Université Libre de Bruxelles

von Karman Institute for Fluid Dynamics
Aeronautics and Aerospace Department

PhD. Thesis

An Object Oriented and High Performance Platform for Aerothermodynamics Simulation

Andrea Lani

April 30, 2009

Promoter: Prof. Herman Deconinck

Contact information:
Andrea Lani
von Karman Institute for Fluid Dynamics
72 Chaussée de Waterloo
1640 Rhode-St-Genèse
BELGIUM

email: lani@vki.ac.be

This manuscript was typeset on LaTeX with KOMA-Script.

All the research work was done with the GNU/Linux operating system.

Grazie alla mia famiglia per il
continuo affetto e supporto morale

Un ringraziamento speciale per Isabel,
che, con la sua infinita pazienza, mi è stata vicina
in tutti i momenti, specialmente nei più difficili

Summary

This thesis presents the author's contribution to the design and implementation of COOLFluiD, an object oriented software platform for the high performance simulation of multi-physics phenomena on unstructured grids. In this context, the final goal has been to provide a reliable tool for handling high speed aerothermodynamic applications.
To this end, we introduce a number of design techniques that have been developed in order to provide the framework with flexibility and reusability, allowing developers to easily integrate new functionalities such as arbitrary mesh-based data structures, numerical algorithms (space discretizations, time stepping schemes, linear system solvers, ...), and physical models.
Furthermore, we describe the parallel algorithms that we have implemented in order to efficiently read/write generic computational meshes involving millions of degrees of freedom and partition them in a scalable way: benchmarks on High Performance Computing (HPC) clusters with up to 512 processors show their effective suitability for large scale computing.
Several systems of partial differential equations, characterizing flows in conditions of thermal and chemical equilibrium (with fixed and variable elemental fractions) and, particularly, nonequilibrium (multi-temperature models) have been integrated in the framework.
In order to simulate such flows, we have developed two state-of-the-art flow solvers:

1. a parallel implicit 2D/3D steady and unsteady cell-centered Finite Volume (FV) solver for arbitrary systems of Partial Differential Equation (PDE) on hybrid unstructured meshes;

2. a parallel implicit 2D/3D steady vertex-centered Residual Distribution (RD) solver for arbitrary systems of PDE's on meshes with simplex elements (triangles and tetrahedra).

The Finite Volume (FV) code has been extended to handle all the available physical models, in regimes ranging from incompressible to hypersonic. As

far as the Residual Distribution (RD) code is concerned, the strictly conservative variant of the RD method, denominated Conservative Residual Distribution (CRD), has been applied for the first time in literature to solve high speed viscous flows in thermochemical nonequilibrium, yielding some preliminary outstanding results on a challenging double cone flow simulation.
All the developments have been validated on real-life testcases of current interest in the aerospace community. A quantitative comparison with experimental measurements and/or literature has been performed whenever possible.

Contents

Acknowledgements

At the end of those many years of doctorate, after a total of 6 and a half years at the Von Karman Institute, I can sincerely say that the list of people to whom I owe my gratitude is really huge. I'll do an effort not to forget anybody and I apologize in advance if this unfortunately happens ...
First of all, I would like to thank my promoter and supervisor Prof. Herman Deconinck that since we first met, at the end of my short experience in NUMECA, has always expressed his trust in me. His being supportive and enthusiastic about every little progress I made or result I got has really motivated me to constantly improve and set the "bar" a little bit higher.
Secondly, I'll be eternally grateful to Koen and David, my former colleagues in NUMECA and two of the brightest persons I have ever known, who literally push me into the von Karman Institute for Fluid Dynamics (VKI) experience. It was really a pleasure to see David back as a colleague at the institute. He has been a huge source of inspiration for me (and many others) and his "can-do" attitude has served as stimulation for boosting my work.
Next in the list, at least chronologically, comes my COOL-mate Tiago: I will never forget that October 2003, when we sat everyday upstairs, in the computer room, next to each other like siamese twins with our own laptops and, despite our occasionally divergent opinions, we gave birth to the first working embryo of COOLFluiD. Well, the rest is recent history ... in three words : "We made it!". I really hope to keep on collaborating with you to make grow "our baby" even more.
Thanks to Dries who introduced me into the realm of parallel programming: our design COOL-meetings in VKI or KU Leuven or our short trips together (while coming to VKI in my car) have been a great occasion to exchange fruitful ideas. I will never thank you enough for all the assistance you gave me in running large simulations on KU Leuven cluster, even during holidays! Many thanks to Kurt, Jirka and Mario who have answered many of my tedious Computational Fluid Dynamics (CFD) questions at the beginning of my project. A special thanks goes to Telis: I was literally addicted to your meshes and you had a great patience to help me also with my Linux troubles whenever I asked you. I cannot speak of Linux without mentioning

Giuseppe, Federico and Raimondo, the computer center shamans, who gave me their support in a countless number of occasions.
Many thanks to Marco whose close collaboration helped me to move a huge step further towards the final goal and often succeeded in distressing me with his constant good mood. Your contribution has been fundamental for achieving the main results presented in this thesis and I really would like to continue working together also in the future, despite the probable long distance.
Thanks to Carlo for the relaxed lunch breaks and long chats in some of the few restaurants in the VKI neighborhood and his technical assistance for my occasional car troubles!
Thanks to Damiano, Alberto and Khalil for our regenerating tennis matches; to Thierry for his advices and the great enthusiasm that he can always transmit; to Thomas (Nierhaus) for some regenerating chats on heavy metal CDs or concerts; to Karin, Nadege and Thomas for having being patient office mates. In particular, additional thanks to Thomas for having given such a shot to the COOLFluiD project, with his admirable capability to master the most diverse and complex applications.
Thanks to all my former students, Sarp, Michel, Janos, Radek, Fabio, Christophe, etc. who all contributed to add critical mass to this thesis.
My gratitude goes to the VKI Director, Mario Carbonaro for having accepted me in the doctoral program; to Patrick for his trust and his quick response whenever I had a need for more computational resources; to Olivier, Doug and Prof. Degrez for having expressed in so many circumstances their support for the COOLFluiD project; to Prof. Luca Formaggia (Politecnico di Milano), Prof. Olivier Gicquel (Ecole Centrale Paris), Dr. Schwane (ESA-ESTEC) and Prof. Pierre Gaspart (ULB) for having accepted to be part of my defense jury.
Last but not least, I would like to say "Grazie mille" to my family who have always loved me, believed in me and encouraged me, especially during the hardest times. Thanks to Isabel for her love, infinite patience and support: you, more than anybody else, know who is the guy and the effort behind this thesis!

Chapter 1

Introduction

In the scientific field, it often happens that a researcher or a team of researchers start writing a software package that is meant to tackle some specific problems. When new problems or new methodologies to solve problems pop up, one has to choose between adding new features into an existing code or writing a new one from scratch.
The first solution can easily lead to an unforeseen and hardly manageable growth of the original software and can result in a series of frequent changes, re-factorings and, in the worst case, full re-design.
The last solution is typically adopted when the needed changes are incompatible with the existing software or when the desired new features are far from the original goals that were planned in the long-term.
In both cases, the importance of allowing and maximizing the software reuse and modularity becomes clear. The question that arises is then: how to maximize the reusability and the flexibility of the software? To answer these needs, a great help has been given to the scientific community during the last 15 years by the Object-Oriented (OO) programming. The latter has pushed developers to improve their way of conceiving and designing software by redirecting the focus more on the way scientific concepts and algorithms interact than on the mere functionalities per se.
At the same time, the increasing power of computer architectures and the concurrent advance towards parallel and distributed computing have allowed researchers to perform more and more challenging simulations of complex physical phenomena, motivating at the same time the development of increasingly sophisticated numerical algorithms.
As an example, CFD represents a field in which powerful computer resources have become fundamental to be able to simulate complex flow fields closer to the demands of industry, medicine and, more generally, technological progress. The required power is not anymore only a matter of execution speed or memory capacity, but it has also become a matter of software design, since scientific researchers need constantly the implementation of new features, models, tools to increase the quantity and enhance the quality of

their results.
Today's CFD applications require a considerable degree of flexibility from numerical tools, since all of the following situations can occur and should be envisioned:

- the same physical phenomena can be solved by means of different space discretization methods (Finite Volume, Finite Difference, Finite Element, Residual Distribution, Discontinuous Galerkin ...), time integrators, boundary condition treatments, iterative and direct linear system solvers etc.;
- the same numerical methods can be applied to different physical problems and their corresponding set of equations;
- some subsets of the governing equations may need to be treated with different numerical schemes;
- different physical models and constitutive relations can be applied to close the same set of governing equations.

Moreover, data structures of research codes have to be flexible and dynamic enough to support parallelization, mesh adaptation, multigrid techniques.
In order to get all these capabilities in one single code, the commonly used procedural programming based on functional decomposition has proven to be insufficient. On the contrary, OO design and programming have been of great help in tackling all these rapidly increasing needs in a more appropriate and scalable way. During the last decade, this has lead to the development of several platforms for research purposes such as DiffPack [6], Deal II [12], LibMesh [70], Life [130], OpenFoam [44]), etc. and the migration of most commercial flow solvers (Fluent, StarCD, Numeca's FINE ...) to the new approach.

1.1 The COOLFluiD project

1.1.1 History of the project

The current work has moved its first steps back in March 2002, within the **COOLFluiD** (**C**omputational **O**bject **O**riented **L**ibrary for **F**luid **D**ynamics) project, aiming at creating an OO framework for the simulation of multi-physics on unstructured meshes.
One of the primary goals of COOLFluiD was to port to a new common

platform the main simulation capabilities spread over a large number of unmaintained legacy codes, developed at the Von Karman Institute by the previous generation of PhD students. Such a framework was also and especially meant to be scalable, i.e. capable of evolving, maintainable and to easily accommodate new developments for many years to come, both from the physical modeling and the algorithmic sides.
The preliminary design was the fruit of the collaboration between the author and two other PhD candidates, namely Tiago Quintino (VKI) and Pieter De Ceuninck (KU Leuven).
Between January and September 2003, Quintino and the author worked on separate tracks, each one on a different implementation of the same concept design. In October of the same year, the author's version of COOLFluiD [74] and Quintino's HOTFluiD merge into one single platform, keeping the best capabilities and properties of both codes, but bringing them to a higher level of flexibility, thanks to a partial synergic re-design and re-implementation effort.
After the definition of a unified framework and the entry in the team of Dries Kimpe (KU Leuven) and, later on, of Thomas Wuilbaut (VKI), the scope of application was progressively enlarged to high-performance computing and more and more complex multi-physics. During the last few years, also thanks to the contributions of many other PhD candidates, diploma course members, post-docs and external partners, the project has been growing very rapidly towards the status of a customizable collaborative environment for CFD, where many researchers can cooperate to extend the simulation technology towards predefined target applications while taking maximum benefit from the others' work.

1.1.2 The COOLFluiD Architecture

COOLFluiD consists of a collection of dynamically linked libraries and application codes, organized in a multiple layer structure, as shown in Fig. 1.1. Two kinds of libraries are identifiable: kernel libraries and modules. The former supply basic functionalities, data structure and abstract interfaces, without actually implementing any scientific algorithm, while the latter add effective simulation capabilities.
The framework was developed according to a *plug-in* policy, based on the self-registration [19, 74, 78] and self-configuration techniques [78, 131], which allow new components or even third party extensions to be integrated at run-time, while only the Application Programming Interface (API) has been exposed to the developers. From top to bottom of the COOLFluiD architec-

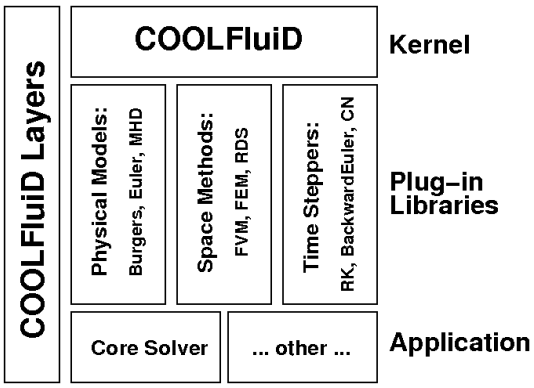


Figure 1.1: Multi-layer structure of COOLFluiD.

ture, one finds, progressively, more functional layers: the kernel defines the interfaces, the modules implement them and the application codes can select and load on demand the libraries needed for an actual simulation. Moreover, COOLFluiD is a component-based platform, where each functionality is enclosed in a separate component which can be connected to others at run time to create one or more actual application solver.

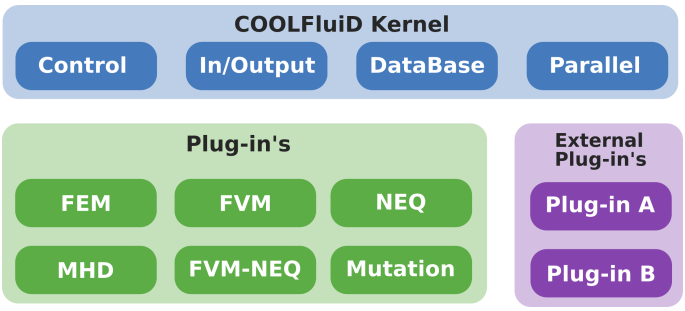


Figure 1.2: Component-based architecture of COOLFluiD: kernel modules, internal and external plug-ins.

Fig. 1.2 shows an example of some of the components available in COOLFluiD and the distinction between kernel modules, internal and external plug-ins

is stressed. Internal plug-ins are typically developed by VKI or by partners with access to the source code, while external plug-ins are usually developed independently by third parties with access only to the API. This multi-layer and multi-component structure guarantees flexibility and modularity at the highest design level and helps managing the development process, since layers and components automatically reflect different implementation levels and, in some cases, permission rights.

1.2 Objectives of the present thesis

The original goals of the COOLFluiD project have traced the guidelines for the present work, aiming at implementing a flexible and extendible object-oriented platform for high-performance simulation of aerothermodynamics on unstructured grids. This rather ambitious target has given the project a multi-disciplinary connotation nature since the start, requiring development of competence and research effort in different areas:

1. OO design tailored to scientific applications to provide the framework with a solid and flexible software infrastructure, allowing new developers to easily integrate new functionalities and fully reuse existing ones;

2. parallel algorithms and programming techniques to support large scale and high-performance computing;

3. development of robust and efficient state-of-the-art numerical solvers suitable for handling a wide range of flow regimes, ranging from incompressible to hypersonic speed;

4. integration of multiple models to provide more accurate description of complex physical phenomena, including transport properties, reaction kinetics and statistical thermodynamics.

As far as the OO design is concerned, a number of structural solutions for creating reusable and extendible scientific software have been developed in collaboration with other team members [68, 131] and will be presented. We will propose suitable design patterns for providing a uniform way of encapsulating numerical algorithms for solving Partial Differential Equation (PDE), physical models and their interaction within multi-physics simulations. We will also present solutions for handling generic mesh data-structures with a transparent treatment of both serial and distributed data.

Given our intention to support complex and computational intensive simulations, we will describe the efficient parallel algorithms for mesh decomposition and parallel input/output that we have implemented and tested on large cluster architectures.
Since our final target is to deal with aerothermodynamic applications in conditions ranging from perfect non reactive gas to thermo-chemical equilibrium to full nonequilibrium, especially in the hypersonic regime, many physical models had to be integrated in COOLFluiD and they will all be described. With regard to this, some details will be left out of this thesis, since physical modeling was not the main focus of this work and since most of the chemistry, thermodynamics and transport algorithms were used as a black box, through the interface of Mutation [94, 116, 140]. The latter is a research library which has been developed at the VKI during the last decade and has been fully linked to COOLFluiD within this work, in order to make it usable in real-life CFD simulations.
Furthermore, we will focus our attention on numerical schemes which combine accurate shock capturing and robustness. The standard cell-centered FV is the best established method and usually the preferred choice for these applications, in combination with implicit time stepping techniques to accelerate the convergence. During the last decade, however, RD schemes have been emerging as a possible valid alternative for the computation of compressible flows on unstructured meshes. We will show that our flexible platform has allowed us to develop and compare both methodologies. In particular, in this thesis we will extend, for the first time in literature, the strictly conservative variant of RD schemes, aka CRD, to the simulation of viscous chemically reactive flows.

Part I

Design Solutions for Scientific Computing

Chapter 2

Object Oriented design solutions

Introduction

Designing a framework for generic PDE solvers is not a trivial task by itself, but additional complexity has come, in our case, from having planned to deal efficiently with arbitrary numerical algorithms (with potentially different parallel data structures) and multi-physics applications.
According to a simplified conceptual view, we can consider a numerical solver for PDE's as made by a limited number of independent building blocks:

- some mesh-related data, typically numerics-dependent, that represent a discretized view in terms of both geometry and solution of the computational domain;
- an equation set describing the physical phenomena to study, typically independent of numerics;
- a group of interacting numerical algorithms to solve the PDE's on the given discretized domain.

We can therefore identify three issues of fundamental importance for developing an environment for solving PDE's:

- storage and handling of mesh data;
- implementation of physical models;
- implementation of numerical methods.

A good OO design should be able to tackle these three aspects orthogonally, independently one from the other as much as possible, in order to minimize coupling and maximize the reusability of the single components. To this aim, a clear separation should be enforced between physics and numerics, the first being the description of the properties, constitutive relations and

quantities that characterize the equation system, the second providing the mathematical tools (algorithms, algebraic and differential operators) to discretize and solve those equations.
This section will start with a short introduction to basic concepts in object-oriented programming and design like inheritance, polymorphism and composition which will be reused heavily later on. After that, our attention will focus on analyzing each one of the three above-mentioned problems (2.2, 2.3, 2.4), and on presenting corresponding possibly effective solutions, as they were implemented in COOLFluiD. Some illustrative examples will also be given in Sec. 2.3.1 and 2.4.1 in order to show the concrete applicability of the proposed ideas.

2.1 Basic concepts in OO Programming

OO programming consists in a new conception of programming based on an accurate conceptual analysis of a given problem that leads to the identification of the objects involved, their factorization into classes, the definition of class interfaces and hierarchies and the establishment of key relationships among them. This, at start, requires considerably more effort than writing a flow chart and translate it into actual code, as procedural (aka functional) programming would normally require, but it's also considerably more powerful, because it allows software developers to go beyond the specific problem and create the basis for reusable code.
OO programming is usually called *programming to the interfaces*, which tells a lot about the effort to generalize and abstract as much as possible to determine what are the objects involved and how these objects interact through their interfaces.
Some key ingredients of the OO approach, as indicated in [33], are

- **data abstraction:** the logical properties of data types are separated from their implementation;

- **data encapsulation:** the physical representation of the data of an algorithm is surrounded, so that the user of the data doesn't see the implementation, but deals with the data only in terms of its logical picture, namely its abstraction;

- **information hiding:** the details of a function or data structure are hidden in order to control access to them;

- **polymorphism:** the ability to determine dynamically (at run-time) or statically (at compile time) which of several operations with the same name is appropriate.

2.1.1 C++ techniques

C++ has been the programming language chosen for the actual implementation of the COOLFluiD framework, since it provides a lot of attractive features and allows developers to use very effective techniques, at the cost, sometimes, of a quite complex syntax. Occasionally, some minor portability problems between one compiler and another can still be encountered, but this has become more and more rare since the standardization of the language. Another valid reason to choose C++ for implementing an OO platform nowadays is the large availability of multi-purpose libraries for scientific computing in C/C++ which are easily linkable, including the ones (LAM, MPICH, OpenMP) implementing Message Passing Interface (MPI) for parallel and distributing computing.
Some of the main OO techniques supported by C++ will now be briefly described, in order to help the reader familiarize with some terminology which will recur often in the rest of the chapter.

2.1.1.1 Class Inheritance

This is a mechanism that lets developers implement one class interface defining an object in terms of another existing one. When a *derived class* inherits from a *base class*, it includes the definition of all the data and the operations defined by the parent class. A class is said to be *abstract* if its only purpose is to define a common interface for its subclasses. This implies that all or some implementations of the operations are deferred to the derived classes, making the abstract class non instantiable. Classes that aren't abstract are called *concrete*. Fig. 2.1 shows the OMT (Object Modeling Technique) [48] notation for class inheritance, with a vertical line and a triangle pointing towards the base class.

2.1.1.2 Dynamic and static binding

When identical requests that may have different implementations are forwarded to the abstract interface of a base class, inheritance can be used to obtain the already mentioned polymorphism (see 2.1). As a consequence, the correct derived class will be selected to fulfill the request appropriately.

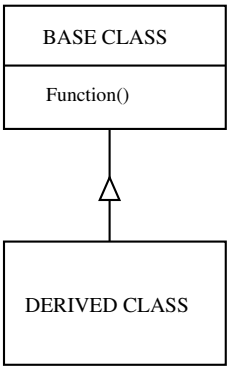


Figure 2.1: OMT diagram representing class inheritance.

When this selection occurs at run-time, the association is called *dynamic binding*, otherwise, when it occurs at compile time, it is called *static binding*.

2.1.1.3 Object Composition

Object composition is an alternative to class inheritance. It consists of assembling or *composing* objects to get more complex functionalities. The composition is defined dynamically at run-time through objects keeping references (or pointers) to other objects and forcing them to respect each others interfaces.
Each object of the composition can be considered as a *black box*, because no internal details (data, algorithms) of the objects are visible from outside the class. This allows to limit implementation dependencies and maintain encapsulation, leading to the so-called *black-box reuse* [48].
The OMT notation for composition is presented in Fig. 2.2. Composition can imply either *aggregation*, indicated by an arrow line starting with a diamond, or *acquaintance*, indicated by a plain arrow line. Aggregation requires that one object owns and/or is responsible for the other one, meaning that aggregatee and aggregator have identical lifetimes. Acquaintance implies that an object simply has knowledge of another one, without having ownership on it.

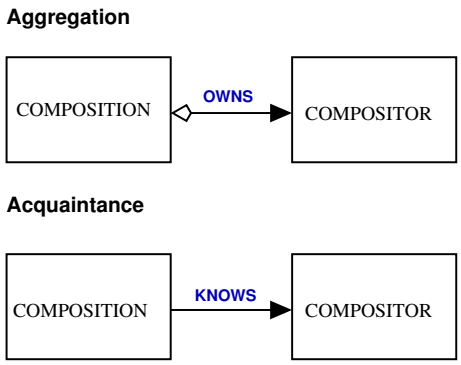


Figure 2.2: OMT diagram representing class compositions.

2.1.1.4 Templates

This technique allows developers to define a type (other name for class) without specifying all the other types it uses. The unspecified types are supplied as *parameters* at the point of use. As an example, we can consider a container (list, vector, set, etc.) that must handle different kind of elements (int, double, char, whatever fancy object) in the same way, and it's therefore parameterized with the element type.
Templates can help to create highly generic, efficient and reusable code. The most powerful example of application of this technique is the so called Standard Template Library (STL), that has been extensively used in the present work.
For further details on how to implement this techniques in practice, which is clearly out of the scope of this work, we particularly recommend to look at [149], [98], [160], [10].
For the interested reader, however, we provide in Appendix (Sec. A.1) an example of powerful use of template-based meta-programming for achieving an optimal implementation of symbolic array algebra. In the latter, we show how to enhance the state-of-the-art Fast Expression Template (FET) technique [57, 66] in order to make it usable within software frameworks with dynamically linked plug-ins, like COOLFluiD, and in order to allow the compiler to perform an optimization known as loop unrolling.

2.2 Mesh Data Handling and Storage

PDE-driven simulations typically involve a considerable amount of data related to the discretization of the computational domain, such as coordinates, state variables, geometric entities, etc., all grouped inside container objects (arrays, lists ...) on which numerical actions are performed. Additional complexity appears when computing on unstructured grids, as in the case of COOLFluiD, since different numerical methods can require the storage and usage of different kind of connectivity information. As a result, genericity in both quantity and types of data must be addressed.
Moreover, in a scalable and safe design of the data handling, framework components should be able to share data without violating encapsulation.
In a high performance environment, developers of new modules should be allowed to elect newly defined data types to be stored, handled efficiently, and shared in parallel communication. Ideally, local and distributed data should be treated uniformly from the developer's point of view, and support for different communication strategies (e.g. message passing and shared memory) should be offered.

2.2.1 Data Storage

As explained in [78, 80], in COOLFluiD, all massive data whose size scales with the problem complexity are encapsulated by a facade object `MeshData`, which, in particular, aggregates and offers safe access to a `DataStorage`, whose class definition is shown in C.L. 2.1.
`DataStorage` defines some template member functions to create, get and delete arrays of data with generic type. Herein, different set of functions are associated to different tag classes, one for each available parallel communication types, e.g. `LOCAL` for serial communication, `GLOBAL` for inter-process communication and `REMOTE` for intra-process communication (not included in C.L. 2.1 for simplicity). In reality, those types are mapped at compile time to more specific types like `MPI` (MPI-based parallel communication), `PVM` (PVM-based parallel communication), `SHM` (shared memory communication), as explained in [69, 131]. The use of macros (see the example in C.L 2.2) can help in concealing implementation details that rely upon specific parallel paradigms. As a result, a transparent parallel layer is defined in COOLFluiD, where only the communication types are exposed outside the parallel layer, i.e. in the numerical modules of the framework.
`DataStorage` maintains a key role within the parallel layer, by providing a user-friendly interface to create and manage arrays of generic data types,

```
// --- DataStorage.hh --- //

class DataStorage {
public:
  // Constructor, destructor

  // get the local data by name and type
  template <class T> DataHandle<T, LOCAL> getData(string name);

  // create and initialize the local storage
  template <class T> DataHandle< T, LOCAL> createData
  (string name, int size, const T& init = T());

  // Deletes the local data
  template <class T> void deleteData(string name);

  // get the global data by name and type
  template <class T>
  DataHandle<T, GLOBAL> getGlobalData(string name);

  // create and initialize the global storage
  template <class T>
  DataHandle<T, GLOBAL> createGlobalData(string name,
  int size, const typename GlobalTypeTrait<T>::GTYPE& init);

  // Deletes the global data
  template <class T> void deleteGlobalData(string name);
private: // data

  // map to store the pointers that hold the data
  std::map<string,void*> m_dataStorage;
}; // end class DataStorageInternal
```

Code Listing 2.1: DataStorage class definition

that are meant to be shared among different numerical components. In particular, `DataStorage` allows its client code to treat both local and distributed data uniformly. It behaves as an archive, where arrays of data with a certain type and size are registered under a user-defined name in an associative container like a `std::map` [149] [150], as can be seen in C.L. 2.1. Pointers to the arrays in question are statically cast to `void*`, in order to let coexist different types within the same instance of `DataStorage`.
Different types of arrays must be used to hold local or distributed data, according to the available parallel communication models: as an example, local data are stored in a `Evector` [32], while global data are hold inside a `ParVectorMPI` [68], if an MPI environment is available. The first offers an

```
#ifdef HAVE_MPI
typedef MPI GLOBAL;
#elif HAVE_PVM
typedef PVM GLOBAL;
#else
typedef LOCAL GLOBAL;
#endif

#ifdef HAVE_SHM
typedef SHM REMOTE;
#else
typedef LOCAL REMOTE;
#endif
```

Code Listing 2.2: Macros definitions for GLOBAL and LOCAL communication models

improved and more efficient implementation of `std::vector` [29, 149, 150], the basic container provided by STL. In particular, `Evector` supports back insertion/deletion of elements in constant time and on-the-fly memory allocation to accommodate new entries, without any overhead. `ParVectorMPI` provides additional parallel functionalities such as synchronization of data belonging to the overlap region, which is defined in Sec. 3.1.1.3.
No built-in garbage collection or deletion policies based on some sort of reference counting are automatically provided by `DataStorage`. This fact implies that data explicitly created with `createData`() must be consistently and explicitly deleted with a call to `deleteData`(): ad-hoc *setup* and *unsetup* `Commands` fulfill this task in each numerical module, as explained in Sec. 2.4.1.

2.2.1.1 Data Handle

Once registered in the `DataStorage`, data can be accessed through a smart pointer, `DataHandle`, that ensures safety by preventing the data array from being accidentally deleted. It also provides inlinable accessor/mutator functions to the individual array entries and offers some additional functionalities. The interface of `DataHandle` is presented in C.L. 2.3.
Different partial template specializations of `DataHandle` exist, one for each tag class. The default behaviour corresponds to the `LOCAL` tag. As an example, the class definition for the `DataHandle` corresponding to the `MPI` tag is given in 2.4.
`DataHandleMPI` declares the same functionalities of `DataHandle` but pro-

```
// --- DataHandle.hh --- //

template <class T , class COMTYPE = LOCAL>
class DataHandle : public  DataHandleInternal<T,COMTYPE>
{
public:
  // Constructors, destructor, assignment operator overloading

   // preallocate memory for an array of given size
  void reserve (int s);

  // return the global (cross-processes) and local size of the
      array
  int getGlobalSize() const;
  int getLocalSize() const;

  // begin and end the synchronization
  void beginSync() {}
  void endSync() {}

  // add ghost and local points and return the corresponding
      index
  int addGhostPoint (int globalID) {}
  int addLocalPoint (int globalID);

  // build the map storing synchronization data
  void buildMap() {}
}
```

Code Listing 2.3: Interface of DataHandle

```
// --- DataHandleMPI.hh --- //

template <class T>
class DataHandle<T,MPI> : public DataHandleInternal<T,LOCAL>
{
public:
  // Constructors, destructor, assignment operator overloading

  // ... same member functions as DataHandle but all implemented
private:
  // pointer to array storing data to communicate
  ParVectorMPI<typename GlobalTypeTrait<T>::GTYPE>* m_globalPtr;
};
```

Code Listing 2.4: DataHandleMPI interface

vides an implementation for the synchronization and handling of data belonging to the overlap region. The actual synchronization and communication are delegated to the aggregated parallel array, `ParVectorMPI`, in which all MPI calls are encapsulated. The same functions must also be provided by the local `DataHandle` in C.L. 2.3 (with an empty implementation) in order to allow the whole code to compile even without having MPI, i.e. when the `GLOBAL` tag is aliased to `LOCAL`.
All specializations of `DataHandles` derive from `DataHandleInternal`, whose interface is reported in C.L. 2.5. In particular, the overloading of the subscripting `operator[]` allows to directly access array entries through the `DataHandle` interface.

```
// --- DataHandleInternal.hh --- //

template <typename T , typename COMTYPE >
class DataHandleInternal {
public:
  typedef typename COMTYPE::template
    StoragePolicy<T>::ContainerType StorageType;

  // constructor
  DataHandleInternal(StorageType *const ptr) : m_ptr(ptr){}

  // copy constructors, assignment operator ...

  // overloading of subscripting operator
  T& operator[] (int idx) {return (*m_ptr)[idx];}

  // overloading of operator ()
  T& operator() (int i, int j, int stride) {return
      (*this)[i*stride+j];}

  // ...  other functions
protected:
  StorageType* m_ptr; // pointer to the array to be handled
};
```

Code Listing 2.5: DataHandleInternal interface

In the case of a `DataHandleMPI`, two data arrays are encapsulated: a local array containing entries of type `T` which is inherited from `DataHandleInternal` and a distributed array meant to store entries of built-in types (integers or floatings) corresponding to the type `GlobalTypeTrait<T>::GTYPE`. For example, while running in a MPI environment, a `DataHandle<State*,GLOBAL>` hides a local array of `State*` and a distributed array of `double`, correspond-

ing to the actual raw storage of the objects of type `State`. This dualism is completely transparent to COOLFluiD developers, who can work locally with `DataHandles` of objects, without seeing the details of their actual storage and parallelization.

2.2.2 Topological Region Sets

The computational domain can be considered as composed by topologically different regions which correspond to the `TopologicalRegionSet` concept in COOLFluiD. Each TopologicalRegionSet (TRS) consists of a list of `TopologicalRegions`, i.e. subsets of the domain ideally linked with a particular Computer Aided Design (CAD) representation. Each set of boundary surfaces on which a certain boundary condition has to be enforced can be interpreted as a different TRS, while the whole boundary can be broken into smaller patches, TopologicalRegion (TR), according, for instance, to the surface definition in the CAD model. If we consider, as an example, the computational domain built around an airplane, FarField, Wing, Nacelle, Fuselage can all be considered as different TRS's. One could alternatively choose to pack Wing, Nacelle and Fuselage together to form a unique continuous TRS, called Airplane, consisting of three TR's.
Because of consistency and conveniency, not only the boundary but also the interior of the domain is assigned to one or more TRS's, according to the needs of the chosen numerical methods (space/time discretizations, mesh adaptation, mesh movement, coupling algorithms, etc.).
The latter typically are applied on geometric entities (cells, faces, edges), which represent the basic entities in the topological dicretization of the domain. A `GeometricEntity` is defined by an algorithm-dependent agglomeration of degrees of freedom, both in geometric (`Nodes` or coordinates) and solution space (`States` or solution vectors). Both `Nodes` and `States` are provided with global (inter-process) and local (intra-process) IDs, which are used to build connectivity information. As a result, each TRS's (or TR's) can be logically viewed as set (or subset) of GeometricEntity (GE) in which each portion of the domain can be decomposed for computational purposes.
A TRS is a *Proxy* object [48] that controls the access to:

- the local and global IDs of all the GE's belonging to the TRS;
- the local IDs of all nodes and states referenced by local GE's;

- the geo-type information (i.e. the unique IDs corresponding to the shape and polynomial order of each GE);
- the geo-to-node and geo-to-state connectivities;

as shown in the diagram in Fig.2.3. In order to minimize the memory requirements, all these data are stored sequentially in unidimensional arrays (but logically bidimensional) which can be addressed by the client code exclusively through the TRS or TR interfaces, with the latter offering only a partial view of the whole data.

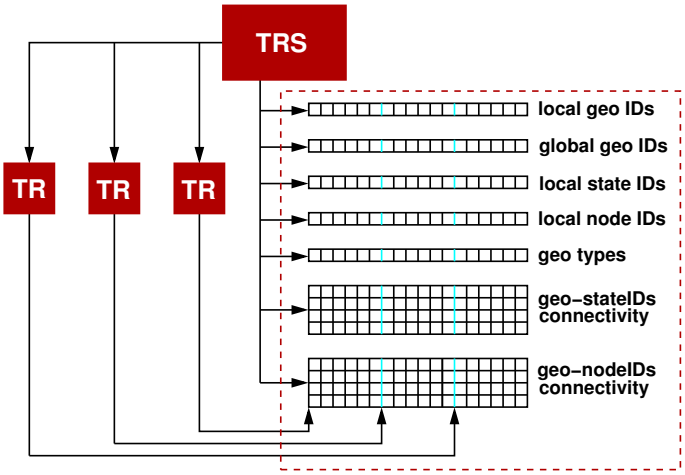


Figure 2.3: Diagram showing the array-based data-structure associated to a TRS and the underlying TR.

2.2.2.1 On-the-fly creation of Geometric Entities

As mentioned in the previous paragraph, numerical algorithms are applied on cells, faces, edges, all corresponding to GE instances in our design. A GE is a locally/globally indexed object which is composed by `States`, `Nodes`, `ShapeFunctions` for the solution and geometric interpolation and it can keep reference to other neighbor GE's, according to the computational stencil needed by the numerical method. A simplified interface for GE is shown in C.L. 2.6.

```
// --- GeometricEntity.hh --- //

class GeometricEntity {
  public:
   // constructor and destructor, utility functions ...

   // accessor functions ...
   SafePtr<vector<State*> > getStates() const;
   SafePtr<vector<Node*> > getNodes() const;
   SafePtr<vector<GeometricEntity*> > getNeighborGeo() const;

   // mutator functions
   void setState(int is, State* state);
   void setNode(int in, Node* node);
   void setNeighborGeo(int ig, GeometricEntity* neighborGeos);

protected: //data
   int m_geoID; // local ID of this GeometricEntity
   std::vector<State*> m_states; // state list
   std::vector<Node*> m_nodes;   // node list
   std::vector<GeometricEntity*> m_neighbors; // neighbor GEs
   ShapeFunction* m_solSF; // solution shape function
   ShapeFunction* m_geoSF; // geometric shape function
};
```

Code Listing 2.6: GeometricEntity interface

As one can easily notice, a `GeometricEntity` is a relatively lightweight object, but it would be prohibitely expensive to carry in memory all the cells and/or faces and/or edges corresponding to the given mesh and numerical algorithm along the whole simulation. It is however extremely convenient to have such an object that encapsulates all the needed local connectivity, shape function-related, stencil-related data and that offers them ready-to-use.

This lead us to the idea of applying a *Builder* pattern [48] to separate the construction of GE's from their representation, starting from a preallocated memory pool. The pattern is represented in Fig. 2.4.

We define a generic interface for `GeoEntityBuilder`, which is a template class parameterized with the actual builder, called `GEOBUILDER` in C.L. 2.7. `GeoEntityBuilder` just forwards actions to the aggregated `GEOBUILDER`, which is imposed by the chosen numerical algorithm.

Each possible choice for the parameter `GEOBUILDER`, corresponds to a different instance of `GeoEntityBuilder`. As an example, C.L. 2.8 shows the class definition of a simple `FEMCellBuilder`. The latter, in particular, provides a

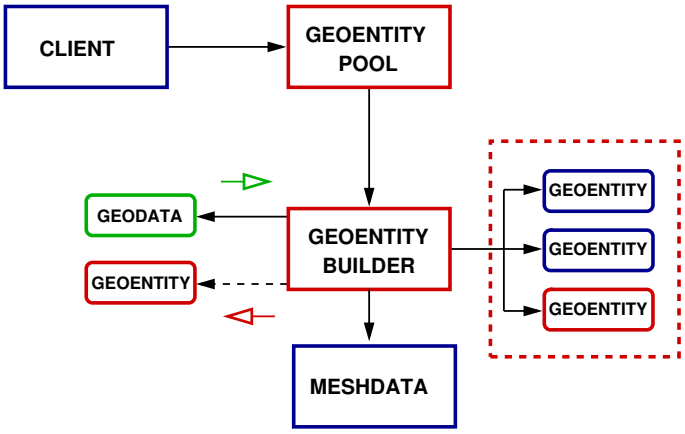


Figure 2.4: Builder pattern for creating GeometricEntities.

```
// --- GeoEntityBuilder.hh --- //

template <class GEOBUILDER>
class GeoEntityBuilder {
public:
  // constructor, destructor

  // allocate the prototype GEs and set up GEOBUILDER data
  void setup() {m_builder.setup();}

  // get the input data for the GEOBUILDER
  typename GEOBUILDER::GeoData& getDataGE()
  {return m_builder.getDataGE();}

  // build the GE corresponding to the given input data
  GeometricEntity* buildGE() {return m_builder.buildGE();}

  // release the built GEs to prepare the next creation
  void releaseGE() {m_builder.releaseGE();}

private: //data
  GEOBUILDER m_builder; // actual GE builder
};
```

Code Listing 2.7: GeoEntityBuilder interface

```
// --- FEMCellBuilder.hh --- //

class FEMCellBuilder {
public:

  // This nested struct groups the data needed by this builder
  struct GeoData {
    SafePtr<TopologicalRegionSet> trs; // pointer to TRS
    int idx; // geo index in TRS
  };

  // allocate the prototype GEs and set up GEOBUILDER data
  void setup();

  // get the input data for the FEMBuilder
  FEMBuilder::GeoData& getDataGE() {return m_data;}

  // build the GE corresponding to the given input data
  GeometricEntity* buildGE();

  // release the built GEs to prepare the next creation
  void releaseGE();

private:
  FEMBuilder::GeoData m_data; // data of this builder
  DataHandle<Node*> m_nodes;  // handle to the Nodes storage
  DataHandle<State*> m_states;// handle to the States storage
  std::vector<GeometricEntity*> m_poolGE; // pool of
      preallocated GEs
};
```

Code Listing 2.8: FEMBuilder interface

nested struct `GeoData`, holding the numerics-dependent input data for the creation of a GE.
During the setup phase of the simulation, a small number of GE objects, corresponding to all the entities belonging to the computational stencil required by the actual application, is created and stored in an array. While applying numerical algorithms, after the `GeoData` are set by the statically bound client code, the builder is asked to assemble a ready-to-use GE, where all the data have been fetched out from the corresponding TRS and `DataHandles`. The builder operates on the prototype GE's that have been already previously allocated in memory, but it changes their content (data pointers) according to the current request. The GE creational algorithm can be as complex as one needs, but all its complexity and data dependencies are hidden to the

client code that, in our example, just reduces to C.L. 2.9.

```
1
  SafePtr<GeometricEntityPool<FEMCellBuilder> > builder;
  builder.setup();
4 ...

  FEMCellBuilder::GeoData& geoData = builder.getDataGE();
7 geoData.trs = getCurrentTRS();
  geoData.idx = getCurrentGeoIDInTRS();

10 builder.buildGE();
  ...
  builder.releaseGE();
```

Code Listing 2.9: FEMBuilder interface

This separation between the GE construction and the GE data object itself has allowed us to easily deal with the most diverse space discretizations and associated data structures such as Finite Element Method (FEM), cell centered FV, Spectral FV, Spectral FD, DG, each one providing an ad-hoc definition and implementation of a concrete GE builder.
`GeoEntityBuilder` offers an example of static polymorphism, where, instead of defining an abstract interface and implement its virtual member functions in the derived classes, actions are delegated by `GeoEntityBuilder` to an aggregated interface which is statically bound to the client code. This choice offers superior performance (no virtual calls on some inlinable functions in tight loops) and it reflects a logical dependence of the concrete numerical method on the concrete way of creating suitable GE's for it.

Memory optimization. On-the-fly creation of GE's has allowed us to heavily optimize the code in terms of memory, compared to a previous approach where all GE's were allocated and actually stored in the TRS's. This improvement made memory usage competitive against procedural (as opposite to object-oriented) highly optimized numerical codes, such as Elsyca's PlatingMaster, an industrial FEM solver (written in C) for electrochemical applications.

2.3 Implementation of Physical Models

Scientists and researchers that work in the field of Fluid Dynamics deal regularly with complex systems of convection-diffusion-reaction PDE's that

can be formulated as

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F^c} = \nabla \cdot \mathbf{F^d} + \mathbf{S} \tag{2.1}$$

where $\mathbf{U}$ (state vector of unknown variables), F^c (convective fluxes), F^d (diffusive fluxes), $\mathbf{S}$ (source or reaction term) depend on the chosen physical model. The latter can be seen as a composition of entities, e.g., transport coefficients, thermodynamic properties and quantities that contribute to define the mathematical description of some physical phenomena. It can be noticed that one can look at the same physical model through different formulations of the corresponding equations, which involve the use of different sets of variables, transformations, or other adaptations tailored towards a specific numerical method.
Moreover, in a framework for solving PDE's which supports different numerical techniques and run-time loading of components, the modules where algorithms are implemented should be as independent as possible from the modules dealing with the physics description. Therefore, when new physical models or numerical components are integrated, the need for modifications on the kernel, where most of the key abstractions are defined, should be avoided. Ideally, it should be possible to build more complex models by simply extending and reusing available ones, or solving the same equations by means of different numerical techniques, without requiring changes in the existing abstract interfaces and without establishing a high level direct coupling between physics and algorithms. Widely used OO CFD platforms like [6] or [44] lack these last features, since the implementation of the physics is directly tangled to the numerical solver, in particular to the space discretization. This is surely a reasonable down-to-earth solution, it is probably efficient, but it does not promote enough reusability and interchangeability between physics and numerics. The `Perspective` pattern, that will now be described, proposes a structural way for overcoming the above mentioned pitfalls and for facilitating dynamical incremental changes and code reuse.

2.3.1 Perspective Pattern

Trying to define a single abstract interface for a generic physical model would be quite a demanding task and it would easily lead to a non maintainable solution, especially if we keep in mind our concern with run-time flexibility. In COOLFluiD, this hypothetical hard-to-define interface is therefore broken into a limited granularity of independent *Perspective* objects [78, 80], each one offering a different view of the same physics, according to the needs

of possibly different numerical algorithms. To make an example, convective, diffusive, inertial, reaction or source terms of the equations can all be Perspectives with a certain abstract interface. Moreover, if one implements a new numerical scheme that needs to access an available physical model, but that requires something not foreseen a priori and, therefore, not offered by the available interfaces, a new Perspective can be created, without affecting the existing ones.
Figure 2.5 shows the OMT class diagram of a Perspective pattern applied to a generic physical model. The base `PhysicalModel` defines a very general and narrow abstract interface. `ConcreteModel` derives from `PhysicalModel`, implements the virtual methods of the parent class and defines another interface to which the concrete Perspective objects (and only those) are statically bound. This other interface offers data and functionalities which are typical of a certain physics, but invariant to all its possible Perspectives.

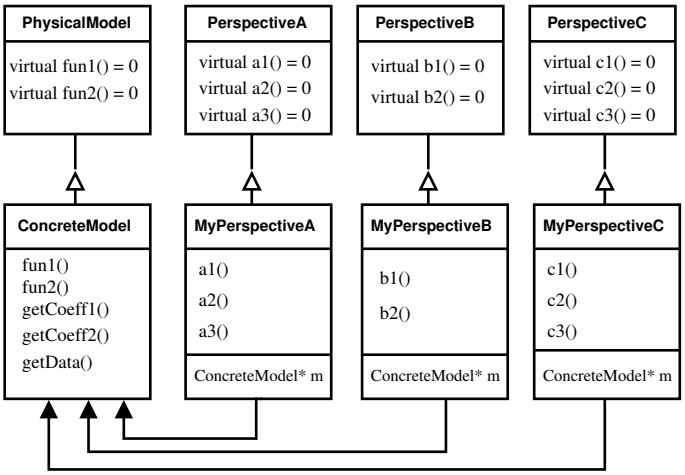


Figure 2.5: Perspective pattern applied to a Physical Model.

The resulting pattern lets the numerical client code make use of the physical model through an abstract layer, dynamically enlargeable if required, given by a number of Perspective objects, while all their collaborations with the `ConcreteModel` are completely hidden. The pattern reflects the composition-based *Adapter* described in [48], but with a single shared *Adaptee*

object, i.e. `ConcreteModel`, and multiple abstract *Targets* (called Perspectives here), each one with a number of derived classes (*Adapters*). The fact that several objects may be defined to describe the same physics may look like a disadvantage, but, in our experience, improves reusability, allows developers to easily decouple physics from numerics and gives better support to a run-time plug-in policy.
Unlike in other OO CFD platforms [28, 44] where no neat distinction between numerical algorithms and physical description is searched and where a close form of reliance (inheritance) binds the equation model and the scheme, in COOLFluiD, physics is generally completely independent and unaware of numerical methods (space and time discretizations, linear system solving, etc.). However, a design that is based on Perspective objects does not even prevent from having numerical objects, such as *Strategies* or *Commands* in 2.4.1, with static binding to the actual physics, e.g. in the case of some special schemes or boundary conditions. In the latter case, the use of Perspectives can still help to limit dependencies, improve reusability and avoid excessive sub-classing.
Some code samples are now presented in order to show the concrete applicability of the described pattern for handling complex physical models.

2.3.1.1 Physical Model Object

At first, the interface of the base `PhysicalModel` is defined in C.L. 2.10.
A polymorphic `BaseTerm` object, as defined in C.L. 2.11, and a corresponding enumerated variable are assigned to each independent physical term of the equation (convection, diffusion, reaction, dispersion, inertia, etc.) and they are registered in an associative container, such as a `std::map`. `BaseTerm` manages a number of unidimensional arrays of floats (`RealVector`), where some physics-dependent data (thermodynamic quantities, transport coefficients, parameters useful for calculating fluxes, eigenvectors, eigenvalues etc.) are stored and continuously recomputed during the simulation.
In order not to affect the total memory requirements of the computation, the maximum number of these arrays is determined by the numerical method and scales with the number of degrees of freedom in one or more geometric entities (cells, faces, etc.), typically depending on the required computational stencil. To make an example, for a cell-vertex based algorithm (FEM, RD, etc.), the total number of physical data arrays which are stored in `BaseTerm` could be the maximum number of quadrature points in a cell. The size and the content of these arrays will be defined by the classes deriving from `BaseTerm`.

```
// --- PhysicalModel.hh --- //

class PhysicalModel {
public:
  // constructor, virtual destructor ...

  // enumerated variables used to define equation term types
  enum TermID {CONVECTION, DIFFUSION, REACTION, DISPERSION};

  virtual void setup() = 0;               //set up private data
  virtual int getDimension() const = 0;   //geometric dimension
  virtual int getNbEquations() const = 0; //number of equations

  // accessor-mutator function returning the BaseTerm
  // corresponding to the given TermID
  SafePtr<BaseTerm> getTerm(TermID t) {return m_tMap.find(t);}

protected:
  // registers the TermID and a pointer to the
  // corresponding polymorphic BaseTerm object
  void registerTerm(TermID t, BaseTerm* bt)
      {m_tMap.insert(t,bt);}
private:
  // mapping between TermID and a polymorphic physical term
  Map<TermID, SafePtr<BaseTerm> > m_tMap;
};
```

Code Listing 2.10: PhysicalModel interface

A template compositor object is defined for each possible combination of physical equation terms (`Convection`, `Diffusion`, `ConvectionDiffusion`, etc.). It derives from `PhysicalModel` and aggregates generic policies [10], i.e. subclasses of `BaseTerms`. In C.L. 2.12 we present the class definition of a `ConvectionDiffusion` compositor, while its constructor implementation is shown in C.L 2.13. The equation terms aggregated by the compositor are registered in the parent class and each one of them is made accessible polymorphically through the `getTerm`() method defined in `PhysicalModel`.

Example: Navier-Stokes Model. We consider the case of a Navier-Stokes model: an overview of its class diagram is provided in Fig. 2.6.
The class definitions for the convective `EulerTerm` and the diffusive `NSTerm` are shown in C.L 2.14 and 2.15.
Besides providing accessors methods for useful parameters and libraries computing physical quantities (i.e. thermodynamics or transport properties), `EulerTerm` and `NSTerm` define the size and, by means of enumer-

```
// --- BaseTerm.hh --- //

class BaseTerm {
public:
  // constructor, virtual destructor ...

  // sets the maximum number of physical data arrays:
  // this size must be set by the numerical solver
  // because it depends on the computational stencil
  void setNbDataArrays(int maxSize);

  // get the size of each array of temporary physical data
  virtual int getDataSize() const = 0;

  // finds the array of data  corresponding to the given
  // degree of freedom (state vector)
  SafePtr<RealVector> getPhysicalData(State* state);
};
```

Code Listing 2.11: BaseTerm interface

```
// --- ConvectionDiffusion.hh --- //

template <class CT, class DT>
class ConvectionDiffusion : public PhysicalModel {
public:
  // constructor, virtual destructor,
  // overridden parent class pure virtual methods
protected:
  std::auto_ptr<CT> m_convTerm; // convective term
  std::auto_ptr<DT> m_diffTerm; // diffusive term
};
```

Code Listing 2.12: ConvectionDiffusion interface

```
template <class CT, class DT>
ConvectionDiffusion<CT,DT>::ConvectionDiffusion(string name) :
 PhysicalModel(name),
 m_convTerm(new CT(CT::getTermName())),
 m_diffTerm(new DT(DT::getTermName()))
{
  // register all the equation terms and their TypeIDs
  registerTerm(PhysicalModel::CONVECTION, m_convTerm.get());
  registerTerm(PhysicalModel::DIFFUSION, m_diffTerm.get());
}
```

Code Listing 2.13: ConvectionDiffusion constructor implementation

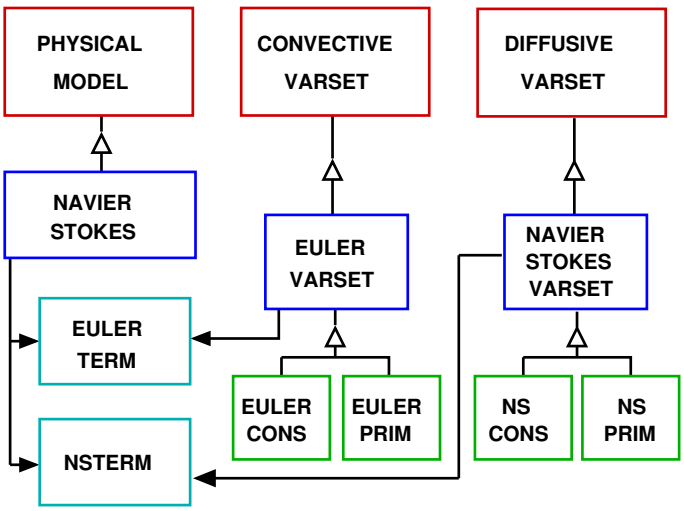


Figure 2.6: Perspective pattern applied to a Navier-Stokes model.

```
// --- EulerTerm.hh --- //

class EulerTerm : public BaseTerm {
public:
  // enumerated variables: density, pressure, enthalpy, etc.
  enum {RHO=0,P=1,H=2,E=3,A=4,T=5,V=6,VX=7,VY=8,VZ=9};

  // number of the enumerated variables
  virtual int getDataSize() const {return 10;}

  CFreal getMachInf() const;  // freestream mach
  CFreal getTempRef() const;  // freestream temperature
  CFreal getPressRef() const; // freestream pressure
  static string getTermName() {return "Euler";}

  SafePtr<TDLibrary> getThermodynamicLibrary() const;
private:
  ... // member data
};
```

Code Listing 2.14: EulerTerm interface

ated variables, the entries for the arrays of physical data that are stored in `BaseTerm` and that have been previously introduced. An instantiable

```
// --- NSTerm.hh --- //

class NSTerm : public BaseTerm {
public:
  // dynamic viscosity, thermal conductivity
  enum {MU=0, LAMBDA=1};

  // number of the enumerated variables
  virtual int getDataSize() const {return 2;}

  CFreal getPrandtl() const;     // Prandtl number
  CFreal getReynoldsInf() const; // Reynolds number
  static string getTermName() {return "NS";}

  // get library to compute transport properties
  SafePtr<TPLibrary> getTransportLibrary() const;
private:
  ... // member data
};
```

Code Listing 2.15: NSTerm interface

`NavierStokesModel`, i.e. what is called `ConcreteModel` in the Perspective pattern, can now be defined as in C.L. 2.16.

```
// --- NavierStokesModel.hh --- //

class NavierStokesModel : public ConvectionDiffusion
                          <EulerTerm, NSTerm> {
public:
  // constructor, virtual destructor
  virtual int getDimension() const;   //geometric dimension
  virtual int getNbEquations() const; //number of equations
  virtual int setup() const; //set up data
};
```

Code Listing 2.16: NavierStokesModel interface

`NavierStokesModel` can exploit the knowledge of both its concrete convective and diffusive terms to compute, for instance, adimensionalizing coefficients.

2.3.1.2 Variable Sets

Numerical algorithms are not direct clients of the `ConcreteModel`. As explained previously, the latter is used through a layer of polymorphic Per-

spective objects, whose collaborations are statically bound to the ConcreteModel and therefore potentially very efficient. A variable set, (`VarSet`), is a Perspective that decouples the usage of a physical model from the knowledge of the used variables. This, for instance, allows the client code to solve equations formulated in conservative variables, to perform intermediate algorithmic steps and to update in other ones (primitive, characteristic, etc.). In order to deal with a system of convection-diffusion equations, two variable sets classes are defined: `ConvectiveVarSet` and `DiffusiveVarSet`, respectively in C.L. 2.17 and 2.18.

```
// --- ConvectiveVarSet.hh --- //

class ConvectiveVarSet {
public:
  // constructor, virtual destructor, etc.

  virtual void setup()=0; //set up private data

  // set physical data starting from a state vector
  virtual void setPhysicalData
           (const State& state, RealVector& data)=0;

  // set state vector starting from physical data
  virtual void setFromPhysicalData
           (const RealVector& data, State& state)=0;

  virtual void setJacobians(...)=0;   //set jacobians matrices
  virtual void splitJacob(...)=0;     //split jacobian
  virtual void computeEigenSystem(...)=0; //set
      eigenvalues/vectors
  virtual void computeEigenValues(...)=0; //set eigenvalues
  virtual void computeFlux(...)=0;//compute the convective flux
};
```

Code Listing 2.17: ConvectiveVarSet interface

Among all the methods declared in the above two classes, particular attention is due to `setPhysicalData`() and its dual `setFromPhysicalData`(): the former starts from a given state vector (unknown variables) and sets the physical dependent data, whose entry pattern is defined by the the subclasses of `BaseTerm`; the latter does the opposite. In the case of the `EulerTerm`, for instance, if we assume that a `State` $\mathbf{P}$ holds primitive variables (pressure, velocity components, temperature)

$$\mathbf{P} = [\mathrm{p}, \mathrm{v}_x, \mathrm{v}_y, \mathrm{v}_z, \mathrm{T}] \tag{2.2}$$

```
// --- DiffusiveVarSet.hh --- //

class DiffusiveVarSet {
public:
  // constructor, virtual destructor, etc.

  virtual void setup()=0; //set up private data

  // set physical data starting from a state vector
  virtual void setPhysicalData
    (const State& state, RealVector& data)=0;

  virtual void computeFlux(...)=0; //compute the diffusive flux

  // set the variables whose gradients appear in the diffusive
  // fluxes, starting from the given State vectors
  virtual void setGradientVars(vector<RealVector*>& states,
       RealMatrix& values, int stateSize) = 0;
};
```

Code Listing 2.18: DiffusiveVarSet interface

`setPhysicalData`() will compute an array $\mathbf{W}$ of physical quantities (density, pressure, total enthalpy, total energy, sound speed, temperature, velocity module and components):

$$\mathbf{W} = [\rho, \mathrm{p}, \mathrm{H}, \mathrm{E}, \mathrm{a}, \mathrm{T}, \mathrm{V}, \mathrm{v}_x, \mathrm{v}_y, \mathrm{v}_z] \tag{2.3}$$

from $\mathbf{P}$ by means of thermodynamic relations. $\mathbf{W}$ will be then reused to compute eigenvalues, fluxes etc. A key point for the flexibility of the design is that each `VarSet` has acquaintance of the physical equation terms, but not of the resulting `ConcreteModel`: this allows the developer to compose progressively more complex models with full reuse of the individual equation terms.

Example: Navier-Stokes VarSets. As shown schematically in Fig 2.6 and, in actual code, in C.L. 2.19 and 2.20, `EulerVarSet` and `NavierStokesVarSet` inherit the interface of the corresponding parent classes.
Some functionalities associated to a `VarSet` are independent on the variables in which one needs to work: advective fluxes, for instance, can always be computed once that physical data like pressure, enthalpy, velocity are known, because their formulation is always the same due to the conservation property.

```
// --- EulerVarSet.hh --- //

class EulerVarSet : public ConvectiveVarSet {
public:
  // constructor, virtual destructor,
  // overridden parent virtual methods
protected:
  SafePtr<EulerTerm> m_model;
};
```

Code Listing 2.19: EulerVarSet interface

```
// --- NavierStokesVarSet.hh --- //

class NavierStokesVarSet : public DiffusiveVarSet {
public:
  // constructor, virtual destructor,
  // overridden parent virtual methods
protected:
  SafePtr<NSTerm> m_model;
};
```

Code Listing 2.20: NavierStokesVarSet interface

However, as a counter example, the Euler equations can be written in conservative, symmetrizing, entropy, characteristic (...) variables and each one of this formulation defines different jacobian matrices for the convective fluxes. The `VarSet` abstraction is particularly useful in these more complex cases, since we can let variable-dependent subclasses of `EulerVarSet` implement the jacobian matrices, according to the chosen formulation. Therefore you can have `EulerCons`(conservative), `EulerPrim`(primitive), `EulerChar` (characteristic), etc.
It is helpful to have derived classes for the `NavierStokesVarSet` too, since, for instance, the transport properties can be computed differently in chemically reactive or non reactive flows, in Local Thermodynamic Equilibrium (LTE) or in Thermo-Chemical Nonequilibrium (TCNEQ).
The method `computeFlux()` that calculates the diffusive flux is a template method [48] in `NavierStokesVarSet` and delegates the computation of the transport properties to subclasses, which have more specific knowledge. In the case of the "basic" Navier-Stokes model, as indicated in Fig. 2.6, we can have `NavierStokesCons` for conservative variables and `NavierStokesPrim` for primitive variables.

2.3.1.3 MultiScalarTerm and MultiScalarVarSet

The Perspective pattern is especially designed to deal with cases where additional sets of equations must be attached to an existing equation system. In all these situations, a general treatment is applied, based on the combined use of `MultiScalarTerm` and `MultiScalarVarSet`. The former implements a generic extension to an existing convective physical term (e.g. `EulerTerm`), allowing to add an arbitrary number of subsets of N_l physical data (scalar variables), each one corresponding to the w_l quantity in a generic conservation equation written in the form:

$$\frac{\partial \rho w_l}{\partial t} + \nabla \cdot (\rho w_l \mathbf{v}) + \cdots = 0, \quad l = 1, \ldots, N_l \tag{2.4}$$

```
// --- MultiScalarTerm.hh --- //

template <typename BASE>
class MultiScalarTerm : public BASE {
public:
  // constructor, virtual destructor, setup functions

  // number of the enumerated variables
  virtual int getDataSize() const
  {return BASE::getDataSize() + m_nbScalarVars.sum();}

  // start position of scalar vars data of subset i
  int getFirstScalarVar(int i) const {return m_firstVars[i];}

  // number of scalar vars data of subset i
  int getNbScalarVars(int i) const {return m_nbScalarVars[i];}

  // number of subsets of scalar vars
  int getNbScalarVarSets() const {return m_nbScalarVars.size();}
private:
  std::valarray<int> m_nbScalarVars;// number of scalar vars
  std::valarray<int> m_firstVars; // ID of the first vars
};
```

Code Listing 2.21: MultiScalarTerm interface

To make a concrete example, in the case of a thermo-chemical non equilibrium model, 2 subsets are needed: one corresponds to the N_s continuity equations for the chemical species, the other one corresponds to the N_v conservation equations for the vibrational energy of molecules. The interface of `MultiScalarTerm` is shown in C.L 2.21. Analogously, `MultiScalarVarSet`, whose interface is shown in C.L. 2.22 allows to treat the convective part

of additional equations like 2.4 in concrete `ConvectiveVarSets`. This allows the developer to incrementally build more and more complex physical models, where new equations can easily be added.

```
// --- MultiScalarVarSet.hh --- //

template <class BASEVS>
class MultiScalarVarSet : public BASEVS {
public:
  typedef MultiScalarTerm<typename BASEVS::PTERM> PTERM;

  // constructor, virtual destructor

  // return the convective term
  SafePtr<PTERM> getModel() const{return m_mScalarModel;}

  // get some data corresponding to the subset of equations
      related with
  // this variable set. The most concrete ConvectiveVarSet must
      set
  // this data
  static std::vector<EquationSetData>& getEqSetData() {
    static std::vector<Framework::EquationSetData> eqSetData;
    return eqSetData;
  }

protected:
   // add an equation data with the specified number of
       equations
   void addEqSetData(int nbEqs);

   // compute eigenvalues and convective fluxes for scalar
       variables
   virtual void computeEigenValues(...);
   virtual void computeFlux(...);
private:
  SafePtr<PTERM> m_mScalarModel; // convective term
};
```

Code Listing 2.22: MultiScalarVarSet interface

In particular, one `EquationSetData` object is associated to each equation subset in order to keep knowledge of the corresponding subset ID and provide the list of all the variables IDs belonging to that subset.

2.3.1.4 Variable Transformers

Another polymorphic Perspective is the `VariableTransformer`. This allows the client code to apply matrix similarity transformations with the form $\partial\mathbf{U}/\partial\mathbf{V}$ between variables or to compute one set of variables from another $\mathbf{U}(\mathbf{V})$ analytically (e.g. Euler primitive from Euler conservative and viceversa).
The combined use of `VarSets` and `VariableTransformers` gives the freedom to use different variables to update the solution, compute the residual, linearize jacobians, distribute residuals, without requiring any modification in the client numerical algorithms that work with dynamically bound Perspective objects, completely unaware of the actual physics.
The support for variable manipulation and variable independent algorithms which is offered by COOLFluiD represents an important feature that none of other well known CFD platforms such as [6] or [44] have.

