Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Final Master Thesis (FMT)

2015-2016 | Autumn Semester

Master in Innovation and Research in Informatics (MIRI)

High Performance Computing (HPC)

Extending SAIPH

Simulating fluid mechanics and chemistry problems

Sandra Macià Sorrosal
(sandra.macia@bsc.es)

Director: **Vicenç Beltran Querol** (vbeltran@bsc.es)
Co-director: **Eduard Ayguadé Parra** (eduard@ac.upc.edu)

Computer Architecture Department (DAC)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

”Well, Mr. Frankel, who started this program, began to suffer from the computer disease that anybody who works with computers now knows about. It’s a very serious disease and it interferes completely with the work. The trouble with computers is you *play* with them. They are so wonderful. You have these switches - if it’s an even number you do this, if it’s an odd number you do that - and pretty soon you can do more and more elaborate things if you are clever enough, on one machine.

After a while the whole system broke down. Frankel wasn’t paying any attention; he wasn’t supervising anybody. The system was going very, very slowly - while he was sitting in a room figuring out how to make one tabulator automatically print arc-tangent X, and then it would start and it would print columns and then bitsi, bitsi, bitsi, and calculate the arc-tangent automatically by integrating as it went along and make a whole table in one operation.

Absolutely useless. We *had* tables of arctangents. But if you’ve ever worked with computers, you understand the disease - the *delight* in being able to see how much you can do. But he got the disease for the first time, the poor fellow who invented the thing.”

– Richard Feynman, Surely You’re Joking, Mr. Feynman!: Adventures of a Curious Character

Acknowledgments

I would like to thank my supervisor Vicenç Beltran for the useful comments, remarks and engagement through the learning process of this master thesis and for giving me the opportunity to work on this exciting and engaging project.

Many thanks also to Sergi and Alejandro for introducing me to the topic as well as for the great support you gave me on the way.
To BSC's CASE department, specially to Daniel Mira for helping me on the theoretical scientific view of the project.
In general, thanks to all my colleagues at BSC.

Special thanks to Marc for being always there, believing in me more than I do. You still keep surprising me with your physicist and global view advices.

Last but not least, I want to thank my family and friends to make this work possible.

Extending Saiph: Simulating fluid mechanics and chemistry problems

Sandra Macià Sorrosal

Computer Architecture Department (DAC)
Universitat Politècnica de Catalunya (UPC)
2016

ABSTRACT

Nowadays, High-Performance Computing (HPC) is assuming an increasingly central role in scientific research. It is becoming frequent to see scientists working in supercomputing environments while computer architectures are becoming more and more heterogeneous and complex with different parallel programming models and techniques. Under this scenario, the only way to successfully exploit an HPC system requires that computer and domain scientists work closely towards producing applications to solve domain problems, ensuring productivity and performance at the same time.
Facing such purpose, Saiph is a Domain Specific Language (DSL) designed to ease the task of simulating complex Partial Differential Equation systems (PDEs) that model real physical phenomena, freeing the users from numerical methods and high-performance complexities.
This project aims to extend Saiph to support fluid mechanics and chemistry -combustion- simulations, two domains with a high interest within the scientific community. Driven by use-cases requirements, extensions have been performed at different levels of abstraction. New user-functionalities, suitable numerical methods and specific domain optimizations have been added leading to validated simulation results of the selected use-cases, obtained through the parallel execution of such Saiph applications.

Keywords: HPC, DSL, PDEs, Fluids, Combustion, Scala, Compilers, Parallelism.

Table of Contents

1 | Introduction

Today, synergy is one of the most important requirements for research and scientific advances. In any domain, it is necessary to work closely with different domain experts with different backgrounds and views but ideally binding them in its own domain of expertise.

For such purpose, Domain Specific Languages (DSLs) are becoming a popular approach. Ideally, domain experts would just need to specify their problems unambiguously, with all the details required to formalize the whole simulation description and the domain-specific compiler would handle the details related to the domain specific problem and the high-performance execution in a supercomputer. And that is exactly what Saiph is meant for; Saiph is a Domain Specific Language being developed at the Barcelona Supercomputing Center (BSC) for simulating physical phenomena modeled by complex Partial Differential Equations (PDE) systems. Saiph eases the development of scientist applications by allowing domain experts to transcribe their equations into Saiph code and then generating HPC-ready code that efficiently exploits the computational resources of modern heterogeneous supercomputers while dealing with all the specific aspects of solving partial differential equations systems.

1.1 Motivation

In supercomputing environments, scientists model real-world phenomena using partial differential equation systems. Afterwards, these models are transcribed into a programming language in order to numerically estimate the solutions of the equations via simulation. This scenario requires a strong collaboration between computer experts, numerical methods experts and scientists lighting up the fact that scientist could not perform as computer (or numerical methods) experts and vice versa. Without a strong collaboration, domain experts are forced to leave their domain of expertise and productivity and efficiency start to plummet. Consequently the productivity can be hindered by issues related to domains that are, in fact, adjacent to the scientific domain of the specific research. The Saiph project is motivated by the fact that regarding simulations involved in science, the unavoidable and essential adjacent domains are the computer science domain and the numerical method domain, which are fields of expertise by

themselves. Domain Specific Languages could, in general, separate and establish the link between those different but dependent domains, allowing the work to progress in the same direction while each expert working in its own domain of expertise. Saiph, in particular, has been designed for users which are not familiar with PDEs resolution, nor numerical methods neither programming for supercomputers. Saiph is a high-level language with domain specific syntax that provides to the users, numerical correctness, specific domain optimizations and high performance computing. However, Saiph is a tool under development and until this project, it has successfully faced the resolution of few and small systems.

The idea behind Saiph is strong and powerful, so Saiph could have a long and exciting trajectory. For this reason, it is important and necessary to validate and use what it has been done until now as well as to extend it to demonstrate its strength to the scientific community.

Given the users targeted by the tool, it appears interesting to have a development process driven by real large, complex and popular use-cases with their specific requirements. To this end, it seems wise to choose a specific scientific domain, wide and popular enough to have several use-cases for conducting the development and the extension of Saiph as a specific language of the domain of resolutions and simulations of physical problems written as PDEs systems.

1.2 Objectives

The main objective of this project is to make real use of Saiph and to extend it. To this end, we have chosen a wide enough scientific domain able to provide us with real applications of interest to drive the development of our tool. This scientific domain corresponds to fluid mechanics and chemistry.

For quite awhile, fluid mechanics and chemistry has been a domain of high interest in the scientific community. As a fact of matter, this domain of expertise can provide us with use-cases as systems of partial differential equations which lead to real, interesting and complex scientific simulations. Moreover, the popularity of the field allows us to be able to validate the obtained results of the simulated systems against other.

Thus, the objective is twofold; make real use of Saiph and extend this domain specific language for the resolution of problems related to fluid mechanics and chemistry offering a high level syntax that directly maps with a concept of the domain, hiding from the user all the complexities related to numerical methods, domain specific optimizations and the generation of specific code to be executed in parallel by a supercomputer, distributing the work across several nodes and applying intra-node techniques to achieve a high performance.

1.2.1 Detailed Objectives

In particular, the main objective is to develop and extend Saiph to end up with the simulation of the real application *Premixed flame* which is a combustion phenomena and an ideal use-case to drive the development of Saiph since it includes the basic principles of fluid mechanics and chemistry systems. The steps to achieve that, were set as follows:

Fluid mechanics applications

Extending Saiph to be able to simulate and validate basic applications of fluid mechanics. In particular *convection* phenomena and the *Sod's shock tube* application.

Chemistry applications

Extending Saiph to be able to simulate and validate basic applications of chemistry. In particular the *autoignition delay time* test.

Fluid mechanics and chemistry applications

Extending Saiph to be able to simulate and validate applications of fluid mechanics and chemistry. In particular, the *premixed flame* application.

1.3 Context

The current project starts with the complete Saiph DSL infrastructure already set and at an advanced stage of development. Within that context, the objectives are to extend and keep developing the tool, in order to enlarge its applicability and numerical safety. This project is being developed at the Computer Science (CS) department of the BSC, by the DSLs group, in collaboration with the Computer Applications in Science & Engineering (CASE) department, for the *Repsolver* project. The current project exploits the natural and required synergy between the two departments and the Repsol entity: CS acting as computer science experts, CASE contributing with numerical methods expertise and Repsol providing real use-cases of interest.

1.4 Document structure

The rest of this document is structured as follows: Chapter 2 gives a briefing over the state of the art on the field of DSLs for HPC for solving PDE systems of fluid mechanics

and chemistry problems. Following on, Chapter 3 then introduces Saiph as it was before the current project which includes all the tools used to develop and extend it, the design of the DSL, the syntax of the Saiph language, how to use it and the internal features hidden for the users. Then, Chapter 4 focus on fluid mechanics and chemistry theory, followed by a methodology overview, in Chapter 5. Chapter 6 nails down the most important features implemented to extend Saiph including new functionalities, application of specific numerical methods and domain specific optimizations. Finally, use-cases' results are reported and evaluated in Chapter 7 together with a brief scalability analysis. Chapter 9 concludes with some remarks regarding the integration of new features in an already functional tool as well as with the promising trajectory a tool as Saiph can have in the scientific community. To end the document, Appendix C contains the complete Saiph's code of the *premixed flame* use-case reported.

2 | State of the art

Saiph is not the first tool designed to provide an abstraction for solving PDE systems. This section comments on some of the already existing approaches to solve large-scale PDE systems using DSLs. We describe two different tools, emphasizing the differences with Saiph to illustrate its potential.

2.1 Liszt

Liszt is a Scala-based domain-specific language for solving partial-differential equations on meshes [1]. The language is designed for code portability across heterogeneous platforms. Similar to Saiph, Liszt applications are translated to an intermediate representation which is then compiled by the Liszt compiler to generate native code for multiple platforms. The aim of Liszt is to exploit information about the structure of the data and the nature of the algorithms in the code and to apply aggressive and platform specific optimizations.

Liszt provides features for parallelism. These semantics should be applied by the users to ensure that the Liszt compiler can infer data dependencies automatically, enabling it to generate a parallel implementation for code written in a serial style.

Regarding usability, the difference between Saiph and this approach is essentially the level of abstraction exposed to the user. Liszt assume that the governing PDEs are already discretized on the mesh over a region of space, on the other hand, Saiph provides a one-to-one mapping from the formal specification of a PDE system to actual code. This enables Saiph' users to perfectly use the language efficiently and without any significant knowledge on the actual execution details.

On the other hand, Liszt comes with its own runtime system that optimize all the operations performed on the geometry. Liszt components and optimizations are completely ad-hoc, not reusable for any other similar DSL and not composable with other libraries. In contrast, Saiph has been build by layers on top of fully reusable infrastructure, so optimizations and functionalities are reused.

2.2 FEniCS

Another approach relies into expressing the PDEs at the mathematical level. FEniCS [2] use, as Saiph, a top level of abstraction defining a high-level language for the specification of finite elements algorithms allowing the users to express the problem in terms of differential equations, leaving the details of the parallel implementation to a lower-level library.

The FEniCS Project [3] is a collaborative project for the development of innovative concepts and tools for automated scientific computing, with a particular focus on automated solution of differential equations by finite element methods. The goals of Saiph are similar to the FeniCS ones; FEniCS aims to set a new standard in Computational Mathematical Modeling (CMM), which can be described as the Automation of CMM, towards the goals of generality, efficiency, and simplicity, concerning mathematical methodology, implementation, and application.

FEniCS has an extensive list of features for automated, efficient solution of differential equations, including automated solution of variational problems, automated error control and adaptivity, a comprehensive library of finite elements, high performance linear algebra and many more. It is organized as a collection of interoperable components that together form the FEniCS Project. These components include a problem-solving environment, a form compiler, a finite element tabulator, a just-in-time compiler, a code generation interface, a form language and a range of additional components. Building on these components, software specialized to solving different problems are organised into separate applications.

Although FEniCS is a very powerful solution for a large number of complex problems, it is not at all meant to be used by scientists without deep expertise in numerical methods. Therefore, even if it provides a complete simulation infrastructure for many real-world problems, its target users essentially differ from Saiph's. In our case, we want a language that scientists can use to quickly simulate and integrate results in their research process, where the simulation is just a part of the whole. As a consequence, Saiph can make much more assumptions regarding the numerical methods used and the abstractions provided, resulting in a much more abstract syntax, close to any scientists with modest knowledge in scientific modeling.

3 | Saiph overview

This chapter introduces the design and the underlying technology used for developing Saiph and the resulting high-level language. We briefly describe the Saiph project and its state of development when the current project (aimed to use and extend Saiph) started. Hence, this is an overview of the tool as it was before this project.

3.1 Saiph design

Saiph, as a DSL, has been designed to be simple, efficient, largely applicable within a certain domain, and safe. At a macroscopic level, synergy between domain and DSL experts should be a must in the production chain.

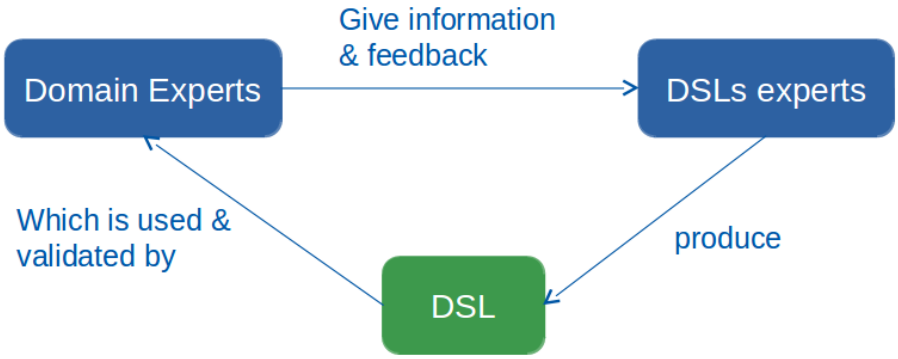

Fig. 3.1: DSL macroscopic production design

Saiph has been implemented as an embedded compiler in Scala[4] using the Lightweight Modular Staging (LMS)[5] as a DSL development platform and the Scala Virtualized Compiler[6].

Saiph applications are compiled with the LMS and the Saiph implementation together using the Scala Virtualized Compiler. After that, the output of this first phase is compiled using our embedded compiler. All the domain specific optimizations are applied at this point. Finally, the output of the embedded compiler (a C++ file) is compiled

and linked with a low-level C++ library developed at BSC that takes care of solving the applications in parallel. Figure 3.2 shows the whole compilation process and the internal design structured by layers:

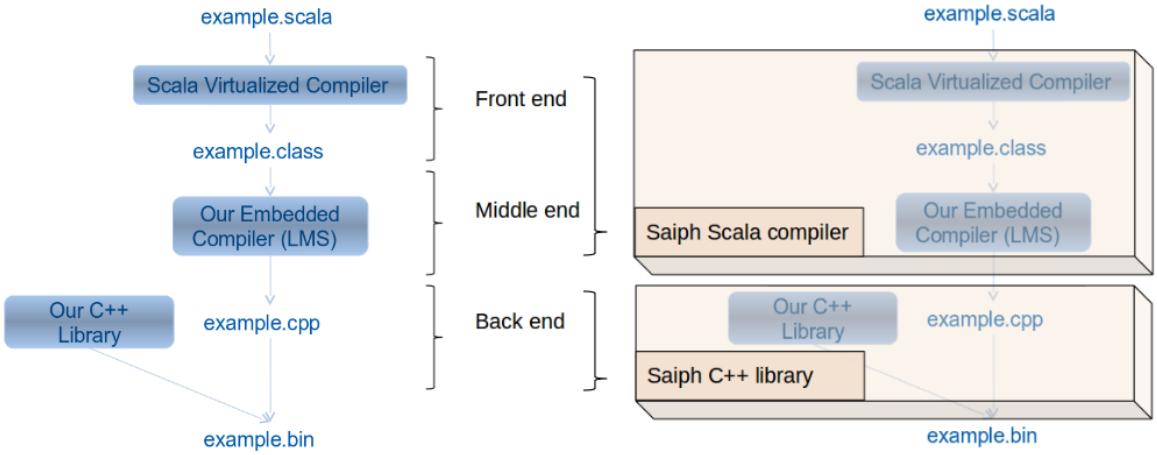

Fig. 3.2: Underlying design and technologies used during the compilation process

At the front end the Saiph application is compiled with LMS and all the compiler implementations together. At the middle end, the domain specific implementations are applied and the LMS generates the corresponding IR nodes. Finally, the back end compiles and links the generated C++ code using our C++ library and produces a binary ready to be executed in parallel using MPI and OpenMP parallelization.

Saiph is internally designed to have two important and separated layers: the Scala compiler and the C++ library. This separation eases the DSL development, as in each of them the efforts are devoted to the *developments* naturally belonging to the layer. In that way, as DSL developers, we can reuse knowledge and take the advantages offered by each tool being used at its natural layer.
High-level domain-oriented syntax and domain specific optimizations are thus implemented at the Scala compiler, while MPI and OpenMP parallelization and auxiliary (mainly numerical) methods should be developed in the C++ library.

3.2 Underlying technology

Since the language is embedded in Scala, it may be useful for users to become familiar with the most basic concepts of Scala. The following sections provide a quick overview of Scala as well as a short description of the main tools used for developing Saiph.

3.2.1 Scala

Scala [7] is a statically typed programming language which unifies and generalizes object oriented and functional programming. It provides a powerful set of mechanisms for composing, abstracting and adapting components. Those mechanisms allow the language to be extensible enough so that users can model their domains writing libraries that are used like they were built-in features provided by the language itself. Scala is thus an attractive language for embedding DSLs. Using Scala and LMS (explained later in this chapter), Saiph is embedded as an internal DSL. Therefore, Saiph applications are actually Scala applications.

The most basic concepts, constructions and features of Scala used in Saiph applications are introduced in appendix A. In this section we solely mention an interesting Scala feature used in Saiph to allow domain specific optimizations; pattern matching.

Pattern matching

One of the functional features that Scala implements is pattern matching. Scala allows the programmer to match values of any type with a match-first policy. However, Scala, as an object-oriented language, extends this concept for objects. This is achieved with a special kind of classes called case classes.

The following code illustrates how Function `findRoom` (line 1) takes a parameter `n` of type `Int`. This parameter is pattern-matched against several cases (lines 2-5). If `n` is equal to one of these values, the corresponding string is returned. Line 5 is the default pattern, which, if reached, it is always executed.

```
1 def findRoom(n: Int): String = n match {
2       case 103 => "Lab"
3       case 105 => "Dean"
4       case 104 => "Secretary"
5       case _ => "Empty"
6 }
```

As previously mentioned, Scala allows the programmer to pattern-match case classes. The definition of a case class is shown as follows:

```
1 case class Rectangle(x: Int, y: Int, w: Int, h: Int)
```

The only difference between case classes and regular Scala classes is that case classes come by default with a constructor with the same name as the case class. Regular scala classes cannot be pattern-matched.

The following code shows how case classes are pattern-matched.

```
1 def assertRectangle(r: Rectangle): String = r match {
2       case Rectangle(0, 0, _, _) => // When rectange is at (0,0)...
3       // More cases...
4       case Rectangle(_, _, w, h) => // Use w and h to...
5 }
```

This last code illustrates how case classes can be pattern-matched against values of their members (line 2). Values of case class members can be bound and used (line 4).

Case classes act like regular Scala classes in terms of class hierarchies. This means that an instance of a superclass can be matched against instances of case classes implemented as subclasses derived from a given type.

3.2.2 Lightweight Modular Staging

LMS [8] is a Scala library for dynamic code generation. Dynamic code generation is inherently more flexible because code can be specialized with respect to parameters only available at runtime. The concept of staging is based on the observation that many computations can naturally be separated into stages distinguished by frequency of execution or availability of information. The staging approach, although introduced initially as a set of compiler transformations, can be thought of as a method for embeddeding domain-specific languages [9]. By means of LMS, the Saiph compiler is embedded as a Scala application, and the Scala code is translated into C++ code when the LMS Scala application is run.

LMS is just a library, so a DSL application is compiled together with the DSL implementation, giving place to an executable code generator. Once the code generator is run, the DSL application gets translated.

Appendix B shows how to build and use a DSL using the Lightweight Modular Staging as DSL development platform.

3.2.3 Scala-virtualized compiler

The Scala-Virtualized compiler [6], offers a set of small extensions to the Scala language to provide even better support for hosting embedded DSLs. LMS allows programmers to stage any kind of computation resulting on an object of a particular type, however, the vanilla Scala compiler does not provide us with staging of control flow constructs such as `if-else` statements or loops. The functionality of staging control flow is achieved through this extension (Scala-Virtualized compiler). The control flow constructs are

compiled down into regular method calls. Thus, Scala-Virtualized extends the idea of virtualizing certain language features by defining them as method calls, so that they can be redefined within the language. Scala-Virtualized redefines most of Scala's expression sub-language in this way, enabling DSL implementations to give domain-specific meaning to core language constructs.

3.3 Saiph as a language

Saiph offers a high level syntax to unambiguously define a complete system of partial differential equations which models a physical phenomena. This system is solved, by Saiph, using finite differences and Euler's integration methods but those numerical methods are hidden to the user, as well as issues related to the generation of parallel code.

It follows a complete description of this new language with the components and constructions needed by the users to describe the complete physical systems and the simulations parameters.

3.3.1 Units

Units are unavoidable components of Saiph to force the users to provide the physical dimensions of variables of their system. Saiph internally validates that all operations within equations are valid, avoiding illegal operations such as adding two variables representing different magnitudes. To support this type-checking, Saiph computes the units of each expression: if x is a variable whose unit is Meters, the unit of the expression $x * x$ is Meters^2. Thus, a unit represents a physical magnitude with its dimensionality information. There are seven attributes that indicate the exponent of each fundamental magnitude: Length, Mass, Time, Electric Current, Temperature, Amount of Substance and Luminous intensity. Apart from those seven fundamental units there is the unit *Unitless* which is used to qualify dimensionless variables, constants or expressions. Despite the fact that is always recommended to specify the real units of the variables, in some specific cases it may be a tedious work. Using this unitless unit as a black box unit type may be helpful in some cases.

Aside from avoiding illegal operations, Saiph also keep away from issues related to the system of units used by the user. Internally, the numerical value of each magnitude is stored in the International System of Units, thus, different units can be correctly used and combined to represent the same magnitude. For instance, miles and kilometers can be used as dimensions of length variables, or Kelvins and Celsius degrees for temperatures.

Table 3.1 shows the predefined unit's set for each fundamental physical magnitude.

Magnitude	Units
Length	Micrometers, Millimeters, Centimeters, Meters, Kilometers
Mass	Tons, Kilograms, Grams
Time	Nanoseconds, Microseconds, Milliseconds, Seconds, Minutes, Hours, Days
Electric Current	Amperes
Temperature	Kelvins, Celsius, Fahrenheit
Amount Of Substance	Moles
Luminous Intensity	Candelas

Table 3.1: Predefined unit's set for each fundamental physical magnitude

Apart from the units in table 3.1, Saiph supports several operators to compare units and combine them to create new ones. The example below shows how to define a new unit using some of these operators and how to use it:

```
// Defining new units
def MetersPerSecond = Meters / Seconds
def Seconds2 = Seconds * Seconds
def MetersPerSecond2 = Meters / Seconds2

// Using a new unit: Gravity of Earth
def g = -9.81 * MetersPerSecond2
// Equivalent to
def g1 = -9.81 * Meters / Seconds2
// And also equivalent to
def g2 = -9.81 * Meters / Seconds / Seconds
```

For the sake of convenience, Saiph has a predefined set of combined units, which is shown in Table 3.2. However, this set is not fixed and it can be extended if needed.

Magnitude	Units
Area	Micrometers2, Millimeters2, Centimeters2, Meters2, Kilometers2
Volume	Micrometers3, Millimeters3, Decimeters3, Centimeters3, Meters3, Kilometers3
Energy	Joules
Pressure	Pascals, Bars, Atmospheres, Psis
Power	Watts
Speed	MetersPerSecond, KmPerHour

Table 3.2: Predefined combined unit's set

In order to combine and compare units, there is a specific set of operations between units that is supported by Saiph:

- **Comparison operators**

```
def infix_>(l: Unit, r: Unit) : bool
def infix_<(l: Unit, r: Unit) : bool
def infix_>=(l: Unit, r: Unit): bool
def infix_<=(l: Unit, r: Unit): bool
```

  The types of the arguments `l` and `r` must match. The result is always a boolean.

- **Add operator**

```
def infix_+(l: Unit, r: Unit) : Unit
```

  The type of Units of the arguments `l` and `r` must match. The result type is the same as the type of the arguments.

- **Minus operator**

```
def infix_-(l: Unit, r: Unit) : Unit
```

  The type of Units of the arguments `l` and `r` must match. The result type is the same as the type of the arguments.

- **Product operator**

```
def infix_*(l: Unit, r: Unit) : Unit
def infix_*(l: Scalar, r: Unit) : Unit
def infix_*(l: Unit, r: Scalar) : Unit
```

  The result type is the product of the argument's types.

- **Division operator**

```
def infix_/(l: Unit, r: Unit) : Unit
def infix_/(l: Scalar, r: Unit) : Unit
def infix_/(l: Unit, r: Scalar) : Unit
```

  The result type is the division of the argument's types.

3.3.2 Cartesian meshes

Saiph works with implicit cartesian meshes. Simulations take place inside a 3D cartesian mesh defined using the following constructor:

```
def Cartesian(xs: Length, ys: Length, zs: Length) : Cartesian
```

Where `xs`, `ys` and `zs` are the respective sizes of each direction X, Y and Z respectively. These sizes should be specified using continuous lengths, that is, the user should model the systems in physical continuous space.

The following example shows how to define a cartesian mesh:

```
val mesh = Cartesian(12.5 * Meters, 25.0 * Meters, 37.5 * Meters)
```

Once the mesh has been defined it must be discretized in order to map the physical coordinates to the actual discrete coordinates. This is done by the mandatory operation `discretize` whose parameters are real values representing the physical spacing between points in each of the dimensions. The following code shows an example on how to define and discretize a mesh:

```
val mesh = Cartesian(12.5 * Meters, 25.0 * Meters, 37.5 * Meters)
mesh.discretize(2.5 * Meters, 1.0 * Meters, 2.5 * Meters)
```

In the previous code, the discretized mesh has 6 points in the X axis (origin + 5 points), 26 in the Y axis (origin + 25 points) and 16 in the Z axis (origin + 15 points).

3.3.3 Terms

Saiph offers two types of components to represent dimensional constants and variables of the problem being simulated. A `Term` represents a variable whereas a `ConstTerm` is used to act for constant values of the problem. The main difference between a `Term` and a `ConstTerm` is how data is allocated: since `ConstTerms` are used when the data does not vary over time neither over space, it shares the same data along all the points of the mesh without being modified at any time. On the other hand, a `Term` privatizes the data for each point of the mesh and this data is conveniently updated.

Saiph supports scalar and vector terms, allocating the requested data (a scalar or a vector) for each point of the mesh. The following code shows how to declare a scalar and a vector term:

```
val T1 = Term(Temperature)("Temp1", mesh, 300 * Kelvins)
val T2 = Term(Temperature)("Temp2", mesh,
          Vector(0 * Kelvins, 300 * Kelvins), List("Gas1", "Gas2"))
```

In the example above, the two terms defined have magnitude `Temperature`. The first term T1 is a scalar term whereas T2 is a vectorial one. In this example, terms are initialized uniformly for each point of the mesh, but it is also possible to provide a function over space specifying how the values vary over the point position of the mesh:

```
val T3 = Term(Temperature)("Temp3", mesh,
          { x => if (x < XSIZE/2) 0 * Kelvins else 300 * Kelvins})
```

In this last example T3 is initialized depending on the X axis position.

