Politecnico di Milano

School of Industrial and Information Engineering
Master's Degree in Mathematical Engineering

Spectral techniques for real-time post-processing and shape optimization

Academic supervisor: Prof. Lorenzo Valdettaro
Industrial supervisor: Dr. Luigi Capone

Massimiliano Leoni
Personal ID 823162

Academic year 2014–2015

Mathematics is the art of giving the same name to different things.

Henri Poincaré

Equipped with his five senses, man explores the universe around him and calls the adventure Science.

Edwin Powell Hubble

Abstract

The present work originates from the internship that I did at Rolls-Royce plc, in Derby (UK) between June and December 2015. The job was concerned with the unifying problem of shape optimization for the flow in an S-duct, a particular type of S-shaped duct. From this starting point two main directions are investigated in this work. The first one, which is more theoretical, is concerned with the spectral analysis of the flow under investigation. Here I show how spectral analysis, which I performed with a software I created (in C++) during the internship, is able to identify the main features, or coherent structures, of the flow and how this can be used to efficiently extract a great deal of useful information. In particular, I describe how spectral analysis can be used to define smart stopping criteria and, more concerned with the original problem, some refined cost functionals to drive a shape optimization procedure. All these uses belong in the category of *real-time post-processing*, the practice of using post-processing techniques while the simulation is still being run, which is a popular topic at the moment of writing. In the second part, which I developed concurrently, I tackle the more practical problem of defining and setting up a fully-fledged shape optimization procedure. Concerning this, I developed a collection of scripts (bash, Python, Lua) and configuration files for various applications that can now interface with each other in an automated way and make up the components of the optimization loop. I was also able to run said loop with a sample configuration and showed that very promising results can be obtained even with very little computational effort. Although less scientific innovation is involved in this second task, it was still of great relevance to the company, which demanded that I spent a good share of my time on it and achieved significant results.

Sommario

Il presente lavoro origina dal tirocinio che ho svolto presso Rolls-Royce plc, a Derby (UK) tra Giugno e Dicembre 2015. Il lavoro è stato focalizzato sull'ampio problema dell'ottimizzazione di forma per il flusso in un S-duct, un particolare tipo di condotto a forma di S. Da questo punto di partenza si sono diramate due direzioni principali su cui ho lavorato. La prima, a carattere più teorico, riguarda l'analisi spettrale del flusso in esame. In questa parte mostro come l'analisi spettrale, che eseguo con un codice da me scritto (in C++) durante il tirocinio, è in grado di identificare le caratteristiche principali, o strutture coerenti, del flusso e come questo può essere usato per estrarre una gran quantità di informazioni utili. In particolare, descrivo come l'analisi spettrale possa essere usata per definire dei criteri d'arresto migliori e, in relazione al problema originale, dei funzionali costo più raffinati per guidare il processo di ottimizzazione di forma. Tutti questi utilizzi fanno parte della categoria del *post-processing in tempo reale*, la pratica di utilizzare tecniche di post-processing mentre la simulazione è ancora in esecuzione, un argomento molto popolare al momento della stesura di questo documento. Nella seconda parte, che ho portato avanti simultaneamente, mi occupo del problema, di carattere più pratico, della definizione e configurazione di una procedura di ottimizzazione di forma completa. Per questo scopo ho creato una collezione di script (in bash, Python e Lua) e file di configurazione per varie applicazioni che adesso possono interfacciarsi l'un l'altra in maniera automatizzata e formano le componenti del ciclo di ottimizzazione. Sono anche stato in grado di eseguire questo ciclo con una configurazione di esempio ed ho mostrato come si possano ottenere risultati molto promettenti anche con poco sforzo computazionale. Sebbene questa seconda parte coinvolga meno innovazione scientifica, è stata comunque di grande rilevanza per la compagnia, la quale ha richiesto che ci dedicassi buona parte del mio tempo, raggiungendo risultati significativi.

Acknowledgements

All journeys come to an end, eventually. All of them, thus including this. Now that I am writing the last chapter of this thesis, and I am so close to the completion of the work, I cannot keep myself from looking over my shoulder at all the people who pushed me to get here, for as far as I have gone, I have to admit I do not think I could have possibly made it without them. I am happy the list is quite long so, my dear reader, take your time.

I first need to thank Luigi Capone, who has been a mentor, other than just an industrial supervisor. From our talking I learnt many invaluable things I could never find on textbooks on the rules of the new world I have first been exposed to under your guidance.

I also need to express my gratitude to my academic supervisor Professor Lorenzo Valdettaro. I much appreciate the great help and the prompt support I received, despite the obvious geographical obstacles, especially in last month's final rally.

I strongly wish to acknowledge the endless support of my parents. Not only on the financial side, without whom I would have been lost, but also, and especially, for supporting me in whatever adventure I decide to embark upon. Knowing that moving to another country will not make you completely alone is a great boost.

Not completely alone, actually, for lately I never was. Miriam took *great* care in ensuring that I never felt alone during my stay, not a single day. I wish to thank you for being part of my journey, and a particularly spicy one; perhaps inadvertently, you taught me a lot about love that I simply conjectured I knew already, and you definitely proved Ovid right in saying that *love is a kind of warfare.*

Not completely alone, as I was saying, for on the other side of the Channel there was a friend already waiting for me, that managed to make everything much less rough than it could have been, despite the inevitable cultural clash. Thank you, Josh, for your priceless help in moving in and adapting to the new culture.

As always, I am deeply indebted to David for being the wonderful master, advisor and, more importantly, friend that he is. Wherever you are and whatever you are doing, I know that I can always disturb you

with some weird doubt about some weird topic, and I am grateful for that.

A big, comprehensive THANK YOU goes to the many friends that I have met in the years and have been walking alongside me: Francesco, for having been very close all the way down the course of study; Roberto, for always providing a valuable, different point of view; Marco, for the warm welcome in his circle; quickly cascading to Francesca, Giulio, Alì, Francesca, Luca, Pasquale, Francesco, Lucio, Luca, Luca, Camillo and all the ones I forgot.

Wow, now *that*'s quite a list.

Contents

Chapter 1

An extended introduction

In this chapter we present the background material we will refer to throughout the rest of the work.

1.1 Large Eddy Simulations

With the name Large Eddy Simulations (LES) we refer to a turbulence model for fluid dynamics simulations. This method is relatively new and in recent years its popularity has been growing in the realm of industrial applications, as it earlier did in the academic one. Several monographs are available on this method, which is used both in compressible and incompressible fluid flows, see for example Pope [Pop00], Garnier et al. [GAS09], and Sagaut [Sag06].

The idea behind an LES is to fully resolve the coarser flow scales as if it were a Direct Numerical Simulation (DNS) and model only the finer ones, introducing the so-called *subgrid stress tensor*. This is the result of a filtering operation as detailed in the following.

We define the filtering $\bar{f}$ of a function $f(\mathbf{x}, t)$ on a domain E as

$$\bar{f}(\mathbf{x}, t) = \int_E f(\mathbf{x} - \mathbf{r}, t) G(\mathbf{r}, \mathbf{x}) \, d\mathbf{r}. \tag{1.1}$$

The function G is called the filter and has unit integral norm. The difference between the original function and the filtered function is called *residual variable* and denoted by $f' := f - \bar{f}$.

Let us now turn our attention to the filtering of the Navier-Stokes equations: starting from the regular continuity and momentum equations

$$\begin{aligned} \nabla \cdot \mathbf{u} &= 0 \\ \partial_t \mathbf{u} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) &= -\nabla p + 2\nu \nabla \cdot S(\mathbf{u}), \end{aligned} \tag{1.2}$$

where $S(\mathbf{u}) = \operatorname{sym} \nabla \mathbf{u} = \frac{\nabla \mathbf{u} + \nabla \mathbf{u}^T}{2}$ is the symmetric part of the gradient of

$\mathbf{u}$, we apply a homogeneous filter*, which has the property that its filter (the integral kernel) does not depend on $\mathbf{x}$, to get

$$\begin{aligned} \overline{\nabla\cdot\mathbf{u}} &= 0 \\ \nabla\cdot\overline{\mathbf{u}'} &= 0 \\ \partial_t\overline{\mathbf{u}} + \nabla\cdot(\overline{\mathbf{u}}\otimes\overline{\mathbf{u}}) &= -\nabla\overline{p} + 2\nu\nabla\cdot\overline{S(\mathbf{u})} - \nabla\cdot(\tau). \end{aligned} \tag{1.3}$$

The term τ, of elements $\tau_{ij} = \nabla\cdot(\overline{u_i u_j} - \overline{u_i}\ \overline{u_j})$, is the subgrid stress tensor. The various LES models differ in how they model this tensor. Many models can be found in the literature, ranging from turbulent viscosity models, where the modeled scales are assumed to simply increase the flux viscosity, to the more complex dynamical models.

Large Eddy Simulations are very different from more classic Reynolds-Averaged Navier-Stokes (RANS) simulations. In the latter, the flow variable is decomposed in its mean and the associate fluctuation $\mathbf{u} = \overline{\mathbf{u}} + (\mathbf{u} - \overline{\mathbf{u}}) = \mathbf{u} + \mathbf{u}'$, resulting is the terms $\{(\mathbf{u}'_i\mathbf{u}'_j)\}$, $i,j = 1,2,3$ of a symmetric tensor to be modeled. In a RANS simulation, the whole flow is influenced by the modelling and what one eventually computes is the mean flow. For this reason, RANS simulations can be done on a reasonably coarse grid, whereas Large Eddy Simulations need finer grids to be able to resolve the non modeled flow scales. Large Eddy Simulations are also necessarily time dependent, whereas a RANS simulation can easily be formulated for statistically steady state problems. The latest two statements suggest that the computational effort required to perform an LES is significantly higher than that for a RANS simulation; this is indeed the case and one of the reasons why Large Eddy Simulations are shyly appearing in industrial applications, now that supercomputing power is more easily available than in a not too far past.

1.2 Shape optimisation

Shape optimisation is a topic of central importance in a wide range of applications of industrial relevance. The concept itself of being able to choose the best shape for something should suggest, even to the unaware reader, that uncountable scenarios can benefit from the mastery of this technique.

In the framework of Computational Fluid Dynamics (CFD), the need to design a system in a way that can control some output quantities is ubiquitous: from drag-minimising airfoils to more efficient turbine blades, component designers are always looking for a shape that is somehow *better* that the others.

*This implies we can commute filtering and differentiation.

The mathematical formulation of the problem of shape optimisation is the following: one could think of a shape optimisation problem as the problem of finding the minimum (or maximum) of a given functional $J(\Omega)$ over a set $\mathcal{U}_{ad}$ of the admissible shapes, or

$$\min_{\Omega \in \mathcal{U}_{ad}} J(\Omega).$$

What makes this problem challenging is that the dependence of J on Ω is indirect, in the sense that most commonly $J(\Omega) = \tilde{J}(u(\Omega))$, where u is the solution of some Partial Differential Equation (PDE) at hand. Think, for example, of Ω as being the shape of an airfoil, J the drag on the airfoil, and u the velocity and pressure field *on the given shape* Ω. It is clear that changing Ω changes u, and thus one is required to solve again the PDE that defines u, adding to the computational cost of the whole procedure. On top of this, the set $\mathcal{U}_{ad}$ of admissible shapes must be chosen in a way that the underlying PDE problem is still well defined.

Due to the computational cost and the intrinsic difficulty of mathematically defining the set $\mathcal{U}_{ad}$, a classic approach in shape optimisation for engineering applications was based on describing the computational domain by means of a small set of parameters and then recasting the shape optimisation problem as a multivariable real optimisation problem. [Sha00; SSG14; SC14].

More recently, advances in the mathematical theory of PDEs gave rise to a new flavour of shape optimisation, based on the treatment of the domain in a topological way, and introducing new objects like the shape derivative. [SZ92; NS13]. This new method is object of active research at the moment, but has not yet gained much popularity in industrial applications.

1.3 Post-processing techniques and spectral analysis

With the growth of computing power, bigger and bigger problems were attacked, leading to ever growing amounts of data produced by numerical simulations. Soon, the need to extrapolate *meaningful* information from the results rose, and a variety of techniques were developed to meet it. The collection of the theory and methods that address this issue is known as *post-processing*.

A field in which post-processing techniques grew quite popular is that of CFD, for example with respect to the problem of the identification of coherent structures in a flow field. A fairly comprehensive review of the most common post-processing approaches employed to tackle this problem can be found in von Terzi et al. [vTSF09].

1.3.1 Proper Orthogonal Decomposition

The Proper Orthogonal Decomposition (POD) was first introduced by Kosambi [Kos43] (see also [Nar11]) and later spread out in a variety of fields ranging from mathematics to statistics and a number of areas in engineering, where it also acquired several new names, becoming known also as Principal Component Analysis or Karhunen-Loève expansion, for example. A very good introduction to the POD can be found in Chatterjee [Cha00]

POD is a classic technique in the post-processing toolbox, and in fact was used extensively over the years. Moreover, although it can be (independently) interpreted as a pure linear algebra thing, it is known to have a meaningful physical interpretation: see for example Kerschen et al. [KG02] for an interpretation in structural mechanics.

A well known use of POD is for the purpose of model reduction of a system, as reported in Kerschen et al. [Ker+05], but other applications, with focus on the field of CFD, is in the analysis of turbulent flows by means of the extraction of the underlying coherent structures. [BHL93; Hol+12; MEH12].

The POD has a handful of interesting properties. First of all, as will be detailed in Chapter 5, its mathematical formulation is always well posed, so there is the guarantee that a result will be provided by the implementing procedure. This is a robustness property very valuable in many fields of Engineering.

Secondly, the method is parameter-free, so no tuning of constants whatsoever is required prior to its usage. This allows the method to be applied in a wide variety of situations without the need for sensitivity studies that typically are problem-dependent.

Thirdly, the results yielded by the POD are always real-valued fields, typically with a straightforward physical interpretation.

All these properties accounted for the huge success POD had in applications.

1.3.2 Dynamic Mode Decomposition

The Dynamic Mode Decomposition (DMD) is a fairly recent spectral decomposition technique. It was first introduced by Schmid [Sch10] as a method to extract coherent structures from fluid flows, and later successfully applied in separate works, such as Schmid et al. [Sch+11], Kalghatgi et al. [KA14], and Muld et al. [MEH12].

The introduction of this method could not avoid a comparison with the already popular POD technique. As we will discuss later, in Chapter 5, DMD is a computationally cheaper technique than POD, but this comes with a price, which is that of tunable parameters.

These parameters are, in general, problem-dependent, and thus DMD requires a certain *a priori* knowledge of the solution of the problem at hand, and this makes it non optimal for use in a first study of a new phenomenon. Nevertheless, when the main features of the solution are qualitatively known, as it often happens in fluid dynamics, the technique can be effectively implemented and saves computational cost. This is particularly helpful when the procedure is implemented as part of a loop that has to be run several times, as the computational cost easily adds up.

1.4 Graphical Processing Units

Graphical Processing Units (GPUs) are special kinds of processor that have been around for a long time, but only recently were employed for large scale parallel computation.

GPUs belong in the wider class of *coprocessors*. Inside a computer, a coprocessor is a piece of hardware separated from the main, traditional processor that is specialised in doing simple, independent operations as fast as possible. Loosely speaking, a processor is made of a single processing unit able to perform a wide range of different tasks while guaranteeing good performance for all of them. A coprocessor, on the contrary, is a cluster of many processing units that are only good at performing basic computation, but that can do so all together, in parallel execution.

The reason for this huge design difference between processors and coprocessors lies in the different aims for which they were developed, the latter being thought, initially, mainly for computer graphics.

Computer graphics, indeed, is mainly concerned with updating the pixels of a monitor. The resolution of common monitors is constantly increasing, with `4k` monitors being the new standard prescribing $4096 \times 2160 = 8\,847\,360$ pixels to be updated, in the case of a standard 30 fps application, 30 times per second. On the other hand, updating the colour of a pixel requires little effort in general, so it was natural to focus the development of GPUs towards the best design to address this *specific* kind of issue, namely having a lot of relatively slow processing units, all taking care of a small part of the whole picture.

For this reason, initially coprocessors were only used for computer graphics, mainly for gaming, and the class of coprocessors only contained GPUs.

Due to the later crisis in the development of traditional processors, which reached the limit of affordable clock speed, traditional processors began to incorporate multiple processing units – or *cores* – in a single component, seeking better performance in parallel execution. This lead a large part of the computer science community to design new algorithms to better exploit the possibility of executing multiple tasks concurrently.

Computational scientists, who had the most to earn from an increase in computing power, fostered the research on this topic, bringing a lot of attention to the development of massively parallel processors, which otherwise might have not grown so popular. When even the limit on the number of cores that can coexist on a single machine[†] was reached, hardware designers started to closely tie several independent computers – or *nodes* – one to each other, giving to each as many cores as possible, but this came with additional costs and problems in the binding of different machines[‡].

At some point, the idea was born that the thirst for parallel computing power computational scientists had could be quenched by a clever usage of the well known coprocessors that were so spread in computer graphics. This is when General Programming for GPUs (GPGPU) was born.

The first GPU producer to offer a dedicated tool for GPGPU was NVidia, with the famous CUDA toolkit. Since it turns out that there are quite a few fields of science in which typical computational tasks involve the execution of simple, independent operations, GPUs were a breakthrough in those fields, achieving immense success in a handful of notable case studies ranging from bioinformatics to computational chemistry, data science, machine learning and medical imaging, just to name a few.

Coprocessors later tried to gain space in the territory that traditionally belonged to processors, and the dispute is still open, with no clear winner. What is certain is that coprocessors are now an important part of scientific computing worldwide, and in fact several other companies developed coprocessors specifically for scientific computing, most notably Intel's Xeon Phi. New development tools were proposed to address the use of coprocessors in scientific computation, such as OpenCL and OpenACC, and existing ones were expanded to support this emerging architecture, such as OpenMP.

1.5 OP2

Code performance is a matter of two components: software and hardware. As long as the hardware involves single processors, perfectly-written code can be faster only by running on more performing hardware: little or no intervention is required on the application source code[§]. With the spread of multi-core processors, shared memory and distributed memory architectures, coprocessors of various kinds, however, improving the performance of an application is now also a matter of writing *better* code.

[†]This is called a *shared memory* architecture.

[‡]This is called a *distributed memory* architecture.

[§]One exception being manual vectorisation, but not for long.

This can potentially involve a substantial re-writing of the code, depending on the available hardware; if, later, the application is to be executed on a significantly different hardware, optimal performance might require another modification of the source code, and so on.

The process of adapting code to a specific architecture is demanding and requires a good deal of knowledge that belongs to the computer science field. In a scenario in which more and more scientists – biologist, chemists, mathematicians, physicists and others sharing the fact of *not* being computer scientists – approach the world of scientific simulations, it becomes unlikely, and undesirable, that a general scientist has to master *also* a branch of computer science in order to achieve good performance with his simulations, to him merely a tool.

One attempt to meet the need for a somewhat automated parallel code generation was made with the Oxford Parallel Library for Unstructured Solvers (OPlus). Little literature is available on this library as it was developed for industrial, hence confidential, purposes. Two useful references are Burgess et al. [BCG95] and Crumpton et al. [CG95]. OPlus later evolved into a second version, still developed at the Oxford e-Research Centre, named OP2. This project is more academic and open source ([GRM]).

The idea behind the two libraries is similar, so we will only describe OP2 in the following. They move from the assumption that basic entities in a computer simulation can be described by means of sets and maps between these sets, and that the most computationally intensive parts of a simulation involve looping on these sets of objects, computing quantities related to the single set element in an independent way.

More specifically, two assumptions are made, namely that the order in which operations are performed does not affect the final result and that mappings between sets do not change over time.

Actually, a great variety of scientific settings fall in this framework. Unstructured grids are the main target, where set elements can be cells, edges or nodes, and the loops represent, for example, flux computations of explicit timestepping. Other use cases can include n-body simulations, lattice Boltzmann methods, looping on graphs and many others.

With the above assumption, the application user – i.e. the scientist – is only required to write a serial application and then to *wrap* the looping functions in the code – the *kernels* – with some functions provided by OP2. A source-to-source compiler provided by the library is then run that reads the application code written by the user and *generates* new source code, specialised for the required architectures.

A wider literature was produced as the development carried on and OP2 proved able to tackle industrial applications with a very good scaling, for example Bertolli et al. [Ber+12] and Mudalige et al. [Mud+12].

A major advantage of OP2 is that most of the burden of recasting a

given serial code for parallel execution is handled by the library developer, not by the library user. This implies that, on the paper, a library user, who has little knowledge of the details of parallel computation, can benefit of state-of-the art techniques in his code by simply updating the library to the latest version and recompiling all his software. New architectures can also be supported as soon as the library supports it, not the particular code.

Another notable benefit is that using OP2, unlike what happens in the development process of common parallel codes, one can easily maintain the serial version, the shared memory version and the coprocessor version of the code in a *single* listing, not needing to keep multiple implementations around. Maintaining several versions of the same application can make development messy and delay the availability of newly implemented features, which typically appear in the serial code as the field scientist advances his research, in the parallel versions.

1.6 Plan of the work

The rest of this thesis is organized as follows.

- In Chapter 2 we briefly recall the fundamental setting for the simulations we performed, we state the industrial problem we are concerned with and outline the solution methods.
- In Chapter 3 we review OP2, a library developed at Oxford University for efficient parallel computation, and assess its usability to achieve the goals of this project; this topic has been of considerable relevance during the internship.
- In Chapter 4 we describe our first results concerned with mesh generation and simulations with the Reynolds-Averaged Navier-Stokes and Large Eddy Simulations models.
- In Chapter 5 we thoroughly describe the problem of the identification of coherent structures and the spectral analysis approach to its solution, including the details of the techniques we used. We then proceed to review the application we developed to perform the actual computation, describe validation test cases, discuss performances and limitations and present the main results of this work.
- Chapter 6 is dedicated to the topic of shape optimization, reporting the general software framework we developed to perform automated optimization for the problem at hand; we present promising results.
- In the final chapter we conclude with a summary of the main achievements of this work and some directions for future research.

Chapter 2

Computational Fluid Dynamics

CFD is nowadays an essential tool in almost any engineering field, most notably aerospace, mechanical and civil engineering. In the many years of its history, Computational Fluid Dynamics made its way to a variety of other fields such as climate modeling and prediction, ocean modeling, geology, astrophysics, biology and many others. In many cases a symbiotic growth characterized these relationships, with CFD helping in the solution of physical problems, whose solutions in turn helped us understanding more about the complicated nature of fluids.