2.4 Implementation of Numerical Methods

The solution of PDE's requires the implementation of different numerical techniques that deal with time discretization, space discretization, linear system solving, mesh adaptation algorithms, error estimation, mesh generation, etc.
In a typical OO design, e.g. like the ones proposed by [28], [43], [104], or [134], each one of these simulation steps is enclosed in separate object (or polymorphic hierarchies of objects) and different patterns are applied to make them all stick together and interact.
It would be advantageous to have just a single pattern, extraordinarily reusable, that allows the developer to encapsulate different numerical methods uniformly, and lets him/her focus more on the algorithm itself rather than on its already well-defined surrounding. This pattern should ease the integration of new algorithms, but also the implementation of different versions or parts of the same algorithm, while looking for optimal solutions and/or tuning for run-time performance.
One of the main achievements of COOLFluiD, as opposed to other similar OO environments, lies in having developed a uniform high level structural solution, potentially able to encapsulate and tackle efficiently, we believe, all possible numerical algorithms: the compound Method-Command-Strategy (MCS)pattern.

2.4.1 Method-Command-Strategy Pattern

The MCS pattern [78, 80, 132] provides a uniform way to implement numerical algorithms and to compose them according to the specific needs.

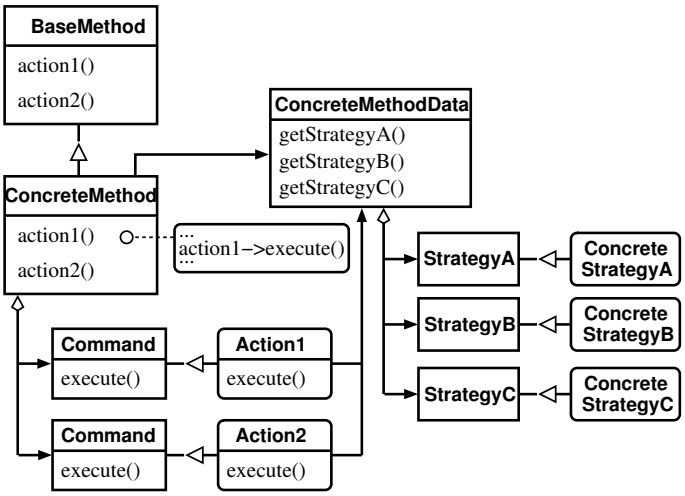


Figure 2.7: OMT diagram of the Method-Command-Strategy pattern.

As sketched in Fig. 2.7, `BaseMethod` defines an abstract interface for a specific type of algorithm (e.g. LinearSystemSolver, ConvergenceMethod, MeshCreator, SpaceMethod, ErrorEstimator ...). `ConcreteMethod` implements the virtual functions of the corresponding parent class by delegating tasks to ad-hoc `Commands`, see [10] [48], that share `ConcreteMethodData`, a tuple aggregating multiple polymorphic receivers (`Strategies`) [48].
Three levels of abstraction and flexibility can be identified in this pattern: `BaseMethod`, `Command` and `Strategies` can all be overridden, allowing one to implement the same task, at the corresponding level, in different ways. This kind of behavioral modularity allows the developers to easily re-implement or tune components (`Methods`, `Commands` or `Strategies`), and gives them the freedom to move code from one layer to another, according to convenience, taste or profiling-driven indications. The fast-path code, critical for the overall performance, can be wrapped inside `Commands` or `Strategies` and it can be substituted with more efficient implementations without implying changes in the upper layer.

Moreover, while freedom is left to define the abstract interface of a new polymorphic `BaseMethod` or `Strategy`, the interface of a `Command` is frozen, it defines only three actions, as shown in C.L. 2.23.

```
  // --- Command.hh --- //
class Command {
public:
  // constructor, destructor
  virtual void setup()   = 0; // setup private data
  virtual void unsetup()= 0; // unsetup private data
  virtual void execute()= 0; // execute the action
```

Code Listing 2.23: Command class definition

In COOLFluiD, managing the collaboration between different numerical methods is eased by the fact that `Commands` can create their own local or distributed data and share them with other `Commands` defined within other `Methods`, by making use of the `DataStorage` facility, whose details are presented in section 2.2.1.
Moreover, Perspective objects, as described in 2.3.1, can be used within the MCS pattern, at the `Strategy` level, as part of the `ConcreteMethodData`, in order to bind a numerical algorithm to the physics polymorphically.
This collaboration between MCS and Perspective patterns is an effective solution that decouples physics and numerics as much as possible. It makes possible, for instance, to employ a certain space method to discretize the equations related to different physical models, but also to apply different space methods, requiring completely different data-structures to discretize the same equations.
Interchangeability of `Methods`, `Commands`, `Strategies` can be facilitated and maximized by making them *self-registering* and *self-configurable* objects [19] [78, 80]. These last two concepts will be explained in Sec. 2.5. A deeper analysis of the MCS pattern and more implementation details are provided in [131, 132]. Among the possible `BaseMethods` that can be identified in a CFD simulation, and among those we have actually implemented in COOLFluiD, we consider now a few examples in order to show the concrete applicability of the MCS pattern.

2.4.1.1 MCS Example: SpaceMethod

In C.L. 2.24, we define the interface of a `SpaceMethod`, that takes care of the spatial discretization of the given set of PDE, according to a specified numerical scheme and on a chosen mesh.

```
// --- SpaceMethod.hh --- //
class SpaceMethod : public Method {
  public:
  SpaceMethod(string name);       // constructor
  virtual ~SpaceMethod();         // virtual destructor
  virtual void setMethod() = 0;   // setup the method
  virtual void unsetMethod() = 0; // unsetup the method

  // initialize the solution
  virtual void initializeSolution(CFbool isRestart) = 0;

  // compute the space part of residual/jacobian
  virtual void computeSpaceResidual(CFreal factor=1.0) = 0;

  // compute the time dependent part of residual/jacobian
  virtual void computeTimeResidual(CFreal factor=1.0) = 0;

  virtual void applyBC() = 0; // apply boundary conditions
};
```

Code Listing 2.24: SpaceMethod class definition

As shown in the sample code above, `SpaceMethod` inherits from a non instantiable `Method` object that provides some configuration functionalities meant to be reused by all its children, namely all the possible `BaseMethods` in Fig. 2.7.

Figure 2.8 shows a simplified class diagram of a concrete space method `ASpaceMethod` module. Specific `Commands`, `ASpaceMethodCom`, are associated to actions like *setup* and *unsetup* (creation and destruction of data needed by the employed scheme), application of *boundary conditions*, computation of the residual (plus jacobian contributions to the system matrix, in case of implicit schemes) coming from the space and time equation terms, as shown in C.L. 2.25.

All self-registering polymorphic objects, including `Commands`, are aggregated by `SelfRegistPtrs`, i.e. smart pointers with intrusive reference counting that keep ownership on them. In order to take full profit of the self-registration and self-configuration facility which will be described in Sec. 2.5, all `ConcreteMethods`, including `ASpaceMethod`, hold the `Command` names (second entry in the *std*::*pair* tuples), which are used as keys for the polymorphic creation and configuration of the `Commands` themselves. Let's consider the constructor of `ASpaceMethod`:

All the `Command` names are set to a default that can be overridden by the user in the COOLFluiD input file: these will cause the object requested by

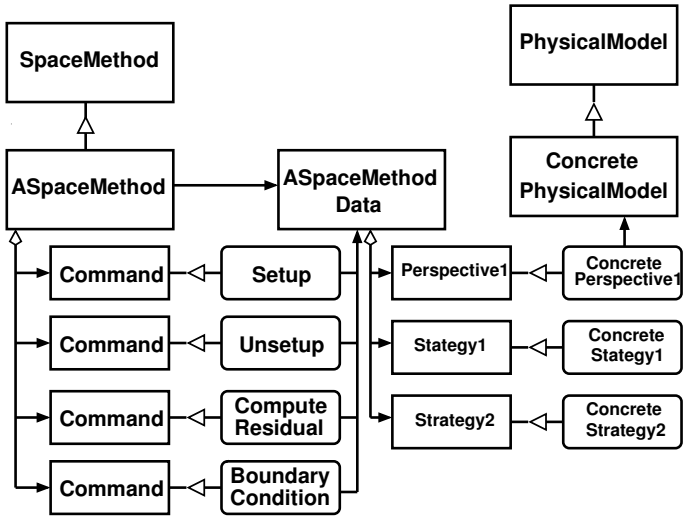


Figure 2.8: OMT diagram of the Method-Command-Strategy pattern applied to a concrete SpaceMethod, called ASpaceMethod.

the user to be instantiated, if available. As clarifying example, let's analyze the following fragment of an input file:

```
SpaceMethod = ASpaceMethod
ASpaceMethod.SetupCom = SecondOrderSetup
ASpaceMethod.UnSetupCom = SecondOrderUnSetup
ASpaceMethod.ComputeRHS = JacobRHS
```

This asks to create the `SpaceMethod` corresponding to the name `ASpaceMethod` and to select *setup* and *unsetup* `Commands` specific for a second order scheme, which might require the allocation of different or additional data than a first order one. Likewise, the `Command` with name `JacobRHS` will be used to compute the residual and jacobian contributions instead of the default one, called `StdComputeRHS`.
All `AComs` are parameterized with a policy class [10], `ASpaceMethodData`, which groups together all the Strategy objects needed by the `Commands` to fulfill their job: `Strategy1`, `Strategy2`, etc. and some Perspectives, `Perspective1`, `Perspective2`, etc. (see 2.3.1), that provide the binding to the physics.

```
// --- ASpaceMethod.hh --- //
class ASpaceMethod : public SpaceMethod {
  public:
  typedef SelfRegistPtr<Command<ASpaceMethodData> > ACom;
  // constructor, destructor, overridden virtual functions

  private:
  // data to share between ACom commands
  std::auto_ptr<ASpaceMethodData> m_data;

  std::pair<ACom,string> m_setup;   // setup Command
  std::pair<ACom,string> m_unsetup; // unsetup Command

  // Commands computing the residual/jacobian
  std::pair<ACom,string> m_computeSpaceRHS; // space part
  std::pair<ACom,string> m_computeTimeRHS;  // time part

  // Commands that initialize the solution in the domain
  std::vector<ACom> m_inits;
  std::vector<string> m_initsStr; // init Commands names

  // Commands that computes the boundary conditions (bc)
  std::vector<ACom> m_bcs;
  std::vector<string> m_bcsStr; // bc Commands names
};
```

Code Listing 2.25: ASpaceMethod class definition

```
// --- ASpaceMethod.cxx --- //
ASpaceMethod::ASpaceMethod(string name) :
SpaceMethod(name),
m_data(new ASpaceMethodData())
{
  m_setup.second = "StdSetup"; // default name
  addConfigOption("SetupCom","Setup",&m_setup.second);

  m_unsetup.second = "StdUnSetup"; // default name
  addConfigOption("UnSetupCom","UnSetup",&m_unSetup.second);

  m_computeSpaceRHS.second = "StdComputeRHS"; // default name
  addConfigOption("ComputeRHS", "Compute space residual",
  &m_computeSpaceRHS.second);
  ...
}
```

Code Listing 2.26: ASpaceMethod constructor

```
// --- ASpaceMethodData.hh --- //
class ASpaceMethodData : public ConfigObject {
  public:
  //constructor, destructor, configuration functions

  // accessor/mutators to Strategies
  SafePtr<Strategy1> getStrategy1() const;
  SafePtr<Strategy2> getStrategy2() const;

  // accessor/mutators to Perspectives
  SafePtr<Perspective1> getPerspective1() const;
  SafePtr<Perspective2> getPerspective2() const;

  // other accessors/mutators ...

  private:

  SelfRegistPtr<Strategy1> m_strategy1;
  string m_strategy1;
  ...

  SelfRegistPtr<Perspective1> m_perspective1;
  string m_perspective1;

  // ... the same for all the other objects
};
```

Code Listing 2.27: ASpaceMethodData class definition

A more detailed example showing the application of the pattern for implementing a FV method is given in Sec. 5.2.8. Other subclasses of SpaceMethod, such as `Finite Element` or `Residual Distribution`, are implemented in a similar way.

2.4.1.2 Collaboration between two Methods: ConvergenceMethod and LinearSystemSolver.

We consider now an example of interaction between two different numerical methods. Figure 2.9 shows the collaboration between two abstract methods: `ConvergenceMethod`, responsible of the iterative procedure, and `LinearSystemSolver`. In this case, `BackwardEuler`, an implicit convergence method, delegates polymorphically the solution of the resulting linear system to `PetscLSS`, which interfaces the `PETSc` library [73].
`BackwardEuler` makes use of `Commands` for the setup, unsetup and solution update, while `PetscLSS` let `Commands` implement the set up, unsetup and

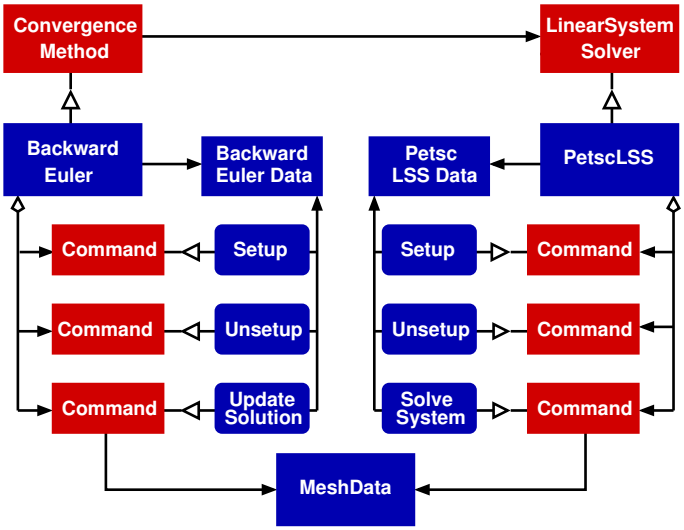


Figure 2.9: MCS pattern applied to two interacting modules: a Backward Euler convergence method and a Petsc linear system solver.

solution of the linear system.
The class definition of `ConvergenceMethod` is the following:
In the code above, `MultiMethodHandle` is a lightweight proxy object that hides the knowledge of multiplicity: it controls the access to one or more underlying Methods with the same polymorphic type and dispatches specified actions on all of them sequentially, similarly to what a *std*::*for_each* function would do [149]. The purpose of accessing `Method` objects through `MultiMethodHandles` is to offer transparent support for weakly coupled simulations, where, in the same process, two or more different linear systems are assembled by one or more `SpaceMethods` and must be solved one after the other.
`SpaceMethod` (SM) and `LinearSystemSolver` (LSS) are the collaborator Methods: a concrete `ConvergenceMethod` uses them polymorphically via `MultiMethodHandles`, without knowledge of their concrete type or their actual number.
As a result, concrete collaborator `Methods` are completely interchangeable with other ones with the same polymorphic type. When running in parallel, in the function `synchAndComputeRes`() the global storage of state vectors

```
// --- ConvergenceMethod.hh --- //
class ConvergenceMethod : public Method {
public:
  //constructor, virtual destructor, configuration methods

  virtual void takeStep() = 0;    // Take one timestep
  virtual void setMethod() = 0;   // Sets up private data
  virtual void unsetMethod() = 0; // Clears up private data

  // Sets collaborator methods (SpaceMethod, LinearSystemSolver)
  void setCollaborator(MultiMethodHandle<SpaceMethod> spaceMtd);
  void setCollaborator(MultiMethodHandle<LinearSystemSolver>
      lss);
protected:
  // Synchronizes the states and computes the norm of the
      residual
  void synchAndComputeRes(CFbool computeResidual);

protected: //data
  // Space Method used to compute spatial discretization
  MultiMethodHandle<SpaceMethod> m_spaceMtd;

  // LinearSystemSolver used to solve the linear system (if any)
  MultiMethodHandle<LinearSystemSolver> m_lss;

  // handle to the global storage of states
  DataHandle<GLOBAL,CFreal> m_statedata;

  // handle to the global storage of nodes
  DataHandle<GLOBAL,CFreal> m_nodedata;
};
```

Code Listing 2.28: ConvergenceMethod class definition

and nodal coordinates are accessed via `DataHandle` and asked to synchronize the underlying parallel array, as shown in C.L 2.29.
In a parallel simulation, the norm of the residual is calculated with a collective operation, between the beginning and the end of the synchronization process, in order to overlap communication and computation and maximize the efficiency of the operation. This is one of the only few COOLFluiD code fragments that makes direct use of the global `DataHandles` and needs to conditionally apply a different treatment in parallel or in serial. We consider now a backward Euler time stepper, implemented as a subclass `BwdEuler` of the `ConvergenceMethod`:
A different `Command` and its name are associated to each one of the pure virtual functions declared by the parent `ConvergenceMethod`. We report

```
// --- ConvergenceMethod.cxx --- //
void ConvergenceMethod::synchAndComputeRes(CFbool
    computeResidual)
{
  // after each update phase states and nodes
  // have to be synchronized
  if (isParallel()) {
    m_statedata->beginSync();
    m_nodedata->beginSync();
  }
  // computation of the residual
  if (computeResidual) {...}

  if (isParallel()) {
    m_statedata->endSync();
    m_nodedata->endSync();
  }
}
```

Code Listing 2.29: ConvergenceMethod synchronization

```
// --- BwdEuler.hh --- //
class BwdEuler : public ConvergenceMethod {
public:
  //constructor, virtual destructor, configuration methods
  typedef SelfRegistPtr<Command<BwdEulerData> > BwdEulerCom;

  void takeStep();    // Take one timestep
  void setMethod();   // Sets up private data
  void unsetMethod(); // Clears up private data
private:
  //data shared by BwdEuler Commands
  std::auto_ptr<BwdEulerData> m_data;
  std::pair<BwdEulerCom,string> m_setup;     // set up
  std::pair<BwdEulerCom,string> m_unSetup;   // unsetup
  std::pair<BwdEulerCom,string> m_updateSol; // update solution
};
```

Code Listing 2.30: BwdEuler class definition

here after the implementation of `takeStep`(), where the polymorphic usage of the collaborators (SM and LSS) and of the solution updating `Command` is shown.
The code is readable, concise, independent from the actual type and number of SM or LSS, extremely flexible, since each part of the algorithm (`Commands` or `Methods`) can be replaced at run-time without any performance overhead. In fact, the frequency of virtual calls in question is exceptionally low and,

```
// --- BwdEuler.cxx --- //
void BwdEuler::takeStep()
{
  // compute residual and jacobian contributions for
  // the spatial and time dependent term of the equations
  m_spaceMtd.apply(mem_fun(&SpaceMethod::computeSpaceResidual));
  m_spaceMtd.apply(mem_fun(&SpaceMethod::computeTimeResidual));

  // solve the resulting linear system
  m_lss.apply(mem_fun(&LinearSystemSolver::solveSys));

  m_updateSol.first->execute(); // update the solution

  // synchronize nodes and states, compute the intermediate
      residual
  ConvergenceMethod::synchAndComputeRes(true);
}
```

Code Listing 2.31: BwdEuler takeStep() function

therefore, does not have an impact on the run-time speed.
The class definitions of the parent LSS and of its subclass `PetscLSS` are shown in C.L. 2.32 and 2.33.
`PetscLSS` delegates tasks to specific `Commands`,`PetscLSSCom`, sharing some data (`PetscLSSData`), such as references to the (parallel) Petsc matrix and the (parallel) Petsc vectors involved in the solution of the linear system.
Only the `PetscLSSCom` can make direct use of PETSc [73] objects like `PC` or `KSP`, which are aggregated by `PetscLSSData`. The knowledge of a specific LSS, `PetscLSS`, in this case, is not assumed anywhere in the numerical modules, thanks to the use of abstractions such as `LSSMatrix`, `BlockAccumulator` and `LSSIdxMapping`. `LSSMatrix` is the parent system matrix from which `PetscMatrix` derives. `BlockAccumulator` bundles blocks of values to be inserted in the matrix. `LSSIdxMapping` stores a mapping from the local numbering to an optimal LSS-dependent global one.
As represented in Fig. 2.9, all the involved `Commands` have acquaintance of `MeshData`, which provides access to `DataStorage` and `DataHandles`, and can therefore use and modify bulk data, qualified by name and type, as explained in 2.2.1. In other words, while, on the one hand, each `ConcreteMethodData`, such as `PetscLSSData` and `BackwardEulerData`, allows intra-Method sharing of ConcreteMethod-dependent data among Commands, on the other hand, `MeshData` is the vehicle for inter-Method data exchange.
The sample code in C.L. 2.35 should help clarifying the last statement in the case of two `Commands`, namely `UpdateSol` in `BackwardEuler` and `SolveSys`

```
// --- LinearSystemSolver.hh --- //
class LinearSystemSolver : public Method {
public:
  //constructor, virtual destructor, configuration methods

  virtual void solveSys() = 0;    // solve the linear system
  virtual void setMethod() = 0;   // setup private data
  virtual void unsetMethod() = 0; // clear up private data

  // create a block accumulator with ad-hoc internal storage
  virtual BlockAccumulator* createBlockAccumulator
  (int nbRows, int nbCols, int subBlockSize) const = 0;

  // accessor/mutator for local to global LSS index mapping
  SafePtr<LSSIdxMapping> getLocalToGlobalMapping()
  {return &_localToGlobal;}

  // get the system matrix
  virtual SafePtr<LSSMatrix> getMatrix() const = 0;
private:
  // idx mapping from local to LSS global
  LSSIdxMapping m_localToGlobal;
};
```

Code Listing 2.32: LinearSystemSolver class definition

in `PetscLSS`.

Data stored in `MeshData` are allowed to cross the Method scope and can be used in `Commands` belonging to different `Methods`. The keys for the data exchange are the storage name and type, that must both match in all the method `Commands` that need the same data.

In parallel simulations, nothing changes, since the `Commands` implementing numerical algorithms work only with `LOCAL` data, as they would do in a serial run. Functions that demands access to the `GLOBAL` storage and perform some parallel action, like the above mentioned `synchAndComputeRes`(), are exceptional.

Furthermore, the example of the collaboration between `ConvergenceMethod` and LSS demonstrates the suitability of the MCS pattern to interface existing libraries, PETSc in this case, without exposing any detail of their actual implementation to their clients.

The *Trilinos* package [72] has also been successfully integrated in COOLFluiD by means of the MCS pattern. While the class definitions of the corresponding concrete linear system solver `Method` and the related `Commands` are basically similar to the Petsc's ones, the interface of `TrilinosLSSData` is defined

```
// --- PetscLSS.hh --- //
class PetscLSS : public LinearSystemSolver {
public:
  //constructor, virtual destructor, configuration methods
  typedef SelfRegistPtr<Command<PetscLSSData> > PetscLSSCom;

  void setMethod();   // setup private data
  void unsetMethod(); // clears up private data
  void solveSys();    // solve the linear system

  // create a block accumulator with ad-hoc internal storage
  BlockAccumulator* createBlockAccumulator() const;

  // get the system matrix
  SafePtr<LSSMatrix> getMatrix() const
  {return &m_data->getMatrix();}

private:
  // The data to share between PetscLSSCom Commands
  std::auto_ptr<PetscLSSData> m_data;
  std::pair<PetscLSSCom,string> m_setup;    // setup Command
  std::pair<PetscLSSCom,string> m_unSetup;  // unsetup Command
  std::pair<PetscLSSCom,string> m_solveSys; // solver Command
};
```

Code Listing 2.33: PetscLSS class definition

```
// --- PetscLSSData.hh --- //
class PetscLSSData {
public:
  //constructor, destructor, configuration methods

  PetscVector& getSolVec() {return m_xVec;} //Petsc solution
      array
  PetscVector& getRhsVec() {return m_bVec;} //Petsc rhs array
  PetscMatrix& getMatrix() {return m_aMat;} //Petsc matrix
  PC& getPreconditioner() {return m_pc;} //Petsc preconditioner
  KSP& getKSP() {return m_ksp;} //Petsc Krylov solver

  // accessors to various Petsc parameters, private data, etc.
};
```

Code Listing 2.34: PetscLSSData class definition

in C.L. 2.37.
In summary, the application of the MCS pattern helps to encapsulate both the single numerical algorithms and their collaborations with other classes of schemes. This contributes to enforce the multi-component-oriented char-

```
// --- UpdateSol.cxx --- //
void UpdateSol::execute()
{
  // get the local state vectors and rhs from MeshData
  DataHandle<LOCAL,State*> states = MeshData::getLocalData()->
    getData<State*>("states");

  DataHandle<LOCAL,CFreal> rhs = MeshData::getLocalData()->
    getData<CFreal>("rhs");

  // compute the solution update ...
}
```

Code Listing 2.35: UpdateSol execute() function

```
// --- SolveSys.cxx --- //
void SolveSys::execute()
{
  // get the local state vectors and rhs from MeshData
  DataHandle<LOCAL,State*> states = MethData::getLocalData()->
    getData<State*>("states");

  DataHandle<LOCAL,CFreal> rhs = MethData::getLocalData()->
    getData<CFreal>("rhs");

  // get the method data shared by all PetscLSS Commands
  PetscMatrix& mat = getDataPtr().getMatrix();
  PetscVector& rhsVec = getDataPtr().getRhsVector();
  PetscVector& solVec = getDataPtr().getSolVector();
  KSP& ksp = getDataPtr().getKSP();

  // perform final assembly and ask PETSc to solve the system
}
```

Code Listing 2.36: SolveSys execute() function

acter of the COOLFluiD framework, where each component can be replaced by another one with the same polymorphic type, without affecting the client code.

2.5 Dynamical plug-ins

In COOLFluiD, each concrete numerical method or physical model is enclosed in a separate *plug-in* library or module. This plug-in policy, which provides our platform with significant modularity and extensibility, relies

```
// --- TrilinosLSSData.hh --- //
class TrilinosLSSData {
public:
  // map for the individual IDs of the unknowns
  Epetra_Map* getEpetraMap() {return m_map;}

  TrilinosVector* getSolVec() {return &m_xVec;} //solution
      vector
  TrilinosVector* getRhsVec() {return &m_bVec;} //rhs vector
  TrilinosMatrix* getMatrix() {return &m_aMat;} //system matrix
  AztecOO* getKSP() {return &_ksp;} // Aztec Krylov solver

  // various options and parameters to control convergence
  // and tune the solver to satisfy the user needs ...
 };
```

Code Listing 2.37: TrilinosLSSData class definition

heavily on two complementary techniques, namely *self-registration* and *self-configuration* of objects, whose basic principles re explained in this section.

2.5.1 Self-registering objects.

The self registration of objects, pioneered by [19] in a C++ context, automatizes the creation of polymorphic objects so that

1. the details of the concrete types are not exposed to the interface of the core code;
2. all possible objects are created using the same interface;
3. the core implementation should be unaffected by future unpredictable extensions of the code.

As a result, both implementation and compilation dependencies are considerably reduced, allowing developers to enclose each new functionality as a separate component, compile it separately and loading it in dynamically on demand into the platform, without even needing a partial recompilation. Our implementation of this technique, which is of great help in easing the integrability of new components in COOLFluiD, has been inspired by [39], but we have evolved it into a more sophisticated and flexible approach, only partially explained in [74, 78] which is described here after.
A singleton [48] `Factory` class for a generic polymorphic object type (here represented by the template parameter `OBJ`) is defined in C.L. 2.38.

```
// --- Factory.hh --- //

template <class OBJ>
class Factory {
public:
  // get a reference to the Factory itself
  static Factory<OBJ>& getInstance()
  {static Environment::Factory<BASE> obj; return obj;}

  // register a provider
  void regist(Environment::Provider<OBJ>* provider)
  {m_map[provider->getName()] = provider;}

  // get the provider corresponding to the given key name
  SafePtr<typename OBJ::PROVIDER> getProvider(string pName)
  {return dynamic_cast<typename OBJ::PROVIDER*>
    (m_map.find(pName)->second);}

private:
  std::map<string, Provider<OBJ>*> m_map; // providers database
};
```

Code Listing 2.38: Definition of Factory class

Factory encapsulates an associative container, a std::map, which stores pairs key-value, where the key is a literal string and the value is a pointer to a `Provider` for objects of polymorphic type `OBJ`. Each `Provider` of a polymorphic object of dynamic type `OBJ` registers its name and itself in the corresponding `Factory` class at construction time, as shown in C.L. 2.39.

```
// --- Provider.hh --- //
template <class OBJ>
class Provider {
public:

  // constructor: each Provider registers itself in Factory
  Provider(const string& name) : m_name(name)
  {Factory<OBJ>::getInstance().regist(this);}

  virtual ~Provider() {} // virtual destructor
  string getName() {return m_name;} // returns the name

private:
  string m_name; // provider name
};
```

Code Listing 2.39: Definition of Provider class

Another intermediate abstract class, called `ConcreteProvider`, needs to be defined: it derives from Provider and defines a virtual `create` function. In fact, several template specializations [160] of `ConcreteProvider` are provided, one for each foreseen number of parameters (0,1,2) accepted by the `create` function. Two of those class specializations are shown in C.L. 2.40.

```
// --- ConcreteProvider.hh --- //
template <class OBJ, int NBARG = 0>
class ConcreteProvider : public Provider<OBJ> {
public:
 // create function accepting 0 arguments
 virtual SelfRegistPtr<OBJ> create() = 0;
};

template <class OBJ>
class ConcreteProvider<OBJ,1> : public Provider<OBJ> {
public:
  // definition of the object type and argument type
  typedef OBJ BASE_TYPE;
  typedef typename OBJ::ARG1 BASE_ARG1;

  // create function accepting 1 argument of type BASE_ARG1
  virtual SelfRegistPtr<OBJ> create(BASE_ARG1 arg1) = 0;
};
```

Code Listing 2.40: ConcreteProvider class

The type `SelfRegistPtr` appearing in C.L. 2.40 is just a proxy pointer class for handling self-registering objects in safety.
The implementation for the `create` functions defined by `ConcreteProvider` is given by `ObjectProvider` in C.L. 2.41, which also needs to be partially specialized for accomplishing the task.
In particular, `ObjectProvider` is parameterized with three template arguments: the concrete type `CONCRETE`, its corresponding polymorphic type `OBJ` and the number of arguments for the creational function.
We now consider an example to show how the whole machinery works in practice. We define an abstract class `Shape` whose constructor accepts one string parameter. In order to be able to apply the self-registration of all its possible derived classes, the typedefs `PROVIDER` and `ARG1` must be explicitly provided with public access in the `Shape` class definition, as shown in C.L. 2.42.
At this point, whatever shape object (`Cube`, `Circle`, `Polyhedron`, etc.) deriving from the parent `Shape` can be self-registered by instantiating a global variable with type equal to the corresponding `ObjectProvider`, somewhere

```
// --- ObjectProvider.hh --- //

template <class CONCRETE, class OBJ, int NBARGS = 0>
class ObjectProvider : public OBJ::PROVIDER {
public:
  // implementation of the create function with 0 arguments
  SelfRegistPtr<OBJ> create()
  {return SelfRegistPtr<OBJ>(new CONCRETE(), this);}
};

template <class CONCRETE, class OBJ>
class ObjectProvider<CONCRETE, OBJ, 1> : public OBJ::PROVIDER {
public:
  // implementation of the create function with 1 arguments
  SelfRegistPtr<OBJ> create(typename OBJ::ARG1 arg)
  {return SelfRegistPtr<OBJ>(new CONCRETE(arg), this);}
};
```

Code Listing 2.41: ObjectProvider class

```
// --- Shape.hh --- //

class Shape {
public:
  typedef ConcreteProvider<Shape, 1> PROVIDER;
  typedef string& ARG1;

  Shape(string& name);
  virtual double getVolume() = 0;
};
```

Code Listing 2.42: Shape class

in the code (e.g. in the implementation file):

```
// --- Cube.cxx --- //
ObjectProvider<Shape, Cube, 1> cubeProvider("Cube");

// --- Circle.cxx --- //
ObjectProvider<Shape, Circle, 1> circleProvider("Circle");
```

Code Listing 2.43: Instatiation of a ObjectProvider for a Cube in the implementation file

The registration of all available `ObjectProviders` occurs before entering the `main()` function of the code, since `Factories` live on the static memory and `ObjectProviders` are instantiated as global variables.

In our example, the string "Cube" (or "Circle"), accepted by the corresponding provider constructor, can then be used as a key to ask the singleton Factory to query its database and create the corresponding polymorphic object whenever needed.

```
1
  string keyName = shapeName(); //can be "Cube" or "Circle" etc.

4 SelfRegistr<Shape> shape = Factory<Shape>::getInstance().
                getProvider(keyName)->create(keyName);
```

Code Listing 2.44: Creation of a polymorphic shape object

This example shows that whatever the actual shape, the way to create it remains unique and the piece of code in C.L. 2.44 doesn't need to be changed anymore even if new shapes are developed and integrated in the platform.
In COOLFluiD, all polymorphic objects are created with this technique, once their corresponding names (e.g. keyName in the example in C.L. 2.44) are read from the CFcase input file.

2.5.2 Self-configurable objects.

In COOLFluiD, objects can be self-configurable [78, 80, 131], i.e. they can create and set their own data. In order to make an object self configurable, some steps have to be followed. First, the object must derive from the abstract parent class ConfigObject, which implements the configuration algorithm [131].
Second, a static member function defineConfigOptions() must be defined and implemented. The implementation consists in declaring the type, name and description for each one of the options, as in the illustrative example in C.L. 2.45 for a TimeStepper object:
Once declared, the options must be linked to the actual data to configure and this is done inside the constructor, as in the example in C.L. 2.46.
In particular, the strings "CFL" and "LSS" in our example are the configuration keys and they are used to map the values of the private data m_cfl and m_lssName with the ones read from the configuration case file (CFcase).
In order to configure our TimeStepper object, the end-user should write something like

```
TimeStepper = BwdEuler  # name of the concrete TimeStepper
BwdEuler.CFL = 3.5      # user-defined CFL number
BwdEuler.LSS = TRILINOS # user-defined linear system solver
```

```
// --- TimeStepper.cxx --- //
void TimeStepper::defineConfigOptions(OptionList& op)
{
  op.addConfigOption<double>
                ("CFL", "CFL number");
  op.addConfigOption<string>
                ("LSS", "Linear System Solver name");
}
```

Code Listing 2.45: Example of implementation of `addConfigOption()` function for a self-configurable TimeStepper

```
// --- TimeStepper.cxx --- //
TimeStepper::TimeStepper() : ConfigObject()
{
  addConfigOptionsTo(this);

  m_cfl = 1.0; // default CFL value
  setParameter("CFL", &m_cfl);

  m_lssName = "PETSC"; // default LSS name
  setParameter("LSS", &m_lssName);
}
```

Code Listing 2.46: Example of configuration directives in the constructor of a self-configurable TimeStepper

in the CFcase file. `BwdEuler` is the self-configuration value for `TimeStepper` which is the configuration key for the homonymous polymorphic object. `BwdEuler` is also the self-registration key (see Sec. 2.5.1) for the concrete time stepper class that will then be instantiated and will configure itself with the specified CFL number and linear system solver name.
In COOLFluiD, Methods, Commands, Strategies, PhysicalModels etc. are all self-configurable objects and this, together with the self-registration technique, gives the end-user full power on their settings. The actual implementation of the described technique, due to [131], allows users to input whatever kind of data (including analytical functions) from file, environmental variables or command line options, but also to define interactive parameters or to substitute on-the-fly polymorphic objects during the simulation.

Chapter 3

High Performance techniques

3.1 Parallelization

In order to tackle the more and more challenging multi-physics problems that science and technology face on a daily base, and in order to keep up with the rapid progress in the development of architectures for large scale computing, modern software for scientific purposes must provide support for parallel and/or distributed computing. As far as CFD is concerned, complex 2D and 3D simulations involving dozens of equations and/or millions of grid points are becoming routine not only in an industrial context but also within academic and research institutions. That's why, one of the fundamental design requirements of the COOLFluiD framework was to provide a user- and developer-friendly parallelization. While in Sec. 2.2 we have already explained how a transparent handling of the data has been achieved for both serial and parallel computations by means of the DataStorage and DataHandles facilities, this section focuses on the description of some algorithms based on the standard MPI [2, 63, 112] that have been developed and implemented by the author for supporting parallel I/O and domain decomposition on arbitrarily complex unstructured hybrid meshes. As a result, a user of COOLFluiD can conveniently always start (or restart) a simulation from a single mesh file in the native CFmesh format and save the solution in the same format on a single file, even when running in parallel.
Moreover, the algorithms here described are rather flexible and can already handle I/O for the different data-structures associated to hybrid cell-vertex and cell-centered mesh data, but also for high order elements, as required by the Spectral Finite Volume, DG or High Order RD space discretizations.

3.1.1 Parallel Mesh Reading

Many parallel codes for unstructured meshes developed in recent years such as THOR [154], US3D [109], Snehurka [40] don't support parallel mesh read-

ing: they simply load the full mesh and perform its decomposition serially, on just one processor, by means of the graph partitioning algorithm implemented in the METIS [67] library. In COOLFluiD, however, we choose to provide a fully parallel algorithm that works on distributed mesh data since the start, with the aim of maintaining full scalability in memory requirements in all phases of the simulation, potentially supporting the loading and partitioning of meshes with extremely large sizes, given that a sufficient number of CPUs is available.
The following four main phases are identifiable while reading the mesh file:

1. reading of the elements data;
2. mesh decomposition;
3. construction of the overlap region;
4. reading of the topological region sets;
5. reading of the nodes and states.

3.1.1.1 Reading of the elements data

During the first phase, each processor reads some global counts (total number of nodes, states, elements, element types) and the full connectivity data for the elements (a sequence of integers ordered by element type), but only stores the continuous portion of it associated to its rank, as shown in C.L. 3.1. According to this scheme, the first processor gets more elements than the others, if the total element count cannot be exactly divided by the processor count.

3.1.1.2 Parallel Mesh Decomposition

The mesh decomposition is partially accomplished by the ParMetis library [67], which supports both nodewise and cellwise partitioning, but in our case, we chose to stick to the latter, because it is more suitable for hybrid meshes. After having read the whole element list data, each processor calls ParMetis and passes its locally stored element-node connectivity to it in Compressed Sparse Row (CSR) format. Once the parallel partitioning is performed, each processor gets back from ParMetis an array indicating the unique destination processor for each locally stored element. Each element must now be redistributed to the only processor which owns it. This can be achieved with a total exchange action [112], a collective communication

```
// during the setup phase, the number of elements to
// be read by each processor is stored in an array
vector<int> nbElemPerRank(nbProcessors);
const int ne = totalNbElements/nbProcessors;

// the first processor can be oversized
nbElemPerRank[0] = ne + totalNbElements%nbProcessors;
for (int i = 1; i < nbProcessors; ++i) {
  nbElemPerRank[i] = ne;
}

int firstElementID = 0;
for (int rank = 0; rank < p; ++rank) {
  firstElementID += nbElemPerRank[rank];
}
const int lastElementID = firstElementID + nbElemPerRank[p];

// loop over the element types (more than one in hybrid
    meshes)

  // loop over the elements in each type

    // store only connectivities corresponding to elements
    // between first and last elementID for the current rank p
```

Code Listing 3.1: Reading of the element-node and element-state connectivity in a generic processor with rank p

operation in which each process sends a distinct collection of data to every other process. The total exchange is accomplished in two steps. First, the size of the data to globally send/receive to/from each processor is communicated by means of the `MPI_Alltoall` function, which supports transfer of data arrays all with the same size. Second, the local elements data packed into two unidimensional arrays (described here after) are distributed via the `MPI_Alltoallv` function, which allows different processors to transfer data arrays with different sizes. Within this context, it is worth to have a look at the `ElementDataArray` proxy class in C.L. 3.2 which has been defined to encapsulate the unidimensional arrays of element data to be distributed, allowing the algorithm to easily iterate and operate over elements in the case of logically hybrid unstructured meshes.

ElementDataArray stores as private data two arrays of integers. In the first one, called `m_eData` in C.L. 3.2 some connectivity information are sequentially stored for all elements: to this end, each element is defined by a global

```
// --- ElementDataArray.hh --- //
class ElementDataArray {
public:
  class Itr; // forward declaration of an iterator class

  // constructor, destructor, memory allocation functions
  int sizeData() const; // 1D data array size
  int getNbElements();  // number of elements
  int* startData(); // get a pointer to the data array
  int* startPtr(); // get a pointer to the pointers array

  void setBeginEptr(); // start inserting one element
  void setEndEptr();   // stop inserting one element

  void add(ElementDataArray& e); // add another ElementDataArray
  void addElement(const Itr& itr); // add one element

  Itr begin(); // return an iterator to the element data
      beginning
  Itr end();   //return an iterator to the element data end
  Itr getElement(int iElem); // get the iterator to the
                               // specified element
private:
  std::vector<int> m_eData; // 1D array storing element data
  std::vector<int> m_ePtr;  // array giving the start ID of a
                              // single element data
};
```

Code Listing 3.2: Class definition for ElementDataArray

ID, the number of its nodes and states, the list of node and state global IDs (i.e. element-node and element-state connectivities). These data can be conveniently inserted in ElementDataArray or accessed via an iterator `ElementDataArray::Itr`, whose class definition is shown in C.L. 3.3.

In the second array of integers, named `m_ePtr` in C.L. 3.2, the IDs corresponding to the beginning of each element in `m_eData` are stored. This 2-array data structure, which extends the data format required by ParMetis [67], is especially designed for handling hybrid meshes, where different data size are associated to different element types and is suitable for minimizing the parallel communication: two calls to the `MPI_Alltoallv` function suffice to redistribute all the element data among processors, so that each processor gets the elements belonging to the mesh partition defined by ParMetis.

Figure 3.1 shows the unstructured mesh of a cube after having been decomposed into four partitions. The grid is hybrid and different element types correspond to different colors: red tetrahedra, green pyramids, blue prisms

```
// --- ElementDataArray.hh --- //
class ElementDataArray::Itr {
public:
    // constructors, destructor
    bool operator!= (const Itr& other); // overloading of !=
        operator
    bool operator== (const Itr& other); // overloading of ==
        operator
    void operator++(); // overloading of post-increment operator

    int size() const; //size of the data corresponding to one
        element
    int getGlobalID() const; // get the element global ID
    int getNbNodesInElem() const; // number of nodes in element
    int getNbStatesInElem() const; // number of states in
        element
    int getNode(int iNode) const; // get one element node
    int getState(int iState) const; // get one element state

private:
    int* m_eDataPtr; // pointer to ElementDataArray::m_eData
    int* m_ePtrPtr;  // pointer to ElementDataArray::m_ePtr
};
```

Code Listing 3.3: Definition of an iterator class for ElementDataArray

and cyan hexahedra.

3.1.1.3 Construction of the Overlap Region

Once that elements have been distributed to the corresponding target processor, an overlap region must be built in order to allow numerical algorithms to work on distributed data with all the needed information. The overlap region consists of a set of elements to be shared among different partitions, in which the nodes and states are not all updated by the same processor. A typical situation is depicted in Fig. 3.2, where, after a cell-wise partitioning, a 1-layer overlap region has been built: it includes all the vertex neighbors of the local cells attached to the partition boundary in each process. The algorithm that builds the overlap region works as follows.

1. At first, each processor flags as updatable all the local node and state IDs while looping over the local elements.

2. Each processor broadcasts the local elements to all the other processors. While looping over the received connectivity data, if the node/s-

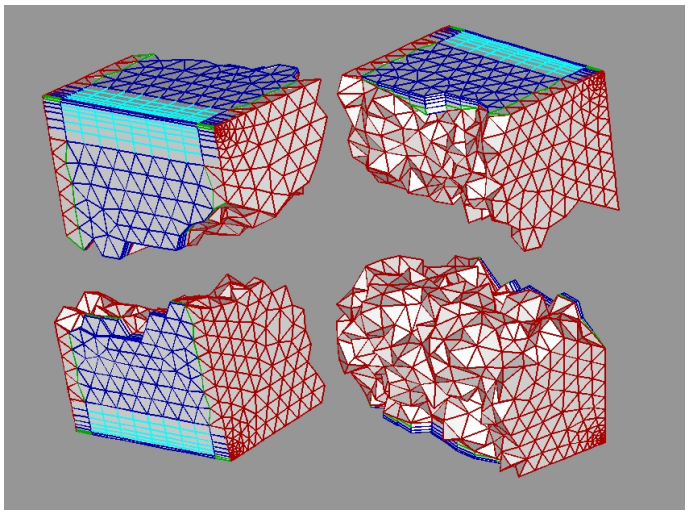


Figure 3.1: View on an unstructured hybrid grid of a cube after the partitioning, where tets are indicated in red, pyramids in green, prisms in blue and hexa in cyan.

tate is marked as local but is broadcast by a process of lower rank, the node/state is marked as ghost in this process. Moreover, if the currently processed "foreign" element shares at least one node with the local elements in the current partition, the current element is appended to a local ElementDataArray for the overlap elements.

3. The previous step is performed recursively for a number of times equal to the number of overlap layers selected by the user, depending on the space discretization algorithm. For example, a cell-centered finite volume method requires 2 layers of overlap if a least square reconstruction [15, 16] is applied, but at least 3 layers are needed if a MUSCL extrapolation [82] for structured grids is chosen. As a real-life example, Figs. 3.3a and 3.4a show the partitioned surface of a 3D unstructured mesh for the EXPERT vehicle (see Sec. 6.3) when 1 or 2 layers of overlap are selected. Corresponding zoomed views in the region where the 3 partitions intersect can be seen in Figs. 3.3b and 3.4b.

4. Two separate lists for updatable and ghost nodes/states are created

and will be used by the parallel `DataHandle` presented in Sec. 2.2.1.1 to build a synchronization pattern. Furthermore, the ElementDataArray storing the overlap elements is appended to the other one listing the local elements.

5. Element data are reordered by element type, in such a way that elements of the same type are stored contiguously in each processor. This rearrangement is mandatory and recommended in order to simplify the handling of elements for the numerical solver and for the parallel writing algorithm.

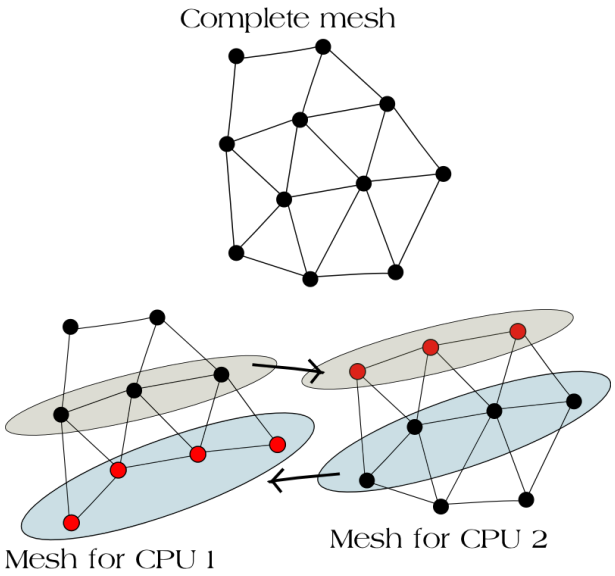
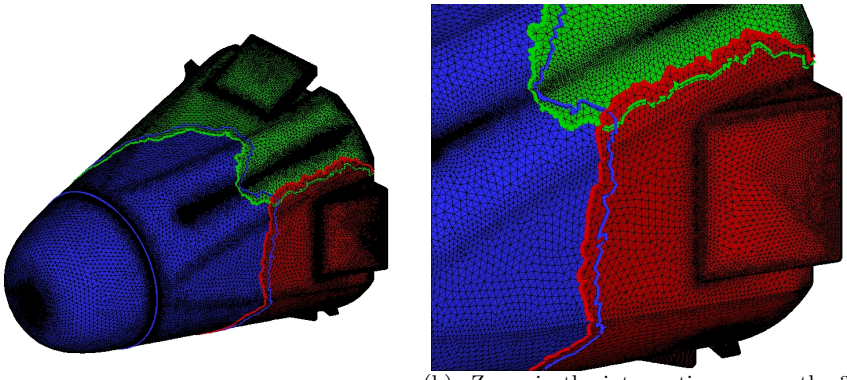


Figure 3.2: Cell-wise mesh partitioning that shows updatable and ghost states (in red) in a 1-layer overlap region.

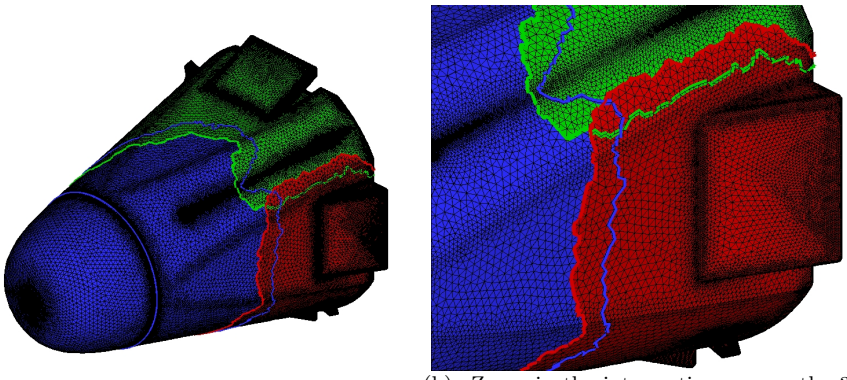
The case in Fig. 3.2 refers to the partitioning for a cell-vertex algorithm (e.g FEM, RD) with isoparametric elements, where nodes and states coincide and 1-layer of overlap is enough to guarantee all the data needed in order to assemble the residual and the residual jacobian for updatable states in each processor. During a synchronization operation, which typically occurs after each time or newton step, the solution stored in the ghost states, i.e. states



(a) Global view of the surface mesh

(b) Zoom in the intersection among the 3 partitions

Figure 3.3: Partitioned surface mesh for the EXPERT vehicle with 1-layer overlap.



(a) Global view of the surface mesh

(b) Zoom in the intersection among the 3 partitions

Figure 3.4: Partitioned surface mesh for the EXPERT vehicle with 2-layer overlap.

not updatable by the current process, gets updated by the only processor owning them. With regard to this, an extensive description and performance evaluation of synchronization algorithms can be found in [68].

3.1.1.4 Reading of the Topological Region Sets

Once the overlap region has been built, enough information is available in each processor in order to be able to read the connectivities of the TRS's and select the boundary faces belonging to each partition, without needing additional inter-processor communication.
While reading all the TRS data from the file, each processor checks if the global ID of the first node of the current boundary face is referenced by any local element (including those inside the overlap region). If that's the case, the algorithm checks if, amongst all the neighbor elements of the first boundary face node, there is one element referencing all the face nodes. If such an element exists, the boundary face is stored as local else it is discarded, because it can only belong to a neighbor partition.

3.1.1.5 Reading of the Nodes and States

Finally, each processor reads all node coordinates and state variables, but only stores those whose global IDs have been marked as local updatable or ghost during the construction of the overlap region.

3.1.2 Parallel Mesh Writing

Besides a parallel reader for a CFmesh file, a parallel writer for such a format has also been implemented. Writing an unstructured mesh file in parallel is not an easy task, especially if we add the constrain of maintaining the same exact element, node and state numbering of a serial file, independently from the number of used processors. In summary, the writing algorithm lets the master processor gather data from all the other processors, while taking care of preserving the order as in the original serial file, and writes them on the output file by ranges.
For consistency with the reading algorithm, element and TRS connectivities are written first, followed by node coordinates and state vectors.

3.1.2.1 Parallel writing by ranges

A range corresponds to a key concept in our implementation: before communicating any kind of massive data (elements, boundary faces, nodes or states), each processor defines the same number of ranges for communicating data, based on the global IDs of the data to send and the global size of each send.
For instance, in the case of an hybrid mesh, each element type is associated

to a range set, each one including a number of ranges equal to the number of processors N_p. The total count of elements per type N_et is divided by the number of sends (= N_p by default). The first range corresponds to the first $N_{et}/N_p + N_{et}\%N_p$ elements and is defined by $[0,N_{et}/N_p + N_{et}\%N_p]$, i.e. the first and the last global IDs of the elements to send, and the size of the data (integers) to communicate $(N_{et}/N_p + N_{et}\%N_p) * N_{ns}$, with N_{ns} being the number of node+state IDs for the current element type. The following ranges for the current element type all concern N_{et}/N_p elements and $(N_{et}/N_p * N_{ns}$ integers to communicate.
Each range information, together with the corresponding list of the local IDs of data to send, is registered in a `WriteListMap` object, whose interface is given in C.L. 3.4.
In order to write a certain data type (elements, boundary faces, nodes or states) in parallel, a loop over the ranges is performed in each processor and if the `WriteListMap::List` (see C.L. 3.4) of local IDs associated to the current range is not empty, the corresponding data are inserted in a preallocated array and gathered in the master process via a call to `MPI_Reduce`. After each communication (1 per range), the master process outputs the chunks of globally collected and ordered data to the CFmesh file. In the case of the elements, which are ordered by type, the whole data will be written by the master process after $N_t \times N_p$ reduce operations, where N_t is the number of element types. A similar procedure is followed for writing the connectivity of the boundary faces (ordered by TR in each corresponding TRS instead of by element type as the elements), the nodes and states.

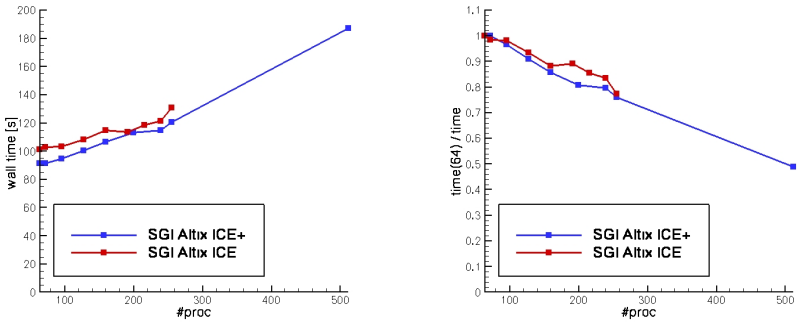
3.1.3 Performance of I/O algorithms

The performance of the parallel input/output has been tested on a SGI Altix ICE cluster with 32 8-core Intel Xeon (X5355) 2.66GHz CPUs on up to 256 cores and on a SGI Altix ICE+ cluster with 256 8-core Intel Xeon "Harpertown" (E5472) 3GHz CPUs on up to 512 cores. The benchmark was performed on the cylinder mesh described in Sec. 6.1.2, whose CFmesh file contains 3,426,300 elements and has a size of 1.79 Gb.
The total wall time for the parallel reading algorithm (including all the five phases described in 3.1.1) and the time relative to the reference case with 64 cores (the mesh could hardly fit on less CPUs) are shown in Figs. 3.5a and 3.5b for both ICE+ and ICE clusters. As expected, the reading time is consistently lower on ICE+, but the relative performance decreases more or less with the same rate on both architectures. This drop in performance is probably due to the increasing communication involved in the total ex-

```
// --- WriteListMap.hh --- //
class WriteListMap {
public:

  // constructors, destructor

  // preallocate the underlying data
  void reserve(int nbTypes, int nbSendsPerType, int
      nbLocalEntries);

  // add the min/max IDs defining the range
  void addRange(int minID, int maxID); // add the min/max IDs
      defining

  void addSendDataSize(int dataSize); // add the size of the
      data to send
  int getSendDataSize(int rangeID) const; //get the size for
      this range

  // insert the type, local and global IDs of an element
  void insertElemLocalID(int localID, int globalID, int iType);

  void endElemInsertion(int rank); // finalize the data
      insertion

  // find the list of local element IDs corresponding to this
      range
  WriteListMap::List find(int rangeID, bool isFound);

private: // data
  ...
};
```

Code Listing 3.4: Class definition for WriteListMap

change procedure to redistribute element data after the partitioning. Better results would be achieved by compiling the same code with MPT, SGI's implementation of MPI, which optimizes communication especially for collective operations. Even better results could probably be obtained by running COOLFluiD with MPT on an architecture like future SGI's Ultra Violet, a new generation of shared memory machines with terabytes of RAM and 1024 (or more) cores, where MPT can take maximum profit of the fact that the same physical memory is addressed even in a distributed computation, allowing communication intensive algorithms to run up to 10 times faster, as recently advocated by SGI's Chief Technology Officer, Dr. Eng. Lim Goh, during a HPC Seminar at ESTEC.



(a) Total wall time required by the parallel reading versus core count

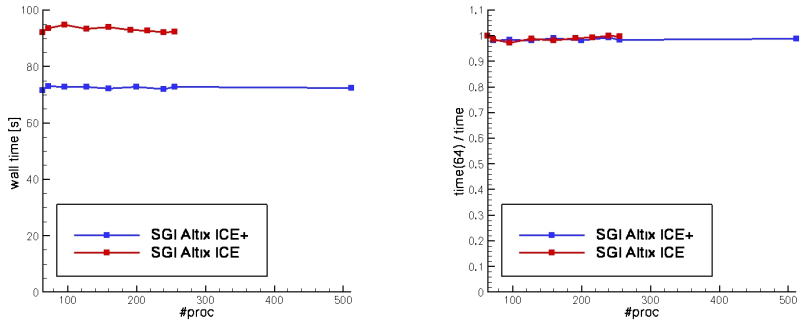
(b) Parallel reading time relative to case at 64 cores versus core count

Figure 3.5: Quantitative performance analysis of the parallel reading algorithm on SGI Altix ICE+ and ICE.

If we consider the performance of the writing algorithm, which is shown in Figs. 3.6a and 3.6b, again in terms of total and relative wall time, this is close to ideal, since the algorithm takes the same time on 64 as well as on the maximum number of cores. This means that the cost of the inter-process communication is basically negligible compared to the time spent accessing the file on the disk. Also in this case as in the previous one, the total wall time is considerably lower on the ICE+ cluster, while the behaviour is similar as far as relative time is concerned.
The results here presented represent, in fact, a *worst case scenario*, since COOLFluiD was compiled with the LAM implementation of MPI, which is known not to offer the best performance on more than a few hundreds processors. Moreover, the efficiency of the parallel reading algorithm could have been very likely enhanced by taking advantage of the available InfiniBand connections which could not be used by LAM MPI and the access to disk during pure I/O could have been speedup if a parallel filesystem like Lustre had been employed.

3.1.4 Speedup and parallel efficiency

We include in this section also some performance tests to evaluate the parallel scalability of the cell-centered FV and RD solvers implemented within



(a) Total wall time required by the parallel writing versus core count

(b) Parallel writing time relative to case at 64 cores versus core count

Figure 3.6: Quantitative performance analysis of the parallel writing algorithm on SGI Altix ICE+ and ICE.

COOLFluiD. A detailed description of those solvers will be given in Chapter 5.
The benchmarks for the FV code were again performed on the above mentioned cylinder mesh (3,426,300 hexaedral elements) with a thermo-chemical nonequilibrium model (11 equations), while the RD solver was run on the same mesh but split into 20,557,753 tetrahedra and with a standard Navier-Stokes set of equations (5 in total).
In our study, we consider two performance metrics, namely *speedup* and *parallel efficiency*. The first is defined by the following formula:

$$S_p = \frac{T_1}{T_p} \tag{3.1}$$

where p is the number of processors, T_1 is the execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm with p processors. *Linear speedup* or *ideal speedup* is obtained when $S_p = p$. We speak about *absolute speedup* when T_1 is the execution time of the best sequential algorithm, and relative speedup when T_1 is the execution time of the same parallel algorithm on one processor. In our case, we will only deal with the relative speedup and we will consider T_1 as wall time.
The parallel efficiency is defined as

$$E_p = \frac{1}{p}\frac{T_1}{T_p} \quad \text{parallel efficiency} \tag{3.2}$$

and it's equal to 1 in an ideal situation.

3.1.4.1 Finite Volume benchmark

The speedup and parallel efficiency of the cell-centered FV solver are shown in Fig. 3.7, where results with 1-layer and 2-layer overlap on the SGI Altix ICE (up to 256 processors) and 2-layer overlap on the SGI Altix ICE+ (up to 512 processors) are presented.

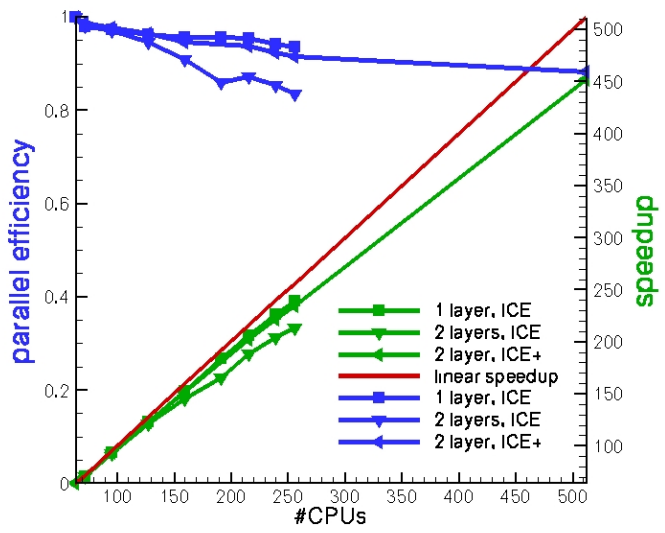


Figure 3.7: Parallel efficiency and speedup for the cylinder mesh (3,426,300 cells) with FV on SGI Altix ICE and ICE+ clusters.

Both the parallel efficiency and the speedup are estimated relatively to the run on 64 processors, since that was the minimum number of CPUs needed to make the case fit into memory. In particular, the parallel efficiency for the case with 2-layer overlap is 0.83 on 256 processors on ICE, 0.92 and

0.88 on respectively 256 and 512 processors on ICE+. This is quite an amazing result and shows the great potential of COOLFluiD for large scale computing. The merit for this result must be attributed to

1. the synchronization algorithm due to Dries Kimpe [68];

2. the author's implicit FV solver which takes care of minimizing the calls to the parallel linear system solver (PETSc in this case) and of avoiding to compute useless jacobian contributions;

3. the efficient implementation of PETSc.

3.1.4.2 Residual Distribution benchmark

Figure 3.8 shows the speedup and parallel efficiency of the RD solver with 1-layer overlap on the SGI Altix ICE cluster, where up to 256 processors were used.

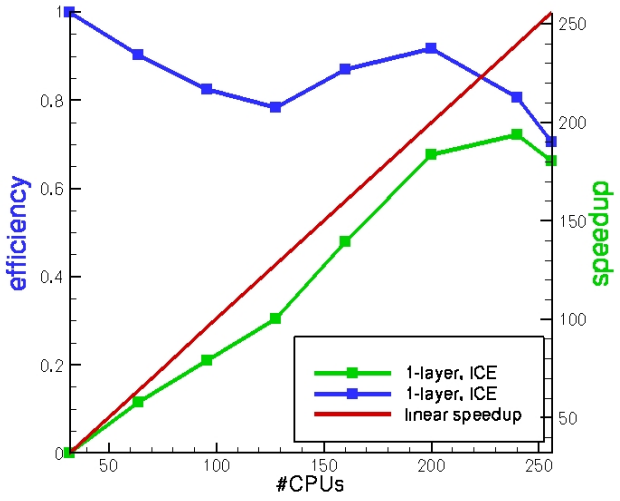


Figure 3.8: Parallel efficiency and speedup for the cylinder mesh (20,557,753 cells) with RD on SGI Altix ICE cluster.

In this case, the parallel efficiency and the speedup are estimated relatively to the run on 32 processors. The parallel efficiency on 256 processors drops to 0.7, which is still reasonably good but considerably worse than the FV counterpart, even with one layer of overlap less. Since the synchronization algorithm is the same, the hit in performance can be due to the RD implementation or to the PETSc library. Our guess is that the considerably different sparsity of the system matrix between the cell-centered FV and the vertex centered RD methods can play a role, especially on the side of PETSc, that probably has to communicate more in the RD case, because of the larger number of off-diagonal terms in each row of the matrix, corresponding to all the vertex neighbors of each degree of freedom.
In any case, despite the lower efficiency of RD in comparison with FV, the absolute value of those results confirm the high performance of COOLFluiD for computations involving millions of degrees of freedom.