Scalar and vector constant terms are also supported. Instead of allocating the requested data for each point of the mesh, a `ConstTerm` share the same data. As a consequence, a `ConstTerm` do not depend on the mesh and their memory consumption is lower than a `Term`. Constant terms can be declared as follows:

```
val T4 = ConstTerm(Temperature)("Temp4", 300 * Kelvins)
```

Because of the nature of a constant term, this component can not be initilized with spatial functions.

Both components, `Term` and `ConstTerm` are aimed to be combined, through operators, with other ones to build the equation system of the problem.

3.3.4 Operators

The operators that can be used to build the equations operate over scalar/vector terms and constant terms. Whenever any of the operators have requirements related to arguments units, Saiph checks the validity of the operation before performing it, and emits an error if units do not match.
The operators supported by Saiph are

- **Add operator**
  This operator defines the + operator over scalars and vectors as follows:

$$x + y$$

$$\vec{u} + x = (u_1, ..., u_n) + x = (u_1 + x, ..., u_n + x)$$
$$\vec{u} + \vec{v} = (u_1, ..., u_n) + (v_1, ..., v_n) = (u_1 + v_1, ..., u_n + v_n)$$

```
def infix_+(x: scalar-expr, y: scalar-expr) : scalar-expr
def infix_+(u: vector-expr, y: scalar-expr) : vector-expr
def infix_+(u: vector-expr, v: vector-expr) : vector-expr
```

  For the sake of simplicity, for this operator and the ones following, we do not show the version in which the first argument is a scalar and the second argument is a vector, even though it is supported.

- **Subtraction operator**
  This operator defines the − operator over scalars and vectors as follows:

$$x - y$$

$$\vec{u} - x = (u_1, ..., u_n) - x = (u_1 - x, ..., u_n - x)$$
$$\vec{u} - \vec{v} = (u_1, ..., u_n) - (v_1, ..., v_n) = (u_1 - v_1, ..., u_n - v_n)$$

```
def infix_-(x: scalar-expr, y: scalar-expr) : scalar-expr
def infix_-(u: vector-expr, x: scalar-expr) : vector-expr
def infix_-(u: vector-expr, v: vector-expr) : vector-expr
```

- **Division operator**

  This operator defines the / operator over scalars and vectors as follows:

$$x/y$$

$$\vec{u}/x = (u_1, ..., u_n)/x = (u_1/x, ..., u_n/x)$$

$$\vec{u}/\vec{v} = (u_1, ..., u_n)/(v_1, ..., v_n) = (u_1/v_1, ..., u_n/v_n)$$

```
def infix_/(x: scalar-expr, y: scalar-expr) : scalar-expr
def infix_/(u: vector-expr, x: scalar-expr) : vector-expr
def infix_/(u: vector-expr, v: vector-expr) : vector-expr
```

- **Product operator**
  This operator defines the $*$ operator over scalars and vectors. If the two arguments are vectors, it defines the dot product between them.

$$x * y$$

$$\vec{u} * x = (u_1, ..., u_n) * x = (u_1 * x, ..., u_n * x)$$

$$\vec{u} \cdot \vec{v} = (u_1, ..., u_n) \cdot (v_1, ..., v_n) = \sum_{i=1}^{n} u_i * v_i$$

```
def infix_*(x: scalar-expr, y: scalar-expr) : scalar-expr
def infix_*(u: vector-expr, x: scalar-expr) : vector-expr
def infix_*(u: vector-expr, v: vector-expr) : scalar-expr
```

- **Unary minus operator**
  This operator defines an unary - operator over scalars and vectors. It changes the sign of a scalar or, in the case of vectors, changes the sign of all their components.

$$-x$$

$$-\vec{u} = (-u_1, ..., -u_n)$$

```
def infix_-(x: scalar-expr) : scalar-expr
def infix_-(u: vector-expr) : vector-expr
```

- **Component-wise product operator**
  This operator computes the component-wise product of two vectors of the same length. Note that we need this special operator because we have already defined the $*$ operator over two vectors as the dot product.

$$\vec{u} * \vec{v} = (u_1, ..., u_n) * (v_1, ..., v_n) = (u_1 * v_1, ..., u_n * v_n)$$

```
def cwiseprod(u: vector-expr, v: vector-expr) : vector-expr
```

- **Exponential function**
  Given a scalar or a vector, this operator computes the exponential function of it.

$$e^x$$

$$e^{\vec{u}} = e^{(u_1,...,u_n)} = (e^{u_1}, ..., e^{u_n})$$

```
def exp(x: scalar-expr) : scalar-expr
def exp(u: vector-expr) : vector-expr
```

  Restriction: the argument's magnitude must be Unitless.

- **Invert function**
  Given a scalar or a vector, this operator computes the invert function of it.

$$1/x$$

$$1/\vec{u} = 1/(u_1, ..., u_n) = (1/u_1, ..., 1/u_n)$$

```
def invert(x: scalar-expr) : scalar-expr
def invert(u: vector-expr) : vector-expr
```

- **Sum operator**
  This operator computes the sum of all the components of a vector.

$$\sum \vec{u} = \sum (u_1, ..., u_n) = \sum_{i=1}^{n} u_i$$

```
def sum(u: vector-expr) : scalar-expr
```

- **Subscripting operator**
  This operator is used to select an specific component of a vectorial expression.

$$t(i)$$

$$(t1 + t2).apply(i)$$

```
def apply(i: Int) : scalar
```

- **Gradient operator**
  Given a scalar, this operator returns the value of the partial derivatives of this scalar in each direction. Thus, the result type is a vector.

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

```
def grad(f: scalar-expr) : vector-expr
```

- **Divergence operator**
  This function computes the divergence operator of a vector.

$$\nabla \cdot \vec{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$$

```
def div(u: vector-expr) : scalar-expr
```

- **Laplace operator**
  The laplace operator is defined as the sum of the second spatial derivatives of each component of a scalar.

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2}$$

```
def lapla(f: scalar-expr) : scalar-expr
```

- **First time derivative**
  This operator represents a first time derivative of a scalar term.

$$\frac{\partial a}{\partial t}$$

```
def dt(a: Term) : Term
```

- **Second time derivative**
  This operator represents a second time derivative of a scalar term.

$$\frac{\partial^2 a}{\partial t^2}$$

```
def dt2(a: Term) : Term
```

3.3.5 Equations

Defining equations in Saiph involves declaring terms and combine them through operators. An equation is formed by the left-hand side and the right-hand side expressions. Consequently, the units of both sides must match, otherwise, Saiph emits and error. The next code shows how to define an equation:

```
val eq1 = Equation(lhs_expr, rhs_expr)
```

Currently, Saiph is not able to manipulate equations, and thus it can not isolate variables of the problems. For that reason, the user should write the equation in the convenient way. Saiph allows the left-hand side of an equation to be one of the following constructions:

- First time derivative of a scalar term

$$dt(t) = rhs_{expr}$$

- Second time derivative of a scalar term

$$dt2(t) = rhs_{expr}$$

The left-hand sides allowed corresponds to the "natural" lhs_{expr} of partial differential equations. This constraint appear to be a restriction that ends up helping to put users on track.

A complete example can be seen as follows:

```
val mesh = CartesianMesh(XSIZE, YSIZE, ZSIZE)
mesh.discretize(H, H, H)

// Terms and ConstTerms
val ux = Term(Speed)("Velocity_ X", mesh, VX_INIT)

val c = ConstTerm(Speed)("Speed constant", SPEED_COEFF)

// Equation
val convection = Equation(dt2(ux), c*c*lapla(ux))
```

In this example, the one-dimensional wave equation is defined as its lhs_{expr} being a second time derivative of the first component of the velocity vector and its rhs_{expr} defined as the product of the square of a certain constant c by the laplacian of the first component of the velocity.

3.3.6 Boundary conditions

Saiph supports different boundary conditions:

- Periodic conditions to define cyclic volumes.
- Dirichlet conditions to force any variable to have a constant value at the faces of the mesh.
- Neumann conditions to force fluxes of variables (first-spatial derivative) to have constant values at the faces of the mesh.
- Absorbing boundary conditions which define an extra region surrounding the mesh in which variable values are attenuated following a certain smoothing function.

Periodic conditions

Periodic conditions force the cartesian simulation volume to act as a periodic continuous space. If any direction of a mesh is marked as periodic, the values of the back face of

the mesh in that direction correspond to the previous (in a continuous sense) values of the ones at the front face of the mesh in the same direction.

In Saiph, periodic conditions are specified over a certain direction of the mesh. Thus, all the terms of the system are affected by these conditions.

```
def setPeriodic(d: Direction)
```

Where `d` is one of the following values: `DirX`, `DirY` or `DirZ`.

The following code shows how to define a periodic condition:

```
val mesh = CartesianMesh(XSIZE, YSIZE, ZSIZE)
mesh.setPeriodic(DirX)
```

Dirichlet conditions

Dirichlet conditions are used to have fixed values of certain variables, at mesh faces, during the whole execution.

Dirichlet conditions are applied over values of Terms at mesh faces. Currently, the fixed value at the face is not position-time dependent. The following code shows the signature of the method that defines a Dirichlet condition:

```
def setDirichlet(cface: Face)(value: scalar-expr)
def setDirichlet(cface: Face)(value: vectorial-expr)
```

Where `cface` can be `CFaceXMIN`, `CFaceXMAX`, `CFaceYMIN`, `CFaceYMAX`, `CFaceZMIN` or `CFaceZMAX`.

The example below shows how to define a Dirichlet condition:

```
val temp = Term(Temperature)("Temp", mesh, 0 * Kelvins)
temp.setDirichlet(CFaceYMIN)(400 * Kelvins)
```

Neumann conditions

Neumann conditions are used to have fixed values of fluxes of certain variables, at mesh faces, during the whole execution.

Neumann conditions are applied over values of first spatial derivative of Terms at mesh faces. Currently, the fixed value at the face is not position-time dependent. The following code shows the signature of the method that defines a Neumann condition:

```
def setNeumann(cface: Face)(value: scalar-expr)
def setNeumann(cface: Face)(value: vectorial-expr)
```

Where `cface` can be `CFaceXMIN`, `CFaceXMAX`, `CFaceYMIN`, `CFaceYMAX`, `CFaceZMIN` or `CFaceZMAX`.

The example below shows how to define a Neumann condition:

```
val temp = Term(Temperature)("Temp", mesh, 0 * Kelvins)
temp.setNeumann(CFaceYMIN)(400 * Kelvins)
```

Absorbing conditions

Absorbing boundary conditions are typically used in problems that pretend to simulate phenomenas in infinite mediums. Saiph, implements these boundary conditions through the use of sponges, a region that covers the mesh and whose goal is to soft the values of variables reaching the end of the mesh. Doing so, we avoid to mistakenly take into account contributions of variables due to rebounds.

To support sponges, Saiph extends the `Cartesian` constructor specifying the sizes of the sponge:

```
def Cartesian(xs: Float, ys: Float, zs: Float)(bxs: Float, bys: Float,
   bzs: Float) : Cartesian
```

Where `xs`, `ys` and `zs` are the sizes of the user's mesh and `bxs`, `bys` and `bzs` are the respective boundary size of the sponge in each direction.

The following example shows how to use this new constructor.

```
val mesh = Cartesian(12.5, 25.0, 37.5)(2.5, 2.0, 2.5)
mesh.discretize(0.5, 0.5, 0.5)
```

The sponge sizes are specified in continuous space, so, as the main mesh, they have to be discretized. But, once the sponge region is defined within the mesh declaration, the user does not have to take care of anything else related to it. Note that in some constructs, like `Term` construct, the user provides a function that depends on the mesh. This function has to be written thinking on the user's mesh sizes, ignoring the sponges.

3.3.7 Point sources

Some physical problems contains point sources of certain magnitudes. A point source is a single identifiable localised source of any of the magnitudes involved in the problem which has its own contribution to the equations. This component defines and initialize the mesh but it is aimed to be combined with terms through equations constructions. For such purposes, we have the `PointSource` construct that models these conditions that can be time-varying:

```
def PointSource(f: (Float) => Float, m: CartesianMesh, int i, int j, int
    k) : PointSource
```

The first argument `f` is a function that only depends on time. This given function has to return the value of the source for each time step. The second argument is the `CartesianMesh` and the last three arguments are the coordinates of the point where the source is placed within the mesh.

The next example shows a synthetic example that defines a temperature point source situated at the center of the mesh which its value decreases over time.

```
def function(t: Rep[MUnit]) = {
   ((300 * Kelvins) / t)
}
val ps = PointSource(Temperature)(f _, mesh, XSIZE/2, YSIZE/2, ZSIZE/2)
```

Once the point source is defined, we can use it in the equations.

3.3.8 Problem

A `Problem` represents a complete finite-difference formalized problem, aimed to be solved by Saiph.

Defining a problem

The following code shows the constructor of a `Problem`:

```
def Problem(delta_time: Time, num_steps: Float, m: Cartesian)(eqs:
    Equation*) : Problem
```

Where:

- `delta_time` is the temporal interval between two consecutive simulation steps, that is, the time step value.
- `num_steps` is the total number of steps. The simulated time is thus $t_{simulation} = \Delta t * num_{steps}$
- `mesh` is the cartesian mesh.
- `eqs` are the equations of the system which are going to be solved one by one. Saiph allows the user to declare a problem which is modelled by one or more equations. The user should simply enumerate the equations separating them by a comma.

The code below shows an example of how to define a problem called *prob* which is defined by the system of equations *eq1* and *eq2*.

```
val prob = Problem(DELTA_TIME, N_STEPS, mesh)(eq1, eq2)
```

Solving a problem

Once a problem has been defined, it should be solved and results post-processed.

The resolution of a PDE system involves an integrative method. Currently Saiph only supports the explicit Euler method which has been implemented as a part of the low-level C++ library. On the other hand, the post-process defines how the results of the simulation are sampled and selects the output format.

As sampling modes, Saiph offers three different options:

- **Final state of the system** (`SamplingMethod.FinalState`):
  The simulation only takes one sample of the system at the end of the simulation.
- **Snapshoot each x steps** (`SamplingMethod.Periodic`):
  This option takes a sample of the system for each x simulation steps. It allows the user to analyze how the system evolved to its final state and make animations.
- **Flush every frame to the output** (`SamplingMethod.Flush`):
  It takes a sample of the system at every simulation step, equivalent to a `SamplingMethod.Periodic` of one. This mode is the most expensive in terms of space and computing performance but can be suitable for some particular case studies.

As output format, Saiph supports the following ones:

- **VTK images** (`OuputFormat.VTI`), ideal format for visualizing the simulation results with tools like Paraview or Visit.
- **XDMF + Raw binary files** (`OuputFormat.Binary`), suitable format to numerically analyze the data.
- Both VTK and XDMF + Raw (`OutputFormat.all`).

The next code shows how to use the `EulerSolver` specifying post-process options:

```
def EulerSolver(pro: Problem)(on: String, of: OutputFormat, sm:
    SamplingFormat, freq: Int = 1 )
```

Where:

- `pro` is a `Problem`.
- `on` is the ouput name.
- `of` is the ouput format.
- `sm` is the sampling mode.
- `freq` is the frequence of the sampling method, only valid when sm is `SamplingMethod.Periodic`.

The following code illustrates how to solve the problem `prob` generating a VTI file for each simulation step.

```
EulerSolver(prob)("myProblem", OutputFormat.VTI, SampingMethod.Flush)
```

3.4 Saiph's internal features

Internal features are completely transparent to the user, which is only concerned about defining the problem and the governing equations. This illustrates again the potential of Saiph as a productivity boosting tool for solving PDEs systems, hiding the end users from the details of numerical methods and generation of parallel code as much as possible.

3.4.1 Numerical methods

Numerical solution of problems described by partial differential equations need the use of suitable numerical methods. Moreover, numerical evaluation requires the discretization of continuous functions, models, and equations which are time-space dependent. Thus, time-space discretization is the basis of numerical solutions of PDEs. Discretization in Saiph is accomplished by using finite difference methods.

The three continuous spatial dimensions are discretized through the use of Cartesian meshes. Within such meshes, equidistant points are the main entities in which the values of the problem are stored. Each point is identified using the three dimension indexes (i, j, k).

Regarding time, the PDE system is solved using the Euler's method. The temporal evolution of the system is discretized through the use of discrete time steps. At each of those temporal instants, the complete PDE system is solved and results updated.

Finite differences

In order to obtain numerical approximations of PDEs, Saiph uses the finite difference method. This method is a discretization method based on the approximations performed by using Taylor expansions around a point.

The derivative of $\phi(x)$ with respect to x can be defined as

$$\begin{aligned}\frac{\partial \phi}{\partial x}\Big|_{x=i} = \frac{\partial \phi(x_i)}{\partial x} &= \lim_{\delta x \to 0} \frac{\phi(x_i + \delta x) - \phi(x_i)}{\delta x} \\ &= \lim_{\delta x \to 0} \frac{\phi(x_i) - \phi(x_i - \delta x)}{\delta x} \\ &= \lim_{\delta x \to 0} \frac{\phi(x_i + \delta x) - \phi(x_i - \delta x)}{2\delta x}\end{aligned}$$

In all these expressions the approximation converges to the derivative as $\delta x \to 0$, thus they are equivalent in a continuous sense. If δx is small but finite, we can deduce the three different form of approximations of the derivative u_x, forward, backward and central differences:

$$\frac{\partial \phi}{\partial x}\Big|_{x=i}^{forward} \approx \frac{\phi_{i+1} - \phi_i}{\delta x}$$
$$\frac{\partial \phi}{\partial x}\Big|_{x=i}^{backward} \approx \frac{\phi_i - \phi_{i-1}}{\delta x}$$
$$\frac{\partial \phi}{\partial x}\Big|_{x=i}^{central} \approx \frac{\phi_{i+1} - \phi_{i-1}}{2\delta x}$$

Finite difference method is basically the discrete analog of the derivative.

By default, Saiph uses central differences when computing spatial derivatives.

Euler's method

The Euler's method is the most basic explicit method for numerical resolution of a system of partial differential equations with given initial values. It is derived from Taylor expansion and basically approximates the PDE solution through the approximation of the temporal derivative using forward finite differences:
From

$$\frac{\partial \phi}{\partial t}\Big|_{t=n}^{forward} \approx \frac{\phi^{n+1} - \phi^n}{\delta t}$$

the Euler's method is

$$\phi^{n+1} = \phi^n + \delta t \frac{\partial \phi}{\partial t}\Big|_n$$

where $\frac{\partial \phi}{\partial t}$ represents the left-hand side of a partial differential equation expressed as:

$$\frac{\partial \phi}{\partial t} = f(t, \phi)$$

At each time step, $f(t^n, \phi^n)$ is evaluated and added to current results ϕ^n in order to obtain the next temporal solution ϕ^{n+1}:

$$\phi^{n+1} = \phi^n + \delta t f(t^n, \phi^n)$$

This methodology is applied starting from the known initial state $\phi(t_0) = \phi_0$ to a final state determined by a fixed temporal progress.

When having a second order partial differential equation such as

$$\frac{\partial^2 \phi}{\partial t^2} = f(t, \phi)$$

we approximate the solution through the approximation of the second temporal derivative using central finite differences:
From

$$\frac{\partial^2 \phi}{\partial t^2}\Big|_{t=n}^{central} \approx \frac{\phi^{n+1} - 2\phi^n + \phi^{n-1}}{(\delta t)^2}$$

the Euler's method is then

$$\phi^{n+1} = 2\phi^n - \phi^{n-1} + (\delta t)^2 \frac{\partial^2 \phi}{\partial t^2}\Big|_n$$

3.4.2 Domain specific optimizations

High order operator discretization

In some cases, approximating the value of a derivative using only its direct neighboring points does not yield an accurate result. For that purpose, some derivative operators can be discretized using higher-order stencil schemes that use multiple points in each dimension to minimize the numerical error. This is the case for high-order derivative operators as $\frac{\partial^n}{\partial t^n}$ or $\frac{\partial^n}{\partial x^n}$. Consequently, those operators imply a higher computational cost of the whole simulation.

Saiph automatically detects when a particular equation in a system contains higher-order time derivatives and automatically uses more expensive numerical schemes when required.

Convective gradient

For numerical reasons, as we are going to see in section 4.2.1, whenever there is a dot product between a vector and a gradient of some magnitude, the gradient should be a convective gradient. This can be easily done inside the Saiph compiler, using Scala's pattern matching features to identify such case. If that operation takes place, the generated code corresponds to an upwind scheme instead of a regular gradient and dot product chain. Below is the Scala code inside the Saiph compiler that recognizes the pattern and applies the IR node replacement in the application tree:

```
// Dot product of two vectors
def vecDot(x: Exp[Vector], y: Exp[Vector]) = (x, y) match {
  // Replace Grad by StabGrad (upwind gradient) on second argument
  case (x, Def(Grad(u))) => VecDot(x, StabGrad(u, x))
  // Same for the first argument case
  case (Def(Grad(u)), y) => VecDot(StabGrad(u, y), y)
  // Otherwise, just compute dot product of x and y
  case _ => VecDot(x, y)
```

```
}
```

3.4.3 Exploiting parallelism

For solving a PDE, it is necessary to have access only to current time-step values. To apply a spatial discretization at a certain point, the direct neighboring of the point is required. Under this scenario Saiph offers either inter-node and intra-node parallelization.

Inter-node parallelization

Inter-node parallelization is achieved through the Message Passing Interface (MPI) [10]. The mesh is partitioned by the last dimension Z and a similar workload is distributed across the available MPI processes. Each process will solve just its part of the mesh for the whole simulation but has to take care of allocating some extra memory for the boundary values since spatial derivatives need some extra data that may does not belong to the current MPI process.

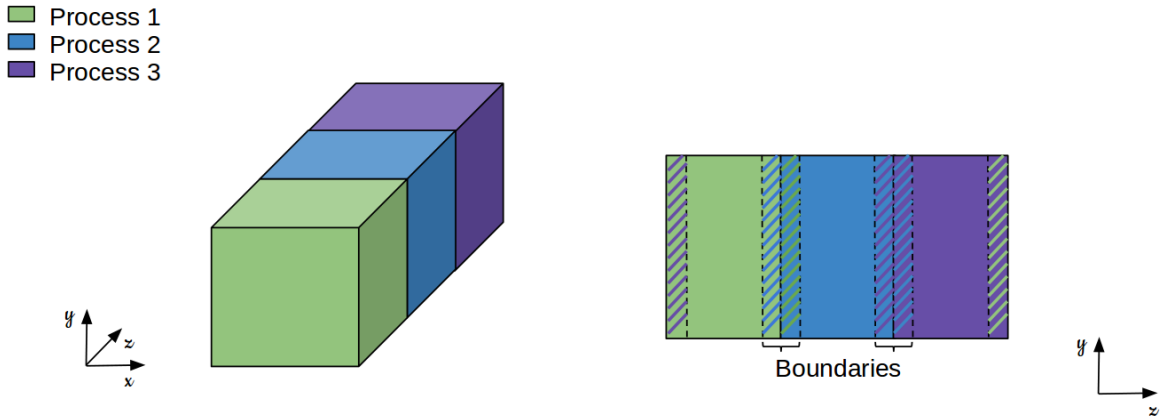

Fig. 3.3: MPI mesh partitioning and shared boundaries

Computations at each time-step are completely parallel and dependence's free but, after each of them, each MPI process has to exchange its boundaries with its neighbors in order to correctly update all the values to be used for the spatial derivatives of future computational steps.

Intra-node parallelization

For the intra-node parallelization, we use OpenMP parallel programming model [11]. At any time, each equation can be integrated in parallel. The current implementation is very simple; by using only one pragma, the three nested loops traversing the mesh are forced to collapse and their iteration spaces are distributed across available OpenMP threads and executed in parallel. There is an implicit barrier at the end of the collapsed loop, so each equation is solved in parallel for all the points of the mesh, one equation after the other.

Inter and intra-node parallelism

Inter and intra-node parallelism are harmoniously combined. For instance, executing a system of equations composed by two equations $eq1$ and $eq2$, with two MPI processes and four OpenMP threads (two threads per MPI process), the execution diagram can be schematized as follows:

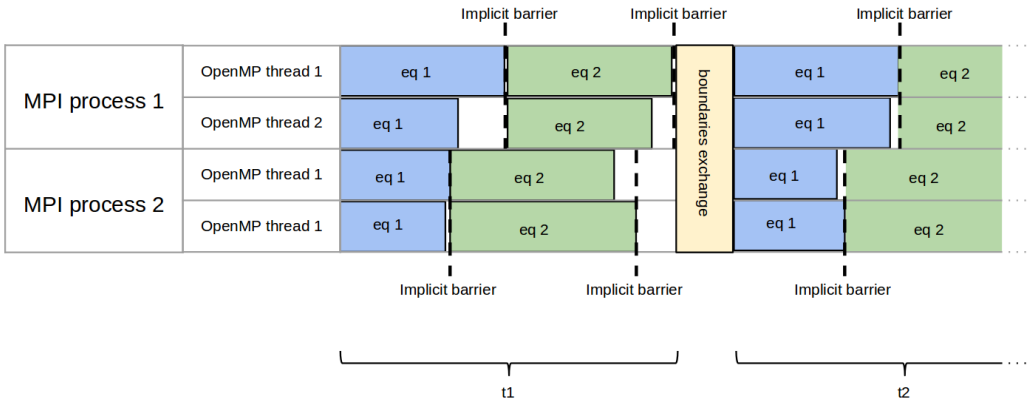

Fig. 3.4: Inter and intra node parallelism

The next figure shows a trace of a time-step of a real execution of a Saiph application modelled by two equations and executed using two MPI processes with two OpenMP threads each. For each MPI process, only the master thread is concerned about the inter-node communications, so *in* the MPI calls. The trace shows how communications are done, taking into account that each process need to exchange boundary values with **two** neighbours[(1)] and of **two** terms; that is 4 message to send and 4 to receive:

[(1)]By default, the left boundary of the first MPI process corresponds to the right boundary of the last MPI process.