The benefits of being able to effectively predict the behavior of a physical system *before* actually building it cannot be understated. The time, energy and money expenses needed to design a particular device, structure, airfoil, engine part, prosthesis, etcetera can be drastically reduced if the process is driven by a solid simulation framework.

Given its wide range of applicability, CFD naturally branched into several, more specialised sub-fields, devoted to the study of, for example, compressible or incompressible flows, single phase or multiple phase flows, fluid-structure interaction, Newtonian fluids, non-Newtonian fluids, and so on. Within each of them, field-specific techniques were developed to take advantage of the additional assumptions and the similarity of the attacked problems.

2.1 Compressible fluid dynamics

The framework in which we settle for our problem is that of compressible fluid dynamics.

Here we recall the derivation of the general equations describing fluid flows, including the flow in an S-duct we are interested into, that is, the compressible Navier-Stokes equations.

2.1.1 Mass conservation

Given a density field ρ, the mass contained in a volume V at time t is $\int_{V(t)} \rho$. By mass conservation, we know that

$$d_t \int_{V(t)} \rho = 0;$$

by Reynolds transport theorem we can take the time derivative inside the integral

$$\int_{V(t)} \partial_t \rho + \nabla \cdot (\rho \mathbf{u}) \, d\mathbf{x} = 0.$$

This can be recast to a differential relation because it has to hold valid for any control volume $V(t)$, thus implying that the integrand must vanish, or

$$\partial_t \rho + \nabla \cdot (\rho \mathbf{u}) = 0. \tag{2.1}$$

Equation (2.1) is called the *continuity equation* and is the first of the Navier-Stokes equations.

2.1.2 Momentum equation

We start again from continuum mechanics, from which we know the integral relation for the momentum,

$$d_t \int_{V(t)} \rho \mathbf{u} = \int_{V(t)} \rho \mathbf{f} + \int_{\partial V(t)} \tau \cdot \mathbf{n},$$

which states that the time variation of the momentum is driven by the sum of volume forces and contact forces.

The second order tensor τ is the stress tensor for the continuum at hand. In the case of viscous fluids, it has the form

$$\tau = 2\mu S(\mathbf{u}) + \lambda \operatorname{tr} S(\mathbf{u}) \; I - pI,$$

where I is the identity tensor and p is the pressure.

The differential form of the momentum equation is

$$\partial_t (\rho \mathbf{u}) + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u} - \tau) = \rho \mathbf{f}. \tag{2.2}$$

Equation (2.2) is the second of the Navier-Stokes equations, and is known as *momentum equation*.

2.1.3 Equations closure

In the case of compressible fluids, these four equations (one for the density ρ and three for the components of the velocity $\mathbf{u}$) are not enough to determine the five unknowns $\mathbf{u}, \rho, p$. This problem does not pose in the incompressible flow case as density cancels out and its equation becomes one for $\mathbf{u}$, thus leaving us with four equations in the four unknowns $\mathbf{u}, p$.

In the general case, the energy conservation equation and the equation of state are provided. The former reads, in integral form,

$$d_t \int_{V(t)} \rho E + \int_{\partial V(t)} \mathbf{q} \cdot \mathbf{n} = \int_{\partial V(t)} \tau \cdot \mathbf{n} \cdot \mathbf{u} + \int_{V(t)} \rho \mathbf{f} \cdot \mathbf{u} + \int_{V(t)} \rho r,$$

where $E = e + \frac{1}{2}\rho|\mathbf{u}|^2$ is the total specific energy, e is the internal energy, $\mathbf{q}$ is the specific heat flux, $\mathbf{f}$ is the specific volume force, r is the specific heat production.

The vector $\mathbf{q}$ can be expressed as $\mathbf{q} = -k\nabla\vartheta$ if we accept Fourier's law, having denoted by ϑ the temperature and by k the thermal conductivity. Again by Reynolds transport theorem, we have the differential form

$$\partial_t(\rho E) + \nabla\cdot(\rho E\mathbf{u} - \tau\mathbf{u} - k\nabla\vartheta) = \rho(\mathbf{f}\cdot\mathbf{u} + r). \tag{2.3}$$

The last thing that is needed is the equation of state, which is a relationship between the thermodynamic quantities. In the case of an ideal gas, we have

$$p = (\gamma - 1)\rho e, \quad \vartheta = \frac{e}{c_v}, \quad \gamma = \frac{c_p}{c_v}.$$

The equations introduced so far are sufficient to completely describe the evolution of the system. In the special cases of barotropic flows, for which a relationship between pressure and density not involving energy is available, the equation of energy is not needed anymore, and the system simplifies. One notable such case, which we used for the simulations throughout this work, is that of isentropic gases, for which we have the relationship

$$\frac{p}{\rho^\gamma} = \text{const.}$$

For the simulations in the present work, the ideal gas model with the isentropic flow hypothesis was always assumed, with $\gamma = 1.4$.

2.2 The Rolls-Royce software HYDRA

Given the central importance of CFD, many solvers exist, either meant to be general purpose or specific for a narrow range of applications.

Rolls-Royce devoted significant effort in the past years to the development of an in-house CFD code, named HYDRA. HYDRA is a finite volume solver specialised in solving the compressible Navier-Sokes equations.

On the model side, it supports several turbulence models, including inviscid, laminar, Spalart-Allmaras to k−ε, Mentor's k−ω SST, k−ω with EARSM, LES and DES, many of which with a specific version supporting wall functions as well.

The default solving strategy, and by far the most widely, implemented in HYDRA, is a parallel multigrid smoothing technique. A lot of freedom is left to the user, who can specify either a V-cycle or a W-cycle type of multigrid loop, with an arbitrary number of levels. A characteristic trait of the solver is that the multigrid algorithm is not purely algebraic, but the coarser grids are actually built by an edge collapse procedure carried out by an auxiliary preprocessor.

The management of parallel execution in HYDRA is delegated to the OPlus library introduced above. In this way, the implementation of CFD related code is separated by that of computer science related code (i.e. parallelisation). [BCG95; CG95]. This is the reason why we are interested in the OP2 library: it is tightly connected to the HYDRA solver which, in a future release, could be adapted to use this more modern version.

Most of the simulations done throughout the six months of this work were performed with HYDRA. Occasionally, ANSYS Fluent was used for preliminary investigations, comparing configurations and for some small, complementary checks.

2.3 Statement of the problem

The study of aerodynamics, and of aeronautical engineering in general, is certainly more challenging when tackling bleeding-edge conditions. The most notable case of extreme condition is aerodynamics is certainly that of fighter planes.

Since their first appearance in World War II, it became clear that fighters were going to play a decisive role in almost all kinds of conflicts, given their superior velocity and ability to strike both air and land targets with high precision and a variety of weapons, including, much later, nuclear missiles. Also, the development of carriers further increased their strategic importance, making it feasible to operate and refuel fighters without the need of a physical, fully-featured airport.

For all these reasons, fighters always received great attention, and a huge share of the technology currently available to the public was initially developed with military applications in mind, in particular to fighters.

Figure 2.1: Lockheed Martin-Boeing F-22 Raptor. The hole under the cockpit is the intake of an S-duct leading to the engine in the rear part. The hole and the engine are not aligned, so the duct needs to be S-shaped.

The industrial problem we address in this work is that of the optimal design of an important component of fighters, known as *S-duct*.

On a fighter, an S-duct is any of the (typically one or two) ducts that extend from the front of the plane to the engine intake in the rear part of the plane. The function of an S-duct is clearly that of taking the air necessary for the correct functioning of the engine to the engine itself; see Figure 2.1 for an example.

Design constraints impose that the inlet of an S-duct cannot, in most cases, be aligned with that of the engine; i.e., the duct cannot be straight but has to do at least – and typically at most – two bends, hence the name. This is due to the fact that, in order to preserve the stealth capabilities of the fighter plane, the engine blades should not be visible from outside the plane, as would happen with a straight duct.

Unfortunately, the S-shaped design typically induces separation and recirculation on the first bend of the duct. This has the undesirable consequence of yielding a spatially inhomogeneous pressure outlet, which coincides with the inlet to the engine. An inhomogeneous pressure inlet to the engine can reduce stability margins of the propulsion system significantly, possibly resulting in reduced fan life and performance. It is therefore of utmost importance that, together with optimal performance, stable propulsion be ensured in the whole flight envelope of a jet aircraft.

The question arises naturally if anything can be done to reduce the effect of the separation; in other words, if the inhomogeneous outflow condition can be somehow recovered.

2.4 Methods

To answer the above question, a designer has to act on the S-duct under examination, the most relevant property of which is its shape. A designer then starts wondering if any two given shapes are equal, with respect to the outflow pressure field they generate. Once he is convinced that the answer is negative, the question poses of how to tell which of two shapes is better. This takes us to the problem of assessing the quality of a given outflow.

A main strategy is that of using the so-called distortion indexes. A distortion index is, basically, a functional of a given flow that aims to quantify the degree of distortion of a flow field. Distortion indexes have been extensively and successfully used in the past; for example, an index called $DC\vartheta$ was used in the European fighter program Eurofighter [BJ00; BB10]. This index evaluates the distortion level of an outlet from the worst ϑ-degrees-wide sector in the outlet, according to

$$DC\vartheta = \frac{\overline{p_{t,out}} - \overline{p_{t,\vartheta}}}{q^2},$$

where $\overline{p_{t,out}}$ is the mean total pressure at the outlet, $\overline{p_{t,\vartheta}}$ is the mean total pressure in the worst ϑ-degrees-wide sector and q is the dynamic pressure at the outlet. In a typical application, $\vartheta = 60$ degrees, at least.

In the present study, as far as distortion indexes are concerned, we used two, more evolved, distortion indexes, named Circumferential Distortion Index (CDI) and Radial Distortion Index (RDI). These indexes try to independently assess the quality of the outflow field in the radial and circumferential components.

To define the distortion indexes, let us consider a circular outlet section. These indexes were introduced for experimental settings, so we will here define them in that context. Consider a rack of sensors positioned in the outflow surface along concentric circumferences C_i, as depicted in Figure 2.2.

To define the CDI, on every circumference C_i we compute a local index

$$CDI_i = \frac{\overline{p_{t,i}} - \min p_{t,i}}{\overline{p_{t,out}}} F_{\vartheta i},$$

where we called $p_{t,i}$ the pressure field defined by sensors on C_i and $F_{\vartheta i}$ an empirical weighing factor.

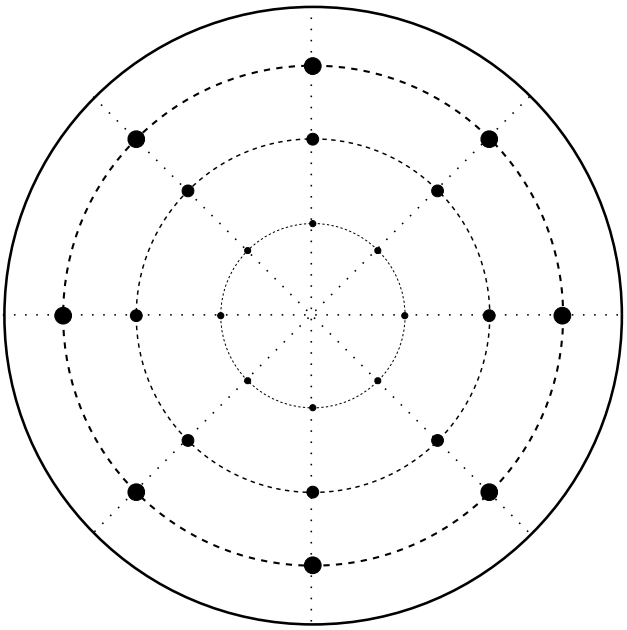

Figure 2.2: A rake of sensors along concentric circumferences: the solid line delimits the outflow section, the filled dots represent the sensors.

With the above numbers, we can finally define the CDI as

$$\mathrm{CDI} = \max_{i} \frac{\mathrm{CDI}_i + \mathrm{CDI}_{i+1}}{2}. \tag{2.4}$$

The RDI is calculated by considering the hub radial (inner) and the tip radial (outer) ring, defining

$$\mathrm{RDI}_i = 1 - \frac{\overline{p_{t,i}}}{\overline{p_{t,\mathrm{out}}}}, \quad i = \begin{cases} \mathrm{inner} \\ \mathrm{outer} \end{cases}$$

and later

$$\mathrm{RDI} = \max(\mathrm{RDI}_{\mathrm{inner}}, \mathrm{RDI}_{\mathrm{outer}}). \tag{2.5}$$

An alternative to understand how bad the situation is in the separation zone is doing a spectral analysis. Spectral analysis comprises more innovative techniques that aim at extracting meaningful information from an unsteady flow field. In general, a spectral analysis technique tries to decompose the given space-time flow field into wisely chosen fundamental components, the most important of which are meant to contain the most relevant information. We will expand on spectral analysis below.

Chapter 3

GPUs and OP2

3.1 More details on GPU architectures

We introduced in Section 1.4 that coprocessors, and especially GPUs, grew popular in scientific computing. We shall now expand on the reason why they are so successful in some applications of scientific computing, with relations to the hardware structure of the GPUs.

The main difference between processors and coprocessors is in fact in their hardware design, and it is sketched in Figure 3.1. A CPU is designed to be able to perform a number of tasks of different kinds, and in fact the Arithmetic-Logic Units (ALUs) are important in the design, but not predominant. GPUs, conversely, dedicate very little space to cache and control facilities and commit almost all of their hardware to processing units and the memory to support them. This heavy unbalance in hardware resources allocation means that GPUs can perform significantly better in mere computation, while falling behind on other kinds of tasks. This disadvantage is, however, of little concern as long as coprocessors are used to perform only the specialized tasks they are designed for.

In order to harvest their potential even further, processing units inside modern GPUs are arranged in two-dimensional arrays of blocks, each of

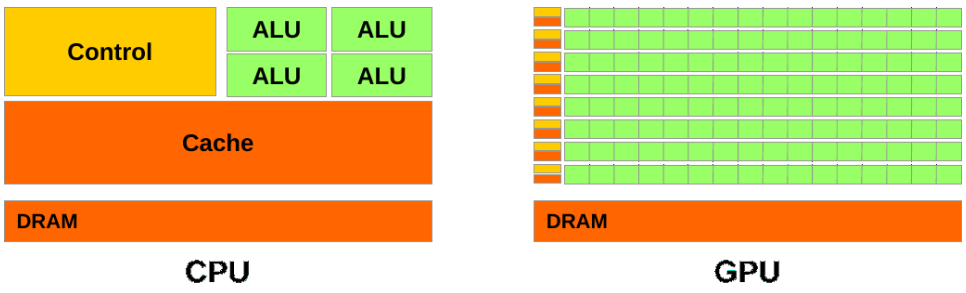

Figure 3.1: Sketch of the designs of CPUs and GPUs.

which contains two-dimensional arrays of threads. This hierarchy has several advantages. On one side, it reflects a variety of configurations which are naturally two-dimensional, such as pixels in an image, entries in a matrix or cells in a structured 2D mesh; this allows the programmer to simplify the development by addressing the single threads, which are appropriately labeled, by their position on the chip, matching virtual and physical positions, which is of great help.

On the other hand, the knowledge that programmers are going to exploit the particular arrangement of the threads allows the maker to design the hardware so that it can take advantage of this information; for example, in this scenario communications between neighbouring threads are more frequent than communications between far threads.

3.2 More details on OP2

The OP2 library, which we introduced in Section 1.5, is a notable example of automatic code parallelisation targeting multiple architectures at once. Several of its advantages were discussed above, here we desire to give a more detailed account of the way it works and it can be used, at least from the user's perspective. Since the principal application of OP2 is in the realm of unstructured simulations, most examples will refer to situations typical of unstructured simulations.

The OP2 library provides an abstraction layer based upon three basic objects, called `sets`, `maps` and `datasets`.

A `set` is any finite, ordered collection of objects that have something in common, similar to the analogous mathematical definition. Notable examples of sets are mesh nodes, edges, vertexes, but they could just as well be nodes in a graph.

A `map` is a relationship between elements of two sets. An interesting characteristic of these maps is that, unlike what happens with mathematical maps, they can associate an element of the first set to several elements of the second set. In other words, they are what in mathematical terms would be called multi-valued maps, or simply maps between a set and the power set of another set. Examples of maps include the mapping of a cell to the neighbouring cells, as well as the mapping of a cell to its edges or vertexes.

Finally, a `dataset` is any field defined on a set, such as density, pressure, or velocity in a cell.

Once the basic terms are defined, the application user is in charge of identifying what entities in his or her code can be classified as sets, how they are mapped to each other and what datasets are defined on them.

At the programming level, for the case of a two-dimensional grid, this is achieved in a way similar to what is shown in Listing 3.1, where we first

```
1  int nElems = nCells + nBdryEdges;
2  op_set s_elems = op_decl_set(nElems,"s_elems");
3  op_set s_cells = op_decl_set(nCells, "s_cells");
4  op_set s_edges = op_decl_set(nBdryEdges, "s_edges");
5  op_set s_vertexes = op_decl_set(nVertexes, "s_vertexes");
6
7  op_map m_ed2vx = op_decl_map(s_edges,s_vertexes,
8                               2,elem2,"m_ed2vx");
9  op_map m_cl2vx = op_decl_map(s_cells,s_vertexes,
10                              3,elem3,"m_cl2vx");
11
12 double* u0 = (double*) malloc(sizeof(double)*nElems);
13 op_dat d_u0 = op_decl_dat(s_elems,1,"double",u0,"d_u0");
```

Listing 3.1: Declaration of basic OP2 elements.

```
1  inline void cellArea(
2          const double* p1,
3          const double* p2,
4          const double* p3,
5          double* area)
6  {
7      *area = 0.5*((p1[0]-p3[0]) * (p2[1]-p1[1])
8                  - (p1[0]-p2[0]) * (p3[1]-p1[1]));
9      if (*area < 0)
10         *area *= -1;
11 }
```

Listing 3.2: Function to compute the area of a cell from the coordinates of the points.

declare sets for elements, cells and edges specifying their size and name, and later define maps between edges (resp. cells) and vertexes, associating to each edge (resp. cell) two (resp. three) vertexes. The variables `elem2` and `elem3` are integer arrays containing the indirections of the mappings. In other words, the array `elem2` contains `2*nBdryEdges` integers, the first two indicating the indexes of the elements of `s_vertexes` to which the first element of the set of edges is mapped, the second two mapping the second edge to the corresponding vertexes, and so on.

Sets and maps are fine, but OP2 unleashes its true power in computation, that is in loops. A common task needed in finite volumes codes is that of computing the area of a given cell, which is needed in the time marching scheme. Given a function to compute the area of a cell as that shown in Listing 3.2, the loop can be written in plain `C` language as shown in Listing 3.3.

```
1 double* mi = (double*) malloc(sizeof(double)*nCells);
2
3 for(int i = 0; i < nCells; i++)
4 {
5     cellArea(double* p1,
6              double* p2,
7              double* p3
8              double* mi[i]):
9 }
```

Listing 3.3: A regular loop to compute the area of a set of cells.

```
1 double* mi = (double*) malloc(sizeof(double)*nCells);
2 op_dat d_mi = op_decl_dat(s_cells,1,"double",mi,"d_mi");
3
4 op_par_loop(cellArea,"cellArea",s_cells,
5        op_arg_dat(d_coords,0,m_cl2vx,3,"double",OP_READ),
6        op_arg_dat(d_coords,1,m_cl2vx,3,"double",OP_READ),
7        op_arg_dat(d_coords,2,m_cl2vx,3,"double",OP_READ),
8        op_arg_dat(d_mi,-1,OP_ID,1,"double",OP_WRITE));
```

Listing 3.4: An OP2 parallel loop to compute the area of a set of cells.

The OP2 parallel version of this loop is quite close to the original one. Indeed, the code simply needs to be wrapped like shown in Listing 3.4. This last listing reads as follows: declare a parallel loop using the kernel (i.e. function) `cellArea`, which loops on the set `s_cells`, and gives a certain list of arguments to it. The argument parameters `op_arg_dat` specify that arguments should be taken from datasets defined to OP2 above in the code. For the case of the first argument, a detailed explanation of the meaning is as follows. Pass as an argument to `cellArea` a value taken from the dataset `d_coords`; but the loop is being defined on the set of cells, `s_cells`, whereas the dataset `d_coords` is defined on the set of vertexes, `s_vertexes`, thus pass the value from `d_coords` taken from the zeroth* element the current cell is mapped to by the map `m_cl2vx`. This argument consists of three doubles, and is intended to be read and not written by the function - the latest specification is important for optimisation purposes.

At this point, an important thing to note is that the code, even though wrapped like this, can still be executed serially, as if it did not include any OP2 function. In fact, OP2 is completely transparent in the original source code and functions like `op_par_loop` simply bind to the regular loop when the original code is executed. This way, the library user can continue the development of his scientific code without potentially have

*Which is the first, as `C` is a zero-based language.

```
1  new execution plan #2 for kernel cellArea
2  number of blocks       = 121
3  number of block colors = 1
4  maximum block size     = 640
5  average thread colors  = 1.00
6  shared memory required =  28.62 KB
7  average data reuse     = 1.91
8  data transfer (used)   = 4.86 MB
9  data transfer (total)  = 7.81 MB
10 SoA/AoS transfer ratio = 1.37
```

Listing 3.5: OP2 output for the execution plan of a kernel.

to debug a parallel implementation, or propagate new features to it.

When the time comes for the library user to deploy his application on parallel architectures, another part of OP2 comes into play, that is the OP2 translator.

The translator is a Python script[†] that acts as a source-to-source compiler, meaning that it reads the serial source code that was wrapped with the special OP2 functions and generates a set of new source codes ready to be compiled. In particular, OP2 is able to automatically generate source code optimised for use with OpenMP, MPI, CUDA, as well as hybrid architectures such as OpenMP+MPI and OpenMP+CUDA. All MPI versions support interfacing to ParMETIS or PT-SCOTCH to efficiently partition and redistribute the workload among all available nodes [KK98; CP08].

Implementations for other architectures are work in progress and more or less stable, such as the support for auto-vectorisation of the code, aimed at exploiting processors' features like Intel's SSE or AVX. Once again, a library user will be able to benefit from these new features without changing his source code.