Part II

Aerothermodynamics

Chapter 4

Physical Modeling

When dealing with gases at high temperatures in which heat exchanges produce a significant effect on the flow, fluid properties such as specific heat, viscosity and thermal conductivity can no longer be considered constant as in traditional aerodynamics, but they vary with temperature, pressure and chemical composition. In fact, under those conditions, phenomena like chemical dissociation, ionization or combustion can occur and an accurate description of the flowfield requires the simultaneous examination of thermal and dynamic phenomena. This leads to the expression *aerothermodynamics* when referring to the multi-physical science that combines the study of classical aerodynamics with thermodynamics, combustion and thermochemistry for applications that range from external flow over space vehicles to internal flow through vehicle propulsion systems such as scramjets.
The main problems that require aerothermodynamic considerations are combustion and high-speed flight or testing. Chemical reactions sustained by combustion flow systems produce high temperatures and variable gas composition. Because of oxidation (combustion) and in some cases dissociation and ionization processes, these systems are sometimes described as aerothermochemical. In particular, during a flight at hypersonic speed, for instance while (re-)entering a planetary atmosphere, the kinetic energy used by a vehicle to overcome drag forces is converted into compression work on the surrounding gas and thereby raises the gas temperature through a strong bow shock, as schematically shown in Fig. 4.1.

At relatively high Mach numbers, the gas temperature may raise enough to cause dissociation (Mach ≥ 7) and ionization (Mach ≥ 12). thus the gas becomes chemically active and electrically conducting. When dealing with chemically reactive gas mixtures, the Damköhler number $Da^c = \frac{\tau_f}{\tau_c}$ is an important adimensional parameter that can be used to describe the flow regime. It is defined by the ratio between τ_f, a characteristic time for the macroscopic processes occurring in the flow, and τ_c, a characteristic time for the chemistry. Three cases can be identified:

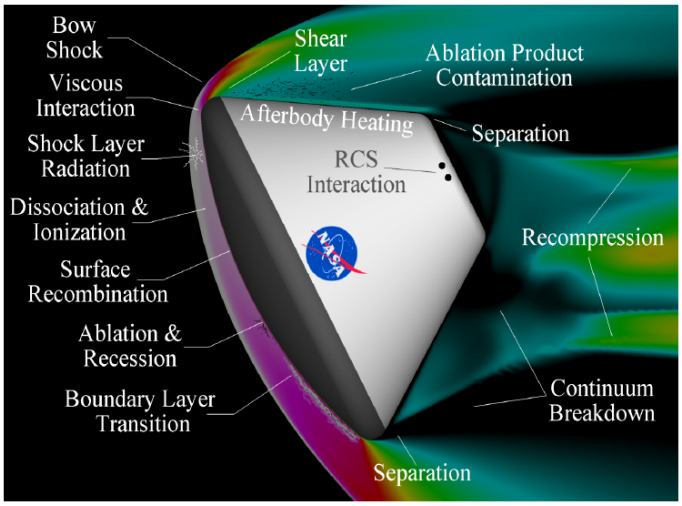


Figure 4.1: Schematic of the flowfield surrounding a capsule during the (re-)entry phase in a planetary atmosphere (courtesy of David Hash).

1. $Da \to 0$: the flow is frozen, meaning that no chemical reactions occur;
2. $Da \to \infty$: the chemical reactions are fast enough to reach equilibrium conditions under which the mixture composition can be directly computed from the local static pressure, temperature and elemental composition;
3. $Da \approx 1$: the flow is in chemical nonequilibrium and the mixture composition can be calculated from a set of continuity equations, one for each chemical component.

Analogously, another Damköhler number $Da^v = \frac{\tau_f}{\tau_v}$ can be used to define conditions of thermal equilibrium, freezing or nonequilibrium. In this case, τ_v represents the relaxation time for the equilibration of energy among the different modes, in particular the roto-translational and vibrational ones. While in thermal equilibrium one temperature is enough to describe the thermodynamics, in conditions of nonequilibrium more temperatures are needed.
The present chapter will present an up-to-date set of physical models that have been integrated in COOLFluiD, allowing us to handle all the above

mentioned thermodynamic and chemical flow regimes appropriately. To this end, the author's work has consisted in implementing the numerous systems of equations that will be described and in defining a suitable and flexible interface for getting physical quantities from dedicated libraries. In particular, all thermodynamic, transport and chemical properties used in this thesis have been implemented into the Mutation library mainly by [94] for what concerns mixtures in chemical equilibrium/nonequilibrium and by [116] with regard to thermal nonequilibrium. Mutation has been enclosed in a COOLFluiD plug-in and has been treated almost as a black-box in the present work. Many details about the algorithms and models actually used inside Mutation will therefore be left out from the following description and the proper references will be addressed whenever needed.
All the sets of governing PDE's that will be considered can be expressed in conservative and hypervectorial form as:

$$\frac{\partial \mathbf{U}}{\partial \mathbf{P}} \frac{\partial \mathbf{P}}{\partial t} + \frac{\partial \mathbf{F}_i^c}{\partial x_i} = \frac{\partial \mathbf{F}_i^d}{\partial x_i} + \mathbf{S} \tag{4.1}$$

where, according to the terminology adopted in COOLFluiD, $\mathbf{U}$ are the conservative variables, $\mathbf{P}$ the *update* variables, $\mathbf{F}^c$ and $\mathbf{F}^d$ respectively the convective and diffusive fluxes, $\mathbf{S}$ the source term.
A slightly different formulation can be employed for axisymmetric cases [26] in cylindrical coordinates:

$$\frac{\partial \mathbf{U}}{\partial \mathbf{P}} \frac{\partial r\mathbf{P}}{\partial t} + \frac{\partial r\mathbf{F}_x^c}{\partial x} + \frac{\partial r\mathbf{F}_r^c}{\partial r} = \frac{\partial r\mathbf{F}_x^d}{\partial x} + \frac{\partial r\mathbf{F}_r^d}{\partial r} + \mathbf{S} \tag{4.2}$$

where x and r are the axial and radial directions. Both the forms 4.1 and 4.2 will be used in the following description of physical models for aerothermodynamics.

4.1 Navier-Stokes

In the case of the unsteady Navier-Stokes equations, describing the conservation of mass, momentum and total energy (including both internal and kinetic contributions), the conservative variables in 4.1 are:

$$\mathbf{U} = (\rho, \rho\mathbf{u}, \rho E)^T \tag{4.3}$$

where ρ is the density, $\mathbf{u}$ are the velocity components and

$$\rho E = \frac{p}{\gamma - 1} + \rho \, \frac{\mid \mathbf{u} \mid^2}{2} \tag{4.4}$$

is the total energy per unity volume, p is the static pressure and $\gamma = c_p/c_v$ is the specific heat ratio. Several choices are instead possible for the update variables $\mathbf{P}$, for instance:

- conservative variables;
- $\mathbf{P} = (\rho, \mathbf{u}, p)^T$, primitive variables;
- $\mathbf{P} = (p, \mathbf{u}, T)^T$, primary variables;
- $\mathbf{P} = (\rho, \mathbf{u}, T)^T$, natural variables.

From a computational point of view, it is generally convenient to choose primary or natural variables as $\mathbf{P}$, because they allow numerical algorithms to readily compute the useful gradients (velocity components, temperature) appearing in the diffusive fluxes.
The convective and diffusive fluxes are defined as:

$$\mathbf{F}_i^c = \begin{pmatrix} \rho\mathbf{u} \\ \rho\mathbf{u}\mathbf{u} + p\hat{I} \\ \rho\mathbf{u}H \end{pmatrix}, \quad \mathbf{F}_i^d = \begin{pmatrix} 0 \\ \bar{\bar{\tau}} \\ (\bar{\bar{\tau}} \cdot \mathbf{u})^T - \mathbf{q} \end{pmatrix} \tag{4.5}$$

In particular, $H = E + p/\rho$ is the total enthalpy and the pressure is given by the perfect gas law:

$$p = \rho \frac{R}{M} T \tag{4.6}$$

where $R = 8314.3$ $[J/\text{kg mole-K}]$ is the universal gas constant and M is the molecular mass of the considered gas. Moreover, $\bar{\bar{\tau}}$ stands for the tensor of viscous stresses:

$$\tau_{ij} = \mu \left[\left(\frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j} \right) - \frac{2}{3} \nabla \cdot \mathbf{u} \delta_{ij} \right] \tag{4.7}$$

computed using Stokes' hypothesis of negligible bulk viscosity effects. The dynamic viscosity μ is given by Surtherland's formula:

$$\mu = C_1 \frac{T^{\frac{3}{2}}}{T + C_2} \tag{4.8}$$

with the constants C_1 and C_2 varying from one gas to another; for air, $C_1 = 1.458 \cdot 10^{-6}$ and $C_2 = 110.4$. The heat flux $\mathbf{q}$ in Eq. 4.5 is defined as

$$\mathbf{q} = -\lambda \, \nabla T \tag{4.9}$$

where $\lambda = \mu c_p / Pr$ is the gas thermal conductivity and Pr is the Prandtl number, taken constant and equal to 0.72.

4.1.1 Axisymmetric case

4.1.1.1 First formulation

In an axisymmetric case, the convective and diffusive fluxes $\mathbf{F}_i^c$ and $\mathbf{F}_i^d$ are formally the same as in 4.5, but the diagonal entries in the viscous stresses tensor in Eq. 4.7 must be corrected as follows:

$$\tau_{ii}^{\text{axi}} = \tau_{ii} - \frac{2}{3}\mu\frac{v}{r} \tag{4.10}$$

where v is the radial velocity and r is the radius (in our case equal to y). Furthermore, as demonstrated analytically in [26], the source term

$$\mathbf{S} = (0, 0, p - \tau_{\theta\theta}, 0)^T \tag{4.11}$$

must be added, yielding a viscous stress component in the circumferential direction θ:

$$\tau_{\theta\theta} = -\frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial r} - 2\frac{v}{r}\right) \tag{4.12}$$

4.1.1.2 Second formulation

An alternative axisymmetric formulation for the Navier-Stokes equations is proposed in [155]. In this case, the set of PDE's are written in a conventional multi-dimensional way as in 4.1, but a source term for both the convective and the diffusive parts of the equations appears:

$$\mathbf{S} = \mathbf{S}^c + \mathbf{S}^d = -\frac{1}{r}\begin{pmatrix} \rho v \\ \rho u v \\ \rho v^2 \\ \rho v H \end{pmatrix} + \frac{1}{r}\begin{pmatrix} 0 \\ \tau_{rx} \\ \tau_{rr} - \tau_{\theta\theta} \\ \tau_{rx} u + \tau_{rr} v - q_r \end{pmatrix} \tag{4.13}$$

where $\tau_{rx} = \mu\left(\frac{\partial u}{\partial r} + \frac{\partial v}{\partial x}\right)$ as in Eq. 4.7, τ_{rr} and $\tau_{\theta\theta}$ are given respectively by 4.10 and 4.12.
While the first axisymmetric formulation in Sec. 4.1.1.1 is especially suitable for being implemented in a FV code, this alternative form can be easily integrated in both FV and RD schemes, since the only difference with a pure 2D case lies in the presence of a source term that has to be discretized.

4.2 Local Thermodynamic Equilibrium

When considering a chemically reacting gas mixture in thermodynamic equilibrium, the concentration of each species can be determined from a corresponding species continuity equation [145]:

$$\frac{\partial \rho y_s}{\partial t} + \nabla \cdot (\rho y_s \mathbf{u}) = -\nabla \cdot J_s + \dot{\omega}_s, \quad s = [1, N_s] \tag{4.14}$$

where $y_s = \rho_s/\rho$ is the mass fraction of species s (ρ_s and ρ stand for the partial and the mixture densities), $\mathbf{u}$ the mass-averaged mixture velocity, J_s the mass diffusion flux of species s and $\dot{\omega}_s$ represents the mass production/destruction term of species s due to chemical reactions.
It is often more convenient to describe the chemical composition through the molar fractions $x_s = n_s/n$, where n_s and n represent the molar densities of individual species and of the entire mixture. We indicate with Φ_s^e the number of atoms of pure element e (e.g. O or N in an air mixture) contained in a species s. The following relations define the *elemental* mole and mass fractions:

$$X_e = \frac{\sum_{s\in S} \Phi_s^e x_s}{\sum_{e\in E}\sum_{s\in S} \Phi_s^e x_s}; \qquad Y_e = \sum_{s\in S} \Phi_s^e y_s \frac{M_e}{M_s}; \qquad M = \sum_{s\in S} x_s M_s, \tag{4.15}$$

where M_s is the molar mass of species s and M the mixture molar mass.
If chemical reactions are sufficiently fast with respect to other macroscopic processes occurring in the flow, i.e. the Damköhler number Da^c tends to ∞, the mixture composition, corresponding therefore to the chemical equilibrium composition, can be computed directly from statistical mechanics for given values of pressure, temperature and elemental mass fractions Y_e [23] instead of by solving Eqs.(4.14):

$$y_s = y_s(p, T, \mathbf{Y}_e). \tag{4.16}$$

Once the composition is determined by imposing the equilibrium of a minimum set of independent chemical reactions as described in [23, 94], all thermodynamic quantities (density, internal energy, etc.) and transport properties (dynamic viscosity, thermal conductivities) can be calculated as indicated in Sec. 4.4 and 4.5.

4.2.1 LTE with Fixed Elemental Fractions

An approximation commonly used in LTE simulations consists in assuming the elemental composition constant throughout the flow and equal to the free stream values. In the case of air, for example, this corresponds to fixing the elemental molar fractions of oxygen and nitrogen as follows:

$$X^O = 0.21\,; \qquad X^N = 0.79.$$

In this case, the convective and diffusive fluxes in Eq. 4.5 are still formally the same in the governing equations 4.1 or 4.2, but combined with the thermodynamic relations in Sec. 4.4 and with the transport properties computed by means of the algorithms described in [94].

4.2.2 LTE with Variable Elemental Fractions

As pointed out by Murphy [102] additional advection-diffusion equations, corresponding to elemental continuity equations, should be solved simultaneously to determine the locally varying elemental mass fractions:

$$\frac{\partial \rho Y_e}{\partial t} + \nabla \cdot (\rho Y_e \mathbf{u}) = -\nabla \cdot \mathbf{J}_e, \quad e = [1, N_e]. \tag{4.17}$$

Note that the source term contribution in the right hand side in the above result is zero, translating the fact that no new elements are generated in the considered chemical reactions.
The formulation given in Ref. [141] is termed explicitly "closed" because the diffusive species and enthalpy fluxes depend in an explicit manner on the solution unknowns. In particular, the elemental mass diffusion flux J_e depends on the gradients of the elemental mass fractions Y_e and of the temperature T:

$$\mathbf{J}_e = -\rho D_e^T \nabla T + \sum_{f \in \varepsilon} \rho D_{ef} \nabla Y^f \quad (e \in \varepsilon) \tag{4.18}$$

where $D_{ef}(p,T,\mathbf{Y}_e)$ and $D_e^T(p,T,\mathbf{Y}_e)$ are respectively the elemental multicomponent diffusion coefficient and the elemental thermal demixing coefficient.
Additionally, the heat flux $\mathbf{q}$ appearing in the total energy equation needs to be modified consistently in order to account for the (de)mixing phenomenon:

$$\mathbf{q} = -(\lambda + \lambda_R + \lambda_D)\nabla T - \sum_{e\in\varepsilon} \lambda_{EL}^e \nabla Y_e \tag{4.19}$$

in which three transport coefficients can be identified:

1. the thermal reactive conductivity λ_R takes into account diffusive transfer of species enthalpies in the absence of elemental diffusion [27],[30];

2. the thermal demixing conductivity λ_D adds the contribution of the additional diffusive heat transfer that occurs due to nonzero elemental diffusive fluxes when $\nabla Y_e = 0$;

3. the elemental heat transfer coefficients λ_{EL}^e take into account heat transfer due to elemental demixing driven by gradients in elemental composition.

Precise analytical expressions for $D_{ef}(p,T,\mathbf{Y}_e)$, $D_e^T(p,T,\mathbf{Y}_e)$, λ_R, λ_D and λ_{EL}^e can be found in the Appendix of [141].
This model has been applied to study and characterize incompressible flows in plasma torches [140], where the assumption of thermal and chemical equilibrium can be considered reasonable. Within this thesis work, the same model has been adopted for the first time to study 2D hypersonic re-entry flows in conditions relatively close to LTE [79], showing some potential for the heat flux prediction for this kind of applications, if compared to chemical nonequilibrium calculations.

4.2.2.1 Conservative vectorial form of the equations

The full system of the governing PDE's for chemically reacting gases under LTE-VEF can be cast under the the conservative form in Eq. 4.1, where the conservative and update variables are given by

$$\mathbf{U} = (\rho, \rho\mathbf{u}, \rho E, \rho\mathbf{Y}_e)^T, \quad \mathbf{P} = (p, \mathbf{u}, T, \mathbf{Y}_e)^T \tag{4.20}$$

and the convective and diffusive fluxes are

$$\mathbf{F}_i^c = \begin{pmatrix} \rho\mathbf{u} \\ \rho\mathbf{u}\mathbf{u} + p\hat{I} \\ \rho\mathbf{u}H \\ \rho\mathbf{u}\mathbf{Y}_e \end{pmatrix}, \ \mathbf{F}_i^d = \begin{pmatrix} 0 \\ \bar{\bar{\tau}} \\ (\bar{\bar{\tau}} \cdot \mathbf{u})^T - \mathbf{q} \\ -\mathbf{J}_e \end{pmatrix}. \tag{4.21}$$

Herein, the viscous stresses $\bar{\bar{\tau}}$ and the heat flux $\mathbf{q}$ are given by Eqs. 4.7 and 4.19.

4.3 Thermo-chemical nonequilibrium

We consider here after the equations governing the motion of a chemically reacting and possibly weakly ionized multi-component mixture of perfect gases in thermal and chemical nonequilibrium.
Under conditions of chemical nonequilibrium, the mixture composition can be determined by solving an advection-diffusion-reaction equation for each chemical species. In particular the chemical activity is driven by mass production/destruction terms that introduce considerable uncertainty in the problem, since each possible reaction rate has to be modeled and different models can yield significantly different results depending on the temperature range.
When thermal nonequilibrium is assumed, a multi-temperature model is needed in order to account for the disequilibration in the energy distribution among the different modes. Under these circumstances, a species total energy can be considered separable into different contributions:

$$e = e_t + e_r + e_v + e_e \tag{4.22}$$

i.e. translational, rotational, vibrational and electronic energies. While all modes are present for a molecule, only the translational and electronic ones can be associated to an atom or to a free electron.
As stressed by [50], the separation in Eq. 4.22 is mathematically possible according to the Born-Oppenheimer assumption only near the conditions of "potential well", i.e. the region surrounding a local minimum for the interaction potential energy. In a more realistic model, it can be proven that, in general conditions far from equilibrium (e.g. during dissociation), only translational and internal energies, without further distinction, can be analytically separated. Unfortunately, this mathematically sound novel model currently lacks of physico-chemical data and it is not yet mature

enough to be implemented into a CFD code.
State-of-the-art multi-temperature models associate distinct temperatures to different energy modes and, in a general case [51], one may think of associating one mode $e_{\delta s}$ and related temperature $T^{\delta s}$ to each mode (or even energetic level) δ admitted by each species s, leading to the following expression for the energy per unity of mass:

$$e = \sum_s \frac{x_s}{M} \sum_\delta e_{\delta s}(T^{\delta s}) \tag{4.23}$$

This approach, though theoretically possible, would be extremely complex, computationally expensive and with no guarantee that this could lead to proportionally more accurate results.
Some simplifications are therefore introduced. First of all, each mode in Eq. 4.22 is considered in equilibrium with the corresponding mode in another species, except for the vibrational one. Moreover, unlike in [65, 168], it is reasonably assumed that the translational energy of the mixture is equilibrated with the rotational energy, since this equilibration process typically takes less than 10 molecular collisions [165], leading to the following full multi-temperature model:

$$e = e_t(T^{tr}) + e_e(T^e) + e_f \qquad \text{atoms} \tag{4.24}$$
$$e = e_t(T^{tr}) + e_r(T^{tr}) + e_v(T^{v,m}) + e_e(T^e) + e_f \qquad \text{molecules} \tag{4.25}$$
$$e = e_t(T^e) \qquad \text{free electrons} \tag{4.26}$$

where e_f is equivalent to the formation enthalpy for atomic species. In this case, the system is represented by:

1. a roto-translational temperature T^{tr} for the heavy particles;
2. a different vibrational temperature $T^{v,m}$ for each of the diatomic and polyatomic species (molecules) in the gas mixture;
3. an electron-electronic temperature T^e characterizing the energy content associated to the electronic excitation of species and to the free electron translation.

A 3-temperature model is obtained by assuming that the vibration of all molecules is described by a single vibrational temperature T^v. Further simplifications can lead to Park's 2-temperature model [118], in which the distribution of vibrational, electronic excitation and electron translation energies

is represented by a single vibro-electronic temperature $T^v = T^e = T^{ve}$. Finally, a 1-temperature model corresponds to thermal equilibrium conditions.

4.3.1 3-Temperature model

We present in this section the conservation equations for a reacting gas flow under conditions of thermo-chemical nonequilibrium with the 3-temperature model [54].

4.3.1.1 Species continuity equation

The species conservation reads:

$$\frac{\partial \rho_s}{\partial t} + \nabla \cdot (\rho_s \mathbf{u}) = -\nabla \cdot (\rho_s \mathbf{u}_s^d) + \dot{\omega}_s \tag{4.27}$$

Herein, $\mathbf{u}_s^d$ are the species diffusion velocities, which can be calculated, as in our case, by solving the Stefan-Maxwell system, as described in [13], or by applying Fick's law and recasting $\rho_s \mathbf{u}_s^d$ into $-\rho \mathcal{D}_s \nabla x_s$, where x_s are the molar fractions and $\mathcal{D}_s$ are diffusion coefficients [54, 128]. In both cases, an approximation has been applied: only molecular diffusion due to concentration gradients is taken into account, while that due to pressure and temperature gradients is usually neglected.
The production/destruction term $\dot{\omega}_s$ will be described in detail in Sec. 4.6. Having assumed to have a single mass averaged velocity field $\mathbf{u} = \sum_s \frac{\rho_s}{\rho} \mathbf{u}_s$ in Eq. 4.27, only one vectorial conservation equation for the mixture momentum is needed, instead of one per species in a more general case.

4.3.1.2 Mixture momentum equations

If we apply the ambipolar diffusion constrain which links electron and ion diffusion in such a way that

$$n_e = n_i = \sum_{\text{s=ions}} n_s \tag{4.28}$$

where n_e is the electron number density and n_i is the ionic number density, the flow field in an ionized gas mixture can be considered electrically neutral, as explained in [128]. Under these circumstances, the overall mixture momentum equation retains the exact expression as in the basic Navier-Stokes

equations in Sec. 4.1 and it is therefore not recalled here.

4.3.1.3 Vibrational energy equation

The vibrational energy conservation [54] can be expressed as follows:

$$\frac{\partial \rho e^v}{\partial t} + \nabla \cdot (\rho e^v \mathbf{u}) = -\nabla \cdot \left(\sum_s \rho_s h_s^v \mathbf{u}_s^d \right) - \nabla \cdot \mathbf{q}^v + \Omega^{vt} + \Omega^{ve} + \Omega^{CV}. \quad (4.29)$$

The first term on the RHS represents the diffusion of vibrational energy due to concentration gradients, while the heat conduction due to the vibrational temperature gradients is given by

$$\mathbf{q}^v = -\lambda^v \nabla T^v \quad (4.30)$$

where λ^v is the vibrational thermal conductivity. Ω^{vt}, the energy exchange (relaxation) between vibrational and translational modes due to collisions, and Ω^{ve}, the relaxation between vibrational and electronic modes, can be expressed as

$$\Omega^{vt} = \sum_m \rho_m \frac{e_m^{v,*} - e_m^v}{\tau_m}, \quad \Omega^{ve} = \sum_m \rho_m \frac{e_m^{v,**} - e_m^v}{\tau_{em}} \quad (4.31)$$

where $e_m^{v,*}$ and $e_m^{v,**}$ are the equilibrium vibrational energies of molecules m evaluated at the roto-translational and electron temperature respectively. Those energy transfer terms have been modeled with the Landau-Teller formulation [11]. The latter assumes mono-quantum energy transfers: in the collision between two molecules or between a molecule and an electron, one colliding particle can gain or lose only one energetic level, while the energy of the other particle remains unchanged. The relaxation times τ_m and τ_{em} are given by Millikan and White [100] with Park's correction for high temperatures [120]:

$$\tau_m = \tau_m^{MW}(p,T) + \overbrace{(\sigma_m c_m n_m)^{-1}}^{\tau_{\text{Park}}} \quad (4.32)$$

where σ_m is the effective cross section for vibrational relaxation processes, c_m is the average molecular velocity of molecule m and n_m is the number density.

Finally, Ω^{CV} stands for the vibrational energy lost or gained due to molecular dissociation or recombination:

$$\Omega^{CV} = \sum_m \dot{\omega}_m \hat{D}_m \tag{4.33}$$

Several possibilities exist for the choice of $\hat{D}_m$. The simplest possibility is to impose it equal to the vibrational energy of the molecule

$$\hat{D}_m = e_m^v. \tag{4.34}$$

Alternatively, if one assumes preferential dissociation and recombination of molecules in the higher vibrational states, it can be taken equal to some fraction of the molecular dissociation energy D_m:

$$\hat{D}_m = c_1 D_m \tag{4.35}$$

with $0 < c_1 \leq 1$. In comparison with the case in Eq. 4.34, this model tends to lower the vibrational temperature behind the shock, where dissociation occurs, and increase it inside the boundary layer, where recombination occurs.

4.3.1.4 Electron and electronic excitation energy equation

The equation can be written as [54]:

$$\frac{\partial \rho e^e}{\partial t} + \nabla \cdot [(\rho e^e + p_e)\mathbf{u}] = \mathbf{u}\nabla p_e - \nabla \cdot \left(\sum_s \rho_s h_s^e \mathbf{u}_s^d \right) - \nabla \cdot \mathbf{q}^e + \Omega^{et} - \Omega^I - \Omega^{ve} - Q_{\text{rad}} \tag{4.36}$$

where p_e is the electron pressure and it is defined as

$$p_e = \rho_e \frac{R}{M_e} T^e \tag{4.37}$$

The term $\mathbf{u}\nabla p_e$ represents the work done by the electric field induced by the gradient of electron pressure.
As in [128], we can partially reformulate this and the convective term in Eq. 4.36 in order to keep the same formalism as in the vibrational-electronic equation for Park's 2-temperature in Sec. 4.3.2:

$$\frac{\partial \rho e^e}{\partial t} + \nabla \cdot (\rho e^e \mathbf{u}) = -p_e \nabla \mathbf{u} \ \cdots \quad \text{exactly as in Eq.4.36} \tag{4.38}$$

The diffusion of species electronic enthalpies due to concentration gradients is expressed in Eq. 4.36 by $-\nabla \cdot \left(\sum_s \rho_s h_s^e \mathbf{u}_s^d\right)$, while the heat conduction due to electron temperature gradients is given by

$$\mathbf{q}^e = -\lambda^e \nabla T^e \tag{4.39}$$

with λ^e being the electronic thermal conductivity.
Ω^{et} represents the energy exchange due to inelastic collisions between the electrons and the heavy particles:

$$\Omega^{et} = 2\rho_e \frac{3}{2} R(T - T^e) \sum_{s \neq e} \frac{\nu_{e,s}}{M_s} \tag{4.40}$$

where $\nu_{e,s}$ is the effective collision frequency of electron with heavy particles which is defined in [54].
Ω^I corresponds to the energy loss due to electron impact ionization and is given by

$$\Omega^I = \sum_{\text{ions}} \dot{n}_{e,s} \hat{I}_s \tag{4.41}$$

where $\dot{n}_{e,s}$ is the molar rate of production of species s and $\hat{I}_s$ is the energy lost per unit mole by a free electron in producing species s by electron impact ionization [128].
The relaxation between electronic and vibrational modes is represented by Ω^{ve}, whose expression is the same as in 4.31, but it appears in Eqs. 4.29 and 4.36 with opposite signs.
The last term in Eq. 4.36 accounts for the rate of energy loss due to radiation during electronic transitions, but it is neglected here, as in [54, 128].

4.3.1.5 Total energy equation

The conservation of the overall energy for the gas mixture is expressed by

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot (\rho H \mathbf{u}) = -\nabla \cdot \left(\sum_s \rho_s h_s \mathbf{u}_s^d \right) - \nabla \cdot \mathbf{q} + \nabla \cdot (\bar{\bar{\tau}} \cdot \mathbf{u})^T - Q_{\text{rad}} \tag{4.42}$$

The first term on the RHS is the diffusion of species enthalpy due to concentration gradients. The global heat conduction $\mathbf{q}$ groups together all the contributions due to roto-translational, vibrational and electron temperature gradients:

$$\mathbf{q} = -\left(\lambda \nabla T + \lambda^v \nabla T^v + \lambda^e \nabla T^e\right) \tag{4.43}$$

The work done by the shear forces is represented by $\nabla(\bar{\bar{\tau}} \cdot \mathbf{u})$. Finally, Q_{rad}, the rate of energy loss due to radiation, is neglected, and so is the work done on the charged particles by the electric field (here not reported). The latter is considered null, because the ambipolar diffusion constraint in Eq. 4.28 is assumed.

4.3.1.6 Conservative vectorial form of the equations

All the equations described in Secs. 4.3.1.1-4.3.1.5 can be recast into the vectorial form of Eq. 4.1, with the conservative variables

$$\mathbf{U} = (\rho_s, \rho\mathbf{u}, \rho E, \rho e^v, \rho e^e)^T \tag{4.44}$$

and the natural variables [52] as update variables

$$\mathbf{P} = (\rho_s, \mathbf{u}, T, T^v, T^e)^T. \tag{4.45}$$

The convective and diffusive fluxes are given by:

$$\mathbf{F}_i^c = \begin{pmatrix} \rho_s \mathbf{u} \\ \rho \mathbf{u}\mathbf{u} + p\hat{I} \\ \rho \mathbf{u} H \\ \rho \mathbf{u} e^v \\ \rho \mathbf{u} e^e \end{pmatrix}, \; \mathbf{F}_i^d = \begin{pmatrix} -\rho_s \mathbf{u}_s^d \\ \bar{\bar{\tau}} \\ (\bar{\bar{\tau}} \cdot \mathbf{u})^T - \mathbf{q} - \sum_s \rho_s h_s \mathbf{u}_s^d \\ -\sum_s \rho_s h_s^v \mathbf{u}_s^d - \mathbf{q}^v \\ -\sum_s \rho_s h_s^e \mathbf{u}_s^d - \mathbf{q}^e \end{pmatrix}. \tag{4.46}$$

and the vector of source terms

$$\mathbf{S} = \begin{pmatrix} \dot{\omega}_s \\ \mathbf{0} \\ \mathbf{0} \\ \Omega^{vt} + \Omega^{ve} + \Omega^{CV} \\ \Omega^{et} + \Omega^I - \Omega^{ve} \end{pmatrix} \tag{4.47}$$

Even though the 3-temperature model has been implemented in COOLFluiD, it did not yield fully satisfactory results (in particular, the electron temperature tended to inexplicably increase in the free stream before the shock) and therefore we will not present them. In fact, this model is often mentioned in literature as in [54, 128], but rarely used [65]. Moreover, there are

still open issues on the physical validity of the non-conservative electron and electronic excitation energy equation and alternatives are being sought, as recently discussed in [166].

4.3.2 2-Temperature model

The 2-temperature or Park's model [119] associates one temperature T for describing the distribution of the roto-translational energy of heavy particles and a second temperature T^{ve} for characterizing the vibrational, electronic and electron translational energies. According to [119], this approximation is justified by (1) the rapid energy transfer between the translational mode of free electrons and the vibrational mode of molecular nitrogen and (2) the rapid equilibration of the low-lying electronic states of heavy particles with the ground electronic state at the electronic temperature. This model is obtained from the 3-temperature one by merging Eqs. 4.29 and 4.36 into a single equation for the conservation of the vibrational-electronic energy:

$$\begin{aligned}\frac{\partial \rho e^{ve}}{\partial t} + \nabla \cdot (\rho e^{ve} \mathbf{u}) &= -p_e \nabla \cdot \mathbf{u} - \nabla \cdot \left(\sum_s \rho_s h_s^{ve} \mathbf{u}_s^d \right) - \nabla \cdot \mathbf{q}^{ve} + \\ &\quad \Omega^{vt} + \Omega^{et} - \Omega^I + \Omega^{CV} - Q_{\text{rad}} \end{aligned} \tag{4.48}$$

where $\mathbf{q}^{ve} = -(\lambda^v + \lambda^e)\nabla T^{ve}$ is the heat flux due to gradients of the vibrational-electronic temperature and all the other individual quantities have been defined earlier.

4.3.3 Multi-Temperature model (neutral mixtures)

In the present work, when dealing with neutral air mixture flows, we make use of a multi-temperature model which associates one temperature T to the roto-translational energy modes and 2 or more temperatures to the vibrational modes of molecular species indicated with m (O_2, N_2, NO). This model can be summarized in conservative and hypervectorial form as in Eq. 4.1, where the variables $\mathbf{U}$ and $\mathbf{P}$ are defined as:

$$\mathbf{U} = (\rho_s, \rho\mathbf{u}, \rho E, \rho_m e_m^v)^T, \quad \mathbf{P} = (\rho_s, \mathbf{u}, T, T_m^v)^T \tag{4.49}$$

and the fluxes are given by:

$$\mathbf{F}_i^c = \begin{pmatrix} \rho_s \mathbf{u} \\ \rho \mathbf{u}\mathbf{u} + p\hat{I} \\ \rho \mathbf{u} H \\ \rho_m \mathbf{u} e_m^v \end{pmatrix}, \; \mathbf{F}_i^d = \begin{pmatrix} -\rho_s \mathbf{u}_s^d \\ \bar{\bar{\tau}} \\ (\bar{\bar{\tau}} \cdot \mathbf{u})^T - \mathbf{q} - \sum_s \rho_s h_s \mathbf{u}_s^d \\ -\rho_m h_m^v \mathbf{u}_m^d - \mathbf{q}_m^v \end{pmatrix}. \quad (4.50)$$

where the heat flux corresponding to the gradient of the vibrational temperature associated to molecule m is defined as

$$\mathbf{q}_m^v = -y_m \lambda_m^v \nabla T_m^v \quad (4.51)$$

where y_m is the mass fraction of molecule m.
Finally, the source term is expressed by:

$$\mathbf{S} = \begin{pmatrix} \dot{\omega}_s \\ \mathbf{0} \\ \mathbf{0} \\ \Omega^{vt} + \Omega^{CV} + \Omega^{VV} \end{pmatrix} \quad (4.52)$$

where Ω^{VV}, accounting for the relaxation between vibrational modes of two molecules, has been neglected in this work.

4.3.4 Implementation issues

All the system of equations that have been presented have been implemented in COOLFluiD by applying the the Perspective pattern presented in Sec. 2.3.1. In order to show the extensibility of such a design solution to complex physical models, we can show its application to the concrete case of the multi-species and multi-temperature model for thermo-chemical nonequilibrium described in Sec. 4.3.

4.3.4.1 Example: Implementation of TCNEQ model

A class diagram corresponding to the application of the Perspective pattern to the TCNEQmodel, as it is implemented in COOLFluiD, is presented in Fig. 4.2.
The class corresponding to the concrete physical model for thermo-chemical nonequilibrium `NavierStokesNEQ` is defined in C.L. 4.1.
In this case, `ConvectionDiffusionReaction` is the compositor object that binds together three different equations terms: convective, diffusive and reaction (and energy transfer) term.

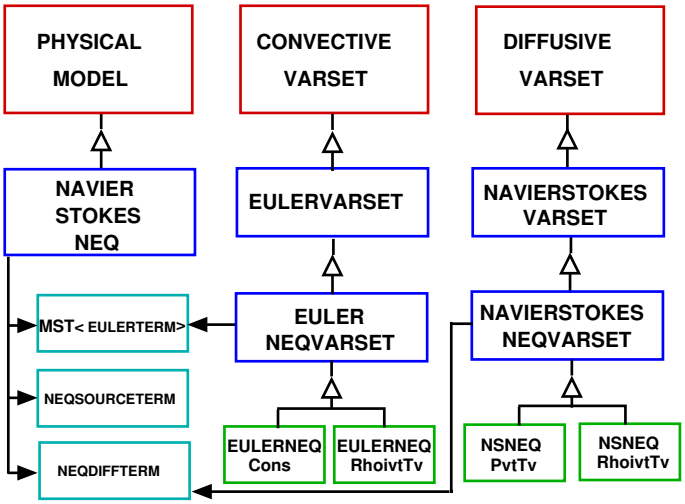


Figure 4.2: Perspective pattern applied to a TCNEQ model.

```
// --- NavierStokesNEQ.hh --- //
class NavierStokesNEQ : public ConvectionDiffusionReaction
  <MultiScalarTerm<EulerTerm>, NEQDiffTerm, NEQSourceTerm> {
public:
  // constructor, virtual destructor
  virtual int getDimension() const;
  virtual int getNbEquations() const;
  virtual int setup() const;
};
```

Code Listing 4.1: NavierStokesNEQ class definition

The convective term is obtained by parameterizing the `MultiScalarTerm`, introduced in Sec. 2.3.1.3, with the `EulerTerm` defined in C.L. 2.14. This has the effect of attaching two subsets of physical data to `EulerTerm`, one corresponding to the species mass fractions y_s, the other to the vibrational (and electron-electronic excitation) energies per unit mole e_m^v (and e_e) for which a conservation equation is considered.
The diffusive term `NEQDiffTerm` in C.L. 4.2 derives from the `NSTerm` class in C.L. 2.15, inherits access to the underlying library (e.g Mutation or Pegase) for computing additional transport properties (vibrational thermal conductivities, diffusion coefficients ...) and stores them.

```
// --- NEQDiffTerm.hh --- //
class NEQDiffTerm : public NSTerm {
public:
  // the values of the source terms are stored
  virtual int getDataSize() const
  {return NSTerm::getDataSize() + m_newProperties;}

  static string getTermName() {return "NEQDiff";}
  ...
};
```

Code Listing 4.2: NEQDiffTerm class definition

The class `NEQSourceTerm` gives access to an underlying library (e.g Mutation) for computing the mass production terms and all the source terms due to energy transfer, as shown in C.L. 4.3.

```
// --- NEQSourceTerm.hh --- //
class NEQSourceTerm : public BaseTerm {
public:
  // the values of the source terms are stored
  virtual int getDataSize() const
  {return m_sourceTerms;}

  static string getTermName() {return "NEQSource";}

  // get library to compute chemical reaction terms
  SafePtr<ChemLibrary> getChemicalLibrary() const;

  // get library to compute energy transfer terms
  SafePtr<TFLibrary> getTransferLibrary() const;
private:
  ... // member data
};
```

Code Listing 4.3: NEQSourceTerm class definition

It can be noticed that, whenever possible, full reuse of existing terms is achieved. The data set is simply extended to accommodate the new model.

TCNEQ VarSets. As explained in Sec. 2.3.1.2, convective `VarSets` are responsible for the computation of physical fluxes, jacobian (rotated) matrices, eigenvectors and eigenvalues, while keeping knowledge of the actual type of variables in function of which all those quantities are calculated. In the TCNEQ case, we normally store the solution in natural

variables **P** and we make also use of the conservative variables **U**. This corresponds to the definition of two distinct concrete convective `VarSets`, namely `EulerNEQRhoivtTv` (update variables) and `EulerNEQCons` (conservative variables), as shown in Fig 4.2. They basically reimplement part of the interface for a `ConvectiveVarSet` (i.e. the polymorphic type which is exposed to numerical algorithms) shown in C.L. 2.17, but reusing as much code as possible from `EulerVarSet` (see C.L. 2.19). For example, C.L. 4.4 shows that `EulerNEQRhoivtTv` derives from a `MultiScalarVarSet` (see C.L. 2.22) parameterized with `EulerVarSet`, which takes care of computing the fluxes and eigenvalues corresponding to the basic Euler equations plus the additional entries due to the advection of partial densities and vibrational (and electron-electronic) energies.

```
// --- EulerNEQRhoivtTv.hh --- //

class EulerNEQRhoivtTv : public MultiScalarVarSet<EulerVarSet> {
public:
  // constructor, virtual destructor,
  // overridden parent virtual methods
  ...
};
```

Code Listing 4.4: EulerNEQRhoivtTv interface

Since the implementation of `MultiScalarVarSet` is generic, the same procedure is reused for many other cases where additional equations are plugged to the basic Euler core (LTE-VEF, turbulence ...). All thermodynamic quantities (pressure, enthalpy, speed of sound ...) are computed by means of the acquainted thermodynamic library (Mutation, in our case) and temporarily stored in the corresponding convective term, `MultiScalarTerm<EulerTerm>`. Similarly, `NavierStokesNEQVarSet` inherits from `NavierStokesVarSet`, provides the implementation of the diffusive fluxes, delegates part to the implementation to variable-dependent subclasses like `NavierStokesNEQRhoivtTv` and queries the transport library (e.g. Mutation) in `NEQDiffTerm` for the computation of transport properties.
For sake of consistency, a source term VarSet could also be defined, but, in our case, we shortcut the access to `NEQSourceTerm` via a Strategy object (`ComputeSourceTerm`) that takes also care of discretizing the source term and that is defined in the module corresponding to the chosen numerical algorithm. This is just a simplification that shows once again the flexibility and tunability of our design solution.

4.4 Thermodynamics

4.4.0.2 Equation of state

According to Dalton's law of partial pressures, under the assumption that each chemical component behaves as a perfect gas and therefore obeys Eq. 4.6, the equation of state for a gas mixture under conditions of thermal nonequilibrium becomes

$$p = p_e + \sum_{s \neq e} p_s = \rho R \left(T^e \frac{y_e}{M_e} + T \sum_{s \neq e} \frac{y_s}{M_s} \right) \tag{4.53}$$

where p_e represents the electron pressure and $T^e = T$ if thermal equilibrium is assumed. In the latter case, if also chemical equilibrium holds (LTE), once that the mixture composition is known, either in terms of mass fractions $y_s = \rho_s/\rho$ or molar fractions $x_s = y_s M/M_s$, density is calculated from Eq. 4.53 and from

$$\rho = \sum_s \rho_s \tag{4.54}$$

If the case of nonequilibrium, Eq. 4.53 is used to calculate pressure from the local values of partial densities ρ_s and temperatures T and T^e.

4.4.0.3 Mixture energy

The mixture internal energy per unit mass is given by

$$e = \sum_s y_s e_s = \frac{1}{M} \sum_s x_s M_s e_s \tag{4.55}$$

where different contributions are considered for each species energy, according to the separation scheme introduced in Eq. 4.22:

$$e_s = e_s^0 + (c_{v,s}^t + c_{v,s}^r)\, T + e_s^v(T_s^v) + e_s^{\mathrm{el}}(T^e) \tag{4.56}$$

where each term is provided by quantum mechanics. In particular, e_s^0 can be calculated from

$$h_s^0 = e_s^0 + \frac{R}{M_s} T_{\mathrm{ref}} \tag{4.57}$$

in which h_s^0 is the *heat of formation* and T_{ref} is chosen equal to 0.

Translational and rotational energy. The specific heat ratio at constant volume for the translational and rotational modes are computed as:

$$c_{v,s}^t = \frac{3}{2}\frac{R}{M_s}, \quad c_{v,s}^r = \frac{d}{2}\frac{R}{M_s} \tag{4.58}$$

where

- $d = 0$ for monoatomic molecules;
- $d = 2$ for diatomic and inline polyatomic molecules;
- $d = 3$ for non-inline polyatomic molecules.

Vibrational energy. The expression of the vibrational energy e_s^v is derived by assuming that the internal quantum states are populated according to a Boltzmann distribution, and that the molecule s behaves as a harmonic oscillator. It reads:

$$e_s^v = \frac{R}{M_s}\frac{\theta_s^v}{e^{\theta_s^v/T_s^v} - 1} \tag{4.59}$$

where T_s^v is the vibrational temperature of molecule s and θ_s^v a characteristic temperature for vibration.

Electronic energy. The species electronic energy e_s^{el} per unit mass in Eq. 4.55 is computed as in [13]:

$$e_s^{\text{el}} = \frac{R}{M_s}\frac{\sum_{e=0}^{\infty} g_{s,e}^{\text{el}}\theta_{s,e}^{\text{el}}\exp(-\theta_{s,e}^{\text{el}}/T^e)}{Q_s^{\text{el}}} \tag{4.60}$$

where Q_s^{el} is the electronic partition function which is defined as

$$Q_s^{\text{el}} = \sum_{e=0}^{\infty} g_{s,e}^{\text{el}}\exp(-\theta_{s,e}^{\text{el}}/T^e) \tag{4.61}$$

Herein, the series in 4.60 and 4.61 are mathematically divergent and must be conveniently truncated at a energy level $e_{\max,s}$, different for each species. The characteristic electronic temperatures $\theta_{s,e}^{\text{el}}$ and the degeneracies $g_{s,e}^{\text{el}}$

associated to each energy level e and species s can be found in [116].
The electronic energy per unit volume appearing in Eq. 4.36 includes all the electronic excitation energies of heavy particles and the free electron translational energy:

$$\rho e^e = \rho_e c_{v,e} T^e + \sum_{s \neq e}^{N_s} \rho_s e_s^{\mathrm{el}} \tag{4.62}$$

4.4.0.4 Mixture enthalpy

As far as the mixture enthalpy is concerned, it can be expressed as:

$$h = \sum_s y_s h_s \tag{4.63}$$

in which the species enthalpies are given by

$$h_s = h_s^0 + (c_{p,s}^t + c_{p,s}^r)\ T + h_s^v(T_s^v) + h_s^{\mathrm{el}}(T^e) \tag{4.64}$$

where the following relations hold:

$$c_{p,s}^t = c_{v,s}^t + \frac{R}{M_s}, \quad c_{p,s}^r = c_{v,s}^r \tag{4.65}$$

$$h_s^v = e_s^v, \quad h_s^{\mathrm{el}} = \begin{cases} e_s^{\mathrm{el}} & \text{for s} \neq \text{e} \\ RT^e/M_e & \text{for s} = \text{e} \end{cases} \tag{4.66}$$

4.5 Transport Properties

The dynamic viscosity μ and the translational thermal conductivity λ^t are computed in the Mutation library by applying the efficient iterative algorithms described in [94, 95] to the rigorous expressions derived from kinetic theory [60] for single temperature gas mixtures. This approach is supposed to be more accurate than the one followed by most of the numerical solvers for chemically reacting flows [7, 54, 105, 128], all resorting to mixture rules such as Yos' [173], Lee's [81] or Blottner's (1971).
The rotational, vibrational and electronic thermal conductivities are modeled by means of the Eucken approximation [56] which requires the corresponding c_p's, i.e. specific heat coefficients at constant pressure.
The electron thermal conductivity and electrical conductivity are computed

with the formulas due to Devoto [151], where two non-vanishing Sonine polynomial contributions were found to yield accurate results [59].
The diffusion fluxes $\rho_s \mathbf{u}_s^d$ have been computed solving the Stefan-Maxwell system [7, 13, 14] of equations which consist of a linear system (in the diffusion fluxes) of as many equations as the chemical species are present in the mixture. This system is supplemented by the auxiliary condition that the sum of the diffusion fluxes is zero plus the ambipolar constraint introduced in Eq. 4.28 [159].

4.6 Chemical kinetic model

We consider a general set of *elementary* chemical reactions involving N_s species:

$$\sum_{s=1}^{N_s} \alpha_{s,r} \Psi_s \overset{\rightarrow}{\leftarrow} \sum_{s=1}^{N_s} \beta_{s,r} \Psi_s \tag{4.67}$$

where Ψ_s indicate the symbols of the chemical components and $\alpha_{s,r}$ and $\beta_{s,r}$ the stoichiometric coefficients for reactants and products respectively.
The time rate of production of species per unit volume, $\dot{\omega}_s$ appearing in the species continuity equations is derived following the kinetic processes occurring in the system. Many kinetic models are available and include the identification of set of reactions like the ones in Eq. 4.67 and the specification of appropriate constants for the evaluation of the corresponding reaction rates. $\dot{\omega}_s$ can be expressed as [11]:

$$\dot{\omega}_s = M_s \sum_{r=1}^{N_r} (\beta_{s,r} - \alpha_{s,r}) \Xi_r (R_{f,r} - R_{b,r}) \tag{4.68}$$

where N_r is the number of reactions belonging to the chosen chemical kinetic model, $R_{f,r}$ and $R_{b,r}$ are the forward and backward reaction rates, which are given by

$$R_{f,r} = k_{f,r} \prod_{s=1}^{N_s} \left(\frac{\rho_s}{M_s} \right)^{\alpha_{s,r}}, \quad R_{b,r} = k_{b,r} \prod_{s=1}^{N_s} \left(\frac{\rho_s}{M_s} \right)^{\beta_{s,r}} \tag{4.69}$$

where $k_{f,r}$ and $k_{f,r}$ are the forward and backward reaction rate coefficients. Moreover, $\Xi_r = \sum_{s=1}^{N_s} \xi_{s,r} \frac{\rho_s}{M_s}$ represents the third body contribution to the dissociation or recombination reactions, with efficiency $\xi_{s,r}$ for reaction r.

Any individual species can be excluded or included as third body in a reaction by setting the corresponding $\xi_{s,r}$ to 0 or 1 respectively.

4.6.1 Reaction rates

The forward reactions rates $k_{f,r}$ introduced in Eq. 4.69 are given by Arrhenius' law:

$$k_{f,r}(T_1) = A_{f,r} T_1^{n_{f,r}} \exp(-E_{f,r}/kT_1) \tag{4.70}$$

Herein, $A_{f,r}$,$n_{f,r}$ and $E_{f,r}$ are determined experimentally for each reaction and are provided by the chemical kinetic models. $E_{f,r}$ is the activation energy, i.e. an energy threshold that must be crossed to activate the corresponding reaction.
The backward reactions rates $k_{b,r}$ are computed from the relation

$$k_{b,r}(T_2) = k_{f,r}(T_2)/K_{c,r}^{eq}(T_2) \tag{4.71}$$

where $K_{c,r}^{eq}$ is the *equilibrium reaction rate constant* which can be expressed as:

$$K_{c,r}^{eq} = e^{-\Delta G_r^0/RT}(RT)^{\Delta n} \tag{4.72}$$

in which $\Delta G_r^0 = \sum_{\text{products}} G_s^0 - \sum_{\text{reactants}} G_s^0$ is the difference between Gibbs' free energy between the products and reactants and Δn_r is the difference in the number of moles of products and reactants.
In a multi-temperature context, T_1 and T_2 are the forward and backward reaction rate controlling temperatures. In Park's model [119, 121], for instance, the rate controlling temperature is empirically defined as

$$T_i = \sqrt{T_a T_b}, \quad i = 1, 2 \tag{4.73}$$

where a and b can be chosen among T, T^v or T^e, with different combinations according to the reaction type, as described in detail in [128].
More advanced CVDV models which couple vibration and dissociation more consistently, such as Treanor-Marrone's [97] or Knab's [110], require a more complex definition of the reaction rate coefficients in function of the involved temperatures.

4.6.2 Air chemistry model

Most of our simulations in TCNEQ conditions are applied to air mixture flows. At ambient temperature, air can be assumed to be made of 79% in volume of N_2 and 21% of O_2, even though some minor components like argon (Ar), carbon dioxide (CO_2) and neon (N_e) are present. As temperature increases, chemical reactions occur and other species appear. Molecular oxygen is the first to dissociate in a temperature range 2000-4000 K at a pressure of 1 atm. By reaction between N_2 and atomic oxygen, nitric oxide is produced above 2000 K and then the main phase of dissociation takes place in the range 3500-8000 K. At around 4000 K, nitric oxide ion NO^+ begins to form and for temperatures higher than 6000 K O^+ and N^+ appear. In any case, the lower the pressure, the lower the temperature at which dissociation activates, with the result of having, for example, fully dissociated oxygen at 3000 K at 100 Pa and at 5000 K at 10000 Pa. Depending on the pressure and temperature range, we can roughly distinguish between a regime in which the degree of ionization is negligible and air is well represented by a 5-species mixture (N,O,N_2,NO,O_2) and another one in which air can be considered as a 11-species ionized mixture (e^-,N,O,N_2,NO,O_2,N^+,O^+,N_2^+,NO^+,O_2^+).
In order to take into account all the chemical activity occurring in air in different regimes, several sets of reaction are considered, as shown in Fig. 4.3:

1. thermal dissociation of molecules by collision with all [120] or some [56] of the heavy particles;
2. bimolecular exchange reactions, which are the main responsibles for the production of NO and remove N_2 from the mixture more efficiently than dissociation;
3. associative ionization or dissociative recombination;
4. charge exchange reactions, which, whenever impact ionization is neglected, are the only responsible for the creation of atomic ions;
5. heavy particle impact ionization, which are present only in some models [56] and have generally a negligible effect, due to their very high activation energy;
6. electron impact ionization of atomic N and O, which also have a very high activation energy, but once triggered, they cause an exponential increase of the free electrons number density.

Thermal dissociation
$O_2 + X \rightleftarrows O + O + X$ $N_2 + X \rightleftarrows N + N + X$ $NO + X \rightleftarrows N + O + X$
Bimolecular exchange
$O_2 + N \rightleftarrows NO + O$ $N_2 + O \rightleftarrows NO + N$
Associative ionization-dissociative recombination
$N + O \rightleftarrows NO^+ + e^-$ $N + N \rightleftarrows N_2^+ + e^-$ $O + O \rightleftarrows O_2^+ + e^-$
Charge exchange
$NO^+ + O \rightleftarrows N^+ + O_2$ $O_2^+ + N \rightleftarrows N^+ + O_2$ $O^+ + NO \rightleftarrows N^+ + O_2$ $N^+ + N_2 \rightleftarrows N_2^+ + N$ $O_2^+ + N_2 \rightleftarrows N_2^+ + O_2$ $O^+ + N_2 \rightleftarrows N_2^+ + O$ $NO^+ + N \rightleftarrows N_2^+ + O$ $O_2^+ + O \rightleftarrows O^+ + O_2$ $NO^+ + N \rightleftarrows O^+ + N_2$ $NO^+ + O_2 \rightleftarrows O_2^+ + NO$ $NO^+ + O \rightleftarrows O_2^+ + N$
Heavy particle impact ionization
$O_2 + N_2 \rightleftarrows NO + NO^+ + e^-$ $NO + X \rightleftarrows NO^+ + e^- + X$
Electron impact ionization
$O + e^- \rightleftarrows O^+ + e^- + e^-$ $N + e^- \rightleftarrows N^+ + e^- + e^-$

Figure 4.3: Chemical reactions model for 11-species air (from [13]).

Chapter 5

Numerical methods

5.1 Implicit time discretization

The hyperbolic system in Eq. 4.1 must be discretized in time and space in order to provide a numerical solution. As far as time discretization is concerned, unconditionally stable fully implicit schemes are most suitable for the viscous and generally stiff applications which are treated in this thesis, because they allow the simulation to converge to steady state with relatively high time steps and in many less iterations, if compared to what explicit or semi-implicit schemes can yield. The good performance in terms of convergence rates of this kind of algorithms is partially counter-balanced by the large memory requirements needed to store the system matrix. However, this problem is being increasingly alleviated by the concurrent growing power and storage capabilities of the computer architectures and, mainly, by the significant performance gain offered by distributed computing.
Keeping this in mind, we chose to devote our effort towards the implementation in COOLFluiD of a flexible Newton method which is able to tackle multiple weakly coupled systems of equations, as described here after. The algorithm represents a generalization of the one proposed in [7], but it is formulated to be independent from the choice of the actual implicit time stepping scheme, or space discretization, or variables in which the update is performed. Moreover, full profit of the available computational resources can be taken, since the overall algorithm has also been parallelized.

5.1.1 Newton method for weakly coupled systems

When integrating in space the system of PDE's in Eq. 4.1, we obtain the following system of conservation laws:

$$\frac{d}{dt}\int_{\Omega} \mathbf{U}\, d\Omega + \int_{\Omega} \nabla \cdot \mathbf{F}^c\, d\Omega = \int_{\Omega} \nabla \cdot \mathbf{F}^d\, d\Omega + \int_{\Omega} \mathbf{S}\, d\Omega \qquad (5.1)$$

After having decomposed the system of equations 5.1 into a number $N_s \geq 1$ of weakly coupled equation subsystems, we apply the selected space discretization to it. We rewrite the system of equations and we include all the spatially discretized terms in the residuals $\mathbf{R_s}(\mathbf{U})$, defining the pseudo-steady residual $\tilde{\mathbf{R}}_\mathbf{s}(\mathbf{U})$, as follows:

$$\tilde{\mathbf{R}}_s(\mathbf{P}) = \frac{d}{dt}\int_\Omega \mathbf{U}_s \; d\Omega + \mathbf{R}_s(\mathbf{P}) = 0, \quad s = 1 \dots N_s \tag{5.2}$$

where $\mathbf{U}$ are the conservative variables and $\mathbf{P}$ are the variables in which the solution is stored and updated, e.g. primitive, natural, conservative etc. $\mathbf{U}_s = \mathbf{U}_s(\mathbf{P})$ represents the subset of n_s conservative variables whose update is associated to the solution of the subsystem s:

$$\mathbf{U}_s = [\mathbf{U}_{s0}, \;\; \mathbf{U}_{s1}, \;\; \dots, \;\; \mathbf{U}_{sn_s-1}] \tag{5.3}$$

The steady residual $\mathbf{R}_s(\mathbf{P})$ includes the convective fluxes $\mathbf{F}_s^c$, the diffusive fluxes $\mathbf{F}_s^d$, and the source terms $\mathbf{S}_s$ for subsystem s:

$$\mathbf{R}_s(\mathbf{P}) = \oint_\Sigma (\mathbf{F}_s^c - \mathbf{F}_s^d)(\mathbf{P}) \cdot \mathbf{n} \; d\Sigma - \int_\Omega \mathbf{S}_s(\mathbf{P}) \; d\Omega \tag{5.4}$$

The actual time discretized expression for the *pseudo-steady* residual $\tilde{\mathbf{R}}_s(\mathbf{P})$ depends on the choice of the scheme for the time integration. As an example, under the assumption of constant volume Ω, i.e. of non-moving mesh, we have:

$$\tilde{\mathbf{R}}(\mathbf{P}) = \frac{\mathbf{U}(\mathbf{P}) - \mathbf{U}(\mathbf{P}^n)}{\Delta t}\Omega + \mathbf{R}(\mathbf{P}) \qquad \text{Backward Euler (BE)} \tag{5.5}$$

$$\tilde{\mathbf{R}}(\mathbf{P}) = \frac{\mathbf{U}(\mathbf{P}) - \mathbf{U}(\mathbf{P}^n)}{\Delta t}\Omega + \frac{1}{2}[\mathbf{R}(\mathbf{P}) + \mathbf{R}(\mathbf{P}^n)] \quad \text{Crank-Nicolson (CN)} \tag{5.6}$$

$$\tilde{\mathbf{R}}(\mathbf{P}) = \frac{3\mathbf{U}(\mathbf{P}) - 4\mathbf{U}(\mathbf{P}^n) + \mathbf{U}(\mathbf{P}^{n-1})}{2\Delta t}\Omega + \mathbf{R}(\mathbf{P}) \quad \text{3-Point Backward (3B)} \tag{5.7}$$

where BE is a first order time accurate scheme, while both CN and 3B yield second order accuracy. After having performed a Taylor expansion in time around $\tilde{\mathbf{R}}_s(\mathbf{P}^n)$ we get:

$$\tilde{\mathbf{R}}_s\left(\mathbf{P}^{n+1}\right)=0 \;\Rightarrow\; \tilde{\mathbf{R}}_s\left(\mathbf{P}^{n+1}\right)=\tilde{\mathbf{R}}_s\left(\mathbf{P}^{n}\right)+\frac{\partial\tilde{\mathbf{R}}_s}{\partial\mathbf{P}_s}\left(\mathbf{P}^{n}\right)\Delta\mathbf{P}_s^n=0 \tag{5.8}$$

where we have neglected the influence of the cross-derivatives between residuals and update variables corresponding to different subsystems:

$$\frac{\partial\tilde{\mathbf{R}}_s}{\partial\mathbf{P}_{w\neq s}}=0 \implies \sum_{w\neq s}\frac{\partial\tilde{\mathbf{R}}_s}{\partial\mathbf{P}_w}(\mathbf{P}^n)\Delta\mathbf{P}_w^n=0 \tag{5.9}$$

Following a standard Newton procedure, the following N_s linear systems, one per each equation subsystem, must be solved, starting with $\mathbf{P}^0=\mathbf{P}^n$:

$$\begin{cases} \left[\frac{\partial\tilde{\mathbf{R}}_s}{\partial\mathbf{P}_s}\left(\mathbf{P}^k\right)\right]\Delta\mathbf{P}_s^k &= -\tilde{\mathbf{R}}_s(\mathbf{P}^k) \\ \mathbf{P}_s^{k+1} &= \mathbf{P}_s^k+\Delta\mathbf{P}_s^k \end{cases} \tag{5.10}$$

In a steady case, where time accuracy is not required, the solution at the new time step $\mathbf{P}_s^{n+1}$ can be directly set equal to $\mathbf{P}_s^{k+1}$ after one Newton subiteration. In an unsteady case, we must keep on iterating till $\parallel\Delta\mathbf{P}_s^k\parallel<\varepsilon$ and then set

$$\mathbf{P}_s^{n+1}=\mathbf{P}_s^{k^{\text{last}}+1}. \tag{5.11}$$

before advancing to the next time level.
Herein, two iterative strategies have been implemented:

1. all linear systems ($s=[1,N_s]$) are first assembled and then solved within the same Newton loop; at the end of the whole iterative procedure, a single solution update is applied to all the variables in $\mathbf{P}$;

2. one linear system at a time is assembled and solved, so that N_s distinct Newton loops (each one with potentially different number of Newton iterations) and corresponding solution updates are performed.

In our applications involving weak coupling, such as turbulent and Inductively Coupled Plasma (ICP) simulations in Chapter 7, only the first scheme has been applied.
Even though the present thesis from now on will focus solely on steady applications, for which the linear Backward Euler scheme (with one Newton step) in Eq. 5.5 has been employed, the algorithm here described has also

been successfully applied to unsteady problems in [172] and especially in [169], in combination with the second order time accurate 3B scheme in Eq. 5.7.
Moreover, the scheme here presented offers flexibility in the choice of the update variables while retaining a conservative form. This is extremely useful in case of flows characterized by complex thermodynamic relations, like the ones presented in this thesis, where one can conveniently choose $\mathbf{P}$ such that the functions $\mathbf{U}_s(\mathbf{P})$ are explicit.

5.1.2 Jacobian computation

A crucial ingredient for the overall performance of an implicit solver is the calculation of the flux jacobian terms to be included in the matrix of the linear system to solve:

$$\left[\frac{\partial \tilde{\mathbf{R}}_s}{\partial \mathbf{P}_s}\left(\mathbf{P}^k\right)\right] = J_t + J_R \tag{5.12}$$

Two contributions to the jacobian matrix can be identified, namely J_t corresponding to the time-dependent part of the equation and J_R related to the discretization of the spatial part (convective and diffusive fluxes, source term). The expression of J_t and J_R depend on the choice of the time stepping method:

$$J_t^{BE} = \frac{\Omega}{\Delta t}\frac{\partial \mathbf{U}_s}{\partial \mathbf{P}_s}(\mathbf{P}^k), \qquad J_R^{BE} = \frac{\partial \mathbf{R}_s}{\partial \mathbf{P}_s}(\mathbf{P}^k) \tag{5.13}$$

$$J_t^{CN} = \frac{\Omega}{\Delta t}\frac{\partial \mathbf{U}_s}{\partial \mathbf{P}_s}(\mathbf{P}^k), \qquad J_R^{CN} = \frac{1}{2}\frac{\partial \mathbf{R}_s}{\partial \mathbf{P}_s}(\mathbf{P}^k) \tag{5.14}$$

$$J_t^{3B} = \frac{3}{2}\frac{\Omega}{\Delta t}\frac{\partial \mathbf{U}_s}{\partial \mathbf{P}_s}(\mathbf{P}^k), \qquad J_R^{3B} = \frac{\partial \mathbf{R}_s}{\partial \mathbf{P}_s}(\mathbf{P}^k) \tag{5.15}$$

The jacobian of the space terms J_R typically incorporates different contributions, depending on the chosen space discretization method, as explained more in detail in Sec. 5.2.6.1 and Sec. 5.3.5.1.