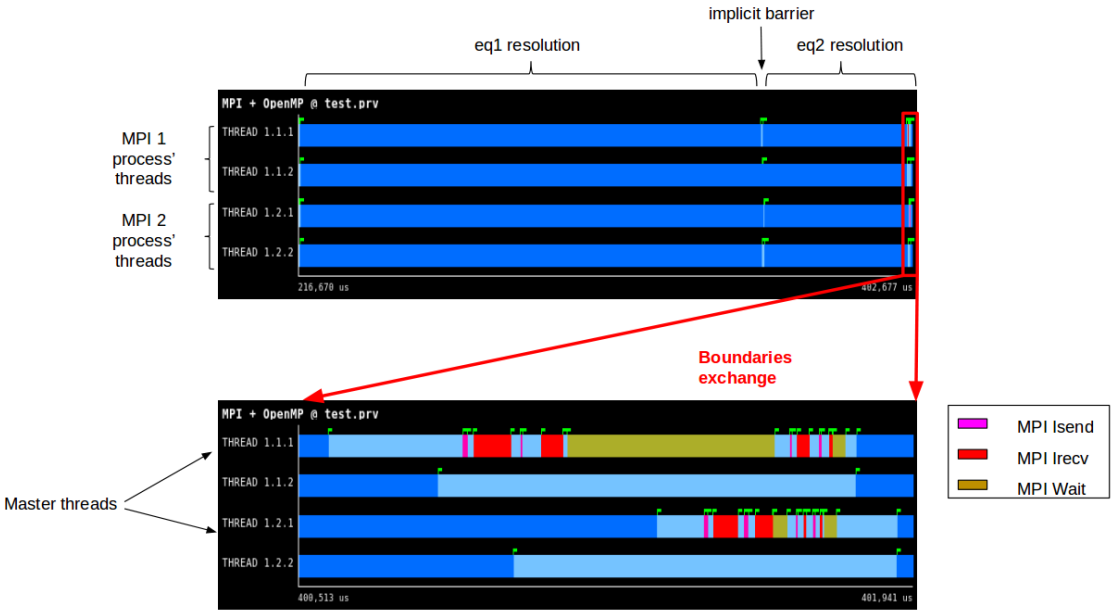

Fig. 3.5: MPI + OpenMP snippet execution trace of a complete time-step

4 | Fluid mechanics and chemistry theory

Fluid mechanics is the study of the effects of forces and energy on liquids and gases, so-called *fluids*. This is a wide domain embracing science, engineering, medicine... with its corresponding large interest within the scientific community. On the other-hand, chemistry, more concretely combustion, produces nowadays around an 85% of the energy we consume, thus, it is also a domain of high interest.

Fluid mechanics correspond to the study of flows. Combustion is basically the study of reacting flows, that is flows in which a chemical process takes place: a reactant (fuel) reacts rapidly with oxygen (oxidizer) and gives off heat. During combustion, new chemical substances (products) are created from the fuel and the oxidizer, when the fuel is hydrogen-carbon-based this products include water and carbon dioxide.

Problems related to those physical domains can be studied through partial differential equation systems (PDEs). This chapter introduces the governing equations of any fluid mechanics and chemistry problem and dissects those equations in order to understand the theoretical meaning of each of its terms and operators.

4.1 Governing equations

The motion of viscous fluid substances is described by the equations of Navier-Stokes. These equations are balance equations coming from applying Newton's second law to fluid motion. Therefore, to take chemistry into account, the equations to be solved correspond to the compressible Navier-Stokes equations for multi-species reactive flows with constant multicomponent mixture properties. The systems are governed by the transport equations of continuity, momentum, species and energy, and they are closed by the equation of state [12]:

- Continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{4.1}$$

- Momentum equation:

$$\frac{\partial(\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot \tau \tag{4.2}$$

- Energy equation:

$$\begin{cases} \frac{\partial(\rho c_p T)}{\partial t} + \nabla \cdot (\rho c_p \mathbf{u} T) = \dot{\omega}_T + \nabla \cdot (\lambda \nabla T) \\ \\ \frac{\partial(\rho E + p)}{\partial t} + \nabla \cdot (\mathbf{u}(\rho E + p)) = \dot{\omega}_T + \nabla \cdot (\lambda \nabla T) \end{cases} \tag{4.3}$$

- Mass equation:
  N species: $k = 1...N$

$$\frac{\partial(\rho Y_k)}{\partial t} + \nabla \cdot (\rho \mathbf{u} Y_k) = \dot{\omega}_k + \nabla \cdot \left(\frac{\lambda}{c_p} \nabla Y_k \right) \tag{4.4}$$

- Equation of state:

$$\begin{cases} p = \rho R^o T \cdot \sum_k^N \frac{Y_k}{W_k} \\ \\ p = (\gamma - 1) \left((\rho E) - \frac{1}{2} (\rho \mathbf{u}) \mathbf{u} \right) \end{cases} \tag{4.5}$$

where ρ is the density of the fluid, u is the fluid velocity, p is the pressure, E is the total energy, T is the temperature, λ is the thermal conductivity, Y_k the mass fraction of each specie k, W_k is the molar mass of each specie k, R^o is the ideal gas constant, c_p is the specific heat capacity at constant pressure and γ is the adiabatic gas index.[(1)]

There are different expression for writing this same system of equations depending on the variables involved. As can be seen, the energy equation 4.3 and the equation of state 4.5 are written above using two different ways. Both equations are going to be used choosing the more convenient one, depending on the context.

This set of equations represents the complete equation system defining a fluid mechanics and chemistry problem. In order to solve a problem of this type, this system should be solved at each spatial point of the domain and for each time-step of the temporal evolution.

4.2 Dissecting the equations

This section correspond to a detailed study and analysis of the PDE system for expressing fluid mechanics and chemistry problems that has been presented in the previous section. The idea is to understand all the terms and operators in a theoretical and also numerical sense.

[(1)] See section 7.1.7 for more details regarding variables, constants and units.

4.2.1 Meaning of terms

Let's take ϕ as an abstract property that represent a scalar property (or a component of a vector property) of the fluid.
Using the mathematical chain rule, this section it is also meant to simplify or simply re-write some of the terms of the equation system.

Variation term

This term indicates the gains or losses of ϕ at each time step t.

$$\frac{\partial(\rho\phi)}{\partial t}$$

This is the term we want to compute at each time step in order to study the temporal evolution of ϕ. Consequently, those variation terms are the terms we want to isolate in the left-hand side of the equation system to know their value through the computation of the time-independent right-hand side.

Convective term

When talking about the convective term we actually refer to the advective term. Convection is actually the phenomena enclosing advection and diffusion.
Advection is a physical process that occurs in a flow of gas or liquid in which some property is transported by the ordered motion of the flow. It can be identified as the divergence of the product of density, velocity and the property being transported.

$$\nabla \cdot (\rho \mathbf{u} \phi)$$

ϕ is being transported by the fluid due to the fluid's bulk motion characterized by the fluid-velocity $\mathbf{u}$.

This term is going to be expressed as

$$\rho \mathbf{u} \cdot \nabla \phi + \phi \nabla \cdot (\rho \mathbf{u})$$

Diffusive term

Diffusion is a physical process that occurs in a flow of gas or liquid in which some property is transported down a concentration gradient of that same property. It can be expressed as the divergence of the product between a diffusion coefficient k and a gradient of the property transported.

$$\nabla \cdot (k \nabla \phi)$$

ϕ is being transported due to the presence of gradients of the property. Diffusion results in transport, without requiring bulk motion.

If k is constant, this term can be written as

$$k\nabla^2\phi$$

Internal source term

In a typical combustion process, the reactants are transformed into products through chemical reactions. For each species k, the internal source term $\dot{\omega}_k$ describes the evolution of species through the chemical reaction. This source terms induces a heat release that is taken into account by the term $\dot{\omega}_T$ in the energy equation that is the internal energy source term.

Consider $k = 1, ..., N$ species reacting through a chemical reaction:

$$\sum_{k=1}^{N} \nu'_k X_k \rightleftharpoons \sum_{k=1}^{N} \nu''_k X_k \tag{4.6}$$

where X_k is a symbol for species k and ν'_k are the molar stoichiometric coefficients of reactants species k and ν''_k are the molar stoichiometric coefficients of product species k. Mass conservation requires:

$$\sum_{k=1}^{N} \nu_k W_k = 0$$

where $\nu_k = \nu''_k - \nu'_k$, and W_k is the molar mass of each specie k. The mass rate $\dot{\omega}_k$ for species k is

$$\dot{\omega}_k = W_k \nu_k Q$$

where Q is the rate of progress of the reaction. This rate Q involves a forward reaction (from left to right in Equation 4.6 and a backward reaction (right to left) and is:

$$Q = K_f \prod_{k=1}^{N} \left(\frac{\rho Y_k}{W_k}\right)^{\nu'_k} - K_b \prod_{k=1}^{N} \left(\frac{\rho Y_k}{W_k}\right)^{\nu''_k}$$

where K_f and K_b are the forward and backward reaction rates constants, given by the Arrhenius expressions:

$$K_f = A_f \exp\left(\frac{-E_a}{R^o T}\right)$$

where A_f is the so-called preexponential factor and E_a is the activation energy.
The heat release term is given by

$$\dot{\omega}_T = -\sum_{k=1}^{N} h_k \dot{\omega}_k$$

where h_k are the specific enthalpies of each specie k.

Viscous term

The viscous term appears in equation 4.2 as the sum of the negative gradient of pressure and the divergence of the viscous tensor τ as

$$-\nabla p + \nabla \cdot \tau$$

where τ is the viscous tensor: $\tau = \mu\left(\left(\nabla \mathbf{u} + \nabla^T \mathbf{u}\right) - \frac{2}{3}\nabla \cdot \mathbf{u}\right)$ and μ is the dynamic viscosity.

In presence of a velocity gradient there is a momentum transfer because of the microscopic motion of the fluid particles. In general, momentum will be transferred from the faster moving layers to the slower moving layers. This net transfer of momentum acts as a friction force in the direction of the gradient and gives rise to the concept of viscosity.

The expression of this term can be simplified depending on the dimensions of the velocity field of the fluid. When having a one-dimensional flow, that is a fluid moving in only one dimension, let say $\mathbf{u} = (u_x, 0, 0)$, the divergence of the viscous tensor can be expressed as:

$$\nabla \cdot \tau = \frac{4}{3}\mu \frac{\partial^2 u_x}{\partial x^2}$$

4.2.2 Operators involved

Besides some concrete operations, the operators involved in the above system of equations are common operators already defined in Saiph or operators than can be emulated through the combination of others. As examples of those operators we have the *product* between scalars and vectors, written as $*$ either on the paper and Saiph code, the *sum operator* written as $\sum$ on the paper and as `sum` in Saiph, the *exponential* written as `exp`, the *laplacian operator* ∇^2 aliased as `lapla` in Saiph, and much others.
Other operators are not implemented in the Saiph version previous to this project; first and second spatial derivatives over specific directions as $\frac{\partial}{\partial x}$, $\frac{\partial^2}{\partial x^2}$ or the divergence of a tensor.

Aside simple operators, some more complicated constructions appear in the system. *Convective gradients*, or *nested derivatives* deserve a more detailed approach.

Convective gradient

As already mentioned the convective term represents the propagation of a certain property in a flow field in the direction of the flow velocity. This is mathematically translated as the dot product of the fluid velocity $\mathbf{u}$ and the gradient of the scalar property being transported:

$$\mathbf{u} \cdot \nabla \phi$$

In order to simulate this propagation of information in the direction determined by the velocity vector it is convenient to use an adaptive finite difference stencil called *upwind scheme* or convective stabilized gradient. This scheme is used for the computation of the gradient of the property being transported and it is based in a differentiation biased in the direction determined by the sign of the characteristic velocity of the flow, thus $\mathbf{u}$. If a central scheme is used when computing this term, numerical errors arise. Those errors are induced by the fact that finite differences schemes discretize the derivative operators in a non-continuous space, thus loosing precision. Therefore, upwind schemes minimize the discretization error by biasing the derivative computation in the direction the data is "coming from", therefore using points in the direction of advection determined by the sign of the vector velocity components.

In order to illustrate this propagation of information, let's imagine a fluid with a velocity field defined as $\mathbf{u} = (u_x, 0, 0)$, where $u_x > 0$. Then, the spatial derivative of a property ϕ in the X direction should be computed as

$$\frac{\partial \phi}{\partial x}\Big|_{x=i}^{backward} = \frac{(\phi_i - \phi_{i-1})}{\delta x}$$

with the sub-indexs of ϕ indicating the spatial position in the X axis. This correspond to the first order accuracy backward finite difference scheme. For a negative velocity, the forward scheme should be used.

The problem arises when the velocity vector has more than one non-zero component. Advection in two or three dimensions can not be modelled through a simple finite differences upwind scheme. If so, the information is not correctly propagated as it can be seen in the following example.

Let's consider a two-dimensional advection of a certain scalar magnitude ϕ being propagated through advection modelled as $\frac{\partial \phi}{\partial t} = -\mathbf{u} \cdot \nabla \phi$ with $\mathbf{u}$ the constant velocity vector. Figure 4.1 illustrates two firsts temporal configurations. To evolves from $t1$ to $t2$, we apply Euler's method at each point.

Taking the (i, j) point as example:

$$\phi_{(i,j)}^{t2} = \phi_{(i,j)}^{t1} + \delta t \frac{\partial \phi}{\partial t}\Big|_{t1,(i,j)}$$

with

$$\frac{\partial \phi}{\partial t}\Big|_{t1,(i,j)} = (-\mathbf{u} \cdot \nabla \phi)\Big|_{t1,(i,j)}$$

In two dimensions, $\nabla \phi = \left(\frac{\partial \phi}{\partial x}, \frac{\partial \phi}{\partial y}\right)$

Computing the gradient using the upwind scheme in finite differences with $u_x > 0$ and

$u_y > 0$, we have:

$$\left.\begin{array}{l} \left.\frac{\partial\phi}{\partial x}\right|^{backward}_{t1,(i,j)} = \frac{\phi^{t1}_{(i,j)} - \phi^{t1}_{(i-1,j)}}{\delta x} \\ \left.\frac{\partial\phi}{\partial y}\right|^{backward}_{t1,(i,j)} = \frac{\phi^{t1}_{(i,j)} - \phi^{t1}_{(i,j-1)}}{\delta y} \end{array}\right\} (\nabla\phi)\Big|_{t1,(i,j)} = (0,0)$$

Thus,

$$\left.\frac{\partial\phi}{\partial t}\right|_{t1,(i,j)} = 0 \quad \Rightarrow \quad \phi^{t2}_{(i,j)} = \phi^{t1}_{(i,j)}$$

which is not the expected result.

The same procedure can be applied for all the points of the two-dimensional mesh, resulting in the wrongly computed configuration (a) at $t2$ of the figure.

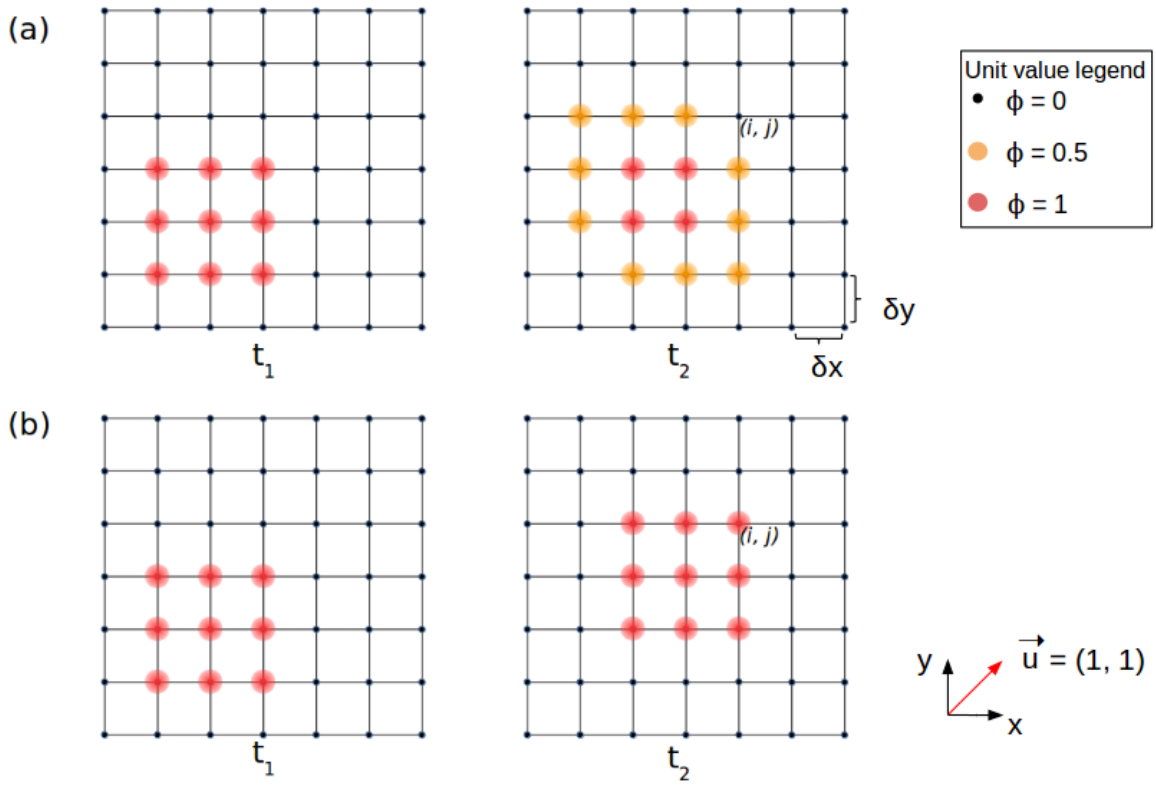

Fig. 4.1: Propagation of information by advection
(a) Using an upwind scheme based on finite differences
(b) Ideal propagation

Therefore, when encountering a convective gradient in which the direction of propagation has more than one component, the finite differences upwind scheme can not be used.

Nested derivatives

As explained above, the diffusive term represents the transport of a certain property in a flow field due to the inhomogeneous repartition of this property in the physical domain. This is mathematically translated as the divergence of the gradient of the scalar property being transported:

$$\nabla \cdot (k \nabla \phi)$$

The diffusion term has two nested spatial derivatives. When the diffusion coefficient k is constant over the space, the diffusion term can be written as $k\nabla^2\phi$. Numerically, ∇^2 and $\nabla \cdot \nabla$ must be equivalent but when nesting two derivative operators using the second-order accuracy central scheme we obtain the following result:

$$\frac{\partial^2 \phi}{\partial x^2}\Big|_{x=i} = \frac{\frac{\partial \phi}{\partial x}\Big|_{x=i+1} - \frac{\partial \phi}{\partial x}\Big|_{x=i-1}}{2\delta x}$$

$$\left.\begin{array}{l} \frac{\partial \phi}{\partial x}\Big|_{x=i+1} = \frac{(\phi_{i+2}-\phi_i)}{2\delta x} \\ \frac{\partial \phi}{\partial x}\Big|_{x=i-1} = \frac{(\phi_i-\phi_{i-2})}{2\delta x} \end{array}\right\} \frac{\partial^2 \phi}{\partial x^2}\Big|_{x=i} = \frac{\phi_{i-2} - 2\phi_i + \phi_{i+2}}{4(\delta x)^2}$$

which is different than applying the ∇^2 operator directly by using the coefficients of the central differences for a two order derivation, defined by the Taylor series as:

$$\frac{\partial^2 \phi}{\partial x^2}\Big|_{x=i} = \frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{(\delta x)^2}$$

Thus, applying central difference twice does not result in the appropriate result. Instead, numerical correctness can be ensured by using a combined scheme of forward and backward differentiation as follows:

$$\frac{\partial^2 \phi}{\partial x^2}\Big|_{x=i}^{forward} = \frac{\frac{\partial \phi}{\partial x}\Big|_{x=i+1}^{backward} - \frac{\partial \phi}{\partial x}\Big|_{x=i}^{backward}}{\delta x}$$

$$\left.\begin{array}{l} \frac{\partial \phi}{\partial x}\Big|_{x=i+1}^{backward} = \frac{(\phi_{i+1}-\phi_i)}{\delta x} \\ \frac{\partial \phi}{\partial x}\Big|_{x=i}^{backward} = \frac{(\phi_i-\phi_{i-1})}{\delta x} \end{array}\right\} \frac{\partial^2 \phi}{\partial x^2}\Big|_{x=i} = \frac{\phi_{i-1} - 2\phi_i + \phi_{i+1}}{\delta x^2}$$

which lead to the correct differenciation.

4.2.3 Vector equations

There are two vector equations in the system, the momentum equation 4.2 and the mass conservation for each specie 4.4. Those equations seem to be written down as single equations with single unknowns $\mathbf{u}$ and Y_k respectively. The velocity vector $\mathbf{u}$ represents actually three unknowns which are the three direction components u_x, u_y and u_z of the vector velocity. Regarding the mass fraction vector, each component represents the mass fraction of an specific chemical specie present in the fluid. Each of those components is an unknown of the system and has its own equation as we can see as follows [(2)]:

$$\frac{\partial(\rho\mathbf{u})}{\partial t} = -[\rho\mathbf{u}\cdot\nabla\mathbf{u} + \mathbf{u}\nabla\cdot(\rho\mathbf{u}) + \nabla p] \Rightarrow \begin{cases} \frac{\partial(\rho u_x)}{\partial t} = -[\rho\mathbf{u}\cdot\nabla u_x + u_x\nabla\cdot(\rho\mathbf{u}) + \frac{\partial p}{\partial x}] \\ \frac{\partial(\rho u_y)}{\partial t} = -[\rho\mathbf{u}\cdot\nabla u_y + u_y\nabla\cdot(\rho\mathbf{u}) + \frac{\partial p}{\partial y}] \\ \frac{\partial(\rho u_z)}{\partial t} = -[\rho\mathbf{u}\cdot\nabla u_z + u_z\nabla\cdot(\rho\mathbf{u}) + \frac{\partial p}{\partial z}] \end{cases}$$

The vector mass equation contains as many equations as specie are in the fluid under study. For a typical combustion process we have the following set of mass fraction equations.

$$\frac{\partial(\rho Y_k)}{\partial t} = -[(\rho\mathbf{u})\cdot\nabla Y_k + Y_k\nabla\cdot(\rho\mathbf{u})] + \nabla\cdot\left(\frac{\lambda}{c_p}\nabla Y_k\right)$$

$$\Rightarrow \begin{cases} \frac{\partial(\rho Y_{CH_4})}{\partial t} = -[(\rho\mathbf{u})\cdot\nabla Y_{CH_4} + Y_{CH_4}\nabla\cdot(\rho\mathbf{u})] + \nabla\cdot\left(\frac{\lambda}{c_p}\nabla Y_{CH_4}\right) \\ \frac{\partial(\rho Y_{O_2})}{\partial t} = -[(\rho\mathbf{u})\cdot\nabla Y_{O_2} + Y_{O_2}\nabla\cdot(\rho\mathbf{u})] + \nabla\cdot\left(\frac{\lambda}{c_p}\nabla Y_{O_2}\right) \\ \frac{\partial(\rho Y_{N_2})}{\partial t} = -[(\rho\mathbf{u})\cdot\nabla Y_{N_2} + Y_{N_2}\nabla\cdot(\rho\mathbf{u})] + \nabla\cdot\left(\frac{\lambda}{c_p}\nabla Y_{N_2}\right) \\ \frac{\partial(\rho Y_{CO_2})}{\partial t} = -[(\rho\mathbf{u})\cdot\nabla Y_{CO_2} + Y_{CO_2}\nabla\cdot(\rho\mathbf{u})] + \nabla\cdot\left(\frac{\lambda}{c_p}\nabla Y_{CO_2}\right) \\ \frac{\partial(\rho Y_{H_2O})}{\partial t} = -[(\rho\mathbf{u})\cdot\nabla Y_{H_2O} + Y_{H_2O}\nabla\cdot(\rho\mathbf{u})] + \nabla\cdot\left(\frac{\lambda}{c_p}\nabla Y_{H_2O}\right) \end{cases}$$

[(2)]For the sake of simplicity we neglect the viscous term of the momentum equation

As we can see, there are terms on the right-hand side of the equations that depend on the component of the vector unknown at the right-hand side.

Vector equations allow to define several equations with a common pattern but that are different in essence.

4.2.4 Non-derivative equation

The equation of state 4.5 is a non-derivative equation. This equation gives us the relation between state variables. This is the thermodynamic equation that describes the state of matter under a given set of physical conditions. As a time-independent equation, the left and right-hand sides must match at any time. Because of the nature of this kind of equations, the values of the variables involved should be from the same time-step.

The a-temporal relation of the equation of state can be seen as follows:

$$p^t = f((\rho T)^t, Y_k^t)$$

being t any moment in time and f a function relating both sides of the equation.
Those non-derivative equations should obviously be also verified at each space coordinate.

4.2.5 Coupled system

As we can see, the Navier-Stokes equations for multi-species reactive flows consist of a time-dependent continuity equation 4.1 for conservation of mass, three time-dependent conservation of momentum equations 4.2, a time-dependent conservation of energy equation 4.3 and N time-dependent conservation of mass (for each specie k of the system) equations 4.4. That is $(5 + N)$ partial differential equations.
In this system of partial differential equations, there are $(6+N)$ unknown variables: density ρ, pressure p, temperature T (or total energy E), three components of the velocity vector $\mathbf{u}$ and N mass fractions of each specie involved in the reaction.

To solve a flow problem, it is necessary to solve all $(5 + N)$ partial differential equations simultaneously using values from previous time-steps; that is why we call this a coupled system of equations. The equation of state 4.5 is actually the closing necessary equation required to solve the system. We have $(5 + N)$ partial differential equations for $(6 + N)$ unknowns. The equation of state relates the pressure, temperature, density and mass fraction of the gas at each time step and thus close the solvable system.

Integration scheme

At each time step, the unknown variables from the partial differential equations are updated using variables values from previous time-steps. Following the Euler's method, the

variation terms are computed and added to the values of previous time-steps applying the corresponding factors.

The integration scheme of the full system can be expressed as follows:

$$\rho^{t+1} = \rho^t + \delta t \cdot f((\rho u)^t)$$

$$(\rho u)^{t+1} = (\rho u)^t + \delta t \cdot f((\rho u)^t, u^t, p^t)$$

$$(\rho T)^{t+1} = (\rho T)^t + \delta t \cdot f((\rho u)^t, (\rho Y_k)^t, T^t)$$

$$(\rho Y_k)^{t+1} = (\rho Y_k)^t + \delta t \cdot f((\rho u)^t, (\rho Y_k)^t, T^t)$$

$$p^{t+1} = f((\rho T)^{t+1}, Y_k^{t+1})$$

Of course, variables should be updated for each time step before being accessed. As we can appreciate, variables in the right-hand side of the equation of state should be updated for the same time-step at which the left-hand side variable is being computed. This fact forces to define an order of equation resolution. Partial differential equations can be solved simultaneously (no order restriction) on the other hand, time-independent equations (equation of state) should be the last equations to be solved at each time step. At a same time step, values updated through the resolution of partial differential equations are used for the resolution of non-derivative equations.

Initial state

Within fluid mechanics and chemistry theory, problems are commonly initialized defining different thermodynamic regions in the spatial domain. This fact induces the existence of interfaces between regions, where the variables should have a defined behaviour. Usually, values traverse those interfaces following a linear function. The following graph illustrates the initial profiles of the temperature in a two-regions domain.

