PELICANS: An Implementation Tool for Solvers of Partial Differential Equations

March 15, 2010

Abstract

In a wide range of cases, the task of building a numerical simulation can be split in three steps: in the mathematical modelling stage, the physical problem under consideration is stated as a system of partial derivative equations, then a numerical method is designed for their solution and, finally, a computational code is developed.

The aim of PELICANS is to reduce the complexity and the cost of this last step. To this purpose, it provides a library of evolutive software components for the implementation of various numerical methods dedicated to the solution of partial derivative equations. Most commonly used spatial discretization methods, namely finite differences, finite volumes and finite elements on both structured or unstructured grids, are addressed. In addition, basic functionalities are made available for the implementation of domain decomposition, moving grids and characteristic methods. For preprocessing, post-processing and linear algebra, in particular, the internal capabilities of the library are completed by the coupling to external softwares.

In the course of the construction of the PELICANS platform, programming techniques and associated tools have been elaborated to facilitate the application in standard `C++` of Component-Based Development principles, such as the Design by Contract, command-query separation, naming and self-documentation issues. This consistent and validated approach also represents an output of the PELICANS project.

Finally, a practical example of numerical simulation is extensively documented. The role played by PELICANS in the global process of development is shown, together with the implications of its use on the final product, namely the code source.

Keywords

PELICANS, Partial Differential Equations, Meshing, Finite Elements, Finite Volumes,
Finite Differences, Object Programming, Software Component, Framework.

INSTITUT DE RADIOPROTECTION ET DE SÛRETÉ NUCLÉAIRE
DIRECTION DE PRÉVENTION DES ACCIDENTS MAJEURS
SERVICE D'ETUDES ET DE MODELISATION DES INCENDIES, DU CONFINEMENT ET DU CORIUM

Centre d'Études de Cadarache - Bâtiment 702 - BP3-13115 SAINT PAUL LEZ DURANCE CEDEX
Téléphone : (33) 4.42.19.96.48 - Fax : (33) 4.42.19.91.66 - E-mail : bruno.piar@irsn.fr

Résumé

La démarche de construction d'une large gamme de simulations numériques est séparée en trois étapes : une phase de modélisation mathématique conduisant à une formulation du problème physique par un système d'équations aux dérivées partielles, puis une phase de modélisation numérique établissant une méthode d'approximation de la solution de ce système, et enfin une phase d'implémentation d'un logiciel calculant cette approximation.

L'objectif de la plate-forme PELICANS est de réduire la complexité et le coût de cette dernière étape. A cet effet, elle met à disposition une bibliothèque de composants logiciels adaptables pour la mise en œuvre d'un ensemble de méthodes numériques dédiées à la résolution des équations aux dérivées partielles. Parmi celles-ci, les techniques de discrétisation spatiale les plus communément utilisées sont représentées : différences, volumes et éléments finis, sur des maillages structurés ou non structurés. Sont offertes également les fonctionnalités nécessaires à l'emploi de méthodes de décomposition de domaine, de maillages mobiles et des caractéristiques. Pour le pre-processing, le post-processing et l'algèbre linéaire, notamment, les potentialités de la plate-forme elle-même sont complétées par couplage avec des logiciels externes.

Dans le cadre de la construction de la plate-forme PELICANS, des techniques de programmation et les outils associés ont été élaborés pour la mise en œuvre en langage `C++` standard des principes de développement à base de composants logiciels, tels que la Programmation par Contrats, la séparation des commandes et des requêtes, l'héritage, le nommage ou l'autodocumentation. Cet ensemble forme un tout cohérent et validé qui constitue également l'un des produits finis du projet PELICANS.

Un exemple pratique de simulation numérique, intégralement documenté, illustre enfin la place de PELICANS au sein de la démarche globale et les implications de son utilisation dans le monde réel du programmeur.

Mots Clés

PELICANS, Équations aux Dérivées Partielles, Maillages, Éléments Finis, Volumes Finis, Différences Finies, Programmation Objet, Composant Logiciel, Canevas Logiciel.

Introduction

The use of numeric simulation software is currently becoming increasingly prevalent: representing a physical situation by a computer programme has a real industrial, economic and even academic advantage. Its role is increased further when a full-scale experiment is impossible, as is especially true of many nuclear safety problems, such as simulation of corium behaviour outside the vessel (CROCO code), which is the basis for the development of the PELICANS platform.

