned versions of each scheme, polymorphism is used to insulate the templates, thereby stopping their propagation to the client code of the numerical schemes, i.e. the *solver* classes. That is, all numerical scheme class templates inherit from a conventional abstract base class. Then, the logic to select (i.e. instantiate) the desired numerical scheme is centralized in a factory method, which also serves as the template instantiation point for all 4 combinations mentioned above.

To simplify the interaction with the hybrid parallel implementation, these numerical classes were designed to be thread safe. This means their state variables must be read-only, which precludes passing data to class methods by "member variable" (which in any case is an anti-pattern), therefore the implementation has a *functional* character. That is, code that can be re-used across different families of numerical schemes is implemented in free functions. Similarly, within families of numerical schemes, for which it makes sense to define a class hierarchy (e.g. centered schemes), any intermediate results must still be passed explicitly when calling class methods. At the expense of longer function signatures, a new developer can clearly see what are the inputs and outputs of any function or method. This also forces all intermediate computation results to be local, stack-allocated variables, which is a performance consideration as it allows them to be truly temporary. Moreover it avoids any aliasing considerations associated with heap-allocated variables (note that a member variable of a heap allocated object is also on the heap).

The implementation components of each numerical method must be inline to minimize register spillage when calling functions, thus design patterns that rely on conventional polymorphism cannot be used. For example, adding viscous fluxes to a convective method could be done via the *decorator* pattern, whereby functionality is added to an existing object by attaching a decorator object to it. To avoid polymorphism, a static variant of this pattern is used where the decorator class (the viscous fluxes) is set at compilation via a template parameter. By making this class the parent of the decorated class, any number of decorators can be chained together (at compile time). Similarly, within a family of schemes, where it would be intuitive to implement common parts in a parent class, and only small details in the final derived classes, static polymorphism is used via the curiously recurring template pattern (see subsection 4.1.2). The pseudo code of listing 4.11 shows the basic structure of these mechanisms.

```cpp
// The abstract base.
struct NumericsSIMD {
  virtual void ComputeFlux(...) const = 0;
}

// One of the viscous flux decorators.
template<int nDim>
class CompressibleViscFlux : public NumericsSIMD {
protected:
  void ViscousTerms(...) const {...}
}

// A common parent for centered schemes, with Derived and Base classes as template
    parameters.
template<class Derived, class Base>
class CenteredBase : public Base {
public:
  void ComputeFlux(...) const final {
    ... // gather data, do some computation
    Derived::DissipationTerms(...); // static polymorphism
    Base::ViscousTerms(...); // static decorator
    ... // scatter results
  }
}

// A final centered scheme, the viscous decorator is forwarded to the parent class.
template<class Decorator>
class JSTScheme : public CenteredBase<JSTScheme<Decorator>, Decorator> {
  static void DissipationTerms(...) {...}
}

// Template instantiation in factory method, e.g. 3-D with viscous fluxes.
NumericsSIMD* Factory(...) {
  return new CJSTScheme<CompressibleViscFlux<3> >(...);
}
```

Listing 4.11: Pseudo code of the template mechanisms used to efficiently compose the vectorized numerical classes.

Evidently this implementation required a complete re-write of the numerical schemes (not just replacing a type). However, note that despite the *template metaprogramming* mechanisms, the guiding principle of the implementation is removing indirection, not metaprogramming by itself, which is only the means to achieve that principle. Notwithstanding, this is certainly the tallest entry barrier in the implementation, as it is seen by many as a paradigm shift from *declarative programming*. It is, however, inextricable from modern high performance C++, and there is an ample supply of conference talks[7] covering it in enough depth to acquaint new developers with its idioms.

Interaction with algorithmic differentiation

Discrete adjoints (based on algorithmic differentiation, AD) are a central feature of SU2 [33] enabling complex multidisciplinary optimizations [101, 118] as discussed in other chapters. A typical reverse mode AD type contains a floating point primal value, and an integer adjoint index. Therefore, in an array of such types the floating point values are not contiguous and thus efficient vectorization is not possible. Nevertheless, when compiling the AD version of SU2, "SIMD arrays" of the AD type are still used, which allows reducing the memory footprint of discrete adjoint solvers. This is due to the use of *preaccumulation* and due to the repetition of node indices within SIMD edge groups (that results from ordering the edges) which reduces the size of the preaccumulated Jacobians. For the simple benchmark problem there is a reduction of 29% in the memory used by the AD tape. Therefore, although the AD version is not vectorized, its runtime is still improved as the smaller memory footprint makes the tape faster to evaluate.

Preaccumulation is one of the less *automatic* features of AD, in the sense that it requires the programmer to explicitly inform the AD tool of which variables are inputs or outputs of the preaccumulated section of code. For this reason, this essential feature of AD is a common source of error for new developers. In this implementation, an additional layer of encapsulation is used to automate the registration of preaccumulation inputs, as shown in listing 4.12.

```
#ifndef SU2_AD // without AD
template<size_t nVar, class Container>
FORCEINLINE VectorDbl<nVar> gatherVariables(Int iPoint, const Container& vars) {
  return vars.template get<VectorDbl<nVar> >(iPoint);
}
#else // with AD
template<size_t nVar, class Container>
FORCEINLINE VectorDbl<nVar> gatherVariables(Int iPoint, const Container& vars) {
```

[7]For example https://cppcon.org/

```
  VectorDbl<nVar> x;
  for (size_t i=0; i<nVar; ++i)
    for (size_t k=0; k<Double::Size; ++k) {
      AD::SetPreaccIn(vars(iPoint[k],i));
      x[i][k] = vars(iPoint[k],i);
    }
  return x;
}
#endif
```

Listing 4.12: Encapsulation of data access to automate aspects of AD preaccumulation.

This encapsulation also allows hiding some unusual template syntax, namely

```
auto x = vars.template get<VectorDbl<nVar> >(iPoint);
```

simplifies to

```
auto x = gatherVariables<nVar>(iPoint,vars);
```

and eventually it may be used to automate prefetching, which was not explored so far. For example, memory addresses and sizes could be pushed onto a LIFO queue (i.e. stack) and then popped and prefetched in between calculations. Note that prefetching instructions should be blended with computations to hide their cost [137] instead of being issued close to *load* instructions.

Numerical demonstration on benchmark problem

The simple benchmark problem is used again to show the effect of vectorization on solution times. Table 4.7 shows the results for the 24 core machine (Broadwell architecture, which supports AVX2 instructions). Table 4.8 shows the results on a machine with two Intel Xeon Gold 6248 CPU and therefore support for AVX512 instructions. For this machine the code was compiled with the Intel compiler version 2019.5 and Intel MPI 2019.6, with the optimization flags "-O2 -funroll-loops -qopt-zmm-usage=high -xCOMMON-AVX512", and run in hybrid parallel mode with 10 MPI ranks of 4 threads.

Version	Wall time	Iterations	Time per iteration
7.0.6	237 s	466	0.509 s
7.0.7 (vectorized)	160 s	450	0.354 s

Table 4.7: Effect of vectorization on the simple benchmark problem, 24 core machine with AVX2.

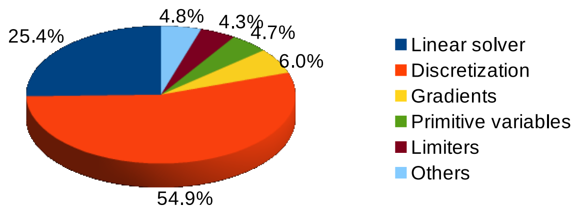
On the 24 core machine the speedup is 1.44, whereas on the 40 core machine the speedup is larger at 1.71, which is expected due to the increased SIMD length (8 vs 4 doubles) and number

Version	Wall time	Iterations	Time per iteration
7.0.6	171 s	490	0.348 s
7.0.7 (vectorized)	97 s	478	0.203 s

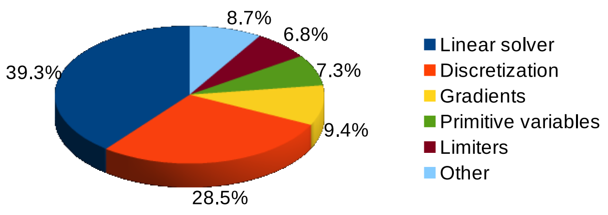
Table 4.8: Effect of vectorization on the simple benchmark problem, 40 core machine with AVX512.

of AVX registers (32 vs 16) which reduces spillage. Note how the use of vectorization on the older architecture is equivalent to a generational improvement in hardware.

It is worth noting that the dedicated 2-D/3-D, viscid/inviscid versions of the schemes also contribute to improved performance, and that speedups of 4 and 8 (for AVX2 and AVX512 respectively) were not expected. The main reason for this is that only part of the code was vectorized, figures 4.17a and 4.17b show how computational time is distributed across the main tasks of the flow solver (i.e. excluding turbulence which accounts for less than 20% of the total) for the scalar and vectorized versions, respectively.



(a) Scalar implementation.



(b) Vectorized (AVX) implementation.

Figure 4.17: Percentage of time spent in different tasks of the flow solver.

This time distribution was obtained by profiling the code (using Perf [8]) running on similar compute-to-bandwidth conditions to those of the 24 core machine but without MPI. Vectorization takes the discretization task (computing fluxes) from using over 55% of the time to only 28.5%, making the linear solver the most costly operation in the vectorized implementation even in a problem that requires only three linear iterations per nonlinear iteration. Note that the high-lift CRM case requires ten linear iterations, which does raise the question of whether the additional code complexity is justified (the answer is yes, depending on the solution method, as will be discussed shortly).

4.4.4 Solution times on a large scale problem

Returning to the high-lift CRM problem, table 4.9 shows the lift and drag coefficients obtained for the four refinement levels, which agree well with published results for other well known CFD codes [108].

[8]https://perf.wiki.kernel.org/index.php/Main_Page

B3-HLCRM Grid	C_L [-]	C_D [-]
Extra Fine	1.74265	0.169493
Fine	1.73726	0.169452
Medium	1.73315	0.169939
Coarse	1.72186	0.170174

Table 4.9: Lift and drag coefficients of the HLCRM at 8 deg AoA.

On 10 of the 40-core AVX512 nodes previously mentioned, the Fine grid converges (to residuals of 10^{-7}) from free-stream conditions after 23 000 iterations in 28.5 h. After 15 000 iterations, and 5 orders of magnitude reduction of the continuity residual, the coefficients settle to within 4 significant digits. This behavior is similar on the other grids, with coarser ones requiring fewer iterations. For example 14 000 for convergence on the Coarse grid, which on 17 of the 24-core Broadwell nodes take 1.6 h. The Extra Fine grid converges in 46 000 iterations and meets the 4 digit criterion in 24 500, on 96 ARCHER nodes (24-core Haswell architecture) this took 31.4 h and 16.7 h respectively. The number of iterations is comparable to what has been published for other codes on similar problems [138, 139]. However, it is clear that for large meshes (>100 million nodes) the simple quasi-Newton solution method scales poorly with problem size, note that the number of iterations doubles from the Fine to Extra Fine grids. This shortcoming is due to the current multigrid strategy, which aside its loss of performance with increased number of MPI ranks, does not cope well with highly stretched boundary layer cells. In other codes this has been attributed to isotropic coarsening of those regions [138].

4.4.5 Algorithmic improvements allowed by better performance

One of the main downsides of the quasi-Newton solution method, is that the typical simplifications of the residual Jacobian matrix, e.g. first order fluxes, thin shear layer, etc. often limit the maximum CFL number and thus the convergence rate. However, simplifications are necessary due to the residual depending on neighbors-of-neighbors, which makes the exact Jacobian difficult to obtain and expensive to store (as discussed in chapter 3). Conversely, Newton-Krylov solution methods, where Jacobian-vector products are obtained via matrix-free approaches (e.g. finite differences), allow much higher CFL since they do not simplify the Jacobian. Of course, preconditioning the resulting linear system is more difficult due to the higher condition number of the exact Jacobian. Nevertheless, this type of method is increasing in popularity [140, 141]. The ability to compute the residual of the discretization quickly is paramount for Newton-Krylov implementations, not only due to matrix-free products, but also to allow the use of backtracking strategies to improve robustness.

In a preliminary implementation in SU2, a Newton-Krylov approach based on finite-differences can be used for the flow equations, while turbulence remains decoupled. The preconditioner for the Krylov solver is based on the approximate Jacobian matrix that would be used in the quasi-Newton method. It is also possible to use the inverse of this matrix as the preconditioner by approximately solving nested linear systems. These are equivalent to the quasi-Newton approach, and can be used as the startup strategy to avoid the breakdown of the Newton-Krylov approach while the residuals are high. This strategy allows higher CFL number to be used for the high-lift CRM problem, namely 100 instead of 25. Fig. 4.18 shows the convergence, residual vs wall-time, for the Coarse grid running on 8 ARCHER2 nodes (1024 total cores).

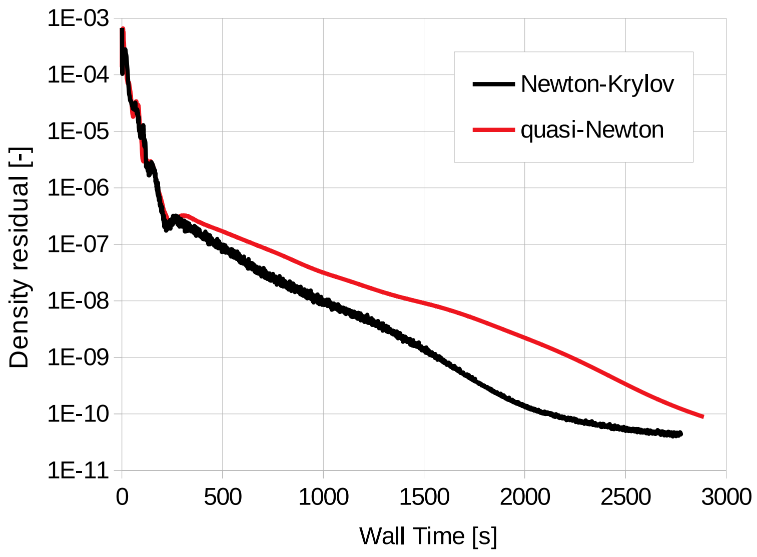


Figure 4.18: Comparison of convergence and solution time between quasi-Newton and Newton-Krylov approaches for the high-lift CRM Coarse grid.

4.4.6 Summary of improvements

Through a number of software implementation optimizations, an average 4-fold performance increase of SU2 has been achieved. These optimizations all build upon each other, for example, it would not have been so advantageous to vectorize the flux computations while keeping the linear system in double precision. Similarly, without the data layout refactoring, the preprocessing-type operations (computing gradients, limiters, and primitive variables) would be two times slower and it would be nearly impossible to implement vectorization cleanly and efficiently. Furthermore, with algorithmic improvements such as better load balancing and Newton-Krylov methods, the compound effect for some relevant problems is 10-fold better performance, with additional benefits to the stability of the solver (which if fundamental for numerical optimization). Finally, the implementation improvements were achieved via highly re-usable and generic code patterns (container types, SIMD libraries, etc.) which will greatly facilitate the optimization of other numerical schemes, or other areas of SU2. Importantly, this will also allow the code to adapt to new hardware capabilities.

Chapter 5

Shape and Topology Optimization with Fluid-Structure Interaction

This chapter presents and discusses two numerical examples that resulted from one of the main research goals of this thesis. Namely, to apply topology optimization to highly flexible aerodynamic structures, in a high-fidelity modelling setting, using the internal material layout to influence the aerostructural response in a significant way. The first example is a compliant airfoil designed for passive load alleviation, and it is the problem that motivated the two-step strategy for discrete topologies described section 2.2.4. The second example is a flexible wing where, by optimizing its internal topology, the same aerodynamic performance can be maintained by an overall more flexible, hence lighter, structure. This 3-D problem, albeit relatively modest in size (4.1 M nodes), would not have been possible to solve in SU2 without the improvements detailed in chapters 3 and 4.