The code generated by OP2 is highly optimised, taking advantage of the field-specific expertise of the developers. What happens behind the scenes is that OP2 builds separate data structures that are optimised for the specific target architecture. It later uses a scheduler to plan the execution of the loop iterations; a sample output can be seen in Listing 3.5. Although very promising, OP2 has a few flaws and limitations, which we will briefly describe. The inner core of OP2 works on two assumptions.

The first of these is the hypothesis that the result of parallel loops does not depend on the order in which the single iterations are executed. This is often the case, for example when dealing with explicit time-stepping, or when computing functionals of a solution like its mean, or again when populating the dataset of the areas of the mesh cells.

[†]A matlab version is also available.

The second assumption is that mapping indirections are only one level deep. This means that if a mapping from cells to their neighbours is given, together with a mapping from cells to their vertices, in looping on the set of cells it would be possible to access datasets defined on the cells, on the neighbouring cells or on a cell's vertexes, but not on the vertexes of neighbouring cells.

In the writer's experience, this can become limiting when implementing a design pattern he came up with, friendly called *injection pattern*. In a common application framework, we may define *element* any mesh entity on which fields are defined (solution, forcing, coefficients, etcetera).

In finite volumes methods, the set of *elements* typically includes[‡] both cells and boundary segments, which are useful to impose boundary conditions. This means that the datasets corresponding to the above fields should be defined on this set of elements. This data setting agrees with what is done, for example, by ANSYS's meshing tool ICEM.

When looping on the cells of a mesh, though, one is seldom interested in looping on both internal and boundary elements together, the former group usually needed to update the solution, the latter group to update the boundary conditions. Since the operations to do for each of them are very different, one possibility would be to loop on all elements and then use an `if` statement inside the loop to distinguish if the element is in internal cell or a boundary edge. This turns out to be extremely inefficient when parallel scalability is taken into account.

For this reason, it is convenient to define, together with the set of elements, additional auxiliary sets for cells and boundary edges, and then declare *injections* of these auxiliary sets into that of elements, *de facto* implementing the mathematical idea of a subset. This enables to easily loop on the boundary edges or on the internal cells separately, increasing performance, and allowing to access datasets through the injection into the set of elements. An example of the implementation of this design pattern is shown in Listing 3.6, where the definitions of Listing 3.1 are assumed.

On the other hand, OP2's limitation makes it now impossible to access datasets defined, for example, on a neighbour of the element corresponding to a given cell.

Another notable limitation in the use of OP2 is its intrinsic unfitness to provide linear algebra capabilities. This follows directly from the nature of unstructured computation framework and the fact that recasting matrix-vector operations in the frame of looping over general sets, although certainly possible, would make the whole overhead of managing structured objects (matrices and vectors) as if they were unstructured, not worth anymore. The best solution is to delegate linear algebra needs to

[‡]In the case of a 2D grid.

```
1  int* ed2el = (int*) malloc(sizeof(int)*nBdryEdges);
2  for (int i = 0; i < nBdryEdges; i++)
3      ed2el[i] = i;
4  op_map m_ed2el =
5          op_decl_map(s_edges, s_elems,1,ed2el,"m_ed2el");
6
7  int* cl2el = (int*) malloc(sizeof(int)*nCells);
8  for (int i = 0; i < nCells; i++)
9  {
10     cl2el[i] = i + nBdryEdges;
11 }
12 op_map m_cl2el =
13         op_decl_map(s_cells,s_elems,1,cl2el,"m_cl2el");
```

Listing 3.6: Implementation of the injection design pattern.

a linear algebra backend and to build an interface between OP2 and the backend library.

Apart from these limitations, which are intrinsic of the design of the library, OP2 also suffers from a few bugs, mainly due to the fact that the library is relatively young and development is still ongoing.

Although the library is not fully reliable for production code, the developers team works continuously on the project and it will soon be. This motivates our interest in OP2 as the base platform for a possible upcoming release of Rolls-Royce's HYDRA software. For this reason, part of the work done in this project involved the assessment of the capabilities of OP2. On this regard, OP2 was widely tested in the development of a small, benchmark application, that we named Heat, to which the following section is devoted.

3.3 Testing OP2: the Heat application

Heat is an OP2 application that solves an evolutionary heat conduction problem on a unit square, with fully Dirichlet boundary conditions.

Heat utilises many of the functionalities of OP2, and is aimed to serve as an example application to help software developers gain an understanding of developing applications with OP2. Accordingly, Heat was accepted as a contributed application in the main OP2 repository.

Heat is meant to solve the following problem

$$\begin{cases} \partial_t u - \nabla\cdot(\mu\,\nabla u) = f & \text{in } \Omega \times [0,T] \\ u(x,t) = g(x,t) & \text{on } \partial\Omega \times [0,T] \\ u(x,0) = u_0(x) & \text{in } \Omega \times \{0\} \end{cases} \tag{3.1}$$

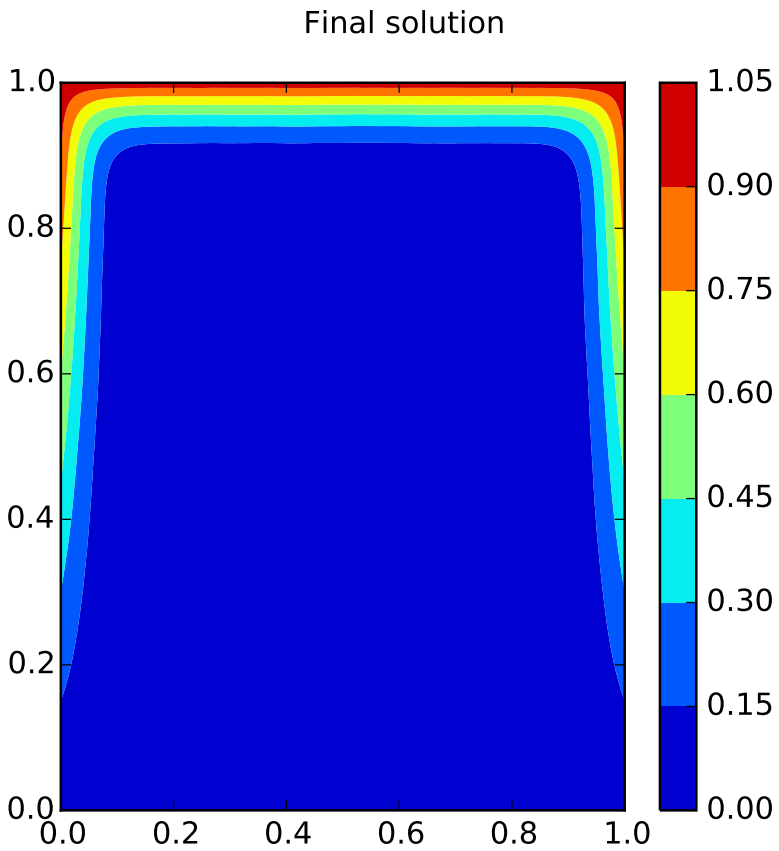

Figure 3.2: The solution produced by the Heat benchmark application for problem (3.1).

In the present version, $\Omega = [0,1]^2$, $u_0 = 0$, $g = y$, $f = 0$, $\mu = 0.001$ and $T = 0.5$, but these values can easily be changed in the code. The final solution is shown in Figure 3.2.

3.3.1 Finite differences version and CPU performance

The first version we developed solves (3.1) with a finite differences spatial discretization scheme and an explicit Euler time marching scheme on a structured, Cartesian grid.

The finite differences approximation is a standard second order accurate LeapFrog scheme: if we call u^l_{ij} the solution at time t^l at node (i,j), then the second order spatial derivative in the x direction becomes

$$\partial_{xx}u(x_i, y_j, t^l) \approx \frac{u^l_{i+1,j} - 2u^l_{ij} + u^l_{i-1,j}}{h_x^2}.$$

A similar formula holds for the derivative in the y direction.

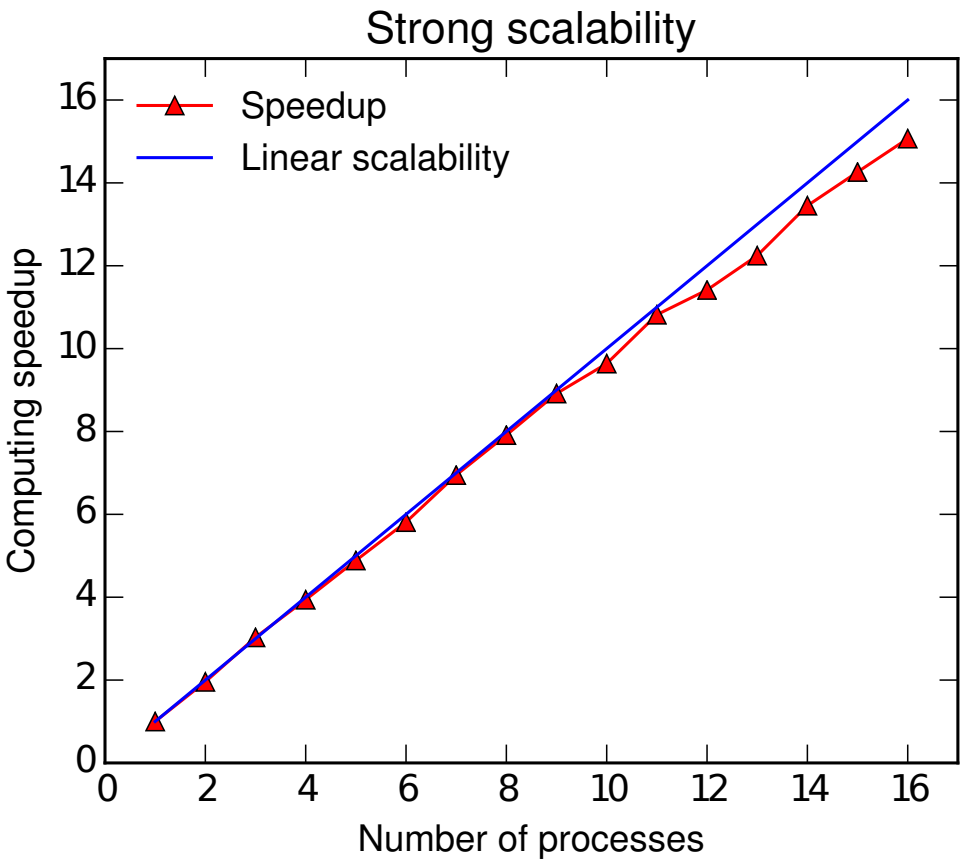

Figure 3.3: Shared memory scalability benchmark for OP2 and (3.1).

The application was wrapped with standard OP2 function calls, processed through the code generator and then benchmarked. The benchmark results for this first case are shown in Figure 3.3. In this first benchmark, an equally spaced grid of size 100×100 intervals was used, and CPU scalability for the shared memory paradigm was tested on an Intel Xeon CPU E5-2665 @ 2.40GHz.

The results are very good, as they show OP2 is able to scale almost linearly even on a relatively small grid and with as many as sixteen processes. The slight degradation of the scalability when the processors count increases can be motivated by the fact that the processor we used has exactly sixteen cores, which means that when all of them are being used the processor's bandwidth might saturate and cause small delays.

3.3.2 Finite volumes version and GPU performance

In a second benchmark, the GPU performance of OP2 was assessed by implementing a second version of the Heat application.

The second version features a finite volumes spatial discretization scheme on a general unstructured grid. This is a more natural framework for OP2, as in the previous example, due to the use of a structured grid, we did not have to use all the features of OP2.

Concerning the spatial approximation, we implemented a cell centered scheme, which means that the control volumes of the discretization coincide with the cells of the mesh. Given a mesh on the domain, we then integrate (3.1) in order to get, for every control volume E

$$\int_E \partial_t u - \int_E \nabla\cdot(\mu \nabla u) = \int_E f.$$

Using Stokes theorem now yields

$$\int_E \partial_t u - \sum_{e\in\partial E} \int_e \mu \nabla u \cdot \mathbf{n} = \int_E f;$$

with equation (3.1) in this form, boundary conditions are implemented in some of the line integrals, in particular those corresponding to the cell edges that belong to the boundary of the domain.

We can get a discrete scheme from the previous equation by approximating the integrals: a first order approximation gives the semi-discrete scheme

$$m_i \partial_t u_i = \sum_{j=i}^{M} m_{ij}\mu_i \frac{u_j - u_i}{\delta_{ij}} + m_i f_i \tag{3.2}$$

where m_i, u_i and f_i are the area, the value of the solution and the forcing of the i-th cell, m_{ij} is the length of the edge separating cells i and j and δ_{ij} is the distance between the barycenters of cells i and j; finally, M is the number of neighbours of a cell, so $M = 3$ in the case of triangular meshes.

To complete the discretization, we need to take care of the term $\partial_t u_i$: this can be done, upon defining a set of equally spaced discrete times $t_1, \ldots, t_n$, by setting, for example, $\partial_t u_i = \frac{u_i^{n+1} - u_i^n}{\Delta t}$. This is the usual Forward Euler method.

Finally, the fully discrete scheme reads

$$u_i^{n+1} = u_i^n + \Delta t \left[\frac{1}{m_i} \left(\sum_{j=i}^{M} m_{ij}\mu_i \frac{u_j^n - u_i^n}{\delta_{ij}} \right) + f_i \right] \tag{3.3}$$

The size of the time step Δt has to satisfy some stability condition depending on the values of the spatial grid step parameter h and the diffusivity coefficient μ. Typically one has $\Delta t \leq C\frac{h^2}{\mu}$, where C is some unknown constant.

Hardware	Release date
Intel Xeon Processor X5650	Q1 2010
NVIDIA Tesla M2070	Q3 2010

Run type	Time (s)
OpenMP - 6 threads	3373
OpenMP - 12 threads	1883
CUDA	311

Table 3.1: Scalability of OP2 for the finite volumes GPU benchmark.

In order to be able to run Heat on very fine meshes, so that we can show OP2's scalability, we would be forced to have an excessively small Δt to keep the Forward Euler scheme stable. To overcome this problem, we implemented a Runge-Kutta 4th order time marching scheme. This still does not allow the use of an arbitrary time step, but remarkably relaxes the constraint (in this scenario, still, $\Delta t = 10^{-5}$ in the final version).

With this final implementation we ran a benchmark to compare CPU and GPU scalability. We created a mesh of 1 948 676 cells and let Heat run on a single, multi-core processor with OpenMP and on a single GPU with CUDA generated code. The results are shown in Table 3.1. The GPU results are very good: OP2 is able to do six times better on a single GPU than on twelve threads. A very fine mesh was needed in order to expose the advantage of using a GPU; because of their architecture, GPUs and coprocessors in general show their real potential only when a great deal of parallelism is involved in the operations in progress, so that the massive amount of cores they have can be effectively exploited.

A general note on the comparison in Table 3.1: comparing GPUs and CPUs is usually somewhat unfair, as there is no such a thing as a conversion table that tells what CPU is equal to what GPU and in what sense. This means that it is in theory possible to obtain arbitrary results by simply changing the processor or coprocessor at hand.

A way out of this is to consider hardware that is either equally old or equally expensive. In our case we chose the former criterion as the CPUs and GPUs we had were released in the same year and finding out the release prices proved more nontrivial.

A final note on Heat. Because of the significance of the implemented benchmark, the Heat application was selected as a contributed benchmark to the OP2 project and will be soon added to the official repository[§].

[§] `http://www.oerc.ox.ac.uk/projects/op2`.

3.4 Final remarks on OP2 and dynamic compilation

The proposed benchmarks show the potential of the OP2 library as a general tool for automated parallelization of scientific computing applications. The innovative concept, inherited by its predecessor OPlus, is the source-to-source compilation process that generates highly optimised code for a variety of architectures. More importantly, the tool is extensible to support new architectures and little to no effort is required to the library user to adapt his or her code to run on yet to come architectures.

The idea of dynamic compilation is also present in other popular software packages. For example, the FEniCS project (see [LW12]) offers a comprehensive programming environment for the solution of PDEs by the finite element method with both a C++ and a Python interface. In the former case, a source-to-source compilation tool called `ffc` is provided to generate highly optimized code for runtime execution; in the latter case, a Just-In-Time (JIT) compiler called `Instant` compiles pieces of Python code into optimized C++ code at runtime, again separating the scientific development from the computer science implementation.

The PyFR project (see [WFV14]) is another case of a dynamic compilation approach: this Python software solves fluid dynamics equations by a Flux-Reconstruction scheme and provides JIT compilation of computational kernels targeting both CPUs and coprocessors in an automated, user-hidden fashion.

Chapter 4

RANS and LES

In this chapter we describe the CFD settings for the simulations performed throughout the work. The problem described is the compressible flow in an S-duct and the geometry used for the work is the one given in Wellborn et al. [WRO94]. The duct itself is accordingly called *Wellborn*.

Wellborn is a diffusive duct, meaning that the outlet area is bigger than the inlet one, and it fits a bounding box of dimensions, in meters, $(-0.4084, 1.7752) \times (-0.125799, 0.125799) \times (-0.399375, 0.102161)$.

The simulation parameters for the flow are given in Table 4.1. The Mach number is not excessively high, which means the flow is still not too far from being incompressible; this accounts for the possibility to use an incompressible solver to compare the results, whenever needed.

4.1 Mesh assessment and parameters

We know from experiments that the flow we studied exhibits separation. Nevertheless, this feature might not be present in the simulation output due to unfitness of the computational mesh used to approximate the flow.

Static pressure (Pa)	101 325.0
Static temperature (K)	288.15
Mean inflow velocity ($\mathrm{m\,s^{-1}}$)	120
Reynolds number (ca)	4×10^6
Mach number	0.27
Mass flow ($\mathrm{kg\,s^{-1}}$)	1.3065
Turbulence intensity (%)	5.0
Viscosity ratio	10.0

Table 4.1: Parameters for the flow in the Wellborn S-duct. The Reynolds number is computed based on the hydraulic diameter of the duct.

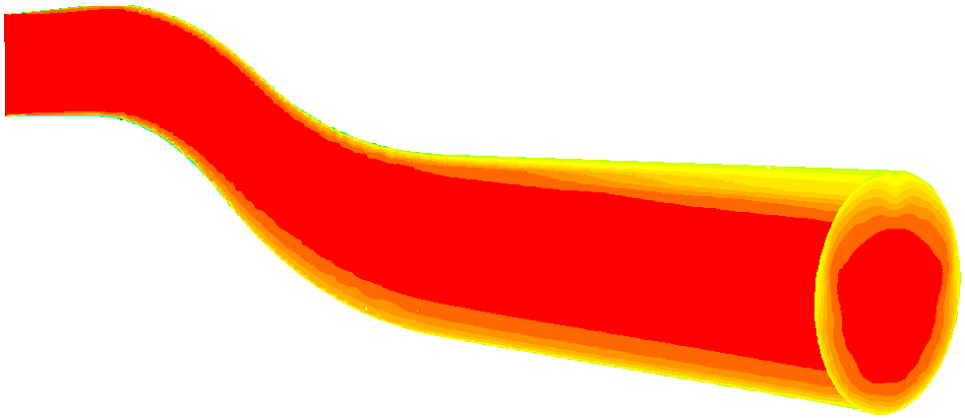

Figure 4.1: Total pressure for the flow on a mesh that does not properly capture the separation phenomenon of the flow. Green is 1.01×10^5 Pa and red is 1.07×10^5 Pa.

An example of this is shown in Figures 4.1 and 4.2, where the flow on the initial and final meshes of the study is represented by clipping the total pressure on a mid-plane and at the outlet. This aspect is of course of great importance and requires a separate study before one can start with the CFD simulations.

The simulations needed for this study, thus including those in Figures 4.1 and 4.2, were performed using Fluent and the RANS k–ω model we described in Chapter 2. The software we used for the creation of the mesh is BoxerMESH, from Cambridge Flow Solutions [Cam]. This software works by first creating an *octree* mesh of a given geometry and then body-fitting it to the boundaries. An octree mesh is a mesh created as follows: given a region to mesh and a bounding box, this box is subdivided into smaller boxes, arranged in a Cartesian grid; the octree mesh is then the collection of all the small boxes that are *fully* contained into the region to be meshed. Body-fitting is achieved by modifying the external boxes of the octree mesh so that they adapt to the boundary of the given shape.

We carried on a preliminary study of the mesh parameters, specifically as far as the boundary layer is concerned, as it turned out after a few attempts that this is the most critical feature to capture the separation. In particular, we found out that the mesh boundary layer, that is the thin layer of cells extruded from the wall to capture the physical boundary layer, needs to be thick enough, which means at least 15 mm, and it also has to satisfy the constraint $y^+ \approx 5$ as we do not want to use wall functions for the simulations. As a brief recall, y^+ is a non-dimensional

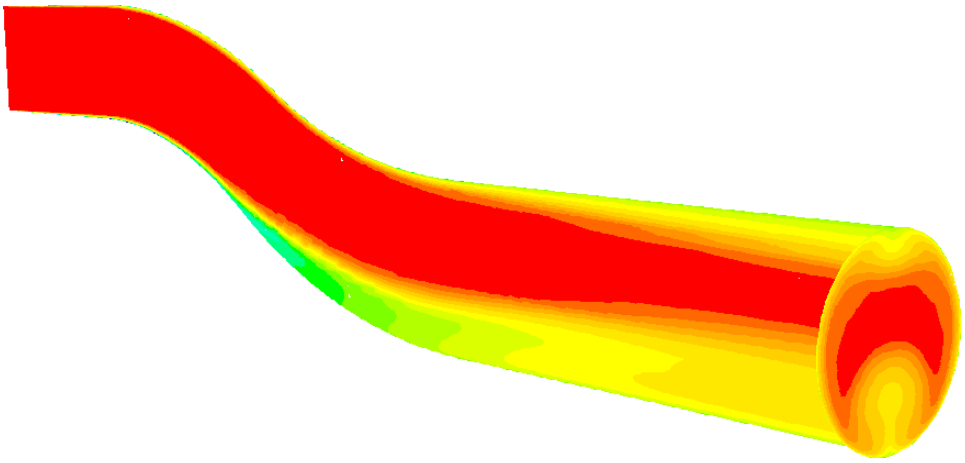

Figure 4.2: Total pressure for the flow on a mesh that properly captures the separation phenomenon of the flow. Green is 1.01×10^5 Pa and red is 1.07×10^5 Pa.