5.1.3 Linear system solver

Each linear system arising from a Newton linearization in Eq. 5.10 is solved in COOLFluiD by a dedicated third party library such as PETSc [73], Trilinos [72] or SAMG [4] which have been encapsulated behind a common

interface, as shown in Sec.2.4.1.2 and are treated as black-boxes by the CFD solver.
A non negligeable effort was needed in order to make these libraries work in parallel inside COOLFluiD in a transparent way for the numerical solvers. This work mainly consisted in defining some extra LSS Command objects that encapsulate a few calls to the MPI based routines of the original libraries. Moreover, the storage of some extra connectivity/indexing information was needed in order to correctly insert the jacobian contributions to the system matrices (one per equation subsystem in weakly coupled cases) only coming from the local updatable States in each processor, without duplicating the entries inside the overlap region. In the case of different weakly coupled systems, our flexible implementation allows the user to associate a different linear system solver package to each one of the systems, according to his/her own needs.
Even though other options were available, all the results presented in this thesis have made use of the Generalized Minimal RESidual (GMRES) method [143] in combination with the restricted additive Schwarz preconditioner provided by PETSc. GMRES is an iterative algorithm that approximates the solution of the given linear system by the vector in a Krylov subspace with minimal residual.

5.2 Cell-Centered Finite Volume

During the last three decades, the Finite Volume method [16, 58, 82, 86, 152] has consolidated itself as the *de facto* standard technique for simulating a variety of flows in regimes ranging from incompressible to hypersonic. The success of the method is mainly due to the capability to adapt to every kind of meshes (structured, unstructured, polyhedral, cartesian, non-conformal) and to the good shock capturing properties, which make it suitable for handling compressible flows exhibiting complex shock interactions and discontinuities in general. Moreover, the algorithm is relatively easy to implement and efficiently parallelizable.
The basic idea of the technique consists in subdividing the computational domain in finite cells or *volumes* and, in a first order approximation, in assuming the solution to be constant inside each cell and stored in the corresponding centroids, as indicated in Fig. 5.1.

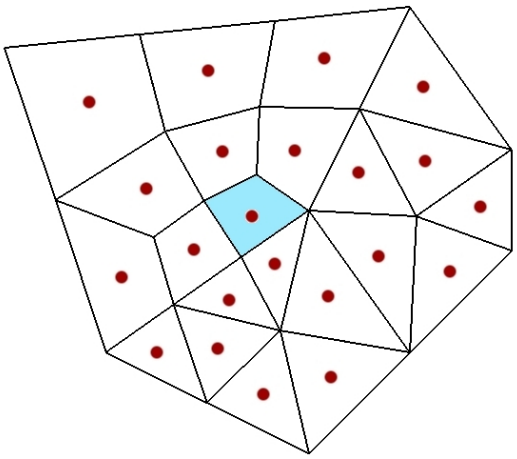


Figure 5.1: Example of a computational domain on which a cell centered Finite Volumes discretization is applied.

The algorithm discretizes the system of PDE's in Eq. 4.1 written in integral form as in Eq. 5.1:

$$\frac{d}{dt}\int_{\Omega} \mathbf{U}\, d\Omega + \oint_{\Sigma} \mathbf{F}^c \cdot \mathbf{n}\, d\Sigma = \oint_{\Sigma} \mathbf{F}^d \cdot \mathbf{n}\, d\Sigma + \int_{\Omega} \mathbf{S}\, d\Omega \tag{5.16}$$

where Ω is the volume of each single cell in the computational domain and the Gauss' theorem has been applied to convert the volume integrals of the divergence of the fluxes into contour integrals of the fluxes on the boundary of each volume. At this point, every single term in the equation can be discretize separately, typically leading to an upwind or central treatment of the convective fluxes, a central discretization of the diffusive fluxes, a cell-centered treatment of the source term in combination with an explicit (forward Euler, Runge Kutta) or implicit scheme (like the one presented in Sec. 5.1) for the (pseudo-)time stepping. Moreover, second order accuracy in space is obtained by evaluating the fluxes with a linear polynomial reconstruction of the solution, while high order in time can be easily achieved by selecting a high order scheme for the time integration like a n-th order Runge Kutta order (explicit) or multi-point backward formulas (implicit).

5.2.1 Discretization of Convective Fluxes

If we discretize the convective term, we obtain:

$$\oint_{\Sigma} \mathbf{F}^c \cdot \mathbf{n} \, d\Sigma = \sum_{f=1}^{N_f} \mathbf{F}_f \Sigma_f \tag{5.17}$$

where $\mathbf{F}_f = \mathbf{F}_f^c \cdot \mathbf{n}_f$ is the numerical convective flux projected onto the normal $\mathbf{n}_f$ to the interface f with area Σ_f. In a first order cell centered Finite Volume method, the numerical flux $\mathbf{F}$ depends on the state vectors corresponding to the left and right neighboring cell centers of the considered face:

$$\mathbf{F} = \mathbf{F}(\mathbf{U}^L, \mathbf{U}^R) \tag{5.18}$$

This is due to the fact that a piecewise constant representation of the solution holds within each control volume in which the domain is subdivided, meaning that neighboring states are in general not equal. This leads to the definition of a nonlinear Riemann problem on the interfaces, as pointed out by Godunov [55], which can be solved with the methods described in [58, 152]. Suitable numerical schemes must be able to resolve monotonically the discontinuities (shocks, contact surfaces) appearing in compressible flows, without yielding overshoots or undershoots in the solution. Moreover, schemes should not violate the entropy condition, i.e. the numerical counterpart of the second principle of thermodynamics, which states that no expansion shocks are admissible as weak solutions of the conservation law

in Eq. 5.16. The effort towards the construction of satisfactory schemes is still ongoing, but many different successful formulations already exist. In particular, three main classes of schemes can be identified: Flux Difference Splitting (FDS), Flux Vector Splitting (FVS) and hybrid schemes. FDS [111, 114, 142] are known to be the most accurate especially for resolving viscous flows, because of their ability to correctly resolve contact discontinuities, while this corresponds to the major deficiency of FVS [83, 148], the advantage of the latter lying in their superior robustness in capturing strong shocks. Hybrid schemes such as HUS [13, 31] or AUSM [87–90] try to combine the best properties of FDS and FVS, while providing, especially in the case of AUSM, a superior computational efficiency and robustness.
In this section, we describe only a few of the schemes we implemented in order to deal with high-enthalpy and chemically reactive flows and for which some results will be presented in Chapter 6.

5.2.1.1 Roe

Roe's flux splitting scheme [142] is based on an exact solution of the linearized Riemann problem across an interface and provides the most accurate results among the finite volume based discretization techniques for convective fluxes. In particular it yields a precise resolution for contact discontinuities and shear layers, which makes it highly suitable for dealing with viscous flows at high Reynolds number, since boundary layers are recognized as contact discontinuities by the Riemann solver. In its general formulation it reads:

$$\mathbf{F}^{Roe} = \frac{1}{2}\left[\mathbf{F}_n(\mathbf{P}^L) + \mathbf{F}_n(\mathbf{P}^R)\right] - \frac{1}{2} \mid \mathbf{A}(\bar{\mathbf{Z}}) \mid \left[\mathbf{U}(\mathbf{P}^R) - \mathbf{U}(\mathbf{P}^L)\right] \tag{5.19}$$

with

$$\mathbf{A} = \frac{\partial \mathbf{F}_n}{\partial \mathbf{U}}, \quad \mid \mathbf{A} \mid = \mathbf{R} \mid \mathbf{\Lambda} \mid \mathbf{R^{-1}}, \quad \mid \mathbf{\Lambda} \mid_{ij} = \mid \lambda \mid_i \delta_{ij} \tag{5.20}$$

where $\mathbf{R}$ is the matrix of right eigenvectors, $\mathbf{\Lambda}$ the diagonal matrix of eigenvalues,$\mathbf{U}$ are the conservative variables and $\mathbf{P}$ are the variables in which the solution is updated (primitive or conservative in our case). A key ingredient of the method is the linearization of the flux jacobian matrix $\mid \mathbf{A} \mid$, projected onto the direction of the face normal $\mathbf{n}$. An exact linearization is possible for relatively simple cases (e.g. Euler equations for perfect non-reactive gas) through the definition of a unique averaged parameter vector

$\bar{\mathbf{Z}}$. However, for the system of equations describing a mixture of perfect gases in thermo-chemical non equilibrium, which is what our work focuses on, such a linearization is not directly available. One of the properties that the matrix $\bar{\mathbf{A}}$ has to satisfy is:

$$\Delta(\mathbf{F}_n) = \bar{\mathbf{A}}\Delta(\mathbf{U}) \tag{5.21}$$

where $\Delta(.) = (.)_R - (.)_L$. Eq.5.21 is rigorously valid for all the flux components which are homogeneous quadratic functions in the components of Z, except for the pressure flux. In particular, in the case of a neutral gas mixture with N_s chemical species, the momentum components of Eq. 5.21 are satisfied if the following condition holds:

$$\Delta(p) = \sum_{s=1}^{N_s} \bar{\alpha_s}\Delta(\rho_s) + \bar{\beta}\Delta\rho(e_{tr}) \tag{5.22}$$

where

$$\beta = \frac{\partial p}{\partial \rho E}, \quad \alpha_s = RT/M_s - \beta e_{tr,s}, \quad e_{tr,s} = (f_s RT + \Delta h_{f,s})/M_s \tag{5.23}$$

with $f_s = 2.5$ for molecules and $f_s = 1.5$ for atoms. Different methods have been proposed in order to satisfy Eq.5.22, like the approximate and relatively complex solution in [91], the recipe based on computational experience in [53], the choice of a simple average of left and right states in conservative variables as in [144].
For our implementation, we have chosen the simple and accurate approach presented in [127], which requires that $\bar{\alpha_s}$ and $\bar{\beta}$ form a consistent set of thermodynamic variables together with the other Roe-average variables. If we define $a = \sqrt{\rho_L}/(\sqrt{\rho_L} + \sqrt{\rho_R})$ and $b = (1 - a)$, the following relations hold:

$$\bar{y}_s = a(y_s)_L + b(y_R)_R, \qquad s = 1, ..., N_s \tag{5.24}$$

$$\bar{\mathbf{u}} = a\mathbf{u}_L + b\mathbf{u}_R \tag{5.25}$$

$$\bar{H} = aH_L + bH_R \tag{5.26}$$

$$\bar{e}_V = a(e_V)_L + b(e_V)_R \tag{5.27}$$

Moreover, if we choose the following definitions for $\bar{\beta}$, $\bar{\alpha_s}$,$\bar{e}_{tr,s}$ and $\bar{a}^2$:

$$\bar{\beta} = \left[\sum_{s=1}^{N_s} \bar{y}_s/M_s\right] / \left[\sum_{s=1}^{N_s} f_s\bar{y}_s/M_s\right] \tag{5.28}$$

$$\bar{\alpha_s} = R\bar{T}/M_s - \beta\bar{e_{tr,s}} \tag{5.29}$$

$$\bar{e}_{tr,s} = (f_s R\bar{T} + \Delta h_{f,s})/M_s \tag{5.30}$$

$$\bar{a}^2 = \sum_{s=1}^{N_s} \bar{\alpha_s}\bar{y}_s + \bar{\beta}(\bar{H} - \bar{q} - \bar{e}_V) \tag{5.31}$$

[127] shows that we get some remarkably simple results like

$$\bar{T} = aT_L + bT_R \tag{5.32}$$

$$\bar{e}_{tr} = a(e_{tr})_L + b(e_{tr})_R \tag{5.33}$$

$$\overline{p/\rho} = a(p/\rho)_L + b(p/\rho)_R \tag{5.34}$$

and this allow us to perform a consistent linearization for the rotated jacobian matrix $\mid \mathbf{A} \mid$.

5.2.1.2 Carbuncle fix

All Riemann solvers, including Roe's, are affected by the so called *carbuncle phenomenon*. The latter consists in a numerical instability which develops from the stagnation stream line in flows characterized by strong bow shocks (typically at Mach higher that 6) and moves outward into the flow field, causing a severe deterioration of the solution. Several fixes have been developed so far to cure the carbuncle specifically for the Roe scheme. Quirk [133] proposes a hybrid scheme, where a pressure based shock detector allows to identify regions in vicinity of shocks, where Roe's scheme is replaced by a more dissipative one. Sanders *et al.* [144] identify the multidimensional nature of the phenomenon and present a parameter-free upwind dissipation modification, best known as *H-correction*, whose stencil is shown in Fig. 5.2. They define the maximum variation of the eigenvalues across a cell interface f with $\mathbf{n}_f$ as unit normal:

$$\eta_f = \frac{1}{2}\max_{l\in\mathcal{H}}(\mid \lambda_l(\mathbf{U}^R, \mathbf{n}_f) - \lambda_l(\mathbf{U}^L, \mathbf{n}_f) \mid) \tag{5.35}$$

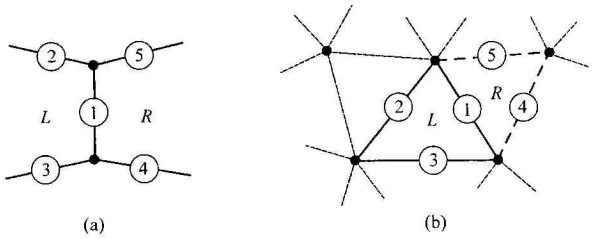


Figure 5.2: Cell interfaces defining the stencil for the H-correction for (a) structured uniform mesh and (b) unstructured triangular mesh (from [123]).

and then they determine a multidimensional correction by selecting the maximum η_f within a H-shaped stencil, centered on the current face:

$$\eta^{SA} = \max(\eta_1, \eta_2, \eta_3, \eta_4, \eta_5) \tag{5.36}$$

Finally, η^{SA} is injected into one of the following entropy corrections in order to calculate $\mid \tilde{\lambda}_l \mid$:

$$\mid \tilde{\lambda}_l \mid = \mid \lambda_l(\bar{\mathbf{Z}}) \mid + \eta \tag{5.37}$$

$$\mid \tilde{\lambda}_l \mid = \max(\mid \lambda_l(\bar{\mathbf{Z}}) \mid, \eta) \tag{5.38}$$

$$\mid \tilde{\lambda}_l \mid = \begin{cases} \mid \lambda_l(\bar{\mathbf{Z}}) \mid & \text{if } \mid \lambda_l(\bar{\mathbf{Z}}) \mid \geq 2\eta \\ \mid \lambda_l(\bar{\mathbf{Z}}) \mid^2 / 4\eta + \eta & \text{otherwise} \end{cases} \tag{5.39}$$

A variant of this method is proposed by Pandolfi *et al.* in [115], where it is proposed to define

$$\eta^{PA} = \max(\eta_2, \eta_3, \eta_4, \eta_5) \tag{5.40}$$

and to apply the fix only to the entropy and shear waves, in order to avoid to introduce unnecessary artificial viscosity. Another fix is designed in [123] especially tailored for truly unstructured meshes. It consists in applying the modified multidimensional dissipation by Pandolfi to the entropy and shear waves (es) and the Van Leer entropy correction [84] to the acoustic ones (ac):

$$\mid \tilde{\lambda}_l \mid = \begin{cases} \mid \lambda_{ac}(\bar{\mathbf{Z}}) \mid & \text{if } \mid \lambda_a c(\bar{\mathbf{Z}}) \mid \geq 2\eta^{VL} \\ \mid \lambda_{ac}(\bar{\mathbf{Z}}) \mid^2 / 4\eta^{VL} + \eta^{VL} & \text{if } \mid \lambda_a c(\bar{\mathbf{Z}}) \mid < 2\eta^{VL} \\ \max(\mid \lambda_{es}(\bar{\mathbf{Z}}) \mid, \eta^{PA}) & \end{cases} \tag{5.41}$$

where $\eta^{VL} = \max(\lambda_R - \lambda_L, 0)$.

5.2.1.3 Analytical Flux Jacobian

The Roe scheme is computationally more expensive than most of the other flux schemes, especially because of its matrix dissipation term. However, when used in combination with implicit time stepping, its flux jacobian can be easily computed analytically and this results in a quite efficient algorithm. If we condider frozen the matrix $\mathbf{A}$ during the derivation as in [42], the jacobian terms with respect to the left and right state are respectively:

$$\frac{\partial \mathbf{F}^{Roe}}{\partial \mathbf{P}_L} = \frac{1}{2}(\mathbf{A}(\mathbf{P}_L) + \mid \mathbf{A}(\bar{\mathbf{Z}}) \mid)\frac{\partial U}{\partial P}(\mathbf{P}_L) \tag{5.42}$$

$$\frac{\partial \mathbf{F}^{Roe}}{\partial \mathbf{P}_R} = \frac{1}{2}(\mathbf{A}(\mathbf{P}_R) - \mid \mathbf{A}(\bar{\mathbf{Z}}) \mid)\frac{\partial U}{\partial P}(\mathbf{P}_R) \tag{5.43}$$

where the linearized rotated matrix dissipation is kept constant during the derivation.

5.2.1.4 Modified Steger-Warming

The original Steger-Warming scheme [148] belongs to the flux vector splitting family and relies on the fact that the inviscid flux vector $\mathbf{F}$ is homogeneous in the vector of conservative variables $\mathbf{U}$. This implies that

$$\mathbf{F} = \frac{\partial \mathbf{F}}{\partial \mathbf{U}}\mathbf{U} = \mathbf{A}\mathbf{U} \tag{5.44}$$

If we diagonalize the Jacobian $\mathbf{A}$ such that

$$\mathbf{A} = \mathbf{R}\mathbf{\Lambda}\mathbf{R}^{-1} \tag{5.45}$$

where $\mathbf{R}$ is the matrix of right eigenvectors, the fluxes can be split into forward and backward moving components, according to the sign of the eigenvalues:

$$\mathbf{F} = \mathbf{F}_+ + \mathbf{F}_- \tag{5.46}$$

with

$$\mathbf{F}_\pm = \left(\mathbf{R}\Lambda_\pm\mathbf{R}^{-1}\right)\mathbf{U} = \mathbf{A}_\pm\mathbf{U} \tag{5.47}$$

$$\Lambda_+ = \frac{\Lambda + \mid \Lambda \mid}{2}, \quad \Lambda_- = \frac{\Lambda - \mid \Lambda \mid}{2}, \tag{5.48}$$

where $\Lambda_\pm$ are the diagonal matrices of the positive and negative eigenvalues. A straightforward discretization of the convective fluxes across a finite volume face with normal $\mathbf{n}$ leads to

$$\mathbf{F}^{SW}_{+,L} \cdot \mathbf{n} = \mathbf{A}_+(\mathbf{U}_L, \mathbf{n})\mathbf{U}_L, \quad \mathbf{F}^{SW}_{+,R} \cdot \mathbf{n} = \mathbf{A}_-(\mathbf{U}_R, \mathbf{n})\mathbf{U}_R \tag{5.49}$$

It is well known that this formulation offers a good robustness in capturing strong shocks and rarefaction waves but it lacks accuracy in resolving boundary and shear layers. The performance of the scheme can be considerably enhanced [42, 105] by evaluating the split Jacobian matrices in states $\tilde{\mathbf{P}}_\pm$, corresponding to the following weighted averages of the primitive variables $\mathbf{P}$:

$$\tilde{\mathbf{P}}_+ = (1-\omega)\mathbf{P}_L + \omega\mathbf{P}_R, \quad \tilde{\mathbf{P}}_- = \omega\mathbf{P}_L + (1-\omega)\mathbf{P}_R \tag{5.50}$$

with pressure-related weight ω defined as

$$\omega = \frac{1}{2}[1/(\mid \nabla\tilde{p} \mid^2 + 1)], \quad \nabla\tilde{p} = \sigma_2(p_R - p_L)/\min(p_R, p_L) \tag{5.51}$$

As suggested in [42], we set $\sigma_2 = 0.5$. The resulting blended scheme reverts to the original Steger-Warming formulation in the vicinity of strong shocks, while it smoothly switches to a more accurate form, having $\mathbf{A}_+$ and $\mathbf{A}_-$ both evaluated in the same averaged data, in regions of small pressure gradients, as in the case of contact discontinuities, shear layers and boundary layers.

5.2.1.5 AUSM Family

The schemes proposed in [88–90] combine the simplicity and efficiency of the flux vector splitting method with the high level of accuracy typical of Godunov type schemes. They rely on the splitting of the flux $\mathbf{F}$ into a convective $\mathbf{F}^{(c)}$ and a pressure component $\mathbf{F}^{(p)}$. Therefore, at a continuum level we have:

$$\mathbf{F}(\mathbf{U}) = \mathbf{F}^{(c)} + \mathbf{F}^{(p)} = \dot{m}\mathbf{\Psi} + \mathbf{F}^{(p)} \tag{5.52}$$

The numerical discretization of the component of flux $\mathbf{F}$ normal to a given interface between left (L) and right (R) states can consequently be defined as:

$$\mathbf{F}_{1/2}(\mathbf{U}_L, \mathbf{U}_R, \mathbf{n}) = \dot{m}_{1/2}\boldsymbol{\Psi}_{L/R} + \mathbf{p}_{1/2} \tag{5.53}$$

where different choices are possible for the quantities $\dot{m}_{1/2} = \dot{m}_{1/2}(\mathbf{U}_L, \mathbf{U}_R, \mathbf{n})$ and $\mathbf{p}_{1/2} = \mathbf{p}_{1/2}(\mathbf{U}_L, \mathbf{U}_R, \mathbf{n})$ which determine the splitting of convective and pressure components. Herein, in the case of a gas mixture in thermal and chemical nonequilibrium with N_s chemical components and N_m molecules, i.e. one of the most comprehensive cases presented in this thesis, the scalar mass flux $\dot{m}$, the vector quantity $\boldsymbol{\Psi}$ and the pressure flux $\mathbf{F}^{(p)}$ can be expressed as:

$$\dot{m} = \rho q_n, \quad \boldsymbol{\Psi} = \begin{pmatrix} y_s \\ \mathbf{u} \\ H \\ y_m E_m^v \end{pmatrix}, \quad \mathbf{F}^{(p)} = \begin{pmatrix} 0 \\ p\mathbf{n} \\ 0 \\ 0 \end{pmatrix} \tag{5.54}$$

where $q_n = \mathbf{u}\cdot\mathbf{n}$ is the velocity projected onto the normal $\mathbf{n}$ to the considered cell interface.

Liou-Steffen's AUSM. In the original AUSM formulation in [90], the following definitions apply:

$$\dot{m}_{1/2} = M_{1/2} \begin{cases} a_L\rho_L & \text{if } M_{1/2} > 1, \\ a_R\rho_R & \text{otherwise} \end{cases} \tag{5.55}$$

$$M_{1/2} = \mathcal{M}^+(M_L) + \mathcal{M}^-(M_R) \tag{5.56}$$

$$\mathbf{p}_{1/2} = \mathcal{P}^+(M_L)p_L\mathbf{n} + \mathcal{P}^-(M_R)p_R\mathbf{n} \tag{5.57}$$

where the split Mach number polynomials $\mathcal{M}^\pm$ read

$$\mathcal{M}^\pm(M) = \begin{cases} \mathcal{M}^\pm_{(1)}(M), & \text{if } \mid M \mid > 1, \\ \mathcal{M}^\pm_{(2)}(M) & \text{otherwise} \end{cases} \tag{5.58}$$

with

$$\mathcal{M}^\pm_{(1)}(M) = \frac{1}{2}(M \pm \mid M \mid) \tag{5.59}$$

$$\mathcal{M}^\pm_{(2)}(M) = \pm\frac{1}{4}(M \pm 1)^2 \tag{5.60}$$

and the split pressure polynomials $\mathcal{P}^\pm$ are given by

$$\mathcal{P}^{\pm}(M) = \begin{cases} \frac{1}{M}\mathcal{M}^{\pm}_{(1)}, & \text{if } \mid M \mid > 1, \\ \pm\mathcal{M}^{\pm}_{(2)}(2 \mp M), & \text{otherwise} \end{cases} \tag{5.61}$$

with the normal Mach number $M_i = \frac{q_{ni}}{a_i}$ and the normal speed $q_{ni} = \mathbf{u}_i \cdot \mathbf{n}$. Then simple upwinding is applied to define $\mathbf{\Psi}_{1/2}$:

$$\mathbf{\Psi}_{1/2} = \begin{cases} \mathbf{\Psi}_L & \text{if } M_{1/2} \geq 0, \\ \mathbf{\Psi}_R & \text{otherwise} \end{cases} \tag{5.62}$$

AUSM$^+$. An improved version of the previous scheme is presented in [89]. The split Mach numbers are redefined as follows:

$$\mathcal{M}^{\pm}_{(4)}(M) = \begin{cases} \mathcal{M}^{\pm}_{(1)}, & \text{if } \mid M \mid \geq 1, \\ \mathcal{M}^{\pm}_{(2)}(1 \mp 16\ \beta\mathcal{M}^{\pm}_{(2)}) & \text{otherwise} \end{cases} \tag{5.63}$$

where $-\frac{1}{16} \leq \beta \leq \frac{1}{2}$.
The split pressure polynomials $\mathcal{P}^{\pm}$ are also modified:

$$\mathcal{P}^{\pm}_{(5)}(M) = \begin{cases} \frac{1}{M}\mathcal{M}^{\pm}_{(1)}, & \text{if } \mid M \mid \geq 1, \\ \mathcal{M}^{\pm}_{(2)}[(\pm 2 - M) \mp 16\ \alpha M \mathcal{M}^{\mp}_{(2)}], & \text{otherwise} \end{cases} \tag{5.64}$$

with $\frac{3}{16} \leq \alpha \leq \frac{1}{8}$. The mass flux at the interface $\dot{m}_{1/2}$ is based on an average sound speed $a_{1/2}$ between the left and right states:

$$\dot{m}_{1/2} = M_{1/2} a_{1/2} \begin{cases} \rho_L & \text{if } M_{1/2} > 1, \\ \rho_R & \text{otherwise} \end{cases} \tag{5.65}$$

Several choices are possible for expressing $a_{1/2} = a(\mathbf{U_L}, \mathbf{U}_R)$:

$$a_{1/2} = \sqrt{a_L\ a_R} \tag{5.66}$$

$$a_{1/2} = \frac{a_L + a_R}{2} \tag{5.67}$$

$$a_{1/2} = \min(\tilde{a_L}, \tilde{a_R}), \quad \tilde{a} = \frac{a^{*2}}{\max(a^*, \mid q_n \mid)} \tag{5.68}$$

[88, 89] recommend the last expression, based on the critical sound speed

$$a^* = \sqrt{\frac{2(\gamma - 1)}{\gamma + 1} H} \tag{5.69}$$

which yields the exact solution for a single stationary shock discontinuity. In our computations dealing with flows in thermo-chemical non-equilibrium, we adopt a more suitable definition for the critical sound speed, as inspired by [146]:

$$a^* = \sqrt{\frac{2\tilde{\gamma}(\bar{\gamma}-1)}{2\bar{\gamma}+\tilde{\gamma}(\bar{\gamma}-1)}H} \tag{5.70}$$

where the frozen specific heat ratio $\tilde{\gamma} = \frac{\sum_s y_s \partial h_s/\partial T}{\sum_s y_s \partial e_s/\partial T}$ and the equivalent specific heat ratio $\bar{\gamma} = 1 + p/\rho e$ are considered.

AUSM$^+$-up. A further extension of the AUSM$^+$ flux to accurately simulate flows from low speed to hypersonic regime is discussed in [87, 88]. According to the new formulation, a pressure diffusion term is integrated in Eq. 5.56:

$$M_{1/2} = M_{1/2}^{AUSM^+} + K_p \max(1-\sigma\bar{M}^2, 0)\frac{p_R - p_L}{\rho_{1/2} a_{1/2}^2} \tag{5.71}$$

where $0 \leq K_p \leq 1$, $\sigma \leq 1$ and the averaged Mach number and interface density are given by

$$\rho_{1/2} = \frac{\rho_L + \rho_R}{2}, \quad \bar{M} = \frac{q_{n_L}^2 + q_{n_R}^2}{2a_{1/2}^2}. \tag{5.72}$$

The pressure flux 5.57 is also modified as follows:

$$\mathbf{p}_{1/2} = \mathbf{p}_{1/2}^{AUSM^+} - K_u \mathcal{P}_{(5)}^+ \mathcal{P}_{(5)}^- (\rho_L + \rho_R)(f_a a_{1/2})(q_{n\,R} - q_{n\,L}) \tag{5.73}$$

with $0 \leq K_u \leq 1$ and the scaling factor f_a given by

$$f_a(M_0) = M_0(2 - M_0), \quad M_0^2 = \min(1, \max(\bar{M}^2, M_{co})) \tag{5.74}$$

where the cut-off Mach number M_{co} is user defined and $\mathbf{O}(M_\infty)$. Moreover, the parameters α and β appearing in the split Mach and pressure functions are set to

$$\alpha = \frac{3}{16}(-4 + 5f_a^2) \in \left[-\frac{3}{4}, \frac{3}{16}\right], \quad \beta = \frac{1}{8} \tag{5.75}$$

As suggested in [87, 88], we set as default values $K_p = 0.25$, $K_u = 0.75$ and $\sigma = 1.0$.

5.2.2 High Order Reconstruction

The basic finite volume method assumes a constant average solution vector on each cell and this leads to a first order accurate discretization in space. As observed by Van Leer [82], higher accuracy can be reached by replacing the piecewise constant left and right states with a piecewise polynomial representation. In particular, in order to get second order accuracy, each one of the cell centered state variables u_i must be linearly extrapolated to the face quadrature points q as follows:

$$\tilde{u}(\mathbf{x}_q) = u_i + \nabla u_i \cdot (\mathbf{x}_q - \mathbf{x}_i) \tag{5.76}$$

where $\mathbf{x}_i$ denotes the centroid position of the control volume Ω_i. The linearly reconstructed state variables $\tilde{u}$ can be calculated with different methods, some of which are described here after. Moreover, when dealing with the simulation of compressible flows, a flux limiter must be employed on the reconstructed states, in order to prevent the appearance of oscillations near discontinuities.
Finally, the convective fluxes presented in Sec. 5.2.1 must be evaluated in function of the reconstructed (and limited, if needed) states $\tilde{u}$.

5.2.2.1 Weighted MUSCL

The procedure of extrapolating the cell centered variables in order to generate high order upwind schemes is often indicated as MUSCL (Monotone Upstream-centered Schemes for Conservation Laws). The original method [82] is based on a one-dimensional reconstruction on both side of an interface, with a stencil involving a number of aligned cell centers and it is applicable only to regular structured meshes, where the required stencil can be easily built.
In this work, we have employed this reconstruction on structured meshes (with only quadrilaterals in 2D or only hexahedra in 3D) but introducing some weights in order to account for the irregularity of the mesh, due to the different size of neighboring cells. When applying a linear extrapolation of the variables $\mathbf{U}$ on the left and on the right of an interface, the resulting formulas are:

$$\tilde{\mathbf{U}}_L = \mathbf{U}_L + \omega_L(\mathbf{U}_L - \mathbf{U}_{LL}) \tag{5.77}$$

$$\tilde{\mathbf{U}}_R = \mathbf{U}_R - \omega_R(\mathbf{U}_{RR} - \mathbf{U}_R) \tag{5.78}$$

where the weights ω_i given by:

$$\omega_L = \frac{\mid \bar{\mathbf{x}} - \mathbf{x}_L \mid}{\mid \mathbf{x}_L - \mathbf{x}_{LL} \mid} \tag{5.79}$$

$$\omega_R = \frac{\mid \bar{\mathbf{x}} - \mathbf{x}_R \mid}{\mid \mathbf{x}_{RR} - \mathbf{x}_R \mid} \tag{5.80}$$

where $\bar{\mathbf{x}}$ is the position of the interface mid point and $\mathbf{x}_i$ are the cell centroid positions. Moreover, on regular meshes we have $\omega_i = \frac{1}{2}$, as in the original MUSCL extrapolation.
A special treatment is applied on the boundary faces. In traditional cell centered structured finite volume codes, two layers of ghost cells are employed. In our case, we only adopt one layer, which means that no $\mathbf{U}_{RR}$ is available for the boundaries. In that case, assuming that no discontinuity lies on the boundary itself, we apply a simple unlimited weighted average variable extrapolation:

$$\tilde{\mathbf{U}}_L = \tilde{\mathbf{U}}_R = \frac{\omega_L \mathbf{U}_L + \omega_R \mathbf{U}_R}{\omega_L + \omega_R} \tag{5.81}$$

with $\omega_i = \frac{1}{|\bar{\mathbf{x}} - \mathbf{x}_i|}$. If the ghost state is located in a symmetric position with respect to the inner cell center and the boundary, those weights are equal to $\frac{1}{2}$. However, in some cases (e.g. isothermal wall condition) we allow the ghost node to move and get closer to the boundary to keep reasonable values for some extrapolated variables (e.g. temperature > 0) and this gives different values for the weights.

5.2.2.2 Least Squares Technique

On a general polyhedral unstructured mesh, the cellwise gradient ∇u can be computed with a least square (LS) approach as the result of the following linear system [15, 16]:

$$[\mathbf{L}_x \;\; \mathbf{L}_y \;\; \mathbf{L}_z] \;\; \nabla u_i = \mathbf{f}_u \tag{5.82}$$

The matrix on the LHS is generally non-square and its column vectors $\mathbf{L}_d$ are defined as:

$$\mathbf{L}_d = [w_1(\Delta x_d)_1, \;\; ..., \;\; w_{N_i}(\Delta x_d)_{N_i}]^T \tag{5.83}$$

where the weights w_k multiply the distances between the centroid of the current cell and the centroids of its N_i neighbor cells, belonging to the chosen computational stencil. Linear weights can be based on the inverse of

distances and computed as $w_j = 1/\| \mathbf{\Delta x}_j \|$.
The non linear RHS vector $\mathbf{f}_u$ reads:

$$\mathbf{f}_u = [w_1 \Delta(u_1 - u_i), \ \ ..., \ \ w_N(u_{N_i} - u_i)]^T \tag{5.84}$$

The system in Eq. 5.82 can be solved in a least squares sense with an orthogonalization technique, leading to

$$[\mathbf{L}_x \ \ \mathbf{L}_y \ \ \mathbf{L}_z]^T \cdot [\mathbf{L}_x \ \ \mathbf{L}_y \ \ \mathbf{L}_z] \ \ (\nabla u)_{\Omega_i} = [\mathbf{L}_x \ \ \mathbf{L}_y \ \ \mathbf{L}_z]^T \cdot \mathbf{f}_u \tag{5.85}$$

After having defined the dot products $l_{jk} = \mathbf{L}_j \cdot \mathbf{L}_k$ and $f_j = \mathbf{L}_j \cdot \mathbf{f}_u$, Eq. 5.85 simplifies to

$$\nabla u_i = \{l_{jk}\}^{-1} \mathbf{f}_l \tag{5.86}$$

where

$$\{l_{jk}\} = [\mathbf{L}_x \ \ \mathbf{L}_y \ \ \mathbf{L}_z]^T \cdot [\mathbf{L}_x \ \ \mathbf{L}_y \ \ \mathbf{L}_z], \quad \mathbf{f}_l = [fx, \ \ fy, \ \ fz]^T \tag{5.87}$$

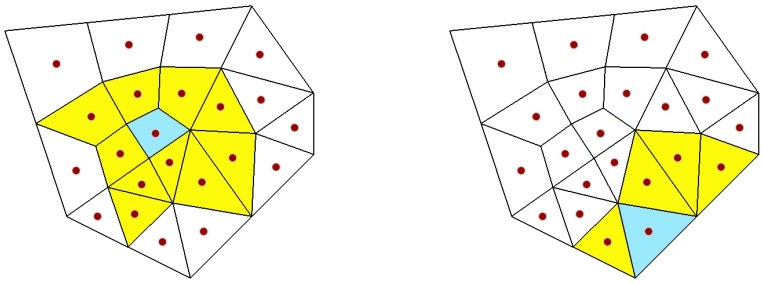
If we take into account the definition of the inverse for a 3x3 matrix, Eq. 5.86 can be develop further and gives

$$\nabla u_i = \frac{1}{\mathtt{det}(\{l_{jk}\})} \begin{pmatrix} \mathtt{det}(\mathbf{M}^L_{xx}) f_x + \mathtt{det}(\mathbf{M}^L_{xy}) f_y + \mathtt{det}(\mathbf{M}^L_{xz}) f_z \\ \mathtt{det}(\mathbf{M}^L_{xy}) f_x + \mathtt{det}(\mathbf{M}^L_{yy}) f_y + \mathtt{det}(\mathbf{M}^L_{yz}) f_z \\ \mathtt{det}(\mathbf{M}^L_{xz}) f_x + \mathtt{det}(\mathbf{M}^L_{yz}) f_y + \mathtt{det}(\mathbf{M}^L_{zz}) f_z \end{pmatrix} \tag{5.88}$$

where $\mathbf{M}^L_{jk}$ is a minor of the matrix $\{l_{jk}\}$. The system 5.85 is not necessarily well posed and a sufficiently large stencil is needed to prevents singularities ($\mathtt{det}(\{l_{jk}\}) \simeq 0$).

Reconstruction Stencil. [37] and [85] show and analyze the importance that the chosen computational stencil for the least square extrapolation has on the accuracy and robustness of the solution. That's the reason why, in our implementation, we offer great flexibility for the choice of the stencil, which is user-defined and assembled in a pre-processing step, independently from the core reconstruction algorithm.
The following possibilities are available:

1. `Face`: includes only the face neighbors for each cell;
2. `Face-Vertex`: includes all the distant-1 cell neighbors, i.e. all cells sharing at least one vertex with the current cell (ghost cells can be included or not), as shown in Fig. 5.3a;



(a) Face-Vertex stencil (b) Face-BVertex stencil

Figure 5.3: Two stencils used by the least square reconstruction algorithm.

3. `Face-BVertex`: includes only the face neighbors for internal faces and the face-vertex neighbors for the boundary faces, as illustrated in Fig. 5.3b;

4. `Face-Edge`: includes all cell neighbors sharing at least one edge with the current cell.

Options (1) and (3), the cheapest ones from a computational point of view, do not provide enough robustness and accuracy, especially in 2D. Options (2) and (4) yield the best results,with the latter to be preferred in 3D computations, because of the prohibitive large stencil associated to the first one, especially for meshes with tetrahedra.

5.2.3 Flux Limiters

In order to prevent the appearance of spurious oscillations near shock waves or contact surfaces, i.e. the so-called Gibbs phenomenon, when using a high order solution reconstruction, a flux limiter, here after indicated with Φ, is typically applied. Therefore, when dealing with compressible flows, the linear reconstruction for a generic variable $u(\mathbf{x})$ in Eq. 5.76 must be modified as follows:

$$\tilde{u}(\mathbf{x}_q) = u_i + \Phi_i \nabla u_i \cdot (\mathbf{x}_q - \mathbf{x}_i) \tag{5.89}$$

with $\Phi = [0, 1]$. This comes at a cost of an unavoidable accuracy deterioration in proximity of flow discontinuities and of a hampering of convergence, which can both vary greatly depending on the choice of the limiter function itself.

5.2.3.1 Venkatakrishnan's Limiter

Venkatakrishnan [163] developed his multidimensional limiter in order to overcome some deficiencies of Barth-Jespersen's limiter [17], namely the degradation of accuracy in nearly smooth flow regions and its poor convergence properties. The $\mathtt{min}(r, y)$ function used in [17] is replaced by a differentiable function:

$$\Phi(r) = \frac{r^2 + 2r}{r^2 + r + 2} \tag{5.90}$$

and the limiter is given by

$$\begin{cases} \Phi_{i,n_q} = \Phi\left(\frac{\Delta^+_{\mathtt{max}}}{\Delta^-}\right) & \text{for} \quad \Delta^- > 0 \\ \Phi_{i,n_q} = \Phi\left(\frac{\Delta^+_{\mathtt{min}}}{\Delta^-}\right) & \text{for} \quad \Delta^- < 0 \\ 1 & \text{for} \quad \Delta^- = 0 \end{cases}$$

where

$$\Delta^- = \boldsymbol{\Delta}\mathbf{x}^T_{n_q} \nabla u_i, \quad \Delta^+_{\mathtt{max}} = \mathtt{max}_i - u_i, \quad \Delta^+_{\mathtt{min}} = \mathtt{min}_i - u_i \tag{5.91}$$

in which

$$\mathtt{max}_i = \mathtt{max}_{j=0,\ldots,N_i}(u_i, u_j), \quad \mathtt{min}_i = \mathtt{min}_{j=0,\ldots,N_i}(u_i, u_j) \tag{5.92}$$

In order to maintain the accuracy in almost uniform regions where $\Delta^+ \approx \Delta^- \approx 0$, the flux function 5.90 is reformulated as follows:

$$\Phi\left(\frac{\Delta^+}{\Delta^-}\right) = \frac{{\Delta^+}^2 + 2\Delta^+\Delta^- + \epsilon}{{\Delta^+}^2 + \Delta^+\Delta^- + 2{\Delta^-}^2 + \epsilon} \tag{5.93}$$

where the ϵ term has to be appropriately scaled to be negligeable in non smooth flow regions and predominant where the flow is nearly uniform. To this end, ϵ can be estimated as

$$\epsilon = KU_0^2 \left(\frac{h}{L}\right)^3 \tag{5.94}$$

where $U_0 \simeq O(\mid u \mid)$ in the whole domain, h is a local characteristic length (e.g the average distance between the centroid of the current cell and the neighbors ones), L is a local characteristic solution length in the smooth flow regions and $K \approx O(1)$ is a user-defined constant.

5.2.3.2 MUSCL Limiters

Within the frame of the MUSCL reconstruction, several limiters are available in literature, e.g. [1, 152, 167]. For sake of completeness, we report here after the ones that have been implemented in COOLFluiD:

$$\Phi(r) = \max(0, \texttt{min}(1, r)) \quad \text{(MINMOD)} \tag{5.95}$$

$$\Phi(r) = \frac{3}{2}\max(0, \texttt{min}(\texttt{min}(2r, (1+3r)/4), 4)) \quad \text{(SMART)} \tag{5.96}$$

$$\Phi(r) = \max(0, \max(\texttt{min}(2r, 1), \texttt{min}(r, 2))) \quad \text{(SUPERBEE)} \tag{5.97}$$

$$\Phi(r) = \max(0, \texttt{min}(\texttt{min}(2r, (1+2r)/3), 2)) \quad \text{(KOREN)} \tag{5.98}$$

$$\Phi(r) = \max(0, \texttt{min}(\texttt{min}(2r, (1+r)/2), 2)) \quad \text{(MC)} \tag{5.99}$$

$$\Phi(r) = \frac{r(3r+1)}{(r+1)^2} \quad \text{(CHARM)} \tag{5.100}$$

$$\Phi(r) = \frac{3}{2}\frac{r + \mid r \mid}{r+2} \quad \text{(HCUS)} \tag{5.101}$$

$$\Phi(r) = \frac{2(r + \mid r \mid)}{r+3} \quad \text{(HQUICK)} \tag{5.102}$$

$$\Phi(r) = \frac{r^2 + r}{r^2 + 1} \quad \text{(Van Albada 1)} \tag{5.103}$$

$$\Phi(r) = \frac{2r}{r^2 + 1} \quad \text{(Van Albada 2)} \tag{5.104}$$

$$\Phi(r) = \frac{r + \mid r \mid}{1 + r} \quad \text{(Van Leer)} \tag{5.105}$$

$$\Phi(r) = \frac{3}{2}\frac{r^2 + r}{r^2 + r + 1} \quad \text{(OSPRE)} \tag{5.106}$$

Herein, r represents the ratio of successive gradients on the solution mesh. For example:

$$r_i = \frac{u_i - u_{i-1}}{u_{i+1} - u_i} \tag{5.107}$$

New arbitrarily complex limiter functions can be easily provided as user-defined functions directly in the COOLFluiD input file, without having to actually implement them in the code.

5.2.3.3 Historical Modification

The choice of one limiter function or another can improve the convergence, but after a drop of a few orders (2-3) of magnitude the residual tends always to oscillate. In order to cure this behaviour, we apply the treatment introduced in [37], namely the so-called *historical modification*. It consists in choosing

$$\Phi_i^n = \texttt{min}(\Phi_i^{n-1}, \Phi_i^n) \tag{5.108}$$

but only after a starting period sufficiently long to obtain a rough convergence of the solution, during which no special treatment is applied.
In our experience, this technique works well for many situations, even in presence of very strong discontinuities, but not in cases where complex shock/boundary layer interactions occur, e.g. in hypersonic flows on double cones. In the latter case, the historical modification might help to converge, but unfortunately to a wrong solution.
Another disadvantage of this technique is that it requires the non-negligeable storage in memory of the limiter values Φ_i^{n-1} computed during the previous iteration for each variable in each computational cell (in combination with the LS reconstruction) or face (if standard MUSCL is used): this cost turns out to be prohibitive for 3D multi-species reactive flows.

5.2.4 Discretization of Diffusive Fluxes

The discretization of the diffusive term leads to:

$$\oint_\Sigma \mathbf{F}^d \cdot \mathbf{n}\, d\Sigma = \sum_{f=1}^{N_f} \mathbf{G}_f \Sigma_f \tag{5.109}$$

where $\mathbf{G}_f = \mathbf{F}_f^d \cdot \mathbf{n}_f$ and the diffusive fluxes $\mathbf{F^d}$ typically depend on the primitive variables $\mathbf{P}$ and their gradients:

$$\mathbf{F^d} = \mathbf{F^d}(\mathbf{P}, \nabla\mathbf{P}) \tag{5.110}$$

5.2.4.1 Gradient Calculation

The application of Green-Gauss' theorem within a chosen control volume Ω^v can be used to determine the above mentioned gradients:

$$\nabla \mathbf{P} = \frac{1}{\Omega^v} \int_{\Omega^v} \nabla \mathbf{P} \, d\Omega^v = \frac{1}{\Omega^v} \oint_{\Sigma^v} \mathbf{P} \, \mathbf{n} \, d\Sigma^v \tag{5.111}$$

A popular choice for Ω^v on unstructured meshes is a diamond-shaped volume [38, 122] like the one in Fig. 5.4 which is built around the considered face and which includes all the face nodes, the left and right cell centers as vertices.

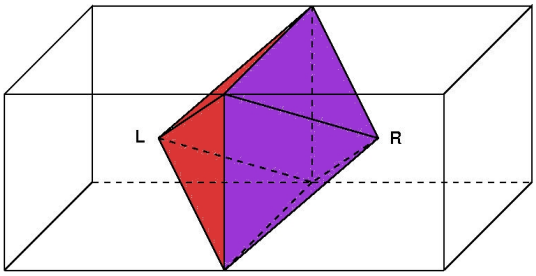


Figure 5.4: Diamond control volume for the calculation of the gradients for the diffusive fluxes.

The discretized version of Eq. 5.111 becomes:

$$\nabla \mathbf{P} = \frac{1}{\Omega^v} \sum_{f=1}^{N_f} \bar{\mathbf{P}}_f \mathbf{n}_f \Sigma_f^v, \quad \text{with} \quad \bar{\mathbf{P}}_f = \frac{1}{N_n^f} \sum_{j=1}^{N_n^f} \mathbf{P}_j^f \tag{5.112}$$

where $\bar{\mathbf{P}}_f$ is a face-averaged value of $\mathbf{P}$ calculated from the values $\mathbf{P}_j^f$ in the vertices of the diamond volume, whose number of faces is N_f. Moreover, N_n^f is the number of vertices in each face f of the control volume, i.e. N_n^f is equal to the space dimension of the problem (2 or 3).

5.2.4.2 Nodal Extrapolation

The values $\mathbf{P}_j^f$ in Eq. 5.112 include both cell centered values (i.e. the "true" degrees of freedom) and mesh vertex values. The latter, here called $\mathbf{P}_v$ are calculated starting from the first ones, $\mathbf{P}_c$, using a weighted averaged extrapolation:

$$\mathbf{P}_v = \frac{\sum_{c\in\zeta_v} \mathbf{P}_c \omega_c}{\sum_c \omega_c} \tag{5.113}$$

where ζ_v is the set of cell centers surrounding the considered vertex and possible definitions for the weights ω_c are

$$\omega_c = \frac{1}{\| \mathbf{\Delta x}_c \|} \qquad \text{distance based} \tag{5.114}$$

$$\omega_c = \frac{1}{\Omega_c} \qquad \text{volume based} \tag{5.115}$$

$$\omega_c = 1 + \lambda \cdot \mathbf{\Delta x}_c \qquad \text{Holmes-Connell} \tag{5.116}$$

with $\mathbf{\Delta x}_c = \mathbf{x}_c - \mathbf{x}_v$. In particular, in the last approach [38, 61, 122], the weighting coefficients $\lambda = (\lambda_x, \lambda_y, \lambda_z)$ are calculated by assuming that expression 5.113 is exact for a linear evolution of $\mathbf{P}$. We can therefore write:

$$\mathbf{P}_c = \mathbf{P}_v + \nabla \mathbf{P}_v \cdot \mathbf{\Delta x}_v \tag{5.117}$$

If we rewrite Eq. 5.113 as

$$\sum_{c\in\zeta_v} \omega_c (\mathbf{P}_v - \mathbf{P}_c) = 0 \tag{5.118}$$

and, because of Eq. 5.117, we obtain

$$\sum_{c\in\zeta_v} \omega_c (\nabla \mathbf{P}_v \cdot \mathbf{\Delta x}_v) \tag{5.119}$$

The latter relation should remain valid for any arbitrary value of the gradients $\nabla \mathbf{P}_v$ if $\mathbf{P}_v$ is linear in (x, y, z). By conveniently choosing three possible values $(1,0,0)$,$(0,1,0)$,$(0,0,1)$ for the gradients, we are left with an algebraic system of equations in the unknowns $(\lambda_x, \lambda_y, \lambda_z)$:

$$\begin{cases} \sum_c \omega_c \Delta x_c = \sum_c \Delta x_c + \lambda_x \sum_c \Delta x_c^2 + \lambda_y \sum_c \Delta x_c \Delta y_c + \lambda_z \sum_c \Delta x_c \Delta z_c = 0 \\ \sum_c \omega_c \Delta y_c = \sum_c \Delta y_c + \lambda_x \sum_c \Delta x_c \Delta y_c + \lambda_y \sum_c \Delta y_c^2 + \lambda_z \sum_c \Delta y_c \Delta z_c = 0 \\ \sum_c \omega_c \Delta z_c = \sum_c \Delta z_c + \lambda_x \sum_c \Delta x_c \Delta z_c + \lambda_y \sum_c \Delta y_c \Delta z_c + \lambda_z \sum_c \Delta z_c^2 = 0 \end{cases}$$

The system 5.2.4.2 simply requires the inversion of a 3x3 matrix for each mesh vertex. Since the weights ω_c do not depend on the flow solution, the system can be solved only once in a pre-processing step, if the nodal

coordinates don't change along the simulation (i.e. no mesh movement or adaptation is performed) as in our case. As far as memory requirements are concerned, the storage of three (two in 2D) λ_i coefficients per mesh vertex is needed to make this reconstruction procedure computationally efficient. Even though all the strategies for computing the weighting coefficients described so far have been implemented, the inverse distance-based approach has been our preferred choice for our calculations, because of its superior robustness.

Treatment of Corner Vertices. The nodal extrapolation may become tricky for boundary vertices. In the case of corner vertices shared by different topological surfaces, for instance, the nodal extrapolation must impose only one boundary condition. In our case, since the application of boundary conditions is based on ghost states, symmetrically built with respect to the internal cell centers and the boundary surface, only contributions from ghost states corresponding to one chosen topological region must be included for the computation of the corner values $\mathbf{P}_v$.

5.2.5 Discretization of Source Terms

The discretization of the source term appearing on the LHS of 5.16 is based on the cell centered value in a given cell i:

$$\int_{\Omega} \mathbf{S}(\mathbf{P})\ d\Omega \approx \mathbf{S}(\mathbf{P}_i)\Omega_i = \mathbf{S}_i\Omega_i \tag{5.120}$$

When the source term includes derivatives of some dependent variable p (e.g. the stress term $\tau_{\theta\theta}$ appearing in the case of axisymmetric Navier-Stokes), these are calculated by applying the Green-Gauss theorem, similarly to what explained in Sec. 5.2.4.1:

$$\nabla p = \frac{1}{\Omega_i}\int_{\Omega_i} \nabla p\ d\Omega_i = \frac{1}{\Omega_i}\oint_{\Sigma_i} p\ \mathbf{n}\ d\Sigma_i \tag{5.121}$$

In this case, however, the chosen control volume coincides with the volume of the current cell, Ω_i, while a diamond-shaped one was used for the computation of the diffusive fluxes. The discretized version of Eq. 5.121 is identical to Eq. 5.112.

5.2.6 Implicit scheme

5.2.6.1 Numerical Jacobian

The analytical jacobian of the space terms J_R in Eq. 5.12 incorporates two distinct contributions: one during a loop over all faces (boundary and internal ones) in order to assemble the contribution from the convective and diffusive fluxes, another one during a global loop over cells to calculate also the source term contributions.
The jacobian term can be computed numerically, by approximating each partial derivative by means of a forward finite difference formula. In our case, the jacobian of the residual $\mathbf{R}$ with respect to the k component of the update variables $\mathbf{P}$ can be expressed as follows [64]:

$$\left(\frac{\partial \mathbf{R}}{\partial \mathbf{P}_m}\right)_k \cdot \vec{1}_k \approx \frac{\mathbf{R}\left(\mathbf{P}_m + \varepsilon_k \vec{1}_k\right) - \mathbf{R}\left(\mathbf{P}_m\right)}{\varepsilon_k}. \tag{5.122}$$

where $k = 1 \dots N_{eq}$ and the index m depends on the degree of freedom in the computational stencil whose corresponding $\mathbf{P}$ variables are perturbed. When looping over the faces to assemble the fluxes, the following choice for m holds:

$$m = \begin{cases} \text{left}, \text{right} & \text{on internal faces} \\ \text{left} & \text{on boundary faces} \end{cases} \tag{5.123}$$

This means that on the boundary face, only the (left) internal state is perturbed, while the (right) ghost state is recomputed to satisfy the numerical boundary condition for the given perturbed internal state, allowing us not to include entries for the ghost states in the jacobian matrix. The same technique has been used in [40]. In the case of the source term jacobian, however, since the discretization is based only on the cell centered state, only that one is perturbed and the contribution from the neighbor states is completely neglected.
The perturbation ε_k in 5.122 is defined as [13, 64]:

$$\varepsilon_k = \delta \ \text{sign}\left(P_{m,k}\right) \max\left(\mid P_{m,k} \mid, u_k^{user}\right), \tag{5.124}$$

where the sign function is given by

$$\text{sign}\left(x\right) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}, \tag{5.125}$$

and where $\delta \leq 10^{-5}$ and u_k^{user} is a user defined value whose magnitude should be of the order of the corresponding update variable P_k.

5.2.6.2 Treatment of Source Terms

The choice of the given cell as control volume leads to an efficient implicit discretization of the source term itself, which does not depend on neighbor states. If we apply a Taylor expansion in time to the source term, we get

$$\mathbf{S}_i\left(\mathbf{P}_i^{n+1}\right) = \mathbf{S}_i\left(\mathbf{P}_i^n\right) + \frac{\partial \mathbf{S}_i}{\partial \mathbf{P}_i}\left(\mathbf{P}_i^n\right)\Delta\mathbf{P}_i^n \tag{5.126}$$

where the cross derivatives $\frac{\partial \mathbf{S}_i}{\partial \mathbf{P}_{j\neq i}}$ are identically equal to zero, meaning that the source term jacobian only contributes to the block diagonal of the global matrix and can be easily calculated on a separate loop over cells. If derivatives are present in the source terms, we keep frozen the extrapolated state values at the cell vertices during the jacobian calculation, so that Eq. 5.126 still holds.

5.2.6.3 Performance Considerations

In order to improve the performance of the solver, the influence of the left-/right state perturbations on the nodal extrapolation (Sec. 5.2.4.2) is neglected. In other words, the extrapolation from cell centers to vertices is performed only once at the beginning of each time step and kept frozen during the calculation of the numerical jacobian. In fact, a certain influence may be expected on the face-based gradient computation for the diffusive fluxes with the diamond volume approach described in Sec. 5.2.4.1, which relies on the face-vertex states and the left and right cell centers. However, numerical experiments didn't fully support this conjecture, since in most cases the convergence histories didn't seem to be affected at all by the inclusion of the perturbation effect on the nodal extrapolation. If we analyze the impact on the performance, the inclusion of this effect increases the global computational time by 20% in 2D and 50% in 3D, as detected while profiling the code.
Another simplification is made by assuming that the perturbation of the left-/right states of a face don't affect other states involved in the stencil for the high order reconstruction, the same approach adopted in [13]. This is theoretically justified by the relative small influence that each state has on the gradient computation in the least squares reconstruction (Sec. 5.2.2.2) and

in the multidimensional limiting procedure (Sec. 5.2.3) based on a full stencil, i.e. involving all the vertex neighbors of the cell, especially in 3D. This assumption is very important because it allows to keep also in second-order the sparsity pattern corresponding to the first-order scheme and reduces the amount of computation within the overlap region in a parallel calculation. Moreover, in a parallel simulation, only contributions to the rows of the global matrix corresponding to locally updatable states are considered, while contributions to rows corresponding to states belonging to the overlap region but locally not updatable are not computed.

5.2.7 Boundary conditions

All boundary conditions are imposed weakly by means of the well known ghost cell (state in our case) approach [58, 86]. This allows the numerical algorithm to treat uniformly both boundary and internal faces, the only minor difference being in the computation of the residual jacobian term as just described in Sec. 5.2.6.1. While a conventional implementation of inlets, outlets and mirror conditions is provided, an innovative algorithm has been developed for handling no-slip walls and is therefore described here after.

5.2.7.1 No-slip wall

This boundary condition [117] is implemented following a ghost node approach, where an ad-hoc treatment is employed in order to keep the temperatures always positive during the transient at the beginning of the computation, while rigorously preserving a consistent gradient calculation. To this end, whenever one (or more) of the temperatures (roto-translational, vibrational) calculated in the ghost node as $T_g^j = 2 * T_w - T_{in}^j$ becomes smaller than a user-specified positive value T_u, the ghost node, originally located in a position $\mathbf{X}_{g0}$, symmetric to the internal cell center $\mathbf{X}_{in}$ with respect to the wall boundary, is repositioned closer to the wall with a recursive dichotomycal algorithm until *all* the temperatures are at the same time bigger than T_u.
In pseudo C++ code this leads to the algorithm in C.L. 5.1, where T_w is the wall temperature, and T_j, the array of temperatures in the ghost node, is computed from a unidimensional linear extrapolation using the same slope $\frac{\Delta T}{\Delta n}$ between the internal (cell-center) and the wall values. The constant K must be set bigger than 1.0, and it is usually set to 2.0, meaning that, during each repositioning, the ghost node is moved half way between the old position and the wall. As a consequence of the repositioning, velocity

components must be linearly interpolated in a similar way to vanish at the wall, while extrapolated components like pressure remain unchanged in the ghost node, even if it is relocated.

```
1
    Xg = Xg_0;
    f = 1;
4
    while (!(T_j > T_u)) {
      f *= K;
7     Xg_1 = ((f - 1.)*X_w  + Xg)/f;  // new ghost node position
      T_j = T_in - (T_in - T_w)/distance(X_in, X_w)*
                                distance(X_in, Xg_1);
10    Xg = Xg_1;  // update ghost position
    }
```

Code Listing 5.1: Repositioning of the ghost node in no-slip wall condition

The simple but original (to the author's knowledge), boundary treatment here proposed is fully consistent with the cell-centered space discretization and, in our experience, contributes to enhance the robustness of the code during the initial transient phase of the simulation after having started from an initial uniform flow field.
To this end, care must be taken so that the solution extrapolation from the cell centers to the cell vertices, which is needed to compute the gradients of primitive variables for the diffusive fluxes, consistently accounts for the possibility of the ghost node movement. In the case of fixed temperature or velocity components, this can be achieved by strongly imposing the desired values on the wall vertices.

Radiative Equilibrium Wall The radiative equilibrium wall condition [13, 128] consists in imposing that the heat released from the gas into the wall by conduction (q_g^{cond}) and convection (q_g^{conv}) is exactly balanced by the heat lost by radiation (q_r) from the wall itself. This translates into the non linear equation:

$$Q(T_w) = q_g^{cond} + q_g^{conv} + q_r = 0 \tag{5.127}$$

to be solved iteratively at the wall. After having substituted the actual expressions for all terms, this expression becomes:

$$Q(T_w) = -\left(\lambda\frac{\partial T}{\partial n} + \sum_m^{N_m}\lambda_m^v\frac{\partial T_m^v}{\partial n}\right) + \sum_s^{N_s} h_s\mathbf{J}_s\cdot\mathbf{n}_w - \sigma(\epsilon_w^e T_w^4 - \epsilon_w^a T_\infty^4) = 0 \tag{5.128}$$

where $\sigma = 5.67\cdot10^{-8}$ $[W/m^2/K^4]$ is the Stefan-Boltzmann constant, ϵ_w^e and ϵ_w^a are respectively the wall emissivity and absorptivity and T_∞ is the distant body temperature. In Eq. 5.128 the contribution $\lambda^e\frac{\partial T^e}{\partial n}$ has been discarded, because if the electron temperature is separated, a condition of adiabatic wall is generally applied to it [118, 128]. If we discretize the normal gradients of temperature in Eq. 5.128 and we plug-in the repositioning algorithm presented above we get the functional:

$$Q(T_w) = -\sum_j^{N_j}\lambda^j\frac{T_{in}^j - T_g^j(T_w)}{\parallel \mathbf{X}_{in} - \mathbf{X}_g(T_w)\parallel} + \sum_s^{N_s} h_s(T_w)J_{s,n}(T_w) - \sigma(\epsilon_w^e T_w^4 - \epsilon_w^a T_\infty^4) \tag{5.129}$$

where the normal species diffusion fluxes $J_{s,n}$ are computed by solving the Stefan-Maxwell equations [13] with the gradients of normal molar fractions as driving forces.
The solution T_w of the non linear equation $Q(T_w) = 0$ is then obtained by applying a Newton procedure, where the possible repositioning of the ghost node is also taken into account at each Newton step. In order to enhance numerical robustness, as recommended in [13], the maximum variation of wall temperature for each boundary face between two time steps is limited to 100 or 200 K.

Local Equilibrium Wall This condition imposes the chemical composition of the gas mixture at the wall to be equal to the equilibrium values. According to what described in Sec. 4.2.1, under the local thermodynamic and chemical equilibrium assumption, the species molar composition at the wall $\mathbf{X}_w$ can be computed from the local temperature T_w, pressure p_w and elemental molar composition $\mathbf{X}_w^e$ by solving iteratively an algebraic system of equations [24, 94, 140]. In our implementation, p_w is extrapolated from the interior cell, whilst the wall temperature is either fixed or calculated from the radiative equilibrium condition, as explained in the previous section. The species mass fractions are then linearly interpolated between the inner and the possibly repositioned ghost node.
While in a full non-catalytic case, the heating due to gas convection is null,

i.e. $q_g^{conv} = \sum_s^{N_s} h_s \mathbf{J}_s \cdot \mathbf{n}_w = 0$, if we assume local equilibrium at the wall, this term plays a role, since the gradient of species fractions across the wall is not null and contributes to increase the heat flux.

5.2.8 Implementation issues

The Finite Volume discretization that has just been presented is tackled in COOLFluiD by means of the MCS pattern introduced in Sec. 2.4.1, as shown in the OMT diagram in Fig. 5.5.