5.1 A compliant airfoil

This example consists of a flexible airfoil operating at two distinct free-stream Mach numbers, 0.25 and 0.5, but at the same angle of attack (AoA) of 2 degrees with respect to the undeformed shape. Henceforth quantities at the lower or higher speed will be super-scripted l or h, respectively. At low speed $C_l^l = 0.5$ must be generated and the deformation of the airfoil kept below a limit, drag should be minimized at the higher speed.

The ideal configuration for minimum drag is a symmetric airfoil operating at zero AoA. Since the airfoil deforms passively and the AoA is fixed, this configuration could only be achieved by a structure with no stiffness, which would then be unable to produce the required lift at low speed. Therefore, constraining the deformation at low speed leads to higher drag at high speed, unless the internal structure of the airfoil responds in a nonlinear way. Here, the objective is to exploit this nonlinear structural behavior to improve passive load alleviation. To that end a baseline design was first obtained, and then the two-step process of chapter 2 was used to design the internal topology. To recapitulate, that process consists in first conducting a topology optimization without any explicit or implicit measure to encourage discrete results. Then, in the second step, a discrete topology is sought by solving an inverse problem, in particular minimizing mass for a target deformation under fixed loads (extracted from the coupled FSI simulation). Later in this section, this strategy is compared with conventional approaches of manipulating the formulation of the topology problem into converging to discrete results (of course, those were tried first, their failure resulted in the proposal of the two-step approach).

5.1.1 Baseline airfoil from aerostructural shape optimization

A baseline compliant airfoil, for which the low speed deformation limit significantly affects performance, was first designed by shape optimization with different allowed values of trailing edge displacement ($y_{\max}$). The starting point for the optimization is a NACA0012 profile with 0.5 m chord, parameterized through the free-form-deformation box (FFD) shown in Fig. 5.1 of which 17 points are allowed to move in the vertical direction, the bottom left point is fixed to avoid translation of the airfoil. A further constraint is added to enforce that the final area be greater or equal than the initial, that is

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & C_d^h(\boldsymbol{\alpha}) \\ \text{subject to}: \quad & C_l^l(\boldsymbol{\alpha}) = 0.5 \\ & y^l(\boldsymbol{\alpha}) \leq y_{\max} \\ & A(\boldsymbol{\alpha}) \geq A_0 \end{aligned} \tag{5.1}$$

where $\boldsymbol{\alpha}$ are the vertical displacements of the control points. For the RANS simulations, the fluid grid is an O-Grid with 77 924 nodes and sufficient wall cell size for $y^+ \approx 1$, the radius of the circular farfield boundary is 30 chords. The sizing of the grid was based on the NACA0012 reference grids used in [48]. The fluid is considered to be ideal and standard-sea-level properties are used for the free-stream state. The solid domain is discretized with 76 800 4 node quadrilateral elements resulting in 77 875 nodes (this level of refinement is to ensure sufficient resolution for topology optimization), the inside of the hollow region close to the leading edge is clamped, the vertical section of this region is located at 5% chord. The elasticity modulus considered is 50 MPa and the Poisson ratio 0.35. Problem (5.1) was solved with algorithm 1 of chapter 2, with the following parameters $a_1^0 = b_1^0 = b_2^0 = 8$ and $r = \sqrt{2}$, the constraint tolerance used was 0.01. The convergence criteria for the optimizer (L-BFGS-B) are as used for the benchmark topology problem of section 2.2.3. Table 5.1 shows the optimized drag and lift for different values of the deformation limit, Figures 5.2 and 5.3 show the undeformed and deformed at Mach 0.5 (in red) airfoils for the 10 mm (0.02c) and 6 mm (0.012c) deformation limits respectively.

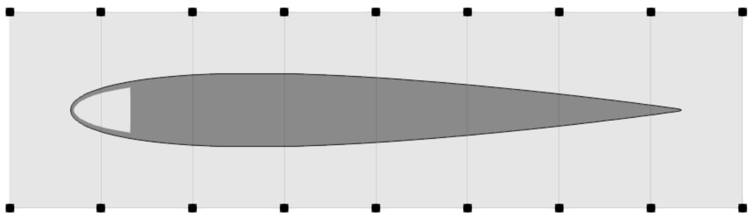


Figure 5.1: Shape optimization parameterization.

$y_{\max}$ [mm]	C_d^h	C_l^h	y^h [mm]	C_d^l
10.0	0.008771	0.251	25.8	0.0106
8.0	0.008909	0.302	22.6	0.0106
6.0	0.009097	0.360	18.3	0.0107

Table 5.1: Shape optimization parametric study results ($()^l$: Ma 0.25, $()^h$: Ma 0.5).

The optimum drag increases 3.7% by reducing the deformation limit from 10 mm to 6 mm. Note that the deformation constraint is not met entirely by an increase in structural stiffness (which could be achieved by increasing area) but mostly by the reduction in pitching moment that results from the reflexed camber line (compare Figures 5.2 and 5.3). This 3.7% increase in drag is a significant-enough trade-off between stiffness and performance, therefore the airfoil optimized for 6 mm is used as the baseline to which the two-step topology optimization strategy is applied.

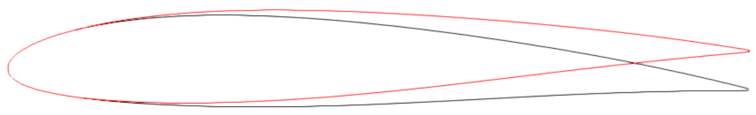


Figure 5.2: Baseline shape optimization results for 10 mm constraint, undeformed (black) and deformed at Mach 0.5 (red) configurations.

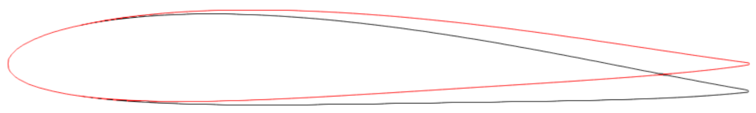


Figure 5.3: Baseline shape optimization results for 6 mm constraint, undeformed (black) and deformed at Mach 0.5 (red) configurations.

5.1.2 Fully coupled topology optimization step

For topology optimization the trailing edge displacement constraint ($y_{\max}$) is replaced by a compliance constraint with upper bound equal to the compliance of the baseline. This change is necessary since, due to the trailing edge region not being forced to be solid, it could be possible for a much more flexible airfoil to respect the local constraint by employing large amounts of camber near the trailing edge in the deformed configuration, and thus producing the required lift at the expense of increased drag. With the area constraint no longer required, the *fully coupled* step is stated as

$$\begin{aligned} \min_{\boldsymbol{\rho}} \quad & C_d^h(\boldsymbol{\rho}) \\ \text{subject to :} \quad & C_l^l(\boldsymbol{\rho}) = 0.5 \\ & W^l(\boldsymbol{\rho}) \leq W^l_{ref} \end{aligned} \tag{5.2}$$

where W^l_{ref} is the compliance ($W = \int_\Gamma \mathbf{u} \cdot \boldsymbol{\lambda} \, \mathrm{d}\Gamma$) of the shape optimized structure at Mach 0.25 (1.141 J), and $\boldsymbol{\rho}$ the design densities (before filtering) introduced in section 2.2. The elasticity modulus is 100 MPa, double the value used in the baseline shape optimization to allow material to be removed while maintaining compliance. The outermost 3 layers of elements (3.6% of the local thickness) are prescribed to be solid, elsewhere the initial density was 0.8, the density filter radius is 2 mm (approximately 3 times the largest element size), a linear weight function was used for the filter, that is

$$w(\mathbf{x}) = \max(0, r - ||\mathbf{x} - \mathbf{x}_0||)/r, \tag{5.3}$$

where r is the filter radius, and $\mathbf{x}$ the coordinate of a point within that distance from the target $\mathbf{x}_0$. For this case the SIMP exponent was gradually increased, taking the values 2, 2.5, and 3 (using the strategy of algorithm 1 of chapter 2). Fig. 5.4 shows the history (inner iterations of L-BFGS-B) of drag and lift coefficients, compliance, and penalized objective function ($\hat{f}$), the large spikes correspond to the increases of SIMP exponent. Fig. 5.5 shows the final topology and Mach number contours at the Mach 0.5 condition.

As expected the material distribution is not discrete, since the objectives and constraints do not benefit from optimally stiff material (obtained for density of zero or one). Until around 30% chord the topology is mostly solid, around the center section the topology confers little bending stiffness acting mostly as support for the wet surfaces. Notably, the trailing edge region has compliant hinges which allow it to work as a mechanism, the importance of this will be explained below.

The drag coefficient is reduced to 0.008812 (3.1% reduction) which is only 0.47% higher than what was obtained for the shape optimization with $y_{\max}$ of 10 mm, but as expected the topology obtained in this step is not discrete.

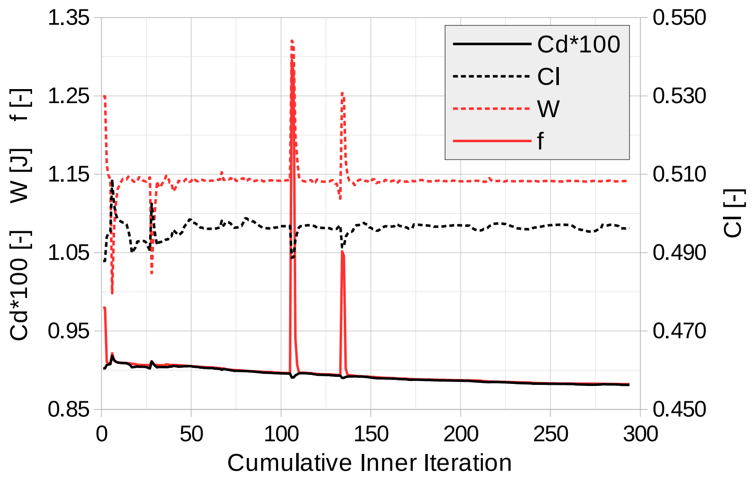


Figure 5.4: *Fully coupled* step convergence history.

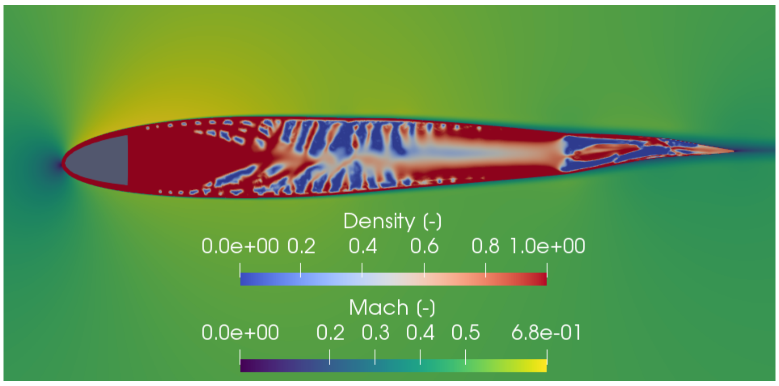

Density (-)
0.0e+00 0.2 0.4 0.6 0.8 1.0e+00
Mach (-)
0.0e+00 0.2 0.3 0.4 0.5 6.8e-01

Figure 5.5: *Fully coupled* step results, filtered structural density and Mach number contours.

5.1.3 Inverse design step

The *inverse design* step aims to obtain a discrete-topology structure. It is stated as a mass minimization problem with constraints on the target deformed shape of the airfoil and on the stability metric (whose formulation was described in chapter 2, equations (2.37) and (2.38) respectively), that is

$$\begin{aligned} \min_{\rho} \quad & \frac{1}{V}\int_V \rho \mathrm{d}\Omega \\ \text{subject to :} \quad & \epsilon_\Gamma^{l/h}(\rho) \leq 0.01 \\ & W_\Gamma^{+l/h}(\rho) \leq 0.01 \end{aligned} \tag{5.4}$$

The results for this step are shown in Fig. 5.6, where the black line shows the target deformed surface, and the red line shows the verified deformed surface obtained from the coupled FSI analysis loop for the topology obtained in this step. To reduce the computational cost of the *inverse design* step, the stability constraints (on W_Γ^+) were only enabled after the design became feasible with respect to the geometric constraints (ϵ_Γ). This allows part of the inverse design step to simulate only the structure, transitioning to one-way FSI (needed to evaluate W_Γ^+) only when the deformed configuration is near the target (for which good initial conditions are available for the fluid problem). Table 5.2 summarizes the two-step topology optimization process listing the aerodynamic coefficients and the equivalent mass (for SIMP exponent of 3) at the major checkpoints.

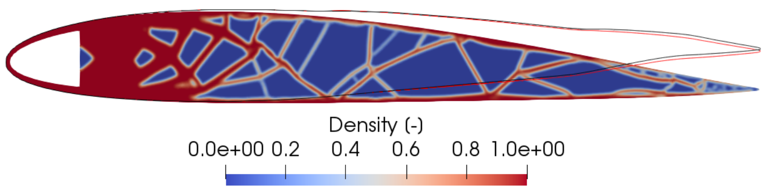


Figure 5.6: Results of *inverse design* step, density contours, target (black) and actual (red) deformed surfaces.

Step	C_d^h	C_l^h	C_d^l	C_l^l	Mass
Shape 6.0 mm	0.009097	0.360	0.0107	0.498	0.794
Fully coupled	0.008812	0.238	0.0107	0.496	0.695
Inv. design	0.008846	0.260	0.0107	0.498	0.438

Table 5.2: Topology optimization summary.

With the *inverse design* step some of the improvement obtained in the *fully coupled* step is lost as the drag coefficient increases 0.39%. However, the equivalent mass is reduced by 37%. The reduced performance is mostly due to strong coupling effects, that amplify the effect of the small discrepancy between target and obtained shapes. In the *inverse design* step the surface error metric is constrained to 1%. However, when the performance of the resulting topology is verified, this metric increases to 2% at Mach 0.5. The main physical mechanism that explains this, is that regions of unsupported airfoil skin tend to form bumps as the airfoil flexes, these bumps cause a reduction of the pressure (due to a local acceleration of the flow) which in turn increases the bump size. This is the main effect the stability function helps to mitigate, Fig. 5.7 shows how this function leads to material being added to the skin to increase its bending stiffness. Coupling a bump-based shape parameterization method (e.g. Hicks-Henne) with the topology variables could potentially reduce the importance of this positive feedback mechanism. That was not deemed necessary for this example due to the subsonic speeds, however the effect would be more significant at transonic speeds as bumps can produce shocks.