▷ Constructing a numeric simulation involves, in our view, three different types of task: mathematical modelling, numeric modelling and computer implementation. The first aims to establish a method of approximating the physical situation in question, the second to establish a method of approximating the mathematical model and the third includes programming software to obtain a computerised approximation of the solution sought.

The incessant evolution of mathematical (motivated in particular by comparisons with experience) and numeric models is a fundamental characteristic of this activity. Falling behind in taking advantage of changes in physical knowledge through the inability of a software program to keep pace with them can be detrimental. In an extreme case, finding it impossible to make any changes at all represents a redhibitory failure (take, for example, the case of the WECHSL corium-concrete interaction software, which had to be abandoned in use due to defective computer structure).

Therefore, to avoid seeing the effectiveness of the whole modelling approach extensively impaired, there is a clear need to satisfy two key requirements:

- reduce the complexity, the length and the cost of the programming phase,
- ensure that quality software is obtained.

Meeting this twofold requirement is the goal of PELICANS.

▷ Two types of approach are commonly adopted nowadays to become more efficient in developing scientific software:

- constructing "software brick" systems dedicated to simulating a particular phenomenon or resolving a particular balance equation.
- in the mathematical models we are considering, exploiting a common denominator at the lowest level: partial differential equation systems. This involves supplying an implementation tool of solvers dedicated to resolving such equations, *i.e.* a software program used to write other software quickly and at minimum cost.

The second perspective has been chosen here, for two main reasons: firstly, it can be seen as a preliminary to the first; second and foremost, there is no universal numeric method in existence today to resolve any given problem. For example, quoting only those IRSN applications based on PELICANS, using a finite element discretization technique to resolve Navier-Stokes equations has proved most suitable for free surface problems as processed by the CROCO code, whereas the finite volume approach is preferred for the ISIS code, describing fire phenomena placing major requirements on the monotonicity of numeric schemes. Let us also add that this choice has opened the door to lesser applications performed frequently in academic work and relating to very varied fields of energy, fluid mechanics and solid mechanics.

▷ In addition, for the development of PELICANS, quality, flexibility and in particular adaptativity constraints, added to the difficulties inherent in scientific calculation, have prompted the development of a set of techniques and software tools. This acquisition can in return be re-used directly for the PELICANS applications and seems to us likely to make a determining contribution to their quality.

▷ Developed with these objectives in mind, PELICANS is a network of modules structured in their shape and their interactions and a set of principles and tools which dictates their design, evolution and use. The aim of this document is to introduce, comment on, justify and illustrate this dual characteristic (software components/conceptual principles) of both the foundation and of PELICANS. *************** REPRENDRE ************* It is laid out as follows: sections I.1, I.2, III.1 and III present the beginnings and ends of using PELICANS in developing numeric simulation software and section I.3 is devoted to a succinct description of programming techniques and tools developed in this context.

▷ The considerations addressed in chapters I and **??** can appear difficult to understand, even abstruse, and their implications in the real world of the programmer may not be immediately obvious. A formal and occasionally succinct tone has been adopted to provide a sufficiently comprehensive, yet concise, overview of PELICANS.

  ************** REPRENDRE ****************** On the other hand, a practical example of numeric simulation is developed fully, from the mathematical and numeric modelling phase up the computer-programming phase using PELICANS. The source code obtained is reproduced in its entirety and commented. It is clear that the physical context is simplistic, but it seemed to us sufficiently detailed to illustrate the practical role played by PELICANS in fine-tuning a numeric simulation as well as serving as a sort of "Getting started with PELICANS" guide.

  The two chapters concerned (§**??** et §IV) can be read independently from the others.

Reading Guide

This document may be read with several aims in mind and at several levels. The table below highlights the different sections with a few FAQs.

Question	Section
Where is PELICANS used?	§I.1
What is the PELICANS domain?	§I.2 §III
How is PELICANS used?	§III.1 §**??** §IV
How is PELICANS implemented?	§**??** §**??**
What is the PELICANS quality strategy?	§I.3 §**??** §I.6
Is there a PELICANS tutorial?	§**??** §IV

Typographic Conventions

The font **`courier`** is used for the fragments of code and for any term or notation in the PELICANS software component names (name of class or member function, for example).