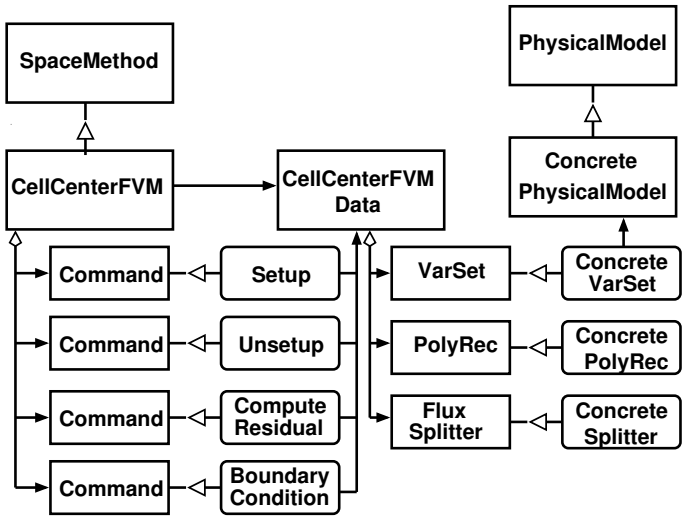


Figure 5.5: OMT diagram for the MCS pattern applied to the Finite Volume method.

To this end, the interface for a base `SpaceMethod` defined in C.L. 2.24 is implemented by a concrete `CellCenterFV` class, whose definition is presented in C.L. 5.2. Each specific action such as initialization, setup, unsetup, residual and system jacobian computation, application of boundary conditions etc. is handled by a dedicated Command.
Each available Command is given a name by the corresponding `Provider` object (see Sec. 2.5). As explained in Sec. 2.4.1.1, this gives to the end-user full control on the choice of the Commands to be used in the simulation for performeing a certain task: he/she can easily select alternative Com-

```
// --- CellCenterFV.hh --- //
class CellCenterFV : public SpaceMethod {
public:
  typedef SelfRegistPtr<Command<CellCenterFVData> > FVCom;
  // constructor, destructor, overridden virtual functions

private:
  // data to share between FVCom commands
  std::auto_ptr<CellCenterFVData> m_data;

  std::pair<FVCom,string> m_setup;   // setup Command
  std::pair<FVCom,string> m_unsetup; // unsetup Command

  // Commands computing the residual/jacobian
  std::pair<FVCom,string> m_computeSpaceRHS; // space part
  std::pair<FVCom,string> m_computeTimeRHS;  // time part

  // Commands that initialize the solution in the domain
  std::vector<FVCom> m_inits;
  std::vector<string> m_initsStr; // init Commands names

  // Commands that computes the boundary conditions (bc)
  std::vector<FVCom> m_bcs;
  std::vector<string> m_bcsStr; // bc Commands names
};
```

Code Listing 5.2: CellCenterFV class definition

mands to the default ones just by specifying their names in the input file. For instance, a user can select setup and unsetup Commands specific for a second order scheme based on least square reconstruction, which requires the allocation of many more data (cell limiters, cell gradients, weights etc.) than a first order one, in this way:

```
SpaceMethod = CellCenterFV
CellCenterFV.SetupCom = LeastSquareP1Setup
CellCenterFV.UnSetupCom = LeastSquareP1UnSetup
```

Analogously, the end-user can specify by name the boundary conditions, their parameters and the TRSs on which they are applied. For example:

```
# list of the BC Commands and corresponding aliases
CellCenterFV.BcComds = SuperInletFVCC MirrorFVCC
CellCenterFV.BcNames = SInlet Mirror

# TRS to which MirrorFVCC will be applied
```

```
CellCenterFV.Mirror.applyTRS = Symmetry

# input inlet profiles (function of x,y) for each variable
CellCenterFV.SInlet.Vars = x y
CellCenterFV.SInlet.Def = if(y<0.1,3,x^2) 50 0 2*sin(3*x)
CellCenterFV.SInlet.applyTRS = Inlet
```

```
// --- CellCenterFVData.hh --- //
class CellCenterFVData : public SpaceMethodData {
public:
  //constructor, destructor, configuration functions

  // polynomial reconstructor
  SafePtr<PolyReconstructor> getPolyReconstructor() const;

  // convective, diffusive flux, source term computers
  SafePtr<FluxSplitter> getFluxSplitter() const;
  SafePtr<ComputeDiffusiveFlux> getDiffFluxComputer() const;
  SafePtr<ComputeSourceTerm> getSourceTermComputer() const;
  SafePtr<NodalExtrapolator> getNodalExtrapolator() const;
  ...

 // linearization variable set
  SafePtr<ConvectiveVarSet> getLinearizationVar() const;

  // vectorial transformer from update to solution variables
  SafePtr<VarSetTransformer> getUpdateToSolutionVecTrans()
      const;

  // other accessors/mutators ...
private:
 SelfRegistPtr<FluxSplitter> m_fluSplitter;
 string m_fluSplitterStr;

 SelfRegistPtr<PolyReconstructor> m_polyRec;
 string m_polyRecStr;

 // ... the same for all the other objects
};
```

Code Listing 5.3: CellCenterFVData class definition

All FVComs share access to a policy class [10], `CellCenterFVData`, which groups together all the Strategy objects needed by the Commands to fulfill their job: for sake of simplicity only `PolyReconstructor` (the polynomial reconstructor) and `FluxSplitter` (the flux splitting scheme) are shown in Fig. 5.5, but, in reality, other Strategies are present, such as

`ComputeDiffusiveFlux` (the diffusive flux discretizer), `ComputeSourceTerm` (the source term computer) and the `NodalExtrapolator`.
Moreover, `CellCenterFVData` also aggregates a number of Perspectives like `VariableSets` and `VariableTransformers` (see Sec. 2.3.1), that provide a dynamic binding of the numerical algorithm to the physics. As shown in C.L. 5.3, `CellCenterFVData` inherits the interface of a parent `SpaceMethodData`, which provides access to parameters, Strategies and Perspectives (e.g. a `ConvectiveVarSet` for the update $\mathbf{P}$ and solution (or conservative) $\mathbf{U}$ variables, a `DiffusiveVarSet`, etc.) which are common to possibly all the concrete SpaceMethods. Likewise all MethodData, `CellCenterFVData` is also a self-configurable object, and this implies that the user can select the concrete Strategies by name at run-time: e.g. *Roe*, *AUSM*, *StegerWarming*, ... as `FluxSplitter`, *Constant* or *LeastSquare* for `PolyRecostructor`, etc., while the developer can implement and register new ones without needing to modify the client code, as long as the core virtual interfaces defined in the kernel or in the module itself are respected.

5.2.8.1 Decoupling between numerics and physics

In order to show the power of an object-oriented approach in decoupling the numerical algorithm from the physical description of the problem, we consider again, as an example, the case of the Roe scheme, presented in Sec. 5.2.1.1:

$$\mathbf{F}^{Roe} = \frac{1}{2}\left[\mathbf{F}_n(\mathbf{P}^L) + \mathbf{F}_n(\mathbf{P}^R)\right] - \frac{1}{2} \mid \mathbf{A}(\bar{\mathbf{Z}}) \mid \left[\mathbf{U}(\mathbf{P}^R) - \mathbf{U}(\mathbf{P}^L)\right] \quad (5.130)$$

In our case, the single terms in Eq. 5.130 are tackled as follows:

- the physical rotated fluxes $\mathbf{F}_n$ and the matrix $\mid \mathbf{A} \mid$ are given by the `ConvectiveVarSet` corresponding to the conservative variables $\mathbf{U}$;
- the average linearized state $\bar{\mathbf{Z}}$ in which the jacobian $\mid \mathbf{A} \mid$ is evaluated is provided by a `JacobianLinearizer` object;
- the analytical vectorial transformation of the type $\mathbf{U}(\mathbf{P})$, i.e. from update variables $\mathbf{P}$ to conservative variables $\mathbf{U}$, is implemented by a `VariableTranformer`;
- the global physics-independent expression of the flux is assembled in `RoeFlux`, a concrete class deriving from `FluxSplitter`.

This design allows developers to focus solely on the implementation of purely physics-dependent terms, spread into a few Perspective objects, in order to extend the applicability of the basic Roe scheme. In this way, Euler, Magneto Hydrodynamics, Thermo-chemical nonequilibrium models etc. have been easily handled without touching the core algorithm. Moreover, additional flexibility if offered on the choice of update **P** and linearization **Z** variables.
In case a different formulation or a more efficient hard-coded version of the scheme is preferred, one can always provide an alternative implementation (a derived class from `FluxSplitter` or `RoeFlux` itself) possibly more strictly bound to the physics, at the cost of a potential loss in flexibility and reusability.

5.3 Residual Distribution Schemes

During the past decade, multi-dimensional upwind RD schemes have proved to be an attractive alternative to the classical upwind finite volume approach based on one-dimensional Riemann solvers for the simulation of both steady [21, 34, 35, 113, 154, 156] and unsteady [8, 40, 99, 138] compressible flows. The main advantages of RD schemes include an outstanding shock capturing, due to the lower cross diffusion associated to a truly multi-dimensional upwinding and to the positivity property, and a compact stencil for a linearity preserving resolution, which does not require expensive polynomial reconstructions in order to guarantee second order accuracy and allow an easy and efficient parallelization [64, 158].
In this work, we have extended the RD method for the simulation of 2D axisymmetric viscous reentry flows in thermal and chemical non-equilibrium. We thereby generalize the work presented in [36] which was concerned with 2D inviscid chemically reacting high speed flows. The convective terms of the equations are discretized by means of a strictly conservative formulation of the RD method, denominated CRD, which is based on a redefinition of the positive system N scheme, so called Nc scheme [9, 137]. The latter conveniently does not require a specific set of variables (e.g. Roe parameter vector used in [36]) for the linearization of the flux jacobian, at the price of an additional contour integration of the convective flux in each computational cell. Second order accuracy is achieved by means of the blended Bx scheme [40, 41] where the blending coefficient is based on a shock capturing sensor.

5.3.1 System Schemes: General Concepts

Consider a parabolic-hyperbolic system in time of PDE's with convective, diffusive and source terms written in divergence form using Einstein summation convention:

$$\frac{\partial \mathbf{U}}{\partial \mathbf{P}} \frac{\partial \mathbf{P}}{\partial t} + \frac{\partial \mathbf{F}_i^c}{\partial x_i} = \frac{\partial \mathbf{F}_i^d}{\partial x_i} + \mathbf{S} \tag{5.131}$$

where $\mathbf{U}$ is the state vector of conservative variables with size N, $\mathbf{P}$ is the state vector of update variables (i.e. conservative, primitive, natural or any other type of variables chosen to store and update the solution at each time step), $\frac{\partial \mathbf{U}}{\partial \mathbf{P}}$ is the jacobian matrix defining the analytical transformation from $\mathbf{P}$ to $\mathbf{U}$, $\mathbf{A}_i^U = \frac{\partial \mathbf{F}_i^c}{\partial \mathbf{U}}$ are non-commuting $N \times N$ flux jacobian matrices and N_d is the space dimension of the problem.
We discretize the domain Ξ, with $\Xi \subset \Re^{N_d}$, into finite elements and we

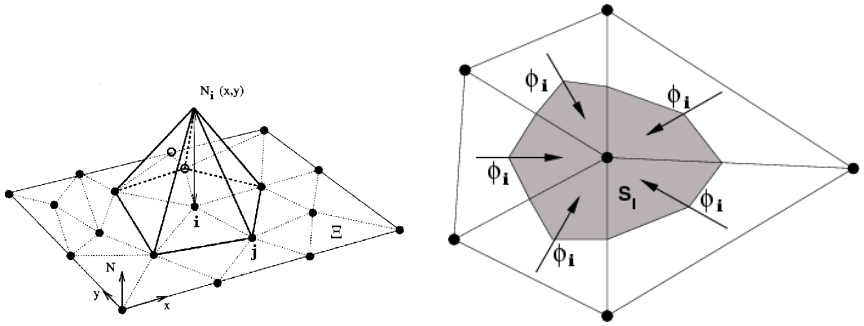
assume that each component of $\mathbf{P}$ is a linear combination of the values in the corresponding degrees of freedom inside each element:

$$\mathbf{P}^h(\mathbf{x},t) = \sum_{j=1}^{\#\text{nodes}} \mathbf{P}_j(t) N_j(\mathbf{x}) \tag{5.132}$$

where $N_j(\mathbf{x})$ represents the nodal basis function which satisfies the property

$$N_j(\mathbf{x}_k) = \delta_{jk} \tag{5.133}$$

where δ_{jk} is the *Kronecker delta*. Supposing to have linear triangular elements, the shape function corresponding to a piecewise linear interpolation is chosen, as shown in figure. 5.6a.



(a) Piece wise linear shape function $N_i(\mathbf{x})$. (b) Median dual cell surrounding a node.

Figure 5.6: Linear shape function and median dual cell surrounding a mesh vertex.

The basic idea of RD methods consists in distributing fractions of the cell residual defined by integrating Eq. 5.135 over an element to the nodes of a given cell, taking into account the directions of propagation of the physical signals, according to an upwind philosophy. The semi-discretized form of Eq.5.131 for a vertex l, assuming a lumped mass matrix [113, 154], then becomes:

$$\frac{\partial \mathbf{U}}{\partial \mathbf{P}}(\mathbf{P}_l)\frac{d\mathbf{P}_l}{dt}\Omega_l + \mathbf{\Phi}_l^c = \mathbf{\Phi}_l^d + \mathbf{\Phi}_l^s \tag{5.134}$$

where $\mathbf{\Phi}_l^c$, $\mathbf{\Phi}_l^d$ and $\mathbf{\Phi}_l^s$ are the three different nodal residuals corresponding respectively to convective, diffusive and source terms. Ω_l is the volume of

the median dual cell, a 2D example of which is shown in Fig. 5.6b for a mesh composed of triangles. Due to the lumping of the mass matrix, accuracy in time is reduced to first order in Eq. 5.134. We now consider in more detail each of the 3 residuals in Eq. 5.134.

5.3.2 Convective Term Discretization

We first define the convective cell residual $\mathbf{\Phi}^{c,\Omega}$ as the flux-balance over the element with volume Ω:

$$\mathbf{\Phi}^{c,\Omega} = \int_{\Omega} \nabla \cdot \mathbf{F}^c \, \mathrm{d}\Omega = \int_{\Omega} \frac{\partial \mathbf{F}_i^c}{\partial \mathbf{U}} \frac{\partial \mathbf{U}}{\partial x_i} \mathrm{d}\Omega = \int_{\Omega} \mathbf{A}_i^U \frac{\partial \mathbf{U}}{\partial x_i} \, \mathrm{d}\Omega \tag{5.135}$$

Applying the RD method to the convective term, we obtain the nodal residual by gathering the contributions of all cell residuals:

$$\mathbf{\Phi}_l^c = \sum_{\Omega_k \in \Xi_l} \mathbf{B}_l^{\Omega_k} \mathbf{\Phi}^{c,\Omega_k} \tag{5.136}$$

where Ξ_l is the set of neighboring cells sharing node l and $\mathbf{B}_l^{\Omega_k}$ are the so-called *distribution matrices*, which define the fraction of residual $\mathbf{\Phi}^c$ sent to node l inside each element with volume Ω_k, i.e. $\sum_{l \in \Omega_k} \mathbf{B}_l^{\Omega_k} = \mathbf{I}$. The actual expression for $\mathbf{B}$ depends on the selected distributive scheme, but it is usually function of the generalized upwind parameters $\mathbf{K}_k$, defined here after:

$$\mathbf{K}_k = \frac{1}{N_d} \bar{\mathbf{A}}_k^U, \quad \mathbf{A}_k^U = \mathbf{A}_i n_{i,k} \tag{5.137}$$

In the previous definition, $\mathbf{n}_k$ is the inward scaled normal with components $n_{i,k}$ corresponding to node k, i.e. the normal to the face opposite to node k, with length equal to the face area. In the case of a triangle with counter clockwise node numbering, the three normals are shown in Fig. 5.7a and defined by:

$$\mathbf{n}_1 = (y_2 - y_3)\,\vec{1}_x + (x_3 - x_2)\,\vec{1}_y \tag{5.138}$$

$$\mathbf{n}_2 = (y_3 - y_1)\,\vec{1}_x + (x_1 - x_3)\,\vec{1}_y \tag{5.139}$$

$$\mathbf{n}_3 = (y_1 - y_2)\,\vec{1}_x + (x_2 - x_1)\,\vec{1}_y \tag{5.140}$$

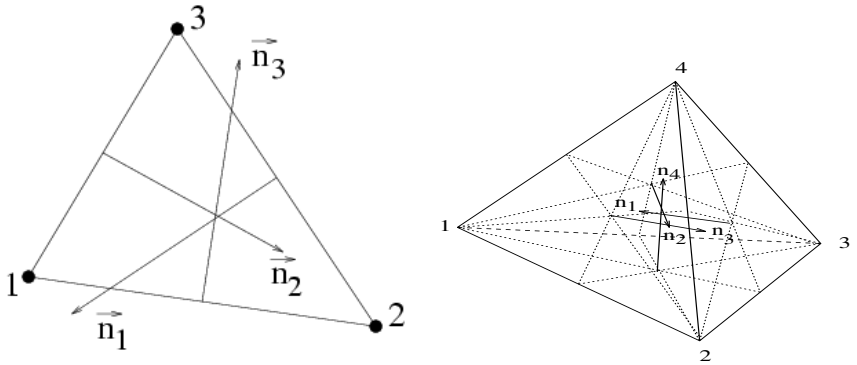
The inward scaled normals can be defined on tetrahedrons as well [20], as shown in Fig. 5.7b. For the given node numbering the normals are given by:

$$\mathbf{n}_1 = \frac{1}{2}\left(\vec{x}_4 - \vec{x}_2\right) \times \left(\vec{x}_3 - \vec{x}_2\right), \tag{5.141}$$

$$\mathbf{n}_2 = \frac{1}{2}\left(\vec{x}_3 - \vec{x}_1\right) \times \left(\vec{x}_4 - \vec{x}_1\right), \tag{5.142}$$

$$\mathbf{n}_3 = \frac{1}{2}\left(\vec{x}_4 - \vec{x}_1\right) \times \left(\vec{x}_2 - \vec{x}_1\right), \tag{5.143}$$

$$\mathbf{n}_4 = \frac{1}{2}\left(\vec{x}_2 - \vec{x}_1\right) \times \left(\vec{x}_3 - \vec{x}_1\right). \tag{5.144}$$



(a) Inward scaled normals in a triangle. (b) Inward scaled normals in a tetrahedron.

Figure 5.7: Inward normals in simplex elements (triangles and tetrahedra).

The method is conservative by construction if the cell residuals in Eq. 5.135 are based on a consistent contour integration, such that the fluxes cancel when summed for the two adiacent cells. Coming back to Eq. 5.137, the flux jacobian matrices $\bar{\mathbf{A}}^U = \mathbf{A}^U(\bar{\mathbf{Z}})$ associated to the distribution variables U are evaluated in a cell-averaged state $\bar{\mathbf{Z}}$, which should, in principle, guarantee conservation, if properly chosen (see further). Since the convective system in Eq. 5.131 is hyperbolic in time, K_k has a complete set of real eigenvalues and eigenvectors and can therefore be written:

$$\mathbf{K}_k = \frac{1}{N_d}\bar{\mathbf{R}}_k\bar{\mathbf{\Lambda}}_k\bar{\mathbf{L}}_k \tag{5.145}$$

where the columns of $\mathbf{R}_k$ contain the right eigenvectors of $\mathbf{A}_k^U$, $\mathbf{\Lambda}_k$ is the diagonal matrix of the eigenvalues and $\mathbf{L}_k = \mathbf{R}_k^{-1}$. The matrices $\mathbf{K}_k^+$ and $\mathbf{K}_k^-$ which are needed to define the upwind splitting are given by:

$$\mathbf{K}_k^{\pm} = \frac{1}{N_d}\bar{\mathbf{R}}_k\bar{\mathbf{\Lambda}}_k^{\pm}\bar{\mathbf{L}}_k. \tag{5.146}$$

Here $\mathbf{\Lambda}_k^+$ contains the positive and $\mathbf{\Lambda}_k^-$ the negative eigenvalues:

$$\bar{\mathbf{\Lambda}}_k^{\pm} = \frac{1}{N_d}(\bar{\mathbf{\Lambda}}_k \pm \mid \bar{\mathbf{\Lambda}}_k \mid) \tag{5.147}$$

5.3.2.1 Properties of the system schemes

Before proceeding further in the description of the numerical schemes, this section summarizes the properties of the RD method which are fully determined by the distribution criterion. The following properties correspond to a generalization to systems of the scalar ones, with the exception of the invariance for similarity transformations, which can only concern systems.

- **Multi-dimensional upwind** ($\mathcal{MU}$)
  In the case of hyperbolic systems the $\mathcal{MU}$ property must be interpreted in a *characteristic* sense, meaning that the residual of the *k-th* characteristic field of node l is not updated if the eigenvalue λ_l^k of the $\mathbf{K}_l$ is negative. Mathematically, this requires that $\mathbf{\Phi}_l^{c,\Omega} \propto \mathbf{K}_l^+$.

- **Positivity** ($\mathcal{P}$)
  If we rewrite the semi-discrete form in Eq. 5.134 as:

  $$\frac{\partial \mathbf{U}}{\partial t} + \sum_j \mathbf{C}_{lj}\left(\mathbf{U}_l - \mathbf{U}_j\right) = 0. \tag{5.148}$$

  then the scheme is positive if all the matrices $\mathbf{C}_{lj}$ are non-negative, i.e all their eigenvalues are positive or zero. Such schemes obey a maximum principle.

- **Linearity Preservation** ($\mathcal{LP}$)
  This property is satisfied if no updates are sent to the nodes when the cell residual $\mathbf{\Phi}^{c,\Omega}$ is zero. This implies that the eigenvalues of the distribution matrices $\mathbf{B}_l^{\Omega}$ are bounded. Such schemes are second order accurate.

- **Invariance for similarity transformations** ($\mathcal{IST}$)
  An important design property of the system distribution schemes is that the residual sent to the nodes in a certain set of variables $\mathbf{U}$, $\mathbf{\Phi}_l^{\Omega,U}$, is independent of the choice of variables for the actual distribution. Consider a set of variables $\mathbf{W}$, defined by the similarity transformation:

$$\partial\mathbf{W} = \frac{\partial\mathbf{W}}{\partial\mathbf{U}}\partial\mathbf{U}. \tag{5.149}$$

  The invariance property requires that the following relation between the residual expressed in the original variables, $\mathbf{\Phi}_l^U = \mathbf{B}_l^U\mathbf{\Phi}^{\Omega,U}$, and in other variables, $\mathbf{\Phi}_l^W = \mathbf{B}_l^W\mathbf{\Phi}^{\Omega,W}$, is satisfied:

$$\mathbf{\Phi}_l^U = \frac{\partial\mathbf{U}}{\partial\mathbf{W}}\mathbf{\Phi}_l^W \tag{5.150}$$

  The following requirement for distribution matrices for the different set of variables can easily be deduced:

$$\mathbf{B}_l^W = \frac{\partial\mathbf{W}}{\partial\mathbf{U}}\mathbf{B}_l^U\frac{\partial\mathbf{U}}{\partial\mathbf{W}}, \quad \mathbf{B}_l^U = \frac{\partial\mathbf{U}}{\partial\mathbf{W}}\mathbf{B}_l^W\frac{\partial\mathbf{W}}{\partial\mathbf{U}}. \tag{5.151}$$

  This shows that the residual can be computed in whatever set of variables (usually the most convenient one) and then transformed back to the one corresponding to the original variables (the variables in which the update is actually done), with exactly the same result that one would get by computing directly in the old variables.

5.3.2.2 Conservative linearization

One fundamental ingredient of the RD method is the linearization of the flux jacobian, which will be examined here after. To this end, let's reformulate Eq. 5.135 by introducing the linearization variables $\mathbf{Z}$ as follows:

$$\mathbf{\Phi}^{c,\Omega} = \int_\Omega \frac{\partial\mathbf{F}_i^c}{\partial\mathbf{U}}\frac{\partial\mathbf{U}}{\partial\mathbf{Z}}\frac{\partial\mathbf{Z}}{\partial x_i}\,\mathrm{d}\Omega = \int_\Omega \frac{\partial\mathbf{F}_i^c}{\partial\mathbf{Z}}\frac{\partial\mathbf{Z}}{\partial x_i}\,\mathrm{d}\Omega \tag{5.152}$$

If the matrix $\frac{\partial\mathbf{F}_i^c}{\partial\mathbf{Z}}(\bar{\mathbf{Z}})$ is assumed constant in the computational cell with volume Ω, by applying the Gauss theorem Eq. 5.152 can be approximated with:

$$\mathbf{\Phi}^{c,\Omega} \approx \tilde{\mathbf{\Phi}}^{c,\Omega} = \frac{\partial\mathbf{F}_i^c}{\partial\mathbf{Z}}(\bar{\mathbf{Z}})\oint_{\partial\Omega}\mathbf{Z}\,\mathbf{n}_i^{\mathrm{ext}}\,\mathrm{d}\partial\Omega = \mathbf{A}_i^U(\bar{\mathbf{Z}})\frac{\partial\mathbf{U}}{\partial\mathbf{Z}}(\bar{\mathbf{Z}})\oint_{\partial\Omega}\mathbf{Z}\,\mathbf{n}_i^{\mathrm{ext}}\,\mathrm{d}\partial\Omega \tag{5.153}$$

where, for an arbitrary choice of $\mathbf{Z}$, a conservation error $\delta\mathbf{\Phi}^{c,\Omega}$ is present:

$$\delta\mathbf{\Phi}^{c,\Omega} = \mathbf{\Phi}^{c,\Omega} - \tilde{\mathbf{\Phi}}^{c,\Omega} \neq 0 \tag{5.154}$$

Eq. 5.153 can be generalized further by introducing

$$\mathbf{A}_i^W = \frac{\partial \mathbf{W}}{\partial \mathbf{U}}\ \mathbf{A}_i^U \frac{\partial \mathbf{U}}{\partial \mathbf{W}} \tag{5.155}$$

i.e. the jacobian matrices associated to the distribution variables $\mathbf{W}$ which can used instead of $\mathbf{U}$ to define the upwind matrices $\mathbf{K}_k^W$:

$$\tilde{\mathbf{\Phi}}^{c,\Omega} = \frac{\partial \mathbf{U}}{\partial \mathbf{W}}(\bar{\mathbf{Z}})\ \mathbf{A}_i^W(\bar{\mathbf{Z}})\ \frac{\partial \mathbf{W}}{\partial \mathbf{Z}}(\bar{\mathbf{Z}}) \oint_{\partial\Omega} \mathbf{Z}\ \mathbf{n}_i^{\text{ext}} \mathrm{d}\partial\Omega \tag{5.156}$$

If we now assume a linear variation of $\mathbf{Z}$ over the faces of the cell (or equivalently we perform the contour integral of Eq. 5.156 with the trapezium rule), we obtain:

$$\tilde{\mathbf{\Phi}}^{c,\Omega} = \frac{\partial \mathbf{U}}{\partial \mathbf{W}}(\bar{\mathbf{Z}})\ \sum_{k=1}^{N_d+1} \mathbf{K}_k^W \tilde{\mathbf{W}}_k \tag{5.157}$$

where the consistent variables $\tilde{\mathbf{W}}_k$ are defined by

$$\tilde{\mathbf{W}}_k = \frac{\partial \mathbf{W}}{\partial \mathbf{Z}}(\bar{\mathbf{Z}})\ \mathbf{Z}_k \tag{5.158}$$

When the jacobian linearization performed in the $\mathbf{Z}$ variables is strictly conservative, i.e. $\mathbf{\Phi}^{c,\Omega} \equiv \tilde{\mathbf{\Phi}}^{c,\Omega}$ and therefore $\delta\mathbf{\Phi}^{c,\Omega} \equiv 0$, Eq. 5.157 yields the correct value of the flux integral over the cell and the jump relations are satisfied within the cell, leading to a proper capturing of discontinuities. The original formulation of RD schemes, a.k.a. Linear Residual Distribution (LRD), relied heavily on this constraint, giving no freedom in the choice of the linearization variables. This however lead to troubles for hyperbolic systems for which an exact linearization was not available, like in multi-phase flows [137, 138], ideal Magneto Hydro Dynamics [8] or multi-component reactive flows [36]. In [9] this limitation has been overcome by the CRD approach, where the fluctuation is computed consistently by contour-integrating the physical fluxes in the cell and the inconsistent linearization only plays a role in defining the distribution matrices, while maintaining the overall scheme strictly conservative.

Some examples of both LRD and CRD schemes will now be presented. For sake of consistency with the general analysis presented up to Eq. 5.157, we make use of the $\mathcal{IST}$ property and we suppose that the residual is distributed in a generic set of variables $\mathbf{W}$, not necessarily equal to the conservative variables $\mathbf{U}$.

5.3.2.3 $\mathcal{LRD}$ schemes

N-scheme $(\mathcal{MU}, \mathcal{P}, \mathcal{IST})$**.** The N-scheme is a first-order positive and multi-dimensional upwind scheme. The fluctuation distributed to node i of the scheme is defined as [156]:

$$\mathbf{\Phi}_i^N = \mathbf{K}_i^+ \left(\tilde{\mathbf{W}}_i - \tilde{\mathbf{W}}_{in} \right), \tag{5.159}$$

where

$$\tilde{\mathbf{W}}_{in} = \left(\sum_{j=1}^{N_d+1} \mathbf{K}_j^- \right)^{-1} \sum_{j=1}^{N_d+1} \mathbf{K}_j^- \tilde{\mathbf{W}}_j, \tag{5.160}$$

is the generalized inflow state $\tilde{\mathbf{W}}_{in}$. The matrices $\mathbf{K}_j^+$ and $\mathbf{K}_j^-$ are defined in Eq. 5.146.

LDA-scheme $(\mathcal{MU}, \mathcal{LP}, \mathcal{IST})$**.** The LDA-scheme is a second order linear multi-dimensional upwind scheme, which enjoys the linearity preservation property and therefore, as a result of Godunov's theorem, it cannot be positive.

$$\mathbf{\Phi}_i^{LDA} = \mathbf{B}_i^{LDA} \mathbf{\Phi}^{\Omega} = \mathbf{K}_i^+ \left(\sum_{j=1}^{N_d+1} \mathbf{K}_j^+ \right)^{-1} \mathbf{\Phi}^{\Omega}. \tag{5.161}$$

Existence of $\left(\sum_j K_j^+ \right)^{-1}$ is proved for any hyperbolic system.

Bx scheme $(\mathcal{MU}, \mathcal{LP}, \mathcal{IST})$**.** By using a non-linear blending of the distribution coefficients of a linearity preserving linear scheme (which is second-order but cannot be positive) with the N-scheme (linear and monotone, but only first-order) a linearity preserving and monotone scheme can be obtained. However, [40] proves that the blended N/LDA scheme cannot be locally positive, even though it normally performs in a positive manner. In this case the fraction of the element residual distributed to node i is defined as:

$$\mathbf{\Phi}_i^{Bx} = (1 - \Theta)\, \mathbf{\Phi}_i^{LDA} + \Theta \mathbf{\Phi}_i^N \tag{5.162}$$

The traditional definition of the coefficient Θ as a diagonal matrix with separate blending coefficients $\theta_m = \frac{|\phi_m^\Omega|}{\sum_j |\phi_{m,j}^N|}$ for each one of the equations of the system [35, 157] doesn't perform well, especially at high speeds and in combination with implicit time stepping, i.e. precisely in our type of applications. In order to overcome these deficiencies, we adopt a new formulation that has been introduced in [41], denominated Bx, in which Θ is based on an elementwise shock capturing sensor sc:

$$\Theta = \min(1, sc^2 h) \tag{5.163}$$

$$sc = \left(\frac{\nabla w \cdot \mathbf{v}}{\delta_{wv}} \right)^+ \approx \left(\frac{\sum_j w_j (\mathbf{n}_j \cdot \mathbf{v})}{N_d \ \Omega \ \delta_{wv}} \right)^+ \tag{5.164}$$

where w is a flow variable but not necessarily the pressure as in [41] (e.g. we use density in cases where contact discontinuities must be detected); the term $\delta_{wv} \approx (w_{max} - w_{min}) \mid \bar{V} \mid$ is a global variation scale for w multiplied by the magnitude of the mean velocity in the whole domain; h is the diameter of a circle/sphere with the same volume Ω as the considered element.

5.3.2.4 $\mathcal{CRD}$ schemes

As already explained in Sec. 5.3.2.2, the RD schemes in their original formulation can only be used if a conservative linearization is available for the jacobians of the fluxes, which is unfortunately not always the case.
[9] shows that the schemes can anyway be reformulated in such a way that conservation is retained whatever kind of linearization variables $\mathbf{Z}$ are employed to compute the cell jacobians $\mathbf{A}_i^W(\bar{\mathbf{Z}})$ in Eq. 5.156. The new formulation relies on the contour integration of the fluxes in the element, i.e. the calculation of the fluctuation by means of an appropriate quadrature rule such as Simpson's or Gauss's:

$$\mathbf{\Phi}^{c,\Omega} = \oint_{\partial\Omega} \mathbf{F}^c \cdot \mathbf{n} \, d\partial\Omega = \sum_{f \in \partial\Omega} \left(m_f \sum_{q=1}^{N_q^f} \left(w_q \mathbf{F}_q^c \right) \cdot \mathbf{n}_f \right) \tag{5.165}$$

where

- the outer sum is extended to each face f of the element;
- N_q^f is the number of quadrature points required by the chosen integration rule on face f;

- w_q is the weight associated to point q;
- $\mathbf{F}_q^c$ is the value of $\mathbf{F}^c$ in the quadrature point q;
- m_f are coefficients depending on the jacobian of the geometric transformation from the reference face element to the actual face element;
- n_f is the outward unit normal.

To give an example, for a 2D Simpson's integration rule one obtains:

$$N_q^f = 3 \tag{5.166}$$

$$w_1 = w_3 = \frac{1}{6},\ w_2 = \frac{2}{3} \tag{5.167}$$

$$m_f = A_f \tag{5.168}$$

with A_f being the area of the considered face.

Nc scheme. The CRD version of the N scheme is defined as:

$$\mathbf{\Phi}_i^{Nc} = \mathbf{K}_i^+ \left(\tilde{\mathbf{W}}_i - \tilde{\mathbf{W}}_{in} \right) \tag{5.169}$$

where $\tilde{\mathbf{W}}_{in}$ can be expressed as:

$$\tilde{\mathbf{W}}_{in} = \left(\sum_{j=1}^{N_d+1} \mathbf{K}_j^+ \right)^{-1} \left(\sum_{j=1}^{N_d+1} \mathbf{K}_j^+ \ \tilde{\mathbf{W}}_j - \mathbf{\Phi}^{c,\Omega} \right) \tag{5.170}$$

With some mathematical manipulation, the same scheme can be rewritten in the following way, as suggested in [8]:

$$\mathbf{\Phi}_i^{Nc} = \mathbf{\Phi}_i^N(\bar{\mathbf{Z}}) + \mathbf{B}_i^{LDA}(\bar{\mathbf{Z}})\ \delta\mathbf{\Phi}^{c,\Omega} \tag{5.171}$$

where $\delta\mathbf{\Phi}^{c,\Omega}$ is given by 5.154, while $\mathbf{\Phi}_i^N$ and $\mathbf{B}_i^{LDA}$ are based on the chosen linearization variables $\mathbf{Z}$. This formulation clearly shows that the Nc scheme reduces to its LRD version if the linearization is strictly conservative, as in the case of the Roe parameter vector for the Euler equations. On the other hand, the Nc scheme is always conservative for arbitrary linearization variables because

$$\sum_{i=1}^{N_d+1} \mathbf{\Phi}_i^{Nc} \equiv \mathbf{\Phi}^{c,\Omega} \tag{5.172}$$

by construction and the conservation error $\delta\mathbf{\Phi}^{c,\Omega}$ is distributed with matrices $\mathbf{B}_i^{LDA}$ based on a non-conservative linearization.

LDAc scheme. The system LDAc scheme is straightforward to compute:

$$\phi_i^{LDAc} = B_i^{LDA}\ \mathbf{\Phi}^{c,\Omega} \tag{5.173}$$

where the distribution matrix $\mathbf{B}_i^{LDA}$ is the same as in Eq. 5.161. The LDAc scheme retains the linear preserving property and the lack of monotonicity due to Godunov's theorem.

Bxc Scheme. The blending of the Nc and LDAc system schemes by means of the Θ coefficient defined in 5.163 gives:

$$\mathbf{\Phi}_i^{Bxc} = (1-\Theta)\,\mathbf{\Phi}_i^{LDAc} + \Theta\mathbf{\Phi}_i^{Nc} \tag{5.174}$$

Since both LDAc and Nc schemes are conservative, the Bxc scheme is also conservative.

5.3.3 Diffusive Term Discretization

As extensively explained in [35, 154, 157], the whole residual distribution technique can be cast into a Petrov-Galerkin FEM. In this section, we show how this analogy can be used in order to discretize the diffusive terms of our system of equations.
To start with, let's transform the diffusive term in Eq. 5.131 into an equivalent variational formulation, by multiplying it by the weight functions $\mathbf{w}_l(\mathbf{x})$ associated to an arbitrary node l and integrating the result by part over the whole domain Ω:

$$\mathbf{\Phi}_l^d = \int_\Omega \mathbf{w}_l\ \frac{\partial \mathbf{F}_i^d}{\partial x_i}\ d\Omega = \int_\Omega \frac{\partial}{\partial x_i}(\mathbf{w}_l \mathbf{F}_i^d)\ d\Omega - \int_\Omega\ \mathbf{F}_i^d\ \frac{\partial \mathbf{w}_l}{\partial x_i} d\Omega \tag{5.175}$$

where $\mathbf{\Phi}_l^d$ is the fraction of diffusive flux which is distributed to node l. If we apply the Gauss theorem to the first integral, we get

$$\mathbf{\Phi}_l^d = \oint_{\partial\Omega} \mathbf{w}_l\ \mathbf{F}_i^d \cdot \mathbf{n}\ d\partial\Omega - \int_\Omega\ \mathbf{F}_i^d\ \frac{\partial \mathbf{w}_l}{\partial x_i} d\Omega \tag{5.176}$$

where the boundary integral vanishes for interior nodes. The second term in Eq. 5.176 is usually calculated as a summation over the individual cells

in which the computational domain is subdivided, leading to the following definition for $\mathbf{\Phi}_l^d$ in Eq. 5.134:

$$\mathbf{\Phi}_l^d = -\sum_{\Omega_k \in \Xi_l} \int_{\Omega_k} \mathbf{F}_i^d \, \frac{\partial \mathbf{w}_l^{\Omega_k}}{\partial x_i} d\Omega_k \tag{5.177}$$

If we choose the $\mathbf{w}_l$ to be Petrov-Galerkin weights, defined as

$$\mathbf{w}_l(\mathbf{x}) = \sum_{\Omega_k \in \Xi_l} \mathbf{w}_l^{\Omega_k}(\mathbf{x}) = N_l(\mathbf{x})\, \mathbf{I} + \sum_{\Omega_k \in \Xi_l} \left(B_l^{\Omega_k} - \frac{1}{N_d + 1}\, \mathbf{I} \right) \alpha^{\Omega_k}(\mathbf{x}) \tag{5.178}$$

where $N_l(\mathbf{x})$ is the nodal basis function, $B_l^{\Omega_k}$ are the distribution matrices and $\alpha^{\Omega_k}(\mathbf{x}) = 1$ inside element Ω_k and 0 outside, then $\frac{\partial \mathbf{w}_l}{\partial x_i} \equiv \frac{\partial N_l}{\partial x_i}\mathbf{I}$. If we make use of the geometric relation $\frac{\partial N_l}{\partial x_i} = \frac{n_{l\,i}}{\Omega_k\ N_d}$, which holds inside a linear simplex element, then Eq. 5.177 can be expressed as:

$$\mathbf{\Phi}_l^d = -\sum_{\Omega_k \in \Xi_l} \frac{1}{\Omega_k\ N_d} \int_{\Omega_k} \mathbf{F}^d \cdot \mathbf{n}_l \; d\Omega_k = -\sum_{\Omega_k \in \Xi_l} \frac{1}{N_d} (\mathbf{F}^d \cdot \mathbf{n}_l)(\bar{\nu}, \mathbf{n}_l, \bar{\mathbf{P}}, \nabla\tilde{\mathbf{P}}) \tag{5.179}$$

where $\bar{\nu}$ and $\bar{\mathbf{P}}$ represent respectively the array of transport properties and the update variables computed in the only quadrature point if we consider a 1-point Gauss integration applied to the element. In this case, such a state also corresponds to the cell-averaged state which is used for the linearization of the jacobian matrices [113]. Moreover, $\mathbf{n}_l$ is the inward nodal normal and $\nabla\tilde{\mathbf{P}}$ are gradients of primitive variables. The latter are calculated with a standard linear Finite Element interpolation in each element:

$$\nabla\tilde{\mathbf{P}} = \frac{1}{\Omega_k\ N_d}\, \frac{\partial \tilde{\mathbf{P}}}{\partial \mathbf{P}} \sum_{j=1}^{N_d+1} \mathbf{P}\ \mathbf{n}_j \tag{5.180}$$

where $\frac{\partial \tilde{\mathbf{P}}}{\partial \mathbf{P}}$ is the jacobian of the transformation from the update variables $\mathbf{P}$ (the ones in which the solution is stored, i.e. the ones for which a linear representation in the element is assumed) to the primitive variables $\tilde{\mathbf{P}}$, whose gradients appear in the definition of the physical diffusive flux.
For example, in the case of flows in thermo-chemical non-equilibrium, the physical diffusive fluxes require the gradients of mass fractions y_s, while the update variables provide the partial densities ρ_s, together with the velocity components and the temperatures:

$$\mathbf{P} = [\rho_s \;\; \mathbf{u} \;\; T \;\; T_v], \quad \tilde{\mathbf{P}} = [y_s \;\; \mathbf{u} \;\; T \;\; T_v] \tag{5.181}$$

As a result, given that $y_s = \rho_s/\rho$, the matrix $\frac{\partial \tilde{\mathbf{P}}}{\partial \mathbf{P}}$ reduces to:

$$\frac{\partial \tilde{\mathbf{P}}}{\partial \mathbf{P}} = \begin{pmatrix} (\delta_{ij} - y_i)/\rho & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.182}$$

5.3.4 Source Term Discretization

If we apply a variational formulation to the source term $\mathbf{S}$ in Eq. 5.131 within the computational domain Ω and we consider its contribution to the nodal residual, as in the semi-discretized form in Eq. 5.134, we obtain [36]:

$$\mathbf{\Phi}_l^s = \sum_{\Omega_k \in \Xi_l} \int_{\Omega_k} \mathbf{w}_l^{\Omega_k} \; \mathbf{S} \; d\Omega_k \tag{5.183}$$

At this point, it only remains to specify a quadrature rule for discretizing the RHS in Eq. 5.183, the simplest possible choice being a one-point quadrature rule, leading to:

$$\mathbf{\Phi}_l^s = \sum_{\Omega_k \in \Xi_l} \mathbf{w}_l^{\Omega_k}(\mathbf{x}_c) \; \mathbf{S}(\mathbf{x}_c) \; \Omega_k = \sum_{\Omega_k \in \Xi_l} B_l^{\Omega_k} \; \mathbf{S}_c \; \Omega_k \tag{5.184}$$

where $\mathbf{x}_c$ is the centroid of the simplex element and $B_l^{\Omega_k}$ is the distribution matrix which is defined as

$$B_l^{\Omega_k} = \frac{1}{\Omega_k} \int_{\Omega_k} \mathbf{w}_l^{\Omega_k} \; d\Omega_k \tag{5.185}$$

Eq. 5.184 shows that the source term can be distributed exactly like the convective term, at least if a distribution matrix can be defined for the convective scheme. However, if the N scheme is used, the LDA distribution matrix is typically applied to the source term [36, 155]:

$$\mathbf{\Phi}_l^s = \sum_{\Omega_k \in \Xi_l} \mathbf{B}_l^{LDA,\Omega_k} \; \mathbf{S}_c \; \Omega_k \tag{5.186}$$

The same can be done with the Nc scheme, but in this case this corresponds to upwind the source term together with the convective fluctuation. To this end, let's consider the sum of the fractions of convective and source term

residuals coming from one element, after having recalled Eqs. 5.154 and 5.171:

$$\begin{aligned}(\mathbf{\Phi}_i^c + \mathbf{\Phi}_i^s)^\Omega &= \mathbf{\Phi}_i^{Nc}\,|_{\mathbf{\Phi}^{c,\Omega}} + B_i^{LDA}\mathbf{S}_c \\ &= \mathbf{\Phi}_i^N + \mathbf{B}_i^{LDA}\,\delta\mathbf{\Phi}^{c,\Omega} + B_i^{LDA}\mathbf{S}_c \\ &= \mathbf{\Phi}_i^N + \mathbf{B}_i^{LDA}(\mathbf{\Phi}^{c,\Omega} - \tilde{\mathbf{\Phi}}^{c,\Omega}) + \mathbf{B}_i^{LDA}\mathbf{S}_c \\ &= \mathbf{\Phi}_i^N + \mathbf{B}_i^{LDA}(\mathbf{\Phi}^{c,\Omega} + \mathbf{S}_c - \tilde{\mathbf{\Phi}}^{c,\Omega}) \\ &= \mathbf{\Phi}_i^{Nc}\,|_{\mathbf{\Phi}^{c,\Omega}+\mathbf{S}_c}\end{aligned} \tag{5.187}$$

Analogously, one can prove that a similar result holds also in the case of the Bxc scheme:

$$\begin{aligned}(\mathbf{\Phi}_i^c + \mathbf{\Phi}_i^s)^\Omega &= \mathbf{\Phi}_i^{Bxc}\,|_{\mathbf{\Phi}^{c,\Omega}} + \mathbf{B}_i^{LDA}\mathbf{S}_c \\ &= (1-\Theta)\,\mathbf{\Phi}_i^{LDAc} + \Theta\mathbf{\Phi}_i^{Nc} + \mathbf{B}_i^{LDA}\mathbf{S}_c \\ &= (1-\Theta)\,\mathbf{B}_i^{LDA}\mathbf{\Phi}^{c,\Omega} + \Theta\mathbf{\Phi}_i^{Nc} + \mathbf{B}_i^{LDA}\mathbf{S}_c \\ &= (1-\Theta)\,\mathbf{B}_i^{LDA}(\mathbf{\Phi}^{c,\Omega} + \mathbf{S}_c) + \Theta(\mathbf{\Phi}_i^{Nc} + \mathbf{B}_i^{LDA}\mathbf{S}_c) \\ &= (1-\Theta)\,\mathbf{\Phi}_i^{LDAc}\,|_{\mathbf{\Phi}^{c,\Omega}+\mathbf{S}_c} + \Theta\mathbf{\Phi}_i^{Nc}\,|_{\mathbf{\Phi}^{c,\Omega}+\mathbf{S}_c} \\ &= \mathbf{\Phi}_i^{Bxc}\,|_{\mathbf{\Phi}^{c,\Omega}+\mathbf{S}_c}\end{aligned} \tag{5.188}$$

5.3.5 Implicit scheme

5.3.5.1 Numerical Jacobian

The jacobian of the space terms J_R in Eq. 5.12 incorporate all the contributions to a given node coming from all neighboring cells sharing that node. The global jacobian matrix is assembled by looping over the nodes of each cells and by accumulating the computed numerical jacobian of the sum of convective, diffusive and source term residual in each node. In other words, residual and residual jacobian for interior nodes are calculated within the same single loop over cells.
As in the case of the Finite Volume solver (see Sec. 5.3.5.1), the jacobian term is computed numerically, by approximating each partial derivative by means of the same forward finite difference formula [64]:

$$\left(\frac{\partial\mathbf{R}}{\partial\mathbf{P}_m}\right)_k \cdot \vec{1}_k \approx \frac{\mathbf{R}\left(\mathbf{P}_m + \varepsilon_k\vec{1}_k\right) - \mathbf{R}\left(\mathbf{P}_m\right)}{\varepsilon_k}. \tag{5.189}$$

where $k = [1, N_{eq}]$ and the index m indicates, in this case, the cell node whose corresponding $k-th$ component of its state vector $\mathbf{P}$ is being perturbed.
Additional jacobian contributions are distributed to the boundary nodes, while applying the corresponding conditions. The actual way of computing these contributions strictly varies with the type of boundary condition (strong or weak) to enforce, as described right after.

5.3.5.2 Implicit boundary conditions

We describe here two boundary conditions which have been implemented in order to deal with viscous flows in thermo-chemical non-equilibrium. They both represent an extension of the concepts introduced in [64] and [154] about strong and weak boundary conditions for RD, which, in their cases, were applied to the simulation of non-reactive Navier-Stokes flows. The interested reader should refer to [64, 154] in order to learn more about other type of boundary conditions conventionally used.

Weak impermeable wall. If the impearmeability condition $u_n = \mathbf{u} \cdot \mathbf{n}$ is imposed on a boundary face with outward normal $\mathbf{n}$, the corresponding flux function at the wall becomes:

$$(\mathbf{F}(\mathbf{P}) \cdot \mathbf{n})_{wall} = \begin{pmatrix} \mathbf{0} \\ p\mathbf{n} \\ 0 \\ 0 \end{pmatrix} \tag{5.190}$$

In reality, in order to enforce such a condition, it is necessary to correct the actual flux function $\mathbf{F}(\mathbf{P}) \cdot \mathbf{n}$ at the wall, which is usually different from Eq. 5.190, by distributing a discrete approximation of the term:

$$\int_{\Sigma_f} \mathbf{F}^{corr}(\mathbf{P}) \cdot \mathbf{n} \; d\Sigma_f = \int_{\Sigma_f} \left[(\mathbf{F}(\mathbf{P}) \cdot \mathbf{n})_{wall} - \mathbf{F}(\mathbf{U}) \cdot \mathbf{n} \right] \; d\Sigma_f \tag{5.191}$$

where in the case of a flow in thermo-chemical non-equilibrium the correcting normal flux is given by

$$\mathbf{F}^{corr}(\mathbf{P}) \cdot \mathbf{n} = \mathbf{F}_n^{corr}(\mathbf{P}) = - \begin{pmatrix} \rho_s u_n \\ \rho \mathbf{u} u_n \\ \rho H u_n \\ \rho e_v u_n \end{pmatrix} \tag{5.192}$$

If the trapezium rule is chosen to approximate the integral in Eq. 5.191, in a 2D case the following residuals are distributed to the nodes:

$$\mathbf{\Phi}_1^{BC} = \frac{1}{2}\left\{\alpha \mathbf{F}_n^{corr}(\mathbf{P}_1) + (1-\alpha)\mathbf{F}_n^{corr}(\mathbf{P}_2)\right\} \tag{5.193}$$

$$\mathbf{\Phi}_2^{BC} = \frac{1}{2}\left\{\alpha \mathbf{F}_n^{corr}(\mathbf{P}_2) + (1-\alpha)\mathbf{F}_n^{corr}(\mathbf{P}_1)\right\} \tag{5.194}$$

where the coefficient α is usually taken equal to 1.0. Moreover, in an implicit simulation also the following jacobian terms must be distributed:

$$\sum_{j=1,2} \frac{\partial \mathbf{\Phi}_1^{BC}}{\partial \mathbf{P}_j}\Delta\mathbf{P}_j = \frac{1}{2}\left\{\alpha \frac{\partial \mathbf{F}_n^{corr}}{\partial \mathbf{P}}(\mathbf{P}_1)\ \Delta\mathbf{P}_1 + (1-\alpha)\frac{\partial \mathbf{F}_n^{corr}}{\partial \mathbf{P}}(\mathbf{P}_2)\ \Delta\mathbf{P}_2\right\} \tag{5.195}$$

$$\sum_{j=1,2} \frac{\partial \mathbf{\Phi}_2^{BC}}{\partial \mathbf{P}_j}\Delta\mathbf{P}_j = \frac{1}{2}\left\{\alpha \frac{\partial \mathbf{F}_n^{corr}}{\partial \mathbf{P}}(\mathbf{P}_2)\ \Delta\mathbf{P}_2 + (1-\alpha)\frac{\partial \mathbf{F}_n^{corr}}{\partial \mathbf{P}}(\mathbf{P}_1)\ \Delta\mathbf{P}_1\right\} \tag{5.196}$$

The jacobian matrix $\partial \mathbf{F}_n^{corr}/\partial \mathbf{P}$ is computed analytically and for $\mathbf{P} = [\rho_s\ \ \mathbf{u}\ \ T\ \ T_v]$ (natural variables) it reads:

$$\frac{\partial \mathbf{F}_n^{corr}}{\partial \mathbf{P}} = \begin{pmatrix} u_n\ \delta_{ij} & \rho_i n_j & 0 & 0 \\ u_i u_n & \rho(u_n\ \delta_{ij} + u_i n_j) & 0 & 0 \\ (\rho H)_{\rho_i} u_n & \rho(H n_j + u_j u_n) & \rho H_T u_n & \rho e_{vT_v} u_n \\ u_n e_{v,j}/M_j & \rho e_v n_j & 0 & \rho e_{vT_v} u_n \end{pmatrix} \tag{5.197}$$

where the thermodynamic derivatives $(\rho H)_{\rho_i}$ and H_T in the case of a neutral mixture are

$$(\rho H)_{\rho_i} = E + \frac{RT + e_{v,j}}{M_j}, \quad H_T = \frac{C_{v,tr}}{M} + \frac{p}{\rho T}. \tag{5.198}$$

Strong isothermal no-slip wall. The no-slip wall boundary condition is enforced strongly, so that each nodal state on the wall obeys the prescribed conditions. In the case of interest, we have to impose

$$\mathbf{u}_b = 0, \quad T_b = T_b^v = T_{\text{wall}} \tag{5.199}$$

Since we use the natural variables $\mathbf{P}$ for the solution update and for the numerical perturbation of the residual, the implicit treatment of such a boundary condition translates into replacing the rows corresponding to the variables $\mathbf{u}, T, T^v$ for the wall states b of the jacobian matrix $\frac{\partial \mathbf{R}}{\partial \mathbf{P}}$ with

$$K_b \; \Delta \mathbf{u}_b = 0, \quad K_b \; \Delta T_b = 0, \quad K_b \; \Delta T_b^v = 0 \tag{5.200}$$

where the coefficient

$$K_b = \frac{\Omega_b}{\Delta t} = \frac{1}{\mathrm{CFL}} \left(\sum_{\Omega_k \in \Xi_b} \frac{1}{N_d} (\bar{\lambda}^+_{\max})_b^{\Omega_k} \| \mathbf{n}_b^{\Omega_k} \| + \sum_{\Omega_k \in \Xi_b} \frac{\bar{\nu} \| \mathbf{n}_b^{\Omega_k} \|^2}{{N_d}^2 \; \Omega_k} \right) \tag{5.201}$$

comes from the enforcement of the positivity condition in the explicit scheme. In Eq. 5.201, $(\bar{\lambda}^+_{\max})_b^{\Omega_k}$ is the linearized maximum positive eigenvalue, $\bar{\nu}$ is the cell averaged kinematic viscosity, $\mathbf{n}_b^{\Omega_k}$ is the inward normal corresponding to node b inside cell Ω_k.
K_b is used for scaling purposes here, in order to keep a good conditioning number of the system matrix when approaching convergence, since $\lim_{\Delta t \to \infty} K_b = 0$.
Finally, no special treatment has to be applied to the jacobian rows corresponding to the partial densities of the wall states.

Chapter 6

Numerical results

We present in this chapter some of the main results we have achieved by applying the numerical methods introduced in Chapter 5 to the aerothermodynamic physical models described in 4. Wherever possible, we make direct validation against experimental data and/or reference solutions available in literature.

6.1 RTO Task Group 43 Topic No. 2

Starting from June 2007, the Von Karman Institute has been involved in a RTO task group dealing with the *Assessment of Aerothermodynamic Flight Prediction Tools Through Ground and Flight Experimentation* together with US leading institutions like NASA Ames, NASA Langley, US Air Force Research Laboratory (AFRL), Calspan University of Buffalo Research Center (CUBRC), University of Minnesota and DLR, ESA and EPFL on the European side. In particular, the topic No. 2 focuses on a further assessment of CFD for the specific issue of shock interactions and control surfaces in nonequilibrium flows. A set of three configurations with growing level of complexity has been adopted for the evaluation of CFD methods. Only contributors who have shown ability to solve the first two entry testcases, namely a 3D cylinder configuration from DLR (blind testcase) and an axisymmetric double cone geometry from CUBRC, are allowed to participate to the third case, the FRESH Fx prototype proposed by the AFRL. On the side of VKI, the author was responsible for handling the simulations and the corresponding testcases will be presented in this section, together with the numerical results obtained with COOLFluiD and the validation with the available experimental data.

6.1.1 Double Cone (CUBRC)

The experimental studies have been conducted in the LENS I shock tunnel at CUBRC to obtain detailed surface and flow characteristics over an axisymmetric double cone configuration with semi angles of 25^o and 55^o and a base diameter of 10.3 inches, as shown in Fig. 6.1.

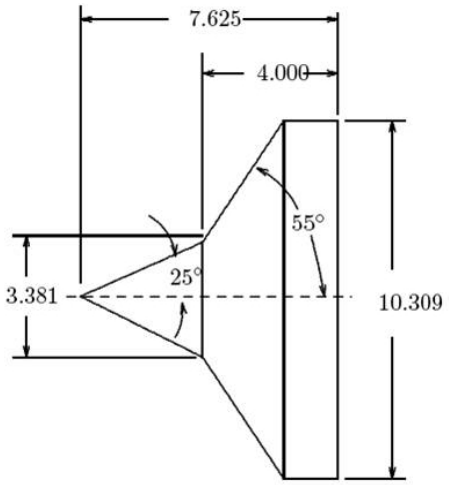


Figure 6.1: Double cone geometry (lengths are given in inches).

Measurements were made in both air and nitrogen for total enthalpy conditions of 5 and 10 MJ/kg, but only numerical simulations of nitrogen flows were required for the first contributors' selection. As extensively explained in [42, 105], this kind of simulations is extremely challenging from a computational point of view, mainly due to the complexity of the shock-shock and shock-boundary layer interactions and to the difficulty of the numerical schemes to approach convergence, because of the unsteadiness associated to the large separation region developing on the cone-cone juncture. Double cone flows are therefore ideal candidates for the purpose of CFD validation of aerothermodynamic solvers.
A sketch of a typical hypersonic flowfield around a double cone is depicted in Fig. 6.2 ([105]). The oblique shock generated by the first cone strongly interacts with the detached bow shock created by the second one and the resulting transmitted shock impinges on the surface downstream of the cone-

cone juncture. Here, the adverse pressure gradient forces the boundary layer to separate. The separated region on the corner generates its own shock, which interacts with the bow shock and causes a shift of the interaction point and this in turns alters the separation zone. This process feeds back on itself until the flow reaches a steady state, if such a state exists. Moreover, a supersonic jet forms along the surface of the second cone, downstream the impingement point, and it undergoes a series of compressions and expansions. An accurate prediction of the aerodynamic field and related quantities (heat flux) for hypersonic double cone flows requires thermo-chemical non equilibrium effects to be taken into account, as demonstrated for instance in [93, 107, 108]. To this end, all our simulations have accounted for those effects with the conventional 2-temperature model for thermo-chemical non equilibrium described in Sec. 4.3.

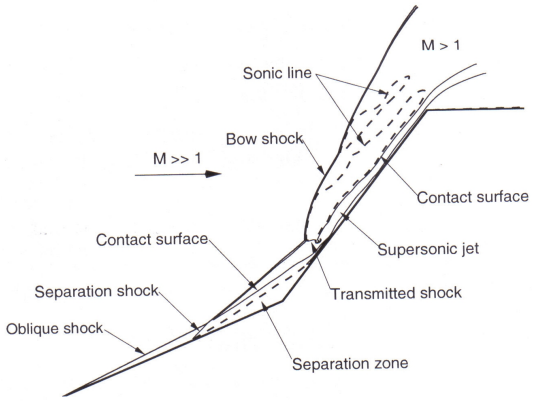


Figure 6.2: Schematic of double-cone flowfield at high enthalpy nitrogen free-stream conditions (run 42).

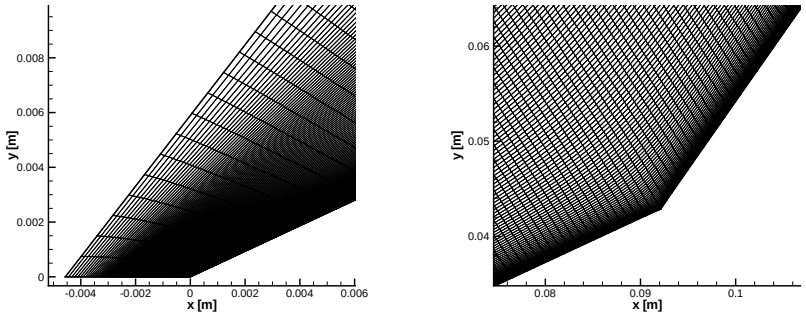
6.1.1.1 Double cone: run 35

This double cone configuration, whose conditions are listed in Table 6.1, was not part of the RTO task, but we report it here because it is a very well documented testcase in literature [42, 108] and has given us the opportunity to verify and validate both our solvers, FV and RD, with the basic non reactive Navier-Stokes model, before coping with the more challenging cases (run 40 and 42) in thermo-chemical nonequilibrium.

M_∞	$\rho_\infty[kg/m^3]$	$U_\infty[m/s]$	$T_\infty[K]$	$T_w[K]$
11.5	0.0005515	2713	138.9	296.1

Table 6.1: Free stream and wall conditions for double cone (run 35).

The computations have been run on a relatively coarse structured mesh with 256x128 cells kindly provided by Dr. Nompelis. Detailed views of the mesh around the two most critical points, namely the leading edge and the junction between the two cones are shown in Fig. 6.3a and 6.3b.



(a) Zoom around the leading edge.

(b) Zoom on the junction between the two cones.

Figure 6.3: Detailed views of the coarse structured mesh (256x128 cells) for the double cone.

The numerical solutions are compared among each other and against experiments in Figs. 6.4 and 6.5 in terms of surface pressure and heat flux. In particular, we show results with

- AUSM+up (Sec. 5.2.1.5), weighted MUSCL reconstruction (Sec. 5.2.2.1) and Van Leer limiter (Sec. 5.2.3.2);
- AUSM+, least square reconstruction (Sec. 5.2.2.2) and Venkatakhrisnan limiter (Sec. 5.2.3.1);
- Roe (Sec. 5.2.1.1), MUSCL extrapolation and Van Leer limiter;
- modified Steger Warming (Sec. 5.2.1.4), least square reconstruction and Venkatakhrisnan limiter.

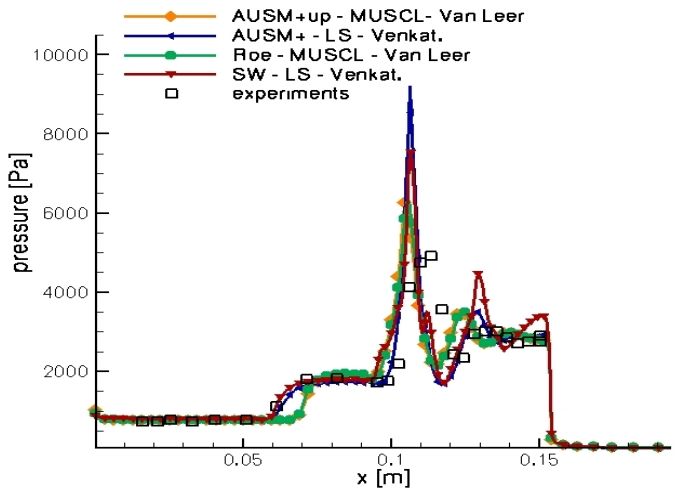


Figure 6.4: Surface pressure with FV schemes for double cone (run 35).

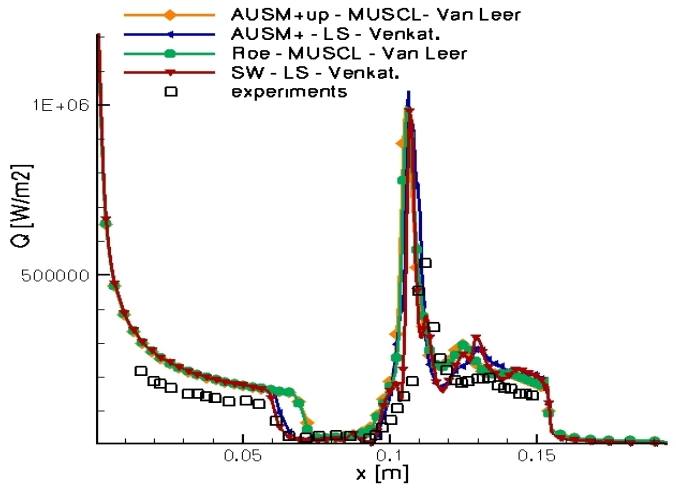


Figure 6.5: Surface heat flux with FV schemes for double cone (run 35).