First layer's thickness (mm)	0.025
Expansion ratio	1.3
Number of layers	20
Full layer height (mm)	16 ca

Table 4.2: Boundary layer parameters for the accepted mesh.

wall distance and is defined as

$$y^+ = \frac{u_* y}{\nu} = \frac{\sqrt{\frac{\tau_w}{\rho}}\, y}{\nu} = \frac{\mu y \left.\frac{\partial u}{\partial y}\right|_{y=0}}{\nu\sqrt{\rho}} = \sqrt{\rho} y \left.\frac{\partial u}{\partial y}\right|_{y=0},$$

where $u_* = \sqrt{\frac{\tau_w}{\rho}}$ is the friction velocity and $\tau_w = \mu \left.\frac{\partial u}{\partial y}\right|_{y=0}$ is the wall shear stress.

Since the definition of y^+ involves the value of u, which is unknown a priori, the mesh study procedure has to be iterative in the sense that the flow is solved on an initial mesh, then the y^+ is computed and the boundary layer is refined accordingly.

Of course, refining the boundary layer means producing a bigger mesh in terms of cell numbers, increasing the computational cost of the whole simulation: this means we have to find a trade-off between the two. The final parameters we accepted for the simulation are shown in Table 4.2 and the corresponding y^+ is shown in Figure 4.3.

We would like to stress that the y^+ alone is not enough to guarantee the efficacy of the mesh. As we said, the boundary layer has to extend at

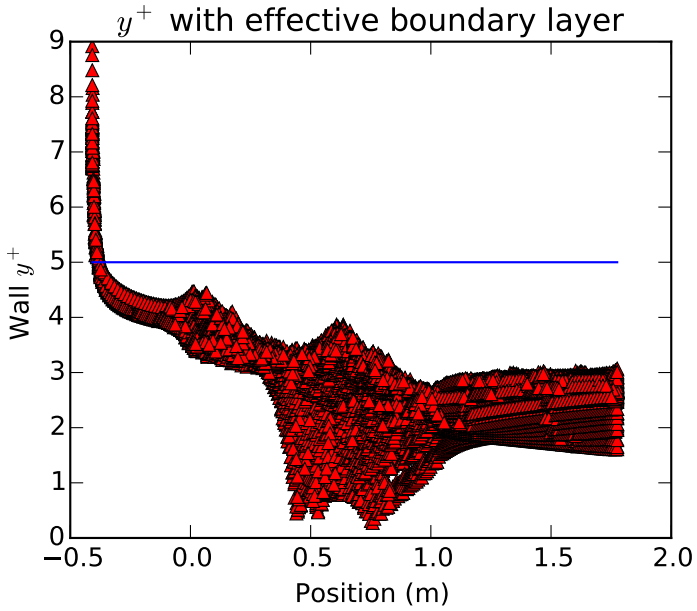

Figure 4.3: y^+ values along the wall for the accepted mesh.

least 15 mm into the flow zone, and refining the internal part of the mesh did not seem to overcome a too short boundary layer. This is probably related to the way the mesh is built: when the body-fitting is performed, hexahedra are converted into prisms, breaking the structure of the mesh and possibly ruining the local numerical accuracy.

4.2 RANS

RANS simulations were used throughout the project to get an initial understanding of the flow properties as well as to drive the distortion index-based optimisation procedure.

The chosen model for the simulations was Menter's SST (Shear-Stress Transport) k–ω model. It is a fairly robust and reliable model, it proved efficient to investigate the general properties of the flow in the S-duct, which is not too complex as compared to other geometries such as a full turbine, but still exhibits peculiar features. Menter's SST model was introduced in Menter [Men94] as a coupling of the k–ω and k–ε models so that the former is used in the inner region of the boundary layer, whereas the latter is used in the free shear flow. For completeness, we

recall the equations of the model in Equation (4.1):

$$\begin{aligned}\partial_t(\rho k) + \nabla\cdot(\rho k u) =& P - \beta^*\rho\omega k + \nabla\cdot[(\mu + \sigma_k\mu_t)\nabla k]\\ \partial_t(\rho\omega) + \nabla\cdot(\rho\omega u) =& \frac{\gamma}{\nu_t}P - \beta\rho\omega^2 + \nabla\cdot[(\mu + \sigma_\omega\mu_t)\nabla\omega]\\ &+ 2(1-F_1)\frac{\rho\sigma_{\omega 2}}{\omega}\nabla k\cdot\nabla\omega\end{aligned} \tag{4.1}$$

where we defined

$$\begin{aligned}P &= \tau\cdot\nabla u\\ \nu_t &= \frac{\rho a_1 k}{\max(a_1\omega, \Omega F_2)}\\ \varphi &= F_1\varphi_1 + (1-F_1)\varphi_2 \quad (\varphi = \sigma_k, \sigma_\omega, \beta, \gamma)\\ F_1 &= \tanh(arg_1^4)\\ arg_1 &= \min\left[\max\left(\frac{\sqrt{k}}{\beta^*\omega d}, \frac{500\nu}{d^2\omega}\right), \frac{4\rho\sigma_{\omega 2}k}{CD_{k\omega}d^2}\right]\\ CD_{k\omega} &= \max\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\nabla k\cdot\nabla\omega, 10^{-20}\right)\\ F_2 &= \tanh(arg_2^2)\\ arg_2 &= \max\left(\frac{\sqrt{k}}{\beta^*\omega d}, \frac{500\nu}{d^2\omega}\right)\end{aligned}$$

d is the distance to the nearest wall, $\Omega = \sqrt{2}\|\frac{\nabla u - \nabla u^T}{2}\|_2$ and we close the equations with

$$\gamma_1 = \frac{\beta_1}{\beta^*} - \frac{\sigma_{\omega 1}\kappa^2}{\sqrt{\beta^*}} \qquad \gamma_2 = \frac{\beta_2}{\beta^*} - \frac{\sigma_{\omega 2}\kappa^2}{\sqrt{\beta^*}}$$

$$\begin{aligned}\sigma_{k1} &= 0.85 & \sigma_{\omega 1} &= 0.65 & \beta_1 &= 0.075\\ \sigma_{k2} &= 1 & \sigma_{\omega 2} &= 0.856 & \beta_2 &= 0.0828\\ \beta^* &= 0.09 & a_1 &= 0.31 & \kappa &= 0.41\end{aligned}$$

As far as the boundary conditions are concerned, at the inlet we set a total temperature of 293.05 K, a total pressure of 107 588 Pa, $k = 1\,m^2/s^2$ and $\omega = 5555.5\,s^{-1}$, which are reportedly common values in engineering practice; we tell Hydra to treat the wall of the S-duct as a viscous wall, thus imposing zero velocity; finally, at the outlet we specify the static pressure to be 101 325 Pa. These boundary conditions are enough for Fluent of Hydra to setup the configuration and run the solver. The missing boundary conditions are deduced by the codes from the given ones; since both are closed-source applications, we could not find out what exactly goes on behind the scenes. The mesh we used, which is the one we obtained at the end of the study we described above, has 701 437 hexahedra.

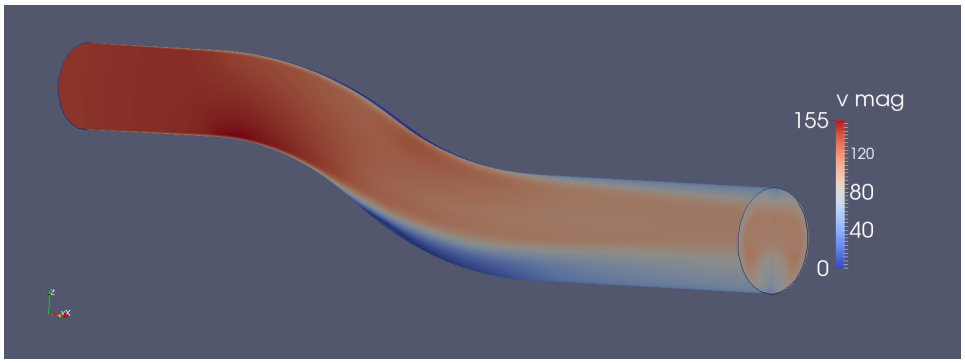

Figure 4.4: Flow simulation using Menter's SST k–ω model on the initial Wellborn shape. The separation zone on the first bend and the inhomogeneous outlet are well captured by the simulation.

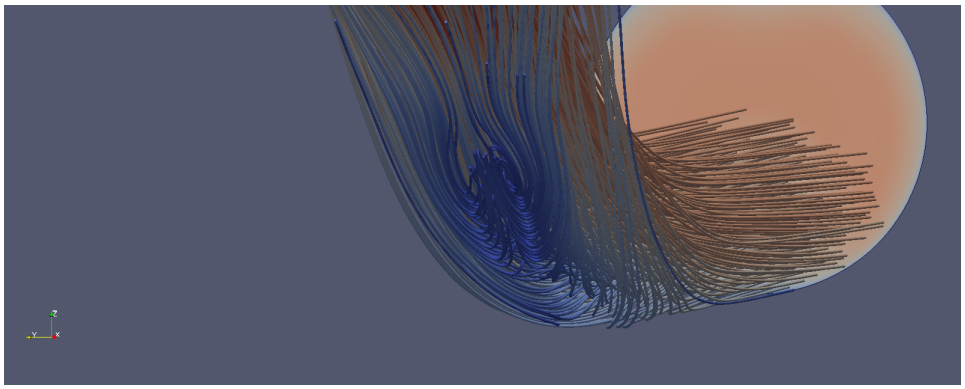

Figure 4.5: Close up on the separation zone of the Wellborn duct for the initial configuration. The recirculation zone is clearly visible.

The flow in the given configuration is shown in Figure 4.4 and a close up on the separation zone, visualized by means of streamlines, is given in Figure 4.5.

The flow exhibits all the features we are expecting, and even more. A close look at the upper wall of the second straight part of the S-duct reveals that a much smaller separation bubble is forming also after the second bend, which acts as a *second* backward-facing step. This second separation bubble is less relevant than the first one, so we will almost completely ignore it in the following; we will, however, refer to it once at the end of Chapter 6.

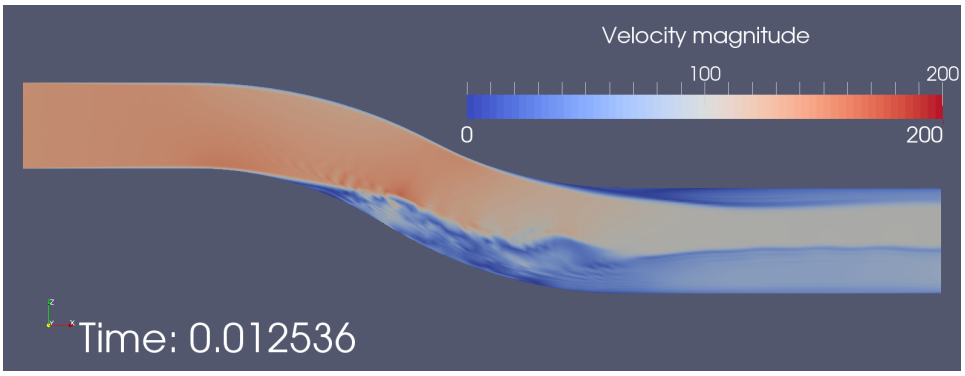

Figure 4.6: LES solution before the flow can complete a first flow-through in the duct. A residual of the smooth initial condition is still visible in the right-most part of the duct.

4.3 LES

Large Eddy Simulations were used for the most scientifically interesting part of the project, namely the spectral analysis.

The scheme used in Hydra is a Kinetic Energy Preserving (KEP) conservative scheme introduced in Jameson [Jam08] and Eastwood et al. [Eas+09].

In Jameson [Jam08], in particular, it is shown that in order to obtain a KEP scheme it is sufficient that, for any two control volumes o and p, the convective term satisfies

$$(\rho u^i u^j)_{op} = \frac{1}{2}(\rho u^i)_{op}(u^j_p + u^j_o), \tag{4.2}$$

and accordingly sets $(\rho u^i)_{op} = \overline{\rho u}^i$ and $(\rho u^i u^j)_{op} = \overline{\rho u}^i \overline{u}^j$ where the bar sign denotes arithmetic averaging between the control volumes.

For the Large Eddy Simulations we used the same boundary conditions that we used for the RANS simulations; of course k and ω are not present anymore. For the LES we used the same mesh as for the RANS simulations. This is mainly because we know that the RANS mesh can correctly capture the boundary layer; we also verified, after running it, that the simulation on this mesh yields reasonable results. As Large Eddy Simulations are unsteady, an initial condition is also needed; we used the steady solution in Figure 4.4 as an initial condition for our LES of the flow in Wellborn. In any case, a certain number of initial time steps needs to be discarded as it is still affected by the initial condition, as shown in Figure 4.6. For the case of our simulation, with a time step of 2.5×10^{-5} s, about 800 time steps are needed to allow the fluid initially at the inlet to flow through the duct and reach the outlet; this is called a flow-through. The initialised flow is shown in Figure 4.7.

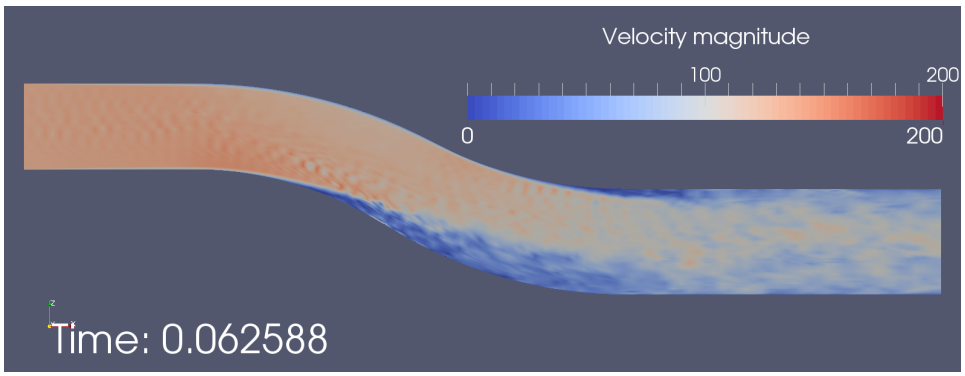

Figure 4.7: A snapshot of the flow after the first flow-through. The solution got rid of the initial condition and is now fully turbulent.

Chapter 5

Post-processing tools and techniques

In this chapter we proceed to the description of our new methodology for real-time post-processing. First, a deeper review of the mathematics behind POD and DMD is given, together with a comparison of the two with each other and with existing methods. After introducing the techniques, a validation test case is presented to show that the implementation is working. We later give details on the code we implemented, highlighting its structure and giving details of the most important parts, and present a brief analysis of its performance and scalability. Finally, we discuss issues and improvements in the implementation.

5.1 The identification of coherent structures

The problem of the identification of coherent structures in a turbulent flow has attracted a lot of attention in the literature in the past years [Hol+12]. Giving a good definition of a coherent structure is trickier than it looks; we follow Chong et al. [CPC90] and define a vortex core as a region of space where the vorticity is sufficiently strong to cause the rate-of-strain tensor to be dominated by the rotation tensor. This definition allows the use of criteria based on invariants of the velocity-gradient tensor, thus being Galilean invariant. The term coherent structure then includes vortical structures and shear layers. The main reason for the interest in coherent structures is that, when they were first observed, they appeared as a viable alternative to the mere – and frustrating – exclusively statistical study of turbulence, which was considered, up to that point, as a purely uncorrelated phenomenon [Fie88].

Many post-processing techniques were since developed and a fairly comprehensive survey is presented in von Terzi et al. [vTSF09]. One of the most popular, that is to some extent related to POD and DMD, is the

so-called Q-criterion, named after the letter used for the quantity

$$Q = \frac{1}{2}(P^2 + \|W\|_2^2 - \|S\|_2^2), \tag{5.1}$$

where we have set $P = -\nabla \cdot \mathbf{u}$, $W = \frac{\nabla \mathbf{u} - \nabla \mathbf{u}^T}{2}$, $S = \frac{\nabla \mathbf{u} + \nabla \mathbf{u}^T}{2}$.

The quantity Q is the second invariant of the gradient of the velocity u, and its value is associated to the vorticity of the flow and consequently used for vortex core identification.

The Q-criterion allows to identify coherent structures in a flow by looking at the contours (iso-surfaces) of Q: if confined zones of higher Q are present, then they will appear as contours in the post-processing.

The limit in these approaches is that they take the flow *as is*, without any kind of advanced processing technique apart from the direct computation of quantities of interest. In other words, these methods simply proceed to the computation of some quantity which is useful for a *visual* representation and investigation of the features of the flow field. In the following, we will show how this limit can be surpassed.

5.2 Detailed description of POD and DMD

5.2.1 POD

Let $\mathbf{u}(\mathbf{x}, t)$ be the flow velocity computed by an unsteady CFD simulation on a domain with m cells and at n different time steps. The main idea behind spectral analysis is that of looking at the flow function as the superposition of several *modes*, each containing a part of the total information. In our case, we assume that the expansion can be done in a separated-variables fashion as

$$\mathbf{u}(\mathbf{x}, t) = \sum_{i=0}^{\infty} a_i(t)\varphi^i(\mathbf{x}). \tag{5.2}$$

It is clear that there are infinitely many pairs of coefficients a_i and basis functions φ^i that can do this, so it is legitimate to ask how to choose one.

It would be useful it the φ^i were orthogonal, as this would make it easy to compute the coefficients a_i. Since in practice one can only add a finite number of components to approximate a function, we would like to guarantee that if we only sum M factors, then we get the best possible approximation, in the least squares sense, with M factors. When both these conditions are satisfied, then (5.2) is the POD of function $\mathbf{u}$.

The mathematical method behind the Proper Orthogonal Decomposition is the Singular Value Decomposition (SVD). Suppose we record data from a simulation in m grid cells at n different time steps: we can then

arrange this data in a matrix $A \in \mathbb{R}^{m\times n}$. The SVD of matrix A is a decomposition of the form

$$A = U\Sigma V^H = \underbrace{\begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_m \end{bmatrix}}_{\in\mathbb{R}^{m\times m}} \underbrace{\begin{bmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_p & & \\ & & & 0 & \\ & & & & \ddots \end{bmatrix}}_{\in\mathbb{R}^{m\times n}} \underbrace{\begin{bmatrix} \mathbf{v}_1 & \dots & \mathbf{v}_n \end{bmatrix}^H}_{\in\mathbb{R}^{n\times n}} \tag{5.3}$$

where the ^{H} superscript denotes Hermitian transposition.

The decomposition is the product of three matrices: a unitary matrix U of m vectors of length m, a rectangular diagonal matrix of ordered entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0 = 0 = \dots$ and a unitary matrix V of n vectors of length n. The vectors u_i are called the left singular vectors, vectors v_i are called the right singular vectors and values σ_i are called the singular values. Once the decomposition is completed, one can reconstruct the initial data as

$$\mathbf{u}(\mathbf{x}, t) = \sum_i \sigma_i \mathbf{u}_i \otimes \mathbf{v}_i, \tag{5.4}$$

that is, as a sum of rank-1 tensors, accordingly with the fact that they have to be basis vectors of a space-time vector space.

It should not take long to convince oneself that the vectors $\mathbf{u}_i$ are the counterpart of the functions φ^i in (5.2) and the vectors $\mathbf{v}_i$ are the counterpart of coefficients a_i in (5.2). Thus, vectors $\mathbf{u}_i$ contain the spatial information of the decomposition, whereas vectors $\mathbf{v}_i$ contain the temporal information. The singular values σ_i act as weighing factors, telling how important each component is in the whole decomposition. A weighing factor *must* be present as both the $\mathbf{u}_i$ and $\mathbf{v}_i$ have unitary norm, as matrices U and V are required to be unitary.

The SVD has some very interesting properties that contributed to its popularity in the literature. First of all, the decomposition always exists even for general rectangular matrices. This is a form of robustness of the method, which guarantees to yield a solution whatever the initial conditions are. Secondly, the SVD of a real matrix always produces real matrices U, Σ and V. This accounts for a direct interpretation of the results in the familiar field of real numbers.

For completeness, we shall now describe how to compute the SVD of a given matrix A. We will present only one of the possible methods, which is known as the *cross* method for the computation of the SVD of a matrix.

By equation (5.3), we have that

$$\mathcal{A} = A^H A = (U\Sigma V^H)^H (U\Sigma V^H) = V\Sigma^2 V^H,$$

where we have used the facts that $U^H U = I$, since U is a unitary matrix, and $\Sigma^H \Sigma = \Sigma^2$ since Σ is diagonal. Right-multiplying by V we have

$$\mathcal{A}V = \Sigma^2 V$$

or, if we take row i,

$$\mathcal{A}\mathbf{v}_i = \sigma_i^2 \mathbf{v}_i.$$

This means that row vectors $\mathbf{v}_i$ are the right eigenvectors of $\mathcal{A}$ with associated eigenvalues σ_i^2. We note that $\mathcal{A}$ is hermitian symmetric as $(A^H A)^H = A^H (A^H)^H = A^H A$ and thus, by the Spectral theorem, is diagonalizable with real positive eigenvalues σ_i^2, so that the values σ_i are actually real positive numbers.

Once the $\mathbf{v}_i$ are known through the solution of the eigenvalue problem, right-multiplying (5.3) by V gives

$$AV = U\Sigma$$

or, reading column i,

$$\frac{1}{\sigma_i} A\mathbf{v}_i = \mathbf{u}_i,$$

which defines the $\mathbf{u}_i$ and completes the decomposition.

It should be noted that, in principle, an equivalent derivation can be carried on using $\mathcal{A}^H = AA^H$ and resorting to an eigenvalue problem for $\mathbf{u}_i$.

Although theoretically equivalent, on the computational side one of the two approaches might be preferable. Using $\mathcal{A}$ yields an eigenvalue problem on a matrix of size $n \times n$, whereas using $\mathcal{A}^H$ yields an eigenvalue problem on a matrix of size $m \times m$, thus the criterion is the shape of A. If $m = n$, there is no difference in the computational cost, whereas if $m \neq n$ using $\mathcal{A}$ is cheaper if $m > n$ and using $\mathcal{A}^H$ is cheaper otherwise.

In our setting, we always have $m \gg n$, as mesh sizes can range up to tens of millions, whereas the number of time steps hardly exceeds a few thousands.

This has important implications on implementation choices in the development of the code, as described later.

Now that the math is set, a few words on the physical meaning of the POD. The most important notion is the relationship between the singular values and the kinetic energy of the flow. Although it is not always legitimate to associate singular values and energy contents, fluid dynamics is a lucky exception [Cha00].

This relationship allows for an important interpretation of the results given by POD in terms of energy content of the flow being simulated: one can now tell how important each mode is in comparison with the others.