Chapter I

Numeric Simulation Software Construction

I.1 Numeric Simulation Activity

Numeric simulation is a technique which aims to represent a given set of natural phenomena using a software program, where the results are reasonably close to the actual details.

Constructing a simulation has to be broken down into several stages.

1. Mathematical modelling: this involves describing the natural phenomena by a set of mathematical equations called mathematical model.
2. Numeric modelling: this involves here fine-tuning a technique which is used to obtain a discrete approximation of the mathematical model solution. The cost and accuracy of this approximation are two essential (and possibly opposing) qualities of the numeric model.
3. Computer implementation: the aim is a software program which applies the numeric modelling and is therefore a practical means of obtaining the desired approximation of the mathematical model.

The mathematical model is a conceptual representation of reality. It is qualified as abstract in that it underlines certain aspects of natural phenomena to eliminate others. This part of reality absent from the mathematical model is ignored either deliberately, for the model developer considers it secondary to the light he wishes to shed on the phenomena studied, or unintentionally, as the understanding of these phenomena is incomplete or even erroneous. The mathematical model is therefore intrinsically partial and imperfect. It is frequently improved, firstly as dictated by changes in underlying sciences, but most importantly as dictated by its confrontations with experimental reality.

The numeric model is hardly likely to be more fixed than the mathematical model it is approximating. Discretisation techniques change rapidly in line with changes in calculation methods. But the most striking aspect is the absence of universality and the many variants which generate method families rather than actual methods as such. “To each mathematical model its approximation technique and to each approximated problem its resolution technique” one could be tempted to say.

The problem of computer implementation is part of this changing context: it involves obtaining a practical resource (software program) to gather the fruits of mathematical and numeric modelling, if possible very quickly with respect to their evolution. In addition, this stage must be as little as possible the limiting factor of the numeric simulation. Being incapable of refining or changing a model because of the programming effort it creates would represent a failure of the overall approach.

This implementation stage could be made so much easier by using a dedicated tool. It is preferable that such a tool has an action radius limiting it exclusively to programming, so that the essential division is maintained between a mathematical modelling dimension, another numeric modelling dimension and lastly computer implementation. In addition, so that such a tool can be used in a wide range of simulations, the following observation has been made: firstly, the partial differential equations (PDE) are a typical ingredient of mathematical models and secondly, controlling a certain number of PDE systems can simulate a far greater number of physical situations. By way of illustration, the very traditional Poisson problem posed succinctly as follows:

$$\text{Find } \mathbf{u} : \Omega \subset \mathbb{R}^d \to \mathbb{R}^p \text{ such that } \begin{cases} \nabla^2 \mathbf{u} = f \quad \text{in } \Omega \\ \text{boundary conditions on } \Gamma \end{cases}$$

occurs in many contexts like calculations of temperature fields, electromagnetic potential vector, stream function in fluid mechanics, pressure correction in numeric models of Navier-Stokes equations, deformation in Arbitrary Lagrange-Euler methods and in numerous other fields, where the author has no hesitation in professing his ignorance.

Thus the numeric resolution of PDE systems seems to be the ideal business domain for a tool dedicated to the computer implementation phase of numeric simulation.

I.2 Numeric Simulation and Use of PELICANS

The preceding observations on the numeric simulation activity have prompted the development of the PELICANS software (French acronym for Open-Ended Platform of Component Libraries for Numeric Analysis ans Simulation: Plate-forme Evolutive de LIbrairies de Composants pour l'Analyse Numérique et Statistique); its essential characteristics are described below.

1. PELICANS is an implementation tool, in other words a software program used to construct other software, with an action radius strictly limited to the third phase in building a numeric simulation: the programming phase.
2. The PELICANS business domain is implementation of numeric solvers of partial differential equation systems (PDE).

An "end user" software program is traditionally defined as software used to calculate a set of natural phenomena, in other words like the finished product developed by a numeric simulation activity. PELICANS is not "end user" software, indeed nothing in PELICANS refers to any physical situation whatsoever. In particular, the notion of dataset, a file specifying the physical context and the calculation parameters for a simulation software program, means nothing in PELICANS: there are no PELICANS datasets.