In all the four schemes, the high order reconstruction has been performed in update variables $\mathbf{P} = [p, \mathbf{u}, T]$. In the case of both surface quantities, AUSM+ and modified Steger Warming in combination with Venkatakhrisnan limiter seem to give the results more in agreement with the experimental measurements. The merit for this can probably be attributed to the multidimensional character of the least square reconstruction and the Venkatakhrisnan limiter compared to the more uni-directional MUSCL approach. The results obtained by us agree qualitatively and quantitatively well with those shown in [42], where several schemes have been tested, but with a more accurate MUSCL formulation, involving 5 points in the face-based stencil.
The same simulation has been performed with the second order accurate Bx RD scheme (see Sec. 5.3.2.3). Since our RD Navier-Stokes solver works only on grids composed of triangles, the original structured mesh used for the previous computations has been split into triangles, while keeping approximately the same number of degrees of freedom. In fact, RD is a vertex centered method and the number of vertices is equal to the number of cell centers, except for boundary effects.
The temperature field corresponding to the RD solution is shown in Fig. 6.6, where a zoomed view with some overposed streamlines near the junction between the first and second cone is also presented, in order to better visualize the flow behaviour in the separation bubble. The convergence history in Fig. 6.7 corresponds to the first order RD computation (N scheme) starting from an initial uniform flowfield: it shows an example of almost quadratic convergence achieved by the implicit Newton method in such a complex testcase. The second order simulation with Bx was restarted from the fully converged first order solution, leading to a less optimal convergence history, with a maximum CFL equal to 25.
The surface pressure and heat flux calculated by the Bx scheme are shown in Figs. 6.8 and 6.9 and compared with the modified Steger Warming solution (i.e. the best result in Figs. 6.4 and 6.5) and the available experimental data. The superior shock capturing of the RD approach yields an impressive improvement in the numerical solution if compared to the standard FV method, in particular on the heat flux distribution on the first cone and on the detection of the bubble size, which almost perfectly match the experiments on this rather coarse mesh. This result alone indicates RD as a strong candidate for enhancing the accuracy of double cone flow simulations on unstructured grids.

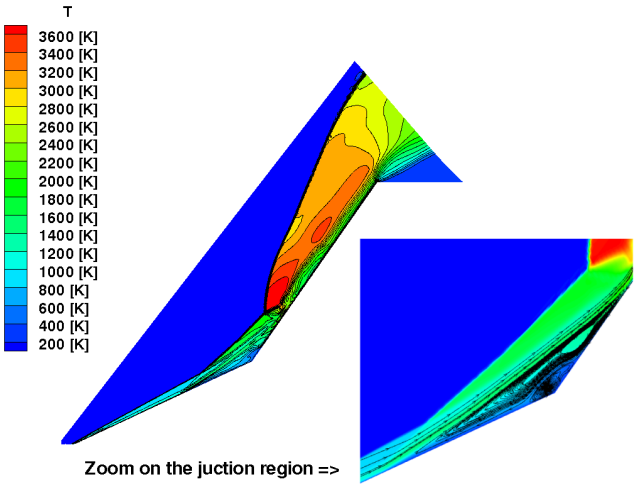


Figure 6.6: Temperature field given by Bx scheme in run 35 with a zoom on the separation region.

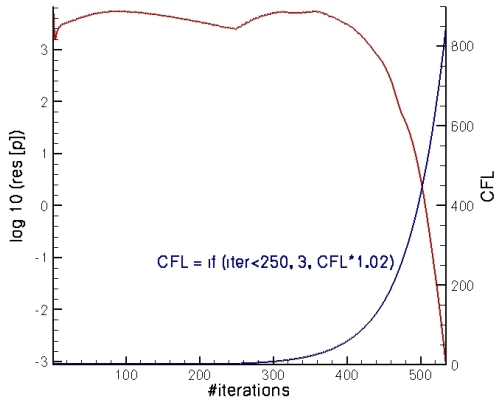


Figure 6.7: Convergence history in terms of pressure residual for run 35 with N scheme and user-defined CFL law.

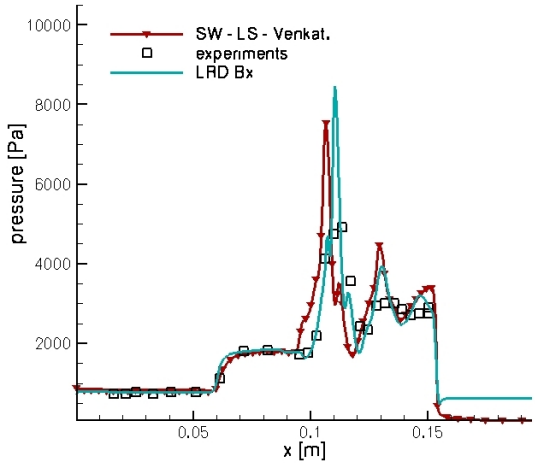


Figure 6.8: Surface pressure with Bx and modified Steger Warming for run 35.

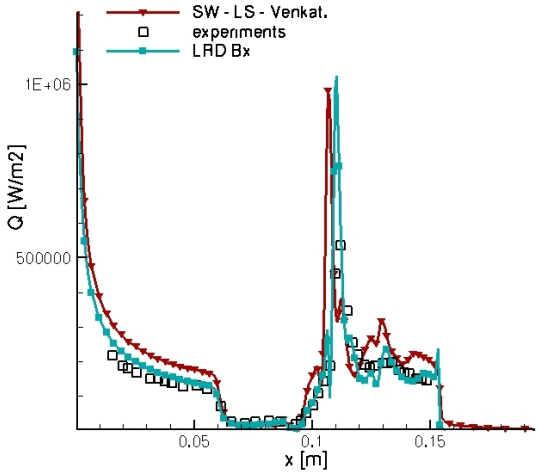


Figure 6.9: Surface heat flux with Bx and modified Steger Warming for run 35.

The overestimation of the heat flux on the first cone can be furtherly reduced, and get closer to the experiments, by including a frozen vibrational energy content in the free stream conditions and considering the effect of vibrational relaxation as demonstrated in [108] on the same specific testcase.

6.1.1.2 Double cone: run 40

The nominal conditions for this low enthalpy (5.38 MJ/kg) configuration are given in Table 6.2.

M_∞	$\rho_\infty[kg/m^3]$	$U_\infty[m/s]$	$T_\infty[K]$	$T_\infty^v[K]$	y_N	y_{N_2}	$T_w[K]$
11.6	0.00253	3104	172	2617	0	1	294

Table 6.2: Free stream conditions for double cone (run 40).

This testcase has a truly unsteady nature, at least numerically, and it doesn't reach steady state convergence. The solution which is presented hereafter depicts the situation on a certain moment during the computation when numerical results and experiments were in a reasonably good agreement. Our simulation has been performed with the AUSM+ scheme in combination with least square reconstruction and Venkatakhrisnan's limiter on the 512x1024 quadrilateral cells mesh provided by Dr. Nompelis and used in [42, 105]. The flow is modeled as a neutral nitrogen mixture (N_2-N) in thermo-chemical nonequilibrium with a 2 temperature model. The corresponding reaction rate coefficients are given in [105]. Mach number and roto-translational temperature contours and isolines are presented in Figs. 6.10 and 6.11. In this case, no chemical reactions occur and molecular nitrogen remains the only component present in the whole flow field. Figs. 6.12 and 6.13 show a good agreement between experimental and computed surface pressure and heat flux up to the corresponding peaks, but both physical quantities are overpredicted on the second cone past the impingement point. This effect is probably due to the least square reconstruction algorithm applied to the supersonic outlet boundary condition that, at the time in which the simulation was performed, was not preserving a zero gradient for the extrapolated variables, leading to some inconsistency in the flowfield upstream.

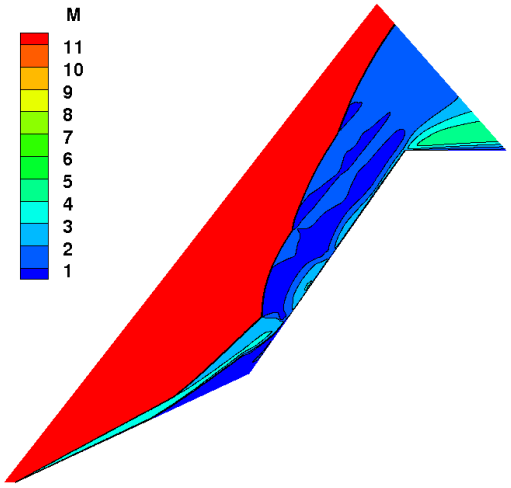


Figure 6.10: Mach number field for run 40 (FV AUSM+).

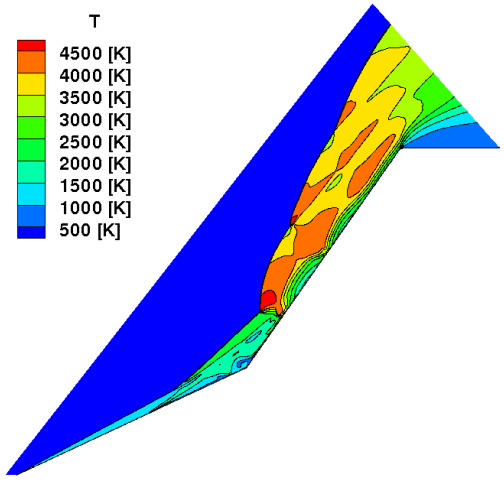


Figure 6.11: Roto-translational temperature for run 40 (FV AUSM+).

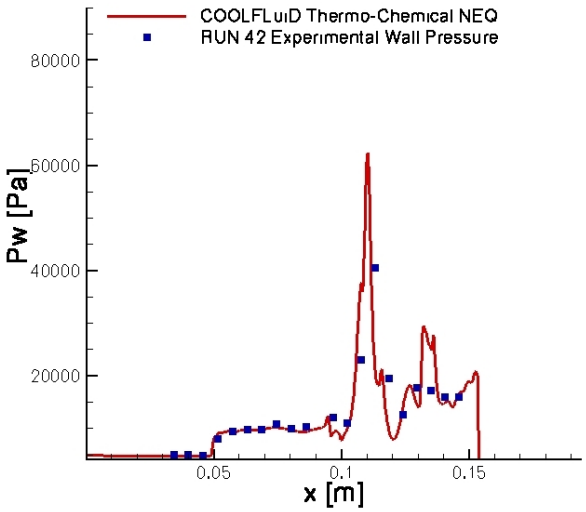


Figure 6.12: Surface pressure vs. experiments for run 40 (FV AUSM+).

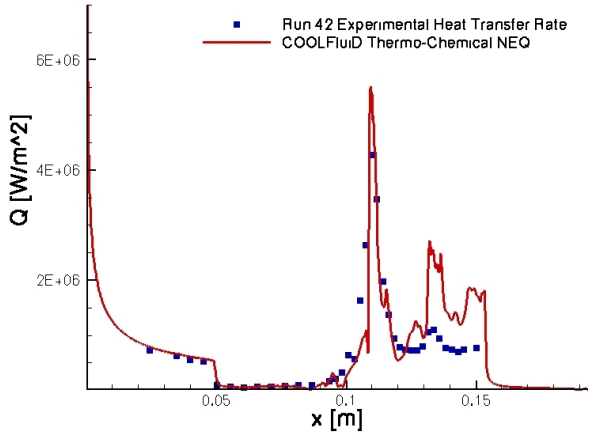


Figure 6.13: Surface heat transfer vs. experiments for run 40 (FV AUSM+).

6.1.1.3 Double cone: run 42

The nominal conditions for this high enthalpy (9.17 MJ/kg) configuration are listed in Table 6.3.

M_∞	$\rho_\infty[kg/m^3]$	$U_\infty[m/s]$	$T_\infty[K]$	$T_\infty^v[K]$	y_N	y_{N_2}	$T_w[K]$
11.5	0.001468	3849.3	268.7	3160	0	1	294.7

Table 6.3: Free stream conditions for double cone (run 42).

The results here shown for this testcase have been run with the CRD solver for the 2-temperature model and nitrogen mixture as described in run 40. In particular, the second order accurate blended Bxc scheme has been employed. Since this novel method works, for the moment, only on simplex cells (triangles in 2D and tetrahedra in 3D), Nompelis's intermediate mesh (256x512 in this case) has been split into triangles, as in run 35.
From the Mach number and roto-translational temperature contours/isolines presented in Figs. 6.14 and 6.15 we can see the excellent shock capturing properties of the CRD method. The vibrational temperature of molecular nitrogen is shown in Fig. 6.16, clearly indicating a non negligeable presence of thermal nonequilibrium in the flow, particularly in the boundary layer and downstream the bow shock, where the roto-translational temperature is higher. Unlike the previous case, i.e. run 40, a moderate dissociation of molecular nitrogen into atoms occurs, as highlighted from the contours/isolines of atomic nitrogen mass fractions in Fig. 6.17. Because of the effect of the chemistry-vibration-chemistry coupling, the region affected by chemical dissociation has also the highest vibrational temperature.
Figs. 6.18a and 6.19a show the comparison between experimental and computed heat flux and surface pressure. The CRD solver yields a resolution competitive with the one given by Nompelis's finite volume solver, but requiring only one quarter of the degrees of freedom. This emerges clearly from comparing Figs. 6.18a and 6.18b for the pressure surface, Figs. 6.19a and 6.19b for the heat flux, where the reference results are taken from [105] and they were computed on a finer 512x1024 cells mesh.

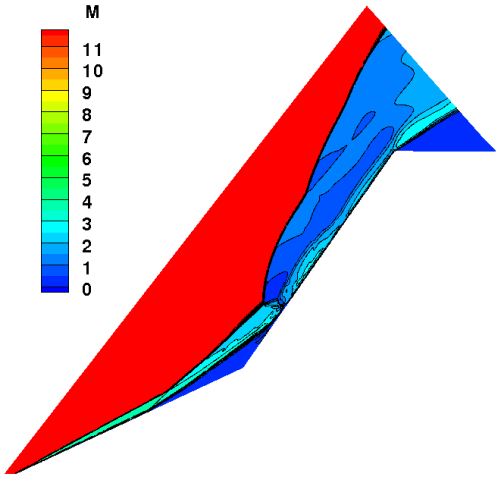


Figure 6.14: Mach number field for run 42 (CRD Bxc).

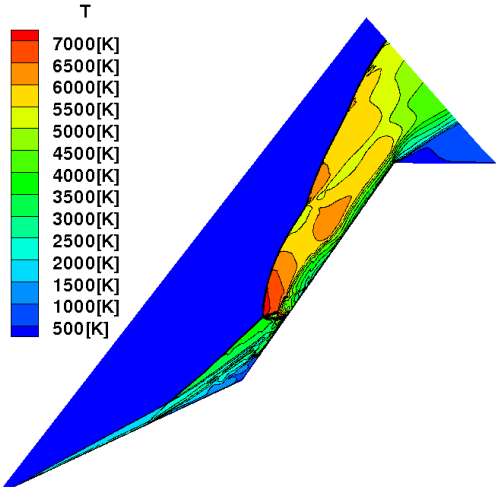


Figure 6.15: Roto-translational temperature for run 42 (CRD Bxc).

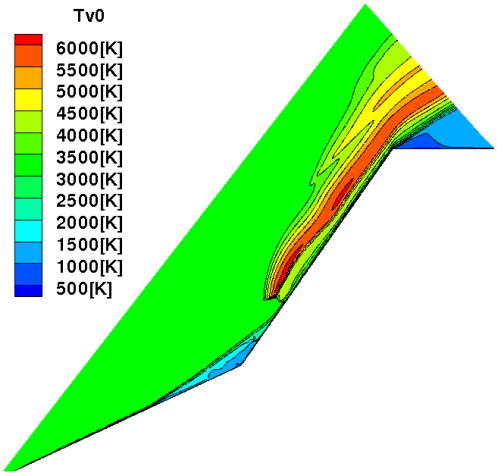


Figure 6.16: Vibrational temperature $T^v_{N_2}$ for run 42 (CRD Bxc).

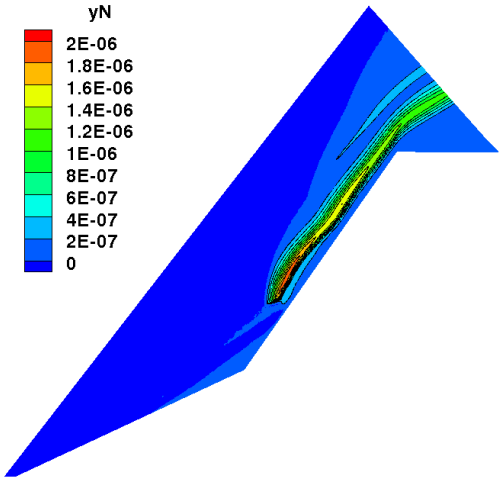
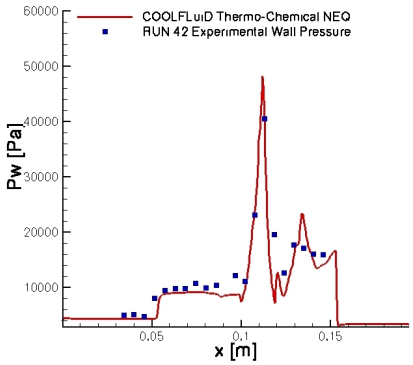
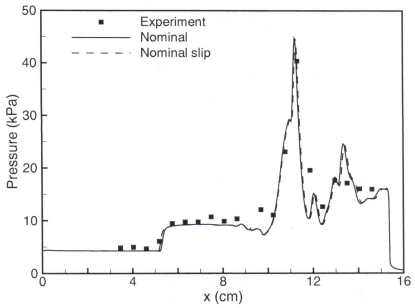


Figure 6.17: Atomic N mass fraction for run 42 (CRD Bxc).

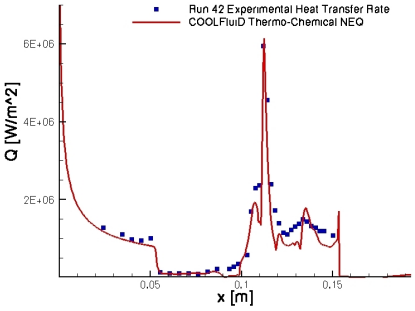


(a) Surface pressure given by Bxc scheme on 256x512 DOF mesh.

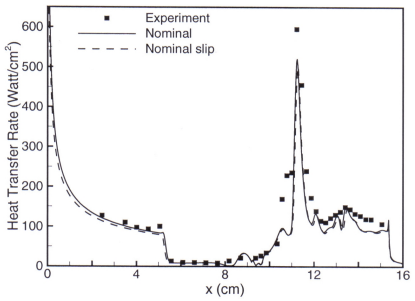


(b) Surface pressure given by [105] on 512x1024 DOF mesh.

Figure 6.18: Computed surface pressure vs. experimental measurements for run 42 (CRD Bxc).



(a) Surface heat flux given by Bxc scheme on 256x512 DOF mesh.



(b) Surface heat flux given by [105] on 512x1024 DOF mesh.

Figure 6.19: Computed surface heat flux vs. experimental measurements for run 42 (CRD Bxc).

6.1.2 Cylinder 3D (DLR)

This experiment was performed in HEG wind tunnel at DLR (Germany). The configuration is a circular cylinder with diameter 90 mm and a length (including shaped holder) of 380 mm, placed with its axis transverse to the

flow. The measurements on the cylinder have been carried out at two different total enthalpies (HEG conditions 1 and 2, 22.4 MJ/kg and 13.5 MJ/kg, respectively) and with air as a test gas. At those experimental conditions, the flow in the shock layer is subject to nonequilibrium chemical relaxation phenomena, leading to a modification of the gas properties (temperature, density, ...), which in turn causes changes in shock properties, such as shape and stand-off distance. A preliminary study on a similar configuration but with different free-stream flow conditions is reported in [106].
A blind numerical study on the two selected configurations have been conducted within the RTO contest, so that experimental measurements have been made available to participants only after the delivery of the computational results.
The partially unstructured mesh for this case has been generated with ANSYS Gambit with the precious help of Janos Molnar. It is composed by 3426300 hexaedral cells: a global view of the surface mesh (except for the inlet boundary that has been blanked for visualization purposes) and a zoom near the tip of the cylinder can be seen in Figs. 6.20 and 6.21a. In particular, the sharp corner of the cylinder has been rounded up in order to avoid singularities in the solution, as shown in Fig. 6.21b, while the shaped holder has not been included in the model.
The implicit cell centered finite volume solver with AUSM+ flux, weighted least square reconstruction and Venkatakrishnan limiter has been used for computing both the flow conditions. A 3-temperature model (including roto-translational T, $T^v_{N_2}$, $T^v_{O_2}$) for thermo-chemical nonequilibrium has been applied to this case. The flow is modeled as a 5-species air mixture with reaction rate coefficients given by [120].

6.1.2.1 Cylinder HEG : condition 1

The HEG reservoir and free stream data for the measurements are listed in Table 6.4. The free stream Mach number, which is not specified in the table, is $M_\infty = 8.8$. Furthermore, the free stream vibrational temperatures $T^v_{N_2}$ and $T^v_{O_2}$ have been assumed in equilibrium with the roto-translational temperature, while $T_w = 300\ K$ is imposed as wall temperature.

$\rho_\infty[kg/m^3]$	$U_\infty[m/s]$	$T_\infty[K]$	y_N	y_O	y_{N_2}	y_{NO}	y_{O_2}
0.001547	5956	901	0.65e-6	0.2283	0.7543	0.01026	0.00713

Table 6.4: Free stream conditions for HEG cylinder (condition 1).

Figures 6.22a and 6.22b show respectively the isolines and contours of Mach

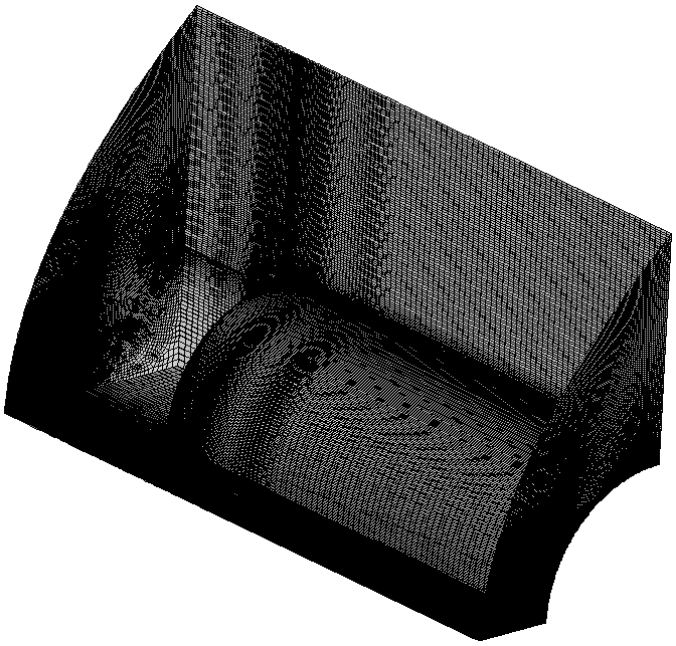
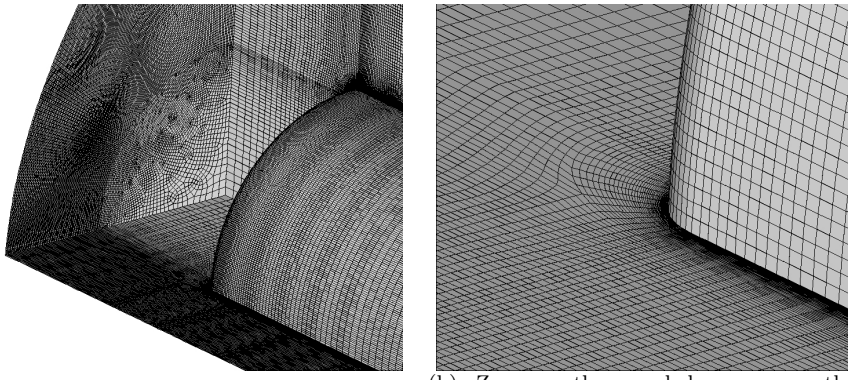


Figure 6.20: Global view of the unstructured surface mesh.

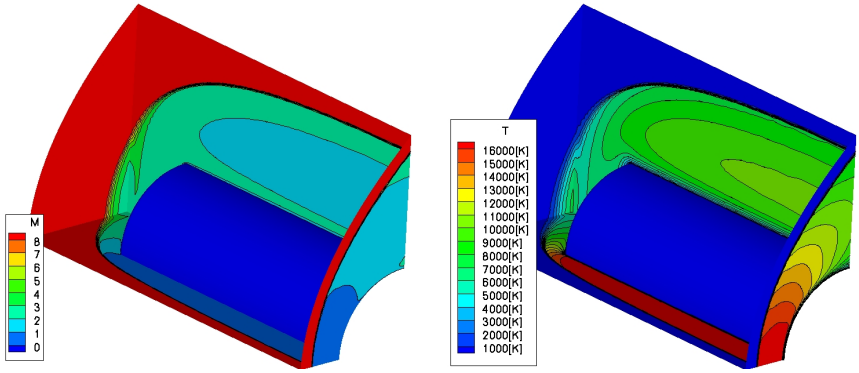
number and roto-translational temperature. In the latter case, a maximum temperature of 17000 K is reached.
The contours and isolines of the mass fractions of atomic species, oxygen and nitrogen, are shown in 6.23a and 6.23b, while the mass fractions of the corresponding molecules are presented in 6.24a and 6.24b: it can be easily seen that the overall composition of the mixture remains frozen to the free stream values, except for some unperceiveable variations clearly due to some small numerical inaccuracy. The latter is especially located near the corner between the far field and symmetry plane, where the shock hits the side boundary: a slightly bigger domain or just a finer mesh in the region past the cylinder would probably have avoided this trouble.
Computed and measured surface pressure and heat flux versus angle (0^o corresponds to the stagnation point) are shown in Figs. 6.25a and 6.25b in three different sections in the spanwise direction ($z/L = 0$, $z/L = 0.5$, $z/L = 0.95$, the latter being close to the tip of the cylinder). On the one hand, the agreement with the experiments is remarkably good for the pres-



(a) Zoom near the tip of the cylinder

(b) Zoom on the rounded corner near the tip

Figure 6.21: Detailed views on the cylinder surface mesh.

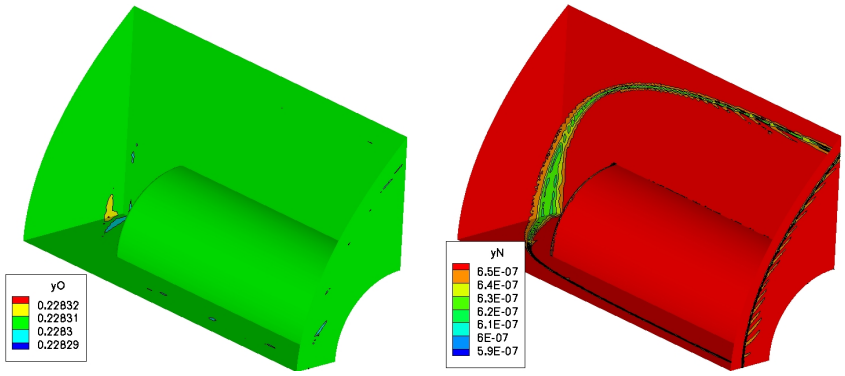


(a) Mach number

(b) Roto-translational temperature

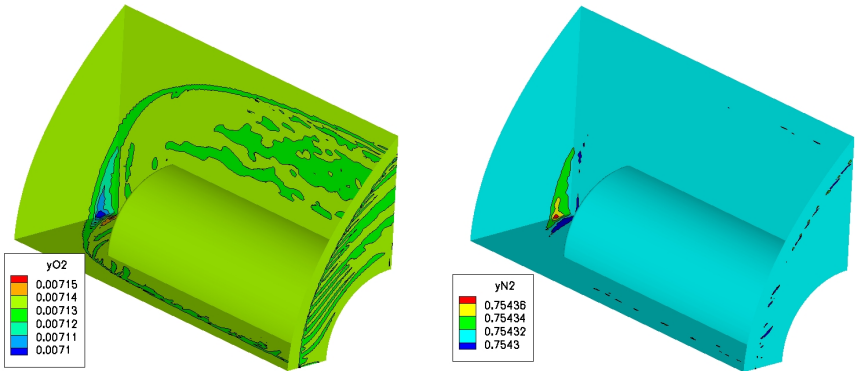
Figure 6.22: Contours/isolines of Mach number and roto-translational temperature for the HEG cylinder (condition 1) with FV AUSM+.

sure, whose peak reaches about 50 kPa. On the other hand, the peak heat flux on the symmetry plane, 7 MW/m^2, is underestimated by roughly 12% which is considered acceptable also in view of the experimental uncertainties.
The matching with the experiments could be probably improved by assum-



(a) Mass fractions of atomic oxygen (b) Mass fractions of atomic nitrogen

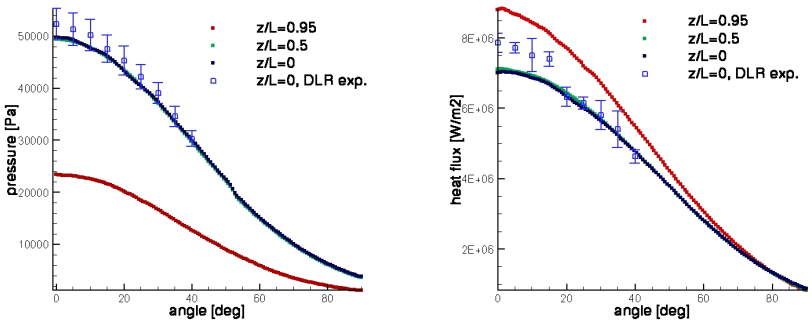
Figure 6.23: Contours/isolines of atomic O and N mass fractions for the HEG cylinder (condition 1) with FV AUSM+, showing the chemically frozen flowfield.



(a) Mass fractions of molecular oxygen (b) Mass fractions of molecular nitrogen

Figure 6.24: Contours and isolines of O_2 and N_2 mass fractions for the HEG cylinder (condition 1) with FV AUSM+, showing the chemically frozen flowfield.

ing some catalytic activity at the wall, as shown in [93] for a number of 2D cases in similar conditions.



(a) Surface pressure

(b) Surface heat flux

Figure 6.25: Computed surface pressure and heat flux vs. experimental measurements for the cylinder (condition 1).

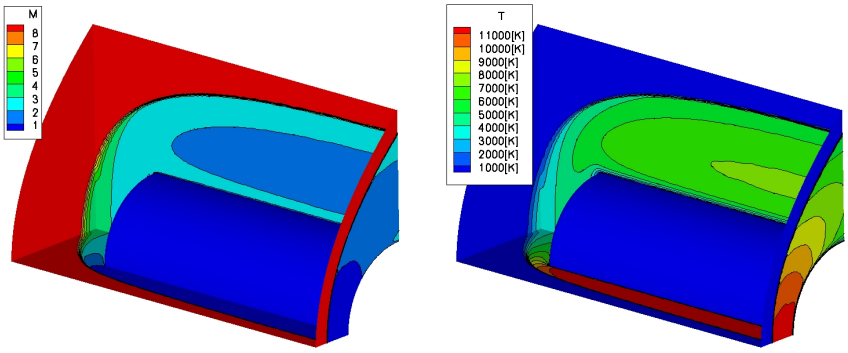
6.1.2.2 Cylinder HEG : condition 2

The free stream conditions are presented in Table 6.5. The free stream mach number is $M_\infty = 8.7$, while, also in this case, the free stream vibrational temperatures $T^v_{N_2}$ and $T^v_{O_2}$ have been considered in equilibrium with the roto-translational temperature. The wall temperature is again $T_w = 300\ K$ as in the previous case.

$\rho_\infty[kg/m^3]$	$U_\infty[m/s]$	$T_\infty[K]$	y_N	y_O	y_{N_2}	y_{NO}	y_{O_2}
0.00326	4776	694	0	0.07955	0.7356	0.0509	0.1340

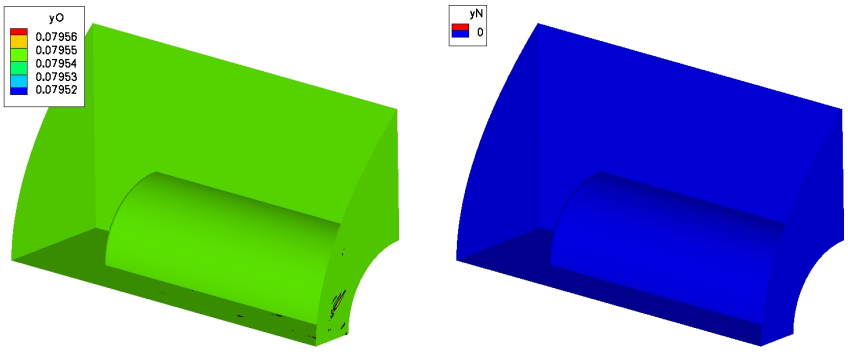
Table 6.5: Free stream conditions for the HEG cylinder (condition 2).

Figs. 6.26a and 6.26b show respectively the isolines and contours of Mach number and roto-translational temperature. In the latter case, a maximum temperature of 11500 K is reached.



(a) Mach number. (b) Roto-translational temperature.

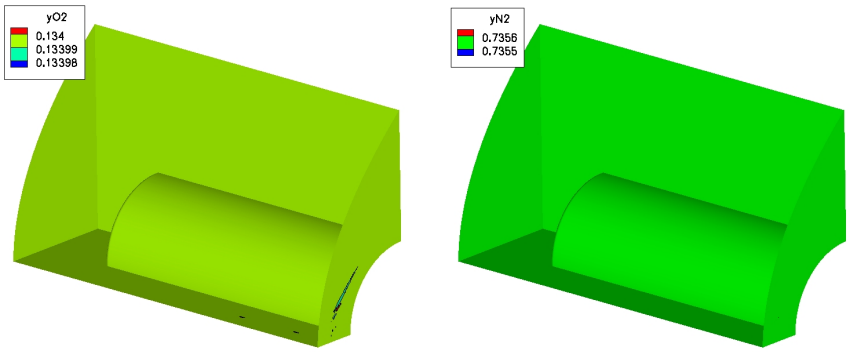
Figure 6.26: Contours/isolines of Mach number and roto-translational temperature for the HEG cylinder (condition 2) with FV AUSM+.



(a) Mass fractions of atomic oxygen. (b) Mass fractions of atomic nitrogen.

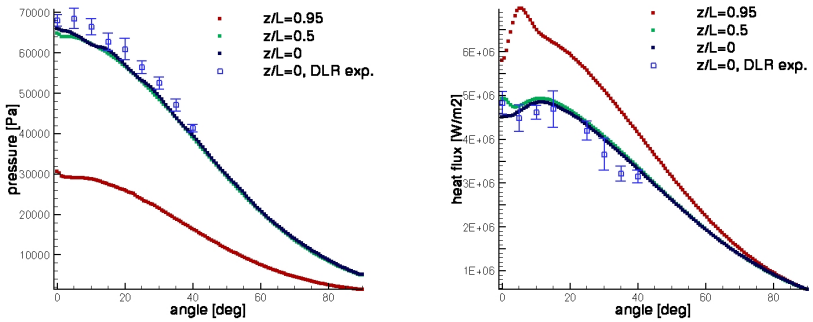
Figure 6.27: Contours and isolines of mass fractions of atomic O and N for the HEG cylinder (condition 2) with FV AUSM+, showing a perfectly chemically frozen flowfield.

The contours and isolines of the mass fractions of atomic species, oxygen and nitrogen, are shown in 6.27a and 6.27b, while the mass fractions of the corresponding molecules are presented in 6.28a and 6.28b. Also in this case,



(a) Mass fractions of molecular oxygen. (b) Mass fractions of molecular nitrogen.

Figure 6.28: Contours and isolines of mass fractions of O_2 and N_2 for the HEG cylinder (condition 2) with FV AUSM+, showing a perfectly chemically frozen flowfield.



(a) Surface pressure. (b) Surface heat flux.

Figure 6.29: Computed surface pressure and heat flux vs. experimental measurements for the HEG cylinder (condition 2) with FV AUSM+.

the overall composition of the mixture remains frozen to the free stream values. In particular, the mass fraction of atomic nitrogen remains identically zero. The solution is smooth even near the corner between the far field and

symmetry plane, which was causing some small trouble in Condition 1. Computed and measured surface pressure and heat flux versus angle are shown in Figs. 6.29a and 6.29b in the same three sections in the spanwise direction as for the previous condition. The maximum pressure is about 64 kPa, while the peak heat flux reaches 5.0 MW/m^2 on the symmetry plane and 6.4 MW/m^2 near the tip. The agreement with experiments is remarkably good for both the surface quantities.

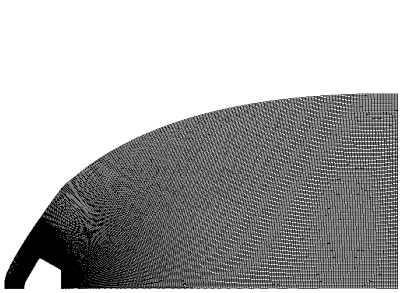
6.2 Stardust Sample Return Capsule

The Stardust Sample Return Capsule (SRC) was launched in February 1999 with the aim of retrieving samples of interstellar dust from the tail of the WILD-2 comet. The capsule reentered Earth atmosphere in January 2006 at a speed of 12.6 km/s, which corresponds to the fastest and most energetic reentry of any artificial vehicle to date. The Stardust observation data recorded by several optical instruments aboard the NASA DC-8 Airborne observatory offers a precious opportunity to compare simulation to flight data at hypervelocity [25, 92, 153]. Figure 6.30a shows the Stardust capsule as seen from NASA's DC-8 Airborne Laboratory aiming at exploring the conditions during reentry from the light emitted by the fireball caused when the capsule streaked through the sky. The aircraft was located near the end of the trajectory. The participating researchers were from NASA Ames, the SETI Institute, the University of Alaska, Utah State University, Lockheed Martin, U.S. Air Force Academy, the University of Kobe (Japan) and Stuttgart University. A closer look at the capsule after its landing is given in Fig. 6.30b.
In this section we present the results of the simulation of a few points points belonging to the flight trajectory of Stardust, corresponding to 34s, 51s and 66s after the reentry in Earth atmosphere. Since the entry of the capsule was nearly ballistic [153], an axisymmetric simulation offers a quite good approximation of the reality and has allowed us to save a lot of computational cost compared to a fully 3D configuration. All our computations have been performed with the AUSM+ scheme, which is exceptionally robust and delivers carbuncle free solutions even at these very high velocities ($>$ 12 km/s at 34s), at least on meshes with cells sufficiently stretched in the direction tangential to the shock. Second order accuracy has been achieved by means of the least square reconstruction in combination with the Venkatakrishnan limiter. An example of mesh used for our calculations and a zoomed view around the capsule can be found respectively in Figs. 6.31 and 6.31b.

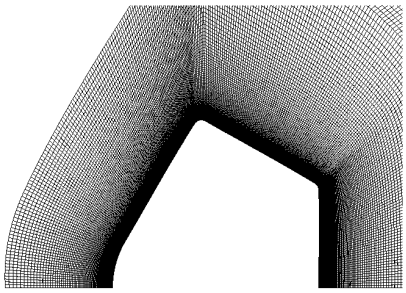
(a) Stardust capsule streaking through the sky as seen from NASA's DC-8 Airborne Laboratory on January 15th, 2006

(b) Stardust capsule after the landing

Figure 6.30: Views of the Stardust capsule during the reentry phase and after landing.



(a) Global mesh view.



(b) Zoom around the capsule.

Figure 6.31: Computational mesh (68300 cells) for the Stardust post flight analysis.

As far as the physical model is concerned, the 2-temperature Park model for ionized mixtures has been adopted, together with the reaction rates for the chemical nonequilibrium given by [120] and a radiative and local thermochemical equilibrium condition have been imposed at the wall. The numerical results given by our baseline solver (without any specific adjustment of chemical rates or thermodynamic properties) will be compared with those obtained with the NASA DPLR code in [25, 92, 153]. The differences be-

tween DPLR and our aerothermodynamic solver are considerable from both the numerical and physical modelling point of view. In particular, DPLR uses a modified Steger-Warming scheme similar to the one presented in Sec. 5.2.1.4, a third order accurate MUSCL reconstruction, a 3-temperature model where the rotational temperature is separated from the translational one, transport properties based on mixing rules and different chemical reaction rates. Moreover, while we assume LTE at the wall, DPLR simulations have been run with full catalicity only for ions.

6.2.1 81.02 Km, t=34 s, Max entry speed, $M_\infty = 41.9$

At this point of the reentry trajectory the capsule reached its maximum speed, which also corresponds to the maximum velocity ever experienced by an artificial object flying within Earth atmosphere. The corresponding flow conditions are given in Table 6.6. It is estimated that at this point the transition between continuum and non-continuum regimes occurs, with $Kn \approx 0.005$ based on the diameter of the capsule.

M_∞	$\rho_\infty[kg/m^3]$	$U_\infty[m/s]$	$T_\infty[K]$	$p_\infty[Pa]$
41.9	0.00001269	12385.12016	217.63406	0.79216

Table 6.6: Free stream conditions for Stardust at t = 34s.

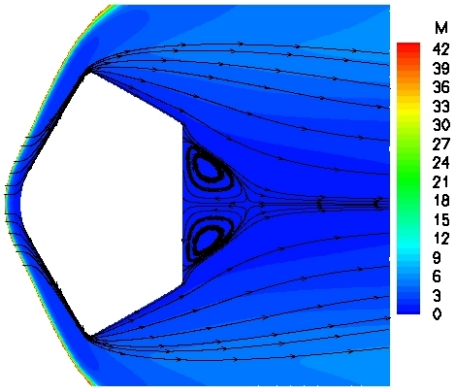


Figure 6.32: Mach number and streamlines around the capsule at t=34s with FV AUSM+.

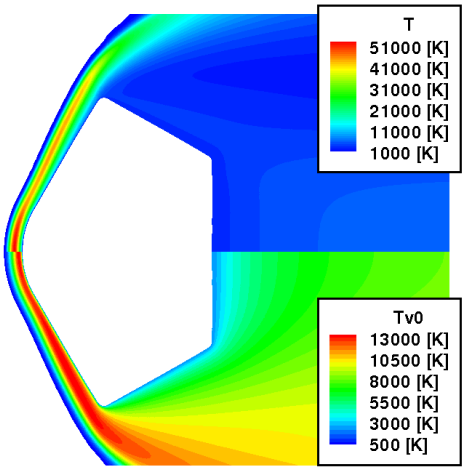


Figure 6.33: Roto-translational and vibrational temperatures at t=34s with FV AUSM+.

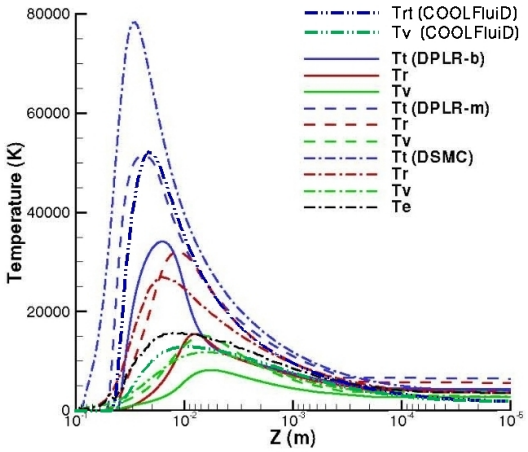


Figure 6.34: Temperatures profiles on the stagnation line at t=34s computed with COOLFluiD and the solvers in [25].

The Mach number field around the capsule is shown in Fig. 6.32, where the superposed streamlines indicate the location of the recirculation bubble. The roto-translational and vibrational temperature contours corresponding to this trajectory point are presented in Fig. 6.33. In particular, the peak post-shock roto-translation temperature reaches 53000 K, while the maximum vibrational temperature is 13400 K. The temperature profiles on the stagnation line are shown in Fig. 6.34 together with the solutions computed in [25] with the Direct Simulation Monte Carlo (DSMC) method and the DPLR code. For the latter case, two solutions with different physico-chemical models, i.e. baseline (b) and modified (m), are given. Our solution is close to the (m) case in terms of roto-translational temperature and it predicts a vibro-electronic temperature that falls right in between the DSMC results for the electronic and vibrational temperature.
The stagnation line profiles for the neutral species number densities in the air mixture are plotted in Fig. 6.35 (COOLFluiD) and 6.36 ([25]). Similarly, the number densities of all ionic components and electrons obtained with COOLFluiD are shown in Fig. 6.37, while only the number densities of N^+ and N_2^+ are shown in 6.38, because other data were not available in [25] for comparison. If we compare the behaviour of the ionic and electron species close to the wall (a logarithmic scale has been adopted to simplify this analysis), our results differ considerably with those of [25]. This is partially justified by the fact that we apply a LTE model at the wall, while [25] imposes full catalycity to ions (that recombine into their neutral atoms and molecules) and atoms (that recombine into molecules).
Generally speaking, the many differences in the thermo-chemical modeling, in the numerical scheme and in the mesh resolution between the DPLR simulations and ours can reasonably justify the partial disagreement between the two solutions.

6.2.2 61.76 Km, t=51 s, Peak heating point, $M_\infty = 35.3$

This trajectory point corresponds to the maximum thermal load, achieved at $M_\infty = 35$. The flow free stream conditions are listed in Table 6.7.

M_∞	$\rho_\infty[kg/m^3]$	$U_\infty[m/s]$	$T_\infty[K]$	$p_\infty[Pa]$
35.3	0.00021100	10871.38098	234.95243	14.21781

Table 6.7: Free stream conditions for Stardust at t = 51s.

The flowfield around the capsule is shown in Fig. 6.39a in terms of Mach number contours and velocity streamlines. COOLFluiD's solution is in per-

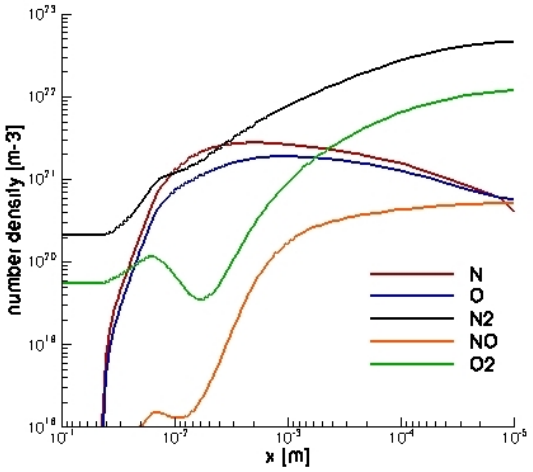


Figure 6.35: Stagnation line profiles for the number densities of the neutral species at t=34s with COOLFluiD (FV AUSM+).

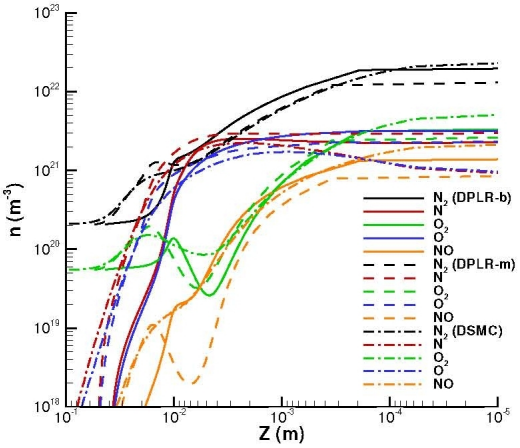


Figure 6.36: Stagnation line profiles for the number densities of the neutral species at t=34s in [25].

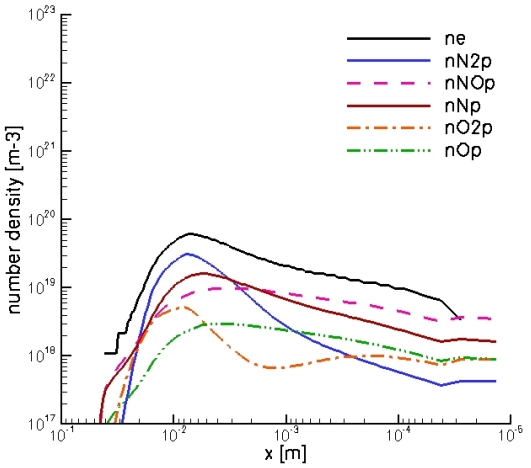


Figure 6.37: Stagnation line profiles for all number densities of the ionic and electron species at t=34s with COOLFluiD (FV AUSM+).

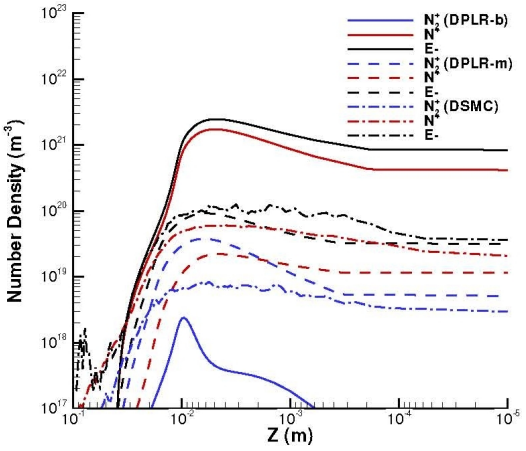
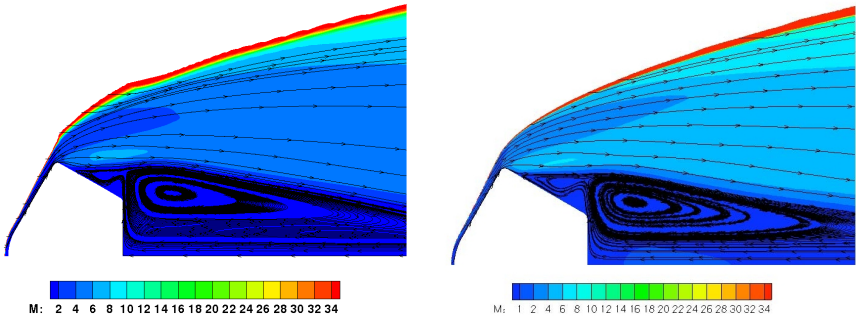


Figure 6.38: Stagnation line profiles for the number densities of N^+ and N_2^+ and electrons at t=34s in [25].

fect agreement with the one in Fig. 6.39b, yielded by the DPLR code. Herein, the length of the capsule for our simulation have been taken from [25] (0.5 m) and is slightly different from the size adopted in [153] (0.56 m), but this have no relevant effect in our code to code comparison. In particular, the three recirculation bubbles that can be identified in the after-body are located in the same positions (relatively to the capsule) and have very similar sizes.
Fig. 6.40 shows the surface heat fluxes calculated by the two solvers and the agreement between the two solutions is again pretty impressive, especially on the predicted peak value. The only remarkable difference consists in a little overshoot due to thickening of the boundary layer on the shoulder which is not properly resolved by COOLFluiD, probably because the mesh was not sufficiently fine in that region.



(a) COOLFluiD solution (FV AUSM+) (b) Solution given by NASA DPLR in [153]

Figure 6.39: Mach number and velocity streamlines at t=51s.

6.2.3 50.98 Km, t=66 s, $M_\infty = 20.3$

The conditions for this trajectory point are shown in Table 6.8.

M_∞	$\rho_\infty[kg/m^3]$	$U_\infty[m/s]$	$T_\infty[K]$	$p_\infty[Pa]$
20.3	0.00085435	6504.45316	255.83878	62.78974

Table 6.8: Free stream conditions for Stardust at t = 66s.

The surface heat fluxes computed with COOLFluiD and with DPLR in [153] are presented in Figs. 6.41a and 6.41b (dashed line), showing a good agreement. In the latter picture, an additional curve corresponding to the heat

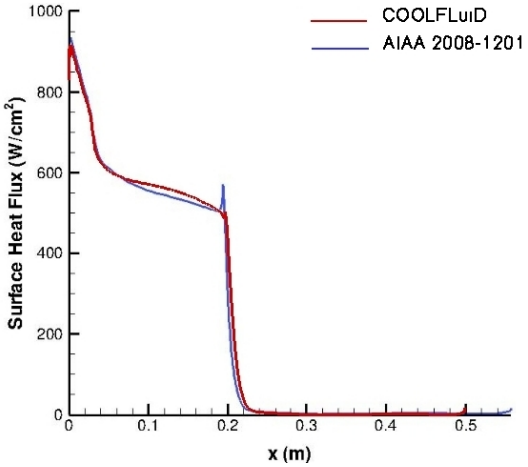
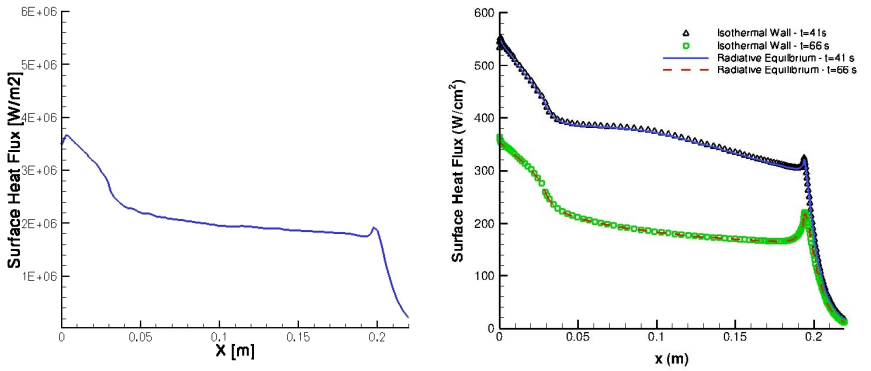


Figure 6.40: Surface heat flux profile for the Stardust capsule at t=51s.

flux at t=41s is plotted which is of no interest here. For this case, the little overshoot on the shoulder of the capsule is predicted also by COOLFluiD.

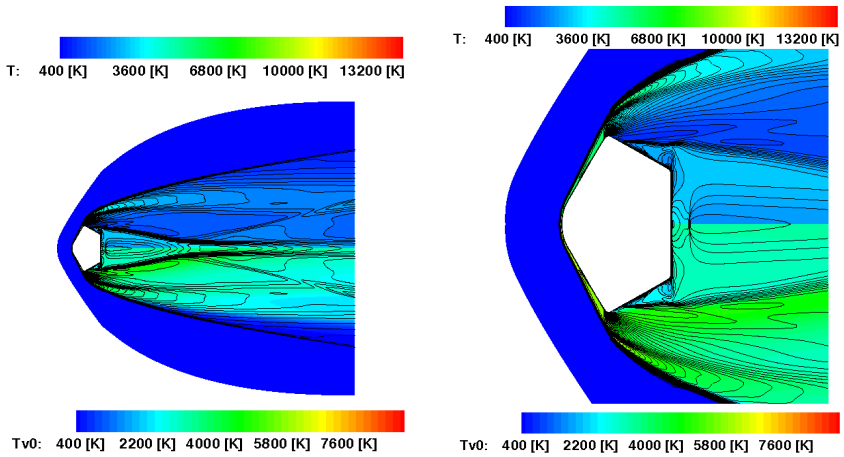


(a) Solution given by COOLFluiD (b) Solution given by NASA DPLR in [153]

Figure 6.41: Surface heat flux profile for the Stardust capsule at t=66s.

A full and a zoomed view of the flowfield surrounding the capsule in terms of roto-translational and vibro-electronic temperature is shown in 6.42. In

this case, in the post-shock region on the stagnation line the peak roto-translational and vibro-electronic temperatures reach 14150 K and 9200 K respectively.



(a) Global view (b) Zoomed view around the capsule

Figure 6.42: Contour and isolines of roto-translational and vibrational temperatures for the Stardust capsule at t=66s.

6.3 EXPERT Vehicle

The European re-entry vehicle EXPERT (EXPERimental Re-entry Test-bed) [103] is part of the ESA aerothermodynamic research program aiming to design a low cost hypersonic flight vehicle for measurement of aerothermodynamic phenomena and thus also validating the numerical codes. The goal of EXPERT is to achieve low cost re-entry experiments focused on flying "problems for design"; that is, complex fluid dynamic phenomena such as boundary layer/ shear layer transition, viscous interactions, flow separation and re-attachment, shock-boundary layer interactions, surface catalysis, blackout and plume-flowfield interactions.
The results to be presented shortly concern the investigation of two points on the trajectory followed by the EXPERT vehicle during the reentry phase.
The first one is the point of maximum convective heating (Condition 1),

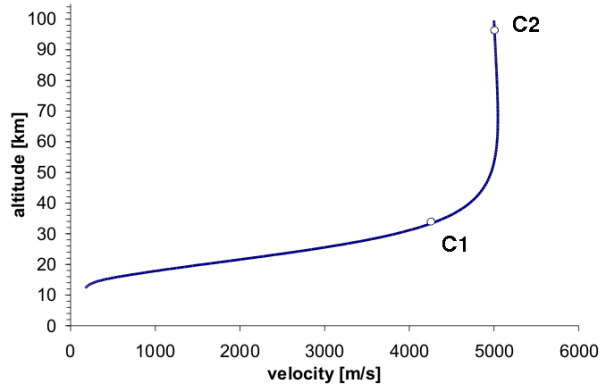


Figure 6.43: Points corresponding to conditions 1 and 2 on the reentry trajectory of the EXPERT vehicle (as foreseen in June 2007).

whereas the second trajectory point has been chosen in order to investigate flow conditions characterized by a stronger thermal and chemical nonequilibrium (Condition 2).
As we shall see in the following, the condition of maximum heating is characterized by relatively low speed and high pressure, reducing in this way the extent of thermal and chemical non equilibrium. The location of the two points within the reentry trajectory (as foreseen till June 2007) are highlighted in Fig. 6.43, while the corresponding flow conditions are shown in Table 6.9.

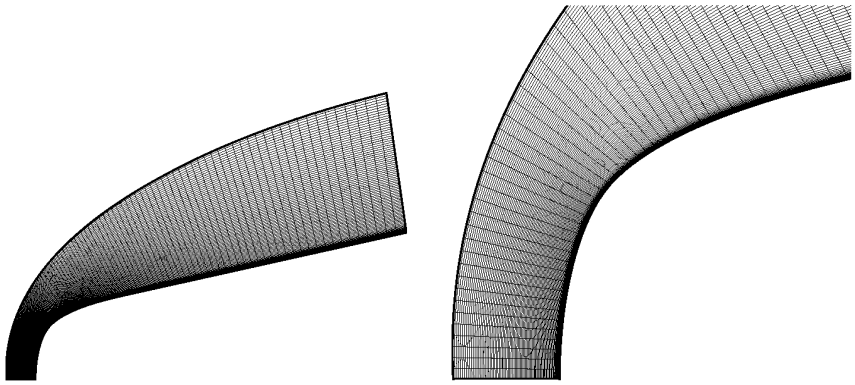
	M_∞	$p_\infty[Pa]$	$U_\infty[m/s]$	$T_\infty[K]$	$T_w[K]$
condition 1	14	668.1	4286.5	233.61	1000
condition 2	18.4	2	5040	186.4	1650

Table 6.9: Free stream conditions for the EXPERT simulations.

6.3.1 Condition 1, $M_\infty = 14$

This condition, corresponding to a low altitude trajectory point characterized by relatively high free stream pressure and temperature, has been selected in order to test different physical models presented in Chapter 4. In particular, the results given by the two LTE models with fixed (LTE-

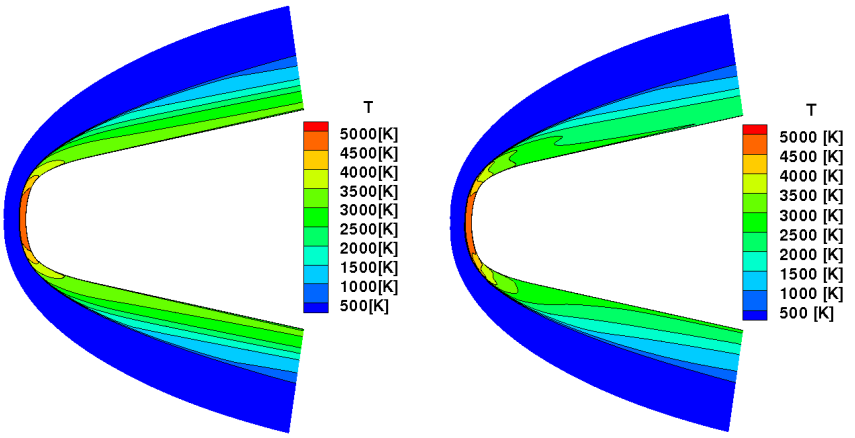
FEF) and variable elemental fractions (LTE-VEF) are compared against each other and against the solution obtained in Chemical Nonequilibrium (CNEQ) and TCNEQ simulations. Hereby, LTE and CNEQ simulations have been run with the AUSM+ scheme, while the Roe scheme described in Sec.5.2.1.1 has been used for TCNEQ, in combination with the carbuncle fix in [144]. All computations have been performed on a 2D axisymmetric structured mesh with 12600 cells, which is shown in Fig. 6.44a and Fig. 6.44b. The temperature flowfields yielded by the four different models are shown in Fig. 6.45a for the chemical equilibrium case (LTE-FEF and LTE-FEF) and in Fig. 6.45b for the nonequilibrium cases (CNEQ and TCNEQ). More detailed views of the flowfield around the vehicle nose are presented in Figs. 6.46a and 6.46b. While the two LTE models give no visible difference on the temperature, the overall flowfield appears a bit different in the two nonequilibrium cases.



(a) Global view. (b) Zoom around the vehicle.

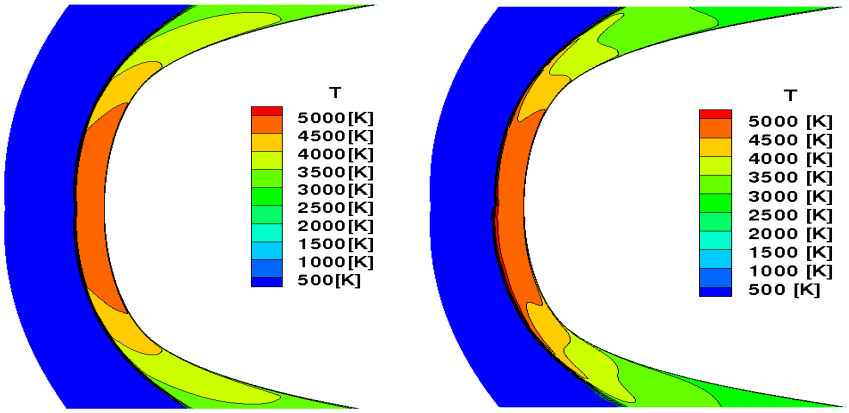
Figure 6.44: Computational mesh (12600 cells) for the EXPERT vehicle.

In Fig. 6.47 we compare the temperature profiles on the stagnation lines for the four models. The shock stand-off distance is slightly bigger in the TCNEQ case, but this is expected, because of the slightly higher post-shock temperature. As far as the LTE models are concerned, they yield different results inside the boundary layer, because of the demixing phenomenon. To better understand the latter, Fig. 6.48 shows temperature and species mass fractions profiles on the stagnation line only for LTE-FEF and LTE-VEF. A logarithmic scale is used on the x axis, in order to better visualize the



(a) LTE-FEF (top) vs. LTE-VEF (bottom) (b) CNEQ (top) vs. TCNEQ (bottom)

Figure 6.45: Contour/isolines of temperature for the EXPERT vehicle in condition 1 ($M_\infty = 14$) with different models.



(a) LTE-FEF (top) vs. LTE-VEF (bottom) (b) CNEQ (top) vs. TCNEQ (bottom)

Figure 6.46: Temperature field around the nose of the EXPERT vehicle in condition 1 ($M_\infty = 14$) with different models.