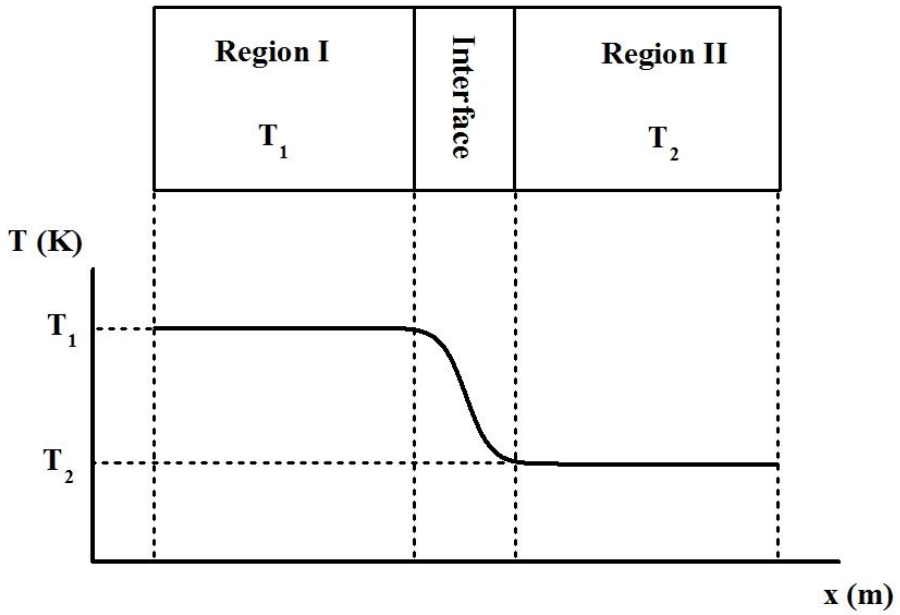

Fig. 4.2: Initial temperature profile within a domain of two regions and the correspondent interface, at t=0

As we have already said, the equation of state should be verified at each moment in time and this includes the $t = 0$ time step, that is the initial state. Of course, it should also be verified at any space coordinate since it precisely relates the state variables at each localization of the domain. Consequently, the state variables at the interfaces must follow exactly the same function in order to fulfill the equation of state at each point in the domain at $t = 0$.

5 | Tools and methodology

This chapter lists the tools used for developing and extending Saiph and reports the development methodology of the project. Bearing in mind that this is a project based on the extension of an already existing tool, Saiph, the tools used for its extension are predefined by its nature; a DSL embedded in Scala with LMS and Scala-virtualized compiler. On the other hand, the concrete methodology of this project has been applied for the first time resulting in a very appropriate and progress drive way of development.

5.1 Tools

As already mentioned, the principal tool used has been Saiph as it was before this project presented in chapter 3. Consequently, the main utensils have been all those tools involved in the creation of the DSL. That is, the Scala language, displayed in 3.2.1 intended for being the final basis language for the end-users' applications; the lightweight modular staging (LMS) shown in 3.2.2 and the scala-virtualized compiler explained in 3.2.3 used for the DSL design as the interfaces between the high level Saiph code and the specific parallel C++ generated code. Because of the design of the DSL, an important part of the development has also been performed at the C++ level, involving all the resources provided by this object-oriented language [13].
Visualization and graphing tools as Paraview [14] and Gnuplot [15] have been used in order to illustrate and validate simulations results and, finally, Extrae [16] and Paraver [17] for the validation of the parallel executions.

5.2 Methodology

The research design has been stated as application driven design. The Saiph's extensions constituting this project have been determined by the requirements encountered on the selected use-cases. Those use-cases have been picked following a certain scientific method design explained below. Once the necessary extensions have been theorized, they have been fulfilled and carried out following a development strategy, also explained as follows.

5.2.1 Scientific method design

As stated, the objective of the project is to extend Saiph in order to model and solve fluid mechanics and chemistry problems. Thus, the goal is to correctly simulate the use-case enclosing those two scientific domains which is the aforementioned *premixed laminar flame*. Before dealing with such a complex use-case, simpler ones have been gradually faced. Thus, a gradually development progress has been followed. Those simpler use-cases basically are divided in two main categories, the ones related to fluid mechanics theory and those related to chemistry, more precisely combustion. Within those two categories, the same progressive method has been followed; each use-case contains more equation's terms than its predecessor. Hence, all the terms explained in section 4.2.1 have been gradually implemented, tested and validated through the adequate choose of the use-cases. Finally, the composition of all those validated terms from the two different scientific domains (fluid mechanics and chemistry) has allowed the correct resolution of the final use-case.

In order to be able to progress from the simplest use-case to the more complex one, a validation process is a must. In that sense, use-cases have been selected among others depending on their popularity within the scientific community and the possibility to solve them either analytically or/and empirically. Thus, all the selected use-cases have been validated against results reported by scientific researches, cited in the current document.

5.2.2 Development strategy

In a first instance, each necessary extension has been faced at the C++ level, fattening up the C++ library with all those constructions required for the selected use-cases. Then, applications have been programmed in C++, designed to be compiled and correctly executed at this *low* level of abstraction. After validating simulation results, our embedded compiler based on LMS has been extended in order to support these new extensions. At that point, the optimizations related to the generation of specific IR nodes have been performed. Finally, each use-case has been re-wrote in Saiph language, compiled, executed and validated again. This development method process has been repeated for each of the faced use-cases.

6 | Extending Saiph

The version of Saiph presented in this project has been implemented in order to fulfill the requirements from fluid mechanics and chemistry problems, presented in chapter 4. In the current chapter, the most important extensions are going to be described. Those extensions can be divided in two main categories; First, the new general functionalities which are going to be handled by users at the high-level and second, domain specific optimizations, transparent to the users and very specific for fluids mechanics and chemistry problems.

6.1 New functionalities

After analysing all the details necessaries to perform the desired simulations, we present the implementation of the required functionalities. The extensions presented in this section are general Saiph extensions useful to the resolution of any problem expressed in PDEs.

6.1.1 Vector equations

In order to allow vector equations, we firstly extend the list-case that the left-hand side expression of an equation can match. That is to allow the lhs_{expr} to be either a vector term [(1)] and a time derivative of a vector term, which is also a vector as shown in the example below.

[(1)] lhs_{expr} being a non-derivative terms is also an extension of this project, see 6.1.2

$$\frac{\partial \mathbf{u}}{\partial t} = \left(\frac{\partial u_x}{\partial t}, \frac{\partial u_y}{\partial t}, \frac{\partial u_z}{\partial t} \right)$$

$$\frac{\partial^2 \mathbf{u}}{\partial t^2} = \left(\frac{\partial^2 u_x}{\partial t^2}, \frac{\partial^2 u_y}{\partial t^2}, \frac{\partial^2 u_z}{\partial t^2} \right)$$

$$\text{with } \mathbf{u} = (u_x, u_y, u_z)$$

Allowing such left-hand sides, the dimensionality of this part of the equation cannot be assumed any more. Thus, when solving any equation, checking the dimensionality of the lhs_{expr} has became a must. Whenever this dimensionality is greater than one, the evaluation of the right-hand side is performed for each component of the left-hand side. This process is easily implemented by forcing an iteration over the components of each lhh_{expr} at each time step and mesh location, during the evaluation an update process. In essence, a vector equation internally represents a set of scalar equations. This set contains as many scalar equations as dimensions of the left-hand side vector.

Those changes have been implemented in the C++ library, modifying the code that traverses the mesh to apply Euler's method at each location for each time step and equation. Before the application of the Euler's method, the query related to the dimensionality of lhs_{expr} and the loop iterating on it, have been added in order to force all the scalar equations of the set to be integrated by the Euler's method.

The left-hand side of the equation can be written as a vector, but the right-hand side should remain being evaluated as a scalar. This is achieved through the *component* operator, a new operator that can be used to select the component of a vector expression with the same index than the component being evaluated at the left-hand side of a vector equation.

$$t.comp$$

$$(t1 + t2).comp$$

```
def comp() : scalar
```

In the following Saiph code the use of this operator is illustrated.

```
// Velocity; vector term
val u = Term(Speed)("Velocity", mesh, V_INIT, List[Rep[String]]("X", "Y"
    , "Z"))
// Pressure; scalar term
```

```
val p = Term(Pressure)("Pressure", mesh, INIT_P)
// Density; scalar term
val rho = Term(Kg_m3)("Density", mesh, INIT_RHO)

// Vector equation
val eq1 = Equation(dt(u), -(rho / (grad(p)).comp))
```

Having a vector lhs_{expr}, through the use of the *component* operator applied to the vector terms of the rhs_{expr}, it can be seen that by writing down a single equation `eq1`, we are actually defining a set of scalar equations:

$$\begin{cases} \frac{\partial u_x}{\partial t} = -\frac{1}{\rho}\frac{\partial p}{\partial x} \\ \frac{\partial u_y}{\partial t} = -\frac{1}{\rho}\frac{\partial p}{\partial y} \\ \frac{\partial u_z}{\partial t} = -\frac{1}{\rho}\frac{\partial p}{\partial z} \end{cases}$$

6.1.2 Non-derivative equations

In order to allow non-derivative equations, we also extend the list-case that the left-hand side expression of an equation can match. That is to allow the lhs_{expr} to be a term. At that point, the left-hand side expression of an equation has to be defined matching one of the following syntax:

- lhs_{expr} is a scalar or vector term:

$$t = rhs_{expr}$$

- lhs_{expr} is a first time derivative of a scalar or vector term:

$$dt(t) = rhs_{expr}$$

- lhs_{expr} is a second time derivative of a scalar or vector term:

$$dt2(t) = rhs_{expr}$$

A non-derivative equation does not have to be integrated. Thus, to solve this kind of equations, we should not go through the Euler's method. Instead, at each point of the mesh, the right-hand side is simply evaluated and assigned to the term at the left-hand side. This is done within the loop traversing the mesh at the update process for each time step. Depending on the equation type (which must be checked for each equation), the Euler's method is applied or the simple assignation is performed.

6.1.3 Coupled scheme

At each time step, the unknown variables of the problem are updated through the correct evaluation and manipulation of its associated right-hand side expression. The three equation types allowed and their requirements to be updated are:

- PDEs
  At each time step, the Euler's method should be applied.
  - First order PDEs need to access values from the previous time-step in order to update the unknown variable.

$$\phi^{n+1} = \phi^n + \delta t \frac{\partial \phi}{\partial t}\Big|_n$$

  - Second order PDEs need to access values from the two previous time-step in order to update the unknown variable.

$$\phi^{n+1} = 2\phi^n - \phi^{n-1} + (\delta t)^2 \frac{\partial^2 \phi}{\partial t^2}\Big|_n$$

  In any case, the evaluation of the rhs_{expr} is used to substitute $\frac{\partial \phi}{\partial t}$ and $\frac{\partial^2 \phi}{\partial t^2}$ respectively and thus, should be evaluated at time step n. That means that the previous time-step values must be accessible when solving PDEs.

- Non-derivative equations
  Those kind of equations need to access values from the current time-step.

$$\varphi^{n+1} = f(\phi^{n+1})$$

  The evaluation of the already updated rhs_{expr} is directly used to be assigned to the unknown variable at each time step. That means that the current time-step values must exist and be accessible when solving non-derivative equations.

Because Saiph is a domain specific language for solving PDE systems, variables at the right-hand side of an expression refer to values from the previous time-step by default. Hence, in order to access updated values, those values should exist and the specific access should be specified.

The existence of the values must be ensured by the users; As explained, the equations are solved in parallel for all the points of the mesh, one after the other. Thus, the order in which the user lists the equations of the problem to be solved, is crucial when facing coupled system problems. Non-derivative equation should be the last equations to be solved at each time-step in order to allow them to access to the most recent updated values.

On the other hand, accessing to an updated value within the same time-step need to be specified. This is achieved through the *time offset* operator, a new operator that can be used to select a value of any variable at the time-step desired.

$$t.toff(i)$$

```
def toff(i: Int) : Term
```

where i represents the reference of the desired time-step:

- $i = 0$ refers to the access at the current time step.
- $i = 1$ refers to the access at the previous time step (this is the default case when the *time offset* operator is not used).
- $i = 2$ refers to the access at the previous of the previous time step.

Of course, this operator can only be applied to `Terms` since `ConstTerms` have the same value at any time-step.

The following Saiph code illustrates how to handle coupled systems.

```
// Velocity; vector term
val u = ConstTerm(Speed)("Velocity", V_INIT, List[Rep[String]]("X", "Y",
    "Z"))
// Pressure; scalar term
val p = Term(Pressure)("Pressure", mesh, INIT_P)
// Density; scalar term
val rho = Term(Kg_m3)("Density", mesh, INIT_RHO)

// Vector equation
val eq1 = Equation(dt(rho), -(div(rho*u)))

// Non-derivative equation
val eq2 = Equation(p, rho.toff(0) * u(0))

// Problem with ordered equation list
val problem1 = Problem(DELTA_TIME, N_STEPS, mesh)(eq1, eq2)
```

It is also possible to require the access to an updated value within a vector equation. The *component time offset* operator has been added for such cases. It is a mix of the 'comp' and the 'toff' operators. It can only be applied to vector terms.

$$t.compWithOffset(i)$$

```
def compWithOffset(i: Int) : scalar
```

Note that allowing non-derivative equations can cause problems from the parallelization point of view; If a non-derivative equations contains spatial derivatives, a boundary exchange between MPI processes must precede the evaluation of the equation. Since such an equation is not present in any of the selected fluid mechanics and chemistry use-cases, this issue is going to be taken into account as future work.

6.1.4 Operations over vector of Units

When a user defines a vector term, it may be useful to support also operations over vector of Units. This extension eases the task of defining the complete initial state of a coupled problem containing non-derivative equations with related vector terms. Hence, a term could be initialized through the adequate combination of other initialized terms in order to verify the relation stated by the non-derivative equation.

The Unit's operators added are in fact the same as the ones for scalar Units listed in section 3.3.1 but extended to allow vector Units to be taken as arguments. Those vector operators are all defined to perform component-wise operations.

Add operators

```
def infix_+(l: Vector[Unit], r: Vector[Unit]) : Vector[Unit]
def infix_+(l: Vector[Unit], r: Unit) : Vector[Unit]
def infix_+(l: Unit, r: Vector[Unit]) : Vector[Unit]
```

Minus operators

```
def infix_-(l: Vector[Unit], r: Vector[Unit]) : Vector[Unit]
def infix_-(l: Vector[Unit], r: Unit) : Vector[Unit]
def infix_-(l: Unit, r: Vector[Unit]) : Vector[Unit]
```

Product operators

```
def infix_*(l: Vector[Unit], r: Vector[Unit]) : Vector[Unit]
def infix_*(l: Vector[Unit], r: Unit) : Vector[Unit]
def infix_*(l: Unit, r: Vector[Unit]) : Vector[Unit]
def infix_*(l: Scalar, r: Vector[Unit]) : Vector[Unit]
def infix_*(l: Vector[Unit], r: Scalar) : Vector[Unit]
```

Division operators

```
def infix_/(l: Vector[Unit], r: Vector[Unit]) : Vector[Unit]
def infix_/(l: Vector[Unit], r: Unit) : Vector[Unit]
def infix_/(l: Unit, r: Vector[Unit]) : Vector[Unit]
def infix_/(l: Scalar, r: Vector[Unit]) : Vector[Unit]
def infix_/(l: Vector[Unit], r: Scalar) : Vector[Unit]
```

The following code shows an example using these vector operations:

```
// Defining a vector of velocities
def velocities = Vector(2 * MetersPerSecond, 1 * MetersPerSecond, -1 *
    MetersPerSecond)

// Scaling velocities. The result is also a vector of velocities
def scaled_velocities = 0.5 * velocities

// Meters walk in 10 seconds. The result is a vector of lengths
def time = 10 * Seconds
def meters_walk = time * velocities
```

6.1.5 Other operators

First spatial derivative

This operator computes the first spatial derivative of a scalar expression in a certain direction using central differences.

$$\frac{\partial f}{\partial d}$$

```
def der(d: Direcion, f: scalar-expr) : scalar-expr
```

The d argument must be one of the following keywords: `DirX`, `DirY` and `DirZ`.

Second spatial derivative

This operator computes the second spatial derivative of a scalar expression in a certain direction using central differences.

$$\frac{\partial^2 f}{\partial d^2}$$

```
def der2(d: Direcion, f: scalar-expr) : scalar-expr
```

The d argument must be one of the following keywords: DirX, DirY and DirZ.

6.2 Optimizations

This section presents the specific domain optimizations internally added to Saiph, transparent to the users. They are new Saiph features or components which does not change the user external interface while changing the Saiph internal behaviour. Those optimizations are specific for the resolution of PDEs systems and even more specific for the resolution of fluids and chemistry problems expressed in PDEs. Thus, specific numerical methods, terms identification and operators utilization are internally implemented and applied to correctly model proper fluids and chemistry phenomenas. Therefore, the optimizations presented below are not general but have been implemented in order to show its potential use and, as a first attempt to face numerical problems arising from real use-cases.

6.2.1 Stabilized gradient

As we have seen in 4.2.2, two or three dimensional velocity fields induce problems for the correct modeling of advection phenomena. The upwind scheme is no longer correct for the two-dimensional phenomena. Nevertheless, the possibility of advecting at different speeds in different dimensions should be contemplated.

Because the equation type (first order partial differential equations) and because the discretization factor δt is constant for the whole execution, we can face the two or three dimensional advection through the Corner Transport Upstream (CTU) method [18] [19].

This method consists of breaking a finite-difference formula into a series of steps. In the 2D problem that we have been using, it corresponds to take the x integration step first, followed by a y integration step.

From the advection equation $\frac{\partial \phi}{\partial t} = -\mathbf{u} \cdot \nabla \phi$ we have:

$$\phi^*_{i,j} = \left(1 - \frac{u_x \delta t}{\delta x}\right) \phi^t_{i,j} + \frac{u_x \delta t}{\delta x} \phi^t_{i-1,j}$$

$$\phi^{t+1}_{i,j} = \left(1 - \frac{u_y \delta t}{\delta y}\right) \phi^*_{i,j} + \frac{u_y \delta t}{\delta y} \phi^*_{i,j-1}$$

where $u_x > 0$ and $u_y > 0$

This last form can be interpreted as applying the upwind scheme successively in each direction, using the latest values at each stage. The following figure illustrates this method.

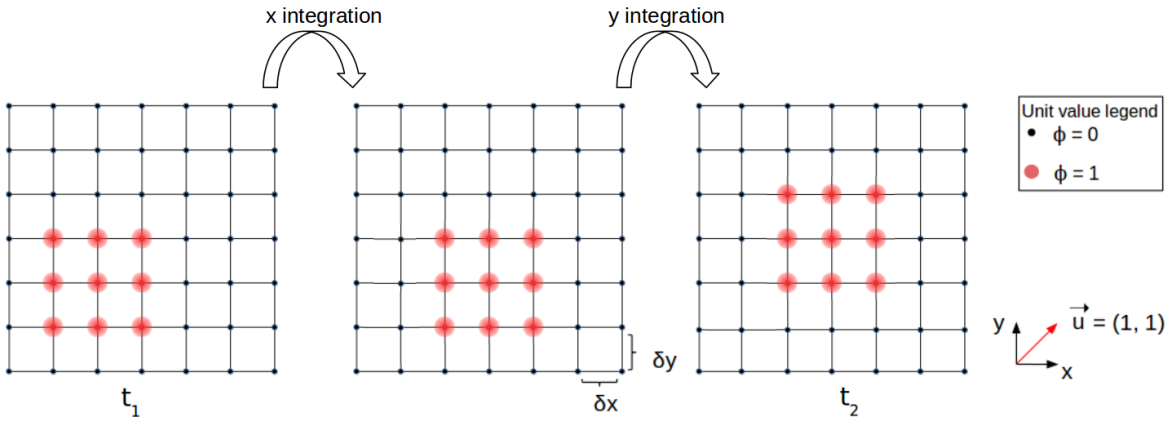

Fig. 6.1: Propagation of information by 2D-advection using a Corner Transport Upstream method

Since a convective gradient can be a term of a right hand-side equation among others, we should ensure that the whole equation is correctly integrated. To ensure that, we need to be able to isolate the contribution from this convective term in order to design and code the most suitable stabilized gradient operator whose result is going to be multiplied by the velocity vector (as dot product) added to the rest of the equations' contributions and finally integrated through Euler's method.

From the CTU expressions, we re-write $\phi^{t+1}_{i,j}$ including the $\phi^*_{i,j}$ definition. Arranging such expression we obtain:

$$\phi_{i,j}^{t+1} = \phi_{i,j}^{t} - u_x \delta t \left(\frac{\phi_{i,j}^{t} - \phi_{i-1,j}^{t}}{\delta x} \right) - u_y \delta t \left(\frac{\phi_{i,j}^{t} - \phi_{i,j-1}^{t}}{\delta y} \right) - u_y \delta t \left(u_x \delta t \left(\frac{\phi_{i-1,j}^{t} - \phi_{i,j}^{t} + \phi_{i,j-1}^{t} - \phi_{i-1,j-1}^{t}}{\delta y \delta x} \right) \right) \quad (6.1)$$

The integration and the dot product are going to be performed latter, thus identifying coefficients we can isolate the gradient contributions; Those terms in 6.1 being multiplied by $u_x \delta t$ must belong to the first component of the gradient, those being multiplied by $u_y \delta t$ belong to the second component. Finally, those terms being multiplied by $u_y \delta t$ and $u_x \delta t$ correspond to crossed spatial derivatives that can be associated either to the first, the second or both (half to each) components of the gradient. The 2D convective gradient can thus, be computed as:

$$(\nabla \phi)_{convective} = \left(\left(\frac{\phi_{i,j}^{t} - \phi_{i-1,j}^{t}}{\delta x} \right), \left(\frac{\phi_{i,j}^{t} - \phi_{i,j-1}^{t}}{\delta y} + u_x \delta t \left(\frac{\phi_{i-1,j}^{t} - \phi_{i,j}^{t} + \phi_{i,j-1}^{t} - \phi_{i-1,j-1}^{t}}{\delta y \delta x} \right) \right) \right)$$

This method relies on the 1D upwind scheme and depending on the sign of the vector velocity, we take backward or forward spatial differentiation. A positive two-dimensional vector has been used for this demonstration although the convective gradient operator of Saiph is generalized to well-perform for any vector velocity up to three dimensions.

The major drawback of this method is the necessity to be integrating a first-order partial differential equation using a constant time discretization factor. If it is not the case, this method can not be applied.

Identifying convective terms

The mathematical definition of convection $\mathbf{u} \nabla \phi$ is already being identified inside the Saiph compiler, and forced to be computed using the convective gradient, see 3.4.2. What becomes interesting is to identify the equation type (non-derivative, first-order temporal derivative, second-order temporal derivative) containing the term, and apply the specific and adequate convective gradient method to compute it. Within fluids and chemistry theory, this term only appears in first-order PDEs and thus, the above method implementation is going to be used any time the convective term is identified.

6.2.2 Nested derivatives

In order to correctly compute two nested spatial derivatives, as we have seen in section 4.2.2, we should apply a combined derivative scheme using forward and backward

differentiation. When possible, the nesting can also be substituted directly by the corresponding high-order derivative operator ∇^2. Those two numerical scenarios are found within fluids and chemistry problems; They correspond to the diffusive term in which the diffusion coefficient can be spatial dependent or not.

When the diffusion constant is not spatial-dependent we should use the already existing *laplace* operator (3.3.4). If not, we need to combine forward and backward derivative schemes. To do so, we have added to the internal *spatial derivative* function, an argument meant to indicate the derivative scheme to be used for a first order derivative:

- 0, being the default value, forces the central scheme to be used.
- 1 forces the forward scheme to be used.
- 2 forces the backward scheme to be used.

When, internally calling the *derivative* function, it is now possible to specify the derivative scheme to be used. Hence, internal derivative operators (not visible for the users) have been added, for such schemes:

- Forward gradient $\nabla^{forward} = \left(\frac{\partial^{forward}}{\partial x}, \frac{\partial^{forward}}{\partial y}, \frac{\partial^{forward}}{\partial z}\right)$
- Backward gradient $\nabla^{backward} = \left(\frac{\partial^{backward}}{\partial x}, \frac{\partial^{backward}}{\partial y}, \frac{\partial^{backward}}{\partial z}\right)$
- Forward divergence $\nabla^{forward}\cdot = \frac{\partial^{forward}}{\partial x} + \frac{\partial^{forward}}{\partial y} + \frac{\partial^{forward}}{\partial z}$
- Backward divergence $\nabla^{backward}\cdot = \frac{\partial^{backward}}{\partial x} + \frac{\partial^{backward}}{\partial y} + \frac{\partial^{backward}}{\partial z}$

When identifying nested derivatives, those operators should be combined in a correct manner to correctly compute them.

Identifying diffusive terms

Inside the Saiph compiler, we use Scala's pattern matching features to identify when the argument of a standard divergence operator is the result of a multiplication, in which one of the arguments is the result of a gradient operation. The latter, is the definition of the diffusive term. Thus, we identify the diffusion phenomena and try to apply optimizations on it; We use pattern matching again to recognize whenever the other argument is a constant expression. In such cases, the generated code corresponds to a product between the constant and a laplacian. If not, we apply the combined derivative scheme to compute nested derivatives.

Below is the Scala code inside the Saiph compiler that recognizes the patterns and applies the IR node replacement in the application tree:

```
//Divergence of an expression x
def term_div(x: Exp[Term]) = {
x match {
    // Diffusive term optimization
    // The second argument is a gradient
    case (Def(TermTimes(x1, Def(TermGrad(x2))))) =>
           // Replace nested derivative by second order derivative
           if (x1 match {
                      case Def(ConstTermNew(_, _, _ , _)) => true
                      case _ => false
               }) TermTimes(x1, TermLapla(x2))
               // Replace central derivatives by combination of forward
                  and backward derivatives
                 else TermDivForward(TermTimes(x1, TermGradBackward(x2)))

    // The first argument is a gradient
    case (Def(TermTimes(Def(TermGrad(x1)), x2))) =>
           // Replace nested derivative by second order derivative
           if (x2 match {
                      case Def(ConstTermNew(_, _, _ , _)) => true
                      case _ => false
               }) TermTimes(TermLapla(x1), x2)
               // Replace central derivatives by combination of forward
                  and backward derivatives
                 else TermDivForward(TermTimes(TermGradBackward(x1), x2))

    // Otherwise, just compute a standard divergence
    case _ => TermDiv(x)
    }
}
```

The following figure illustrates in a schematic way the pattern matching driving to the emission of correct (or more adequate) IR nodes in the application tree, when facing a diffusive term.

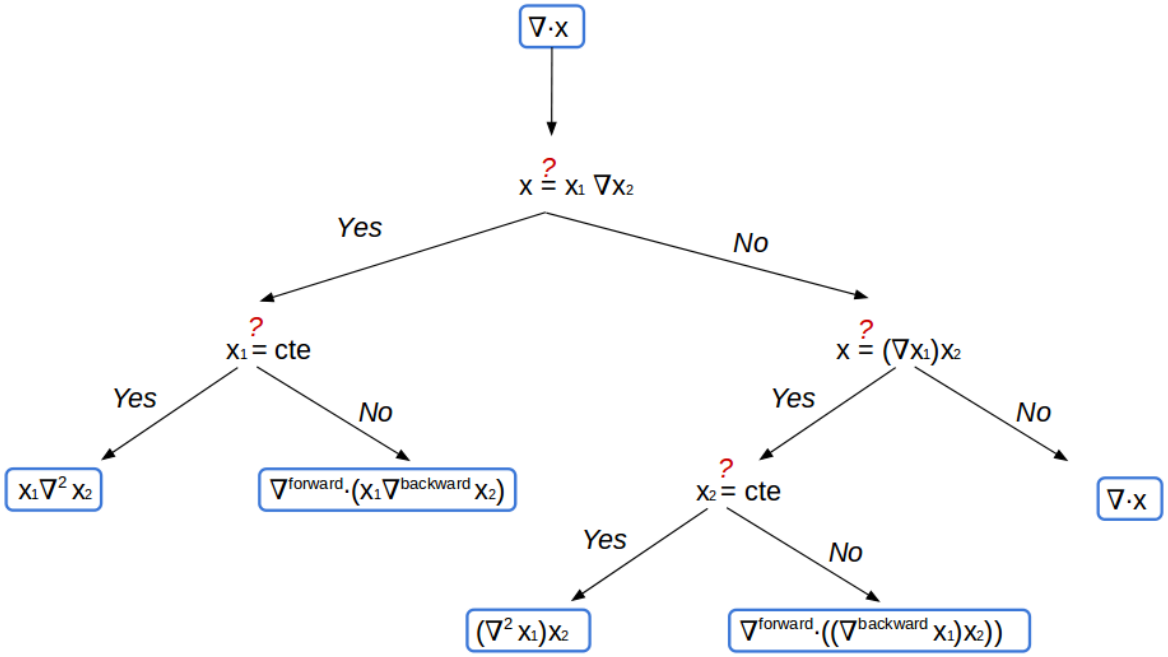

Fig. 6.2: Schematized pattern matching for diffusive term optimizations

7 | Results and evaluations

This chapter is devoted to show the complete use-cases that have been simulated with Saiph and some scalability results of the most expensive application.