The seemingly insignificant increase in surface error metric (from 1% to 2%) results in a 9% increase in lift (but no significant changes to the flow field). This high sensitivity of the design is not entirely due to the proposed two-step method, but also due to the performance metric that was optimized, which results in a system with highly nonlinear characteristics as its apparent stiffness changes significantly between low and high speeds. Finally, note that topology optimization results often require some form of post-processing to which the results may likewise be sensitive (for example to remove vestigial areas of intermediate density, smooth jagged edges, etc.). Therefore, it may be necessary to resort to robust design techniques when

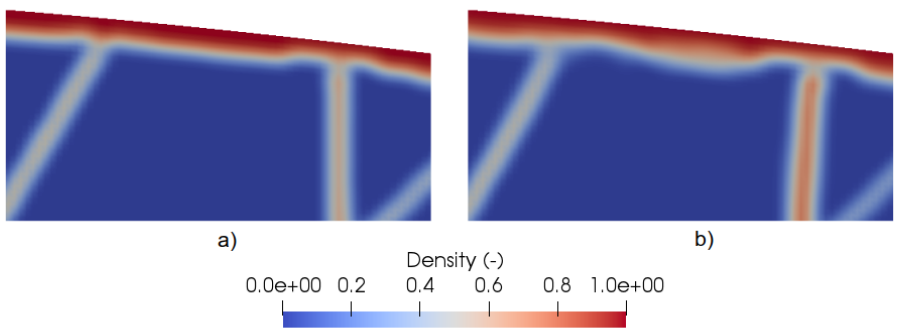


Figure 5.7: Detail of *inverse design* step results w/o (left) and with (right) stability function.

the formulation of the optimization problem results in structures with strong nonlinear response to loads.

5.1.4 Comparison with conventional approaches to encourage solid-void topologies

Two well known ways to encourage solid-void solutions, in problems that do not necessarily benefit from them, are to penalize intermediate densities more severely (e.g. using higher values of SIMP exponent) and to manipulate the problem formulation such that overall stiffness becomes important (e.g. embedding mass reduction into the optimization goals). Both approaches were investigated (in fact, before proposing the two-step process) and the results are presented and discussed in this subsection.

Weighted objective function

First note that the low speed lift and compliance constraints require the airfoil to have a minimum stiffness, recall that the AoA is set relative to the undeformed configuration and so an airfoil that is too flexible will not produce the target lift. Therefore, with a weighted objective function of mass and drag a stiffness-based problem can be recovered by giving no importance to drag and focusing solely on mass minimization (which would not be desirable). To test the *weighted objective* approach, weights of 0.8 for drag and 0.2 for mass were used (after scaling the functions), which based on the drag-mass trade-off from the *fully coupled* step to the *inverse design* step (see table 5.2) should be sufficient. Moreover, numerical experiments with higher weighing of mass were less successful due to poor stability of the optimization.

The optimization process is as described for the *fully coupled* step except for the change of objective function and the SIMP exponent which was not ramped, being fixed at 3 from the start instead (ramping it also made little difference in the results of the previous section). The convergence history is shown in Fig. 5.8, the optimization stalls relatively early (the last 8 iterations required 66 function and gradient evaluations) resulting in the topology of Fig. 5.9, where the deformed configuration from the *fully coupled* step is also shown for comparison.

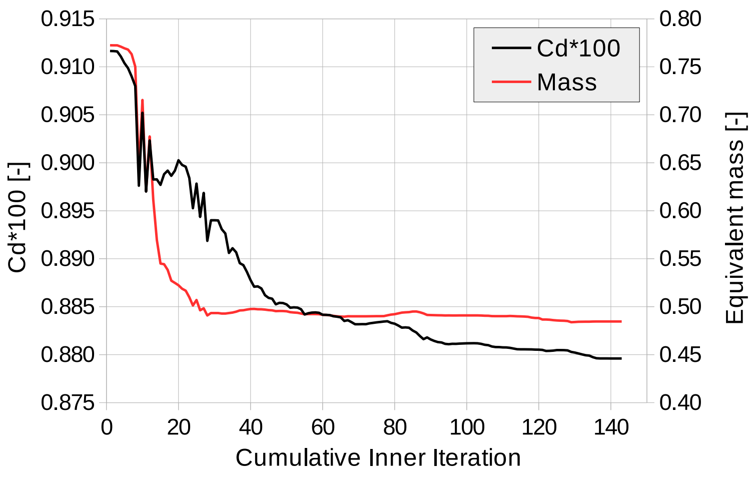
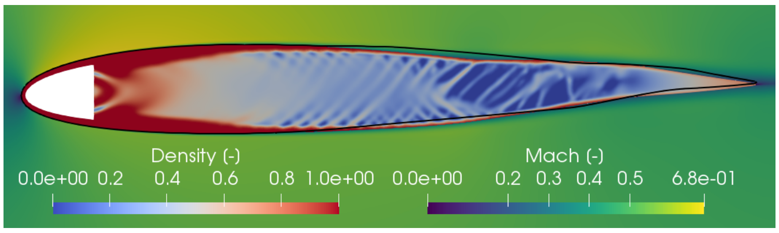


Figure 5.8: *Weighted average* approach, convergence history.

The response of the structure is similar and the drag is lower but within one count of what was obtained with the two-step approach. The equivalent mass is 0.485, lower than in the *fully coupled* step, as expected, but higher than in the *inverse design* step, due to the less discrete material distribution (see Fig. 5.9).



Density (-)
0.0e+00 0.2 0.4 0.6 0.8 1.0e+00
Mach (-)
0.0e+00 0.2 0.3 0.4 0.5 6.8e-01

Figure 5.9: *Weighted average* approach results, filtered structural density and Mach number contours, deformed configuration from *fully coupled* step in red for reference.

Higher penalization

It is known that challenging topology optimization applications may require a SIMP exponent higher than that suggested by the Hashin-Shtrikman bounds to converge to solid-void solutions. To test this approach the *weighted objective* optimization was continued with the SIMP exponent increased from 3 to 4 in steps of 0.25 every outer iteration of the exterior penalty method. The resulting material distribution, shown in Fig. 5.10, is almost indistinguishable with no significant topological changes. The drag coefficient was not improved and the equivalent mass increased to 0.542 as a result of the stiffness reduction in intermediate density areas. The lack of change could be due to the solution for SIMP exponent of 3 being a local optimum. However, the general features of the solution, dense truss-like structure at mid chord and large voids towards the trailing edge, develop very early in the optimization (note the quick decrease in mass in Fig. 5.8). As material is removed mostly from low strain energy areas, it is possible that these solution features are inherent to the presence of the mass objective, and how quickly and easily it can be targeted by the optimizer (note that the mass function is linear).

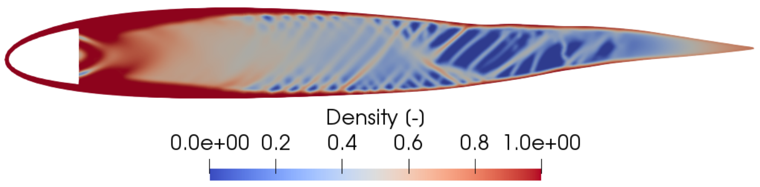


Figure 5.10: Density contours on the deformed geometry for the *weighted average* objective with SIMP exponent of 4.

Gradual mass minimization

Based on the observation just above, it could be expected that gradually introducing the mass objective would avoid rapid convergence to the poor local optima described. This was attempted for the results of the *fully coupled* step in two-steps. First by adding a constraint to the optimization problem (5.2) to gradually lower the equivalent mass to 0.5, the initial penalty function parameters were selected such that the initial penalization was equivalent to two drag counts. Then by switching to constraining drag below 0.00885 while minimizing mass, this is required since once both mass and compliance constraints are satisfied there is no longer an incentive to improve discreteness (one of the two must be a scarce resource). Fig. 5.11 shows the convergence history, illustrating the more gradual decrease in mass, and Fig. 5.12 the obtained topology.

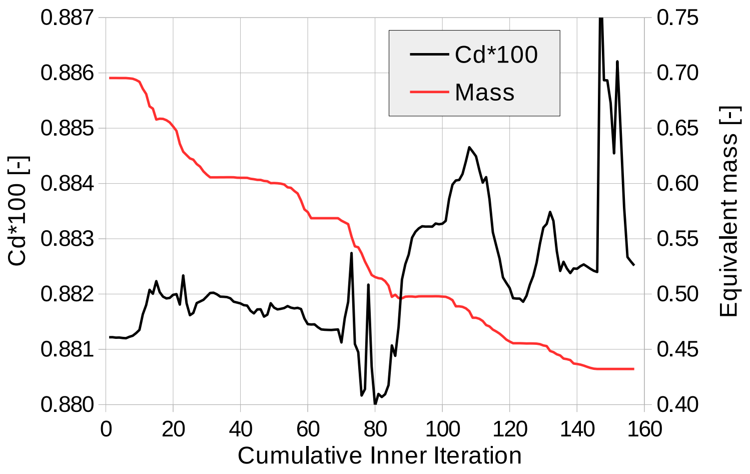


Figure 5.11: Convergence history for *gradual mass minimization* approach.

Although mass is reduced without significantly increasing drag, and both values are marginally lower than those obtained after the *inverse design* step, the resulting topology is still far from discrete. Note that the optimization does not converge fully as the gradient of the penalized

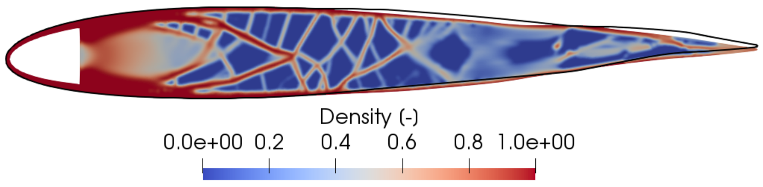


Figure 5.12: Density contours after the *gradual mass minimization* approach (deformed geometry), *fully coupled* step deformed outline (in black) for reference.

objective function (see Fig. 5.13) is not zero. Nevertheless the line searches fail due to the much larger gradients at the weakly connected regions around 80% chord, which result in poor search directions.

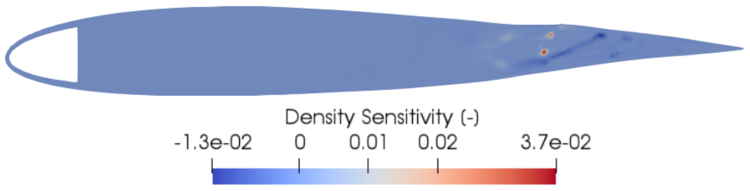


Figure 5.13: *Gradual mass minimization* approach, contours of derivative of penalized objective function with respect to density.

5.1.5 Discussion

The trailing edge region is not highly stressed as bending moments are low, therefore from a purely structural perspective it does not require large regions of solid material. However, from the aerodynamic standpoint the trailing edge is responsible for most of the load alleviation. While the topology features close to the leading edge mostly confer stiffness to the airfoil, the trailing edge behaves like a mechanism, one that under passive actuation notably increases the camber near the trailing edge at the higher speed. This localized camber leads to a more aft loaded pressure distribution (see Fig. 5.14 light color line) that counteracts the effect of the reflexed camberline of the undeformed airfoil.

It is plausible, therefore, that this interference between aerodynamic and structural objectives leads to an intermediate design and search direction that stall the optimization. Although the problem is posed such that it benefits globally from high stiffness to mass ratio, locally (near the trailing edge) that is not what minimum drag requires. Different strategies can potentially be used to mitigate the interference between objectives without decoupling them. However, it is worth noting that a strategy akin to the *gradual mass minimization* approach above nearly doubled the computational cost, whereas the *inverse design* step only added 15% to the cost of the *fully coupled* step.

Although simple, this problem proved challenging for density-based topology optimization. This is in part due to the flexibility that is required to explore large displacements in 2-D, which

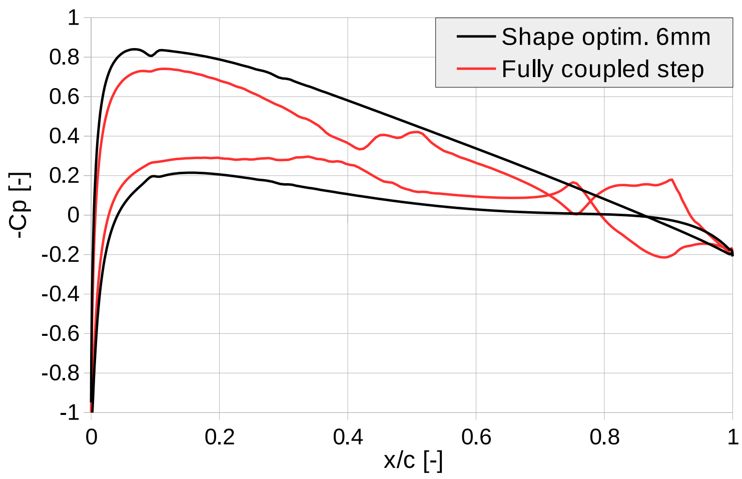


Figure 5.14: Pressure coefficient distribution at Mach 0.5 for the baseline airfoil (6 mm constraint) and for the *fully coupled* step result.

compromises the smoothness of the outer shape and thus increases drag. Note how the topology that resulted from the *fully coupled* step provides much more support to the airfoil shell than the topology from the *inverse design* step. In effect, this discrete-topology was only possible due the relaxation of the problem, that results from approximately matching a deformed shape instead of minimizing drag. Obtaining a discrete topology with the same level of outer layer support (as the non-discrete result) could be possible, but it would require an impractical level of detail, especially for a 3-D problem. Finally, note that the subtle increases in drag due to irregularities of the deformed surface would not be captured by lower fidelity fluid models.

5.2 A flexible wing

Similarly to the problem of section 5.1, the purpose of this example is to study the feasibility of using the internal topology of a wing to alter its aerodynamic performance. In effect, this is a proof of concept for a new design strategy for wings built using additive manufacturing techniques. Specifically, in this example the internal material distribution will be used to obtain a more flexible wing (and thus presumably lighter) that produces the same lift at the same angle of attack as a rigid baseline design. This is effectively a load augmentation problem, where the deformation of the structure creates a positive feedback mechanism that increases the aerodynamic load (i.e. the lift). Although changing the angle of attack would be the most practical way to change the lift, doing so would not serve the objective for this example. In effect, introducing the angle of attack as a design variable in this problem with a single operating point, would convert it almost into a typical minimum-compliance problem. Shape optimization is first used to produce a minimum-drag design with small deformations, this is described in section 5.2.1. Topology optimization is then used on this design, which should show the impact of load augmentation on aerodynamic performance. The previous 2-D example already showed the potential of using the internal layout to passively adapt to different operating conditions. However, to assess feasibility in 3-D, only one operating point will be considered. Moreover, given the challenges encountered in the 2-D problem of section 5.1, a different relaxation approach is used. Specifically, to guarantee that the outer layer is never left without support, the stiffness of the "void" material is increased, which effectively results in a two-material optimization where the entire structure is solid. Such a structure may be easier to manufacture than one with hollow regions. For example, if additive manufacturing techniques are used, auxiliary support structures are not needed since the two materials support each other. This approach was recently used in [21] to design wind tunnel models with tuned flutter characteristics using medium fidelity modelling. Here, the focus will be on static characteristics and large deformations, but using high fidelity models, which the previous example showed to be important for that type of problem.