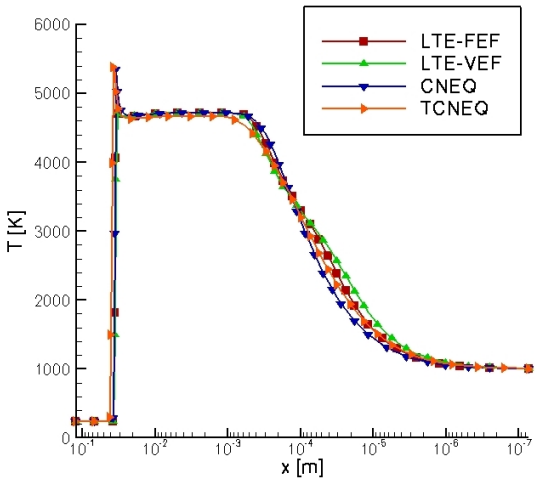


Figure 6.47: Comparison of the stagnation line profiles of temperature for LTE-FEF, LTE-VEF, CNEQ and TCNEQ in condition 1 ($M_\infty = 14$).

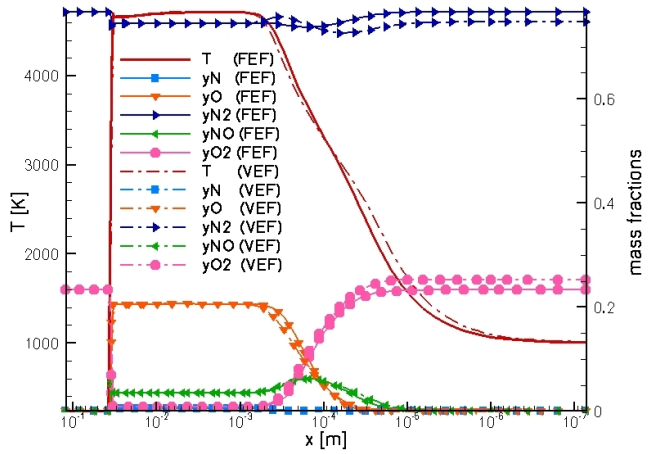


Figure 6.48: Stagnation line profiles of temperature and species mass fractions for LTE-FEF and LTE-VEF in condition 1 ($M_\infty = 14$).

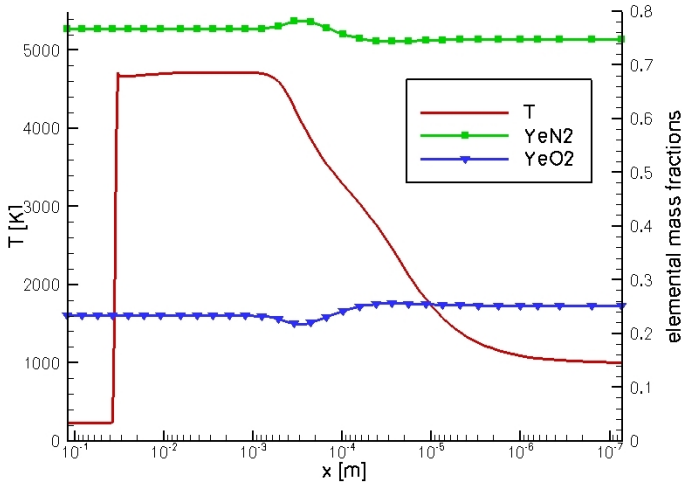


Figure 6.49: Oxygen and nitrogen elemental mass fractions distribution on the stagnation line for LTE-VEF in condition 1 ($M_\infty = 14$).

behaviour inside the boundary layer, where the demixing effect causes an increase of the oxygen element (and corresponding atomic) concentration, and a decrease of nitrogen element (and corresponding atomic) concentration with respect to the case with constant elemental fractions. As underlined in [141], the presence of oxygen tends to increase in the direction opposite to the temperature gradient at relatively low temperatures (below a certain threshold which varies with pressure), while the inverse situation occurs at higher temperatures. This trend, which is mainly due to the non-monototic behaviour with respect to temperature of the elemental thermal demixing coefficient D_O^T in Eq. 4.18 [79, 141], is clearly identifiable in Fig. 6.49, where the gradient of elemental fraction of oxygen presents a sign change from negative to positive: after the gradient inversion, the elemental fraction of oxygen increases with the decrease of temperature while approaching the wall.
The temperatures (roto-translational and vibrational for TCNEQ, the only temperature for CNEQ) and composition on the stagnation line with nonequilibrium conditions are presented in Fig. 6.50.

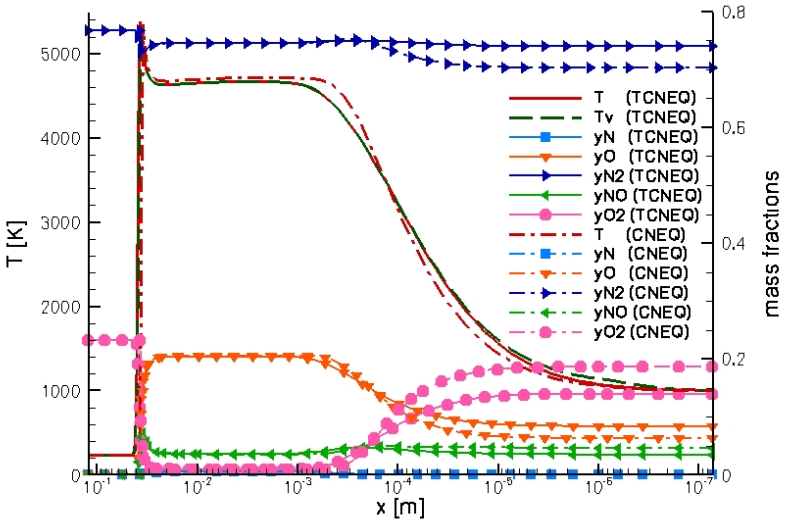


Figure 6.50: Comparison of the stagnation line profiles of temperature and species mass fractions for TCNEQ and CNEQ in condition 1.

6.3.2 Condition 2, $M_\infty = 18.4$

This 3D computation has been run in parallel on a 3.3 million hexaedra mesh (generated by Fabio Pinna), on up to 316 AMD64 processors on KU Leuven cluster. The geometry includes 4 open flaps (with the same angle), which anyway don't affect significantly the flowfield, since the secondary shocks generated in front of them don't interact with the bow shock. A global view of the surface mesh is presented in Fig. 6.51 while zoomed views near the nose and flap regions, including the surface mesh on the symmetry boundary, are shown in Figs. 6.52a and 6.52b.
The Liou Steffen AUSM scheme has been used for this computation. In this case, we have restricted ourselves to the multi-temperature model described in Sec. 4.3.3 and 5-species air mixture with reaction rate coefficients given by [120]. The full three dimensional flowfield can be viewed in Figs. 6.54 and 6.55 in terms of Mach number and roto-translational temperature respectively.

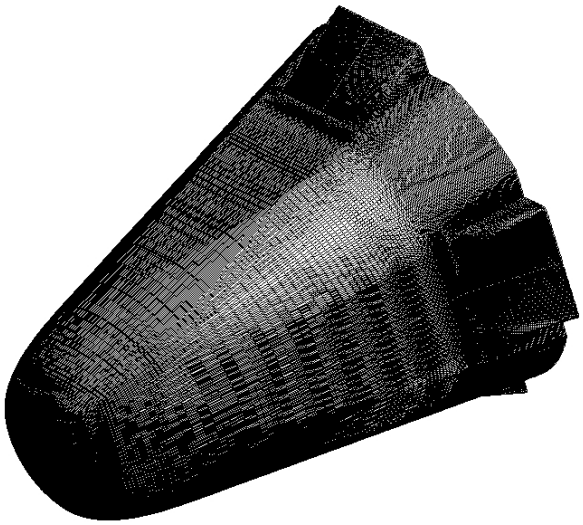
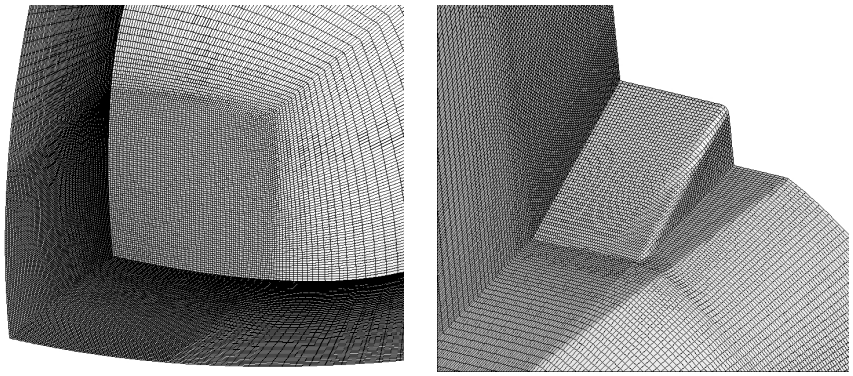


Figure 6.51: Global view of the wall surface mesh for the EXPERT vehicle.



(a) Zoom on the wall and mirror mesh surfaces near the nose

(b) Zoom on the wall and mirror mesh surfaces near one flap

Figure 6.52: Detailed views on the surface mesh for the EXPERT vehicle.

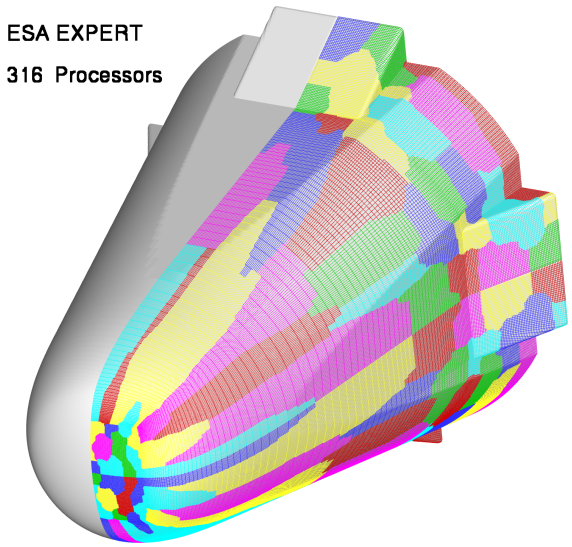


Figure 6.53: Partitioned surface mesh for the EXPERT vehicle.

The contours and isolines of vibrational temperatures of N_2 and O_2 are presented in Figs. 6.56 and 6.57, showing a considerable degree of thermal nonequilibrium, since the fields of both temperatures are very far from the one in Fig. 6.55. In particular, while the peak roto-translational temperature is higher than 12100 K, the post-shock vibrational temperatures are about 6300 K for $T^v_{N_2}$ and 5100 K for $T^v_{O_2}$.
In this simulation, however, the boundary layer is very likely not to be accurately resolved in 3D, due to the relative coarseness of the grid near the body (the first cell size is about 1 mm) and therefore the heat flux is not shown here. Further investigation on finer meshes and validation against results obtained by other codes would therefore be needed.

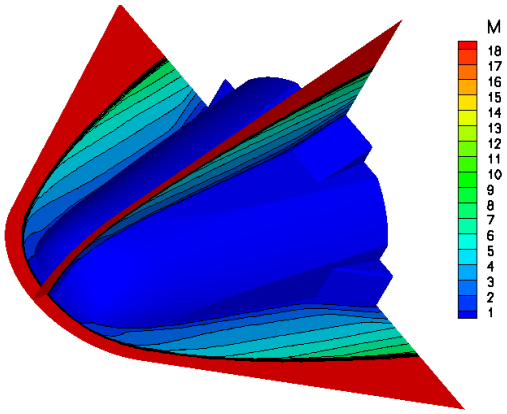


Figure 6.54: Mach number field on the EXPERT vehicle ($M_\infty = 18.4$).

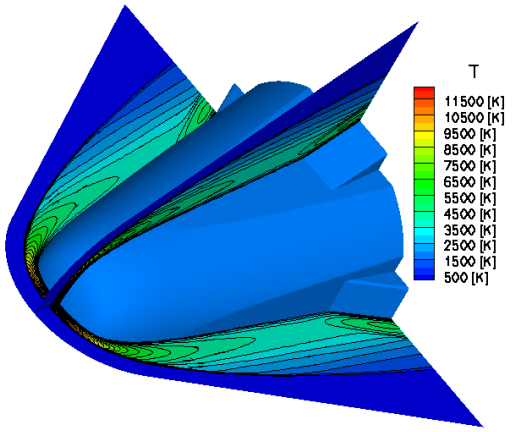


Figure 6.55: Roto-translational temperature on the EXPERT vehicle ($M_\infty = 18.4$).

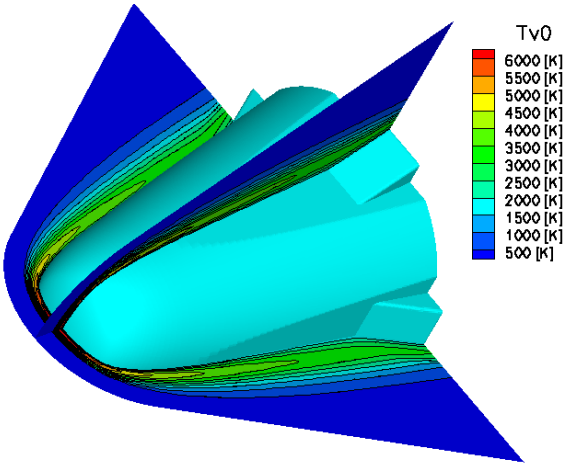


Figure 6.56: Vibrational temperature of N_2 field on the EXPERT vehicle ($M_\infty = 18.4$).

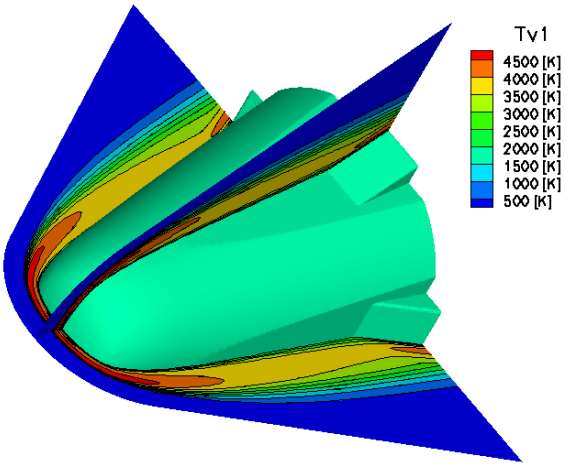


Figure 6.57: Vibrational temperature of O_2 field on the EXPERT vehicle ($M_\infty = 18.4$).

Part III

Gallery of COOLFluiD Results

Chapter 7

Gallery of COOLFluiD results

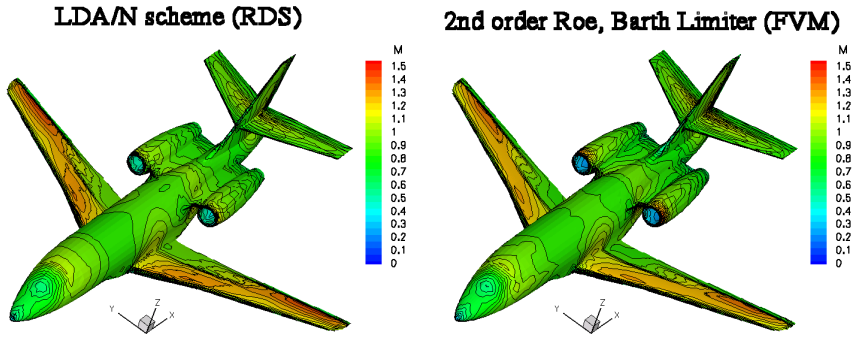
Although the emphasis of the thesis has been on aerothermodynamics applications, the developments reported in Part I and in Part II Chapter 5 have been used in many other applications, which was also part of this thesis and the main goals of the COOLFluiD project. In this chapter, we present a gallery of CFD simulations that have been performed with COOLFluiD, while reusing some of the functionalities. The aim is to demonstrate the wide range of applicability of the platform and to show how the framework and the many modules implemented during this thesis have been the basis for the developments and research study of many other team members.

7.1 Aeronautics

The FV and RD solvers implemented in this thesis have been applied to solve inviscid flows on complex configurations. Fig. 7.1 shows the solution in terms of Mach number on a Dassault Falcon aircraft at $M_\infty = 0.85$ and $\alpha = 1^o$ given by the second order FV and RD methods on the same fully tetrahedral mesh.
The result of a more challenging simulation on a F15 geometry (621,636 nodes, 3,558,667 tetrahedral cells) using the first order N scheme with $M_\infty = 0.95$ and $\alpha = 0^o$ is presented in terms of pressure coefficient C_p in Fig. 7.2. The Mach number field on the windward and leeward sides is shown in Figs. 7.3a and 7.3b.

7.2 Magneto Hydro Dynamics

After having developed the RD solver for Euler equations, we ported some of the functionalities developed in [8] in order to handle ideal Magnetohydrodynamics (MHD) applications. The first and second order solutions of a MHD flow inside a channel with a sinus bump, computed with the Nc and



(a) Mach field given by LDA/N scheme

(b) Mach field given by Roe scheme with LS reconstruction and Barth limiter

Figure 7.1: Mach number isolines/contours in the Euler flow on the Dassault Falcon at $M_\infty = 0.85$, $\alpha = 1^o$.

Bc schemes respectively, are presented in Figs. 7.4a and 7.4b, in terms of density and magnetic field lines (which, in this case, are supposed aligned with the flow).
In [170, 172], the basic explicit/implicit FV solver has been extended to handle ideal MHD applications with an Artificial Compressibility Approach. In this case, the augmented set of PDE's equations to solve is given by:

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \rho\mathbf{u} \\ \mathbf{B} \\ \rho E \\ \Phi \end{pmatrix} + \nabla \cdot \begin{pmatrix} \rho\mathbf{u} \\ \rho\mathbf{u}\mathbf{u} + (p + \mathbf{B}\cdot\mathbf{B}/2)\hat{I} - \mathbf{B}\mathbf{B} \\ \mathbf{u}\mathbf{B} - \mathbf{B}\mathbf{u} + \mathbf{I}\Phi \\ (\rho E + p + \mathbf{B}\cdot\mathbf{B}/2)\mathbf{u} - \mathbf{B}(\mathbf{u}\cdot\mathbf{B}) \\ \beta^2\mathbf{B} \end{pmatrix} = \mathbf{0} \tag{7.1}$$

where, unlike in the traditional approach based on the non conservative Powell source term to guarantee solenoidality of the magnetic field $\mathbf{B}$, an additional advection equation for the scalar potential function Φ is considered. Herein, β is a reference speed and the total energy can be expressed as

$$E = \frac{p}{\rho(\gamma - 1)} + \frac{1}{2}\left(V^2 + \frac{B^2}{\rho}\right) \tag{7.2}$$

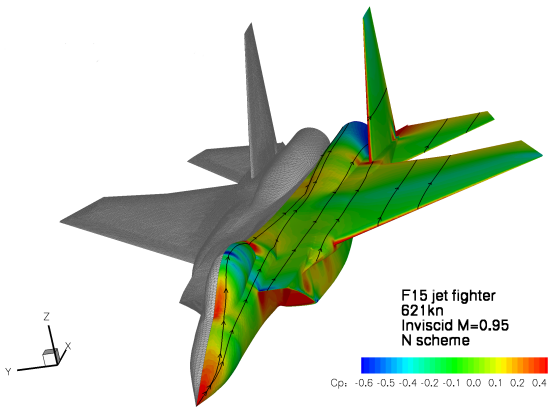
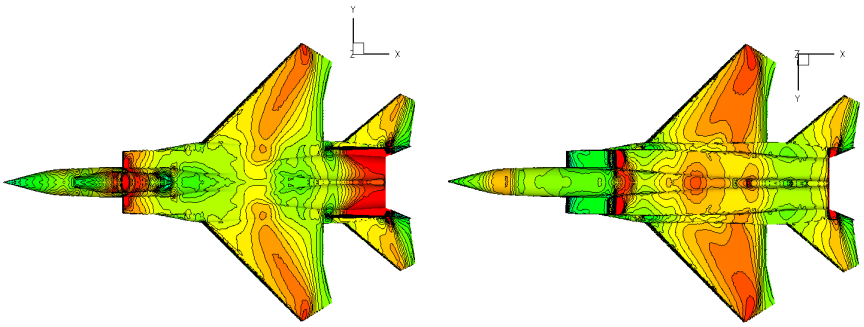


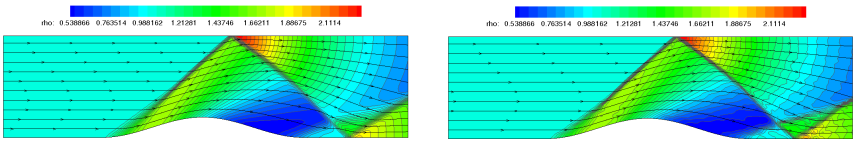
Figure 7.2: C_p distribution in the inviscid flow on a F15 (N scheme) at $M_\infty = 0.95$, $\alpha = 0^o$.



(a) View on the windward side (b) View on the leeward side

Figure 7.3: Mach number field on the F15 windward and leeward sides at $M_\infty = 0.95$, $\alpha = 0^o$.

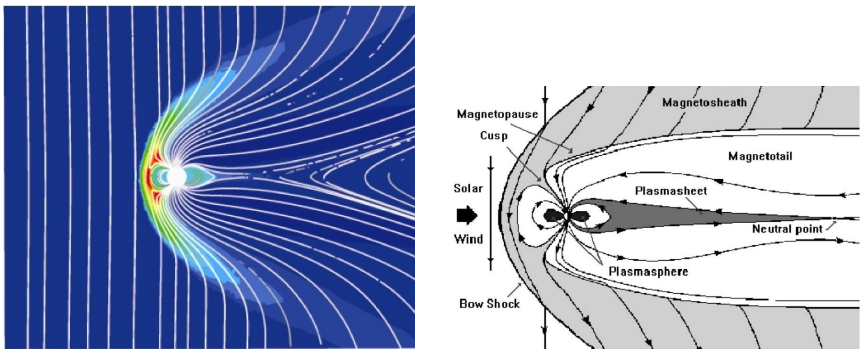
The result of the 3D simulation of the interaction between the Solar wind and the Earth magnetosphere, modeled as a dipole, obtained in [172] by means of a Lax-Friedrichs scheme with interactively tunable dissipation is shown in Fig. 7.5a in terms of isolines of magnetic field **B**. The visual com-



(a) Nc solution (b) Bc solution

Figure 7.4: Density contours and overposed magnetic field lines in the MHD flow inside a channel with a sinusoidal bump.

parison between the numerical solution and the expected flowfield depicted in Fig. 7.5b indicates that the main physical phenomena have been captured correctly.



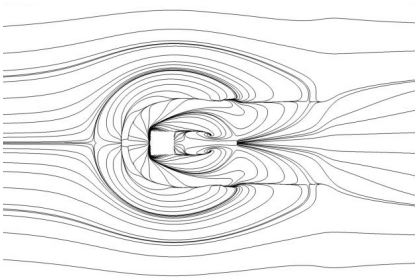
(a) Isolines of magnetic field in Earth magnetosphere on the symmetry plane (b) Schematic of the interaction between Solar wind and Earth magnetosphere

Figure 7.5: Simulation of the interaction between Solar wind and Earth magnetosphere in [172].

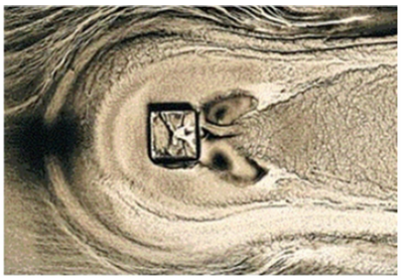
7.3 Complex laminar viscous flows

In the context of transition from laminar to turbulent regime, some preliminary work has been conducted on the numerical investigation of roughness induced vortical structures on a 3D flat plate by [125]. The conditions for this testcase are: $T_\infty = 300$ [K], $M_\infty = 0.6$, $Re_L = 1289$, $L = 2$ [mm], where the latter is the size of the roughness. The simulation was run with the im-

plicit Navier-Stokes FV solver on a hexaedral mesh generated with ANSYS Gambit, using AUSM+-up scheme and LS reconstruction. Fig. 7.6a shows the skin friction line pattern around a cubic roughness element on top of a flat plate. The computed laminar solution compares qualitatively reasonably well with the experimental measurements in Fig. 7.6b, where the flow past the cube is in fact turbulent. More detailed views of the flow near the cubic roughness, on the symmetry plane and on the plate surface, are shown in Figs. 7.7a and 7.7b.



(a) Skin friction line pattern in the computed solution



(b) Experimental visualization in oil of the skin friction line pattern

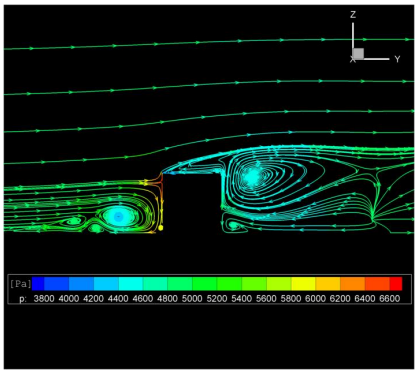
Figure 7.6: Simulation of vortical structures induced by a cubic roughness element on a flat plate at $M_\infty = 0.6$ and $Re_L = 1289$ in [125].

7.4 Turbulence RANS flows

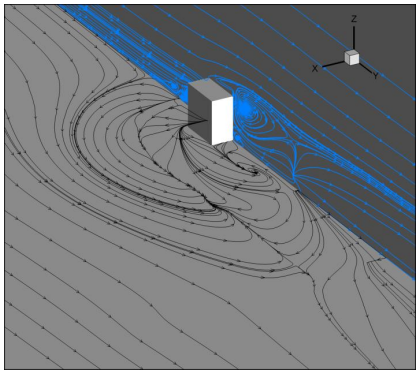
In [126, 169, 174] some turbulence models were integrated in the COOLFluiD framework and the FV solver was adapted to handle them. This has also offered a good occasion to validate our implementation of the Newton method for weakly coupled systems explained in Sec. 5.1.1. The overall linear system associated to the discretization of the Reynolds Averaged Navier-Stokes equations, in the example of a 2-equation model like $k-\omega$, can be cast as

$$\begin{pmatrix} \frac{\partial \tilde{\mathbf{R}}_1}{\partial \mathbf{P}_1}(\mathbf{P}) & 0 & 0 \\ 0 & \frac{\partial \tilde{\mathbf{R}}_2}{\partial \mathbf{P}_2}(\mathbf{P}) & 0 \\ 0 & 0 & \frac{\partial \tilde{\mathbf{R}}_3}{\partial \mathbf{P}_3}(\mathbf{P}) \end{pmatrix} \begin{pmatrix} \Delta \mathbf{P}_1 \\ \Delta \mathbf{P}_2 \\ \Delta \mathbf{P}_3 \end{pmatrix} = - \begin{pmatrix} \tilde{\mathbf{R}}_1(\mathbf{P}) \\ \tilde{\mathbf{R}}_2(\mathbf{P}) \\ \tilde{\mathbf{R}}_3(\mathbf{P}) \end{pmatrix} \quad (7.3)$$

where the update variables are given by:



(a) Pressure isolines on the symmetry plane



(b) Skin friction line pattern on the plate surface and pressure isolines on the symmetry plane

Figure 7.7: Different views of the flow induced by a cubic roughness element on a flat plate at $M_\infty = 0.6$ in [125].

$$\mathbf{P} = \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{pmatrix}, \quad \mathbf{P}_1 = (p,\ \mathbf{u},\ T)^T, \quad \mathbf{P}_2 = k, \quad \mathbf{P}_3 = \omega \tag{7.4}$$

Herein the *weak coupling* translates into the fact that the off-diagonal blocks $\frac{\partial \tilde{\mathbf{R}}_i}{\partial \mathbf{P}_j}$ with $i \neq j$ in the global system matrix can be assumed to be equal to 0, allowing us to solve three independent systems and save memory storage. The 2D Delery channel testcase has been chosen in [169] to validate the FV solver for turbulent flows. The geometry of the problem is sketched in Fig. 7.8.

The distributions of pressure on the upper and lower wall of the Delery channel given by BSL and SST turbulent models with COOLFluiD are presented in Figs. 7.9a and 7.9b and compared with the corresponding THOR solutions [154] and experimental measurements. The AUSM+-up scheme and LS reconstruction have been used for this simulation.

In particular, for both COOLFluiD and THOR, the SST model shows the better agreement with the reference experimental data. This is testified by the good shock capturing exhibited in Fig. 7.10, where the Mach isolines corresponding to the COOLFluiD solution with SST are plotted.

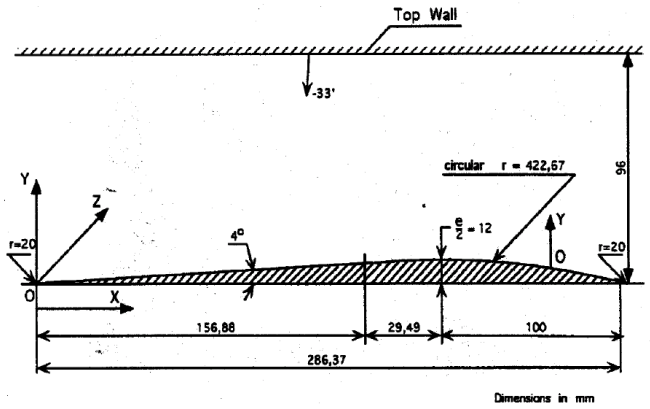
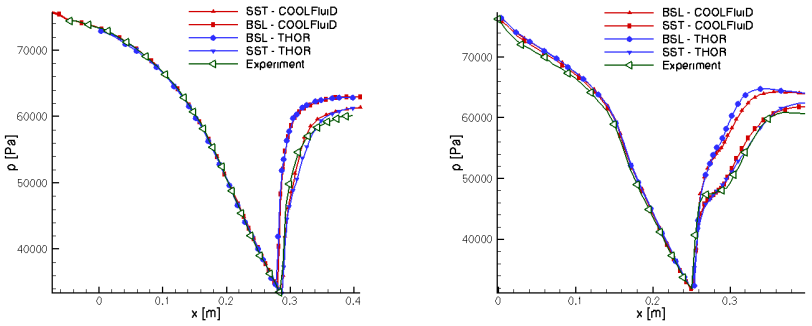


Figure 7.8: Geometry of the Delery channel.



(a) Distribution on the upper wall (b) Distribution on the lower wall

Figure 7.9: Static pressure distribution on the upper and lower walls of the Delery channel in [169]. Comparison against THOR results and experiments.

7.5 Inductively Coupled Plasma flows

In order to design thermal protection systems for space vehicles prior to flight, high-temperature wind tunnels have been developed, offering an al-

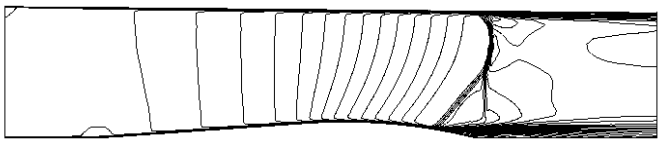


Figure 7.10: Mach number isolines computed on the Delery channel [169] at $M_{in} = 0.6$ with the SST model.

ternative way to the prohibitively expensive option of inflight testing. In ICP torches, the test samples are subjected to a hot plasma jet, which is created by exposing the test gas to an intense electro-magnetic field, generated by a radio-frequency electric current running through an inductor surrounding the plasma torch, as shown in Fig. 7.11.

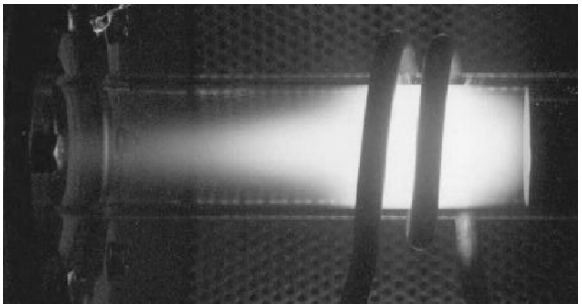


Figure 7.11: An argon plasma inside the VKI facility.

The induced electric current heats up the gas through Ohmic dissipation to a partially ionized plasma, reaching peak temperatures of approximately $10000K$. ICP torches are typically used to perform experimental measurements in subsonic chemically reacting boundary layers and stagnation regions downstream bow shocks.
In order to validate inhouse experiments, a structured FORTRAN90 code (ICP) for the simulation of axisymmetric plasma flows in thermo-chemical equilibrium or nonequilibrium in ICP torches was developed at the VKI by [7, 94, 140].
Under the author's supervision, some effort has been devoted to transfer these simulation capabilities from the ICP code to COOLFluiD [62, 129].

At the time of writing the migration is yet to be completed, but a 2D axisymmetric parallel unstructured LTE-FEF solver for incompressible ICP flows is already available.
In this truly multi-physical case, two subsystems of equations, one corresponding to the incompressible Navier-Stokes equations in LTE-FEF (1) and the other one being the magnetic induction (2) equations, are solved in a weakly coupled manner with the implicit Newton procedure explained in Sec. 5.1.1. The global linear system to be solved is

$$\begin{pmatrix} \frac{\partial \tilde{\mathbf{R}}_1}{\partial \mathbf{P}_1}(\mathbf{P}) & 0 \\ 0 & \frac{\partial \tilde{\mathbf{R}}_2}{\partial \mathbf{P}_2}(\mathbf{P}) \end{pmatrix} \begin{pmatrix} \Delta \mathbf{P}_1 \\ \Delta \mathbf{P}_2 \end{pmatrix} = - \begin{pmatrix} \tilde{\mathbf{R}}_1(\mathbf{P}) \\ \tilde{\mathbf{R}}_2(\mathbf{P}) \end{pmatrix} \tag{7.5}$$

where the update variables $\mathbf{P}$ are given by

$$\mathbf{P} = \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix}, \quad \mathbf{P}_1 = (\delta p,\ \mathbf{u},\ T)^T, \quad \mathbf{P}_2 = (E_r,\ E_I)^T \tag{7.6}$$

Herein, δp is the pressure fluctuation, while E_r and E_I are the real and imaginary parts of the induced electro-magnetic field. The Finite Volume method has been used for discretizing the ICP equations and, particularly, the pressure-stabilized Rhie-Chow scheme described in [62, 136] has been applied to calculate the convective flux of the incompressible Navier-Stokes equations.
We present here the two testcases corresponding to simulations of 8-species CO_2 and 11-species air flow in LTE inside the Plasmatron torch with a pressure of 10000 [Pa], a mass flow of 16 [g/s] at a power of 90 [Kw].
The solutions for CO_2 yielded by COOLFluiD and ICP code on the same mesh are shown in Figs. 7.12 and 7.13 in terms of temperature field. Some streamlines are superposed, in order to visualize the recirculation bubble near the inlet boundary. The qualitative good agreement is confirmed by the quantitative comparison given in Fig. 7.14, where corresponding profiles at $x = 0.2$ [cm] from the inlet are overposed. The differences between the two curves could be justified by the usage of two different thermochemical libraries (Mutation in COOLFluiD vs. Pegase in ICP) with slightly different models.
In the testcase with 11-species air, under the same operational conditions, the differences between COOLFluiD and ICP solutions are even smaller, as witnessed by the visual comparison between temperature contours in Figs. 7.15 and 7.16, but especially in Fig. 7.17, where the corresponding temperature profiles in the section at $x = 0.2$ [cm] overlap almost perfectly.

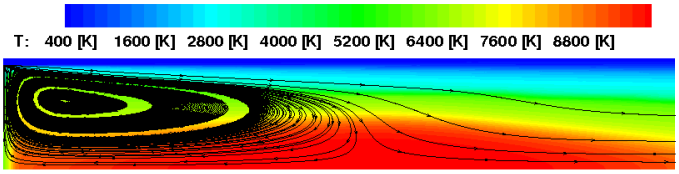


Figure 7.12: Temperature field and streamlines in the plasmatron torch given by COOLFluiD at 90 [Kw] for 8-species CO_2.

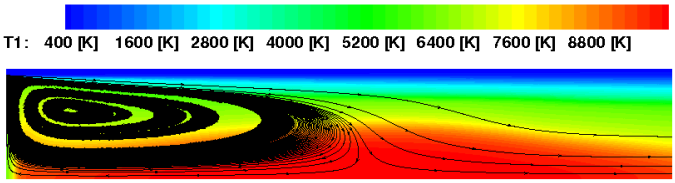


Figure 7.13: Temperature field and streamlines in the plasmatron torch given by the ICP code at 90 [Kw] for 8-species CO_2.

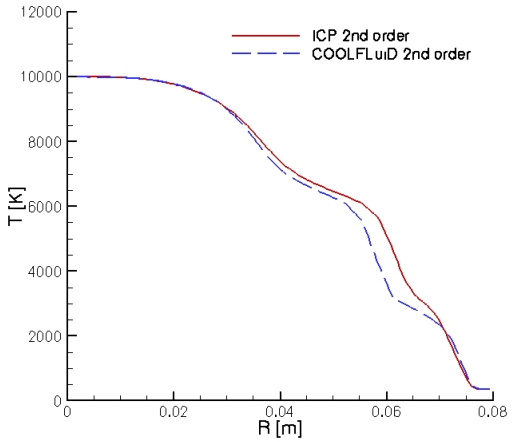


Figure 7.14: Comparison of temperature profiles at $x = 0.2$ [m] between ICP and COOLFluiD codes for 8-species CO_2.

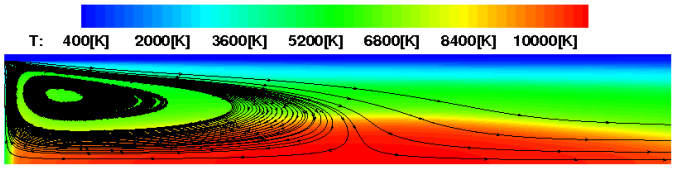


Figure 7.15: Temperature field and streamlines in the plasmatron torch given by COOLFluiD at 90 [Kw] for 11-species air.

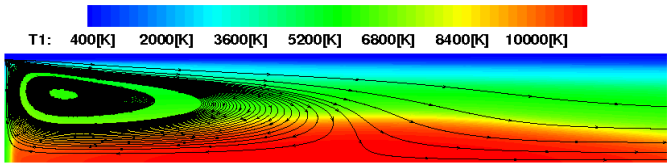


Figure 7.16: Temperature field and streamlines in the plasmatron torch given by the ICP code at 90 [Kw] for 11-species air.

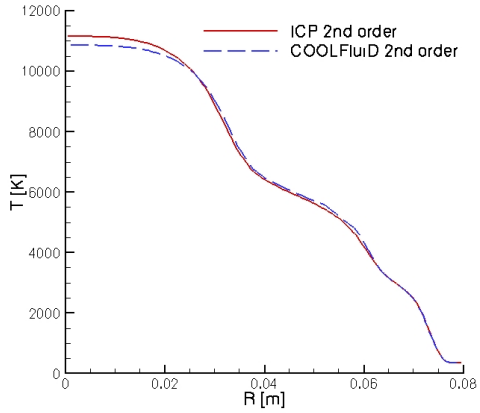


Figure 7.17: Comparison of temperature profiles at $x = 0.2$ [m] between ICP and COOLFluiD codes for 11-species air.

Part IV

Conclusion

Chapter 8

Conclusions and perspectives

In this thesis, we have described the author's contribution to the design and development of COOLFluiD, a high-performance and object-oriented framework for solving systems of PDE's on unstructured meshes governing coupled multiphysical phenomena. After having explained some of the main design and programming solutions developed to give a solid and flexible software infrastructure to the platform, particular effort has been devoted to present the systems of PDE's and the numerical algorithms that have been implemented for performing aerothermodynamics simulations.
To summarize, the overall project has required multi-disciplinary work consisting in

1. developing OO design techniques tailored for scientific computing in order to provide flexibility and scalability (here also intended as capacity for evolution) to the framework;

2. developing parallel algorithms based on MPI and making use of advanced programming techniques in C++ to support large scale and high-performance computing;

3. implementing multiple systems of equations for describing real gas flows in thermal and chemical equilibrium/nonequilibrium;

4. implementing state-of-the-art FV and RD numerical solvers for simulating a variety of flows, particularly in the hypersonic regime.

5. validating the aerothermodynamic solvers on a number of challenging testcases of current interest in different flow regimes.

8.1 Original contributions of this thesis

8.1.1 Object oriented design

As far as software design is concerned, in this thesis we have focused our attention on some specific issues and solutions that have been developed and implemented in COOLFluiD in close collaboration with Tiago Quintino (mostly) and Dries Kimpe for parallel computation issues. These design solutions could be successfully reused in other similar platforms as well.
In particular, the available literature related to object-oriented simulation of PDE's lacks design solutions that can offer flexibility in the serial and parallel data handling, run-time interchangeability between physics and numerics and structural support to encapsulate generic numerical algorithms in a scalable and reusable way. The ideas presented in Chapter 2 provide answers to all those questions:

- Serial and distributed mesh-related data are handled transparently by means of the DataStorage and DataHandle facilities [69, 80, 131] as described in Sec. 2.2, independent from the type of method (FV, FEM, etc.).

- The connectivity data corresponding to the edges, faces or cells belonging to different topological regions are interfaced by TopologicalRegionSets, which are numerics independent.

- Ready-to-use GeometricEntities with numerical-dependent stencil information are created on the fly from preallocated memory to apply numerical algorithms efficiently.

- The Perspective pattern [78, 80], introduced in Sec. 2.3, tackles the implementation of complex physical models, allowing a clean separation from the numerical method within certain limits.

- The Method-Command-Strategy concept [80, 131, 132] allows developers to encapsulate different classes of numerical algorithms (time and space discretizations, linear system solvers, mesh manipulators, coupling algorithms etc.) with a single highly customizable structural pattern, as shown in Sec. 2.4.1.

- An improved version of the self-registration technique [74, 78, 131], which allows the different components to be treated as dynamically linked plug-ins in COOLFluiD, is the main achievement of Sec. 2.5.

- The kernel for typesafe and size deducing fast expression templates presented in Appendix A offers an unrivaled potential for aggressive optimization of symbolic algebra operations among small arrays, even within software frameworks consisting of dynamically linked libraries like COOLFluiD.

All those architectural solutions have played a major role in allowing our platform to constantly enlarge its target applications during recent years. In particular, thanks to the growing network of developers (including VUB, KU Leuven and partners of the EU-ADIGMA and FP7 projects), several space discretization algorithms (FV, High Order and Space Time RD, FEM, Spectral FV, Spectral Finite Difference, DG), time integration schemes (Forward Euler, Runge Kutta, Backward Euler, Crank-Nicholson, 3-Point Backward, Time Limited), external linear system solvers (PETSc [73], Trilinos [72], Pardiso [3], SAMG [4], FlexMG) and physical models (compressible and incompressible Navier-Stokes, aerothermochemistry, MHD [172], structural analysis, heat transfer [131], turbulence, aeroelasticity [169], aeroacoustics, etc.) have been integrated, as well as error estimation based on the Hessian of a tracer variable, different mesh deformation algorithms, coupling facilities [169], input and output data format converters.
The growing success of the platform is probably the most reliable proof for the soundness of its original design, in which the author has played a major role. However, much more work is needed, especially for validating the above mentioned functionalities.

8.1.2 High Performance Computing

Given the aim of performing complex computational and memory intensive simulations, a considerable effort has been devoted in COOLFluiD since the start, in order to achieve an efficient parallelization.
With regard to this, a large part of the merit goes to Dries Kimpe [68] who implemented a first mesh partitioning algorithm, a transparent distributed vector and a robust synchronization algorithm, which at the time sufficed in order to perform relatively small parallel simulations using explicit time stepping.
On top of this, the author contributed to enhance the high-performance computing capabilities of COOLFluiD as follows, summarizing the content of Chapter 3.

- The implicit FV and FEM-like solvers have been parallelized by appropriately interfacing the linear system solvers (e.g. PETSc and Trilinos)

and by making sure to take into account only jacobian and residual contributions coming from updatable state vectors.

- A fully parallel algorithm to read the mesh starting from one single CFmesh (i.e. COOLFluiD's mesh data format) file has been implemented for arbitrary discretization stencils. In this case, the Parmetis library is used for the mesh decomposition and the element connectivity data are then redistributed to the target processors by means of a collective total exchange action. The algorithm works on hybrid meshes and allows the user to decide the size of the overlap region (up to n-layers of neighboring elements), according to the needs of the selected numerical method.

- An efficient parallel algorithm to write a single CFmesh file has also been developed: in this case the data are written by the master process by ranges, after having gathered data from all the other processes. The original node, state, geometric entities numberings in the output file are kept unaltered with respect to the initial input file.

All the above mentioned algorithms have been tested in dozens of cases and meshes on different computer architectures, with mesh sizes up to 20 million elements and running on up to 512 processors, as shown in Chapter 3.
The I/O performance results of Sec. 3.1.3, on the one hand, demonstrate the optimal behaviour of the parallel writing algorithm, which basically requires the same time on 512 as on 64 processors, testifying an implementation which minimizes communication. On the other hand, as the number of processes increases, the performance of the parallel reading deteriorates probably due to the heavy use of collective communication. In any case, even with a 20 million element mesh, the required reading+partitioning time is reasonably low (a few minutes) even on 512 processors and remains a negligible fraction of the global computational time.
The benchmarks in Sec. 3.1.4 show the overall remarkably good performance of the parallel solvers implemented in COOLFluiD, which makes them suitable and competitive for the simulation of challenging industrial problems. In particular, a parallel efficiency of 0.88 has been achieved in the case of the second order FV solver (with 2-layer of overlap) on 512 processors. However, at least in the case under examination, the RD solver enjoys a close to optimal performance only up to 200 processors, after which the parallel efficiency rapidly decreases. At the time of writing, the reason for this drop in performance is still to be investigated and some optimization effort will probably still be needed.

8.1.3 Simulation of aerothermodynamics

The author's contribution to the COOLFluiD framework was ultimately targeted towards aerothermodynamic applications. To this end, many physical models have been integrated and two numerical solvers have been implemented.

8.1.3.1 Physical modeling

As far as physical modeling is concerned, our work has mainly consisted in implementing different sets of PDE's describing high enthalpy gas physics by means of the Perspective pattern described in Sec. 2.3.1. Progressively, more and more complex models have been integrated: Euler, compressible and incompressible Navier-Stokes, LTE-FEF, LTE-VEF, ICP, chemical and thermal non-equilibrium with an arbitrary number of species and temperatures. In order to handle appropriately the physical phenomena occurring in these high-temperature reactive flows, a flexible interface for Mutation [94, 116, 140] has been designed and repeatedly upgraded to include the latest developments in close collaboration with Marco Panesi. This intense and fruitful collaboration driven by common interest is one of the best illustrations of the power of the COOLFluiD platform, which allows researchers to focus on their own developments and applications by taking maximum profit of others' work.
Mutation, a Fortran77 library still under development at the VKI, computes all thermodynamic, transport, chemistry and energy relaxation terms appearing in our systems of equations for aerothermochemistry. Inside COOLFluiD, Mutation offers just one concrete implementation of a general abstract interface that could be implemented differently by another library. As a proof of concept, PegaseC, a new version of Mutation and Pegase [22] written in ANSI-C, has already been successfully integrated [139] and can be currently used as an alternative to Mutation for LTE simulations.
The originality of our contribution from the physical modeling side is therefore also due to the accomplished integration of many state-of-the-art models all within the same software environment, giving it a unique multi-physics character if compared to similar well-known codes like NASA's DPLR [168] and LAURA [53], US3D [109], URANUS [49], EURANUS [128] which are more monolithic and , moreover, use less rigorous (but generally computationally more efficient) models for thermodynamics and transport.

8.1.3.2 Numerical solvers

Two parallel explicit/implicit general-purpose flow solvers for 2D/3D unstructured grids have been developed for this thesis: a steady/unsteady cell-centered FV solver for hybrid meshes and a steady vertex-centered RD solver (beit limited to 2D axisymmetric for some physical models).
Both codes have been designed with the MCS pattern presented in Sec. 2.4.1 and the core algorithms are truly physics independent, allowing the most of the developers to solve new sets of equations by simply working on the physics via the Perspective pattern 2.3.1 and by implementing a few specific boundary conditions, which are dependent on both physics and numerics.
Moreover, because of the polymorphic relations between interacting Methods, a dynamic link exists between a space discretization, a time integration scheme and a linear system solver, resulting in an even superior flexibility and extendibility of the current implementation, since the three Methods are all independent one from the other.

Finite Volume. The FV solver can handle 2D, axisymmetric and 3D cases and offers a great choice of schemes for the upwind treatment of the convective fluxes (several variants of AUSM, Lax Friedrichs, Roe, Steger-Warming, HLL, HUS, Van Leer, Rhie-Chow), while the diffusive fluxes are discretized in the central manner, with the computation of the viscous fluxes based on a face-centered diamond control volume. Particularly, our preferred choice for computing hypersonic chemically reacting flows on blunt bodies has been the AUSM+ scheme, which, although not as popular among researchers as Roe scheme or modified StegerWarming, enjoys excellent shock capturing and robustness, being generally not affected by the carbuncle instability. Moreover, in this thesis we have proposed a modification to the baseline AUSM+ scheme, consisting in a more consistent definition of the interface speed of sound in the case of flows in thermochemical nonequilibrium, proposed by [146] in a similar context. This has substantially increased the robustness of the AUSM+ scheme for the applications of interest in this thesis.
Second order accuracy is achieved with a weighted least square polynomial reconstruction with a multiple choice of computational stencils or with a MUSCL type approach suitable for structured meshes. In case of compressible flows, oscillation free solutions are obtained by applying flux limiters and, in case the limiter is applied, the historical modification technique is adopted to drive the solution to convergence. The same FV engine can be used in combination with almost all the available physical models, including cases like turbulent or ICP flows, in which multiple weakly coupled subsys-

tems of are solved. During the last 4 years, our solver has been the basis for the developments of several Diploma Course [18, 62, 101, 124, 126, 129, 135, 139, 171, 174] and PhD students [125, 169, 172].

Residual Distribution. The RD solver works on 2D, axisymmetric and 3D geometries, even though more validation has to be done, especially in this last case and for the complex physical models. Different schemes have been implemented by the author, including both LRD and CRD versions of system N, LDA, B and Bx. As pointed out in [40], the latter is currently the only second order scheme that works in combination with implicit time stepping if strong discontinuities are present in the flowfield. Viscous fluxes are discretized with a Petrov-Galerkin approach and source terms are upwinded together with the convective fluctuation.
Also in this case, as in FV, the method has been implemented in a flexible way, decoupled from the physics. As an example, the user can decide which variables (primitive, conservative, symmetrizing, etc.) to use for storing/updating the solution, distributing the residual, linearizing the flux jacobians and treating the diffusive terms.
During this PhD, a considerable effort has been devoted to transfer most of the simulation capability of the legacy THOR solver [154, 157] (except for the turbulence), with application to the Navier-Stokes equations and even MHD (cfr Sec. 7.2).
Afterwards, we have focused on the extension of the CRD method to handle 2D and axisymmetric viscous flows in thermo-chemical nonequilibrium [75, 76]. The latter represents a major step since the first attempt proposed in [36], in which the LRD approach was applied to 2D inviscid Euler equations in chemical nonequilibrium. Hence, this part can be considered the most innovative contribution of this thesis as far as numerical schemes development is concerned.

8.1.3.3 Results validation

The aerothermodynamic solvers that we have developed in COOLFluiD have been validated on a number of real-life high-speed testcases, which are subject of ongoing research in the aerospace community.
In particular, some challenging double cone and cylinder configurations have been investigated within the RTO Task Group 43 and the numerical results have been compared against experimental measurements. In the cylinder case, a blind study with unknown experimental data have been conducted. Our results have been considered among the best produced within the in-

ternational working group and have deserved to be presented at the coming 6th European Symposium on Aerothermodynamics for Space Vehicles, organized by ESA.
Moreover, we have performed some computations of a few points in the re-entry trajectory of the Stardust sample return capsule with a maximum speed up to 12.4 Km/s (Mach 42), in order to demonstrate the robustness and reliability of our code even at those extreme conditions, where considerable thermochemical nonequilibrium effects occur.
Finally, we have presented the results of some simulations on the EXPERT vehicle, including in 3D, in order to compare different schemes and physical models that we have implemented.

8.2 Future work and perspectives

We discuss now the possible improvements and the future perspectives for the work presented in this thesis.

8.2.1 COOLFluiD platform

The contribution presented in this thesis has been of fundamental importance for providing the COOLFluiD platform with a solid software infrastructure for large scale computing and with advance simulation capability on unstructured grids. This, in principle, can potentially make COOLFluiD competitive with widely used CFD packages like [44] for the simulation of a variety of applications mainly but not necessarily related to fluid dynamics. At the time of writing, the range of applicability of the framework has already been extended by [131] and more specifically by [169] to structural Finite Element analysis, conjugate heat transfer, aeroelasticity and truly multi-physics simulations, where arbitrary coupling algorithms are applied for solving different sets of equations in multiple domains, with matching and non-matching interface meshes. At a moment where industrial and technological progress require more and more multi-disciplinarity and can take profit of the increasing performance of HPC clusters for handling computationally intensive simulations, the real challenge now is to make this kind of complex multi-physics technology more user-friendly, requiring less inputs from the users and less implementation effort from new developers, while keeping a good performance in large scale parallel computations.
Within this context, the COOLFluiD architecture with its extreme modularity can play a significant role, by offering a solid base as a collabora-

tive software environment, where multiple partners can work synergically, bringing together their different expertises, while enjoying the benefits of an object oriented multi-physics environment where all the existing and new developments can be reused by everybody with minimal effort.

8.2.2 Aerothermodynamic solvers

As far as the aerothermodynamic solvers implemented by the author are concerned, more work is expected in order to enhance their performance and usability for making complex 3D simulations of flows in thermo-chemical nonequilibrium routinely possible.

8.2.2.1 Finite Volume

The fully implicit FV solver could be optimized significantly (about 30% in speed, according to some profiling driven considerations) by providing an analytical treatment for the jacobian of the diffusive fluxes. To this end, some modifications are needed inside the Mutation library in order to provide explicitly the multicomponent diffusion coefficients. A curve fit based re-implementation of some key subroutines for the thermodynamic and transport properties inside Mutation could contribute for another 30-40 % to the overall performance.
Moreover, a dramatic reduction of the memory requirements (probably higher than 50 %) and a considerable gain in run-time speed could be achieved by developing matrix free techniques or alternatively by implementing a parallel data line relaxation algorithm for hybrid unstructured meshes like the one described in [109].
Another alternative could be the integration of multigrid agglomeration algorithms such as in [122], but combined with implicit time stepping or data line relaxation as smoother, in order to keep the necessary robustness for tackling the stiff source terms, and not necessarily applied on hierarchical mesh data structures.
Mesh adaptation could also help to improve the overall performance, by reducing the number of degrees of freedom required in order to achieve accurate solutions. To this end, a straightforward extension of the current FV solver to handle non conformal meshes with hanging nodes would ease the mesh refinement/coarsening in cases where hybrid or fully quadrilateral/hexaedral meshes are employed, i.e. the most of the cases for hypersonic computations.

8.2.2.2 Residual distribution

The pioneering results obtained in this thesis in the simulation of the nonequilibrium hypersonic flow on a double cone configuration show the great potential of the CRD method for this kind of applications, for which, during the last 30 years, little alternative to FV has been proposed. However, in order to make our CRD solver usable in high mach number computations on 2D and 3D blunt bodies, involving strong bow shocks, an effective carbuncle fix has to be incorporated in the solver, e.g. as proposed by [35].
Moreover, in the current implementation, some robustness problems still exists; to cure them, an alternative to the Bxc scheme (for instance the limited Rusanov scheme) and different treatments of the source term could be investigated. Another possible improvement would consist in distributing the residual in symmetrizing variables as attempted in [36] for chemical nonequilibrium flows: this would lead to a more efficient scheme, where the species continuity equations would be decoupled and could be discretized with scalar schemes such as the strictly positive PSI.
Additionally, given the ongoing effort in COOLFluiD [164] to extend CRD to third or higher order of accuracy, the current aerothermodynamic solver could be easily modified to incorporate such changes. Finally, the use of adaptive tetrahedral remeshing as already developed by [96] could lead to major improvements in resolution and efficiency.

8.2.2.3 Physical modeling

Future work in the short term should focus on completing the integration of the collisional radiative models developed in [116], which are already available in Mutation, but not fully combined with the numerical solvers.
Moreover, state-of-the-art models for wall catalicity should be integrated shortly in order to improve the prediction of thermal loads associated to gas-surface interaction.
Another major development foreseen in the mid term consists in the coupling with external solvers for radiation and ablation. As schematically depicted in Fig. 8.1, the latter should imply a dynamical integration of two new plug-ins, one for radiation and one for ablation, developed by different project partners in order to respect some predefined interfaces at the kernel level.
Ideally, both FV and RD solvers should be able to use the new packages, since the corresponding interfaces would be unique and numerics-independent, consistently with our base philosophy.

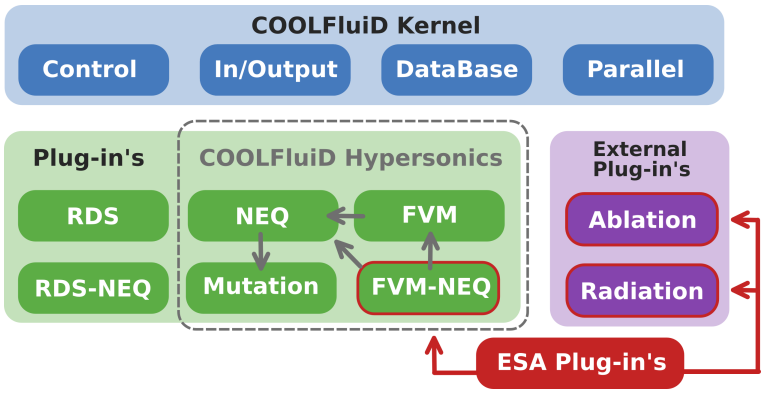


Figure 8.1: Schematic showing the possible integration of ESA's ablation and radiation modules into COOLFluiD.

8.2.2.4 Towards coupled aerothermoelastic simulations

A preliminary effort to perform a coupled aerothermoelastic simulation on the EXPERT geometry has been made in [169]. Fig. 8.2 shows a schematic and the computed temperature field in such a case, where our FV code for chemically reacting flows has been coupled with a multi-material FEM solver for conjugate heat transfer, including the effect of thermal expansion in the metallic portion of the thermal shield (with associated mesh deformation) and radiative equilibrium on the body surface. This challenging simulation shows the potential offered by COOLFluiD for tackling aerothermodynamic problems in all their complexity, with a truly multi-physical approach. With some additional work, these capabilities could be enhanced and extended to include important phenomena such as gas-surface interaction, gas radiation and ablation in a reasonably short time frame.

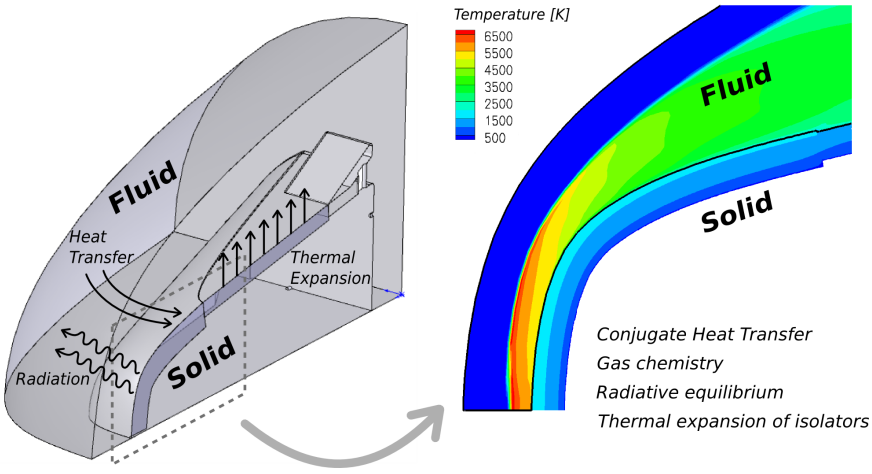


Figure 8.2: Schematic (left) and calculated temperature contours (right) in the aerothermoelastic simulation on the EXPERT vehicle in [169].

Part V

Appendices

Appendix A

Appendix

A.1 Typesafe and Size-Deducing Fast Expression Templates

When dealing with numerical algorithms it is common to handle expressions involving algebraic operations among arrays like

```
a = K1*(b + d) + K2*(c - d + K3) - K4*(e - f)
```

where a, b, c, d are arrays and Ki are constants, or between matrices and arrays like

```
a = A*(b + d) - K1*B*(b-d) + C*f
```

where A, B, C are matrices. While symbolic algebra is a built-in feature for popular programming languages like MATLAB and Fortran90, in C++ such feature is not readily available, but can be easily obtained via operator overloading [149]. Unfortunately, a straightforward implementation of this kind, which is not shown here but can be found for example in [147], would lead to unacceptable performance overheads, mainly due to unnecessary copying and on-the-fly allocation/deallocation of temporaries.
About one decade ago, however, when Expression Templates (ET) were developed concurrently by Veldhuizen [161] and Vandervoorde (a former VKI student!) [160], new perspectives for high performance object oriented computing arose. The technique makes use of template (meta-)programming [160], [162] to implement mathematical expressions efficiently and to overcome all the above mentioned traditional problems related with operator overloading. The key idea consists in encoding generic expressions in template arguments and delaying the actual evaluation till the result assignment takes place.
Since ET birth, many software packages like [45] (historically the first one)

or [71] have been exploiting that technique. Even the C++ Standard Template Library provides an own implementation of ET based arrays, known as *valarray* [160].
As a matter of fact, the power and weakness of ET rely both heavily on the optimization capabilities of the compilers, that are strongly architecture dependent and that not always deliver the expected performance. In particular, [46], [47] discovered and pointed out some efficiency problems in the available ET implementations, mainly related with the complex ET constructs and the aliasing of pointers. In order to cure those problems, a novel implementation of ET, denominated FET [66],[57], introduces the concept of enumerated variables and the use of static data members. FET show a consistent performance enhancement if compared to classical ET, especially on large size arrays and on computers enabling automatic parallelization. However, in their original implementation, here summarized in Sec. A.1.1, FET are strongly type unsafe and because of the unmanageable use of global enumeration, they could not be employed within a large object-oriented framework supporting dynamical plug-ins, like COOLFluiD [78, 80].
In this section, which summarizes the paper [77], we show how we have improved the FET concept to make it usable in a much wider range of applications, including ours. We first introduce a type safe correction in Sec. A.1.2. We then focus our attention on enhancing the performance of FET on small arrays and, to this end, in Sec. A.1.3 we present a size-deduction technique allowing the compiler to perform loop-unrolling if at least one of the arrays involved in a generic expression has a fixed size.
Some illustrative examples showing the applicability of the proposed ideas can be found in Sec. A.1.4. Finally, some benchmark results are presented in Sec. A.1.5 to validate our contribution.

A.1.1 Fast Expression Templates

As explained in [66] and [57], FET require the definition of a parent `Expr` class:

```
template <typename DERIVED, typename T>
class Expr {};
```

from which all expression classes, including the array classes themselves, must derive. In this case, a technique known as *Curiously Recurring Template Pattern* (CRTP) [160] is employed. The latter allows the class `Expr` to delegate the actual evaluation to a generic concrete class, that is indicated

by the first template parameter `DERIVED`. The second template parameter `T` simply stands for the return type of the expression (`double`, `float`, etc.). A possible derived class is `Add` and the corresponding overloaded operator has to be defined:

```
template <typename E1, typename E2, typename T>
struct Add : public Expr<Add<E1,E2,T>, T> {
  static inline T at (unsigned int i)
  {
    return E1::at(i) + E2::at(i);
  }
};

template <typename E1, typename E2, typename T>
inline Add<E1,E2,T> operator+ (const Expr<E1,T>& e1,
                               const Expr<E2,T>& e2)
{
  return Add<E1,E2,T>(e1,e2);
}
```

As a consequence of the straightforward definition of the `Add` expression, the `Vector` class must provide a static `at`() function as well. This implies the usage of a static data member in `Vector` and the addition of an enumerated variable, `NUM`, as template parameter, in order to be able to handle different `Vector` types within the same expression.

```
template <typename T, unsigned integer NUM>
class Vector : public Expr<Vector<T,NUM>,T> {
public:
  ...
  static inline T at (unsigned integer i) {return m_v[i];}

  template <typename EXPR>
  const Vector& operator= (const Expr<EXPR,T>& expr)
  {
    for (unsigned integer i = 0; i < m_size; ++i) {
      m_v[i] = EXPR::at(i);
    }
    return *this;
  }

private:
  unsigned integer m_size;
```

```
  static T* m_v;
};
```

As exhaustively discussed and analyzed in [66] and [57], the use of FET allows available compilers to overcome the efficiency penalty due to the aliasing of pointers [46], [47] which arises in expressions involving the same array more than once, and to approach the performance of hand-coded C loops more than classical ET implementations.