7.1 Fluid mechanics and chemistry use-cases

The use-cases that have driven the development of Saiph are shown below. We present a complete theoretical definition and translation of the problems, to Saiph applications, for each of the use-cases faced. Results, validation and nomenclature are also reported. Those use-cases put together all the language features shown along this document.

7.1.1 Convection

This first application corresponds to the simulation of two different phenomenas and the combination of them. This is not a real use-case, it is only meant to verify the computation of basic fluid terms that are going to be used for the *real* use-cases and the correct behaviour of the main method's implementation and optimizations presented above.

Background

As said, convection is the transport of a fluid, through the combination of advection and diffusion phenomenas. In order to correctly model this combined phenomena, we firstly simulate advection and diffusion separately. After visual validations, we end up simulating the combination of them, the convection. Each of those phenomenas are mathematically translated as follows:

- Advection equation

$$\frac{\partial T}{\partial t} = -\mathbf{u} \cdot \nabla T \tag{7.1}$$

- Diffusion equation

$$\frac{\partial T}{\partial t} = \nabla \cdot (k \nabla T) \tag{7.2}$$

- Convection equation

$$\frac{\partial T}{\partial t} = -\mathbf{u} \cdot \nabla T + \nabla \cdot (k \nabla T) \tag{7.3}$$

This equations are translated to Saiph as follows:

For the advection application:

```
val advection = Equation(dt(T), -u*grad(T))
```

For the diffusion application:

```
val difussion = Equation(dt(T), div(k*grad(T)))
```

For the convection application:

```
val convection = Equation(dt(T), -u*grad(T) + div(k*grad(T)))
```

Initial and boundary conditions

We simulate a small 3D hot cube. The domain is initialized as

$$T(x, y, z, 0) = \begin{cases} 300K & \text{for } \frac{X_{mesh}}{2} - X_{cube} \leq x, y, z \geq \frac{X_{mesh}}{2} + X_{cube} \\ 100K & \text{otherwise} \end{cases}$$

This cube is moving along two dimensions[(1)] by the advection phenomena, due to a constant velocity vector set as $\mathbf{u} = (1, -1, 0)$. The advection, when present, is forced to be periodic through the periodic boundary condition applied at each direction of the movement. Regarding diffusion, a constant diffusion coefficients k has been used.

Those initial and boundary conditions are translated to Saiph as follows:

[(1)] 1D or 3D convection can also be simulated. For the sake of illustrative comprehension, we present the two-dimensional.

```
// Function to initialize the magnitude being transported
def cube(x: Rep[MUnit], y: Rep[MUnit], z: Rep[MUnit]) = {
   if (x >= CUBEX - EDGE_SIZE && x <= CUBEX + EDGE_SIZE &&
     y >= CUBEY - EDGE_SIZE && y <= CUBEY + EDGE_SIZE &&
     z >= CUBEZ - EDGE_SIZE && z <= CUBEZ + EDGE_SIZE) HOT_TEMP
   else AMBIENT_TEMP
}
    ...
// Periodic boundaries
mesh.setPeriodic(DirX)
mesh.setPeriodic(DirY)

// Initializing Variables
val T = Term(Temperature)("Temperature", mesh, cube _)

val u = ConstTerm(Speed)("Velocity", V_INIT, List("X", "Y", "Z"))
val k = ConstTerm(M2_s)("Diffusion coeff", DIFFUSION_COEF)
```

Results

- Advection
  Figures in 7.1 show the results of the simulation by different temporal frames. Starting at $t = 0$ and transporting the cube until it reaches the initial position again.

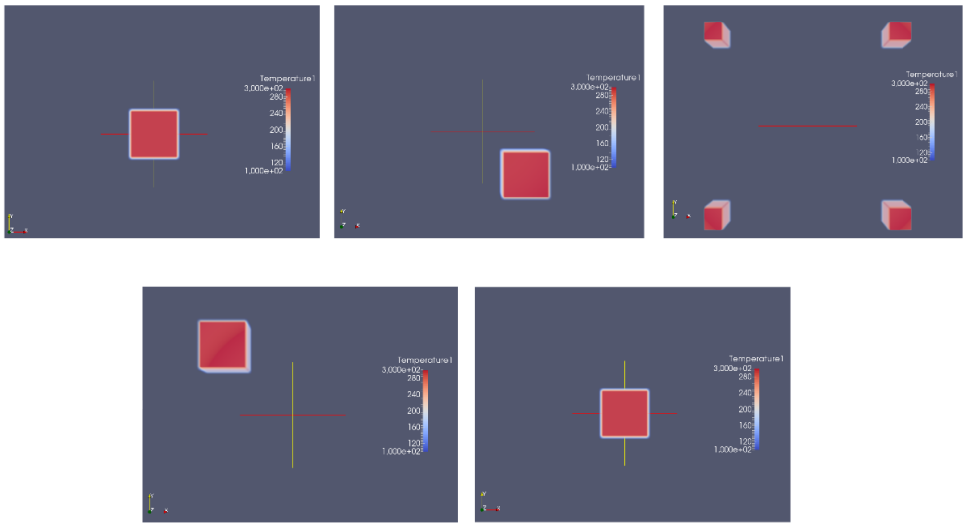

Fig. 7.1: Two-dimensional advection

- Diffusion
  For the diffusion application we show the initial state and a latter one illustrating how temperature is transported only by the diffusion phenomena.

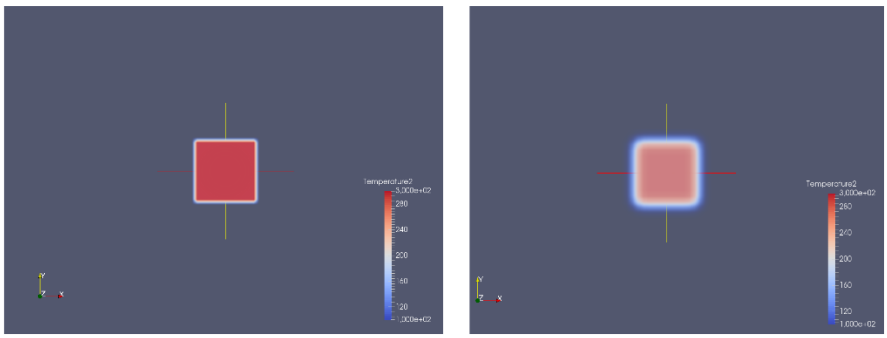

Fig. 7.2: Two-dimensional diffusion

- Advection & Diffusion
  Finally, figure 7.3 shows the results from the combination of the advection and the diffusion phenomena.[(2)]

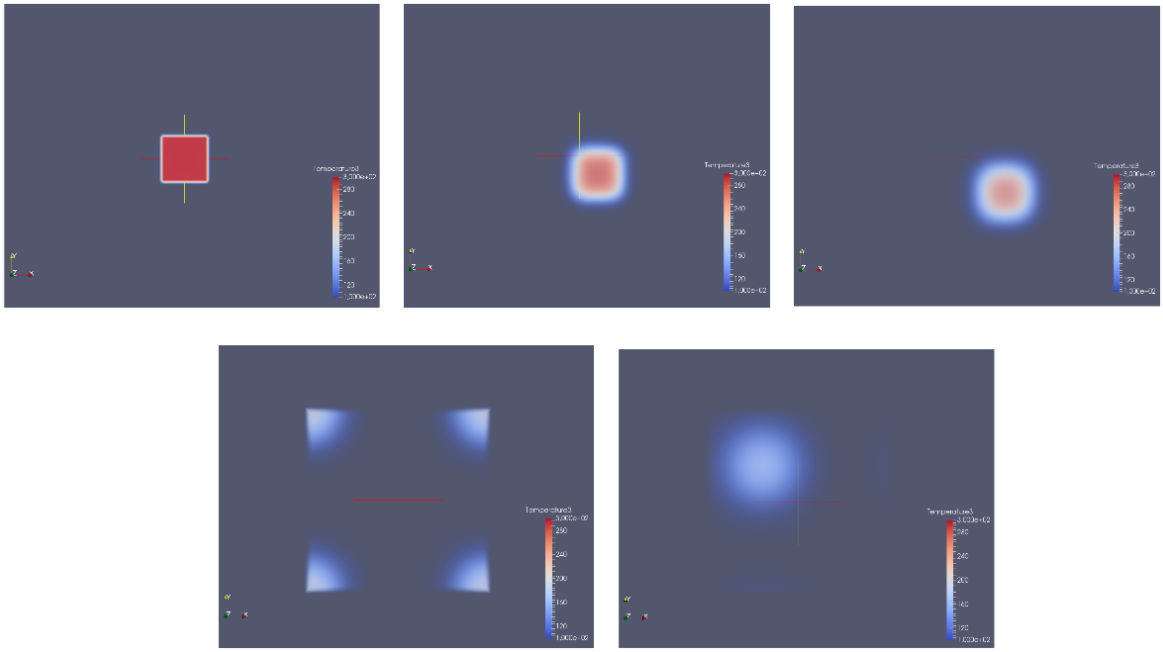

Fig. 7.3: Two-dimensional convection

[(2)]In order to combine, and visually identify advection and diffusion phenomenas, the coefficients and initial values of the convection application have been adequately modified with respect to the ones from advection and diffusion applications.

7.1.2 Sod's shock tube

Background

This is a use case of a temporal integration scheme based on properties updated at different stages. The problem corresponds to the well known shock tube originally proposed by Sod[20]. It is a transient case that has analytical solution. It consists of a one-dimensional tube, closed at its ends and divided into two equal regions by a thin diaphragm. Each region is filled with the same gas, but with different thermodynamic parameters; the left and right sides of the tube present different density, energy and pressure. The region with the highest pressure is called the *driven section* of the tube, while the low-pressure part is the *working section*.
The fluid is initially at rest, that is, it has zero initial velocity. It starts to move because of the sudden breakdown of the diaphragm, the discontinuous initial condition of the left and right states entails a shock wave that propagates to both left and right sides.
The equations that are being solved correspond to the Euler equations, a particular case of the Navier-Stokes equations for which the viscous forces are neglected. The system is governed by the continuity, momentum and energy transport equations and closed by the equation of state. The problem is assumed to be one-dimensional (1D) and the 1D equations are given by:

- Continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

$$\Rightarrow \boxed{\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{u})} \tag{7.4}$$

- Momentum equation:

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) = -\nabla p$$

$$\Rightarrow \boxed{\frac{\partial (\rho u_i)}{\partial t} = -\left[\rho \mathbf{u} \cdot \nabla u_i + u_i \nabla \cdot (\rho \mathbf{u}) + \frac{\partial p}{\partial x_i}\right]} \tag{7.5}$$

- Energy equation:

$$\frac{\partial (\rho E)}{\partial t} + \nabla \cdot (\rho \mathbf{u} E) = -\nabla \cdot (\mathbf{u} p)$$

$$\Rightarrow \boxed{\frac{\partial (\rho E)}{\partial t} = -\left[\rho \mathbf{u} \cdot \nabla E + E \nabla \cdot (\rho \mathbf{u}) + \mathbf{u} \cdot \nabla p + p \nabla \cdot \mathbf{u}\right]} \tag{7.6}$$

- State equation:

$$\Rightarrow \boxed{p = (\gamma - 1)\left((\rho E) - \frac{1}{2}(\rho \mathbf{u})\,\mathbf{u}\right)} \tag{7.7}$$

The fluid behaves like an ideal gas, so γ is assumed to be constant and equal to $\gamma = 1.4$. To control the Mach regimes of this use-case we also compute the Mach number as

$$\Rightarrow \boxed{Ma = \frac{u_x}{c}} \tag{7.8}$$

where c is the speed of sound in the medium

$$c = \sqrt{\gamma\left(\frac{p}{\rho}\right)}$$

The problem contains 4 unknowns: ρ, (ρu), (ρE) and p and has 4 governing equations (Equation 7.4, 7.5, 7.6 and 7.7) which should be solved in the appropriate order. The full system can be expressed as:

$$\rho^{t+1} = \rho^t + \delta t \cdot f((\rho u)^t)$$

$$(\rho u)^{t+1} = (\rho u)^t + \delta t \cdot f((\rho u)^t, u^t, p^t)$$

$$(\rho E)^{t+1} = (\rho E)^t + \delta t \cdot f((\rho u)^t, u^t, E^t, p^t)$$

$$p^{t+1} = f((\rho E)^{t+1}, (\rho u)^{t+1}, u^{t+1})$$

This governing equations are translated to Saiph as follows:

```

  // Equation 1: Continuity equation
  val density = Equation(dt(rho), -(div(rho_u)))

  // Equation 2: Momentum equation (rho*u)
  val mass_flow = Equation(dt(rho_u), -( (u.comp * div(rho_u)) + (rho_u
     * grad(u.comp)) + (grad(p)).comp))

  // Equation 3: Energy equation (rho * E)
  val rho_energy = Equation(dt(rho_e), -( (e * div(rho_u)) + (rho_u *
     grad(e)) + (u * grad(p)) + (p * div(u))))

```

```
11  // Equation 4: State equation (Pressure)
12  val pressure = Equation(p, (gamma - unit) * (rho_e.toff(0) - (cte *
        rho_u.compWithOffset(0) * u.compWithOffset(0))))
```

Initial and Boundary conditions

The initial state is defined with a diaphragm in the middle of the tube separating the two states. The density, energy and pressure are discontinuous across the diaphragm at $t = 0$ with values [21]:

$$\rho(x,0) = \begin{cases} 1.0 & \text{for } x < \frac{1}{2}x_{tube} \\ 0.125 & \text{for } x > \frac{1}{2}x_{tube} \end{cases},$$

$$E(x,0) = \begin{cases} 2.5 & \text{for } x < \frac{1}{2}x_{tube} \\ 2 & \text{for } x > \frac{1}{2}x_{tube} \end{cases},$$

$$p(x,0) = (\gamma - 1)\rho(x,0)E(x,0) = \begin{cases} 1.0 & \text{for } x < \frac{1}{2}x_{tube} \\ 0.1 & \text{for } x > \frac{1}{2}x_{tube} \end{cases}$$

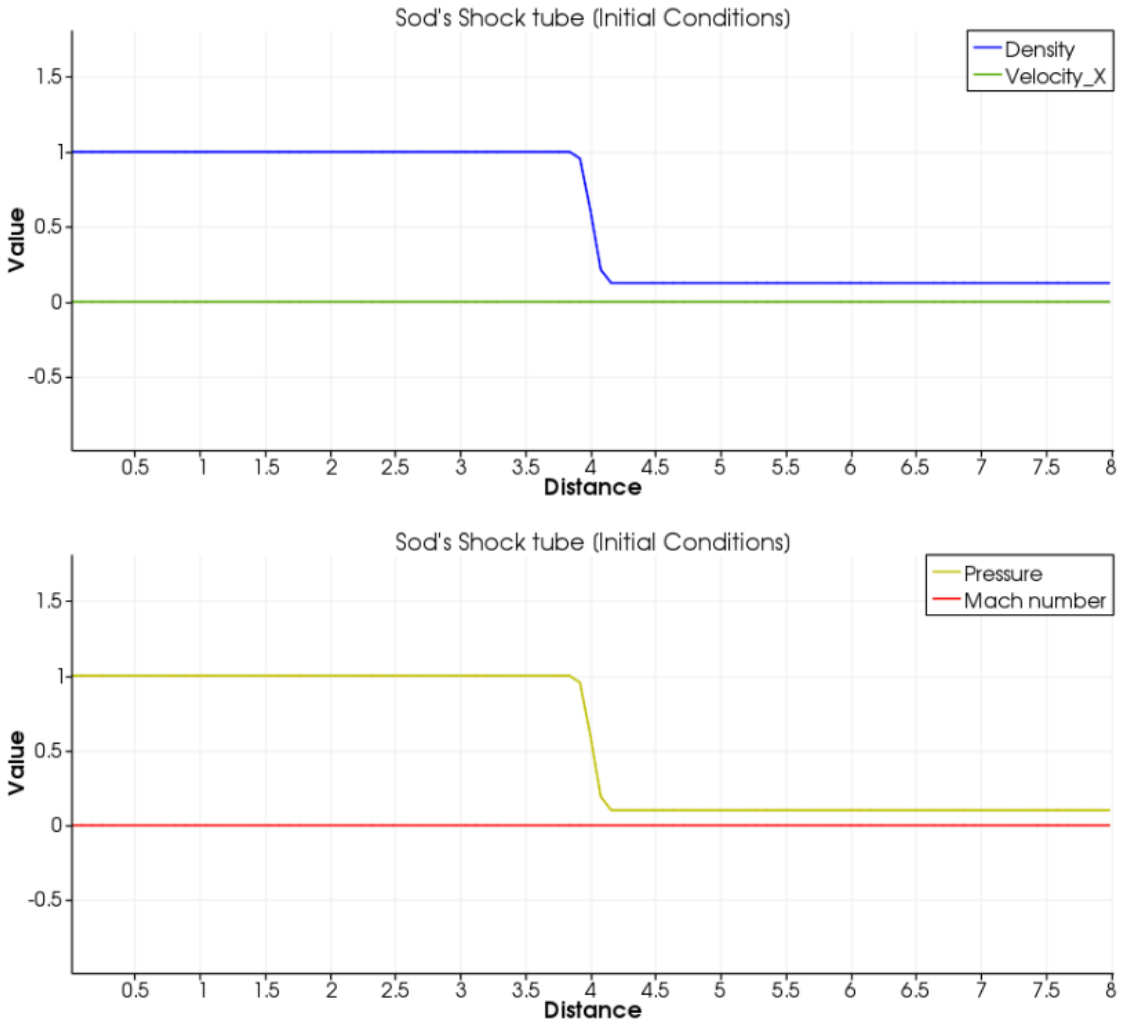

Fig. 7.4: Sod's shock tube profiles of main variables at t=0

This initial conditions are translated to the code as follows:

```
    // Initial values (with units)

      // Domain parameters
          // Discontinuity thinness
          def JUMP = 10 * H;
          def preJUMP = (XSIZE - JUMP)/2;
          def postJUMP = preJUMP + JUMP - H;

              ...

      // Variables
```

```
12            // Density
13            def Kg_m3 = Kilograms / Meters3;
14            def DENSITY_LEFT = 1.0 * Kg_m3;
15            def DENSITY_RIGHT = 0.125 * Kg_m3;
16
17            // Velocity
18            def MpSec = MetersPerSecond;
19            def V_INIT = Vector(0 * MpSec, 0 * MpSec, 0 * MpSec);
20
21                ...
22
23    // Initialization
24
25        // Variables
26            // Density
27            val rho = Term(Kg_m3)("Density", mesh, { x =>
28                if (x < preJUMP) DENSITY_LEFT
29                else if (x > postJUMP) DENSITY_RIGHT
30                else DENSITY_LEFT + (((DENSITY_RIGHT-DENSITY_LEFT)/JUMP)
                      *(x-preJUMP))
31            })
32
33            // Velocity
34            val u = Term(Speed)("Velocity", mesh, V_INIT, List[Rep[String
                  ]]("X", "Y", "Z"))
35
36                ...
```

At $t = 0$ the diaphragm is broken and the system evolves in time. The tube is supposed to be closed at both ends but the computation stops before the waves reach the end-walls of the tube. Note that the boundary conditions should not influence the solution as they are defined far away from the region of interest.

Physical Description

At time $t = 0$, when the diaphragm breaks, a process that naturally tends to equalize the pressure in the tube is generated. The gas at high pressure (L) expands through an *expansion wave* and flows into the working section, pushing the gas of this part. This expansion is a continuous process that takes place inside a well-defined region (the expansion fan) and propagates to the left (region (E), whose width grows in time).

The compression of the low-pressure gas (R) generates a *shock wave* propagating to the right. The expanding gas is separated from the gas being compressed by a *contact discontinuity*, which can be regarded as a fictitious membrane travelling to the right at constant speed [22].

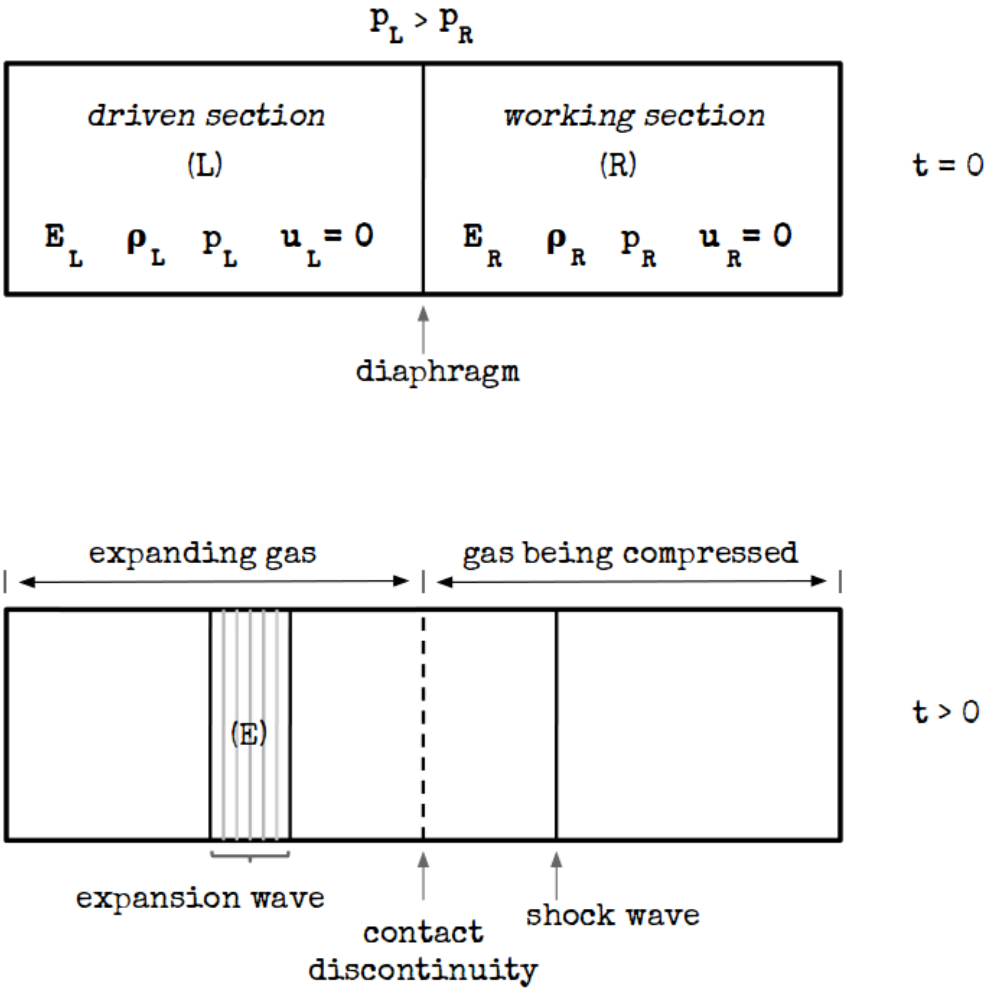

Fig. 7.5: Sketch of the initial configuration of the shock tube ($t = 0$) and waves propagating in the tube after the diaphragm breakdown ($t > 0$)

7.1.3 Results and validation

Figure 7.6 shows the results of the simulation showing profiles of the main variables at $t > 0$. For this use-case, the results from Moragues (2015) [23] are used for comparison. The results indicate an acceptable level of correlation with the reference data for all quantities.

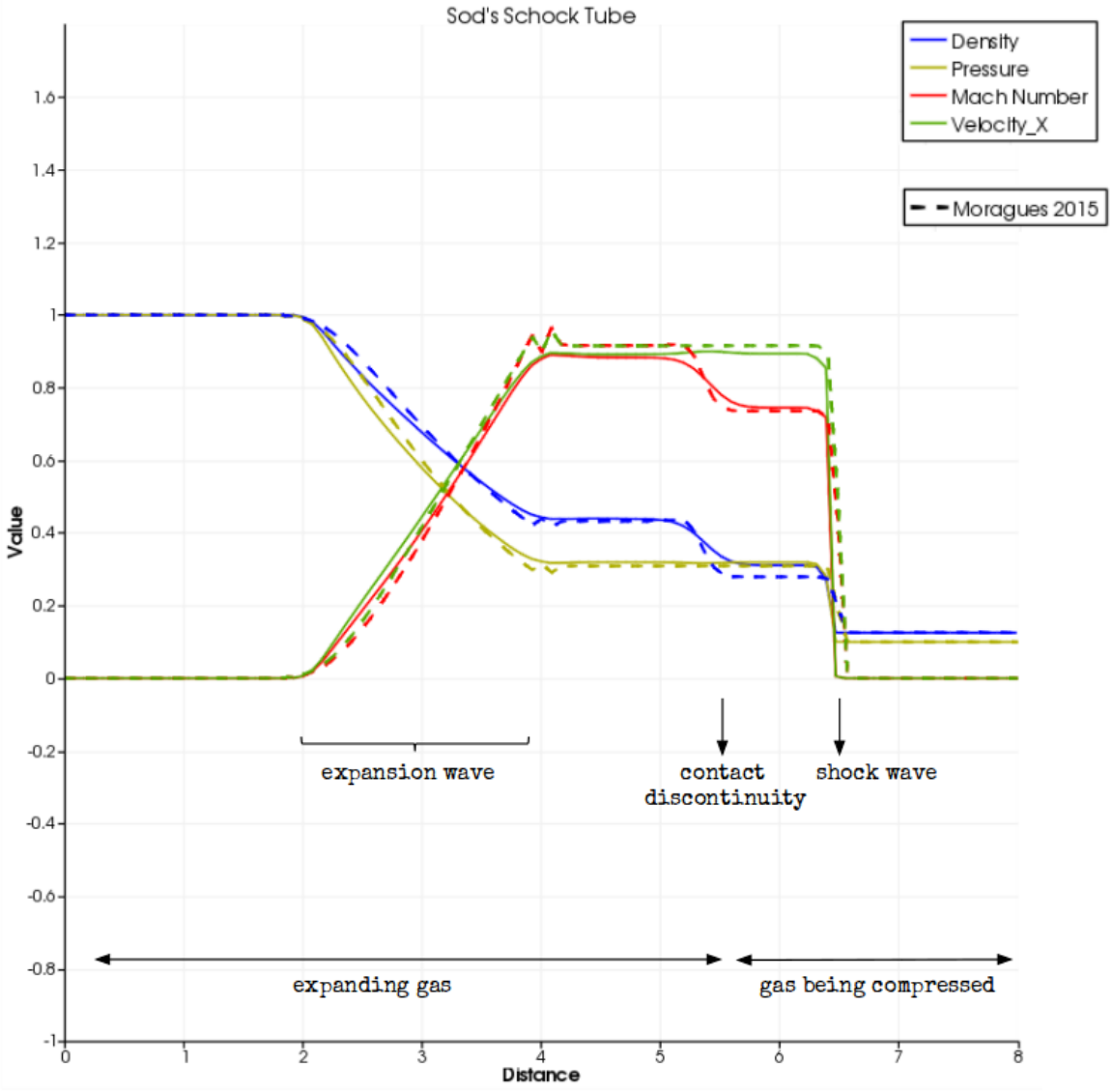

Fig. 7.6: Resulting profiles of the shock tube use-case at $t > 0$ and its validation results

7.1.4 Autoignition delay time

Background

The study of combustion systems is of primary interest to reduce global warming and requires the investigation of alternative yet more efficient fuel blends. Combustion performance is dependent of many factors such as fuel composition, mixture preheating and system configuration. An important aspect in engine development and research is the ignition delay time τ_{ign}, since the residence time of the combustor can induce

undesired phenomena depending on the autognition delay time of the fuel. There is a need to increase the autoignition characteristics of methane-based fuels to gain a better understanding of the effect of fuel composition on ignition phenomena. This use-case aims to simulate the temporary evolution of a premixed fuel/air mixture at a given initial temperature to determine its autoignition delay time.