5.2.1 Baseline rigid design

A baseline design is obtained by optimizing the shape of a solid wing with uniform material properties for minimum drag, with constraints on lift, pitching moment, and minimum thickness (over the multiple spanwise sections). The initial geometry shown in Fig. 5.15, is generated by *lofting* two symmetric 4-digit NACA profiles. The root profile has 0.25 m chord and 9% thickness, whereas the tip has 0.175 m chord and 7.2% thickness. The wing span is 1 m, the 25% chord-line is swept back 5 degrees, and a linear twist distribution of -3 degrees is used. Fig. 5.15 also shows the FFD box used to parameterize the shape. Of the 98 control points, 97 are allowed to move in the vertical direction (± 30 mm), the bottom right point of the root section (right side of the page) is fixed. Note that these variables are able to change the effective angle of attack by pitching the wing, despite the flow direction not being changed by the optimizer. Chord and translation (chord-wise displacement) are defined for each of the 7 spanwise sections of the FFD box by controlling the horizontal coordinates of the 14 control points in each section. The tip section is not allowed to translate to help maintaining the quality of the mesh. The bounds for these two variables are (± 60 mm) at the root and decrease linearly to (± 20 mm) at the tip. There are 110 variables in total. A single operating point was considered, namely Mach 0.6 at sea-level and 273.15 K (the Reynolds number is 3.7 M for the root chord). Convective fluxes are computed using Roe's scheme with 1% entropy correction, the second order reconstruction of flow variables uses Green-Gauss gradients and Venkatakrishnan and Wang's limiter. The Spalart-Allmaras turbulence model is used with first order convective fluxes. The fluid and structural grid are composed mostly by hexaedra and have 4.1 million nodes and 220 000 nodes, respectively. The grid sizing of spanwise sections is comparable to the level 5 NACA0012 grid used in [48]. The sizing in the spanwise direction is such that the aspect ratio of surface elements near the tip is close to one, and such that the maximum aspect ratio of the first layer of volume elements was below 200. Although a rigorous mesh independence study was not conducted, the results with this level of refinement were in agreement with coarser meshes (below 1 million nodes), of the same type, and for similar flow conditions, used for performance benchmarking and adjoint verification (in chapters 4 and 3, respectively). The radius of the farfield boundary is 25 span lengths, Fig. 5.16 shows a detail of the structural grid near the wing tip. The solid material is modelled as hyperelastic with Poisson's ratio of 0.35 and Young's modulus of 75 GPa, which results in small deformations (comparable to the thickness).

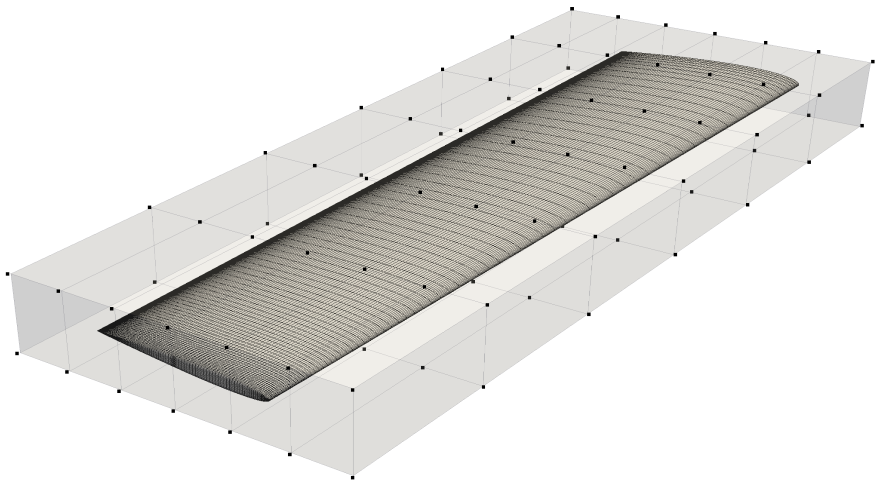


Figure 5.15: Initial wing geometry and FFD box.

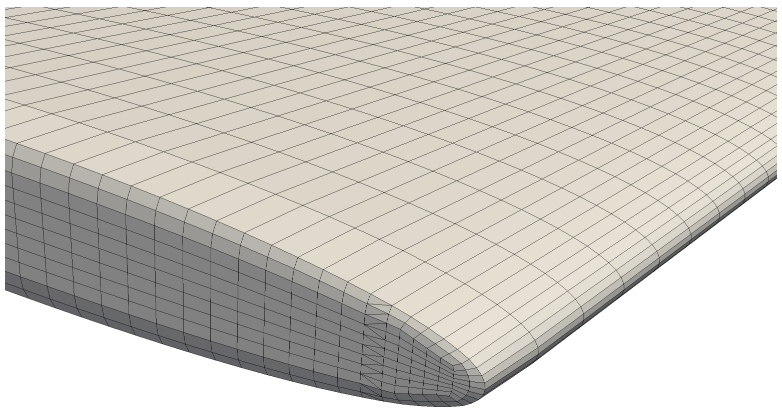


Figure 5.16: Structural mesh near wing tip.

At 4 degrees AoA the initial geometry has lift, drag, and pitching moment coefficients of 0.295, 0.0103, and -0.0429 respectively. To obtain a baseline for topology optimization, drag was minimized with lower bounds of 0.29 on lift and -0.06 on pitching moment coefficients. A geometric constraint was also included to prevent the maximum thickness of the thinnest spanwise section from decreasing. The SLSQP implementation from SciPy was used to solve the optimization problem, all constraints and variables were scaled by their bounds and the objective (drag) by its initial value.

Fig. 5.17 shows a comparison of the planforms and spanwise sections at the root, mid-span, and tip of the initial and optimized geometries. All constraints are active on the optimized design and the drag coefficient was reduced to 0.00829. The design achieves this by reducing the loading on tip sections and increasing it in the first half of the span. The sections towards the tip are significantly cambered, which requires the root sections to operate with more incidence to counteract the effect of camber on pitching moment, as shown by the pressure coefficient contours in Fig. 5.18. However, note that despite the increase in incidence near the root, on the optimized design those sections produce less lift due to a reduction in chord. Similarly, the sections near the tip produce more lift due to camber, despite the lower incidence. This is shown by the lift distributions in Fig. 5.19. As expected when minimizing drag at subsonic conditions, the result is a more elliptical lift distribution.

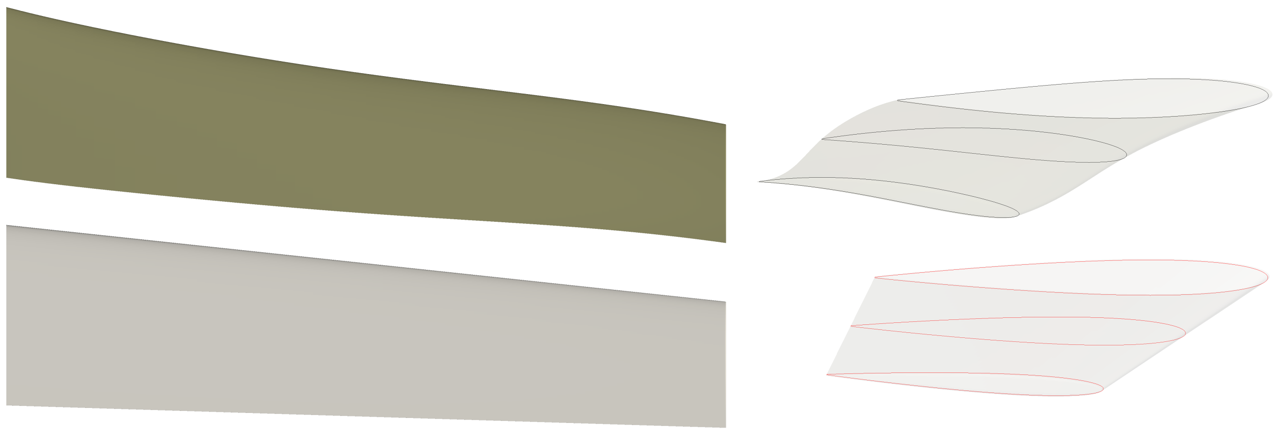


Figure 5.17: Comparison of initial (bottom) and optimized (top) wing.

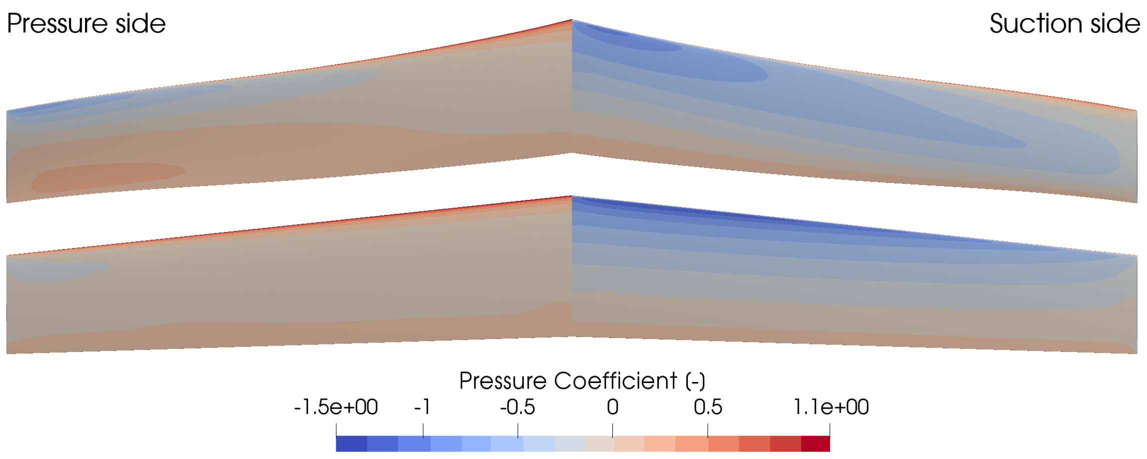


Figure 5.18: Pressure coefficient contours on initial (bottom) and optimized (top) designs.

Due to this camber (and moment) distribution, lift decreases if a more flexible material is considered in the simulation, note that the aerodynamic forces act to twist the wing, which reduces incidence and thus lift. Of course, the AoA can be increased to counteract this effect (up to the point of static divergence), here however, topology optimization is used to obtain a

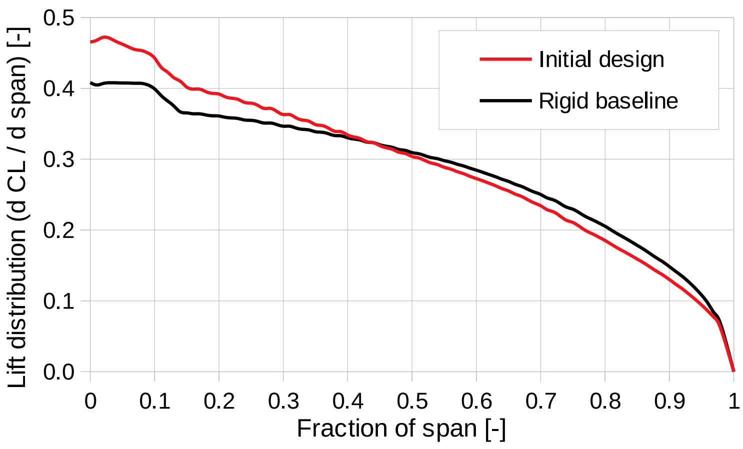


Figure 5.19: Spanwise lift distribution on initial (light color) and optimized (dark color) designs.

two-material structure that, although more flexible, is able to compensate the twisting action of the aerodynamic forces. The optimized geometry was re-meshed before proceeding to optimize the internal topology.

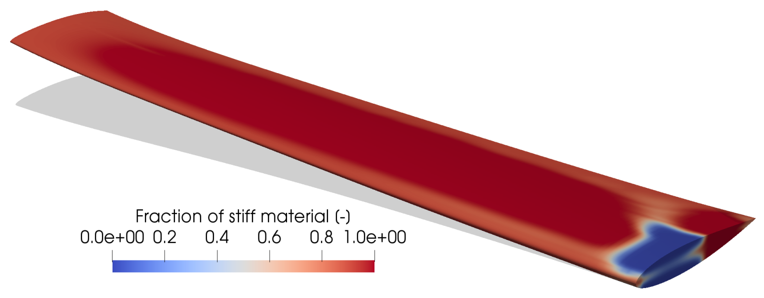
5.2.2 Topology optimization

For topology optimization the soft and stiff materials were considered to have elasticity modulus of 1.5 and 30 GPa, respectively. It is not relevant for the purpose of this study if such materials exist, these values were simply chosen based on expected deformation at 50% material fraction (comparable to the chord). The structural elements are stretched (see Fig. 5.16), because using a smaller size in the spanwise direction would result in an impractical number of elements. This presents a challenge for density filtering strategies, since setting the filter radius based on the longest element size would span the entire thickness of the wing. To avoid this, the neighborhood search was constrained to consider only neighbors of neighbors, on a 2-D grid this limits the filter to the equivalent of a 13-point stencil. The filter radius was then set as 1.5 times the largest element size (in the spanwise direction), and a linear weight function was used. Finally, a fixed SIMP exponent of 3 is used throughout the investigation.

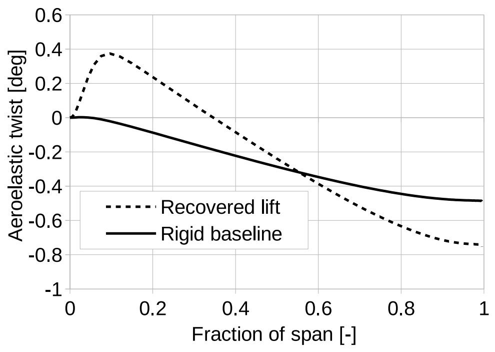
Due to the reasons described above, with a material with elasticity modulus of e.g. 22.3 GPa (equivalent to the starting point in 5.2.2) the lift coefficient drops to 0.24. Then, a first objective is to recover the target value of 0.29, using the minimum amount of stiff material. Note that a density of zero corresponds to the softer material and a density of one to the stiffer material, therefore minimizing the fraction of the latter is equivalent (in formulation) to minimizing mass in a classical "solid-void" problem. Note that it is not possible to do so "simply" by stiffening the structure, since the stiffer material is still 60% more flexible than the one used for the baseline rigid design. Of course, some structural constraint has to be introduced (the softer material is too flexible), although maximum stress would be most appropriate, it is less challenging (numerically) to limit compliance. Moreover, given the challenges associated with aerodynamic-driven topology optimization (described in the previous section), it was judged as beneficial to divide the optimization in several steps, also as a way to make the trade-offs between goals more evident. These steps are discussed next. The starting point for each step are the results from the previous, unless noted otherwise. While this may lead to some loss of performance, such incremental strategies are necessary to understand the challenges of new problems, especially those with large design spaces.

Recovering lift

The first step was to recover lift, which also serves as evidence for the feasibility of the optimization problem. To that end the exterior penalty algorithm (2.35) was used with a single equality constraint with fixed penalty factor of 32. Note that this method would start by recovering lift even if other goals were included because this constraint is not respected. Fig. 5.20a shows the surface view of the resulting topology after the 11 L-BFGS-B iterations required to meet the constraint, starting with uniform fraction of 0.9. By removing stiff material close to the leading edge, the incidence on spanwise sections close to the root negates the pitch-down moment of the tip sections, as shown by the distribution of aeroelastic twist (i.e. relative to the jig shape, and thus the variation in AoA due to the deformation) in Fig. 5.20b. In part this is also necessary to compensate the smaller projected area that results from larger displacements (because the structural model accounts for geometric non-linearity).



(a) Material distribution to meet lift constraint.



(b) Spanwise distribution of twist.

Figure 5.20: Results of recovering lift.

Determining a suitable bound for the fraction of stiff material

In the previous step there was no incentive to use a minimal amount of stiff material. That is introduced in this step alongside an upper bound on compliance of 75 J (the compliance after the initial step is 44.8 J). Although it is known that minimizing mass for a given compliance may give rise to structures that are close to a stability limit, this formulation helps determining a suitable target for the stiff fraction (which can then be constrained instead of outright minimized). The penalty factors for the two constraints were fixed at 8 and the objective (stiff fraction) was not scaled since its initial value (resulting from the previous step) was close to one. Fig. 5.21 shows the resulting topology after 55 L-BFGS-B iterations. Mass fractions below 0.5 were made transparent to reveal the internal structure, which is mostly composed of the softer material. On the figure, the spanwise slices are taken from 0.25 to 0.55 span in increments of 0.075. The main mechanism to maintain lift remains the same as in the initial step to recover lift, and the stiffer material was removed from less strained areas, e.g. close to the wing tip. The stiff fraction was reduced to 0.33, which made the compliance constraint active. However, the material distribution is not discrete, mainly for two reasons: first, due to the relatively small number of iterations (for a topology problem); and second, due to the lift constraint, which requires material to be removed from a high strain region (the root). Note that this constraint is responsible for the main load applied to the structure (and thus in part for its compliance) and this contradiction between the two constraints makes the problem more difficult to solve.