This feature of POD, although important, still requires that the orthogonal modes have a physical meaning in the picture of the simulated flow. This turns out to be true and, as reported for instance in von Terzi et al. [vTSF09], the orthogonal modes computed by the POD represent the coherent structures of the flow. This fundamental fact lets us now interpret the whole process of Proper Orthogonal Decomposition as a means to not only *separate* the various coherent structures composing a flow, but also to weigh them with their energy content. This feature gives techniques based on spectral analysis, such as the POD, an edge over previously existing methods such as the Q-criterion, which is able to identify coherent structures in general, but not to tell the energetic content they have.

An interesting property of POD is that not all the singular values are needed to analyse experimental data, but only the most important ones. This perfectly matches the fact that numerical techniques for eigenvalue computation rely on iterative procedures which more easily approximate eigenvalues of bigger magnitude.

5.2.2 DMD

The Dynamic Mode Decomposition has a slightly different approach from the POD.

Let again $\mathbf{u}(\mathbf{x}, t)$ be a simulated flow field and let us arrange the data from m cells at $n-1$ time steps as columns in a matrix U_1^{n-1}, and similarly build U_2^n.

The core assumption of DMD takes place now: we assume that there is a linear mapping A such that

$$AU_1^{n-1} = A \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_{n-1} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_2 & \dots & \mathbf{u}_n \end{bmatrix}. \tag{5.5}$$

This assumption is extremely interesting: on one side, we know that in general this *cannot* happen unless $m = n-1$ and the $\mathbf{u}_i$ are linearly independent; on the other side, this assumption can be seen to be somewhat reasonable in the special case of turbulent fluid flows.

The idea behind this is that the evolution of a fluid flow not only is not completely random, but also lies in a subspace of $\mathbb{R}^m$ of dimension $n-1$.

Turbulent flows like the one we are concerned with in this work often exhibit a sort of periodic behavior, for example in the velocity oscillations at a separation interface. The DMD is meant to be applied to this kind of phenomena.

If the assumption is justified, then matrix A rules the evolution of the flow, at least from time steps $1, \dots, n-1$ to n, and its eigenvalues and eigenvectors contain the precious information about the flow field that we are looking for. This is the claim of DMD, that coherent structures of the flow can be identified by studying the eigenvalues and eigenvectors of matrix A.

The first problem to solve now is how to find A. A clever solution to the problem is found by a simple algebraic manipulation. From equation (5.5), exploiting the structure of U_1^{n-1} and U_2^n, we can write

$$U_2^n = \begin{bmatrix} \mathbf{u}_2 & \dots & \mathbf{u}_n \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_{n-1} \end{bmatrix} \begin{bmatrix} 0 & & & & c_1 \\ 1 & 0 & & & c_2 \\ & \ddots & \ddots & & \vdots \\ & & 1 & 0 & c_{n-2} \\ & & & 1 & c_{n-1} \end{bmatrix} = U_1^{n-1} S. \tag{5.6}$$

Equation (5.6) defines the *companion matrix* S, whose action is to translate the first $n-2$ columns of U_1^{n-1} to the left, removing the first one, and to add the last one as a linear combination of the previous ones. In fact, considering the equations by columns, the last one reads

$$\mathbf{u}_n = \sum_{i=1}^{n-1} c_i \mathbf{u}_i = c_1 \mathbf{u}_1 + c_2 \mathbf{u}_2 + \dots + c_{n-1} \mathbf{u}_{n-1}$$

This equation defines the coefficients c_i of matrix S.

The eigenproblem on A has thus been recast into an eigenproblem on S, a much easier matrix to compute.

For the reasons mentioned above, the c_i are not guaranteed to exist in the general case. What we would do, to perform the actual computation, is to solve a least squares problem with the last column-equation of (5.6), namely

$$\begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_{n-1} \end{bmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} \mathbf{u}_n \end{pmatrix}, \tag{5.7}$$

which means that the linear approximation now reads

$$\mathbf{u}_n = \sum_{i=1}^{n-1} c_i \mathbf{u}_i = c_1 \mathbf{u}_1 + c_2 \mathbf{u}_2 + \dots + c_{n-1} \mathbf{u}_{n-1} + \mathbf{r}$$

where $\mathbf{r}$ is the residual of the least squares solution of (5.7). Incidentally, one can consider the norm of $\mathbf{r}$ as an a posteriori measurement of the

fitness of the hypothesis for the particular case at hand, i.e. if the residual norm is zero, then the hypothesis is fully legitimate, whereas if it is big then the hypothesis is not really justified.

Once the c_i are known, the eigenvalues σ_i and eigenvectors ψ_i of S can be computed. The last step in the decomposition is to compute the dynamic modes $\Phi_i = U_0^{n-1}\psi_i$; additionally, the eigenvalues can be transformed into a physically more significant form by $\lambda_i = \frac{\ln(\sigma_i)}{\Delta t}$, which separates in the real and imaginary parts of λ_i the modulus and phase of σ_i [Sch10].

This representation allows to tell at a glance which eigenvalues are unstable, which are stable and how fast they grow or decay, together with the oscillatory frequency of the associated eigenvectors.

This property suggests the physical interpretation of the DMD: it is a procedure that extracts dynamic modes – which turn out to identify coherent structures – *by frequency*. Instead of expanding the given data as a combination of space-time basis functions, DMD yields spatial-only fields, together with a modulus and an oscillation frequency.

If we order the λ_i by descending eigenvalue modulus, we find that the first eigenvalue is real, and so is its associated eigenvector, which represents the mean flow.

As with POD, DMD also gets along with numerical techniques as the most important modes are the ones associated to the eigenvalues with bigger modulus.

5.3 The need for a linear algebra back-end

Both POD and DMD involve linear algebra computations, thus we need a linear algebra back-end. OP2 is not fit for this kind of operations, so we need to look for something else.

Since we want to be able to target a variety of parallel architectures at once, we must carefully choose the library to interface with. Two main alternatives are available: a recent, interesting project is Matrix Algebra on GPU and Multicore Architectures (MAGMA), whereas a more classic choice is the Portable, Extensible Toolkit for Scientific computation (PETSc), coupled with the closely related Scalable Library for Eigenvalue Problem Computations (SLEPc).

MAGMA is a project run by the Innovative Computing laboratory at The University of Tennessee [BCP97]. Its purpose is to provide a dense linear algebra library that can run efficiently on a vast combination of heterogeneous architectures. The focal point of the development team is the ability to fully exploit *all* the available hardware at the same time. That is, if both CPUs and GPUs are available, then MAGMA aims to share the workload across all the hardware in order to fully exploit it,

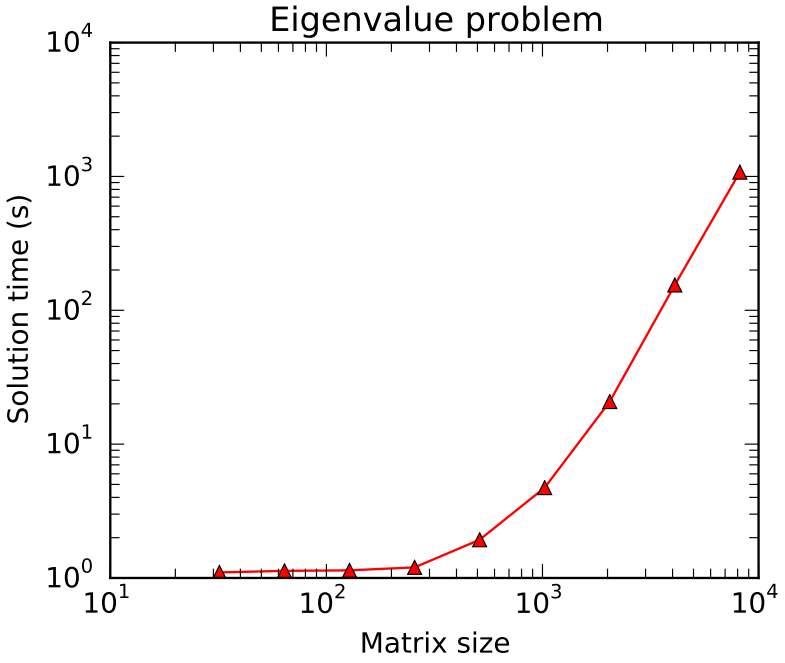

Figure 5.1: Performance of eigenvalue problem solution with MAGMA at various matrix sizes.

instead of performing computations exclusively on the CPUs or on the GPUs.

MAGMA provides a very similar API to the one provided by the famous LAPACK package. In particular, the whole parallelization layer is user-transparent, which makes it easy to use to users who are only familiar with LAPACK. On top of that, a higher-level API is provided to perform more elaborate operations such as solving eigenvalue problems.

As part of the preliminary study for the software development phase, we tested the scalability performance of MAGMA on a computer node with two NVIDIA Tesla M2070 and twelve Intel Xeon CPU X5650. The scalability result is shown in Figure 5.1. The performance is good as long as the matrix order is below a thousand. Past that limit, the burden of managing a dense matrix storage becomes predominant and the execution time grows exponentially.

Since the purpose of the piece of software we want to develop is to implement POD and DMD procedures, meaning the matrix order for the eigenvalue problem is the number of time steps being considered, this could not be terribly bad in principle, as we anticipated that we wish to keep the number of time steps to be computed limited. In practice, though, one cannot exclude the possibility that more complicated flow cases re-

quire more time steps to be successfully analyzed. Also, the eigenvalue problem of DMD is very sparse, so using a dense storage is a clear waste of resources. Finally, at the time of writing MAGMA lacks support for distributed memory parallelism, which limits its usage to single nodes. For these reasons, we opted for PETSc+SLEPc as our linear algebra backend of choice.

PETSc is a famous linear algebra package that supports distributed memory parallelization, with good scaling up to thousands of cores. It is notably written in C, but with an object-oriented paradigm; it is mainly meant to handle sparse matrices, but provides little support for dense matrices as well. Since a few years PETSc added support for GPU computation by adding subclasses for matrices and vectors that perform computation with either CUDA through CUSP or OpenCL through ViennaCL [MSK10].

SLEPc is a generalized eigenvalue problem solver library written on top of PETSc, meaning that it provides additional higher-level tools implemented using PETSc and exposes a very similar API to the end user. Thus, SLEPc inherits all the benefits PETSc already has. For these reasons, we decided to base our code on PETSc+SLEPc for the linear algebra computation. This will later pose the problem of converting data from a format intelligible to Hydra into one that is understandable by PETSc.

We do not present a scalability benchmark for PETSc+SLEPc at this point, as this is deferred to a later section. What we stress here is that PETSc allows to use different matrix types in different parts of the same application without any additional implementation cost. This makes it very easy to experiment with both dense and sparse matrices, as well as running on CPU or GPU harware.

5.4 Detailed description of `Spectre`

In this section we describe the code that we developed for the project, called `Spectre`.

We recall that the aim of the projects is twofold: on one side we want to implement POD and DMD techniques for spectral analysis of unsteady flows, whereas on the other side we also need to provide capabilities to compute distortion indexes. Despite the dual scope, the higher importance of the spectral analysis part accounts for the given name.

A first constraint to take into account is that, due to security reasons, we were not given access to the Hydra source code during the project period, so this first implementation will need to read Hydra data files from the hard disk, this operating mode is called *offline* as data production and data analysis happen in two separate, although possibly time parallel, stages.

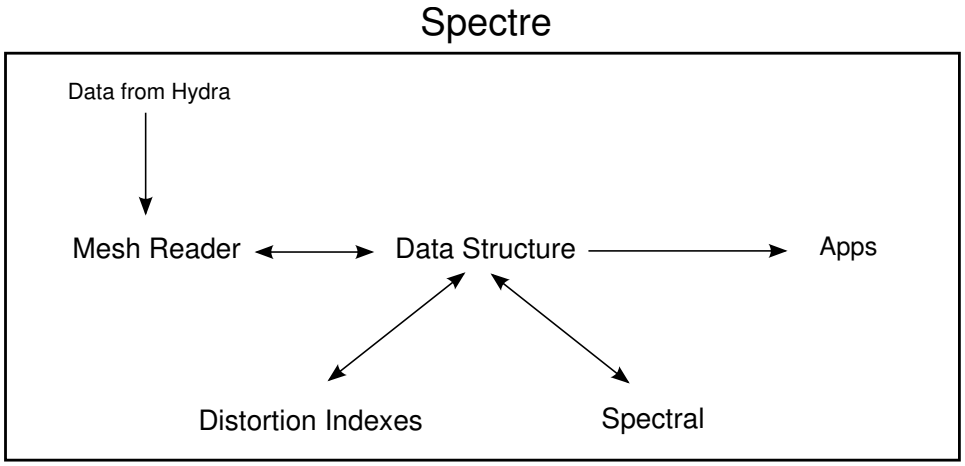

Figure 5.2: A schematic representation of the code.

`Spectre` also needs to be able to interface to different data formats in a modular way, so that the user can read Hydra output and pass it to PETSc without effort.

The whole code should also be well designed, as it is not meant for a single use case, but rather to be incorporated into production code in the near future. This will turn it into an *online* analysis tool, where simulation and post-processing happen in a single stage of the same application.

To meet these needs, we decided to write `Spectre` in C++. We would like to point out that this does not pose compatibility problems with HYDRA, which is written in FORTRAN and C, because C and C++ libraries can be linked together with little effort: all that is required is to expose a C API to the C++ library, that is a collection of function wrappers.

Before going into the details of the code, a schematic representation of the code structure is presented in Figure 5.2.

According to the above description, `Spectre` features a central namespace, `dataStructure`, that holds classes to represent various dataset types. These classes interface with classes in a `meshReader` namespace that handle data read from HYDRA files and perform various operations on them. Once the data is read into appropriate datasets, two processing modules provide capabilities to elaborate the data: `distortionIdx` and `spectral`. We will now give a brief description of the code structure, highlighting the main features and strengths of the code.

5.4.1 Mesh reader module

This module is concerned with the reading of mesh data stored in HYDRA files. The main class in this namespace is the `Domain` class. The `Domain` class only takes care of storing the geometrical informations about

the domain, that is the mesh cells coordinates, boundary conditions and similar data. A critical capability we want to have in `Spectre` is the ability to consider only a part of the mesh, say only a box region or the inlet boundary. This is useful for several reasons: on one side, spectral analysis can be performed only in some regions of interest of the whole simulation domain, saving a good deal of computational time; on the other hand, one might also want to keep track of quantities such as the pressure drop from inlet to outlet, thus needing to compute quantities on the boundary. Cell coordinates in the `Domain` class are stored in an `std::vector<std::array<double,3>> coords` object whose i-th element holds the coordinates of the i-th cell, so we need to somehow be able to filter, or mask, this vector to expose only the elements corresponding to the cells we want to consider at a given time; we will call these cells *active*. This masking should also be transparent to the user, who is always supposed to access data from a domain with the same function, and yield a small computational overhead. The way we decided to implement this masking is to add a data member `std::vector<size_t> range` that stores the list of indexes of the active cells. A boolean flag will now keep track of the status of the domain instance: we call *restricted* a domain whose active cells are not all the mesh cells. Accordingly, the *size* of the domain, which is an integer indicating the number of cells in the mesh, will now be either the number of total cells or the number of active cells, depending on the value of the flag variable. Implementing suitable access methods, a class user can now simply loop over the mesh elements, driven by the domain size as specified above, and access either the whole domain or a subset with the same programming interface.

All the geometrical operations on the domain are now handled internally by the `range` object. Suitable methods are provided to modify this vector, the most important ones are collected in a set of classes specifically designed for geometric operations. The fundamental class for geometric operations is the `Region` class. This is an abstract interface that provides only a method `virtual bool inside (double,double,double) = 0` that tells if a given cell, whose coordinates are passed as parameters, is inside or outside the region. Implementing several types of restriction operations is now simply a matter of inheriting from this abstract class and reimplementing the `inside` method so that it reflects the particular region being described by the child class. Implemented examples of subclasses of `Region` are `Box`, that describes a three-dimensional box region, and `HalfSpace`, that allows the user to cut the mesh with a plane of equation $ax + by + cz + d = 0$, given the parameters a, b, c, d.

Another great advantage of the way this feature is implemented is that it easily allows to join two regions, to cut a region with another or to take the whole domain but a given region. All these operations require simple manipulation of the `range` vector and are a reimplementation of

the well know set operations union, difference and complementation. It is clear how powerful this approach is: with only a handful of operations the user has great control over mesh partitioning.

It should be pointed out that, when running in parallel, this kind of partitioning does *not* require to redistribute data over the computational nodes. This is because data is also time-dependent and thus parallel distribution of data can be effectively achieved by distributing time steps. Because of the nature of the implemented spectral analysis techniques described above, this is the most natural way to distribute data in parallel also because each time step corresponds to a single row or column in the matrix operations involved, so this approach carries very little overhead to the parallel linear algebra back-end.

5.4.2 Dataset module

Once the mesh information has been read into a `Domain` instance, the user can read the flow field information. These two steps are split into two separate operations to reflect HYDRA's output strategy: HYDRA creates one file for the mesh information and another file for the field data. These latter always include density, velocity and static pressure, and include k, ε, ω or other quantities depending on the particular turbulence model used for the simulation. Because of this feature, the class needs to be able to handle a variable number of fields, not necessarily known at compile time.

The same class is used to handle both input and output of both steady and unsteady datasets. A number of convenience member functions are defined to access the field variables or to compute derived quantities, for example the total pressure. All the methods that access data or mesh coordinates are implemented as wrappers to the underlying `Domain` object, that provides much of the functionality.

A critical feature of this class is the ability to push and pop time steps dynamically. This is needed in order for the post-processing technique to follow the CFD simulation step-by-step and to update the spectral analysis at each time step.

5.4.3 Distortion indexes and spectral analysis

We introduced distortion indexes in Section 2.4 and we will need them in Chapter 6 for shape optimisation.

When dealing with the computation of these indexes, the capabilities of `Spectre` to consider the individual boundaries of the mesh become very useful. Recalling Equations (2.4) and (2.5), it is clear how restricting the mesh and intersecting restrictions makes the computation a very easy task. In fact, the actual implementation of the functions that yield the

values of the distortion indexes are simply a clever combination of the methods provided by the other modules of `Spectre`.

This shows once again that our effective code design saved us a good share of effort later on.

Concerning the spectral analysis module, two main classes are defined, `Pod` and `Dmd`. They naturally represent the technique with the same name and have a similar design. Their main task is to interface data from a `Dataset` object to PETSc and SLEPc matrices. In particular, these classes take care of assembling the local portions of the matrices used in the spectral analysis computation and call the appropriate routines to perform the actual matrix operations.

Methods are also provided to output the principal values (resp. dynamic eigenvalues) and to save the orthogonal modes (resp. dynamic modes) to disk in HYDRA format.

5.5 Validation of POD and DMD

Before going into the spectral decomposition of the S-duct, we present test cases for POD and DMD routines that we used to validate `Spectre`. The test cases are also useful as they give insight into what to expect and how to interpret the results. As we will see, this is not always trivial.

5.5.1 POD validation

The best way to show the behavior of the Proper Orthogonal Decomposition is to show the dependence of the result on the type of input data it gets. The POD tries to describe the given data in the best way (in the least-squares sense) by projecting on rank-1 subspaces. Consequently, it will yield a better approximation if the input data's information content is, in a sense, structured. If there is an underlying pattern, or correlation, in the given data then the POD technique will identify it in the first modes, which will have a higher associated singular value. If, however, the input data is random (uncorrelated), then no pattern will be found and all the modes will have a similar singular value, thus carrying a similar contribution to the field reconstruction.

Let us study an example. First of all, we consider a set of uncorrelated, two dimensional data sampled from a uniform random distribution. We take 200 samples for the x coordinate and 100 samples for the t coordinate. We recall that there is no distinction between time and space in POD, because this technique does not take into account the *nature* of the dimensions.

Performing the POD on this data gives the results shown in Figure 5.3. As we can see, the input data does not have any particular structure; ac-

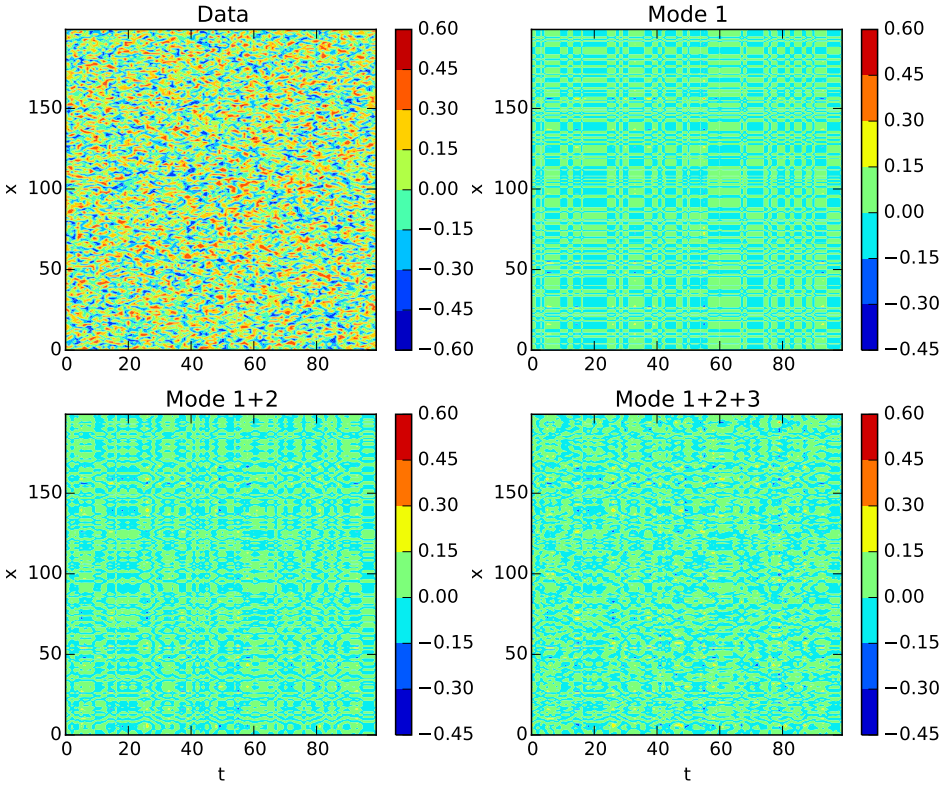

Figure 5.3: POD decomposition of uncorrelated data: $\mathrm{f} \sim \mathcal{U}$.