The equations that are being solved correspond to the conservation of each specie k and temperature:

- Species equations:

$$\Rightarrow \boxed{\frac{\partial(\rho Y_k)}{\partial t} = \dot{\omega}_k} \tag{7.9}$$

- Temperature equation:

$$\frac{\partial(\rho c_p T)}{\partial t} = \dot{\omega}_T$$

$$\Rightarrow \boxed{\frac{\partial(\rho T)}{\partial t} = \frac{1}{c_p}\dot{\omega}_T} \tag{7.10}$$

c_p is assumed constant.

The problem contains $(N+1)$ unknowns: (ρY_k) and (ρT) and has $(N+1)$ governing equations (Equation 7.9 and 7.10). The full system can be expressed as:

$$(\rho Y_k)^{t+1} = (\rho Y_k)^t + \delta t \cdot f_1 \left(\left(\prod_{k=1}^{N} (\rho Y_k) \right)^t, T^t \right)$$

$$(\rho T)^{t+1} = (\rho T)^t + \delta t \cdot f_2 \left(\left(\prod_{k=1}^{N} (\rho Y_k) \right)^t, T^t \right)$$

The governing equations are translated to Saiph as follows:

```
1
2       // Rate constant
3       val Kf = Af * exp(Ea / (R*T))
4       // Species concentrations
5       val XCH4 = rho_Y(0) / MolarMassVector(0)
6       val XO2 = rho_Y(2) / MolarMassVector(2)
7       // Reaction rate progress
8       val Q = Kf * XCH4 * XO2 * XO2
9       // Mass rate
```

```
10        val omega_k = cwiseprod(MolarMassVector, nu) * Q
11        // Heat release
12        val omega_T = -(sum(cwiseprod(h, omega_k)));
13
14
15        // Equation1: Species equation
16        val rho_species = Equation(dt(rho_Y), omega_k.comp)
17
18        // Equation2: Temperature equation
19        val rho_temperature = Equation(dt(rho_T), omega_T / cp)
```

Initial and Boundary conditions

The initial conditions correspond to a premixed methane flame at fixed equivalence ratio and under constant initial temperature, using a **1-step** chemical mechanism ($K_b = 0$). The chemical reaction rate is given by:

$$\mathrm{CH_4} + 2\,\mathrm{O_2} \longrightarrow \mathrm{CO_2} + 2\,\mathrm{H_2O}$$

where the reaction coefficients are

Specie	ν'_k	ν''_k	ν_k
CH_4	1	0	-1
N_2	0	0	0
O_2	2	0	-2
CO_2	0	1	1
H_2O	0	2	2

Table 7.1: Reaction coefficients

which leads to

$$Q = K_f \left(\frac{\rho Y_{CH_4}}{W_{CH_4}}\right)^1 \cdot \left(\frac{\rho Y_{O_2}}{W_{O_2}}\right)^2$$

Simulations are performed using two different initial gas compositions

Mass fraction	ER= 0.8	ER= 1.0
Y_{CH_4}	0.0445	0.0550
Y_{N_2}	0.7328	0.7248
Y_{O_2}	0.2227	0.2202
Y_{CO_2}	0.0	0.0
Y_{H_2O}	0.0	0.0

Table 7.2: Two initial mass fractions for gas composition at ER $= 0.8$ and ER $= 1.0$

and different initial temperatures T={1000 K, 1100 K, ...,1900 K, 2000 K}

The fluid is supposed to behave as a perfect gas, so the equation of state holds, the density is initialized as $\rho = \frac{Wp}{R^o T}$ where R^o is the ideal gas constant and W, the molar mass of the mixture is $\frac{1}{W} = \sum_k \frac{Y_k}{W_k}$.

Finally, the pre-exponential factor A_f and the activation energy E_a are coefficients depending on the composition of the gas. Simulations have been performed using values from two well-established chemical kinetics mechanisms: Mantel at al. [24] and Liñán et al. [25]:

Reference	$A_f \left(\frac{m^6}{mol^2 s}\right)$	$E_a \left(\frac{kJ}{mol}\right)$
1-step Mantel et al. (1996) [24]	$1.1 \cdot 10^{10}$	167.360
1-step Liñán et al. (2006) [25]	$6.9 \cdot 10^{8}$	132.200

Table 7.3: Reference coefficients for the 1-step chemical mechanism of the $CH_4 + 2\,O_2 \longrightarrow CO_2 + 2\,H_2O$ reaction

Note that the boundary conditions should not influence the solution as this application is restricted to study the temporary evolution of the system.

This initial conditions are translated to Saiph as follows:

```
1     // Initial values (with units)
2
3       // Constants
4           // Ideal gas constant
5           def JpMK = Joules / Moles / Kelvins
6           def R_VALUE = 8.314462175 * JpMK
7           // Activation energy
8           def JpMol = Joules / Moles
9           def EA_VALUE_MANTEL = -167360 * JpMol
10          def EA_VALUE_LINAN = -132200 * JpMol
11              ...
```

```
12
13          // Variables
14              // Mass fractions
15              def INIT_Y = Vector(0.0445 * Unitless, 0.7324 * Unitless,
                    0.2231 * Unitless, 0 * Unitless, 0 * Unitless)
16              // Temperature
17              def INIT_T = 1500 * Kelvins
18              // Pressure
19              def INIT_P = 1 * Atmospheres
20                  ...
21
22      // Initialization:
23
24          // Constants
25              val R = ConstTerm(JpMK)("Ideal gas constant", R_VALUE)
26              val Ea = ConstTerm(JpMol)("Activation Energy",
                    EA_VALUE_MANTEL)
27                  ...
28
29          // Variables
30              val rho_T = Term(Kg_ms2)("Rho*T", mesh, DENSITY*INIT_T)
31              val rho_Y = Term(Kg_m3)("Rho*Y", mesh, INIT_rho_Y, GAS_NAMES)
32                  ...
```

Physical Description

Departing from an initial composition and thermal state $(p, T)_0$, the system evolves oxidizing the fuel until either the fuel or oxidizer are totally consumed.
The consumption time of reactants is measured for each temperature investigated. As an example, the evolution of the system using the chemical scheme from Mantel et al. (1996) at T = 1500K is shown in figure 7.7.

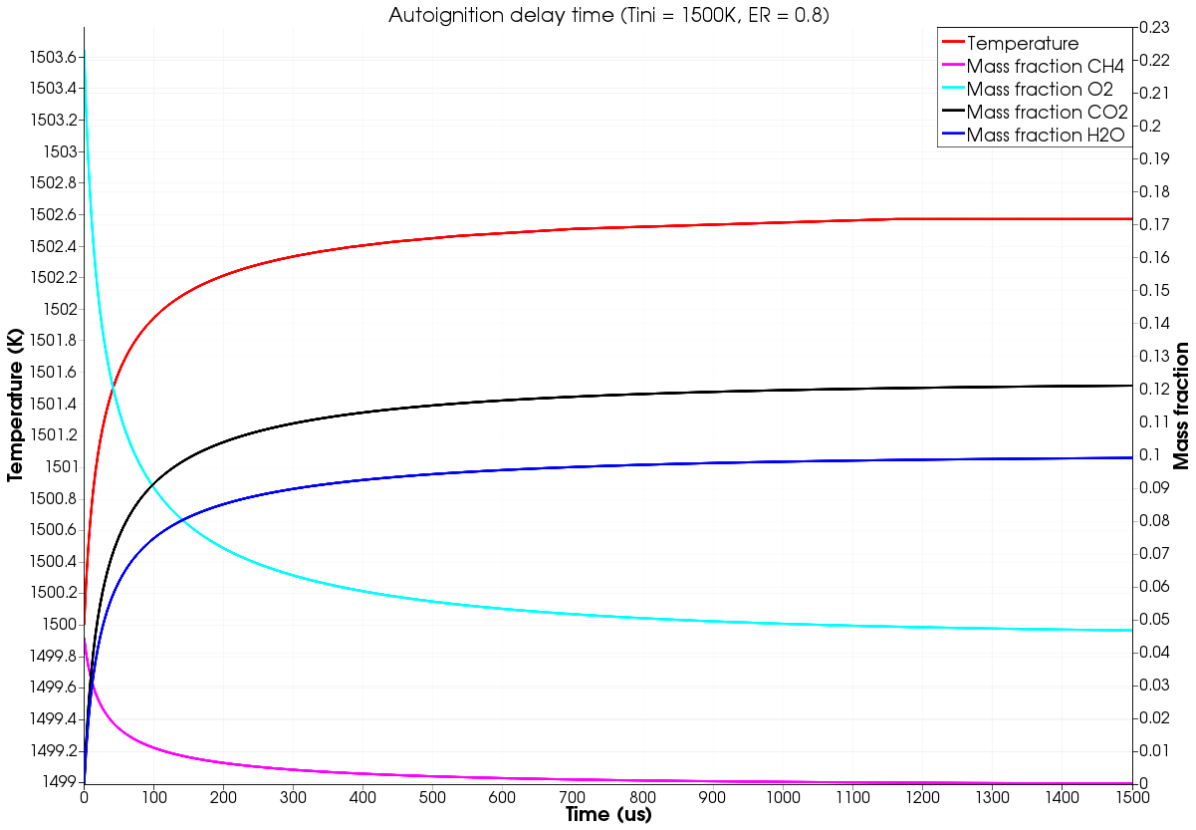

Fig. 7.7: Methane autoignition temporal evolution. Chemical scheme from Mantel et al. (1996) at $T_{ini} = 1500K$ and chemical composition ER=0.8

Results and validation

Developing the tests for the two-schemes and the two compositions, the autoignition delay times of the mixtures under investigation can be obtained for different initial temperatures. The results are shown in figure 7.8 and 7.9.

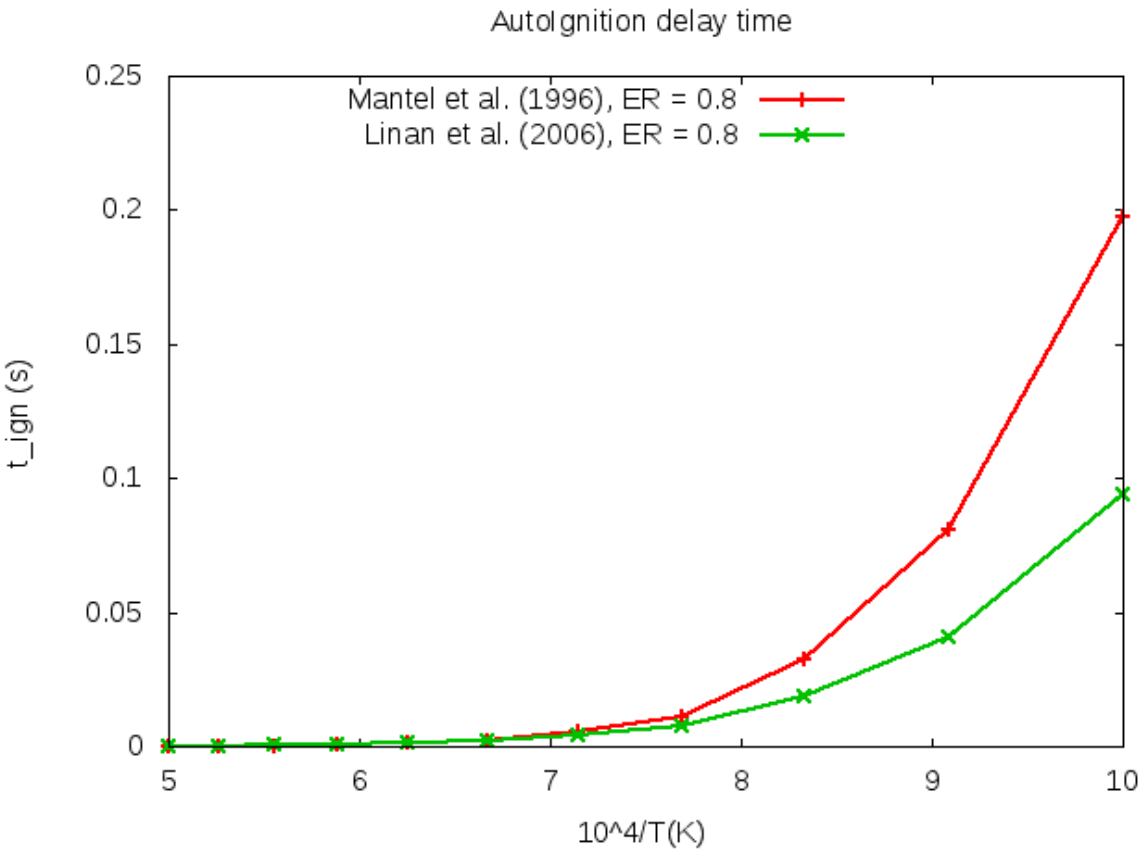

Fig. 7.8: Methane autoignition delay time measurements using Mantel et al. (1996) and Liñán et al. (2006) chemical schemes, with $\mathrm{ER} = 0.8$

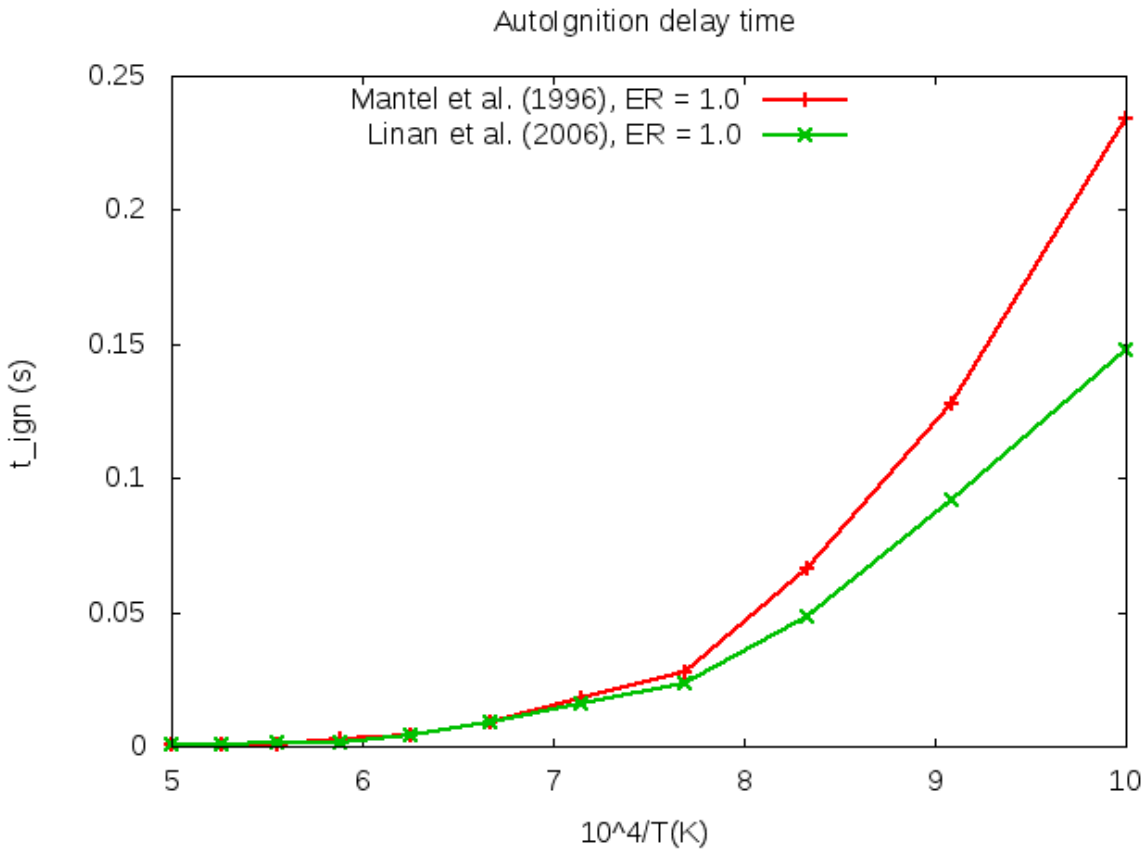

Fig. 7.9: Methane autoignition delay time measurements using Mantel et al. and Liñán et al. chemical schemes, with ER $= 1.0$

For this use-case, the results from Holton (2008) [26] are used for comparison. The results, in figure 7.10 indicate an acceptable level of correlation. For clarity purposes simulation results and reference data have been plotted using a logscale in time axis.

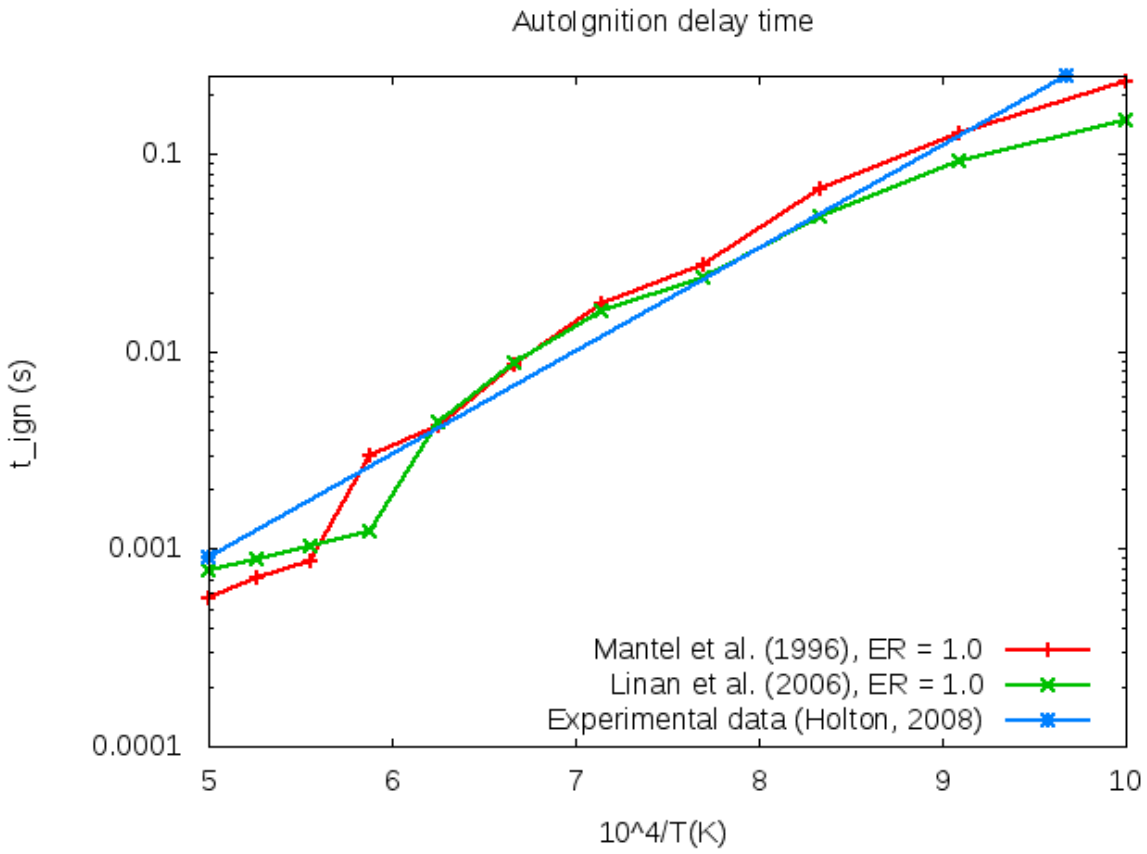

Fig. 7.10: Methane autoignition delay time measurements (ER = 1.0) plotted alongside reference data (Holton (2008)[26])

7.1.5 Premixed laminar flame

This is the final use-case enclosing fluids mechanics and chemistry domains, stated as the final objective of this project. The premixed laminar flame contains all the theoretical terms seen in Chapter 4 and being progressively validated by the above simpler use-cases. The complete Saiph code of this application can been seen in Appendix C.

Background

The premixed laminar flame is another example use-case of a temporal integration scheme based on properties updated at different stages. The problem to be solved corresponds to a premixed, freely propagating, steady, laminar, flat flame. The system is characterized by a premixed mixture of fuel (CH_4) and air entering in the reaction zone of the computational domain, where it reacts chemically releasing energy.
The problem is assumed to be one-dimensional (1D); the domain represents a tube in

which the dimension of motion is much longer than the vertical and spanwise directions. The initial conditions correspond to a situation of unburnt and burnt conditions separated in the center of the tube. The flame front will be located at this position. Each of these regions is initialized with its corresponding physical conditions: density, velocity, species concentrations, temperature and pressure [27].
The equations being solved correspond to the compressible Navier-Stokes equations for multi-species reactive flows with constant multicomponent mixture properties. In this case, the system is governed by the transport equations of continuity, momentum, species and energy, and it is closed by the equation of state [12]:

- Continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

$$\Rightarrow \boxed{\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{u})} \tag{7.11}$$

- Momentum equation:

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot \tau$$

where τ is the viscous tensor: $\tau = \mu \left(\left(\nabla \mathbf{u} + \nabla^T \mathbf{u} \right) - \frac{2}{3} \nabla \cdot \mathbf{u} \right)$

$$\Rightarrow \boxed{\frac{\partial (\rho u_x)}{\partial t} = -\left[\rho \mathbf{u} \cdot \nabla u_x + u_x \nabla \cdot (\rho \mathbf{u}) + \frac{\partial p}{\partial x} \right] + \frac{4}{3} \mu \frac{\partial^2 u_x}{\partial x^2}} \tag{7.12}$$

- Species equation:
N species: $k = 1...N$

$$\frac{\partial (\rho Y_k)}{\partial t} + \nabla \cdot (\rho \mathbf{u} Y_k) = \dot{\omega}_k + \nabla \cdot \left(\frac{\lambda}{c_p} \nabla Y_k \right)$$

$$\Rightarrow \boxed{\frac{\partial (\rho Y_k)}{\partial t} = -\left[(\rho \mathbf{u}) \cdot \nabla Y_k + Y_k \nabla \cdot (\rho \mathbf{u}) \right] + \nabla \cdot \left(\frac{\lambda}{c_p} \nabla Y_k \right) + \dot{\omega}_k} \tag{7.13}$$

- Temperature equation:

$$\frac{\partial (\rho c_p T)}{\partial t} + \nabla \cdot (\rho c_p \mathbf{u} T) = \nabla \cdot (\lambda \nabla T) + \dot{\omega}_T$$

$$\Rightarrow \boxed{\frac{\partial (\rho T)}{\partial t} = \frac{1}{c_p} \left[-c_p \left((\rho \mathbf{u}) \cdot \nabla T + T \nabla \cdot (\rho \mathbf{u}) \right) + \nabla \cdot (\lambda \nabla T) + \dot{\omega}_T \right]} \tag{7.14}$$

- State equation:

$$\boxed{p = \rho R^o T \cdot \sum_k^N \frac{Y_k}{W_k}} \tag{7.15}$$

The problem contains $(4 + N)$ unknowns: ρ, ρu_x, ρT, ρY_k and p and has $(4 + N)$ governing equations (Equation 7.11, 7.12, 7.13, 7.14 and 7.15), which should be solved in the appropriate order. The full system can be expressed as:

$$\rho^{t+1} = \rho^t + \delta t \cdot f((\rho u)^t)$$

$$(\rho u)^{t+1} = (\rho u)^t + \delta t \cdot f((\rho u)^t, u^t, p^t)$$

$$(\rho T)^{t+1} = (\rho T)^t + \delta t \cdot f((\rho u)^t, (\rho Y_k)^t, T^t)$$

$$(\rho Y_k)^{t+1} = (\rho Y_k)^t + \delta t \cdot f((\rho u)^t, (\rho Y_k)^t, T^t)$$

$$p^{t+1} = f((\rho T)^{t+1}, Y_k^{t+1})$$

This governing equations are translated to Saioh language as follows:

```
// Equation 1: Continuity equation
val density = Equation(dt(rho), -div(rho_u))

// Equation 2: Momentum equation
val momentum = Equation(dt(rho_u), -((rho_u * grad(u.comp)) + (u.comp *
    div(rho_u)) + (grad(p)).comp) + (mu_cte * mu * der2(DirX)(u.comp)))

// Auxiliar equations: Species source term & heat release
    // Reaction rate progress
      val Q = Af * exp(Ea / (R*T)) * (rho_Y(0) / MolarMassVector(0)) * (
         rho_Y(2) / MolarMassVector(2) * rho_Y(2) / MolarMassVector(2))

val omegak = Equation(omega_species_k, (MolarMassVector.comp * nu.comp *
     Q))
val omegaT = Equation(omega_heat, - (sum(cwiseprod(h, omega_species_k)))
    )

```

```
// Equation 3: Species equations
    // Convective & Diffusion term
    val Yconv = - ((rho_u * grad(Y.comp)) + (Y.comp * div(rho_u)))
    val Ydiff = div((lambda / cp) * grad(Y.comp))
val rho_species = Equation(dt(rho_Y), Yconv + Ydiff + omega_species_k.
    comp)

// Equation 4: Temperature equation
    // Convective & Diffusion term
    val Tconv = - (cp * ((rho_u * grad(T)) + (T * div(rho_u))))
    val Tdiff = div(lambda * grad(T))
val rho_temperature = Equation(dt(rho_T), (Tconv + Tdiff + omega_heat) /
     cp)

// Equation 5: State equation (pressure)
val pressure = Equation(p, rho_T.toff(0) * R * sum(Y.toff(0) /
    MolarMassVector))
```

Initial and Boundary conditions

The initial state is defined with a flame front in the middle of the tube separating the two states: fresh gases or unburnt conditions and products or burnt conditions. The temperature, mass fraction, velocity and density are discontinuous across the flame at $t = 0$. The pressure is initially constant and equal to the atmospheric pressure.