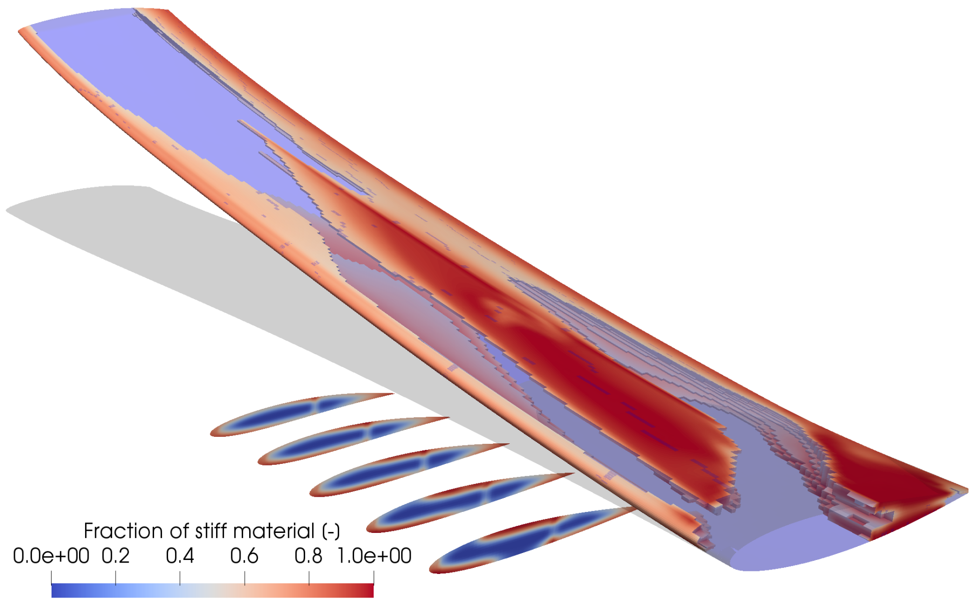


Figure 5.21: Material distribution after minimizing stiff fraction.

Lift-constrained compliance minimization

To avoid the contradiction between constraints (described above), the roles of compliance and stiff fraction were switched from 5.2.2, i.e. compliance should be minimized under the 0.29 lift coefficient constraint and a maximum stiff fraction of 0.4 (if this value was known, e.g. from experience, the previous step would not have been necessary). The initial penalty factors were again set to 8 but increased by 50% (up to 64) every 30 L-BFGS-B iterations for constraints that did not meet a 0.5% tolerance. Note that constraints are scaled by their bounds and the objective by initial value. Fig. 5.22 shows the resulting topology after 225 iterations, which has compliance of 38.1 J, lift coefficient of 0.286, and stiff fraction of 0.403. The material distribution is significantly more discrete than before, but not as much as desired. Although the type of filter used (linear weight) will never produce a discrete result, only 50% of the variables are at their bounds (0 or 1), it should be possible, therefore, to improve this result.

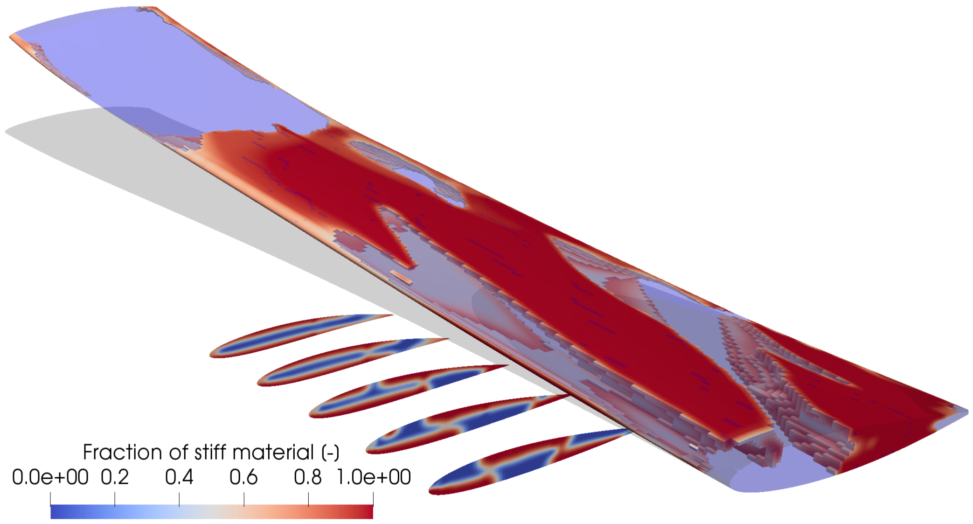


Figure 5.22: Material distribution after minimizing compliance.

Improving the discreteness of the material distribution

Ideally a topology optimization problem should benefit from a discrete solution due to the way it is posed and parameterized. However, in this problem (and others like it), the requirement on lift makes this difficult to achieve, since the structure is both responsible for creating the load and resisting it. It is worth showing that posing the problem as a compliance (and lift) -constrained mass minimization is not beneficial. To that end the roles of stiff fraction and compliance were switched (again). The penalty factors were fixed at 64 and the bound on compliance set to 40 J to direct the optimization towards the current local optimum (i.e. the result of the previous step). Fig. 5.23 shows the resulting topology after 180 iterations.

The stiff material fraction is reduced to 0.336 with lift coefficient of 0.285 and compliance of 40.2 J. However, it is clear that conventional formulations do not produce a discrete result for this problem. Along with reducing the size of areas with intermediate material fraction (note the larger gaps from Fig. 5.22 to Fig. 5.23), i.e. moving the boundaries, the optimization also removed stiff material from areas that were already completely stiff (note the change near the leading edge at 60 to 70% span on the same two figures). A plausible explanation for this is that, to support a distributed load, it is better to distribute the stiff material than to concentrate it to produce stiff regions. In the literature, authors have used explicit penalization of non-discreteness metrics for problems that do not converge naturally to discrete solutions (see e.g. [142]). It is known that such strategies need to be managed with care (e.g. ramped),

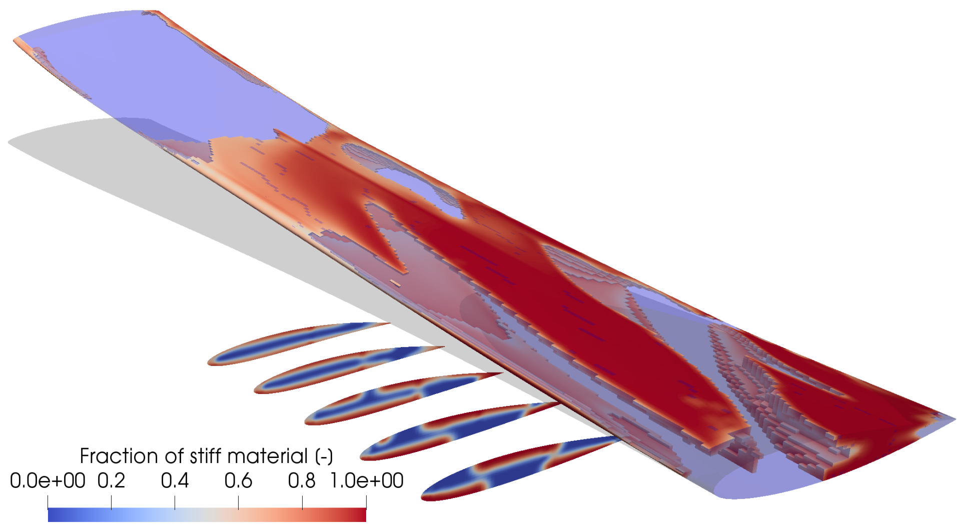

Fraction of stiff material (-)
0.0e+00 0.2 0.4 0.6 0.8 1.0e+00

Figure 5.23: Material distribution after minimizing stiff fraction for a lower compliance constraint.

to avoid fast convergence to locally optimal solutions before any topological features begin to develop. However, ramping strategies have the downside of increasing the computational cost of the optimization, which is already significant for this problem due to high-fidelity modelling. Moreover, even for topology problems that naturally converge to discrete solutions, most of the objective function reduction takes place in the first iterations, where the main features of the topology develop. The remainder of the iterations serve mostly to refine those features, and in general, do not contribute substantially towards improving the objective function. For example, note how in this problem the lift constraint can be met by a wide range of topologies (from mostly soft to mostly stiff). Based on these observations it is tempting to propose an early termination of the optimization, followed by a post-processing operation to force the discreteness of the result (for example considering all values above a certain threshold to be completely stiff, and vice versa). Though this may be viable for more linear problems, in the presence of strong FSI it is likely that some constraints will no longer be respected after the post-processing.

Therefore, to address the challenges of high computational cost and of obtaining discrete topologies, the solution proposed here is as follows. After obtaining a feasible solution (e.g. either 5.2.2 or the one in this section), the discreteness of the solution is improved by explicitly targeting (minimizing) a non-discreteness metric (d), namely

$$d = 4\frac{\boldsymbol{\rho} \cdot (1 - \boldsymbol{\rho})}{N}, \tag{5.5}$$

where $\boldsymbol{\rho}$ are the filtered densities (material fractions in this case) and N the number of design variables. The reduction of computational cost is achieved by simulating only the structure under fixed fluid loads (obtained from the last FSI simulation). To ensure those loads are appropriate throughout the entire process, a deformation target is introduced for the surface (Γ) nodes, that is,

$$\epsilon_\Gamma = \frac{||\mathbf{u}_\Gamma - \mathbf{u}_\Gamma^*||^2}{N_\Gamma}. \tag{5.6}$$

This error metric is both minimized (as a secondary objective) and constrained to a small value (10^{-7} in this case). Despite the similar formulation, this step has a very different role than the inverse-design approach used in section 5.1. Here, the initial topology is nearly discrete, and thus only requires some incremental refinement. Conversely the inverse approach does not use initialization and therefore produces substantially different material layouts.

On the result from stiff fraction minimization for compliance of 40 J the discreteness metric is 0.3. This is reduced to 0.2 over the course of 1000 (inexpensive) iterations, after which the error metric is $4 \cdot 10^{-8}$ and the stiff fraction 0.339. To ensure that the final solution would be as discrete as the type of filter allows, the weight of the non-discreteness metric was progressively increased up to 500. In the end 75% of the variables were within 2% of their bounds. Furthermore, evaluating (5.5) for the design variables (i.e. unfiltered fractions) yields 0.008.

As expected, the lift constraint is not respected by this solution, for which the value is 0.272. To restore it to the target 0.29, the discretized solution was used as starting conditions for an optimization with coupled FSI simulations, where non-discreteness is also minimized. After 25 iterations both this metric and the stiff fraction were maintained, lift was restored but compliance increased to 42.2 J. The resulting material distribution is shown in Fig. 5.24 and Fig. 5.25.

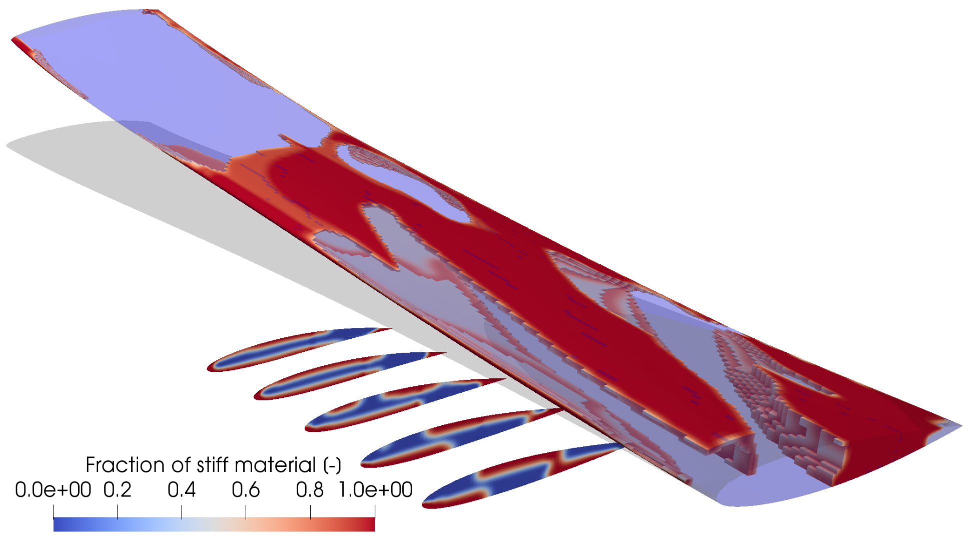


Figure 5.24: Material distribution after improving discreteness.

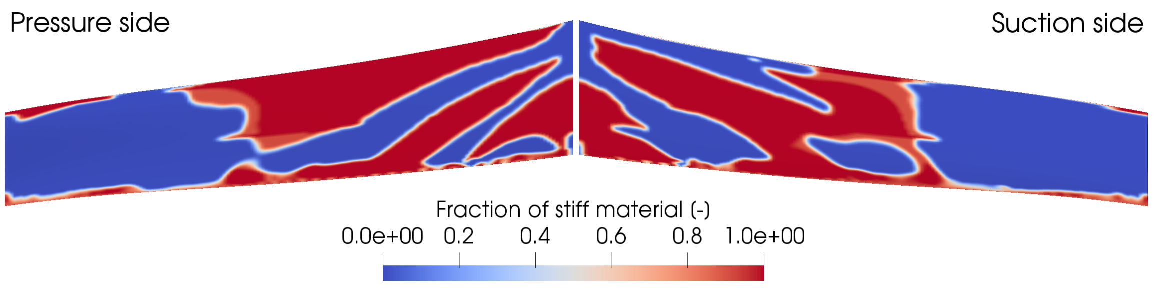


Figure 5.25: Material distribution after improving discreteness (different view).

Discussion

The optimization process in the subsections so far can be summarized in the following steps.

1. Determining the feasibility of the aerodynamic goals, e.g. meeting lift ($O(10)$ iterations).
2. Approximately solving the optimization for suitable structural targets ($O(100)$ iterations). This may include determining those targets via different formulations of the problem (as done here).
3. Accelerating convergence to a discrete solution considering only the structure and fixed loads (equivalent to $O(10)$ iterations).
4. Correcting the aerodynamic goals for the discrete solution ($O(10)$ iterations).

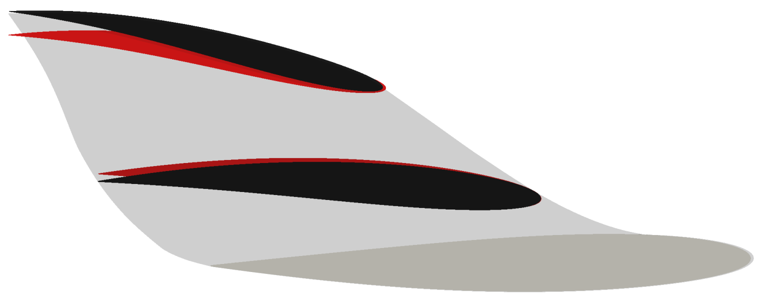
The main part of the optimization takes place in step 2, with steps 3 and 4 constituting a post processing operation. Moreover, note that step 1 may be dispensed if the aerodynamic targets are guaranteed by other variables (e.g. shape or AoA). The drag coefficient for the design that only meets the lift constraint is 0.00917 (step 1). Some of the increase with respect to the baseline is due to re-meshing, most however, is due to the larger deformation which reduces the projected area of the wing, and thus requires the loading to be increased past the optimum found by shape optimization. This is expected to become worse with larger deformations at smaller fractions of stiff material. It is worth noting that single-point optimum shapes for flexible wings tend to result in flat deformed shapes (i.e. straight line from root to tip), which should be expected since for a given span this maximizes the projected area (after deformation). For the designs with smaller stiff fraction (after 5.2.2), the drag coefficient increases to 0.011 and in the final discrete solution (that meets the lift constraint more strictly) it is 0.0115. Table 5.3 shows a summary of the key performance values after each step of the optimization.