σ
6.933 879
6.609 005
6.597 251
6.556 410
6.319 744

Table 5.1: Values of some singular values for the POD of uncorrelated data.

σ
56.463 529
12.495 285
0.036 736
0.036 507
0.035 672

Table 5.2: Values of some singular values for the POD of structured data.

cordingly, the POD procedure is unable to find any underlying pattern or trend, and the resulting modes are all equally important, in the sense that their value is similar to each other's; the values for this validation case are presented in Table 5.1. Because of this, the reconstruction with one mode is not really representative of the input data, and the reconstruction with three modes is not decidedly better than that with one or two modes. In fact, in this example many orthogonal modes are required to get to a satisfactory approximation of the input data.

Let us see what happens when the input data has a structure: in this second test case we consider a generic function $f(x,t) = g(x)h(t)$. Performing a POD on this input data yields a drastically different result (Figure 5.4).

Now the first mode is already a qualitatively good description of the given data, and the second mode adds the missing details. Adding more modes does not appreciably contribute to the reconstruction of the input data. Accordingly, the singular values associated with the modes are all but equal, as shown in Table 5.2 What happened here is that the POD technique was able to identify an underlying data structure and expressed it as the first mode.

A convenient feature of the POD technique is that the modes it outputs are real fields with a clear interpretation, even in the so-called eye-norm, meaning that the result can be visually inspected to have an idea of its quality. This is important in view of an *online* post-processing setting in

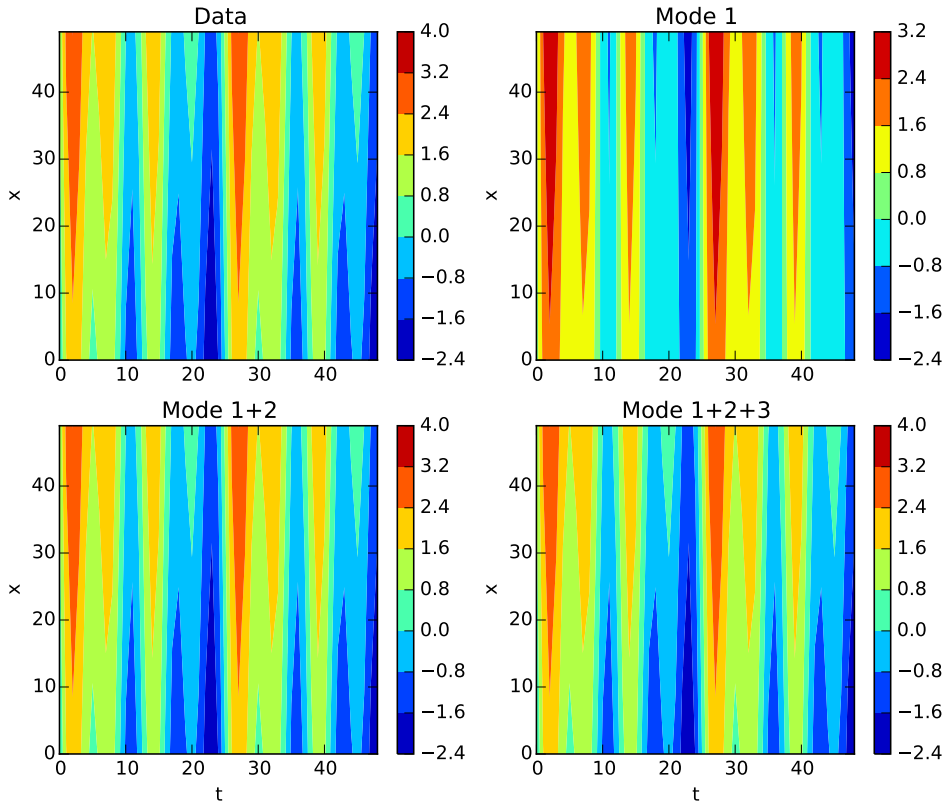

Figure 5.4: POD decomposition of structured data: $f(x,t) = g(x)h(t)$

λ
0.001721
$-0.001996 \pm 50.062431i$
$-0.405412 \pm 79.099868i$
$-1.774495 \pm 43.043489i$
$-1.908716 \pm 86.493690i$

Table 5.3: Eigenvalues for the DMD test case.

which the scientist running the simulation can *look* at the simulation – or some quantities derived from it – in real time.

From this example we see what we hope to achieve by applying this decomposition technique to the flow field variables for the simulation of the flow in the S-duct, namely the identification of coherent structures as orthogonal modes, which we might later use in various ways.

5.5.2 DMD validation

As we stated above, the DMD technique identifies dominant modes by their frequency, its idea being to express the given field as a superposition of oscillating fields, each with its own amplitude and frequency, which are given by the associated eigenvalue.

To test `Spectre` on the Dynamic Mode Decomposition, we chose again a two dimensional, space-time sample data input. This time space and time *do* matter and are not interchangeable.

We sample from the function

$$u(y,t) = y^2 + y\cos(50t) + \cos(80t) + \mathcal{U}(-0.005, 0.005), \tag{5.8}$$

Which is the sum of a parabola, and two oscillating parts with angular frequencies of 50 and 80. To make things more challenging, we add a small perturbation in the form of a uniform distribution over the y domain.

Performing a DMD on the sampled data yields the results shown in Figure 5.5, with eigenvalues shown in Table 5.3.

First of all, we note that the zeroth mode in Figure 5.5 has zero imaginary part. This is *always* the case because the zeroth mode represents the mean flow, and in the figure it is, accordingly, a parabola.

What is important here is that the DMD procedure correctly identified the frequencies in the input data in the first two modes (cfr. the matching colors in (5.8) and Table 5.3). We know that those are the frequencies we have to look at because they correspond to the eigenvalues with greater

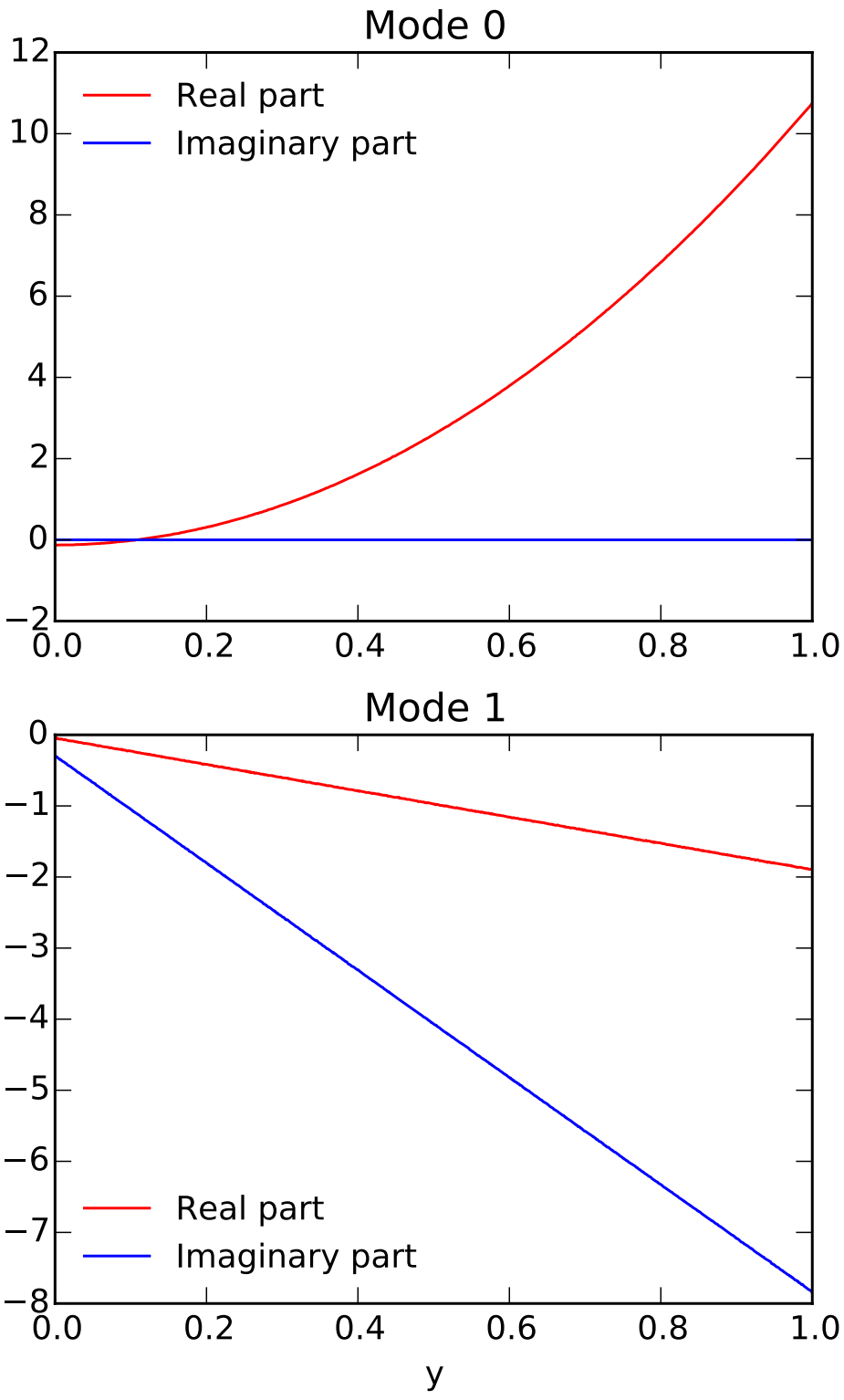

Figure 5.5: DMD modes from structured data.

(less negative) real part. Since modes decay as $\sim e^{\lambda t}$, we see that the first two eigenvalues are associated to more important modes, which we know are the ones in the generating function. The other ones are simply a result of the noise in the generating function and the numerical error, but they are clearly identified by their lower real part.

We also note that the first eigenvalue is slightly positive. Strictly speaking, this should not happen as the associated mode is unstable. It is known that this can happen due to numerical approximation error, but the effect does not compromise the procedure as long as the eigenvalue is small and the time interval is bounded.

5.6 Performance and scalability analysis

In this section we spend a few words on the measured performance and scalability of `Spectre`. First of all, the setting: `Spectre` is supposed to run as a parallel code on distributed memory clusters, possibly exploiting coprocessors when available. Keeping this in mind, we tested both strong and weak scalability on both CPU and GPU. The timing was taken for the computationally intensive parts of the execution only, that is the execution of a spectral decomposition. We correctly ignore input/output operations when calculating scalability, also because when `Spectre` will run *online* there will be little to no disk access.

We briefly recall the definitions of strong and weak scalability of a code. Strong scalability is the ability to scale the execution time of a problem of a *fixed* size as the inverse of the number of processors involved in the computation. This ability is measured by the ratio of the execution time on n processors to the one on a single processor for a given problem. A code that scales strongly runs on n processors taking approximately $\frac{1}{n}$ times the time it takes to run on one processor. Weak scalability is the ability to maintain a constant execution time as the problem size scales like the number of processors involved in the computation. This ability is measured by the inverse ratio of the execution time on one processor to the one on n processors for a problem n times as big.

For the benchmark we are going to present, we performed a POD on a sample matrix of size 50000×100 as a base case, rescaling it appropriately for the weak scalability test.

Due to the lack of hardware, we only had access to four GPUs during the testing, so we present results for scalability up to four computing nodes. The point of this is that we are more interested in a comparison between scalability on CPUs and on GPUs, rather than in a generic scalability test for CPUs, because `Spectre` mainly relies on PETSc, whose scalability properties for CPUs are well known and discussed in countless works in the literature.

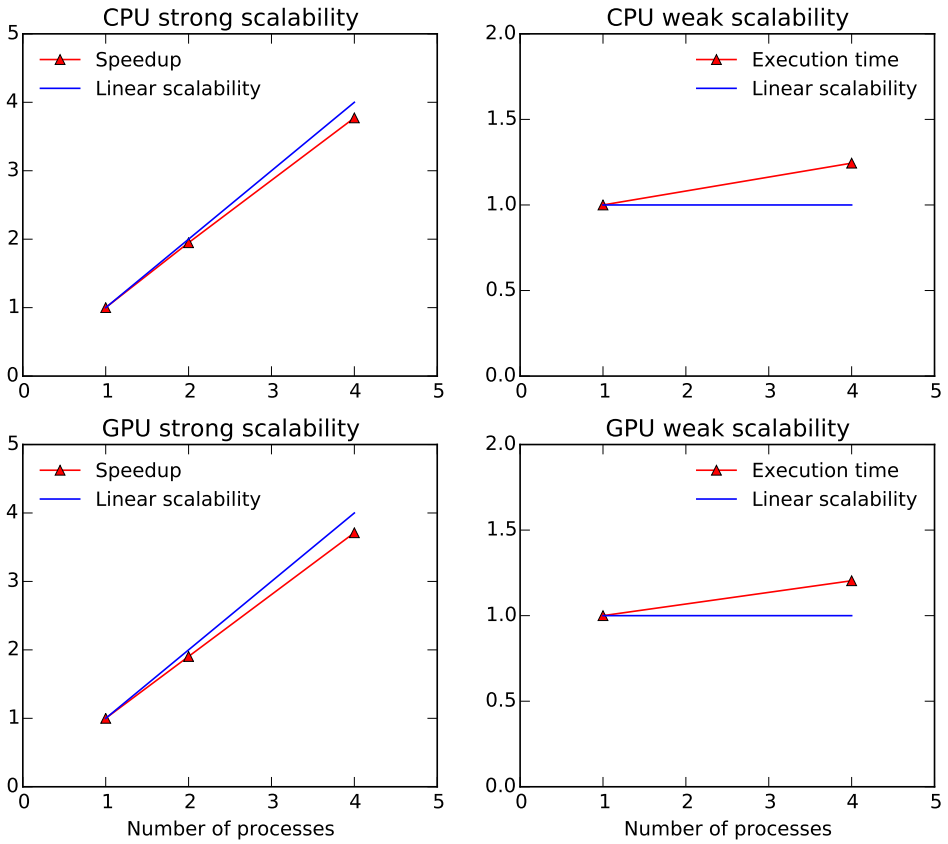

Figure 5.6: Strong and weak scalability results for the implemented code on benchmark cases.

The results of the scalability tests are shown in Figure 5.6. We can see that the scalability performance is very similar in the two cases. This is very important because, when using GPUs, an additional bottleneck appears in the pipeline, namely the CPU-GPU bus memory bandwidth. The fact that the scalabilities are very similar tells us the good news that said bottleneck is not negatively affecting the performance of the code. The benefits of using coprocessors are visible in the actual execution time of the individual simulations, with computations on GPUs being, for the present benchmark case, about 40-45% faster than the CPU counterpart.

5.7 Issues and improvements

Although all the tests we ran were satisfactory, we do not fear to admit that there is still room for improvement. One major issue is the fact that the matrices handled in the computation, although dense matrices, are

used and memorized as sparse matrices. This defect is due to the lack, at the moment of writing, of support in PETSc/SLEPc for parallel dense matrices.

Quantitative tests with a single core showed, however, great performance benefits when running with dense matrices, yielding a 40-60% time gain when compared to a single core with sparse matrices. A good feature of PETSc is that switching between dense and sparse matrix format is just a matter of changing a single line of code, and can even be done at runtime, without the need to recompile the code. This means that if a later version of PETSc provides better support for dense matrices on CPUs and GPUs, then little to no code modification will be required and `Spectre` will be able to exploit its full potential.

5.8 Spectral analysis results

For the spectral analysis of the LES flow in the S-duct we considered a fully developed turbulent flow. First of all, we let the simulation run for enough time for the air to travel from the inflow section to the outflow section, that is, completing a flow-through of the duct. This was necessary in order to get rid of the initial condition of the flow, which was the RANS solution of the same problem.

We run 2500 time steps of the simulation, discarding the initial 800 ones because of the reason just explained, and then performed a spectral decomposition on 200 time steps, from time step 1400 to time step 1600.

For comparison reasons, before presenting the results of the spectral decomposition we want to show flow visualization and investigation by means of more traditional techniques. We recall that we are most concerned with the identification of flow features and shape optimization, so we will focus on applications to these topics. In particular, in the simple flow we are dealing with the main feature to identify is the separation bubble; separation itself provides the link with shape optimization, as introduced in Section 2.4 and discussed in more detail in Chapter 6. Plus, since the most interesting part of the flow is the one on the first bend, we restricted the spectral analysis to a portion of the S-duct comprising the bend, but not the straight parts.

First off, the most basic visualization technique is the plot of contours of the velocity; an example of this can be seen in Figure 5.7. The velocity contours hint at the fact that there is a separation on the bend, but we are able to deduce it simply because we already know the separation is there; moreover, this plot does not identify vortex cores. This is a well known result, as vortex cores are not tied to the velocity magnitude, but rather to the vorticity.

For this reason a better criterion, used in a number of works in the

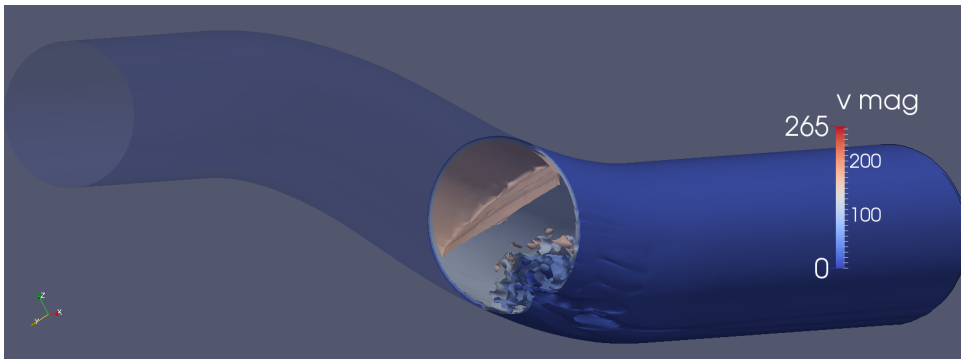

Figure 5.7: Velocity contours for the LES flow in the S-duct. Little information can be deduced from this plot, which suggests the presence of a separation to the trained eye alone.

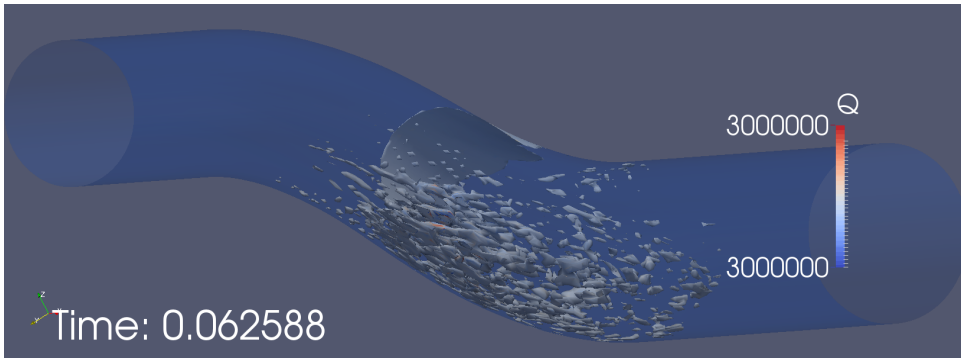

Figure 5.8: Identification of coherent structures in the S-duct with the Q-criterion. Vortex cores are easily identified by the contours, but no information is available on possible features of the flow field such as the separation bubble we are interested into.

literature, is the Q-criterion introduced in (5.1). We show a plot of the coherent structures identified by this technique in Figure 5.8. The Q-criterion is able to identify vortex cores and turbulent structures in the given flow. This ability is commonly exploited in many papers, and in fact the Q-criterion is, by now, standard practice. We shall now discuss how spectral analysis can improve on this.

First of all, let us consider the first POD mode of the velocity magnitude, shown in Figure 5.9 The first POD mode is often said to represent the mean flow. This is intuitively correct, but to be more precise we should say that its time average is the mean flow, because of course it cannot be both the mean flow *and* time-dependent. This issue is trivially solved, however, by recalling that all POD modes are rank-1 fields, which means that the flow fields at any two times t_1 and t_2 are multiples of

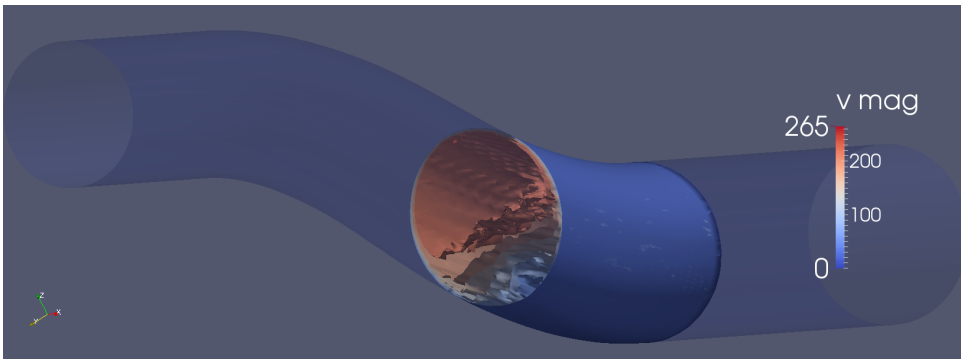

Figure 5.9: First POD mode of the velocity magnitude. The iso-surfaces are neatly organized in layers that reveal the presence of the separation bubble on the first bend.

each other, thus the time average is again a multiple of the POD mode at any fixed time interval. We can therefore conclude that the flow field portrayed in Figure 5.9 is a multiple of the mean flow. In particular, this means that we have recovered all the information we could have gotten from a RANS simulation, which yields the mean flow, in the first POD mode.

Let us now take a closer look at it: unlike in Figure 5.7, the iso-surfaces are neatly stratified, and this is a clear sign that the first mode is indeed identifying the separation bubble that characterizes the turbulent flow in the S-duct. Since separation is the main feature of this flow field, it is correct that we found it in the first POD mode, as it clearly contains most of the field energy.

The fact that POD does not give us the mean flow directly, but rather a time-dependent field, has an important consequence: it allows us to see how the separation bubble grows or reduces as time passes. Recalling the reconstruction formula (5.4), we can of course express the flow at a given time as the superposition of the modes; the first POD mode is thus telling us that the flow field is given by the superposition of the separation bubble *plus* further contribution given by the other modes. The fact that the decomposition is orthogonal guarantees that the other modes do not influence the characteristics of the separation bubble as expressed in the first mode.