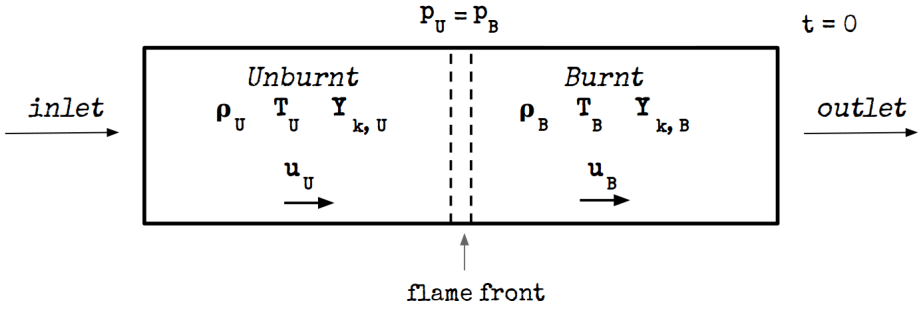

Fig. 7.11: Domain and initial conditions for the one-dimensional premixed flame

The initial conditions in the *unburnt* section correspond to a premixed methane flame at fixed equivalence ratio ER = 0.8 and under constant initial pressure. The *burnt* conditions are initialized at equilibrium conditions.
The chemical reaction:

$$CH_4 + 2\,O_2 \longrightarrow CO_2 + 2\,H_2O$$

is defined using a 1-step chemical mechanism (K_b= 0). [(3)]

Initial values are:

$$T(x,0) = \begin{cases} 300 \text{ K} & \text{for } x < \frac{1}{2}x_{Unburnt} \\ 2011 \text{ K} & \text{for } x > \frac{1}{2}x_{Burnt} \end{cases},$$

$$p(x,0) = 101325 \text{ Pa},$$

Mass fraction	Unburnt	Burnt
Y_{CH_4}	0.0445	0.0
Y_{N_2}	0.7328	0.7328
Y_{O_2}	0.2227	0.0360
Y_{CO_2}	0.0	0.1270
Y_{H_2O}	0.0	0.1042

Table 7.4: Initial mass fractions for gas composition at $\text{ER} = 0.8$

[(3)]See the Autoignition use-case chapter 7.1.4 for more details regarding chemical reactions and its coefficients

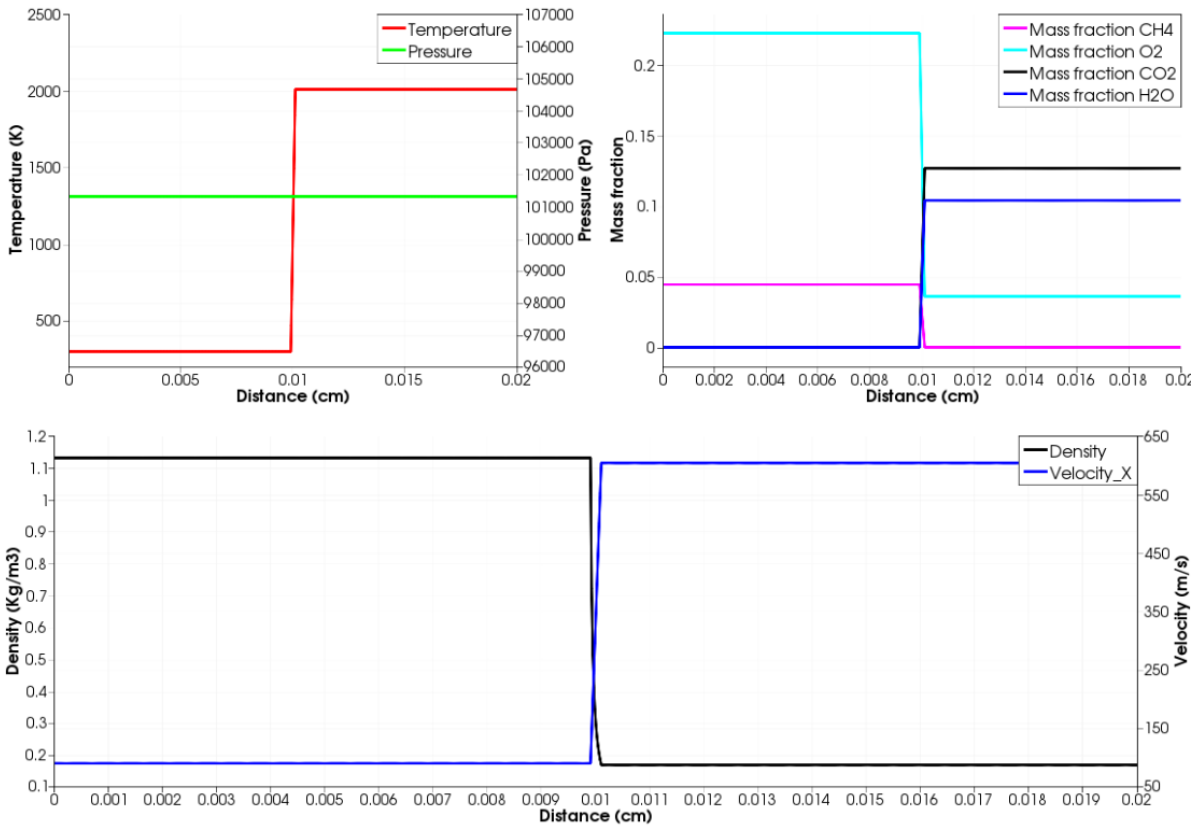

Fig. 7.12: Initial profiles of the premixed laminar flame use-case

Two types of boundary conditions are employed: *inlet* and *outlet*. The *inlet boundary conditions* correspond to the unburnt conditions, which can be imposed as Dirichlet-type. The *outlet* boundary conditions are of Neumann-type; all fluxes are set equal to zero.

This initial conditions are translated to Saiph language as follows:

```
// Initial values (with units)
    // Domain parameters
        // Discontinuity thinness
        def FLAME_THICKNESS = 10 * DXYZ
        def FLAME_START_X = (LX - FLAME_THICKNESS) / 2
        def FLAME_END_X = FLAME_START_X + FLAME_THICKNESS - DXYZ
            ...
    // Constants
        // Thermal conductivity
        def W_mK = Watts / Meters / Kelvins
        def LAMBDA_INIT = 0.0457 * W_mK
            ...
```

```
13      // Variables
14          // Temperature
15          def INIT_T_UNBURNT = 300 * Kelvins
16          def INIT_T_BURNT = 2011 * Kelvins
17          def TFLUX = 0 * KpS
18          // Pressure
19          def INIT_P = 1 * Atmospheres
20              ...
21
22  // Initialization
23      // Constants
24          val lambda = ConstTerm(W_mK)("Thermal conductivity", LAMBDA_INIT)
25              ...
26      // Variables
27          // Temperature
28          val T = Term(Temperature)("Temperature", mesh, { x =>
29            if (x < FLAME_START_X) INIT_T_UNBURNT
30            else if (x > FLAME_END_X) INIT_T_BURNT
31            else INIT_T_UNBURNT + (((INIT_T_BURNT - INIT_T_UNBURNT) /
                  FLAME_THICKNESS) * (x - FLAME_START_X))
32            })
33            T.setDirichlet(CFaceXMIN)(INIT_T_UNBURNT)
34            T.setNeumann(CFaceXMAX)(TFLUX)
35
36           // Pressure
37           val p = Term(Pressure)("Pressure", mesh, INIT_P)
38              ...
```

Physical Description

In the flame front, the temperature of the gas rises due to heat release and the effects of diffusion smoothes out the flame front. Eventually, the temperature reaches the point at which combustion takes place [(4)] releasing energy. From that point, the temperature rises up to the equilibrium conditions. The heat release and mixture composition (kinetics and chemistry) control the flame speed and the different thermal states. As we can see in figure 7.13 the transition from one state to the other takes place in a short spatial scale so-called the flame front.

[(4)] See the Autoignition use-case chapter 7.1.4

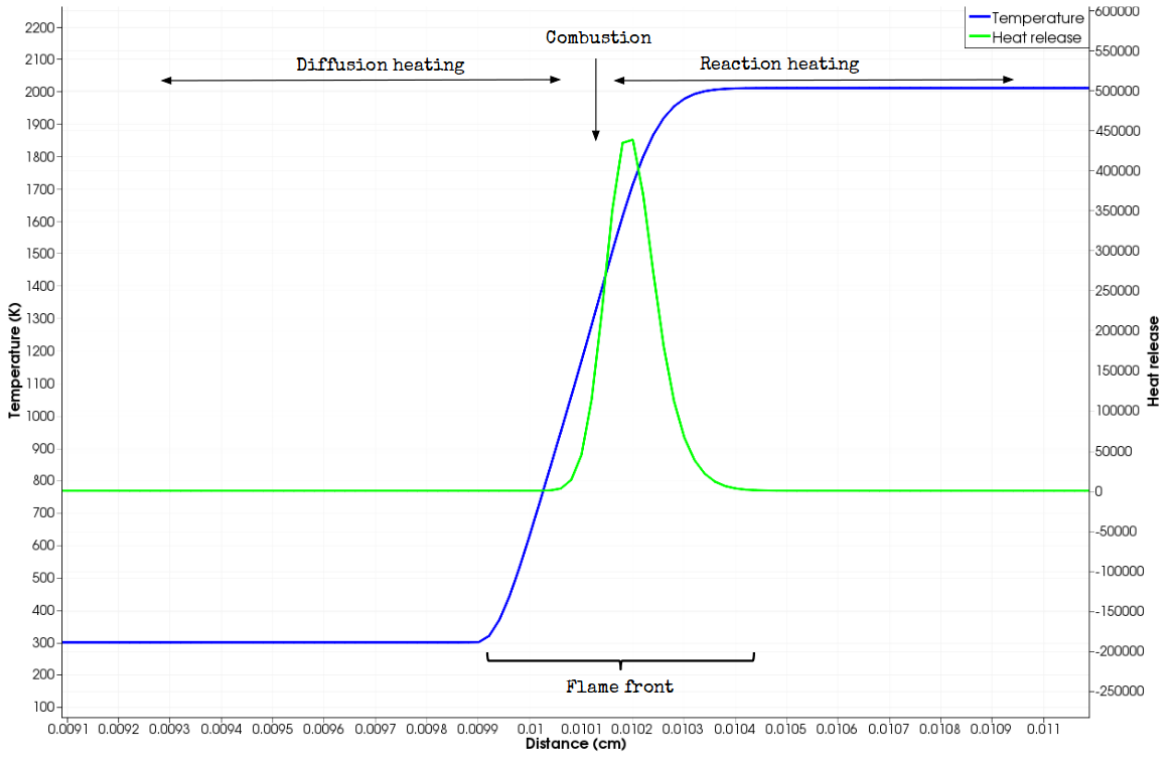

Fig. 7.13: Premixed laminar flame evolution. Zoom at the flame front

7.1.6 Results and validation

Simulating the temporal and spatial evolution of the system, its steady state and the conditions of the combustion products under investigation are given below, figure 7.14 and 7.15.

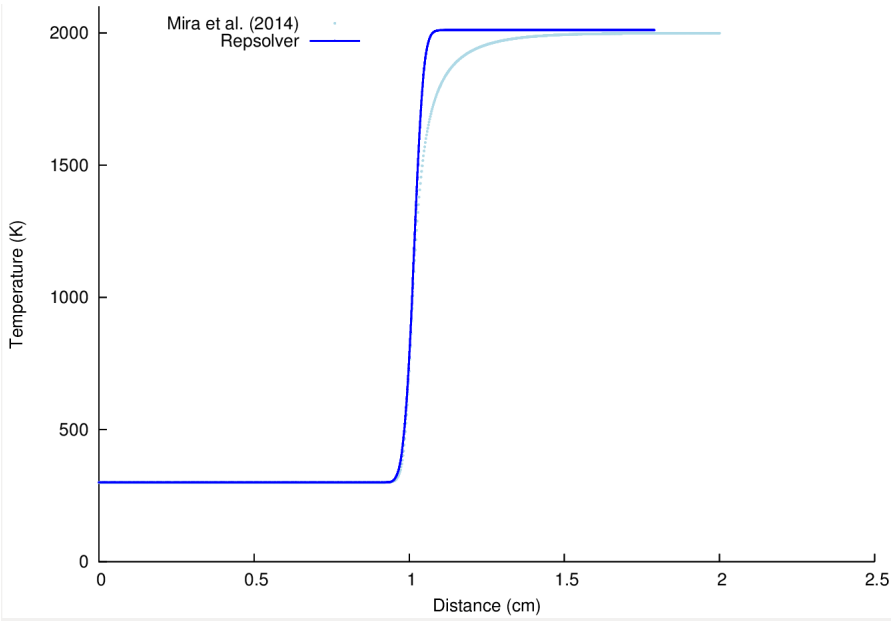

Fig. 7.14: Resulting temperature profiles of the premixed laminar flame use-case at $t > 0$ and its validation results

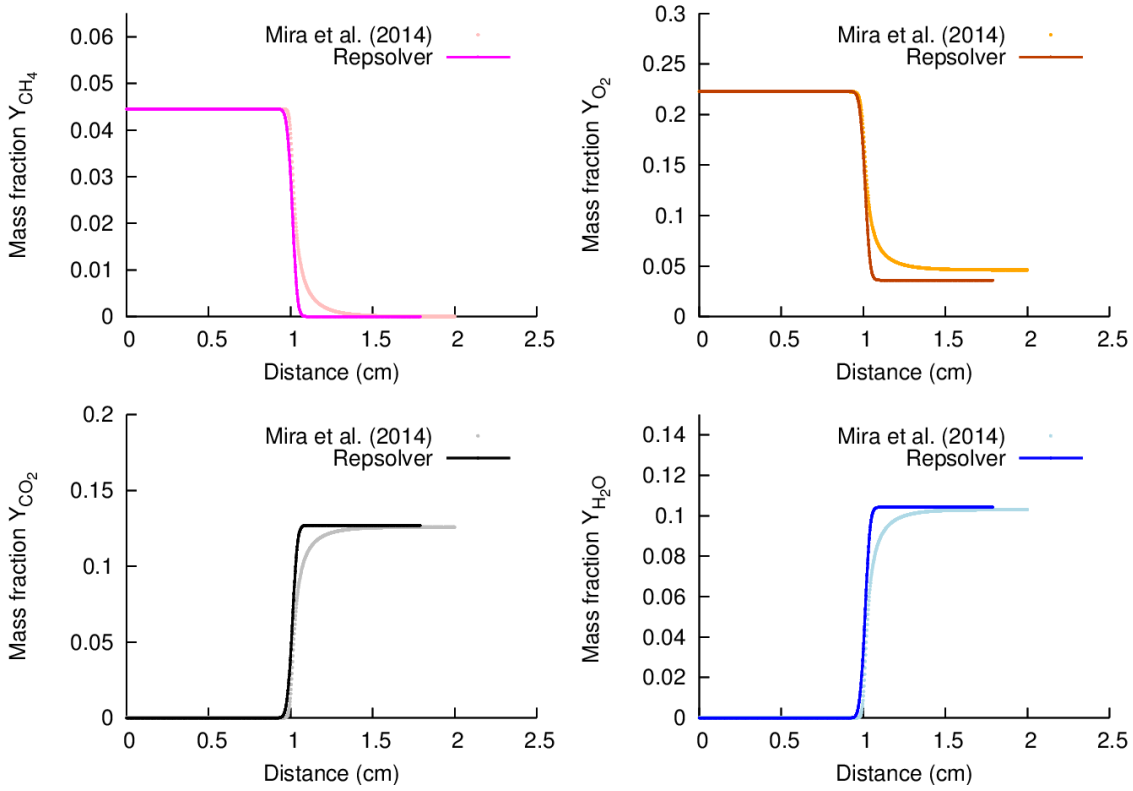

Fig. 7.15: Resulting mass fractions profiles of the premixed laminar flame use-case at $t\ 0$ and its validation results

For this test-case, the results from Mira et al. [28] are used for comparison. The results, in figure 7.14 and 7.15 indicate an acceptable level of correlation despite the use of constant transport properties. Including mixture and temperature dependency on transport properties and heat capacity would increase the level of correlation.

7.1.7 Nomenclature, units and constants

- Density: ρ
  Units:
  $$[\rho] = \frac{kg}{m^3}$$
- Velocity: $\mathbf{u} = (\mathbf{u_x}, \mathbf{u_y}, \mathbf{u_z})$
  Units:
  $$[u_i] = \frac{m}{s}$$
- Energy: $\mathbf{E}$
  Units:
  $$[E] = \frac{m^2}{s^2}$$
- Temperature: $\mathbf{T}$
  Units:
  $$[T] = K$$
- Pressure: $\mathbf{p}$
  Units:
  $$[p] = Pa = \frac{kg}{ms^2}$$
- Adiabatic gas index: γ
  Units: Adimensional magnitude
- Speed of sound: $\mathbf{c}$
  Units:
  $$[c] = \frac{m}{s}$$
- Mach number: $\mathbf{Ma} = \frac{u}{c}$
  Units: Adimensional magnitude
- Ideal gas constant: $\mathbf{R^o}$
  $$R^o = 8.31447 \quad \frac{J}{mol \cdot K}$$
  Units:
  $$[R^o] = \frac{J}{mol \cdot K}$$

- Thermal conductivity[5]: λ
  It is assumed constant at:
  $$\lambda = 0.0457 \frac{W}{K \cdot m}$$
  Units:
  $$[\lambda] = \frac{W}{K \cdot m}$$

- Dynamic viscosity: μ
  It is assumed constant at:
  $$\mu = 1.8 \cdot 10^{-5} \frac{Kg}{m \cdot s}$$
  Units:
  $$[\mu] = \frac{Kg}{m \cdot s}$$

- Mass fraction of the k^{th} specie k: $\mathbf{Y_k}$
  Ratio of one specie k with mass m_k to the mass of the total mixture. The sum of all the mass fractions is equal to one.
  Units: Adimensional magnitude

- Molar mass of the mixture: $\mathbf{W}$
  $$\frac{1}{W} = \sum_k \frac{Y_k}{W_k}$$
  Units:
  $$[W] = \frac{kg}{mol}$$

- Molar mass of each specie k: $\mathbf{W_k}$

Species	**Molar mass (kg/mol)**
CH_4	0.0160425
N_2	0.0280134
O_2	0.0319988
CO_2	0.0440095
H_2O	0.0180153

  Units:
  $$[W_k] = \frac{kg}{mol}$$

[5]The thermal conductivity and the specific heat are actually functions of the concentrations of the species. They are assumed constants in these use-cases

- Specific heat[6]: $\mathbf{c_p}$
  It is assumed constant at:
  $$c_p = 1012 \frac{J}{kg \cdot K}$$
  Units:
  $$[c_p] = \frac{J}{kg \cdot K}$$

- Specific enthalpy of each specie k: $\mathbf{h_k}$

Species	Specific enthalpy (J/kg)
CH_4	-4666.97834
N_2	0.0
O_2	0.0
CO_2	-8941.70577
H_2O	-13423.368

  Units:
  $$[h_k] = \frac{J}{kg}$$

- Coefficients of species k for the reaction: $\nu_\mathbf{k}$
  $\nu'_k \rightarrow$ Reactants species
  $\nu''_k \rightarrow$ Products species

  Units: Adimensional magnitude

- Reaction rate constant: $\mathbf{K_f}$
  $$K_f = A_f e^{\frac{-E_a}{RT}}$$
  Units:
  $$[K_f] = \frac{m^{3(n-1)}}{s \cdot mol^{(n-1)}}$$
  where n is the order of the reaction.

- Reaction rate progress: $\mathbf{Q}$
  Units:
  $$[Q] = [K_f] \cdot \left(\frac{mol}{m^3} \right)^{\sum_k \nu'_k}$$

- Pre-exponential factor: $\mathbf{A_f}$
  Units:
  $$[A_f] = [K_f]$$

[6]The specific heat is actually function of the concentrations of the species. It is assumed constants in these use-cases

- Activation energy: $\mathbf{E_a}$
  Units:
$$[E_a] = \frac{kJ}{mol}$$

7.2 Parallel execution analysis

Finite-difference stencil computations are very common in numerical modeling and they exhibit high degree of data parallelism and regular structure. In this section we analyse the scalability of the application *premixed laminar flame* setting the following parameters:

- Number of steps: 200 time-steps
- Temporal discretization factor: 5 ns
- Z dimension: 8 cm
- X, Y dimensions: 100 μm
- Spatial discretization factor: 20 μm
- Total number of spatial points: $1 \cdot 10^5$

7.2.1 Platform

We describe here the hardware used to collect real performance data on the Saiph application.
The platform used is Marenostrum III cluster, comprised (among others resources) of:

- 3108 IBM dx360 M4 compute nodes.
- Every IBM dx360 M4 node has:
  - 2x E5-2670 SandyBridge-EP 2.6HHz cahce 20MB 8-core
  - 32GB of RAM

The networks that interconnect the cluster using 296 switches are:

- Infiniband Network: High bandwidth network used by parallel applications communications (MPI).l'
- Gigabit Network: 10GbitEthernet network used by the GPFS Filesystem.

The results that follow are intrinsically subjected to this architecture.

7.2.2 Experimental Setup

In order to start the evaluation, we first of all, install the application in the Marenostrum cluster. Saiph applications allow sequential and parallel (MPI and/or OpenMP) executions. To create the desired executable, we use a Makefile for configuring the compilers, linkers, and their command line flags.
Finally, in order to submit jobs we have created scripts according to the requirements of the machine. Inside those scripts we can finally tune the number of nodes and/or threads we want to use for the execution.

At that point, we want to test if Saiph applications can be advantageously executed in a distributed environment. Because Saiph still lacks on efforts devoted to increase performance, we only present results from inter-node parallel executions. For a good use of the resources, we bind 1 MPI process per core resulting in 16 MPI processes per node. As said, OpenMP parallelism is not set.

7.2.3 Scalability Results

The complete sequential execution took around 28 min to finish. On the other hand, the most resource-consuming execution tested (160 MPI processes), took less than 12 s. The following figure shows the execution times and Speed up of several parallel executions taking the sequential one as reference.

# MPI processes	Execution time (s)
1	1677
16	108
32	57
48	36
64	28
80	22
96	18
112	16
128	15
144	13
160	11

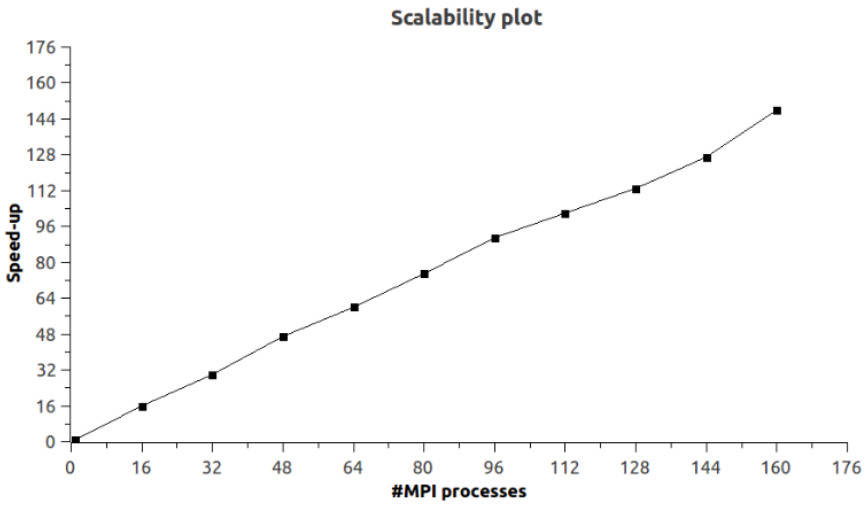

Fig. 7.16: Scalability plot for the Saiph application *premixed laminar flame*.

Linear speed up is reached up to 10 nodes, which corresponds to 160 MPI processes.

8 | Summary

The computational modelling of any physical problem is composed by three important steps: (1) the problem definition, (2) the mathematical model and (3) the computer simulation. Saiph has been extended in this project in order to ease the whole modelling process for those physical problems related to fluid mechanics and chemistry theory.

The problem definition, being the first natural step, is devoted to define the problem of interest in terms of a set of relevant quantities. Initial and boundary conditions are thus defined at that point. For this first step, Saiph provides an easy way to define and initialize variables and constants of the problem. Through the use of `Terms`, `ConstTerms` and *units* constructions users can unambiguously define their problems. Regarding initialization, Saiph offers the possibility to set variable values through spatial-dependent functions and a large set of operations over *units*, allowing mathematical relations between sets of dependent variables. In particular, Saiph provides intuitive constructions to well-define a spatial discontinuity over the physical domain ensuring a proper behaviour of linked variables at the interfaces. Although the domain expert is in charge of this first stage, Saiph put at his disposal a scientific high-level syntax to allow him to define the problem by directly translating his on-paper physical problem definition.

The second step of the modeling process is to represent the physical reality by a mathematical model: the governing equations of the problem. This task also belongs to the domain expert, but again, Saiph allows the direct and intuitive translation of the governing equations into Saiph' equations. Our tool, permit to express vector equations, partial differential equations (first and second order) and non-derivative equations commonly used to close PDE systems to define coupled problems. Users can correctly combine any of the aforementioned type of equations turning to the adequate use of Saiph' operators.

For this two first steps, the domain expert has been in charge, devolving upon him the responsibility of the correct problem definition and mathematical model statements. The rest of the modelling process is conceived within Saiph to be completely transparent to the users.

After the selection of an appropriate mathematical model, together with suitable boundary and initial conditions, we can proceed to the numerical solution through the computer simulation. Is in that point where all the specific numerical methods are applied. From methods for the resolution of PDEs and non-derivative equations, to specific meth-

ods directly related to the computation of specific terms from fluid mechanics and/or chemistry problems. Alternatives to Euler's method have been added for the non-derivative equations, methods for modelling convective terms, internal operators to be combined for the computation of diffusive terms, identification of such terms... everything being transparent to the user.

Nowadays a fourth step on the described computational modelling process, should be take into account, that is the parallel execution. Saiph also takes care internally of adapting the applications for being executed in parallel. MPI and OpenMP parallelization are thus implemented to allow Saiph' applications to be efficiently executed in a high-performance environment.

The described 4-steps process has been successfully followed by the different fluid mechanics and chemistry use-cases faced in this project.

Saiph appears to be a powerful tool that can be advantageously used by scientists without knowledge on numerical methods and supercomputing while internally providing the advantages of such expertise.

9 | Conclusions and future work

In this project, general user-level functionalities have been added to Saiph enlarging and easing the modelling constructions offered to the users. Internally, new features and optimizations have also been implemented in order to ensure the specific and correct computation of those new available user-constructions. Saiph provides more and easier features. Still, is important to remark that some of those extensions are subjected to the specific scientific domain faced. Those extensions still lack of genericity before being implicitly and systematically applied to any Saiph application. The development of Saiph should continue being driven by use-cases, whose requirements and impositions can light up the development progress. New use-cases can require the computation of more nesting derivatives levels or will need a convective gradient computation in a non-first-order differential equation or could contain a non-derivative equation enclosing a spatial differentiation as well as other situations leading, currently, to incorrect results. Therefore, some of the specific optimizations performed, are still not as general as desired but have resolved the problems and necessities encountered in this fluid mechanics and chemistry support extensions. In that sense, we have proposed solutions displaying the potential of Saiph ant its layer structure and they can be seen as first approaches to face more general requirements that can be encountered in the future.
As a present and future recommendation, all the new features and functionalities should be integrated without breaking backward compatibility of the tool and trying to take into account all the possible situations that can be confronted in the future. Saiph' extensions should be general enough to be applied for any computational modelling of any physical problem expressed through a partial differential equation system. Facing more use-cases from different scientific domains is a must for the future development of Saiph.