The 23 count increase in drag from the first to last step is significant, and not entirely justified by larger deformations. Recall that the lift-generation mechanism used by the topology optimization is to allow the mid span to pitch-up to both reduce and compensate the effects of higher pitch-down moments near the tip. In addition, to reduce the stiff fraction with limited

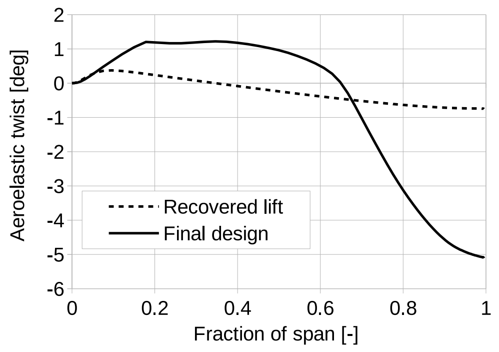
Step	$C_D[-]$	$C_L[-]$	Compliance $[J]$	Stiff fraction $[-]$
Recover lift	0.00917	0.290	44.8	0.938
Min. compliance	0.0109	0.286	38.1	0.403
Min. stiff frac.	0.0113	0.285	40.2	0.336
Final design	0.0115	0.290	42.3	0.338

Table 5.3: Performance values at different optimization steps.

compliance requires material to be removed from low strain regions, note how past 60% span the design uses almost no stiff material. Consequently, this allows those areas to deform more, that is, incidence is reduced due to the pitch-down moments. This reduces the lift generated by those sections of the wing, in turn requiring the inboard sections to produce even more lift to compensate (drag increases as the spanwise distribution of lift moves away from the optimized baseline). This effect is visible in Fig. 5.26, where the deformation of the final design is compared with the one from section 5.2.2, and in Fig. 5.27 where the lift distribution is compared with the one from shape optimization.



(a) Deformation at mid-span and tip, of the final design (dark colors, reduced by 20%), and the design obtained to meet the lift constraint.



(b) Spanwise distribution of twist.

Figure 5.26: Comparison of deformed configurations, between recovering lift and final design.

The vestigial areas of stiff material near the tip of the wing, whose placement at the leading and trailing edges maximizes torsional stiffness, are justified by the need to maintain some stiffness near the tip to avoid further lift reduction. Note that these reinforced areas are not connected to the larger areas of stiff material, which in effect, reduces the coupling between the two (note the rapid change of aeroelastic twist after 0.6 span in Fig. 5.26b). Moreover, although the final design deforms more in torsion and bending, the corresponding moments are lower. These interruptions of the main load paths (i.e. stiff material portions) are necessary for the aerodynamic goals but raise structural concerns. In particular they cause the soft material to be more strained than its stiff counterpart, due to being compressed and sheared between floating regions of stiff material that are not connected to the root. Figure 5.28 shows the von Mises stresses divided by the local elasticity modulus, which is used here in lieu of a safety margin since the two materials do not exist in practice. It would important, therefore, to include stress

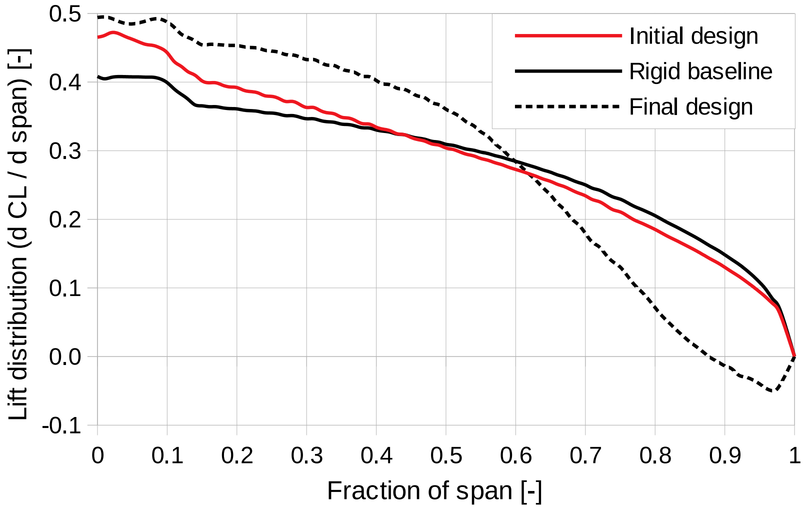


Figure 5.27: Spanwise lift distribution on initial (light color), shape-optimized (dark color), and final (dashed line) designs.

constraints in future work.

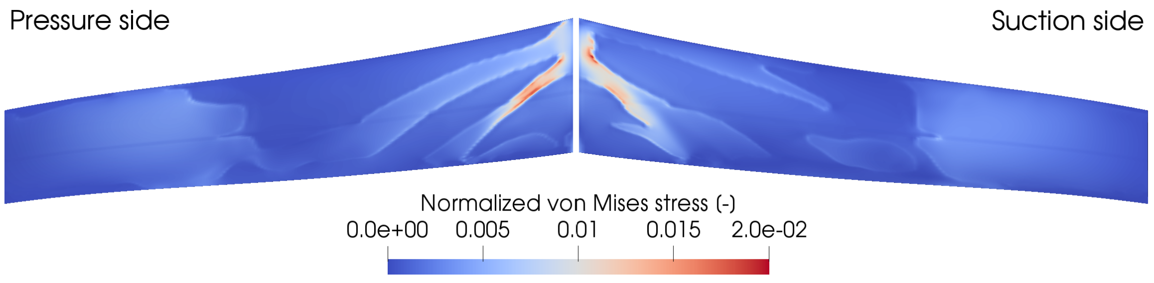


Figure 5.28: Von Mises stress normalized by local elasticity modulus.

5.2.3 Final remarks

This work aimed to explore the merits and challenges of applying a technique (topology optimization) that is well established for purely structural applications, to problems where the structure is largely responsible for the aerodynamic performance. This was done by restricting the example problem to a single operating point and only topology variables. To fully assess the challenges of topology optimization in state-of-the-art wing design methods, many other aspects would have to be considered (shape optimization, multiple operating points, off-design performance, composite materials, manufacturability, cost, etc.). Nevertheless, this use of topology optimization appears to have merit, to the extent that the topological features can be traced back to aerodynamic effects. Furthermore, such features develop in a similar number of iterations as they would for purely structural problems. However, as a consequence of the limited means the optimization was given to realize its goals, the aerodynamic performance of the final design was reduced. Better performance could be expected, for example, from simply allowing the angle of attack to change, thereby allowing the inboard sections to operate at higher incidence without having to compromise the stiffness of the structure so severely by removing stiff material near the root. However, in so doing, the (desired) load augmentation characteristic of the problem would be lost. Ideal performance, on the other hand, can only be expected from a simultaneous optimization of shape and topology, which proved challenging for this type problem. In particular, the exterior penalty approach is much less suited for shape optimization than SLSQP, especially if the design space is augmented with thousands more topology optimization variables (conversely SLSQP is not adequate for large scale optimization problems due to its dense approximation of the Hessian matrix). In [6], shape (twist distribution) and topology optimization were successfully combined using the SNOPT optimizer, which is based on similar methods to the ones used by IPOPT. It is not clear why the latter method was not suitable for shape and topology problems in this work. It is possible that the problem studied in the aforementioned work (wing box of the common research model) is less challenging numerically due to the use of linear elasticity and panel methods for the fluid, and also due to the isotropic element sizes that the shape of the structural domain allowed.

One of the challenges in this type of topology optimization is the higher computational cost of using high-fidelity models, when approximately 10 times more iterations are required than for shape optimization. The greatest challenge, however, is that aerodynamic goals do not benefit from an optimally stiff structure, and thus the optimization will not converge to a discrete solution, unless that goal is explicitly introduced. Of course, level-set topology optimization methods (which were not explored in this work) guarantee that the solution will be discrete. However, a further contributor to the discreteness challenge is that loads are distributed over a large surface, whose smoothness is important for aerodynamic performance. Since it is more efficient to maintain that smoothness with a large number of small supports, than with a few highly rigid connections, the mesh size required to resolve those features may render 3-D application impractical. In that respect density-based methods may have some advantage due to their connection with homogenization theory. In any case, the solutions proposed in this work address both the high computational cost and the discrete-solution challenges. Notwithstanding, comparing level-set approaches with density-based methods for the type of problem studied here should be the subject of future work. For that it will be important to study the interaction between shape and topology optimization, whether by proposing efficient sequential approaches or by using an optimization method that can consider both sets of variables simultaneously.

Chapter 6

Conclusions

This thesis has investigated means to perform concurrent shape and topology optimization of very flexible wings using high-fidelity FSI modeling, accounting for structural non-linearities, and aimed to apply the methods to relatively large-scale problems. To that end, generic discrete adjoint approaches (for arbitrary multi-physics problems), and high performance implementation strategies were also investigated. Chapter 1 introduced the research topics, their challenges, and reviewed part of the existing work (concerning applications). Chapter 2 discussed the simulation and optimization methodology used in this work. Most importantly, it presented a new approach for topology optimization with FSI, which accelerates convergence to discrete topologies (even when that would not be the natural result given the objectives and constraints). Discrete adjoint methods were then discussed in chapter 3, the focus throughout that chapter was on how methods and implementation choices influence the trade-off between performance and generality. A pitfall of the highly generic AD-based fixed-point approach was identified (i.e. the treatment of linear solvers) and solutions for it were proposed, which now allow well know aerodynamic benchmark problems to be solved with SU2. Chapter 4 delved further into the implementation details of unstructured solvers, in particular it details how modern C++ code patterns were used to achieve up to a 10-fold performance increase of SU2 for relevant problems, while simultaneously facilitating the development of future high performance features in the code. Finally, chapter 5 discussed two FSI topology optimization problems that were made possible via the cumulative effect of the work in previous chapters. These showed that the use of high-fidelity models is justified especially when structural deformations affect the smoothness of the surface. Furthermore, they demonstrated how topology optimization can be used to significantly alter aerodynamic characteristics (in particular to produce passive load adaptation). The present chapter starts with a review of the main research outcomes, in light of the goals set in chapter 1, but listing also the secondary outcomes. Finally some advice for future work is given.

6.1 Research outcomes

Considering the objectives and questions of section 1.3. This work has shown that topology optimization can be used to obtain aerodynamic goals, instead of purely structural ones, by carefully tailoring the interaction between a structure and a fluid (question 1). This capability was used for passive load alleviation on a 2-D problem, and the methodology was shown to work also for a 3-D problem. Furthermore, the use of high-fidelity models (nonlinear elasticity and RANS equations), which had not been tried before, proved to be important to capture the effect of surface deformations on aerodynamic drag (question 3), especially for the 2-D example. This made the challenges of applying density-based topology optimization to FSI problem more evident, in particular that the presence of aerodynamic objectives (or constraints) makes the convergence to discrete material distributions more difficult (question 4). A new approach was proposed to deal with this challenge, namely an inverse design process that allows decoupling the main optimization goals from that of obtaining a discrete topology. This strategy was also effective to accelerate the convergence of the optimization from $O(1000)$ to $O(100)$ iterations (question 5). This is an important development since the high computational cost of topology optimization (compared with shape optimization) is one of the main obstacles to the use of high-fidelity models in large scale problems (as discussed in chapter 1).

Another outcome of this work, in collaboration with other researchers, was the development of a generic framework for multiphysics discrete adjoints within SU2. This is highly relevant to facilitate future research, involving even more complex physical phenomena, as it greatly shortens the development time for new multiphysics applications (question 6). However, generic approaches have limitations, ranging from potentially higher memory usage (depending on implementation strategy) to reduced stability or performance due to the more restricted range of methods that can be applied. Some of these have been identified in this work (in chapter 3), and the solutions proposed here should allow larger problems to be studied in future work with the tools developed here (question 7).

Finally, as a secondary outcome, countless other improvements were made to SU2 to improve its performance (up to 10 times), robustness, and correctness. Furthermore, this was done while substantially reducing the number of lines of code, which should hopefully help future developers write better code, and more advanced features, in less time.

6.2 Future work

Although it was shown that topology optimization can be used for aerodynamic goals, it was also clear that the mechanisms used to achieve them may require significant compromise. For example, using the structural response to increase lift resulted in a lift distribution that sacrificed drag, and to some extent the integrity of the structure. Although the combination of shape and topology variables was initially identified as an objective (question 2), and should avoid this issue, it proved difficult to achieve with common optimization algorithms. Different algorithms should be explored in future work, as well as efficient sequential strategies (i.e. alternating between shape and topology), which may allow current methods to be used so far as they are resilient to convergence to local optima. The challenges encountered with obtaining discrete topologies also warrant exploration of other topology optimization approaches, in particular level-set methods, which guarantee discreteness. However, the ability to combine shape and topology should be kept present when exploring other options. The concerns with structural integrity can only be incorporated via stress constraints and the basic functionality for this was put in place (i.e. Kreisselmeier–Steinhauser stress aggregation in SU2). Nevertheless, more work will be required to determine adequate strategies to include them in an already challenging optimization (for example, how to ramp the parameters of the stress aggregation function). Although manufacturability was never a concern for the conceptual problems studied in this work, it is worth commenting on the feasibility of the structures obtained, given how different they are from typical aircraft structures (briefly described in Chapter 1). Some of those differences can also be found in the work of other authors [6, 20, 12]. One of the most notable differences is that spar-like structures only develop with significant resolution [20, 12]. Furthermore these spars typically have truss features, which are unusual. Features akin to wing ribs cannot be found even with very high levels of resolution (i.e. in [12]), this may be due to the redundant nature of this element [10]. Finally, the type of structures that form during the optimization to transfer the loads from the skin (panels) to the internal load-bearing elements are unlike stringers. Instead of extending along the span, these intricate structures are highly three-dimensional and appear almost biological in nature [12]. Moreover they also connect the suction and pressure sides, thereby functioning partly as ribs (which may explain why those do not form). Noting that it was also clear in this work that closely spaced supports of the outer layers perform better than sparsely distributed supports. In effect, topology optimization cannot be expected to recreate traditional wing designs, without somehow including in the optimization the manufacturing, economic, and safety constraints that have led human designers to those structures over time. One way to include such constraints is to define a design domain with the desired features (spars, ribs, etc.), as done in [9], another is to use techniques such as feature mapping to constrain the shape of the topological features. In [143] the authors report on the additional challenges of feature mapping compared to density based topology optimization.