We would like to point out how neither the velocity contours (Figure 5.7) nor the Q-criterion (Figure 5.8) are able to identify so clearly the structure of the bubble and, even more importantly, its size.

The size of the separation bubble is indeed a very useful piece of information in view of the original problem we are concerned with, shape optimization. In the preliminary analysis we carried out in Chapter 4 we clearly identified (see Figures 4.2, 4.4 and 4.5) separation as the main

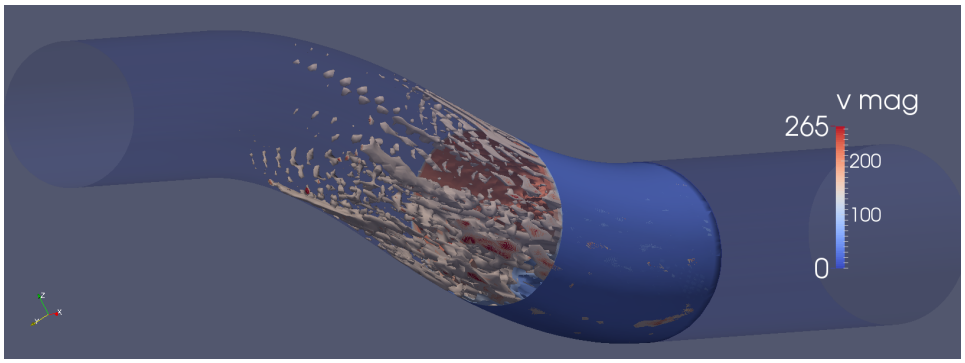

Figure 5.10: First and second POD modes. No more global flow features are present, so the POD falls back on identifying the local vortex cores, similarly to what the Q-criterion does.

cause of the inhomogeneities in the flow at the outflow surface. Accordingly, we might be tempted to conclude that reducing the size of the separation bubble would result in reducing the inhomogeneity at the outflow. This hypothesis is not rigorously motivated by theoretical arguments, but we deem it reasonable. If we accept this hypothesis, then it is clear how convenient it is to be able to measure the size of the separation bubble in the first POD mode: we could indeed use this number as a cost functional to drive the shape optimization procedure. The time-dependency of the modes allows us to fine-tune this cost functional even more: we could in fact decide to minimize the time-averaged size of the bubble or its maximum size, just to name a couple.

We shall now consider the second POD mode. Figure 5.10 shows a superposition of the first two POD modes of the velocity magnitude. As we were expecting, the second POD mode does not identify a particular flow pattern, but instead only the vortex cores. This is due to the fact that the flow at hand presents only one macroscopic (global) feature, namely the separation, and then the following modes fall back on identifying small, local features such as the vortex cores.

This behavior is similar to that of the Q-criterion shown in Figure 5.8. An important difference is that the Q-criterion does not distinguish between more and less energetic vortex cores, but simply gathers them all together. Another important remark is that if the Q-criterion is applied to the instantaneous flow field, then the vortex cores it identifies can appear and disappear with time; conversely with POD, since the modes are rank-1 fields, the vortex cores undergo, at most, a scaling, but their shape and structure remain otherwise unchanged.

An interesting feature of the identified vortex cores is that they are clustered in the lower part of the bend, which is where the separation bubble is. We claim that this is not a coincidence, but rather that the

recirculation due to the separation bubble is reflected in the presence of these structures, identified by the second POD mode. This property allows us to define another possible cost functional for the shape optimization using the volume of these vortex cores: minimizing the total volume of the vortex cores would reduce the turbulence in the separation, and this might help making the outflow field more homogeneous. Furthermore, reducing the vortex cores would also positively affect the pressure loss, which is another important factor in the design of a good S-duct.

POD modes beyond the second one look similar to it, and in fact they identify more vortex cores based on their energy content; accordingly, they may be included in the definition of a cost functional aimed at reducing the vorticity content of the flow field.

Another possible use of spectral analysis is in the definition of a stopping criterion for unsteady flows. One problem with unsteady flows is in fact in understanding when the simulation is converged. Since many configurations do not have stable steady solutions, the simulation eventually reproduces an unsteady, typically periodic, behavior, so the notion of convergence is to be intended in a statistical sense. The idea is then to monitor the values of the singular values yielded by POD (or the eigenvalues given by DMD) and claim that the simulation is fully converged when a certain number of these values are time-converged. This approach was tried for POD, see Ahmed et al. [AB04], who found it successful with the clause that the correct number of snapshots has to be chosen, taking into account some features of the flow to be simulated such as its main frequencies.

The definition of a stopping criterion based on the singular values (or the eigenvalues) poses the new question of how many values should be considered. The answer comes again from the physical meaning of the modes, as described above. Once it is understood that individual modes represent individual features, one can decide that the simulation is converged when some, or all, of the macroscopic flow features are converged. In our case, we could decide that a satisfactory approximation of the separation bubble is sufficient for the simulation to be considered converged; in case we are using the second mode as a cost functional, we might want to ask that the second mode is converged as well. In the case of POD, a second option is to consider the value of the singular values: since they are proportional to the energy content associated with the mode, it is possible to estimate how much energy of the flow is being approximated and choose the modes to track accordingly*.

All the above remarks are not exclusively related to POD: DMD shares a good deal of properties and most of what we said applies to this technique as well. It should be noted that, as described in Subsection 5.2.2,

*The total energy can be estimated knowing that the singular values are decreasing.

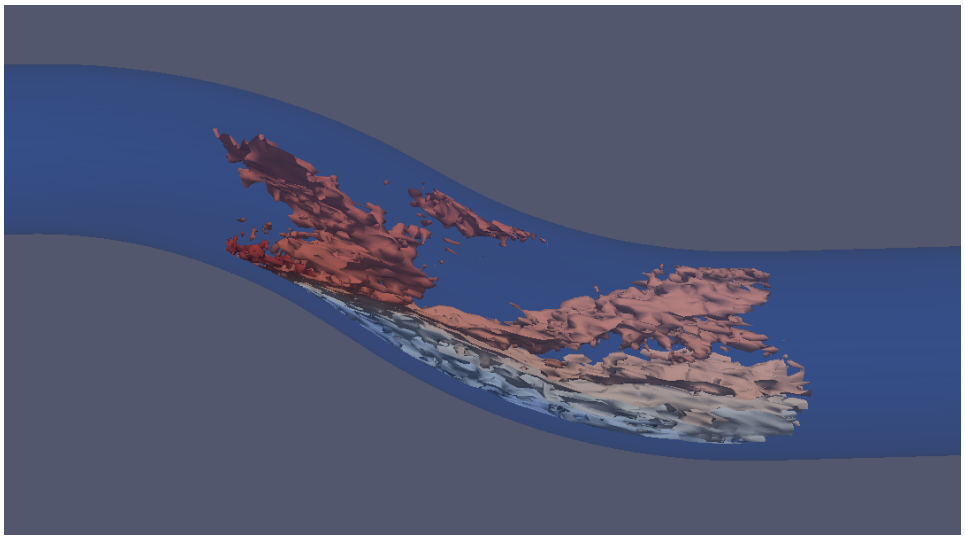

Figure 5.11: Real part of the first DMD mode of the velocity magnitude. Stratification of the contours identifies the separation bubble.

this technique is less trivial to setup, so the quality of the results can depend on procedure parameters such as the number of snapshots, and in general on some *a priori* knowledge of the flow to be simulated; we present the best visualizations we were able to get, tuning the DMD parameters basing on experience. Let us consider the first real mode of the velocity magnitude given by the DMD decomposition, which is shown, up to a scaling factor we do not really care about, in Figure 5.11 (recall that the imaginary part of the first mode is identically zero). Again, we can see that the iso-surfaces form some layers that identify the separation bubble, so the Dynamic Mode Decomposition is also able to extract the main feature of the flow.

Higher order modes are again very similar to each other, according to the fact that no other macroscopic structures are featured by the flow. As an example, we show the real part of the third DMD mode, which we present in Figure 5.12. Similarly to the second POD mode in Figure 5.10, the third DMD mode shows concentration of vortex cores in the lower part of the bend, which is in fact where the velocity of the simulated flow field is *lower*. Again, we can give the same interpretation as in the case of POD modes and use these vortex cores to define a cost functional for our shape optimization problem.

For the case of the flow in the S-duct, we can therefore conclude that spectral analysis proved a very powerful tool able to identify both the most important global flow feature and the secondary, local vortex cores. Proper Orthogonal Decomposition and Dynamic Mode Decomposition yield, to some extent, comparable results, in some sense validating each other.

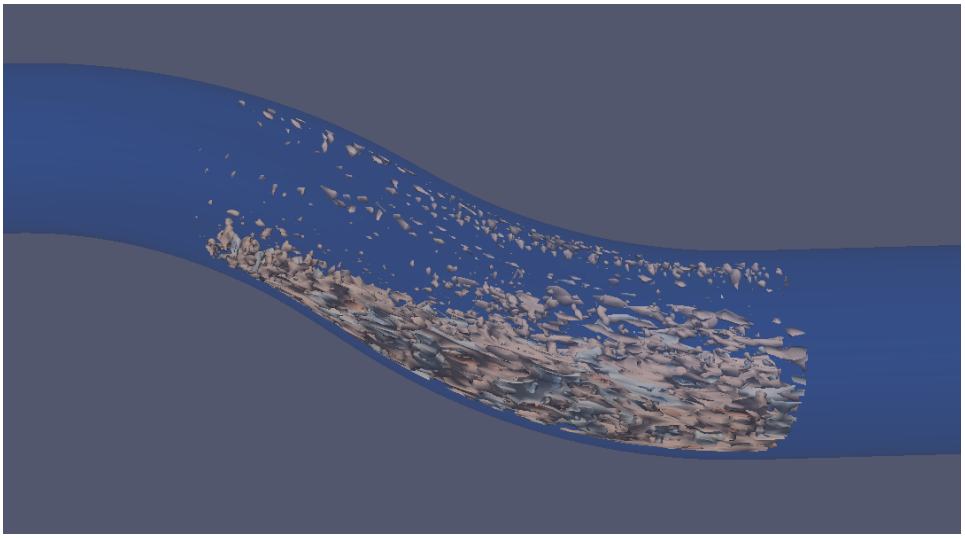

Figure 5.12: Real part of the third DMD mode of the velocity magnitude. Vortex cores in the separation bubble identify the most turbulent part of the flow.

Chapter 6

Optimization

6.1 Shape optimization

Let us now turn to optimization part of this work. In particular, we want to solve a shape optimization problem in which we seek an optimal shape for the S-duct such that the flow field at the outflow interface is as uniform as possible, for the reasons we discussed above.

For a shape optimization procedure we need some ingredients. One main ingredient is the PDE $F(u) = 0$ describing the phenomenon at hand; later, a base configuration is needed as a starting point for the procedure. The search domain is equally important: this is a set $\mathcal{U}_{ad}$ of admissible shapes where we look for an optimal solution to the shape optimization problem. A cost functional $J(u(\Omega))$ is the driving function that allows to tell when a solution is better than another. Finally, we need an algorithm that actually makes the choice of the next candidate shape Ω at each iteration of the procedure.

Some of these ingredients are easy to deal with. For the PDE, we resort to the standard compressible Navier-Stokes model, modeling turbulence with Menter's k–ω SST RANS scheme. For the base configuration; we will be using the wellborn S-duct that we mentioned above.

Concerning the cost functional, for this test case we will use the distortion indexes RDI and CDI defined in Section 2.4, together with the pressure loss from inlet to outlet. Alternative possible choices of the cost functional will be discussed later.

The definition of $\mathcal{U}_{ad}$ is definitely less trivial. Ideally, we want to give enough room for changes to have a big enough set $\mathcal{U}_{ad}$ in which to search, at the same time without introducing too many parameters, because the computational cost is proportional to the dimension of the parameter space to be explored. In doing this, we must keep in mind the geometrical constraint that it must not be possible to see the outflow surface from the inflow one, to safeguard the stealth capabilities of the

plane. We will properly address this issue in Section 6.2.

The algorithm that performs the choice of the candidate is another tricky point. Since we recast the shape optimization procedure as a multivariate real optimization problem, we can exploit the huge literature available on the topic. We will discuss the optimization approach in Section 6.3.

6.2 Free-Form Deformation

Here we deal with the problem of the definition of the set of admissible shapes $\mathcal{U}_{\text{ad}}$. We said we want to have the shapes controlled by a set of real parameters, so we want to be able to deform the S-duct parametrically, while keeping it a valid S-duct shape.

A tool that lets us tackle this problem is a technique called Free-Form Deformation, or FFD for short. Free-Form Deformation originates a computer graphics technique and was first described in Sederberg et al. [SP86].

The idea behind FFD is to enclose the geometric object to deform inside a cube, or another hull solid, and to deform this latter shape. As a result, the enclosed shape deforms accordingly. FFD can be used with any solid modeling system, for example Computational Solid Geometry, and it can deform surface primitives of any type or degree. The deformation can be local or global; in the case of local deformation, it is possible to enforce continuity of derivatives up to an arbitrary degree.

The tool by which we implement Free-Form Deformation is again BoxerMESH. BoxerMESH allows to import CAD files in various formats and to manipulate them in various ways, including of course Free-Form Deformation, before meshing them and exporting the mesh. Moreover, BoxerMESH is fully scriptable in Lua, and this provides the level of automation we will need to integrate it in an optimization loop.

Let us now describe the FFD setting for the S-duct. The Free-Form Deformation algorithm requires one or more convex solids to enclose the shape that should be deformed. A three-dimensional array of cubes works just fine, and we call it a lattice.

For the deformation of the S-duct, we defined the lattice shown in Figure 6.1. This is a simple lattice of $4 \times 3 \times 3$ cubes that surrounds the curved part of the S-duct. In other words, we leave the straight parts unchanged and only influence the curved part of the duct.

Next, we have to choose the degrees of freedom of our FFD, that is we have to define what transformations of the lattice are admissible. For this test case, we decided to enforce continuity of the duct wall (this is obvious) but not of the first derivative of the shape. In other words, sharp edges can form where the straight parts connect to the curved one;

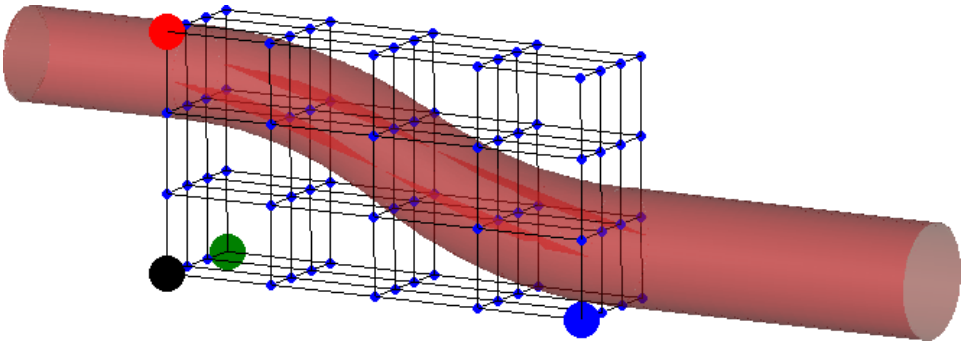

Figure 6.1: The lattice defined to apply Free-Form Deformation on the S-duct.

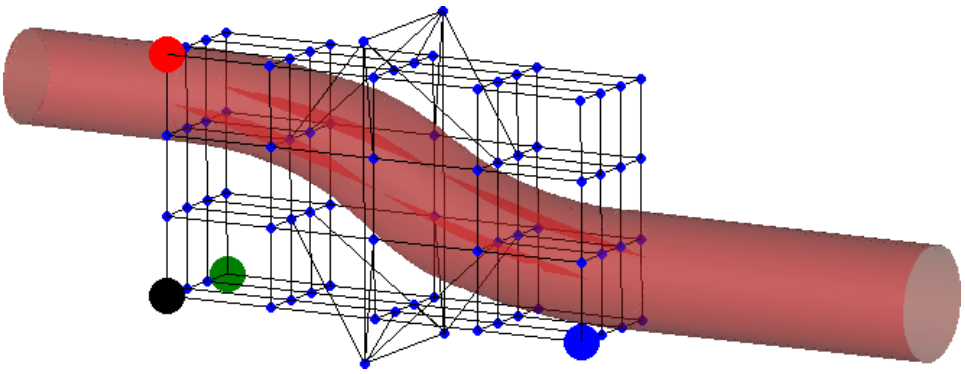

Figure 6.2: Example transformation of the FFD lattice. Here the second x-normal plane is scaled.

we decide not to worry about this because more regular shapes are still admissible and we believe the optimization procedure will recover them in case they are optimal.

Speaking of the actual transformations, we allow the internal nodes of the second, third and fourth x-normal planes* to scale with respect to their plane's center and, later, to translate in the xz plane. In particular, we enforce the plane symmetry of the duct with respect to the xz plane. We show an example of one such transformation in Figure 6.2.

The total number of degrees of freedom results to be twelve, accounting for six scaling parameters (the planes can have different scale factors in the y and z directions) and six translation parameters (each plane translates in x and z independently).

As far as the transformation ranges are concerned, all scaling parameters are allowed to vary in $[0.1, 3]$, whereas translations are admissible if in the range $[-200, 200]$, where the unit of measure for translations

*Recall that the flow is in the x direction.

is millimeters. As a result, we have now recast the shape optimization problem as an optimization problem in $[0.1, 3]^6 \times [-200, 200]^6 \subset \mathbb{R}^{12}$.

We would like to point out that all these parameters are independent of the procedure, and all the design decisions we made are arbitrary. For example, we could have chosen a finer lattice or different transformations with different bounds, and everything would have been the same, just more complex. For the purpose of our work, that is intended to be a proof of concept of the feasibility of the technique, we decided to limit the complexity to a manageable one. As we will see later on, the results are still satisfactory for the given test case.

6.3 Smart Optimisation For Turbomachinery

As anticipated, our shape optimization problem is actually a multivariate real optimization problem. For this kind of problems many techniques are known and the literature is very vast. Popular families of techniques include gradient (steepest descent) methods as well as genetic algorithms.

All these methods are well understood and their application is straightforward. It is therefore natural to expect to find them already implemented in some application.

In Rolls-Royce, a good deal of effort is put in the development of a nice piece of software called Smart Optimisation For Turbomachinery, or SOFT for short. SOFT is an abstract interface to a general optimization solver in $\mathbb{R}^n$ that allows for a general specification of an optimization problem.

The way it works is as follows. A configuration wizard takes the user through the problem specification, allowing him or her to define the number of objectives (cost functionals) that should be optimized, the parameters that span the parameter space, their domain of definition and range of admissible values, the simulation to be run and the optimization algorithm.

What is remarkable about this setup is the abstraction level allowed by the software. The simulation to be run can be any arbitrary collection of scripts, allowing for great generality of the problems that can be solved. Similarly, the cost functional is implemented as a list of numbers read from some user-defined text files, regardless of the way they are computed; this means that very different cost functionals can be used so long as they output their value to some text file that SOFT can read afterwards.

The optimization algorithm can be changed without touching the rest of the configuration: from quasi-Newton to genetic algorithms it is just a matter of changing a setting in a dialog. This high level abstraction layer allows for easy testing of several different optimization strategies.

We used SOFT for the optimization of the shape of the wellborn S-

duct, but there is still a missing piece to the puzzle: the parameter space has an unknown shape.

6.4 Design of Experiments

The shape of the parameter space is another problem that shall be taken care of. Let us try to state the problem itself in a more precise framework.

We have a phenomenon, in our case the distorted outflow field, and some parameters that we are able to tune to try to influence it, in our case these are the twelve FFD parameters. How can we assess the influence of each parameter on the phenomenon? Perhaps the phenomenon is very sensitive to some of them, perhaps some others are completely irrelevant.

In a trivial setting, if we had a phenomenon determined by the toss of a coin and the roll of a dice, then we could easily explore the whole set of possibilities, comprising $2 \times 6 = 12$ possible configurations. If we had a hundred dice, then the total number of configurations would be 6^{100}, and if instead of a dice we had a parameter that can take values in a continuous range, then the same naive formula does not even make sense anymore.

Clearly, we need a clever way to identify the influence of a parameter on the outcome of an experiment. This is the problem of assessing the shape of the parameter space in our shape optimization procedure.

Even if we exclude the case of continuous ranges of values by appropriate quantization of the ranges, having p parameters, the i-th of which can take n_i values, still results in

$$\prod_{i=0}^{p} n_i = \text{very big}$$

possible configurations.

It is clear that this combinatorial approach is the only one that can give *full* information on the influence of the single parameters on the outcome of the experiment, but luckily several methods are known in the literature, under the theory of sensitivity analysis, to greatly reduce the computational cost by sacrificing little information. Notable examples of such methods include the Latin hypercube method or the LP$-\tau$ method [Sal+04].

All these methods are sampling methods: they try to answer the question of how to choose a limited set of parameter values in order to extract as much information as possible on the sensitivity of the target experiment on the single parameters.

The technique we adopted for our study case is called Design of Experiments, and was first introduced in the 1926 paper, republished in 1992

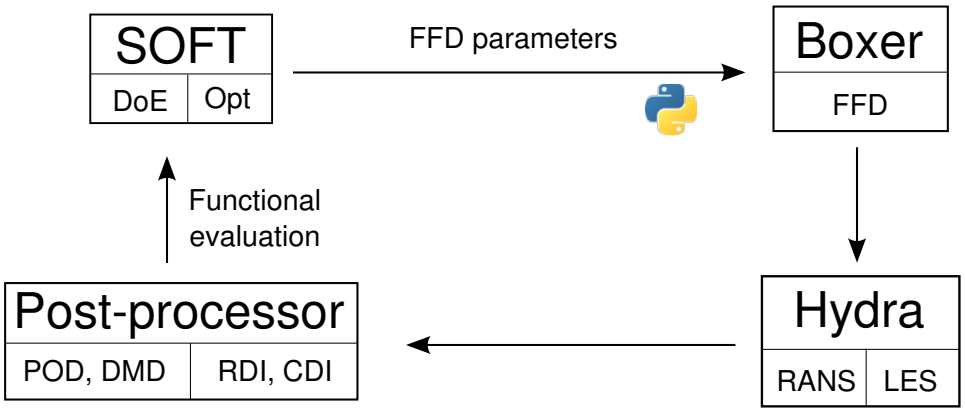

Figure 6.3: A representation of the cycle that we implemented to perform the shape optimization process.

in Fisher [Fis92], and later expanded in the 1935 book Fisher [Fis35]. Although it was originally intended for applications in agriculture, the theory was later successfully applied to a much wider variety of cases.