From the numerical point of view, lot of work can be done; The addition of an internal check of the numerical stability, the dynamic computation of the more efficient time discretization factor at each computational step, the support to either 1D, 2D or 3D simulation problems are just examples of features that can be incorporated to Saiph to ease and enlarge its correctness, usability and efficiency.
Regarding high performance complexities, we have demonstrated that a real and expensive Saiph application can be executed in a distributed environment without wasting the available resources. Nevertheless, in the current scenario, after validating the functioning of the tool, Saiph performance should be faced intensively.

A | Scala

This appendix introduces the most basic concepts, constructions and features of Scala that are being used in Saiph applications.

Variables

The most relevant basic types for Saiph users are: `Int`, `Float`, `Boolean` and `String`. Scala has type inference, therefore, it automatically deduces the type of the initializing expression when it is known at compile time. Yet, this is not always the case, as it can be seen in line 4, if a value is initialized with an empty list, the type of this list should be specified either in the value or in the constructor. Values in Scala are declared as:

```
1 val azero = 0
2 val pi = 3.1416
3 val l1 = List(1, 2, 3)
4 val l2 : List[Int] = List() // Type specified as the value type
5 val l3 = List[Int]() // Type inferred from the constructor type
```

Functions

Functions are defined using the `def` keyword. Specifying the formal parameter types is mandatory and, for recursive functions, the return type must also be specified:

```
1 def add(x: Int, y: Int) = x + y
2 def fibonacci(n: Int) : Int = {
3     if (n < 0) -1
4     else if (n == 0 || n == 1) n
5     else fibonacci(n-1) + fibonacci(n-2)
6 }
```

Objects and classes

Scala provides classes, objects and traits. These language constructs are used to implement abstract data types. Objects are created through class instantiation so, through keyword `class`. Scala provides singleton objects which act as group of static functions (they operate on no particular implicit instance) declared as `object`. If a class and an object share the same name they are called companion classes/objects. Companions objects and classes have access to each other's private members. Scala allows declaring values, methods and types as abstract. Abstract type members are used to build generic components.
Abstract classes cannot be instantiated and its abstract members need to be defined in their subclasses. This is achieved through the `extends` keyword. Subclasses inherit non-private members of its superclasses. In Scala, multiple inheritance is not allowed, meaning classes cannot inherit from multiple superclasses.

Traits and mixin components

Traits are declared in a similar way as regular classes with the exception that the programmer uses keyword `trait` instead of keyword `class`. A *trait* is actually, a special form of an abstract class which does not have any value parameters for its constructor.

Consider the following abstraction for iterators.

```
1 trait AbsIterator[T] {
2     def hasNext: boolean
3     def next: T
4 }
```

Traits can be used in all context where other abstract classes appear but only traits can be used as mixins. Mixin class composition is a form of multiple inheritance. By mixing in several traits into classes Scala provides stackable composition.

The following code illustrates how a trait extends `AbsIterator` with a new method (`foreach` method):

```
1 trait RichIterator[T] extends AbsIterator[T] {
2     def foreach(f: T => unit): unit =
3         while (hasNext) f(next)
4 }
```

Scala classes can mix in several traits with the keywords `extends` and `with`. The order in which traits are mixed in matters.

B | Lightweight Modular Staging

Building a DSL

The definition of a DSL in LMS consists of three main parts:

- DSL interface

  The interface of the DSL must be defined with types and operations the programmer will use in the application. We build *Vector DSL* as an example. This DSL provides a generic type `VT[_]` and vector addition operation. The interface of *Vector DSL* is as follows:

```
1 trait VectorOps extends Base {
2       class VT[O]
3       def infix_+[O:Manifest](x: Rep[VT[O]], y: Rep[VT[O]]) =
4             vector_plus(x,y)
5       def vector_plus[O:Manifest](x: Rep[VT[O]], y: Rep[VT[O]]):
6             Rep[VT[O]]
7 }
```

  The basic type of the language, `VT[_]`, is represented here by an empty Scala class (line 2). In line 3, operation for adding two vectors is defined. The body of this method consists on a call to an abstract method `vector_plus` (line 5). This abstract method will be implemented in a later phase of the process.

  The meaning of having type `Rep[VT]` is that `x` and `y` *represent* a computation that will yield a `VT` in the next stage. For example, type `Rep[Int]` indicates that the staged computation will result in type `Int` in the next stage[5]. `VectorOps` is a trait mixed in with `Base`. `Base` is a part of LMS library.

- Internal representation

  The following code shows the implementation of the *Vector DSL*:

```
1 trait VectorOpsExp extends VectorOps with Expressions {
2       case class VtPlus[O:Manifest](x: Exp[VT[O]], y: Exp[VT[O]])
3             extends Def[VT[O]]
4       def vector_plus[O:Manifest](x: Exp[VT[O]], y: Exp[VT[O]]) =
5             (x,y) match {
6                   case _ => VtPlus(x, y)
7       }
8 }
```

The implementation is defined in trait `VectorOpsExp` (line 1). It implements the interface of our DSL, `VectorOps`. The trait `Expressions` provides an infrastructure for implementing an intermediate representation (IR) of our DSL as expression tree. The tree contains IR nodes of our DSL.

The IR nodes are implemented as case classes (line 2) which take arguments of type `Exp[_]`. This LMS type constructor represents constants and symbols. Each case class extends a parametrized type constructor `Def[_]` which represents definitions. Symbols are bound to definitions. Each composite construct (such as case class `VtPlus`) refers to its parameters through symbols.

Line 4 shows the body of method `vector_plus`. This method was declared abstract in the interface. This is the way the implementation is bound to its interface. However, this binding is loose, meaning interface and implementation of the DSL are implemented as separate traits.

As we can see in the previous code, definitions are implemented as Scala case class, so they can be pattern-matched. This in turn allows the implementation of optimizations as we are going to see later. Here, since there is only the default case, a regular addition IR node is emitted (`VtPlus`) each time any argument is passed to the + operator.

- Code generation

  This last stage, implements code generator for the DSL to target language. The following code shows the code generetor to C++ of the *Vector DSL*:

```
1 trait VectorCppGen extends CGen {
2       val IR: VectorOpsExp
3       import IR._
4       override def emitNode(sym: Sym[Any], rhs: Def[Any]): Unit =
5             rhs match {
6             case VtPlus(x, y) => emitValDef(sym,
7                   quote(x) + " + " + quote(y))
8             case _ => super.emitNode(sym, rhs)
```

```
9         }
10 }
```

The code generator is implemented as a trait that extends the LMS trait `CGen` (line 1). Trait `CGen` extends the core code generation traits with code generators for regular C code, so we do not have to redefine regular assignment or looping, for example. The `quote` method obtains the symbol corresponding to a given particular definition.

In lines 2 and 3, there is the information about the intermediate representation and the implementation of the DSL interface. The `import` statement in line 3 injects the types of IR nodes into the scope of the code generator so it can refer to them through pattern matching (body of method `emitNode` in lines 4-9).

While LMS is traversing an expression tree of our DSL, each IR node is pattern-matched (lines 4-8). If the definition is matched (left-hand side of a `case` statement), C code is generated for a given IR node (right-hand side of the statement). If no match is found, the symbol and definition of the IR node are passed forwarded to the next mixed-in trait (or superclass) on the chain.

Using the DSL

Once we have all the parts of a DSL (representation, implementation and code generation), our DSL is ready to translate applications into the target language. Let's consider the following application for the *Vector DSL*

```
1 object VectorDSLApp {
2       trait SimpleApp {
3             val v = Vector(1, 2, 3)
4             val w = Vector(2, 5, 2)
5             val sum = v + w
6             show(sum)
7       }
8 }
```

For simplicity, we will not show all the details of the operations not described in this section (like the `Vector(...)` constructor or the `show` operation. For a complete documentation about LMS, see the referenced material and its official website [29].

The code emitted after running the generator of the application is shown as follows.

```
#include <iostream>
#include "vectorDataStruct.h"
int main() {
      Vector x0(1, 2, 3);
      Vector x1(2, 5, 2);
      Vector x2 = x0+x1;
      for (size_t i = 0; i < x2.len(); ++i) std:cout << x2[i] << " ";
      std::cout << std::endl;
}
```

C | Premixed laminar flame; Saiph complete code

```
package saiph.app

import saiph._
import java.io.PrintWriter

/** One-dimensional premixed flame simulation.
  * Coupled system of species concentration,
  * energy (temperature) and density.
  * Low Mach regime: hydrodynamics + reactions at low gas spedd: delfagration.
  * ==Overview==
  * Illustrates the use of a good amount of Saiph operators, complex custom
  * unit definitions and non-differential equations
  */
 object Flame1D {
   trait Flame1DProg { this: SaiphOps =>

      def LX = 2 * Centimeters
      def LYZ = 0.1 * Millimeters
      def DXYZ = 20 * Micrometers
      def FLAME_THIKNESS = 10 * DXYZ
      def FLAME_START_X = (LX - FLAME_THIKNESS) / 2
      def FLAME_END_X = FLAME_START_X + FLAME_THIKNESS - DXYZ

      // Simulation parameters
      def DELTA_TIME = 5 * Nanoseconds
      val N_STEPS = 15000

      // Gas names
      val GAS_NAMES = List[Rep[String]]("CH4", "N2", "O2", "C2O", "H2O")
```

```
30
31      // Constants
32      // Activation energy
33      def JpMol = Joules / Moles
34      def EA_VALUE = -167360 * JpMol
35
36      // Pre-exponential factor
37      def M6pM2s = MUnit(DLength -> 6, DAmount -> -2, DTime -> -1)
38      def AF_VALUE = 1.1e10 * M6pM2s
39
40      // Thermal conductivity
41      def W_mK = Watts / Meters / Kelvins
42      def LAMBDA_INIT = 0.0457 * W_mK
43
44      // Air viscosity constant
45      def Kg_ms = Kilograms / Meters / Seconds
46      def AIR_VISCOSITY = 1.8e-5 * Kg_ms
47
48      // Specific heat capacity (at cte pressure)
49      def JpKKg = Joules / Kelvins / Kilograms
50      def CP_INIT = 1200 * JpKKg
51
52      // Adiabatic index (cp/cv)
53      def ADIABATIC_INDEX = 1.4 * Unitless
54
55      // Ideal gas constant
56      def JpMK = Joules / Moles / Kelvins
57      def R_VALUE = 8.314462175 * JpMK
58
59      // Reaction coefficients
60      def nu_v = Vector(-1 * Unitless, 0 * Unitless, -2 * Unitless, 1 *
            Unitless, 2 * Unitless)
61
62      // Specific Enthalpy
63      def JpKg = Joules / Kilograms
64      def h_v = Vector(-4666.97834 * JpKg, 0 * JpKg, 0 * JpKg,
            -8941.70577 * JpKg, -13423.368 * JpKg)
65
66      // Molar mass of each gas
67      def GramsPerMoles = Grams / Moles
68      def wCH4 = 16.0425 * GramsPerMoles
69      def wN2 = 28.0134 * GramsPerMoles
70      def wO2 = 31.9988 * GramsPerMoles
```

```
71      def wCO2 = 44.0095 * GramsPerMoles
72      def wH2O = 18.0153 * GramsPerMoles
73      def vW = Vector(wCH4, wN2, wO2, wCO2, wH2O)
74
75
76      // Variables
77      // Mass fractions
78      def UNBURNT = Vector(0.0445 * Unitless, 0.7324 * Unitless, 0.2231
            * Unitless, 0 * Unitless, 0 * Unitless)
79      def BURNT = Vector(0 * Unitless, 0.7329 * Unitless, 0.0360 *
            Unitless, 0.1270 * Unitless, 0.1041 * Unitless)
80      //flux boundary conditions
81      def pM = Unitless / Meters
82      def YFLUX = Vector(0 * pM, 0 * pM, 0 * pM, 0 * pM, 0 * pM)
83
84      // Temperatures
85      def INIT_T_UNBURNT = 300 * Kelvins
86      def INIT_T_BURNT = 2011 * Kelvins
87      //flux boundary conditions
88      def KpM = Kelvins / Meters
89      def TFLUX = 0 * KpM
90
91      // Flow velocity
92      def MpSec = MetersPerSecond;
93      def V_UNBURNT = Vector(90 * MpSec, 0 * MpSec, 0 * MpSec)
94       //flux boundary conditions
95      def pS = Unitless / Seconds
96      def UFLUX = Vector( 0 * pS, 0 * pS, 0 * pS)
97
98      // Pressure
99      def INIT_P = 1 * Atmospheres
100      //flux boundary conditions
101     def PpM = Pascals / Meters
102     def PFLUX = 0 * PpM
103
104     // Density
105     def Kg_m3 = Kilograms / Meters3
106     def mol_Kg = Moles / Kilograms
107     def Winv_UNBURNT = (UNBURNT(0) / vW(0)) + (UNBURNT(1) / vW(1)) + (
            UNBURNT(2) / vW(2)) + (UNBURNT(3) / vW(3)) + (UNBURNT(4) / vW
            (4))
108     def DENSITY_UNBURNT = ((INIT_P / (R_VALUE * INIT_T_UNBURNT)) / (
            Winv_UNBURNT))
```

```
109
110     def Winv_BURNT = (BURNT(0) / vW(0)) + (BURNT(1) / vW(1)) + (BURNT
            (2) / vW(2)) + (BURNT(3) / vW(3)) + (BURNT(4) / vW(4))
111     def DENSITY_BURNT = ((INIT_P / (R_VALUE * INIT_T_BURNT)) / (
            Winv_BURNT))
112
113     // Rho*u, rho*T and omega units
114     def Kg_m2s = Kilograms / Meters2 / Seconds
115     def KgK_m3 = Kilograms * Kelvins / Meters3
116     def Kg_m3s = Kilograms / Meters3 / Seconds
117     def J_m3s = Joules / Meters3 / Seconds
118
119     def Flame1D = {
120       val mesh = CartesianMesh(LX, LYZ, LYZ)
121       mesh.discretize(DXYZ, DXYZ, DXYZ);
122
123       // Combustion constants
124       val MolarMassVector = ConstTerm(GramsPerMoles)("Molar mass", vW,
              GAS_NAMES)
125       val R = ConstTerm(JpMK)("Ideal gas constant", R_VALUE)
126       val Ea = ConstTerm(JpMol)("Activation Energy", EA_VALUE)
127       val Af = ConstTerm(M6pM2s)("Pre-exponential factor", AF_VALUE)
128       val nu = ConstTerm(Unitless)("Reaction coefficients", nu_v,
              GAS_NAMES)
129       val h = ConstTerm(JpKg)("Enthalpy", h_v, GAS_NAMES)
130       val lambda = ConstTerm(W_mK)("Thermal conductivity", LAMBDA_INIT)
131       val cp = ConstTerm(JpKKg)("Specific heat (cte pressure)", CP_INIT
              )
132       val cv = ConstTerm(JpKKg)("Specific heat (cte volume)", CP_INIT /
               ADIABATIC_INDEX)
133       val mu = ConstTerm(Kg_ms)("Viscosity", AIR_VISCOSITY)
134       val mu_cte = ConstTerm(Unitless)("Mu factor (4/3)", 4/3 *
              Unitless)
135
136       def linear_window_vector(x: Rep[MUnit], left_value: Rep[Vector[
              MUnit]], right_value: Rep[Vector[MUnit]]) : Rep[Vector[MUnit
              ]]= {
137         left_value + (((right_value - left_value)/FLAME_THIKNESS)* (x
                -FLAME_START_X))
138       }
139       def linear_window_scalar(x: Rep[MUnit], left_value: Rep[MUnit],
              right_value: Rep[MUnit]) : Rep[MUnit] = {
```

```
140             left_value + (((right_value - left_value)/FLAME_THIKNESS)* (x
                    -FLAME_START_X))
141           }
142
143           // Mass fraction
144           val Y = Term(Unitless)("Yk, mass fractions", mesh, { x =>
145             if (x < FLAME_START_X) UNBURNT
146             else if (x > FLAME_END_X) BURNT
147             else linear_window_vector(x, UNBURNT, BURNT)
148           }, GAS_NAMES)
149         Y.setDirichlet(CFaceXMIN)(UNBURNT)
150         Y.setNeumann(CFaceXMAX)(YFLUX)
151
152           // Temperature
153           val T = Term(Temperature)("Temperature", mesh, { x =>
154             if (x < FLAME_START_X) INIT_T_UNBURNT
155             else if (x > FLAME_END_X) INIT_T_BURNT
156             else linear_window_scalar(x, INIT_T_UNBURNT, INIT_T_BURNT)
157         })
158         T.setDirichlet(CFaceXMIN)(INIT_T_UNBURNT)
159         T.setNeumann(CFaceXMAX)(TFLUX)
160
161           // Pressure
162           val p = Term(Pressure)("Pressure", mesh, INIT_P)
163         p.setDirichlet(CFaceXMIN)(INIT_P)
164         p.setNeumann(CFaceXMAX)(PFLUX)
165
166           // Density
167           val rho = Term(Kg_m3)("Density", mesh, { x =>
168             if (x < FLAME_START_X) DENSITY_UNBURNT
169             else if (x > FLAME_END_X) DENSITY_BURNT
170             else {
171               val vec = linear_window_vector(x, UNBURNT, BURNT)/vW
172               val sum_win = vec(0) + vec(1) + vec(2) + vec(3) + vec(4)
173               INIT_P / sum_win / (R_VALUE * linear_window_scalar(x,
                      INIT_T_UNBURNT, INIT_T_BURNT))
174             }
175         })
176
177           // Velocity
178           val u = Term(Speed)("Velocity", mesh, { x =>
179             if (x < FLAME_START_X) V_UNBURNT
```

```
180           else if (x > FLAME_END_X) V_UNBURNT * (DENSITY_UNBURNT/
                 DENSITY_BURNT)
181           else {
182             val vec = linear_window_vector(x, UNBURNT, BURNT)/vW
183             val sum_win = vec(0) + vec(1) + vec(2) + vec(3) + vec(4)
184             V_UNBURNT * (DENSITY_UNBURNT / (INIT_P / sum_win / (R_VALUE *
                    linear_window_scalar(x, INIT_T_UNBURNT, INIT_T_BURNT))))
185           }
186         }, List[Rep[String]]("X", "Y", "Z"))
187       u.setDirichlet(CFaceXMIN)(V_UNBURNT)
188       u.setNeumann(CFaceXMAX)(UFLUX)
189
190         // Mass flow (rho*u = cte)
191         val rho_u = Term(Kg_m2s)("Rho * u", mesh, { x => DENSITY_UNBURNT
               * V_UNBURNT }, List[Rep[String]]("X", "Y", "Z"))
192
193         // Rho * Y
194         val rho_Y = Term(Kg_m3)("Rho * Y", mesh, { x =>
195           if (x < FLAME_START_X) DENSITY_UNBURNT * UNBURNT
196           else if (x > FLAME_END_X)DENSITY_BURNT * BURNT
197           else {
198             val vec = linear_window_vector(x, UNBURNT, BURNT)/vW
199             val sum_win = vec(0) + vec(1) + vec(2) + vec(3) + vec(4)
200             (INIT_P / sum_win / (R_VALUE * linear_window_scalar(x,
                    INIT_T_UNBURNT, INIT_T_BURNT))) * linear_window_vector(x,
                    UNBURNT, BURNT)
201           }
202         }, GAS_NAMES)
203
204         // Rho * T
205         val rho_T = Term(KgK_m3)("Rho * T", mesh, { x =>
206           if (x < FLAME_START_X) DENSITY_UNBURNT * INIT_T_UNBURNT
207           else if (x > FLAME_END_X) DENSITY_BURNT * INIT_T_BURNT
208           else {
209             val vec = linear_window_vector(x, UNBURNT, BURNT)/vW
210             val sum_win = vec(0) + vec(1) + vec(2) + vec(3) + vec(4)
211             (INIT_P / sum_win / (R_VALUE * linear_window_scalar(x,
                    INIT_T_UNBURNT, INIT_T_BURNT))) * linear_window_scalar(x,
                    INIT_T_UNBURNT, INIT_T_BURNT)
212           }
213       })
214
215
```

```
216         // Omega. Species source term
217         val omega_species_k = Term(Kg_m3s)("Species source terms", mesh,
                Vector(0 * Kg_m3s, 0 * Kg_m3s, 0 * Kg_m3s, 0 * Kg_m3s, 0 *
               Kg_m3s), GAS_NAMES)
218
219         // Omega. Heat release
220         val omega_heat = Term(J_m3s)("Heat release", mesh, 0 * J_m3s)
221
222
223       // Premixed laminar flame equations
224
225         // Species source term
226         // Rate constant
227         val Kf = Af * exp(Ea / (R*T))
228         // Species concentrations
229         val XCH4 = rho_Y(0) / MolarMassVector(0)
230         val XO2 = rho_Y(2) / MolarMassVector(2)
231         // Reaction rate progress
232         val Q = Kf * XCH4 * XO2 * XO2
233         val omegak = Equation(omega_species_k, (MolarMassVector.comp * nu
               .comp * Q))
234
235       // Heat release
236         val omegaT = Equation(omega_heat, - (sum(cwiseprod(h,
               omega_species_k))))
237
238         // Equation 1: Continuity equation
239         val density = Equation(dt(rho), -div(rho_u))
240
241         // Equation 2: Momentum equation
242       val momentum = Equation(dt(rho_u), -((rho_u * grad(u.comp)) + (u.
            comp * div(rho_u)) + grad(p).comp) + (mu_cte * mu * der2(DirX)(
            u.comp)))
243
244       // Equation 3: Species equations
245         // Convective term
246         val Yconv = - ((rho_u * grad(Y.comp)) + (Y.comp * div(rho_u)))
247         // Diffusion term
248         val Ydiff = div((lambda / cp) * grad(Y.comp))
249       val rho_species = Equation(dt(rho_Y), Yconv + Ydiff +
            omega_species_k.comp)
250
251       // Equation 4: Temperature equation
```

```
252         // Convective term
253         val Tconv = - (cp * ((rho_u * grad(T)) + (T * div(rho_u))))
254         // Diffusion term
255         val Tdiff = div(lambda * grad(T))
256         val rho_temperature = Equation(dt(rho_T), (Tconv + Tdiff +
                omega_heat) / cp)
257
258       // Auxiliar equations:
259       // Species
260       val species = Equation(Y, rho_Y.compWithOffset(0) / rho.toff(0))
261
262       // Temperature
263       val temperature = Equation(T, rho_T.toff(0) / rho.toff(0))
264
265       // Velocity
266       val velocity = Equation(u, rho_u.compWithOffset(0) / rho.toff(0))
267
268       // Equation 5: State equation (pressure)
269       val pressure = Equation(p, rho_T.toff(0) * R * sum(Y.toff(0) /
              MolarMassVector))
270
271         val flame1D = Problem(DELTA_TIME, N_STEPS, mesh)(omegak, omegaT,
              density, momentum, rho_species, rho_temperature, species,
              temperature, velocity, pressure)
272
273         EulerSolver(flame1D)("FLAME", OutputFormat.VTI, SamplingMethod.
              Periodic, 1)
274     }
275   }
276   def main(args: Array[String]): Unit = {
277     new Flame1DProg with SaiphOpsExp { self =>
278       val codegen = new SaiphCGen { val IR: self.type = self }
279       codegen.emitSource(Flame1D _, new PrintWriter(Console.out))
280     }
281   }
282 }
```

Bibliography

[1] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. Liszt: A domain specific language for building portable mesh-based pde solvers. 2011. 5

[2] Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. *FFC: the FEniCS Form Compiler*, chapter 11. Springer, 2012. 6

[3] Todd Dupont, Johan Hoffman, Claus Johnson, Robert C Kirby, Mats G Larson, Anders Logg, and L Ridgway Scott. *The FEniCS project*. Chalmers Finite Element Centre, Chalmers University of Technology, 2003. 6

[4] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004. 7

[5] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314. URL http://doi.acm.org/10.1145/1868294.1868314. 7, 97

[6] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 117–120, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1118-2. doi: 10.1145/2103746.2103769. URL http://doi.acm.org/10.1145/2103746.2103769. 7, 10

[7] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004. 9

[8] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010. 10

[9] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, September 2009. ISSN 0956-7968. doi: 10.1017/S0956796809007205. URL http://dx.doi.org/10.1017/S0956796809007205. 10

[10] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999. 30

[11] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008. 31

[12] TJ Poinsot and DP Veynante. Combustion. *Encyclopedia of Computational Mechanics*. 33, 79

[13] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++ 11 and C++ 14*. " O'Reilly Media, Inc.", 2014. 45

[14] Amy Henderson, Jim Ahrens, Charles Law, et al. *The ParaView Guide*. Kitware Clifton Park, NY, 2004. 45

[15] Thomas Williams and Colin Kelley. many others, gnuplot 4.4: an interactive plotting program, 2010. 45

[16] Extrae. User guide manual -for version 2.5.1. URL https://www.bsc.es/computer-sciences/performance-tools/trace-generation/extrae/extrae-user-guide. 45

[17] Paraver: a flexible performance analysis tool . URL https://www.bsc.es/computer-sciences/performance-tools/documentation. 45

[18] Hugo Winter and John Thuburn. Numerical advection schemes in two dimensions. 2011. 54

[19] Dale R Durran. *Numerical methods for wave equations in geophysical fluid dynamics*, volume 32. Springer Science & Business Media, 2013. 54

[20] G. Sod. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. *Journal of Computational Physics*, 27(1):1–31, 1978. ISSN 00219991. URL http://dx.doi.org/10.1016/0021-9991(78)90023-2. 64

[21] Ali Ben Moussa and Hatem Ksibi. Numerical simulation of a one-dimensional shock tube problem at supercritical fluid conditions. *International Journal of Fluid Mechanics Research*, 35(1):38–50, 2008. ISSN 1064-2277. 66

[22] Gas dynamics: The riemann problem and discontinuous solutions: Application to the shock tube problem. In Ionut Danaila, Pascal Joly, SidiMahmoud Kaber, and Marie Postel, editors, *An Introduction to Scientific Computing*, pages 213–233. Springer New York, 2007. ISBN 978-0-387-30889-0. doi: 10.1007/978-0-387-49159-2_10. 69

[23] M. Moragues. Variational multiscale stabilization and local preconditioning for compressible flow. phd thesis submitted to universitat politécnica de catalunya. 147, 2015. 69

[24] T. Mantel, F.N. Egolfopoulos, and C.T. Bowman. A new methodology to determine kinetic parameters for one- and two-step chemical models. Center for Turbulence Research, Proceedings of the Summer Program, 1996. 73

[25] Eduardo Fernández-Tarrazo, Antonio L. Sánchez, Amable Liñán, and Forman A. Williams. A simple one-step chemistry model for partially premixed hydrocarbon combustion. *Combustion and Flame*, 147(1-2):32–38, 2006. 73

[26] M.M. Holton and College Park. Mechanical Engineering University of Maryland. *Autoignition Delay Time Measurements for Natural Gas Fuel Components and Their Mixtures*. University of Maryland, College Park, 2008. ISBN 9780549957867. 77, 78

[27] Dipl-Ing Dieter Kaufmann and Ing Paul Roth. The method of fractional steps applied to laminar flat flame calculations. *Forschung im Ingenieurwesen A*, 53(4): 113–117, 1987. 79

[28] D. Mira, M. Zavala, M. Avila, Herbert Owen, J.C. Cajas, G. Houzeaux, and M. Vázquez. Heat transfer effects on a fully premixed methane impinging flame, 17-10 Sept. 2014. 87

[29] LMS Official website. URL http://scala-lms.github.io/. 99