Regarding the progress made to make SU2 capable of larger FSI optimization problems, substantial work is still required to reach the full-aircraft scale. For one the current mesh deformation strategy does not scale well and it is not suited for highly stretched meshes. It is not clear whether a completely different strategy should be sought, or if the solid elasticity analogy still has room for improvement, for example by aggregating stretched elements (thereby reducing the size of the stiffness matrix and consequently its condition number). Similarly, the current solution method for structural problems requires the use of a direct sparse solver (the condition number is higher due to fewer Dirichlet boundaries), which scales quadratically with problem size and will therefore become a bottleneck for larger topology optimization problems. Algebraic multigrid is the state of the art solution for this challenge [12, 89]. Finally, on the fluid side there are also some limitations, the main one is the ineffectiveness of the geometric multigrid strategy for highly stretched and unstructured meshes. Although the initial Newton-Krylov implementation may allow working around the issue for primal problems, this is not carried into adjoint problems (since the Krylov method is not differentiated). Especially when upwind schemes are used due to the more approximate nature of the residual Jacobian, and thus reduced efficiency of the linear preconditioners derived from it. Therefore, algebraic multigrid is not expected to help and the geometric agglomeration strategy should be improved instead. It will also be necessary to revise the way it is differentiated with AD, which currently results in a memory usage comparable to recording multiple iterations (due to pre/post smoothing iterations).

Bibliography

[1] F. Afonso, J. Vale, E. Oliveira, F. Lau, and A. Suleman, "A review on non-linear aeroelasticity of high aspect-ratio wings," *Progress in Aerospace Sciences*, vol. 89, pp. 40 – 57, 2017.

[2] M. Patil and D. Hodges, "On the importance of aerodynamic and structural geometrical nonlinearities in aeroelastic behavior of high-aspect-ratio wings," *Journal of Fluids and Structures*, vol. 19, no. 7, pp. 905 – 915, 2004.

[3] J. K. Kaldellis and D. Zafirakis, "The wind energy (r)evolution: A short review of a long history," *Renewable Energy*, vol. 36, no. 7, pp. 1887 – 1901, 2011.

[4] F. Palacios, T. D. Economon, and J. J. Alonso, "Large-scale aircraft design using SU2," *53rd AIAA Aerospace Sciences Meeting*, no. January, pp. 1–20, 2015.

[5] G. K. W. Kenway and J. R. R. A. Martins, "Multipoint High-Fidelity Aerostructural Optimization of a Transport Aircraft Configuration," *Journal of Aircraft*, vol. 51, no. 1, pp. 144–160, 2014.

[6] K. A. James, G. J. Kennedy, and J. R. Martins, "Concurrent aerostructural topology optimization of a wing box," *Computers and Structures*, vol. 134, pp. 1–17, 2014.

[7] L. J. Smith, L. J. Halim, G. Kennedy, and M. J. Smith, "A High-Fidelity Coupling Framework for Aerothermoelastic Analysis and Adjoint-Based Gradient Evaluation," in *AIAA Scitech 2021 Forum*, American Institute of Aeronautics and Astronautics, Jan 2021.

[8] K. Maute, M. Nikbay, and C. Farhat, "Conceptual layout of aeroelastic wing structures by topology optimization," in *43rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, (Denver, Colorado), Apr 2002.

[9] K. Maute and M. Allen, "Conceptual design of aeroelastic structures by topology optimization," *Structural and Multidisciplinary Optimization*, vol. 27, no. 1-2, pp. 27–42, 2004.

[10] L. Krog, A. Tucker, M. Kemp, and R. Boyd, "Topology Optimisation of Aircraft Wing Box Ribs," in *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, September 2004.

[11] T. W. Chin and G. Kennedy, "Large-Scale Compliance-Minimization and Buckling Topology Optimization of the Undeformed Common Research Model Wing," in *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, January 2016.

[12] N. Aage, E. Andreassen, B. S. Lazarov, and O. Sigmund, "Giga-voxel computational morphogenesis for structural design," *Nature*, vol. 550, no. 7674, pp. 84–86, 2017.

[13] J. H. Zhu, W. H. Zhang, and L. Xia, "Topology Optimization in Aircraft and Aerospace Structures Design," *Archives of Computational Methods in Engineering*, vol. 23, no. 4, pp. 595–622, 2016.

[14] K. K. Maute and G. W. Reich, "Integrated Multidisciplinary Topology Optimization Approach to Adaptive Wing Design," *Journal of Aircraft*, vol. 43, no. 1, pp. 253–263, 2006.

[15] B. Stanford and P. Ifju, "Aeroelastic topology optimization of membrane structures for micro air vehicles," *Structural and Multidisciplinary Optimization*, vol. 38, no. 3, pp. 301–316, 2009.

[16] B. Stanford and P. Beran, "Optimal Structural Topology of a Platelike Wing for Subsonic Aeroelastic Stability," *Journal of Aircraft*, vol. 48, no. 4, pp. 1193–1203, 2011.

[17] B. Stanford, P. Beran, and M. Kobayashi, "Simultaneous Topology Optimization of Membrane Wings and Their Compliant Flapping Mechanisms," *AIAA Journal*, vol. 51, no. 6, pp. 1431–1441, 2013.

[18] S. Townsend, R. Picelli, B. Stanford, and H. A. Kim, "Structural Optimization of Platelike Aircraft Wings Under Flutter and Divergence Constraints," *AIAA Journal*, vol. 56, no. 8, pp. 3307–3319, 2018.

[19] P. D. Dunning, B. K. Stanford, and H. A. Kim, "Coupled aerostructural topology optimization using a level set method for 3D aircraft wings," *Structural and Multidisciplinary Optimization*, vol. 51, no. 5, pp. 1113–1132, 2015.

[20] S. Kambampati, S. Townsend, and H. A. Kim, "Coupled Aerostructural Level Set Topology Optimization of Aircraft Wing Boxes," *AIAA Journal*, vol. 58, no. 8, pp. 3614–3624, 2020.

[21] B. Stanford, "Topology Optimization of Low-Speed Aeroelastic Wind Tunnel Models," in *AIAA Scitech 2021 Forum*, American Institute of Aeronautics and Astronautics, January 2021.

[22] G. K. W. Kenway and J. R. R. A. Martins, "Buffet-Onset Constraint Formulation for Aerodynamic Shape Optimization," *AIAA Journal*, vol. 55, no. 6, pp. 1930–1947, 2017.

[23] T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso, "SU2: An Open-Source Suite for Multiphysics Simulation and Design," *AIAA Journal*, vol. 54, no. 3, pp. 828–846, 2016.

[24] R. Sanchez, T. Albring, R. Palacios, N. R. Gauger, T. D. Economon, and J. J. Alonso, "Coupled adjoint-based sensitivities in large-displacement fluid-structure interaction using algorithmic differentiation," *International Journal for Numerical Methods in Engineering*, vol. 113, no. 7, pp. 1081–1107, 2018.

[25] J. R. R. A. Martins and A. B. Lambe, "Multidisciplinary Design Optimization: A Survey of Architectures," *AIAA Journal*, vol. 51, no. 9, pp. 2049–2075, 2013.

[26] S. Kamali, D. J. Mavriplis, and E. M. Anderson, “Sensitivity analysis for aero-thermo-elastic problems using the discrete adjoint approach,” in *AIAA Aviation 2020 Forum*, June 2020.

[27] “Fun3d manual: 13.6,” in *NASA TM 2019-220416*, 2019.

[28] D. S. Makhija and P. S. Beran, “Concurrent shape and topology optimization for steady conjugate heat transfer,” *Structural and Multidisciplinary Optimization*, vol. 59, no. 3, pp. 919–940, 2019.

[29] C. Lundgaard, J. Alexandersen, M. Zhou, C. S. Andreasen, and O. Sigmund, “Revisiting density-based topology optimization for fluid-structure-interaction problems,” *Structural and Multidisciplinary Optimization*, vol. 58, no. 3, pp. 969–995, 2018.

[30] R. Picelli, S. Ranjbarzadeh, R. Sivapuram, R. S. Gioria, and E. C. N. Silva, “Topology optimization of binary structures under design-dependent fluid-structure interaction loads,” *Structural and Multidisciplinary Optimization*, vol. 62, no. 4, pp. 2101–2116, 2020.

[31] M. Towara, J. Lotz, and U. Naumann, “Discrete adjoint approaches for CHT applications in OpenFOAM,” in *Advances in Evolutionary and Deterministic Methods for Design, Optimization and Control in Engineering and Sciences*, pp. 163–178, Springer, 2019.

[32] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, “A tensorial approach to computational continuum mechanics using object-oriented techniques,” *Computers in physics*, vol. 12, no. 6, pp. 620–631, 1998.

[33] T. A. Albring, M. Sagebaum, and N. R. Gauger, “Efficient aerodynamic design using the discrete adjoint method in SU2,” in *17th AIAA/ISSMO multidisciplinary analysis and optimization conference*, (Washington, D.C.), June 2016.

[34] M. Sagebaum, T. Albring, and N. R. Gauger, “High-performance derivative computations using CoDiPack,” *ACM Transactions on Mathematical Software*, vol. 45, no. 4, 2019.

[35] C. Michler, S. Hulshoff, E. van Brummelen, and R. de Borst, “A monolithic approach to fluid–structure interaction,” *Computers & Fluids*, vol. 33, no. 5, pp. 839 – 848, 2004.

[36] J. Degroote, “Partitioned simulation of fluid-structure interaction,” *Archives of Computational Methods in Engineering*, vol. 20, no. 3, pp. 185–238, 2013.

[37] U. Küttler, M. Gee, C. Förster, A. Comerford, and W. A. Wall, “Coupling strategies for biomedical fluid–structure interaction problems,” *International Journal for Numerical Methods in Biomedical Engineering*, vol. 26, no. 3-4, pp. 305–321, 2010.

[38] G. Hou, J. Wang, and A. Layton, “Numerical methods for fluid-structure interaction — a review,” *Communications in Computational Physics*, vol. 12, no. 2, p. 337–377, 2012.

[39] C. S. Peskin, “The immersed boundary method,” *Acta Numerica*, vol. 11, p. 479–517, 2002.

[40] C. B. Dilgen, S. B. Dilgen, D. R. Fuhrman, O. Sigmund, and B. S. Lazarov, “Topology optimization of turbulent flows,” *Computer Methods in Applied Mechanics and Engineering*, vol. 331, pp. 363 – 393, 2018.

[41] J. Alexandersen, O. Sigmund, and N. Aage, "Large scale three-dimensional topology optimisation of heat sinks cooled by natural convection," *International Journal of Heat and Mass Transfer*, vol. 100, pp. 876 – 891, 2016.

[42] N. Jenkins and K. Maute, "An immersed boundary approach for shape and topology optimization of stationary fluid-structure interaction problems," *Structural and Multidisciplinary Optimization*, vol. 54, no. 5, pp. 1191–1208, 2016.

[43] J.-I. Choi, R. C. Oberoi, J. R. Edwards, and J. A. Rosati, "An immersed boundary method for complex incompressible flows," *Journal of Computational Physics*, vol. 224, no. 2, pp. 757–784, 2007.

[44] W. A. Wall, A. Gerstenberger, P. Gamnitzer, C. Förster, and E. Ramm, "Large deformation fluid-structure interaction – advances in ale methods and new fixed grid approaches," in *Fluid-Structure Interaction*, (Berlin, Heidelberg), pp. 195–232, Springer Berlin Heidelberg, 2006.

[45] R. Sanchez, R. Palacios, T. D. Economon, H. L. Kline, J. J. Alonso, and F. Palacios, "Towards a Fluid-Structure Interaction Solver for Problems with Large Deformations Within the Open-Source SU2 Suite," in *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, January 2016.

[46] F. Menter, "Zonal two equation kw turbulence models for aerodynamic flows," in *23rd fluid dynamics, plasmadynamics, and lasers conference*, (Orlando,FL,U.S.A), July 1993.

[47] J. Blazek, *Computational Fluid Dynamics: Principles and Applications.* Oxford: Butterworth-Heinemann, third ed., 2015.

[48] P. Gomes, T. D. Economon, and R. Palacios, "Sustainable High-Performance Optimizations in SU2," in *AIAA Scitech 2021 Forum*, American Institute of Aeronautics and Astronautics, Jan 2021.

[49] Y. Bazilevs, K. Takizawa, and T. E. Tezduyar, *Computational Fluid – Structure Interaction.* John Wiley & Sons, Ltd, 2013.

[50] J. Bonet and R. D. Wood, *Nonlinear continuum mechanics for finite element analysis, 2nd edition.* Cambridge University Press, 2008.

[51] A. de Boer, M. van der Schoot, and H. Bijl, "Mesh deformation based on radial basis function interpolation," *Computers & Structures*, vol. 85, no. 11, pp. 784 – 795, 2007.

[52] R. Löhner and C. Yang, "Improved ale mesh velocities for moving bodies," *Communications in Numerical Methods in Engineering*, vol. 12, no. 10, pp. 599–608, 1996.

[53] B. T. Helenbrook, "Mesh deformation using the biharmonic operator," *International Journal for Numerical Methods in Engineering*, vol. 56, no. 7, pp. 1007–1021, 2003.

[54] F. J. Blom, "Considerations on the spring analogy," *International Journal for Numerical Methods in Fluids*, vol. 32, no. 6, pp. 647–668, 2000.

[55] G. K. W. Kenway, G. J. Kennedy, and J. R. R. A. Martins, "Scalable Parallel Approach for High-Fidelity Steady-State Aeroelastic Analysis and Adjoint Derivative Computations," *AIAA Journal*, vol. 52, no. 5, pp. 935–951, 2014.

[56] A. Beckert and H. Wendland, "Multivariate interpolation for fluid-structure-interaction problems using radial basis functions," *Aerospace Science and Technology*, vol. 5, no. 2, pp. 125 – 134, 2001.

[57] C. Farhat, M. Lesoinne, and P. L. Tallec, "Load and motion transfer algorithms for fluid/structure interaction problems with non-matching discrete interfaces: Momentum and energy conservation, optimal discretization and application to aeroelasticity," *Computer Methods in Applied Mechanics and Engineering*, vol. 157, no. 1, pp. 95 – 114, 1998.

[58] B. M. Irons and R. C. Tuck, "A version of the aitken accelerator for computer iteration," *International Journal for Numerical Methods in Engineering*, vol. 1, no. 3, pp. 275–277, 1969.

[59] S. Deparis, M. Discacciati, G. Fourestey, and A. Quarteroni, "Fluid-structure algorithms based on Steklov-Poincaré operators," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 41-43, pp. 5797–5812, 2006.

[60] J. Degroote, "Partitioned Simulation of Fluid-Structure Interaction," *Archives of Computational Methods in Engineering*, vol. 20, no. 3, pp. 185–238, 2013.

[61] D. G. Anderson, "Iterative Procedures for Nonlinear Integral Equations," *Journal of the ACM*, vol. 12, no. 4, pp. 547–560, 1965.

[62] G. M. Shroff and H. B. Keller, "Stabilization of Unstable Procedures: The Recursive Projection Method," *SIAM Journal on Numerical Analysis*, vol. 30, no. 4, pp. 1099–1120, 1993.

[63] M. P. Bendsøe and N. Kikuchi, "Generating optimal topologies in structural design using a homogenization method," *Computer Methods in Applied Mechanics and Engineering*, vol. 71, no. 2, pp. 197 – 224, 1988.

[64] H. A. Eschenauer, V. V. Kobelev, and A. Schumacher, "Bubble method for topology and shape optimization of structures," *Structural Optimization*, vol. 8, no. 1, pp. 42–51, 1994.

[65] O. Sigmund and K. Maute, "Topology optimization approaches," *Structural and Multidisciplinary Optimization*, vol. 48, no. 6, pp. 1031–1055, 2013.

[66] G. Allaire, F. Jouve, and A.-M. Toader, "Structural optimization using sensitivity analysis and a level-set method," *Journal of Computational Physics*, vol. 194, no. 1, pp. 363–393, 2004.