A *PELICANS user* is a programmer who develops a software program using PELICANS as a tool. A *PELICANS application* is "end user" software programmed using the PELICANS functionalities.

I.3 Perceived Quality Criteria

As an implementation tool, in other words software used to construct other software, PELICANS provides a set of "off-the-shelf" elementary functionalities and a formalised code development methodology. This methodology is of course the one used in its own programming, but it is also available, with the related tools, to implement PELICANS applications, which thus benefit from quality criteria governing the PELICANS development with the least effort possible.

▷ In our software quality approach, we distinguish clearly between two families of criterion:

1. quality criteria perceived by the users, or external factors, such as ease of use, robustness and operating speed;
2. quality criteria perceived by the programmers of the software in question, or internal factors such as modularity or readability.

Ultimately, only the external factors count (otherwise known as "the customer is always right"). But for the user to benefit from visible qualities (external factors), the developers must use a set of programming techniques which satisfy internal factors (figure I.1).

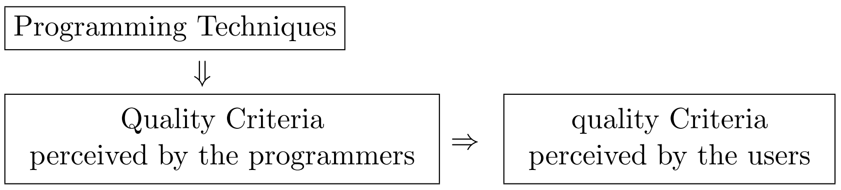

Programming Techniques
Quality Criteria perceived by the programmers
quality Criteria perceived by the users

Figure I.1: division of quality criteria into external and internal factors.

▷ We are going to describe succinctly the external factors under the priority given to them in the PELICANS development process and also the related programming techniques, as they can be adopted by the PELICANS user who is, remember, a programmer of a particular numeric simulation software (figure I.2).

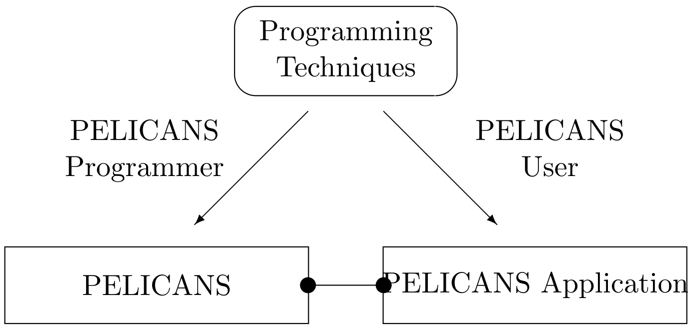


Figure I.2: programming techniques used during PELICANS implementation can be adopted by the PELICANS user, who is in turn programmer of a particular application.

Table I.1 presents a set of quality external factors, assembled into four categories. The importance of each of these factors is widely recognised [14] and we shall not repeat them here. On the other hand, the classification made is governed by a bias which merits discussion.

▷ It is normally considered that satisfying quality criteria involves a compromise: "we cannot have everything, choices have to be made". This judgement can be moderated: the now-established principles of software engineering can satisfy virtually all the factors in table I.1 quite correctly. For example, the most effective implementation of all is probably non-portable and non-open-ended. If nowadays a drop of a few percent in efficiency is acceptable, even a few tens of percent (which is really not very pessimistic when considering the lifetime of a software program and the evolution time of hardware performances), a fully-portable code can be obtained which is entirely open-ended and flexible (to the point of accepting non-portable but optimised implementation variants, thus recovering the supposed initial losses subsequently).

Table I.1 must be understood in the light of the previous discussion.

▷ The three factors revealed are *ease of use*, *re-usability* and *extendibility*, for they are primordial given the changing aspect of numeric simulation activity (§I.1). It also seems that they form objectives still difficult to achieve today in the field of scientific calculation[1], unlike the three criteria – *correctness*, *verifiability* and *robustness* –, which we consider absolutely essential but for which satisfaction raises fewer conceptual difficulties.