A.1.2 Typesafe Enumeration Policy

The key ideas behind FET are also their major pitfalls, because of the intrinsic type unsafety given by the concept of enumerators and the use of a static pointer data member associated to each enumerated variable. FET in the original formulation [66] are in fact hardly usable within large object-oriented frameworks or dynamically linked libraries, where each module can behave as a black-box for the others and it would be impossible to assign a unique enumerator to each newly defined array safely. The simple solution proposed in [66], which consists in defining each vector on a different line and choosing the `__LINE__` macro to determine the enumerator is extremely dangerous and non portable, since vectors defined in the same line of different files would share the same static data pointer, even without this being in the developer's intentions.
To improve the situation, we can define an empty *Tag* class:

```
template <typename TYPE, unsigned integer NUM> class Tag {};
```

The first parameter is a generic type, while the second one is the enumerated variable that was introduced in [66]. The difference is that now the uniqueness of each array structure is related to the pair `TYPE` + `NUM` and this allows the developer to define more easily and in a safer way different vectors in different classes. Our modified `Vector` class becomes:

```
template <typename T, typename TAG>
class Vector : public Expr<Vector<T,TAG>,T> {
 ...
};
```

The replacement of the old integer with the `TAG` typename as second template parameter is the only modification needed in order to integrate the typesafe enumeration policy in the original FET-based `Vector` implementation.

Another issue that may arise now is the need for a certain degree of automatization in choosing the `Tag` template parameters for each newly defined array inside a `GivenClass`. A macro definition and a simple `typedef` can serve our purpose:

```
#define DOUBLEVEC(nvar) Vector<double, Tag<SELF, nvar> >

class GivenClass {
public:
  ...
private:
  typedef GivenClass SELF;

  DOUBLEVEC(0) m_v1;
  DOUBLEVEC(1) m_v2;
};
```

The usage of the `DOUBLEVEC` macro within a class requires the developer to define a `typedef` with the name `SELF`: this must be declared `private`, so that it remains visible only within the class scope, without possible name clashes with derived classes. The `__LINE__` macro could still be used in place of the macro argument `nvar` to improve readability, at the price of not being able to define arrays safely in the implementation files because, accidentally, the line of instantiation for an array inside a class member function could match one of the lines in which one array data member is defined in the header.
Even in our improved solution, however, a possibly severe limitation still exists: in general, it is not guaranteed to use more than one instance of `GivenClass` safely within the same code, because all the `Vectors` with the same `Tag`, but aggregated by different objects, would share access to the same static memory, with foreseeable bad consequences. Assuming that developers have been properly informed of this risk, this limitation does not compromise the applicability of the proposed idea in a wide range of cases, i.e. in all those cases in which two instances of the same class, with the exact same type, are never going to be used at the same time.

A.1.3 Size-Deducing FET for Small Arrays

To the author's knowledge, in the available literature or software packages related to ET (or FET), including [71] and [45], the case of ET that could handle both dynamic and fixed-size arrays within the same expression has

never been considered before. This feature turns out to be advantageous for small arrays, i.e. with size $O(1)$ or $O(10)$ which are heavily used in numerical codes for scientific computing, because it can enable that powerful compiler optimization known as loop unrolling. The latter is beneficial only for algebra on small arrays (and small corresponding loops), since its positive effect can be counter-balanced and annulled by the excessive code expansion and cache misses occurring if the size of the loop to unroll is too big [29]. In our experience with the design and development of COOLFluiD featuring dynamical *plug-in* objects [78, 80], we often come out with the possibility of working locally (i.e. within a member function in some specific classes) with mathematical expressions involving an hybrid combination of dynamically allocated arrays with unknown size at compilation time and fixed-size ones, without fully exploiting the optimization capabilities offered by such a favourable situation.
We will now present a technique that allows to detect and take profit of a situation where at least one of the arrays involved in a generic expression has a fixed size.

A.1.3.1 Compare Class

The first ingredient in our implementation is a `Compare` class performing some trivial meta-programming [160],[162] to compare two unsigned integers and determine the maximum between the two:

```
template <unsigned int N, unsigned int M>
struct Compare;

template <unsigned integer N>
struct Compare<N,0> {enum {MAX=N};};

template <unsigned integer M>
struct Compare<0,M> {enum {MAX=M};};

template <unsigned integer N>
struct Compare<N,N> {enum {MAX=N};};

template <>
struct Compare<0,0> {enum {MAX=0};};
```

The default template class `Compare` that compares two integers both different from zero is on purpose not instantiable and any attempt to use it would result in a compilation error. Only the available template partial and

explicit specializations are meant to be used and the following three cases are identifiable:

- zero and N (or vice-versa), with N bigger than zero
- both N and M bigger than zero
- both zero

The last case with both zeros must be provided explicitly, otherwise the compiler is not able to disambiguate among all the available specializations. The following two macros will turn out to be useful later on:

```
#define NMAXT2(t1,t2) Compare<t1::SIZE,t2::SIZE>::MAX

#define NMAXNT(num,type) Compare<num,type::SIZE>::MAX
```

The `NMAXT2` macro expects two types (`t1` and `t2`) and compares the corresponding enumerators, denominated `SIZE`, that are associated to these types. `NMAXNT` is meant to accomplish a similar task, but accepting one integer `num` and a type as arguments.

A.1.3.2 Size-Deducing Expression

The classical FET wrapper expression class `Expr` [66], [57] is modified, and here renamed `FExpr`, as follows:

```
template <typename DERIVED, typename T, unsigned int NUM>
class FExpr {
public:
  enum {SIZE=NUM};
};
```

An additional integer parameter `NUM` is added to record the knowledge of the size in a locally defined enumerated variable. We consider now a possible derived class `Add`:

```
template <typename E1, typename E2, typename T>
struct Add : public FExpr<Add<E1,E2,T>, T, NMAXT2(E1,E2)> {
  static inline T at(unsigned int i)
  {
    return E1::at(i) + E2::at(i);
  }
};
```

The sizes of the two expressions `E1` and `E2` are compared by the above described `NMAXT2` macro and the bigger size is passed as third template parameter to the base `FExpr`, that assigns the value to its `SIZE` enumerated variable. At the end of the full expression template recursive evaluation, the biggest available size is recorded in the final enclosing expression.
The operator overloading function corresponding to the new `Add` object needs also a tiny modification, which consists in assigning a value equal to each expression `SIZE` to the third template parameter of the `FExpr` function arguments:

```
template <typename E1, typename E2, typename T>
inline Add<E1,E2,T> operator+ (const FExpr<E1,T,E1::SIZE>& e1,
                               const FExpr<E2,T,E2::SIZE>& e2)
{
  return Add<E1,E2,T>(e1,e2);
}
```

A.1.3.3 Improved Vector Class

We define a new `Vector` class provided with an additional integer parameter `N`, with zero as default value:

```
template <typename T, typename TAG, unsigned integer N = 0>
class Vector : public FExpr<Vector<T,TAG,N>,T,N> {
public:
  ...
  static T at (unsigned integer i) {return m_v[i];}
  unsigned integer size() const {return m_size;}

  template <typename EXPR>
  const Vector& operator= (const FExpr<EXPR,T,EXPR::SIZE>& expr)
  {
    const unsigned integer nmax = NMAXNT(N,EXPR);
    for (unsigned integer i = 0; i < GETSIZE(nmax); ++i) {
      m_v[i] = EXPR::at(i);
    }
    return *this;
  }

private:
  unsigned integer m_size;
  static T* m_v;
};
```

This new implementation makes already use of the safe enumeration policy and the related `TAG` template parameter that were presented in Sec. A.1.2. Moreover, the new `Vector` class is provided with a third template parameter, an integer, whose default value is 0. The size deduction mechanism relies on this last parameter.
If all the arrays involved in the full expression

```
*this = EXPR
```

have the last parameter set to 0, the `NMAXNT` macro (see Sec. A.1.3) returns 0, while `GETSIZE`, defined as

```
#define GETSIZE(nn) ((nn > 0) ? nn : size())
```

returns the actual size of the assignee `Vector`, `size`(), which is only known at run-time. This corresponds to a standard FET case and no loop unrolling is possible.
However, if at least one of the involved arrays is instantiated with $N > 0$, `NMAXNT` and `GETSIZE` will both return `N` at compile time and the loop defined in the assignment operator will be unrollable. Moreover, the strong type checking will produce a compile time error if two arrays declare two different values of `N`, both different from 0.
Besides providing a way of activating the size deduction, the usage of the integer template parameter can also be used to provide two different implementations (i.e. specializations) for the cases with fixed ($N > 0$) and dynamical size ($N = 0$), in order to offer even more optimization capabilities on some architectures. On one hand, a static C style array could be used instead of a dynamically allocated one in the case with fixed `N`, with consequent no need to hold an additional integer member data to store the size:

```
template <typename T, typename TAG, unsigned integer N = 0>
class Vector : public FExpr<Vector<T,TAG,N>,T,N> {
 ...
private:
  static T m_v[N];
};
```

On the other hand, a partial specialization of `Vector` would be needed for the default case with $N = 0$:

```
template <typename T, typename TAG>
class Vector<T,TAG,0> : public FExpr<Vector<T,TAG,0>,T,0> {
 ...
private:
  unsigned integer m_size;
  static T* m_v;
};
```

A.1.4 Applications

Typesafe and size-deducing FET (SDFET) are especially designed to improve the performance of large object-oriented scientific computing frameworks, e.g. COOLFluiD [78, 80], which make lot of use of algebra on small arrays (e.g. physical state or coordinate vectors). In the ideal situation for the applicability of the described technique, the size of those arrays can only be known at run-time at the higher level of abstraction, but not so at the lower level, where the usage of generic algorithms can embed the knowledge at compile time of the actual size.

A.1.4.1 Integration of SDFET in Arbitrary Assignee Array

The size-deducing mechanism can easily be applied to expressions where the assignee array is implemented in an arbitrary way and not necessarily as described in Sec. A.1.3.3. A certain generic array class, here called `MyVector`, can be made SDFET-compatible by simply adding an overloading of the assignment operator (plus eventually $+=$, $*=$, etc.) which accepts a `FExpr` as argument. Another issue consists in selecting the appropriate value of the first argument for the macro `NMAXNT`. In the general case in which `MyVector` does not have compile time knowledge of its size, i.e. the latter is not a template parameter, the required value must be 0, as shown below.

```
class MyVector {
public:
...
 template <typename EXPR>
  const MyVector& operator= (const FExpr<EXPR,T,EXPR::SIZE>& expr)
  {
    const unsigned integer nmax = NMAXNT(0,EXPR);
    for (unsigned integer i = 0; i < GETSIZE(nmax); ++i) {
      (*this)[i] = EXPR::at(i);
```

```
    }
    return *this;
  }
};
```

In terms of performance, there is no dependency of the SDFET on the assignee array implementation. Therefore, in any case, SDFET can enhance the efficiency of an existing array implementation, by enabling loop unrolling whenever possible.

A.1.4.2 Example: Flux Computation

As an illustrative example, let's consider a polymorphic functor object [10] that computes a convective numerical flux through a face in a Finite Volume fashion:

```
class ConcreteFlux : public BaseFlux {
public:
 ...
 virtual void operator()(const Face& face, Vec& result)
 {
  const Vec& uR = face.getState(RIGHT);
  const Vec& uL = face.getState(LEFT);
  ...
  result = a*(m_fR + m_fL) + b*(m_fR - m_fL) - c*(uR -uL);
 }

private:
  Vec m_fR;
  Vec m_fL;
};
```

In order to incorporate the typesafe and SDFET, the class above needs to be templatized with the array size and modified as follows:

```
template <unsigned int N>
class ConcreteFlux : public BaseFlux {
public:
 ...
 ~ConcreteFlux() {m_uR.release(); m_uL.release();}

 virtual void operator()(const Face& face, Vec& result)
 {
```

```
   m_uR = face.getState(RIGHT);
   m_uL = face.getState(LEFT);
   ... // similar expression as before
 }

private:
  typedef ConcreteFlux<N> SELF;

  DOUBLEVEC(0) m_uR;
  DOUBLEVEC(0) m_uL;
  DOUBLEVEC(N) m_fR;
  DOUBLEVEC(N) m_fL;
};
```

First of all, we note that the `DOUBLEVEC` macro needs to be upgraded to include the additional integer template parameter for Vector:

```
#define DOUBLEVEC(ns) Vector<double, Tag<SELF, __LINE__>, ns>
```

The typedef `SELF` must then be defined, preferably in the private scope of the class, in order to prevent name clashes with possible derived classes. Let's assume that `Vec` is of type `MyVector`. The additional data members `m_uR` and `m_uL` must be introduced, in order to assign all the possible non FET arrays on the right hand side of the expression (`uR` and `uL`) to FET arrays. As explained before, the size-deduction can work if the assignee array is an arbitrary one, but the expression to be assigned must involve only FET arrays. Therefore, in our case, a lightweight overloaded assignment operator has to be added to the `Vector` interface, more precisely to the partial specialization corresponding to $N = 0$:

```
template <typename T, typename TAG>
class Vector<T,TAG,0> : public FExpr<Vector<T,TAG,0>,T,0> {
 ...

 template <class TYPE>
 const Vector& operator=(MyVector<TYPE>& array)
 {
  m_v = &array[0];
  m_size = array.size();
  return *this;
 }

 void release() {m_v = NULL; m_size = 0;}
```

```
private:
  unsigned integer m_size;
  static T* m_v;
};
```

A `release()` function should also be included in `Vector` and called before the destruction of `m_uR` and `m_uL`, in order to avoid the risk of double deleting the pointer `m_v`.
In our case, even though the most of the sizes of the arrays involved in the flux expression are only known at run-time, the computation with FET can take full profit of loop unrolling, thanks to the compile time detection of the size of the locally instantiated arrays `m_fR` and `m_fL`. Moreover, the client code can hold a pointer to the abstract `BaseFlux` and call polymorphically the virtual function `operator()`, letting the developer combine a clean object-oriented design with superior performance. The only minor drawback is that the new implementation requires multiple instantiations of the `ConcreteFlux` object, one for each "useful" value of the parameter `N`, but the use of design patterns like *Abstract Factories* [10] [48] or techniques like the *self-registration* of objects [19] [78, 80] can help automatizing this.
As demonstrated by this example, the technique presented in this paper offers an easy way of introducing performance enhancements incrementally, without affecting the client code.

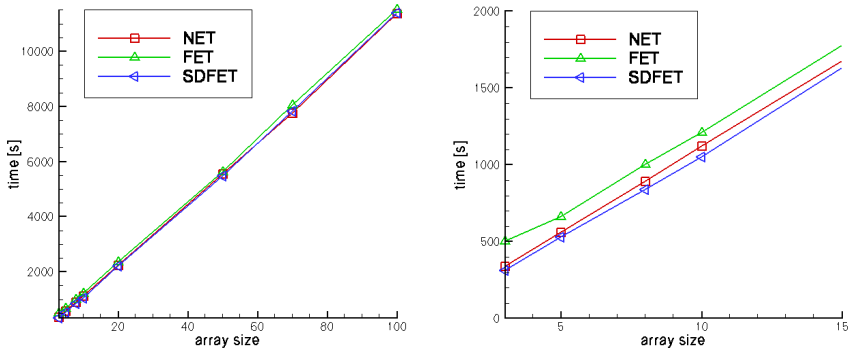
A.1.5 Performance Results

Some performance tests have been run serially on different machines and with different compilers, in order to verify the effect of size-deduction and consequent loop unrolling on FET with small arrays. To this end, we analyze the benchmark results concerning one expression resembling the realistic case presented in Sec. A.1.4.2:

```
a = b*(c + d) + b*(c - d) - b*(e - f);
```

where a, b, c, d, e, f are all arrays and only one of them has a fixed size. A comparison between SDFET with standard FET and non ET (NET) is performed. The SDFET implementation uses 2 different template specializations for `Vector`, with different storages, as described at the end of Sec. A.1.3.3. In the NET case, C style arrays with fixed size are used in order to enable the loop unrolling and make the most restrictive comparison with SDFET.

The first results, shown in Fig. A.1, are obtained on a Pentium4, 2.0 Ghz, 512Mb RAM with GNU 3.3.5 compiler. SDFET appear to outperform FET, as expected, and their result match almost perfectly the NET ones, even at very low sizes, i.e. $O(1)$. It is especially in this last case that the effect of loop unrolling is non-negligible, with a benefit in speed up to 15%.



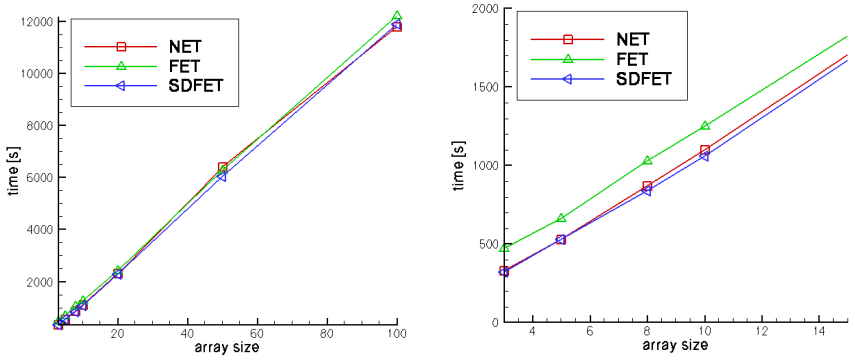
(a) Time vs. array size (3-100). (b) Zoom: time vs. array size (3-15).

Figure A.1: Benchmark on Pentium4 2.0 Ghz with GNU compiler

A similar behaviour is obtained also in the second test, which was run on a AMD 3000+, 1Gb RAM with GNU 3.4.5 compiler. The results are presented in Fig. A.2 and the general trend does not differ considerably from the previous case, with the SDFET faster than FET especially for array sizes of $O(1)$.

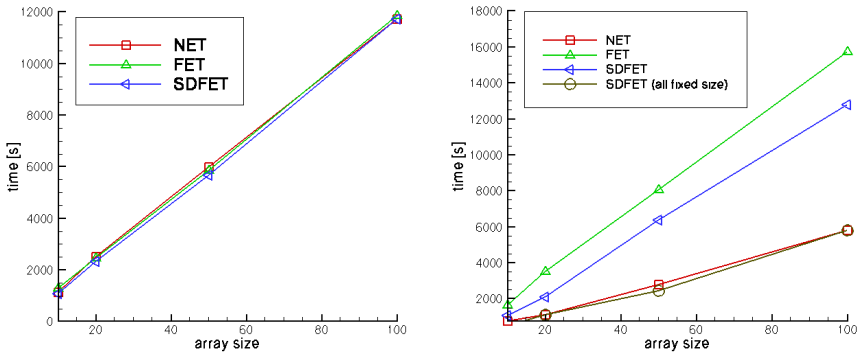
The last test was performed on a Pentium4, 3.4 Ghz, 2Gb RAM. The results obtained with the GNU 3.4.5 compiler are shown in the first picture in fig. A.3. No considerable performance difference is noticeable, even though SDFET behaves slightly better than the others.

Significantly more interesting is the second graph in fig. A.3, which exhibit the results when the Intel 9.0 compiler is chosen. In this case, loops were vectorized and a considerable difference exists between FET and SDFET results, in favour of the latter. However, NET widely outperforms both of them. An additional curve is plotted, i.e. the one related to SDFET with all arrays of type `DOUBLEVEC(N)` (see Sec. A.1.4.2), corresponding to the explicit specialization of `Vector` using C style arrays as storage: this implementation turns out to be the most efficient, since for some array sizes it performs even better than NET.



(a) Time vs. array size (3-100). (b) Zoom: time vs. array size (3-15).

Figure A.2: Benchmark on AMD 3000+ with GNU compiler



(a) Time vs. array size for GNU compiler. (b) Time vs. array size for Intel compiler.

Figure A.3: Benchmark on Pentium4 3.4 Ghz with GNU (left) and Intel (right) compiler

A.1.6 Loop fusion

We present here a further extension of the SDFET implementation in order to support loop fusion, i.e. the evaluation of several expressions within the same loop. Let's suppose, for example, to aim at writing a SDFET that, once optimized, logically reduces to

```
for (int i = 0; i < N; ++i) {
 e[i]  = 0.5*(a[i] + b[i])+ c[i] - d[i];
 f[i] += a[i] - 2*b[i] + 3*c[i] + d[i];
}
```

where one single loop of size N (= the size for all arrays) is needed to compute all the required operations to fill in arrays `e` and `f`, instead of two distinct loops, one per array, as in a standard FET or ET implementations. The basic idea is that we would like to avoid fetching the arrays `a`,`b`,`c`,`d` twice from memory if we evaluate the expressions in blocks which are small enough to fit into cache. This way, the arrays stay in cache and only need to be fetched from main memory once. The corresponding FET code looks like

```
fuse(e = 0.5*(a + b)+ c - d,
     f += a - 2*b + 3*c + d);
```

The implementation requires the definition of several functions `fuse()`, one overloaded version for each foreseen number of arguments (expressions) to fuse. In our example with two arguments we have:

```
template <class L1, class R1, class BinOp1,
          class L2, class R2, class BinOp2>
inline void fuse(const LeftRightExpr<L1,R1,BinOp1>& ex1,
 const LeftRightExpr<L2,R2,BinOp2>& ex2)
{
  const int ns = ex1.size();
  for (int i = 0; i < ns; ++i) {
    BinOp1::op(ex1.left[i], R1::at(i));
    BinOp2::op(ex2.left[i], R2::at(i));
  }
}
```

Herein, the template function accepts references to two template classes `LeftRightExpr` as arguments and is parameterized with two triplets of parameters (`L1`,`R1`, `BinOp1` and `L2`,`R2`, `BinOp2`), each one corresponding to the template parameters of one of the two LeftRightExpr. The latter is defined as follows:

```
template <class LeftExpr, class RightExpr, class BinOp>
class LeftRightExpr {
public:
  // constructor
```

```
  LeftRightExpr(LeftExpr& l, int size) : left(l),_nmax(size) {}

  // the size of the expression
  int size() const {return _nmax;}

  LeftExpr& left; // reference to the left expression
private:
  const int m_nmax; // expression size
};
```

where the three template parameters represent the assignee expression (`LeftExpr`), the expression to be assigned (`RightExpr`) and the binary operator (`BinOp`) to use for the assignment (=, +=, etc.).
In order to make the whole machinery work, we still need to define an overloading of the assignment operators inside the Vector class:

```
template <typename T, unsigned integer NUM>
class Vector : public Expr<Vector<T,NUM>,T> {
 public:
   ... // everything as before

 template <class EXPR>
 LeftRightExpr<Vector<T,TAG,N>, EXPR, EqualBin<T> >
  operator= (const Expr<EXPR,T,EXPR::SIZE>& expr)
 {
   typedef Vector<T,TAG,N> VEC;
   return LeftRightExpr<VEC,EXPR,EqualBin<T> >(*this, N);
 }

 ... // the same for +=,-=, etc.
};
```

where `EqualBin<T>` is a binary operator function which implements an elementwise assignment between two variables `a` and`b` of generic type:

```
template <class T>
class EqualBin {
  public:
  static inline void op(T& a, const T& b) {a = b;}
};
```

The overloading of the assignment operator (=) returns a LeftRightExpr which is partially specialized with a Vector. This ensures than when the

RHS of the expressions inside `fuse()` is parsed, the expression is assigned to a LeftRightExpr. Therefore, the expression evaluation is deferred to the `fuse()` function itself and not directly assigned to a Vector (case without loop fusion).
In our benchmarks (here not reported) with the GNU compiler version 3.4 on a 32-bit AMD machine and with version 4.1.2-13 on a 64-bit Dual Core AMD Opteron(tm) Processor 280, the use of loop fusion did not show any performance improvement, independently from the size of the arrays and the complexity of the expressions involved. When we repeated the benchmarks with the latest Intel compiler (10.1) on the same 64-bit machine (unfortunately not an Intel one), we unfortunately found the same result. This, however, does not mean that this technique, which in principle should enhance the cache friendliness of the code, cannot provide any advantage. It could probably perform better on other architectures and/or with some compilers possessing superior capabilities to optimize fat loops.

Bibliography

Each reference is followed by the list of the pages where it is cited.

[1] Flux limiter, 2008. URL `http://en.wikipedia.org/wiki/Flux_limiter`. 124

[2] Message passing interface forum. URL `http://www.mpi-forum.org`. 57

[3] Pardiso solver project, 2007. URL `http://www.pardiso-project.org`. 217

[4] Samg (algebraic multigrid methods for systems), 2008. URL `http://www.scai.fraunhofer.de/samg.html?&L=1`. 106, 217

[5] von karman institute for fluid dynamics, 1956-. URL `http://www.vki.ac.be`. 268

[6] E. J. H. A. M. Bruaset and H. P. Langtangen. Increasing the efficiency and reliability of software development for systems of pdes. In *Modern Software Tools for Scientific Computing*. Birkhäuser, 1997. 2, 25, 37

[7] D. V. Abeele. *An Efficient Computational Model for Inductively Coupled Air Plasma Flows under Thermal and Chemical Non-Equilibrium*. PhD thesis, Katholieke Universiteit Leuven, Chaussée de Waterloo, 72, 1640 Rhode-St-Genèse, Belgium, Nov. 2000. 97, 98, 103, 208

[8] *Á*. Csík. *Upwind Residual Distribution Schemes for General Hyperbolic Conservation Laws and Application to Ideal Magnetohydrodynamics*. PhD thesis, Katholieke Universiteit Leuven, Faculteit Wetenschappen Centrum voor Plasma-Astrofysica, Belgium, 2002. 139, 145, 148, 201

[9] *Á*. Csík, M. Ricchiuto, and H. Deconinck. A conservative formulation of the multidimensional upwind residual distribution schemes for general nonlinear conservation laws. *J. Comput. Phys.*, 179(2):286–312, 2002. 139, 145, 147

[10] A. Alexandrescu. *Modern C++ Design.* C++ In-Depth Series. Addison-Wesley, 2001. ISBN 0-201-70431-5. 13, 28, 38, 41, 136, 239, 241

[11] J. D. Anderson. *Hypersonic and High Temperature Gas Dynamics.* McGraw-Hill, 1989. 86, 98

[12] W. Bangerth, R. Hartmann, and G. Kanschat. deal.ii: A finite element differential equations analysis library. URL `http://www.dealii.org`. 2

[13] P. F. Barbante. *Accurate and Efficient Modeling of High Temperature Nonequilibrium Air Flows.* PhD thesis, von Karman Institute, Rhode-Saint-Genèse, Belgium, 2001. 85, 96, 98, 101, 110, 129, 130, 132, 133

[14] P. F. Barbante, G. Degrez, and G. S. R. Sarma. Computation of Nonequilibrium High-Temperature Axisymmetric Boundary-Layer Flows. *Journal of Thermophysics and Heat Transfer*, 16(4):490–497, 2002. 98

[15] T. Barth. Numerical methods and error estimation for conservation laws on structured and unstructured meshes. 33rd Computational Fluid Dynamics Lecture Series. Von Karman Institute, March 2003. 62, 120

[16] T. Barth. Aspects of unstructured grids and finite volume solvers fro the euler and navier-stokes equations. 25th Computational Fluid Dynamics Lecture Series. Von Karman Institute, March 1994. 62, 108, 120

[17] T. J. Barth and D. C. Jespersen. The design and application of upwind schemes on unstructured meshes. In *AIAA Paper 89-0366*, Reno(NV), Jan 1989. 37th AIAA Aerospace Science Meeting and Exhibit. 123

[18] K. Bensassi. Numerical investigations of hypersonic flow in the long-shot facility. Diploma course report, Von Karman Institute for Fluid Dynamics, 2007. 221

[19] J. Beveridge. Self-registering objects in c++. *Dr. Dobbs Journal*, Aug. 1998. 3, 39, 51, 241

[20] A. Bonfiglioli. *Studio di algoritmi per l'approssimazione di sistemi differenziali iperbolici. Applicazioni al calcolo di flussi comprimibili*

e non viscosi in configurazioni tridimensionali. Tesi di dottorato di ricerca in ingegneria delle macchine, Politecnico di Bari, 1995. 142

[21] A. Bonfiglioli and H. Deconinck. Multidimensional upwind schemes for the 3d euler equations on unstructured tetrahedral meshes. In H. Deconinck and B. Koren, editors, *Notes on Numerical Fluid Mechanics*, volume 57, pages 141–185. Vieweg, Braunschweig, 1997. 139

[22] B. Bottin. Pegase 4.4—perfect gas equations for arbitrary mixtures at low pressures and high temperatures. , VKI Manual 47-A-B-C, von Karman Institute for Fluid Dynamics, St.-Genesius-Rode, Belgium, november 1997. 219

[23] B. Bottin, D. Vanden Abeele, M. Carbonaro, G. Degrez, and G. S. R. Sarma. Thermodynamic and transport properties for inductive plasma modeling. *J. Thermophys. Heat Transf.*, 13(3):343–350, 1999. 80, 81

[24] B. Bottin, D. Vanden Abeele, M. Carbonaro, G. Degrez, and G. S. R. Sarma. Thermodynamic and transport properties for inductive plasma modeling. *Journal of Thermophysics and Heat Transfer*, 13:343–350, 1999. 133

[25] I. D. Boyd, J. Zhong, D. A. Levin, and P. Jenniskens. Flow and radiation analyses for stardust entry at high altitude. In *AIAA Paper 2008-1215*, Reno (Nevada), Jan 2008. 46th AIAA Aerospace Sciences Meeting and Exhibit. 179, 180, 182, 183, 184, 185, 186

[26] R. Broglia, M. Manna, H. Deconinck, and G. Degrez. Development and validation of an axisymmetric solver for hypersonic flows. VKI TN 188, von Karman Institute for Fluid Dynamics, St.-Genesius-Rode, Belgium, May 1995. 77, 79

[27] R. S. Brokaw. Thermal conductivity of gas mixtures in chemical equilibrium. II. *Journal of Chemical Physics*, 32(4):1005–1006, 1960. 82

[28] A. M. Bruaset and H. P. Langtangen. *A Comprehensive Set of Tools for Solving Partial Differential Equations; Diffpack, Numerical Methods and Software Tools in Industrial Mathematics.* Birkhäuser, 1997. 27, 37

[29] D. Bulka and D. Mayhews. *Efficient C++. Performance programming techniques.* Addison-Wesley, Oct. 2000. ISBN 0-201-37950-3. 16, 234

[30] J. N. Butler and R. S. Brokaw. Thermal conductivity of gas mixtures in chemical equilibrium. *Journal of Chemical Physics*, 26(6):1636–1643, 1957. 82

[31] F. Coquel and M. S. Liou. Hybrid upwind splitting by a field-by-field decomposition. TM 106843, NASA, Jan 1995. 110

[32] S. V. D. Kimpe and S. Poedts. Evector: an efficient vector implementation. In *POOSC'05 Workshop Notes*, Glasgow, Jul 2005. Von Karman Institute. 15

[33] N. Dale. *C++ data structures*. Jones and Bartlett, 1999. 10

[34] H. Deconinck, P. L. Roe, and R. Struijs. A multidimensional generalization of roe's difference splitter for the euler equations. *Computer and Fluids*, 22(2/3):215–222, 1993. 139

[35] H. Deconinck, M. Ricchiuto, and K. Sermeus. Introduction to residual distribution schemes and stabilized finite elements. In *VKI LS 2003-05, 33rd Computational Fluid Dynamics Course*, Rhode Saint Genese, 2003. 139, 147, 149, 224

[36] G. Degrez and E. van der Weide. Upwind residual distribution schemes for chemical non-equilibrium flows. In *14th AIAA Computational Fluid Dynamics Conference*, Norfolk, USA, 1999. AIAA 99-3366. 139, 145, 151, 221, 224

[37] M. Delanaye. *Polynomial reconstruction finite volume schemes for the compressible Euler and Navier-Stokes equations on unstructured adaptive grids*. PhD thesis, University of Liege, Faculty of Applied Sciences, 1996. 121, 125

[38] M. Delanaye, M. J. Aftosmis, M. J. Berger, Y. Liu, and T. Pulliam. Automatic hybrid-cartesian grid generation for high-reynolds number flows around complex geometries. In *AIAA Paper 99-0777*, Reno(NV), Jan 1999. 37th AIAA Aerospace Science Meeting and Exhibit. 126, 127

[39] J. V. Diijk. *Modelling of Plasma Light Sources, an object oriented-approach*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, 2001. 51

[40] J. Dobeš. *Numerical Algorithms for the Computation of Unsteady Compressible Flows over Moving Geometries. Applications to Fluid-Structure Interaction*. PhD thesis, Czech Technical University, Prague, Czech Republic, Université Libre de Bruxelles, Belgium, November 2007. 57, 129, 139, 146, 221

[41] J. Dobes and H. Deconinck. A shock sensor-based second-order blended (bx) upwind residual distribution scheme for steady and unsteady compressible flow. In *Hyperbolic Problems: Theory, Numerics, Applications*, 978-3-540-75711-5 (print), 978-3-540-75712-2 (online), pages 465–473. Springer Berlin Heidelberg, 2008. 139, 147

[42] M. C. Druguet, G. V. Candler, and I. Nompelis. Effect on numerics on navier-stokes computations of hypersonic double-cone flows. *AIAA Journal*, 43(3), March 2005. 114, 115, 158, 159, 162, 165

[43] B. H. et al. Object-oriented tools for solving pdes in complex geometries, 2004. URL `http://www.llnl.gov/casc/Overture`. 37

[44] H. W. et al. Openfoam: The open source cfd toolbox, 2004. URL `http://www.opencfd.co.uk/openfoam`. 2, 25, 27, 37, 222

[45] T. V. et al. Blitz++, 2006. URL `http://oonumerics.org/blitz`. 229, 233

[46] K. D. F. Bassetti and D. Quinlan. Towards fortran 77 performance from object-oriented c++ scientific framework. In *HPC 98*, Apr 1998. 230, 232

[47] K. D. F. Bassetti and D. Quinlan. C++ expression templates performance issues in scientific computing, Oct 1997. 230, 232

[48] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994. ISBN 0-201-63361-2. 11, 12, 19, 21, 26, 34, 38, 51, 241

[49] U. Gerlinger, H.-H. Fruehauf, and T. Boenisch. Implicit upwind navier-stokes solver for re-entry nonequilibrium flows. In *AIAA Paper 1997-2547*, Atlanta (GA), June 1997. 32nd Thermophysics Conference. 219

[50] D. Giordano. Impact of the born-oppenheimer approximation on aerothermodynamics. *Journal of Thermophysics and Heat Transfer*, 21(3):647–657, 2007. 83

[51] D. Giordano. Hypersonic-flow governing equations with electromagnetic fields. In *VKI LS 2004-01, Introduction to Magneto-fluid-dynamics for aerospace applications*, Rhode Saint Genese, Oct 2004. 84

[52] V. Giovangigli. *Multicomponent flow modeling*. Birkhäuser, 1999. 89

[53] P. A. Gnoffo. Upwind-biased, point-implicit relaxation strategies for viscous hypersonic flows. *AIAA Paper*, (89-1972-CP), 1989. 111, 219

[54] P. A. Gnoffo, R. N. Gupta, and J. L. Shinn. Conservation equations and physical models for hypersonic air flows in thermal and chemical non-equilibrium. Technical Paper 2867, NASA, 1989. 85, 86, 87, 88, 89, 97

[55] S. K. Godunov. A difference scheme for numerical computation of discontinuos solution of hydrodynamic equations. In *Math. Sbornik*, number 47, pages 271–306. 109

[56] R. N. Gupta, J. M. Yos, R. A. Thompson, and K. P. Lee. A review of reaction rates and thermodynamic and transport properties for an 11-species air model for chemical and thermal non-equilibrium calculations to 30 000 K. Reference Publication 1232, NASA, August 1990. 97, 100

[57] J. Hardtlein and C. Pflaum. Blocking techniques with fast expression templates. In *POOSC'05 Workshop Notes*, Glasgow, Jul 2005. Von Karman Institute. 13, 230, 232, 235

[58] C. Hirsch. *Numerical Computation of Internal and External Flows*, volume 2. John Wiley & Sons, 1990. 108, 109, 131

[59] J. O. Hirschfelder, C. F. Curtiss, and R. B. Bird. *Molecular Theory of Gases and Liquids*. Wiley, New York, 1964. 98

[60] J. O. Hirschfelder, C. F. Curtiss, and R. B. Bird. *Molecular theory of gases and liquids*. John Wiley and Sons, New York, 1967. 97

[61] D. G. Holmes and S. D. Connell. Solution of the 2d navier-stokes equations on unstructured adaptive grids. In *AIAA Paper 89-1932*, 1989. 127

[62] R. Honzátko. Implementation of a finite-volume inductively coupled plasma model into the object-oriented solver coolfluid. Diploma course report, Von Karman Institute for Fluid Dynamics, 2006. 208, 209, 221

[63] C. Hughes and T. Hughes. *Parallel and Distributed Programming Using C++*. Addison-Wesley, 2004. 57

[64] E. Issman. *Implicit Solution Strategies for Compressible Flow Equations on Unstructured Meshes*. Phd thesis, Université Libre de Bruxelles, Belgium, 1997. 129, 139, 152, 153

[65] D. R. O. J. C. Taylor and H. A. Hassan. Upwind methods for ionized flows. In *6th AIAA/ASME Joint Thermophysics and Heat Transfer Conference*, Colorado Springs, CO, June 1994. AIAA 94-1957. 84, 89

[66] A. L. J. Hardtlein and C. Pflaum. Fast expression templates: Object-oriented high performance computing. In P. M. A. S. V. S. Sunderam, G. D. van Albada and J. J. Dongarra, editors, *Computational Science ICCS 2005*, volume 2 of *LNCS 3514*, pages 1053–1063, Atlanta, GA, USA, May 2005. Emory University, Springer. 13, 230, 232, 235

[67] G. Karypis. Parmetis: Parallel mesh partitioning. URL `http://www-users.cs.umn.edu/~karypis/metis/parmetis`. 58, 60

[68] D. Kimpe. *On the Scalable Parallelisation of Unstructured Mesh Applications*. PhD thesis, Katholieke Universiteit Leuven, 2008. 5, 15, 64, 71, 217

[69] D. Kimpe, A. Lani, T. Quintino, S. Poedts, and S. Vandewalle. The coolfluid parallel architecture. In D. K. B. Di Martino and J. J. Dongarra, editors, *Proc. 12th European Parallel Virtual Machine and Message Passing Interface Conference*, pages 520–527, Sorrento, Oct 2005. Springer. 14, 216

[70] B. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. `http://dx.doi.org/10.1007/s00366-006-0049-3`. 2

[71] L. A. N. Laboratories. Pete: Portable expression templates engine, 2007. URL `http://www.acl.lanl.gov/pete`. 230, 233

[72] S. N. Laboratories. The trilinos project, 2007. URL `http://software.sandia.gov/trilinos`. 48, 106, 217

[73] A. N. Laboratory. Petsc: Portable, extensible toolkit for scientific computation, 2007. URL `http://www-unix.mcs.anl.gov/petsc`. 43, 47, 106, 217

[74] A. Lani. Development of an object oriented framework for pde solvers on unstructured grids. Travail réalisé pour obtenir le dea en sciences appliquées, Université Libre de Bruxelles, Faculté des Sclences Appliquées, Sep 2003. 3, 51, 216

[75] A. Lani and H. Deconick. A residual distribution method for hypersonic flows in thermo-chemical non-equilibrium. accepted at the 6th European Symposium on Aerothermodynamics for Space Vehicles, Versailles, France, Nov 2008. 221

[76] A. Lani and H. Deconick. Conservative residual distribution method for hypersonic flows in thermochemical nonequilibrium. In *accepted for presentation*, Orlando (FL), Jan 2009. 47th AIAA Aerospace Science Meeting and Exhibit. 221

[77] A. Lani and H. Deconinck. Typesafe and size-deducing fast expression templates for small arrays. In *POOSC'06 Workshop Notes*, Nantes, Jul 2006. Von Karman Institute. 230

[78] A. Lani, T. Quintino, D. Kimpe, H. Deconinck, S. Vandewalle, and S. Poedts. The COOLFluiD framework: Design solutions for high-performance object oriented scientific computing software. In P. M. A. S. V. S. Sunderan, G. D. van Albada and J. J. Dongarra, editors, *Computational Science ICCS 2005*, volume 1 of *LNCS 3514*, pages 281–286, Atlanta, GA, USA, May 2005. Emory University, Springer. 3, 14, 25, 38, 39, 51, 55, 216, 230, 234, 238, 241

[79] A. Lani, J. Molnar, D. V. Abeele, P. Rini, T. Magin, and G. Degrez. In E. O. P. Wesseling and Periaux, editors, *Numerical study of elemental demixing in atmospheric entry flow regimes near local thermodynamic equilibrium*, Egmond Aan Zee (Netherlands), Sep 2006. ECCOMAS 2006, TU Delft. 82, 193

[80] A. Lani, T. Quintino, D. Kimpe, H. Deconinck, S. Vandewalle, and S. Poedts. Reusable object-oriented solutions for numerical simulation of pdes in a high performance environment. *Scientific Programming. Special Edition on POOSC 2005*, 14(2):111–139, 2006. 14, 25, 38, 39, 55, 216, 230, 234, 238, 241

[81] J. H. Lee. Basic governing equations for the flight regimes of aeroassisted orbital transfer vehicles. Technical Paper 84-1729, AIAA, Snowmass, Colorado, June 1984. 97

[82] B. V. Leer. Towards the ultimate conservation difference scheme. iv. a new approach to numercial convection. *Journal of Computational Physics*, (14):361–370, 1979. 62, 108, 119

[83] B. V. Leer. Flux vector splitting for the euler equations. In *8th International Conference in Numerical Methods in Fluid Dynamics*. Springer Verlag, 1982. 110

[84] B. V. Leer, W. Lee, and K. Powell. Sonic point capturing. In *AIAA Paper 89-1945-CP*, Buffalo(NY), 1989. 9th AIAA Computational Fluid Dynamics Conference. 113

[85] I. Lepot. *A parallel high-order implicit finite volume method for three-dimensional inviscid compressible flows on deforming unstructure meshes*. PhD thesis, University of Liege, Faculty of Applied Sciences, 2004. 121

[86] R. J. LeVeque. *Numerical Methods for Conservation Laws*. Birkhäuser, 1990. 108, 131

[87] M. S. Liou. A further development of the ausm$^+$ scheme towards robust and accurate solutions for all speeds. In *16th AIAA Computational Fluid Dynamics Conference*, Orlando, Florida, June 2003. AIAA 2003-4116. 110, 118

[88] M. S. Liou. A sequel to ausm, part ii: Ausm$^+$-up for all speeds. *Journal of Computational Physics*, 214:137–170, 2006. 115, 117, 118

[89] M. S. Liou. A sequel to ausm: Ausm+. *Journal of Computational Physics*, 129:363–382, 1996. 117

[90] M. S. Liou and C. J. J. Steffen. A new flux splitting scheme. *Journal of Computational Physics*, 107:23–39, 1993. 110, 115, 116

[91] Y. Liu and M. Vinokur. Upwind algorithms for general thermochemical nonequilibrium flows. In *27th AIAA Aerospace Sciences Meeting*, Reno, Nevada (US), Jan 1989. AIAA 89-0201. 111

[92] Y. Liu, D. Prabhu, K. A. Trumble, D. Saunders, and P. Jenniskens. Radiation modeling for the reentry of the stardust sample return capsule. In *AIAA Paper 2008-1213*, Reno (Nevada), Jan 2008. 46th AIAA Aerospace Sciences Meeting and Exhibit. 179, 180

[93] M. MacLean, M. Holden, T. Wadhams, and R. Parker. A computational analysis of thermochemical studies in the lens facilities. In *45th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, Nevada (US), Jan 2007. AIAA 207-121. 159, 175

[94] T. Magin. A model for inductive plasma wind tunnels. Ph.d. thesis, von Karman Institute for Fluid Dynamics, St.-Genesius-Rode, Belgium, June 2004. 6, 77, 81, 97, 133, 208, 219

[95] T. Magin and G. Degrez. Transport algorithms for partially ionized and unmagnetized plasmas. *Journal of Computational Physics*, 198: 424, 2004. 97

[96] J. Majewski and A. Athanasiadis. Anisotropic solution-adaptive technique applied to simulations of steady and unsteady compressible flows. In *Computational Fluid Dynamics 2006*, Ghent, 2006. ICCFD4, Springer. to appear in 2008. 224

[97] P. Marrone and C. E. Treanor. Chemical relaxation with preferential dissociation from excited vibrational levels. *Physics of Fluids*, 6(9), 1963. 99

[98] S. Meyers. *Effective C++ CD - 85 ways to improve your programs and designs*. Addison-Wesley, 1999. 13

[99] M. Mezine, M. Ricchiuto, R. Abgrall, and H. Deconinck. Monotone and stable residual ditribution schemes on space-time elements for unsteady conservation laws. In *VKI LS 2003-05, 33rd Computational Fluid Dynamics Course*, Rhode Saint Genese, 2003. 139

[100] R. C. Millikan and D. R. White. Systematics of vibrational relaxation. *Journal of Chemical Physics*, 39(12):3209–3213, 1963. 86

[101] J. Molnar. Numerical simulation of hypersonic reentry flows in chemical equilibrium and nonequilibrium using an object-oriented solver. Diploma course report, Von Karman Institute for Fluid Dynamics, 2006. 221

[102] A. B. Murphy. Diffusion in equilibrium mixtures of ionized gases. *Physical Review E*, 48:3594, 1993. 81

[103] J. Muylaert and L. Walpot. Preparing for reentry with expert: the esa in flight atd research program. Bremen, Germany, 2003. 54th International Congress of the International Astronautical Federation, the International Academy of Astronautics, and the International Institute of Space Law. 188

[104] G. Nelissen and P. F. Vankeirsbilck. Electrochemical modelling and software genericity. In *Modern Software Tools for Scientific Computing*. Birkhäuser, 1997. 37

[105] I. Nompelis. *Computational Study of Hypersonic Double-Cone Experiments for Code Validation*. PhD thesis, University of Minnesota, May 2004. 97, 115, 158, 165, 168, 171

[106] I. Nompelis and G. V. Candler. Computational investigation of high enthalpy flows past a finite cylinder in heg. In D. E. Zeitoun, J. Periaux, J. A. Désidéri, and M. M. (Eds.), editors, *West East High Speed Flow Fields*, CIMNE, Barcelona, Spain, 2002. 172

[107] I. Nompelis and G. V. Candler. Investigation of hypersonic double-cone flow experiments at high enthalpy in the lens facility. In *45th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, Nevada (US), Jan 2007. AIAA 207-203. 159

[108] I. Nompelis, G. V. Candler, and M. S. Holden. Effect of vibrational nonequilibrium on hypersonic double-cone experiments. *AIAA Journal*, 41(11), November 2003. 159, 165

[109] I. Nompelis, T. W. Drayna, and G. Candler. A parallel unstructured implicit solver for hypersonic reacting flow simulation. In *AIAA Paper 2005-4867*, Toronto (Canada), June 2005. 17th AIAA Computational Fluid Dynamics Conference. 57, 219, 223

[110] H. F. O. Knab and S. Jonas. Multiple temperature description of reaction rates constant with regard to consistent chemical-vibrational

coupling. Conference Paper 92-2947, AIAA, Nashville, TN, July 1992. 99

[111] S. Osher and F. Solomon. Upwind difference schemes for hyperbolic systems of conservation laws. *Math. Of Computation*, (38):339–374, 1982. 110

[112] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997. 57, 58

[113] H. Paillere. *Multi-dimensional Upwind Residual Distribution Schemes for the Euler and Navier-Stokes Equations on Unstructured Grids.* PhD thesis, Universit*é* Libre de Bruxelles, 1995. 139, 140, 150

[114] M. Pandolfi. A contribution to the numerical prediction of unsteady flows. *AIAA J.*, 22(5):602–610, 1984. 110

[115] M. Pandolfi and D. D'Ambrosio. Numerical instabilities in upwind methods: analysis and cures for the c̈arbunclep̈henomenon. *Journal of Computational Physics*, (166):271–301, 1979. 113

[116] M. Panesi. *Physical models for nonequilibrium plasma flow simulations at high speed re-entry conditions.* PhD thesis, Universit*á* di Pisa, Von Karman Institute of Fluid Dynamics, 2008. 6, 77, 97, 219, 224

[117] M. Panesi, A. Lani, T. Magin, F. Pinna, O. Chazot, and H. Deconinck. Numerical investigation of the non equilibrium shock-layer around the expert vehicle. In *AIAA Paper 2007-4317*, Miami (Florida), Jun 2007. 38th AIAA Plasmadynamics and Lasers Conference. 131

[118] C. Park. *Nonequilibrium Hypersonic Aerothermodynamics.* John Wiley and Sons, New York, 1989. 84, 133

[119] C. Park. Assessment of a two-temperature kinetic model for dissociating and weakly ionizing nitrogen. Technical Paper 86-1347, AIAA, Boston, Massachussets, June 1986. 90, 99

[120] C. Park. Review of chemical-kinetic problems of future nasa mission, I: Earth entries. *Journal of Thermophysics an Transfer*, 7:385–398, July-Sept 1993. 86, 100, 172, 180, 194

[121] C. Park, J. T. Howe, and R. L. Jaffe. Review of chemical-kinetic problems of future nasa mission, II: Mars entries. *Journal of Thermophysics an Transfer*, 8(1):9–23, January-March 1994. 99

[122] A. Patel. *Développement dún solveur RANS adaptatif sur maillages non-structurés hexaédriques.* PhD thesis, Universitè Libre de Bruxelles, Faculté des Sciences Appliquèes, February 2003. 126, 127, 223

[123] S. Phongthanapanich and P. Dechaumphai. Modified multidimensional dissipation scheme on unstructured meshes for high-speed compressible flow analysis. *Journal of Computational Fluid Dynamics*, 18 (8):631–640, 2004. 113

[124] F. Pinna. Numerical simulation of thermal non equilibrium effects in hypersonic flow. Diploma course report, Von Karman Institute for Fluid Dynamics, 2007. 221

[125] F. Pinna. Numerical study on roughness induced transition from low to high mach number. annual PhD presentation, April 2008. 204, 205, 206, 221

[126] J. P. Pinto. Spalart allmaras turbulence model implementation in the coolfluid framework. Diploma course report, Von Karman Institute for Fluid Dynamics, 2006. 205, 221

[127] R. K. Prabhu. An implementation of a chemical and thermal nonequilibrium flow solver on unstructured meshes and application to blunt bodies. Technical Report NASA-CR-194967, NASA Langley Center, Aug 1994. 111, 112

[128] A. V. S. Prasad. *Numerical simulation of hypersonic flows and associated systems in chemical and thermal nonequilibrium.* PhD thesis, Dept. of Fluid Mechanics, Vrije Universiteit Brussel, Brussels, Belgium, Feb 1997. 85, 87, 88, 89, 97, 99, 132, 133, 219

[129] V. Prokop. Developement of a local thermodynamic equilibrium model of inductively coupled plasma. Diploma course report, Von Karman Institute for Fluid Dynamics, 2007. 208, 221

[130] C. Prudh́omme. A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations. *Scientific Programming*, 14(2):81–110, 2006. 2

[131] T. Quintino. *A Component Environment for High-Performance Scientific Computing. Design and Implementation.* PhD thesis, Katholieke Universiteit Leuven, 2008. 3, 5, 14, 39, 55, 56, 216, 217, 222

[132] T. Quintino and A. Lani. Method-command-strategy pattern: A multi-component solution for high-performance scientific computing. In *POOSC'05 Workshop Notes*, Glasgow, Jul 2005. Von Karman Institute. 38, 39, 216

[133] J. Quirk. A contribution to the riemann solver debate. *Int. J. Num. Meth. in Fluids*, (18):555–574, 1994. 112

[134] T. S. R. Vilsmeier and N. Stuntz. A c++ library for finite volume simulations, 2002. URL `http://www.vug.uni-duisburg.de/MOUSE`. 37

[135] M. Rasquin. Object-oriented unstructured grid solver for high-enthalpy flows in local thermodynamic equilibrium. Diploma course report, Von Karman Institute for Fluid Dynamics, 2005. 221

[136] C. M. Rhie and W. L. Chow. Numerical study of the turbulent flow past an isolated airfoil with traling edge separation. *AIAA Journal*, 21(11):1525–1532, 1982. 209

[137] M. Ricchiuto. Space time residual distribution schemes and application to two-phase flow computation on unstructured meshes. Dc report, Von Karman Institute for Fluid Dynamics, Rhose Saint Genese, Belgium, 2001. 139, 145

[138] M. Ricchiuto. *Construction and Analysis of Compact Resdidual Discretizations for Conservation Laws on Unstructured Meshes*. PhD thesis, Universit*é* Libre de Bruxelles, 2005. 139, 145

[139] C. Riga. Numerical simulation of diffusion flames close to local thermodynamic equilibrium. Diploma course report, Von Karman Institute for Fluid Dynamics, 2008. 219, 221

[140] P. Rini. *Analysis of differential diffusion phenomena in high enthalpy flows*. PhD thesis, von Karman Institute for Fluid Dynamics, Rhode-St-Genese, Belgium, March 2006. 6, 82, 133, 208, 219

[141] P. Rini, D. Vanden Abeele, and G. Degrez. Closed form for the equations of chemically reacting flows under local thermodynamic equilibrium. *Physical Review E*, 71(5), 2005. 81, 82, 193

[142] P. Roe. Approximate riemann solvers, parameters vectors and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981. 110

[143] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986. 107

[144] R. Sanders, E. Morano, and M.-C. Druguet. Multidimensional dissipation for upwind schemes: stability and applications to gas dynamics. *Journal of Computational Physics*, 145(CP986047):511–537, 1998. 111, 112, 190

[145] G. S. R. Sarma. Physico-chemical modeling in hypersonic flow simulation. *Progress in Aerospace Sciences*, pages 281–349, 1958. 80

[146] V. Selmin. Artificial dissipation methods for euler equations. Multidisciplinary Computation and Numerical Simulation. 118, 220

[147] T. K. Shi and W. H. Steeb. *Symbolic C++: An Introduction To Computer Algebra Using Object-Oriented Programming.* Springer, 1998. 229

[148] J. L. Steger and R. F. Warming. Flux vector splitting of the inviscid gas-dynamics equations with applications to finite diffference methods. *Journal of Computational Physics*, 40(2):263–293, 1981. 110, 114

[149] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition edition, Mar. 2000. ISBN 0-201-70073-5. 13, 15, 16, 44, 229

[150] H. Sutter. *More exceptional C++. 40 new engineering puzzles, programming problems, and solutions.* Addison-Wesley, Dec. 2002. ISBN 0-201-70434-X. 15, 16

[151] K. Sutton and P. A. Gnoffo. Multi-component diffusion with application to computational aerothermodynamics. Technical Paper 98-2575, AIAA, Albuquerque, New Mexico, June 1998. 98

[152] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics.* Springer, 1997. 108, 109, 124

[153] K. A. Trumble, I. Cozmuta, S. Sepka, and P. Jenniskens. Post-flight aerothermal analysis of the stardust sample return capsule. In *AIAA Paper 2008-1201*, Reno (Nevada), Jan 2008. 46th AIAA Aerospace Sciences Meeting and Exhibit. 179, 180, 186, 187

[154] E. van der Weide. *Compressible Flow Simulation on Unstructured Grids using Multi-dimensional Upwind Schemes.* PhD thesis, Delft University of Technology, Netherlands, 1998. 57, 139, 140, 149, 153, 206, 221

[155] E. van der Weide. A parallel implicit multidimensional upwind cell vertex navier-stokes solver for aerothermodynamic applications. development of parallel algorithms for aerothermodynamic applications. Cr 1996-35, Von Karman Institute for Fluid Dynamics, Rhode Saint Genese, Belgium, June 1996. 79, 151

[156] E. van der Weide and H. Deconinck. Matrix distribution schemes for the system of euler equations. In H. Deconinck and B. Koren, editors, *Notes on Numerical Fluid Mechanics*, volume 57, pages 113–140. Vieweg, Braunschweig, 1997. 139, 146

[157] E. van der Weide, K. Sermeus, and H. Deconinck. Thor v.0 and v.1.0. basic euler solver. Tr 2.1/2.2/2.3, von Karman Institute for Fluid Dynamics, 1998. 147, 149, 221

[158] E. van der Weide, H. Deconinck, E. Issmann, and G. Degrez. A parallel implicit multidimensional upwind residual distribution method for the navier-stokes equations on unstructured grids. In *Comp. Mech*, volume 23, pages 199–208, 1999. 139

[159] D. Vanden Abeele and G. Degrez. Efficient computational model for inductive plasma flows. *AIAA Journal.*, 38(2):234–242, 2000. 98

[160] D. Vandevoorde and N. M. Josuttis. *C++ Templates.* Addison-Wesley, 2003. ISBN 0-201-73484-2. 13, 53, 229, 230, 234

[161] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995. 229

[162] T. Veldhuizen. Using c++ template metaprograms. *C++ Report*, 7 (4):36–43, 1995. 229, 234

[163] V. Venkatakrishnan. Convergence to steady state solutions of the euler equations on unstructured grids with limiters. *Journal of Computational Physics*, 118:120–130, 1995. 123

[164] N. Villedieu. High order rds discretization for residual distribution schemes. annual PhD presentation, April 2008. 224

[165] W. Vincenti and C. Kruger. *Introduction to Physical Gas Dynamics*. John Wiley and Sons, New York, 1965. 84

[166] *Non-equilibrium gas dynamics, from physical models to hypersonic flights (RTO-AVT-VKI)*, Sep 2008. Von Karman Institute. 90

[167] N. P. Waterson and H. Deconinck. Design principles of bounded higher-order convection schemes - a unified approach. *JCP*, 224:182–207, 2007. 124

[168] M. J. Wright, G. V. Candler, and D. Bose. Data-parallel line relaxation method for the navier-stokes equations. *AIAA Journal*, 36(9):1603–1609, 1998. 84, 219

[169] T. Wuilbaut. *Multi-Physics*. PhD thesis, Université Libre de Bruxelles, 2008. 106, 205, 206, 207, 208, 217, 221, 222, 225, 226

[170] M. Yalim, D. V. Abeele, and A. Lani. Simulation of field-aligned ideal mhd flows around perfectly conducting cylinders using an artificial compressibility approach. In *Proc. of the 11th International Conference on Hyperbolic Problems*, pages 1085–1092, Lyon, France, July 17-21 2006. Ecole Normale Supérieure, Springer-Verlag. 202

[171] M. S. Yalim. Efficient object-oriented implementation of a finite volume flow solver. Diploma course report, Von Karman Institute for Fluid Dynamics, 2004. 221

[172] S. M. Yalim. *An artificial compressibility approach for solving the compressible ideal magnetohydrodynamics equations and application to space weather simulation*. PhD thesis, Université Libre de Bruxelles, 2008. 106, 202, 203, 204, 217, 221

[173] J. M. Yos. Approximate equations for the viscosity and translational thermal conductivity of gas mixtures. Contract Report AVSSD-0112-67-RM, Avco Corporation, Wilmington, Massachusetts, April 1967. 97

[174] M. Zaloudek. Large shape optimization of turbulent transonic internal flows. Diploma course report, Von Karman Institute for Fluid Dynamics, 2007. 205, 221

List of Acronyms

Each entry is followed by the list of the pages where it is refered.

Application Programming Interface (API)

A set of functions, procedures, methods or classes that an operating system, library or service provides to support requests made by computer programs. [3, 5]

Computer Aided Design (CAD)

Use of computer technology to aid in the design and especially the drafting (technical drawing and engineering drawing) of a part or product. [19]

Computational Fluid Dynamics (CFD)

One of the branches of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. [xi, 1–3, 6, 25, 27, 37, 39, 57, 84, 107, 157, 158, 201, 222]

Chemical Nonequilibrium (CNEQ)

A gas mixture can be considered in chemical nonequilibrium if the relaxation times of the chemical processes occurring in the flow are comparable to the characteristic time of the macroscopic fluid dynamic phenomena. [190, 193]

Conservative Residual Distribution (CRD)

A strictly conservative variant of the RD method. [vi, 6, 139, 145, 148, 168, 221, 224]

Finite Element Method (FEM)

A discretization method for PDE's based on weak formulation. [24, 27, 63, 149, 216, 217, 225]

Finite Volume (FV)

A numerical method for discretizing PDE's written in integral form. [v, 6, 24, 43, 68–72, 80, 159, 162, 201, 202, 205, 206, 215–218, 220, 221, 223–225]

GeometricEntity (GE)

Object representing a geometric entity in COOLFluiD. [19–21, 23, 24]

High Performance Computing (HPC)

The use of supercomputers and computer clusters to solve advanced computing problems. [v, 67, 222]

Inductively Coupled Plasma (ICP)

Plasma generated by magnetic induction. [105, 208, 209, 219, 220]

Linear Residual Distribution (LRD)

The original variant of RD method based on a multidimensional Roe linearization. [145, 148, 221]

Local Thermodynamic Equilibrium (LTE)

A gas mixture can be considered in LTE if both the relaxation times of the energy transfer among different modes and of the chemical processes occurring in the flow are much smaller than the characteristic time of the macroscopic fluid dynamic phenomena. [34, 81, 82, 94, 95, 181, 183, 189, 190, 209, 219]

Method-Command-Strategy (MCS)

Design pattern to implement numerical algorithms in COOLFluiD. [37–39, 48, 49, 134, 220]

Magnetohydrodynamics (MHD)

The academic discipline which studies the dynamics of electrically conducting fluids, modeled as a combination of the Navier-Stokes equations of fluid dynamics and Maxwell's equations of electromagnetism. [201, 202, 217, 221]

Message Passing Interface (MPI)

A specification for an API that allows many computers to communicate with one another. It is used in computer clusters and supercomputers. [11, 15, 18, 57, 67, 68, 107, 215]

Object-Oriented (OO)

is a programming paradigm that models the entities within a computer program as *objects* and their interactions. An object contains encapsulated data and functions grouped together to represent an entity. The way to interact with the object is defined via its *interface*. [1, 2, 5, 9–11, 25, 27, 37, 215]

Partial Differential Equation (PDE)

A type of differential equation involving partial derivatives in space and/or time of the unknown functions. [v, 5, 9, 14, 24, 25, 37, 39, 77, 79, 82, 103, 108, 139, 202, 215, 216, 219]

Residual Distribution (RD)

A discretization method for PDE's based on weak formulation especially designed for compressible flows. [v, vi, 6, 27, 57, 63, 68, 69, 71, 72, 80, 139–141, 143–145, 147, 153, 159, 162, 201, 215, 217, 218, 220, 221, 224]

Thermo-Chemical Nonequilibrium (TCNEQ)

A gas mixture can be considered in thermodynamic and chemical nonequilibrium if both the relaxation times of the energy transfer among different modes and of the chemical processes occurring in the flow are comparable to the characteristic time of the macroscopic fluid dynamic phenomena. [34, 91, 93, 100, 190, 193]

TopologicalRegion (TR)

Object representing a subset of a TRS typically linked to a specific CAD definition. [19, 20]

TopologicalRegionSet (TRS)

Object representing a topological surface or portion of the domain in COOLFluiD. [19, 20, 23, 24, 65, 66]

von Karman Institute for Fluid Dynamics (VKI)

NATO institute in Belgium focused on research in fluid dynamics[5]. [xi, xii, 3, 5, 6, 157, 208, 219]