[67] M. P. Bendsoe and O. Sigmund, *Topology Optimization: Theory, Methods and Applications*. Springer, 2004.

[68] O. Sigmund and J. Petersson, "Numerical instabilities in topology optimization: A survey on procedures dealing with checkerboards, mesh-dependencies and local minima," *Structural Optimization*, vol. 16, no. 1, pp. 68–75, 1998.

[69] J. Aguilar Madeira, H. Rodrigues, and H. Pina, "Multi-objective optimization of structures topology by genetic algorithms," *Advances in Engineering Software*, vol. 36, no. 1, pp. 21–28, 2005.

[70] M. P. Bendsøe, “Optimal shape design as a material distribution problem,” *Structural optimization*, vol. 1, no. 4, pp. 193–202, 1989.

[71] M. Stolpe and K. Svanberg, “An alternative interpolation scheme for minimum compliance topology optimization,” *Structural and Multidisciplinary Optimization*, vol. 22, no. 2, pp. 116–124, 2001.

[72] J. Petersson and O. Sigmund, “Slope constrained topology optimization,” *International Journal for Numerical Methods in Engineering*, vol. 41, no. 8, pp. 1417–1434, 1998.

[73] L. Ambrosio and G. Buttazzo, “An optimal design problem with perimeter penalization,” *Calculus of Variations and Partial Differential Equations*, vol. 1, no. 1, pp. 55–69, 1993.

[74] T. Borrvall and J. Petersson, “Topology optimization using regularized intermediate density control,” *Computer Methods in Applied Mechanics and Engineering*, vol. 190, no. 37, pp. 4911 – 4928, 2001.

[75] V. Pomezanski, O. M. Querin, and G. I. Rozvany, “CO-SIMP: Extended SIMP algorithm with direct COrner COntact COntrol,” *Structural and Multidisciplinary Optimization*, vol. 30, no. 2, pp. 164–168, 2005.

[76] T. E. Bruns and D. A. Tortorelli, “Topology optimization of non-linear elastic structures and compliant mechanisms,” *Computer Methods in Applied Mechanics and Engineering*, vol. 190, no. 26-27, pp. 3443–3459, 2001.

[77] O. Sigmund, “On the design of compliant mechanisms using topology optimization,” *Mechanics of Structures and Machines*, vol. 25, no. 4, pp. 493–524, 1997.

[78] J. K. Guest, J. H. Prévost, and T. Belytschko, “Achieving minimum length scale in topology optimization using nodal design variables and projection functions,” *International Journal for Numerical Methods in Engineering*, vol. 61, no. 2, pp. 238–254, 2004.

[79] O. Sigmund, “Morphology-based black and white filters for topology optimization,” *Structural and Multidisciplinary Optimization*, vol. 33, no. 4-5, pp. 401–424, 2007.

[80] B. S. Lazarov and O. Sigmund, “Filters in topology optimization based on Helmholtz-type differential equations,” *International Journal for Numerical Methods in Engineering*, vol. 86, no. 6, pp. 765–781, 2011.

[81] A. B. Lambe, G. J. Kennedy, and J. R. Martins, “An evaluation of constraint aggregation strategies for wing box mass minimization,” *Structural and Multidisciplinary Optimization*, vol. 55, no. 1, pp. 257–277, 2017.

[82] G. Cheng and Z. Jiang, “Study on topology optimization with stress constraints,” *Engineering Optimization*, vol. 20, no. 2, pp. 129–148, 1992.

[83] P. Duysinx and M. P. Bendsøe, “Topology optimization of continuum structures with local stress constraints,” *International Journal for Numerical Methods in Engineering*, vol. 43, no. 8, pp. 1453–1478, 1998.

[84] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, “Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization,” *ACM Transactions on Mathematical Software*, vol. 23, no. 4, pp. 550–560, 1997.

[85] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online; accessed 08/06/2021].

[86] K. Svanberg, "The method of moving asymptotes—a new method for structural optimization," *International Journal for Numerical Methods in Engineering*, vol. 24, no. 2, pp. 359–373, 1987.

[87] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006.

[88] S. Rojas-Labanda and M. Stolpe, "Benchmarking optimization solvers for structural topology optimization," *Structural and Multidisciplinary Optimization*, vol. 52, no. 3, pp. 527–547, 2015.

[89] G. Kennedy and Y. Fu, "Topology Optimization Benchmark Problems for Assessing the Performance of Optimization Algorithms," in *AIAA Scitech 2021 Forum*, American Institute of Aeronautics and Astronautics, January 2021.

[90] T. Buhl, C. Pedersen, and O. Sigmund, "Stiffness Design of Geometrically Nonlinear Structures Using Topology Optimization," *Structural and Multidisciplinary Optimization*, vol. 19, pp. 93–104, 2000.

[91] S. S. Rao, *Engineering Optimization.* John Wiley & Sons, Ltd, 2009.

[92] S. Schlenkrich, A. Walther, N. R. Gauger, and R. Heinrich, "Differentiating fixed point iterations with adol-c: Gradient calculation for fluid dynamics," in *Modeling, Simulation and Optimization of Complex Processes*, (Berlin, Heidelberg), pp. 499–508, Springer Berlin Heidelberg, 2008.

[93] B. Christianson, "Reverse accumulation and attractive fixed points," *Optimization Methods and Software*, vol. 3, no. 4, pp. 311–326, 1994.

[94] A. Griewank and C. Faure, "Piggyback differentiation and optimization," in *Large-Scale PDE-Constrained Optimization*, (Berlin, Heidelberg), pp. 148–164, Springer Berlin Heidelberg, 2003.

[95] M. B. Giles, M. C. Duta, J.-D. Muller, and N. A. Pierce, "Algorithm Developments for Discrete Adjoint Methods," *AIAA Journal*, vol. 41, no. 2, pp. 198–205, 2003.

[96] J. E. Peter and R. P. Dwight, "Numerical sensitivity analysis for aerodynamic optimization: A survey of approaches," *Computers & Fluids*, vol. 39, no. 3, pp. 373–391, 2010.

[97] G. K. Kenway, C. A. Mader, P. He, and J. R. Martins, "Effective adjoint approaches for computational fluid dynamics," *Progress in Aerospace Sciences*, vol. 110, 2019.

[98] E. J. Nielsen, J. Lu, M. A. Park, and D. L. Darmofal, "An implicit, exact dual adjoint solution method for turbulent flows on unstructured grids," *Computers & Fluids*, vol. 33, no. 9, pp. 1131–1155, 2004.

[99] D. J. Mavriplis, "Multigrid Solution of the Discrete Adjoint for Optimization Problems on Unstructured Meshes," *AIAA Journal*, vol. 44, no. 1, pp. 42–50, 2006.

[100] M. Nemec and M. J. Aftosmis, “Adjoint sensitivity computations for an embedded-boundary Cartesian mesh method,” *Journal of Computational Physics*, vol. 227, no. 4, pp. 2724–2742, 2008.

[101] O. Burghardt, N. R. Gauger, P. Gomes, R. Palacios, T. Kattmann, and T. D. Economon, “Coupled Discrete Adjoints for Multiphysics in SU2,” in *AIAA AVIATION 2020 FORUM*, American Institute of Aeronautics and Astronautics, June 2020.

[102] M. B. Giles, “Collected matrix derivative results for forward and reverse mode algorithmic differentiation,” *Lecture Notes in Computational Science and Engineering*, vol. 64 LNCSE, no. 08, pp. 35–44, 2008.

[103] S. Akbarzadeh, J. Hückelheim, and J.-D. Müller, “Consistent treatment of incompletely converged iterative linear solvers in reverse-mode algorithmic differentiation,” *Computational Optimization and Applications*, vol. 77, no. 2, pp. 597–616, 2020.

[104] M. S. Campobasso and M. B. Giles, “Effects of Flow Instabilities on the Linear Analysis of Turbomachinery Aeroelasticity,” *Journal of Propulsion and Power*, vol. 19, no. 2, pp. 250–259, 2003.

[105] S. Xu, D. Radford, M. Meyer, and J.-D. Müller, “Stabilisation of discrete steady adjoint solvers,” *Journal of Computational Physics*, vol. 299, pp. 175–195, 2015.

[106] M. S. Campobasso and M. B. Giles, “Stabilization of a Linear Flow Solver for Turbomachinery Aeroelasticity Using Recursive Projection Method,” *AIAA Journal*, vol. 42, no. 9, pp. 1765–1774, 2004.

[107] E. J. Nielsen and W. K. Anderson, “Aerodynamic Design Optimization on Unstructured Meshes Using the Navier-Stokes Equations,” *AIAA Journal*, vol. 37, no. 11, pp. 1411–1419, 1999.

[108] C. L. Rumsey, J. P. Slotnick, and A. J. Sclafani, “Overview and Summary of the Third AIAA High Lift Prediction Workshop,” *Journal of Aircraft*, vol. 56, no. 2, pp. 621–644, 2019.

[109] R. P. Dwight and J. Brezillon, “Effect of Approximations of the Discrete Adjoint on Gradient-Based Optimization,” *AIAA Journal*, vol. 44, no. 12, pp. 3022–3031, 2006.

[110] P. He, C. A. Mader, J. R. Martins, and K. J. Maki, “An aerodynamic design optimization framework using a discrete adjoint approach with OpenFOAM,” *Computers & Fluids*, vol. 168, pp. 285–303, may 2018.

[111] R. Roth and S. Ulbrich, “A Discrete Adjoint Approach for the Optimization of Unsteady Turbulent Flows,” *Flow, Turbulence and Combustion*, vol. 90, pp. 763–783, jun 2013.

[112] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, second ed., 2008.

[113] E. J. Nielsen and W. L. Kleb, “Efficient Construction of Discrete Adjoint Operators on Unstructured Grids Using Complex Variables,” *AIAA Journal*, vol. 44, pp. 827–836, apr 2006.

[114] C. A. Mader, J. R. R. A. Martins, J. J. Alonso, and E. van der Weide, "ADjoint: An Approach for the Rapid Development of Discrete Adjoint Solvers," *AIAA Journal*, vol. 46, no. 4, pp. 863–873, 2008.

[115] Z. Lyu, G. K. Kenway, C. Paige, and J. R. R. A. Martins, "Automatic Differentiation Adjoint of the Reynolds-Averaged Navier-Stokes Equations with a Turbulence Model," in *21st AIAA Computational Fluid Dynamics Conference*, (San Diego, CA), American Institute of Aeronautics and Astronautics, June 2013.

[116] K. Maute, M. Nikbay, and C. Farhat, "Analytically based sensitivity analysis and optimization of nonlinear aeroelastic systems," in *8th Symposium on Multidisciplinary Analysis and Optimization*, (Long Beach,CA), American Institute of Aeronautics and Astronautics, September 2000.

[117] J. R. Martins, J. J. Alonso, and J. J. Reuther, "High-fidelity aerostructural design optimization of a supersonic business jet," *Journal of Aircraft*, vol. 41, no. 3, pp. 523–530, 2004.

[118] P. Gomes and R. Palacios, "Aerodynamic-driven topology optimization of compliant airfoils," *Structural and Multidisciplinary Optimization*, vol. 62, no. 4, pp. 2117–2130, 2020.

[119] C. Venkatesan-Crome and R. Palacios, "A Discrete Adjoint Solver for Time-Domain Fluid-Structure Interaction Problems with Large Deformations," in *AIAA Scitech 2020 Forum*, American Institute of Aeronautics and Astronautics, January 2020.

[120] A. Yildirim, C. A. Mader, and J. R. R. A. Martins, "Accelerating parallel CFD codes on modern vector processors using blockettes," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, (New York, NY, USA), pp. 1–9, ACM, jul 2021.

[121] A. A. Stepanov and D. E. Rose, *From Mathematics to Generic Programming.* Addison-Wesley Professional, 1st ed., 2014.

[122] A. Fog, *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs.* Technical University of Denmark, 2020.

[123] R. Lohner, *Applied Computational Fluid Dynamics Techniques.* John Wiley & Sons, Ltd, 2008.

[124] I. Hadade, F. Wang, M. Carnevale, and L. di Mare, "Some useful optimisations for unstructured computational fluid dynamics codes on multicore and manycore architectures," *Computer Physics Communications*, vol. 235, pp. 305–323, 2019.

[125] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[126] R. Eberhardt and M. Hoemmen, "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures," *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pp. 663–672, 2016.

[127] J. P. Ecker, R. Berrendorf, and F. Mannuss, "New Efficient General Sparse Matrix Formats for Parallel SpMV Operations," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10417 LNCS, pp. 523–537, 2017.

[128] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, “Sparse Matrix-Vector Multiplication on GPGPUs,” *ACM Transactions on Mathematical Software*, vol. 43, no. 4, pp. 1–49, 2017.

[129] T. D. Economon, F. Palacios, J. J. Alonso, G. Bansal, D. Mudigere, A. Deshpande, A. Heinecke, and M. Smelyanskiy, “Towards High-Performance Optimizations of the Unstructured Open-Source SU2 Suite,” *53rd AIAA Aerospace Sciences Meeting*, no. January, pp. 1–30, 2015.

[130] *Intrinsics Guide.* Intel, Jun 2020. [Online; accessed 08/06/2021].

[131] F. Witherden, A. Farrington, and P. Vincent, “PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach,” *Computer Physics Communications*, vol. 185, no. 11, pp. 3028–3040, 2014.

[132] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” in *Proceedings of the 1969 24th National Conference*, ACM ’69, (New York, NY, USA), p. 157–172, Association for Computing Machinery, 1969.

[133] E. Jeannot, Y. Fournier, and B. Lorendeau, “Experimenting task-based runtimes on a legacy Computational Fluid Dynamics code with unstructured meshes,” *Computers & Fluids*, vol. 173, pp. 51–58, 2018.

[134] J. Blühdorn, M. Sagebaum, and N. R. Gauger, “Event-Based Automatic Differentiation of OpenMP with OpDiLib,” 2021.

[135] M. Benzi and M. Tuma, “A Sparse Approximate Inverse Preconditioner for Nonsymmetric Linear Systems,” *SIAM Journal on Scientific Computing*, vol. 19, no. 3, pp. 968–994, 1998.

[136] E. Chow and A. Patel, “Fine-Grained Parallel Incomplete LU Factorization,” *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015.

[137] *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel, May 2020. [Online; accessed 08/06/2021].

[138] N. Kroll, S. Langer, and A. Schwöppe, “The dlr flow solver tau - status and recent algorithmic developments,” in *52nd Aerospace Sciences Meeting*, (National Harbor, Maryland), 2014.

[139] M. S. Rivers, C. Hunter, and V. N. Vatsa, “Computational fluid dynamics analyses for the high-lift common research model using the USM3D and FUN3D flow solvers,” in *55th AIAA Aerospace Sciences Meeting*, (Grapevine, Texas), 2017.

[140] A. Yildirim, G. K. Kenway, C. A. Mader, and J. R. Martins, “A Jacobian-free approximate Newton–Krylov startup strategy for RANS simulations,” *Journal of Computational Physics*, vol. 397, 2019.

[141] L. Wang, B. Diskin, E. J. Nielsen, and Y. Liu, “Improvements in Iterative Convergence of FUN3D Solutions,” in *AIAA Scitech 2021 Forum*, American Institute of Aeronautics and Astronautics, January 2021.

[142] T.-Y. Chen and S.-C. Wu, “Multiobjective optimal topology design of structures,” *Computational Mechanics*, vol. 21, no. 6, pp. 483–492, 1998.

[143] P. D. L. Jensen, F. Wang, I. Dimino, and O. Sigmund, “Topology optimization of large-scale 3d morphing wing structures,” *Actuators*, vol. 10, no. 9, 2021.