A strong feature of the Design of Experiments approach is that it is able to explore a parameter space choosing the optimal n tuples of values of the parameters. In other words, Design of Experiments is able to choose the best n configurations, where n is a user defined parameters, to shape the unknown parameter space. This feature has a very strong implication: since n is typically the number of experiments we are able to perform, Design of Experiments allows us to fix *a priori* the computational effort we can afford for the whole shape optimization procedure and obtain the best result with that fixed cost. Not only this: as the choice of the n tuple is optimal, the optimal $n+1$ tuple is the optimal n tuple plus a new entry, the $(n+1)$-th one. This means that if we *later* decide that the computed result is not accurate enough we can still increase the number of experiments *without* having to re-run the previous simulations. This is a very powerful feature of Design of Experiments that allows for a fine tuning of the required computational cost and, equally important, to setup an iterative procedure.

6.5 Summary of the optimization procedure

Now that all the pieces are in place, we can give a final picture of the optimization procedure. We will focus specifically on a practical description, describing the algorithm step-by-step. The optimization cycle is sketched in Figure 6.3; let us start from the top left corner.

The problem is defined within SOFT, meaning that the optimization parameters are identified, their range is specified and the application is

told where to read the cost functional from. Later on, SOFT, which implements Design of Experiments, generates a list of n tuples containing the values of the optimization parameters, which in our case are the Free-Form Deformation parameters, to be used in n independent experiments in order to explore the parameter space.

With the aid of a little bit of Python scripting, the output file produced by SOFT is parsed and a folder tree is generated to accommodate n folders, each one for a separate simulation; in each of them, a different Lua script is generated, containing the FFD parameters BoxerMESH will use. BoxerMESH is then run with each of these parameter sets, generating the corresponding mesh.

Pre-processing tools are run on these meshes, which are prepared to be eventually fed to Hydra, the CFD software we used. Hydra runs n independent simulations and writes its own output files, which are later read by `Spectre`.

`Spectre` reads in the flow files and computes the quantities designed to be used as cost functionals by SOFT. For example, we could have Hydra run RANS simulations and `Spectre` compute the distortion indexes and the pressure loss, or we could have Hydra simulate an LES and `Spectre` compute some more sophisticated cost functional based on spectral decomposition.

After the functional evaluation is completed, it is SOFT's turn once again. This time SOFT reads in the values associated with the n tuples it generated earlier and uses them to build an approximation of the parameter space in the form of a hyper-surface. SOFT supports various kinds of reconstruction techniques, including linear and cubic polynomial interpolation.

At this point, SOFT runs one of its several built-in optimization techniques and finds the optimum of the approximated hyper-surface. It is important to point out that in this part of the optimization procedure no CFD simulation is run, accounting for a very low computational cost.

The optimal values for the FFD parameters are then written to disk by SOFT, they are passed to BoxerMESH, an optimal geometry is generated and Hydra is run on it. Running `Spectre` on the resulting flow field finally gives a functional evaluation of the optimized solution, allowing to assess the actual improvement over the starting configuration.

As a stopping criterion, either the absolute figure of the optimized cost functional or its increment relatively to the basic configuration can be used. In case the improvement is satisfactory, the loop can exit, otherwise the user can choose a second number of experiments $m > n$ and start over with more points to evaluate in the Design of Experiments. Again, in this case only $m-n$ more CFD simulations have to be run, and the previously run simulations are not lost, but rather recycled.

Another interesting point of this procedure is that the n runs of Box-

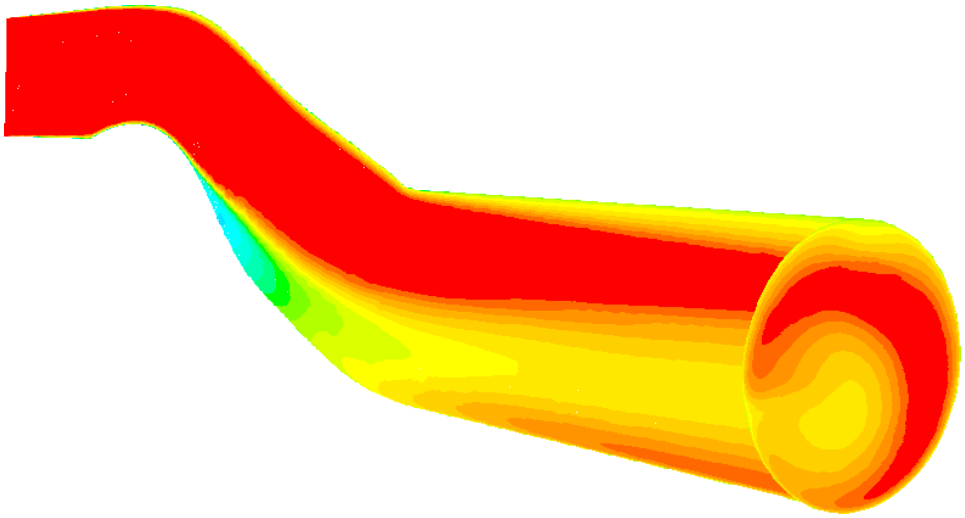

Figure 6.4: Total pressure for the flow in one of the configurations given by SOFT. Green is 1.01×10^5 Pa and red is 1.07×10^5 Pa.

erMESH and Hydra, for the meshing and CFD simulation, respectively, are *completely independent*; In jargon, that part of the loop is said to be *embarrassingly parallel*, meaning that it can be run on a number of cores, expecting the total run time to scale linearly. To be precise, this property holds up to n parallel processing units, that is when each core takes care of exactly one meshing or CFD run. Beyond this limit, parallelization is still possible, but it will have to rely on the internals of the mesher and the solver, respectively. Concerning BoxerMESH and Hydra, they both show good scalability performance with at least a few cores per task.

The convenient structure of the optimization loop allows to push this limit much farther than it usually would be when running in a classic parallel fashion.

6.6 Shape optimization results

Here we show the results of the shape optimization procedure, discussing the figures and commenting on possible improvements.

We performed a single complete iteration of the optimization loop, in order to prove its feasibility and set the ball running for future work. For the Design of Experiments part we chose to perform $n = 48$ RANS simulations in order to explore the parameter space. We then set up SOFT and obtained the 48 parameter tuples, meshed with BoxerMESH and simulated with Hydra. As an example, we show the flow in one of these configurations in Figure 6.4. This particular shape shows that sharp edges are allowed and used by Design of Experiments. Even though

	Initial value	Optimized value	Improvement (%)
p_{loss}	0.068 23	0.013 98	79.5
RDI	0.177 25	−0.010 84	93.9
CDI	1.075 15	1.018 71	5.2

Table 6.1: Summary of the results of the first iteration of the optimization cycle.

counter-intuitive configurations might be explored by the algorithm, this should not give rise to any concern on the efficacy of the procedure, as it is simply DoE exploring the parameter space. If a configuration proves unfruitful, then the optimization algorithm will look in regions far from it, in the parameters space, for the optimal shape.

A note on the post-processing is in oder here: from the definition of the CDI, the RDI and the pressure loss it is clear that the optimal value for all of them is zero. Although it is not reasonable to expect that the pressure loss be negative, we cannot say the same for the distortion indexes, as their definition does not prevent them to be negative, nor there is a physical reason why they should not be. To prevent SOFT from searching the most negative values for the distortion indexes, we setup the post-processing code to output the square of the computed cost functionals. This clearly does not influence the final result, as now zero is still the optimal value for all of them, and furthermore is the global minimum of the cost functionals.

Concerning the second part, once the 48 basic configurations are simulated and post-processed, we used a technique from the family of Adaptive Range Multi-Objective Genetic Algorithms (ARMOGA) for the optimization in $\mathbb{R}^{12}$. Genetic algorithms are known to work well in high dimensional spaces, and ARMOGAs have been used successfully in aerodynamic design in the past, for example in Obayashi et al. [OS04]. In our test case we used the ARMOGA implementation in SOFT, with default parameters for simplicity. Only one thing is to be mentioned here: having three cost functionals, one might ask if they all weigh the same or some are more important than others. SOFT allows to specify different weighing for the various cost functionals to be optimized, but we chose to keep them all with the same weight. The reason for this is that any educated guess of unequal weighing factors should be motivated by a dedicated study on the effects of the singular factors on the engine performance and life, which we of course could not perform.

We show the results of the optimization procedure in Table 6.1 The results are overall very promising. Both the pressure loss and the RDI were drastically reduced. The CDI was not so positively affected by the

optimization procedure, and this is probably due to the fact that the chosen parameters are not able to effectively influence it. On the optimized RDI, we note that it is negative, but this is not an issue at all.

As far as the optimal shape is concerned, we regret Rolls-Royce declared it classified, so we cannot show it here. We can, however, give a qualitative description: the most interesting trait of the optimal shape is its enlarged mid-section – similar to the example one given in Figure 6.2 – meaning that the third x-plane underwent a significant scaling. To be more precise, the optimized S-duct's section starts enlarging on the first bend, reaching its maximum area at around half of the duct's total length, to later restrict and match the section of the outflow part.

Another visible modification to the original shape is that the mid-length section is translated towards the outflow (positive x direction) and slightly upwards (positive z direction). This implies that the curvature on the first bend is reduced, while the one on the second bend is increased.

Sharper edges than in the base configuration are also visible, indicating that smoothness of the duct shape is not a critical parameter for the cost functionals we have used.

We recall that the optimization procedure we described above treats the evaluation of the cost functional as a black-box, simply looking at the generated number without taking into account the meaning of the input or the output. This means that the transformation of the base shape into the optimal shape has, *a priori*, no physical meaning to the optimization algorithm. It is nevertheless interesting to try to give, *a posteriori*, a physical explanation of what happened.

The fact that the optimal duct's section varies so appreciably in its area implies that the flow is slowed down on the first bend and then accelerated on the second one. The change in the curvature also reveals that the first bend is now less abrupt than in the original shape, while the opposite holds for the second bend. Both these modifications seem to indicate that the optimal shape tries to help the fluid pass the first bend in the easiest possible way, to the detriment of what happens on the second one.

This behavior seems to confirm the claim we made in Section 5.8 that inhomogeneities at the outlet are caused by the separation bubble. In fact, we know from the classic backward-facing step benchmark that slower flows generate smaller recirculation eddies, so the growth of the duct's section, that slows down the flow, and the reduced curvature, that makes the bend less abrupt, might implicitly be an attempt to reduce the size of the separation bubble.

In a wider interpretation, we can deem this trend of favoring the first bend over the second one as an attempt to balance the dimensions, and thus the importance, of the main separation bubble and the second, little one that we named at the end of Section 4.2

All in all, the results we obtained are quite good. We set up a fully-featured optimization procedure to perform shape optimization of an S-duct, based on a real geometry from an industrially relevant study case. We thoroughly analyzed the implemented recipe, highlighting its strengths and discussing key features. Finally, we successfully applied the procedure to a proof-of-concept setup to show that it is actually feasible and it works. The promising results we were able to obtain with our simple test case prove that the strategy is worth investigating deeper. Possible directions for further study include the choice of the optimal parameter space, especially in view of the fact that the CDI can likely be further reduced; this means both choosing the correct type of transformations to allow on the given shape and the number of parameters to optimize (the fineness of the lattice). Different weights for the cost functionals are also of interest.

Conclusions

The present work is characterized by two goals, which we developed simultaneously during the internship.

On one front we have been concerned with spectral analysis techniques with applications in real-time post-processing of simulation data and flow analysis in terms of the identification of coherent structures. To target applications in the realm of High-Performance Computing we considered possible implementations targeting modern parallel architectures, involving both processors and coprocessors. This resulted in the creation of `Spectre`, a fully-featured application able to interface to existing Rolls-Royce software and state-of-the-art parallel linear algebra libraries such as PETSc and SLEPc. `Spectre` is designed to run both on distributed-memory parallel architectures and coprocessors, either offline, reading data from disk, or online, being integrated into another software. This tool was fundamental for all the results presented in this work.

Investigating the spectral analysis of the simulated flow fields we showed how these techniques can give a much deeper insight into the characteristics of the flow under consideration. In particular, we introduced the Proper Orthogonal Decomposition (POD) and the Dynamic Mode Decomposition (DMD) and showed how both are able to identify the main features of the flow and how they surpass a classic, still very popular technique as the Q-criterion.

On a parallel track, we also setup from scratch a complete shape optimization procedure, combining capabilities from various pieces of software. This involved defining the parameter space in terms of geometric transformations via Free-Form Deformation, its exploration via Design of Experiments and the configuration of the optimization algorithm, together with a good deal of low-level scripting to glue all the parts together.

We were able to run a loop of the complete shape optimization cycle and to report very promising results. This is valuable both as an achievement *per se*, as it provides a first, perhaps rough solution to the relevant industrial problem of the optimization of the flow in an S-duct, and as a proof-of-concept to demonstrate the feasibility of the approach and open the way to a much broader range of applications where shape optimization is needed.

These two pillars are not independent of each other, but are connected by the unifying problem of shape optimization. POD and DMD are not limited to flow description, they also provide useful information which can be turned into innovative cost functionals to drive shape optimization procedures. These cost functionals are in fact defined moving from the clearer understanding of the flow features that popular techniques cannot give. In this work we described two possible approaches: one based on minimizing the size of the separation bubble identified by the first modes, the other one based on minimizing the volume of the vortex cores identified by higher-order modes.

In our development of the shape optimization loop we were asked to use some classic cost functionals (also called distortion indexes) to drive the procedure. A possible direction for future development of this work is to try and setup an optimization loop driven by cost functionals defined via spectral analysis. The theory behind this approach has been thoroughly described in this work, but time constraints prevented us from putting it into practice.

The flow field we analyzed in this work, although relevant for the industrial application it describes, is not extremely complex if compared with, for example, the flow past a full turbine. Now that it served its purpose as a first benchmark to prove the effectiveness of spectral analysis, further developments clearly include the investigation of more complex flows in order to better understand their features.

Bibliography

[AB04] M. Ahmed and T. Barber. "POD Convergence Criterion For Numerically Solved Periodic Fluid Flows". In: *WSEAS ISA 2004 - Adv. Inf. Wirel. Commun. Syst.* 2004, p. 6.

[BB10] N. C. Bissinger and T. Breuer. "Basic Principles - Gas Turbine Compatibility - Intake Aerodynamic Aspects". In: *Encycl. Aerosp. Eng.* Vol. 1. Chichester, UK: John Wiley & Sons, Ltd, Dec. 2010, pp. 1–11.

[BCG95] D. A. Burgess, P. I. Crumpton, and M. Giles. "A parallel framework for unstructured grid solvers". In: *IFIP WG10* 3.January (1995), pp. 5–8.

[BCP97] W. Bosma, J. Cannon, and C. Playoust. "The Magma Algebra System I: The User Language". In: *J. Symb. Comput.* 24.3-4 (1997), pp. 235–265.

[Ber+12] C. Bertolli et al. "Design and performance of the OP2 library for unstructured mesh applications". In: *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 7155 LNCS (2012), pp. 1–10.

[BHL93] G. Berkooz, P. Holmes, and J. L. Lumley. "The Proper Orthogonal Decomposition in the Analysis of Turbulent Flows". In: *Annu. Rev. Fluid Mech.* 25.1 (Jan. 1993), pp. 539–575.

[BJ00] N. C. Bissinger and M. Jost. "Thrust Vectoring for Advanced Fighter Aircraft - High Angle of Attack Intake Investigations". In: *DaimlerChrysler Aerosp. AG*. May. 2000, 13(1–13).

[Cam] Cambridge Flow Solutions. *BoxerMESH*.

[CG95] P. I. Crumpton and M. Giles. "Aircraft computations using multigrid and an unstructured parallel library". In: 95 (1995), pp. 1–21.

[Cha00] A. Chatterjee. "An introduction to the proper orthogonal decomposition". In: *Curr. Sci.* 78.7 (2000), pp. 808–817.

[CP08] C. Chevalier and F. Pellegrini. "PT-Scotch: A tool for efficient parallel graph ordering". In: *Parallel Comput.* 34.6-8 (July 2008), pp. 318–331.

[CPC90] M. S. Chong, A. E. Perry, and B. J. Cantwell. "A general classification of three-dimensional flow fields". In: *Phys. Fluids A Fluid Dyn.* 2.5 (1990), p. 765.

[Eas+09] S. J. Eastwood et al. "Developing large eddy simulation for turbomachinery applications". In: *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.* 367.1899 (July 2009), pp. 2999–3013.

[Fie88] H. Fiedler. "Coherent structures in turbulent flows". In: *Prog. Aerosp. Sci.* 25.3 (1988), pp. 231–269.

[Fis35] R. A. Fisher. *The design of experiments*. Hafner Pub. Co., 1935.

[Fis92] R. A. Fisher. "The Arrangement of Field Experiments". In: *Break. Stat.* Ed. by S. Kotz and N. Johnson. Springer New York, 1992, pp. 82–91.

[GAS09] E. Garnier, N. Adams, and P. Sagaut. *Large eddy simulation for compressible flows*. Reston, Virigina: American Institute of Aeronautics and Astronautics, June 2009.

[GRM] M. Giles, I. Reguly, and G. Mudalige. *OP2*.

[Hol+12] P. Holmes et al. *Turbulence, Coherent Structures, Dynamical Systems and Symmetry*. Cambridge: Cambridge University Press, 2012.

[Jam08] A. Jameson. "Formulation of kinetic energy preserving conservative schemes for gas dynamics and direct numerical simulation of one-dimensional viscous compressible flow in a shock tube using entropy and kinetic energy preserving schemes". In: *J. Sci. Comput.* 34.2 (2008), pp. 188–208.

[KA14] P. Kalghatgi and S. Acharya. "Modal Analysis of Inclined Film Cooling Jet Flow". In: *J. Turbomach.* 136.8 (2014), p. 081007.

[Ker+05] G. Kerschen et al. "The Method of Proper Orthogonal Decomposition for Dynamical Characterization and Order Reduction of Mechanical Systems: An Overview". In: *Nonlinear Dyn.* 41.1-3 (Aug. 2005), pp. 147–169.

[KG02] G. Kerschen and J.-c. Golinval. "Physical interpretation of the proper orthogonal modes using the singular value decomposition". In: *J. Sound Vib.* 249.5 (Jan. 2002), pp. 849–865.

[KK98] G. Karypis and V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (Jan. 1998), pp. 359–392.

[Kos43] Kosambi. "Statistics in function space". In: *J. Indian Math. Soc.* 7 (1943), pp. 76–88.

[LW12] A. Logg and G. N. Wells. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by A. Logg, K.-A. Mardal, and G. Wells. Vol. 84. Lecture Notes in Computational Science and Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 724.

[MEH12] T. W. Muld, G. Efraimsson, and D. S. Henningson. "Flow structures around a high-speed train extracted using Proper Orthogonal Decomposition and Dynamic Mode Decomposition". In: *Comput. Fluids* 57 (Mar. 2012), pp. 87–97.

[Men94] F. R. Menter. "Two-equation eddy-viscosity turbulence models for engineering applications". In: *AIAA J.* 32.8 (Aug. 1994), pp. 1598–1605.

[MSK10] V. Minden, B. Smith, and M. Knepley. "Preliminary implementation of PETSc using GPUs". In: *... 2010 Int. Work. GPU ...* (2010), pp. 1–10.

[Mud+12] G. Mudalige et al. "OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures". In: *2012 Innov. Parallel Comput.* 2. IEEE, May 2012, pp. 1–12.

[Nar11] R. Narasimha. "Kosambi and proper orthogonal decomposition". In: *Resonance* 16.6 (June 2011), pp. 574–581.

[NS13] A. A. Novotny and J. Sokołowski. *Topological Derivatives in Shape Optimization*. Interaction of Mechanics and Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[OS04] S. Obayashi and D. Sasaki. "Multi-objective optimization for aerodynamic designs by using ARMOGAs". In: *High Perform. Comput. Grid Asia Pacific Reg. 2004. Proceedings. Seventh Int. Conf.* 2004, pp. 396–403.

[Pop00] S. B. Pope. *Turbulent Flows*. Cambridge: Cambridge University Press, 2000.

[Sag06] P. Sagaut. *Large Eddy Simulation for Incompressible Flows: An Introduction*. Scientific Computation. Berlin/Heidelberg: Springer-Verlag, 2006, p. 575.

[Sal+04] A. Saltelli et al. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. Wiley, 2004, p. 232.

[SC14] S. Shahpar and S. Caloni. "Automatic Design Optimisation of Profiled Endwalls including Real Geometrical Effects to Minimize Turbine Secondary Flows". In: *ASME Turbo Expo*. JUNE 2014. 2014, pp. 1–13.

[Sch+11] P. J. Schmid et al. "Applications of the dynamic mode decomposition". In: *Theor. Comput. Fluid Dyn.* 25.1-4 (June 2011), pp. 249–259.

[Sch10] P. J. Schmid. "Dynamic mode decomposition of numerical and experimental data". In: *J. Fluid Mech.* 656.July 2010 (Aug. 2010), pp. 5–28.

[Sha00] S. Shahpar. "A Comparative Study of Optimisation Methods for Aerodynamic Design of Turbomachinery Blades". In: *Vol. 1 Aircr. Engine; Mar. Turbomachinery; Microturbines Small Turbomach.* ASME, May 2000, V001T03A087.

[SP86] T. W. Sederberg and S. R. Parry. "Free-form deformation of solid geometric models". In: *ACM SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 151–160.

[SSG14] R. C. Schlaps, S. Shahpar, and V. Gümmer. "Automatic Three-Dimensional Optimisation of a Modern Tandem Compressor Vane". In: *Vol. 2B Turbomach.* JUNE 2014. ASME, 2014, V02BT39A035.

[SZ92] J. Sokołowski and J.-P. Zolesio. *Introduction to Shape Optimization*. Vol. 16. Springer Series in Computational Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992.

[vTSF09] D. von Terzi, R. Sandberg, and H. Fasel. "Identification of large coherent structures in supersonic axisymmetric wakes". In: *Comput. Fluids* 38.8 (Sept. 2009), pp. 1638–1650.

[WFV14] 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". In: *Comput. Phys. Commun.* 185.11 (2014), pp. 3028–3040.

[WRO94] S. R. Wellborn, B. A. Reichert, and T. H. Okiishi. "Study of the compressible flow in a diffusing S-duct". In: *J. Propuls. Power* 10.5 (Sept. 1994), pp. 668–675.