As for portability, for us it is a vital condition: dependence on the hardware or software environment in any way whatsoever is unthinkable. PELICANS therefore uses a standardised programming language,

[1]Let us quote A.Prosperetti and G.Tryggvason: "In many cases, a sophisticated piece of software does not survive the graduation of the student who wrote it" [International Journal of Multiphase Flow, 2003].

widely used everywhere, for which free, effective compilers exist independently of this or that hardware platform.

The efficiency factor is put to one side. * It is clearly central in scientific calculation. Our approach is to permit, if necessary, a slight loss in efficiency for the benefit of the three priority factors above. Experience shows that applying this strategy and the software engineering principles which we discuss briefly further on gives slightly greater gain in flexibility than loss in efficiency. The overall cost advantage is immediately apparent.

Ease of Use	ease with which people of different training and qualifications can install and use the software
Re-usability	ease of use in constructing numerous different applications
Extendibility	ease in adapting to changes in specification
Correctness	compliance with specifications
Verifiability	ease in defining acceptance procedures and in detecting failures during validation and operating phases
Robustness	appropriate reaction to abnormal conditions
Efficiency	ability to place as little load as possible on the hardware resources.
Portability	ease of transfer towards different hardware and software environments

Table I.1: external quality factors perceived by a software user.

I.4 An Object Technology Approach

▷ The internal factors ensuring the quality perceived by the users (external factors described above) are based firstly on formalisation of the object technology, available in the open literature [13, 14], and secondly, on an original implementation technique for concepts it deploys. The first point is described in this section, the second in the next.

▷ A computer program, or software system, is a network of interacting modules.

Simulation software programs are constantly being amended, dictated by changes in mathematical models and numeric methods. New functionalities are added and bugs are corrected. Note that the more the initial design phase is far away in time, the more the effort required to maintain a system in working order, to the point that the proportion of the global cost allocated to this task becomes dominant. The re-usability and extendibility criteria are therefore transformed from the status of objective to that of Utopia.

A detailed analysis of the architecture of such software systems shows that this pathology is caused by improper dependencies between the various modules.

▷ The PELICANS approach uses the object programming ideas and techniques to manage inter-module dependencies.

In object programming, the behaviour of a module can be modified without for all that changing its source, thanks to the notions of abstract base classes and derived (or inherited) classes [14]. An abstract class is fixed, but nevertheless represents an infinite set of behaviours through its potential derived classes. A module which handles an abstract class has a source code which cannot be modified as well as an open-ended behaviour which can be extended by the inheritance mechanism [14, 13].

▷ Adopting a design strategy based on the use of abstractions results in the *Dependency Inversion Principle*: high-level modules must not depend on low-level modules, everything depends on abstractions [13]. Following this rule leads to the fundamental internal software quality factor ensuring extendibility and re-usability, namely the *Open-Closed Principle*: software modules must be open for extensions but closed to any modification[2].

The abstractions are the cornerstone of this strategy: they allow firewalls to be installed to limit the spread of dependencies between modules.

▷ Object languages provide the basic *inheritance* mechanism for handling abstractions and the two related tools: *dynamic polymorphism* and *dynamic binding*. Designing an object architecture satisfying the quality criteria invoked previously requires a good grasp of these notions. But inheritance has two possible semantic interpretations.

- The first is *extension*: if a class **C** inherits from class **A**, all the services made available by **A** are automatically made available by **C**; in addition, **C** can add other services of its own or redefine those of **A**.
- The second is *specialisation*: if a class **C** inherits from class **A**, any object of type **C** is also an object of type **A** and any objects of type **C** can be used indiscriminately wherever an object of type **A** is used (Liskov Substitution Principle).

These two senses for inheritance can appear to contradict each other; each one has an element of truth, but from a different perspective.

Thus the notion of inheritance is subject to interpretation and therefore to misinterpretation. For this reason we have adopted the axiomatic formalism of the Design by Contract; this has the tremendous merit (and it is not the only one) of regulating its use and therefore of guiding the programmer through the semantic difficulties presented above.

▷ Under the Design by Contract theory [14], the relations between the elementary units of a software system are based on a formaised specification of mutual obligations: the contracts. All communication brings two parties into play: the *client* and the *supplier*. A class contract regulates the interaction of the supplier (*i.e.* an object of the class) with its clients (*i.e.* the rest of the world: all the other objects). It is issued and applied by using three types of *assertion* (or Boolean expression): *preconditions*, *postconditions* and *invariants*.

A precondition is an assertion which is evaluated at a function input and which formalises the constraints required for this function to perform correctly. The client is responsible for satisfying a precondition.

A postcondition is an assertion which is evaluated after the last instruction in a function has been executed and which formalises the properties guaranteed at the output of this function. The supplier is responsible for satisfying a postcondition.

Using Design by Contract has fruitful repercussions at all stages of a software program's life cycle and one of the most spectacular is the axiomatisation of the notion of inheritance: if a class **C** inherits from a class **A**, the Liskov Substitution Principle states that any object of type **C** must be capably of honouring all the contracts that an object of type **A** can honour. Thus, a member function redefined in class **C** must accept all the calls that the original function can accept and must guarantee at least as much as the original function. In terms of assertions: the redefinition of a member function can only replace

[2]Copying and modifying a module's source file to adapt its behaviour only facilitates the initial development phase, which remains the least costly of the software life cycle. This practice comes down to appropriating the copied code plus the related charges and responsibilities. In no way does it pool the qualification, maintenance and distribution efforts. Efficient re-use of a code involves treating it as a product where no notice is taken of the implementation (it is closed to modifications), but which can nevertheless have its behaviour modified (it is open for extensions) thanks to an inheritance mechanism, for example.

the original precondition by an identical or less restrictive assertion and the original postcondition by an identical or more restrictive assertion ("require no more, promise no less"). This rule for redefining assertions can be verified member function by member function for all the inheritance relationships to ensure correct use of abstractions and thus eliminate the inherent risk of erroneous interpretations.

▷ Thus the association of Object Programming and Design by Contract provides a software construction technique where the re-usability and extendibility criteria are central.

In addition, it can also increase considerably the ease of use by organising available functionalities as software components, a notion distinctly more advanced that the notion of sub-program (subroutine) in procedural programming.

A component is a unit element of a software system which can be qualified as customer-oriented in the sense that it has the following characteristics:

1. it can be used by other program elements (its customers);
2. it does not force the customers (and their authors) to be known to the author of the component.

In other words, a software component must be usable by programmers who develop new applications theoretically not envisaged by the component author.

A component client only knows the author through his formalised specification in terms of software contract, as assertions. The internal details are kept concealed, not to hide them as something with a negative connotation, but to help the programmer of a clinet module by limiting the volume he has to learn.

Thus, a PELICANS component is a data structure enhanced with functions (called member functions or methods) representing firstly, the operations required to manipulate this data structure and secondly, the range of services offered to the outside world. The clients of a component interact with it through as small an interface as possible, formalised in its main thrusts as follows:

$$\begin{array}{c} \text{for each member fuction} \\ \text{(of a class and its ancestors)} \end{array} \quad : \quad \begin{cases} + \text{ name} \\ + \text{ return type} \\ + \text{ parameters type} \\ + \text{ preconditions, postconditions} \end{cases}$$

A component's documentation reveals all these headings for each member function and can be extracted from a source code (preconditions and postconditions are executable instructions) thanks to a specific tool.

I.5 An Implementation Technique for Object Concepts in C++

▷ The programming language adopted by PELICANS is C++.

▷ The following reasons are behind the choice of C++ to implement PELICANS.

1. `C++` is a language standardised since 1 September 1998, under the reference ISO/IEC 14882:1998.
2. `C++` is very widespread in both industry and academic research and it is taught in many university curricula.
3. Free, high-performing `C++` compilers exist for the vast majority of hardware platforms and operating systems.
4. `C++` encompasses the bulk of object potential required by the desired architecture.
5. Programming intensive calculation software programs in `C++`, compared with programming in a conventional procedural language (like FORTRAN or C), does not penalise their efficiency significantly, as long as certain recognised and fully documented pitfalls are avoided [16, 15].

`C++` does however have some weaknesses when used to put the precepts laid down in **??** into practice. No native support is offered to Design by Contract (especially for redefining assertions in the derived classes) or for the self-documentation principle. In addition, certain delicate, technical tasks are left entirely up to the programmer, like the memomry management[3]. Lastly, the language `C++` and its standard library are together tremendously rich, as much through the multitude of functionalities (and their variants) proposed as through the number of concepts undertaken.

▷ In this context, the practical application of object technology principles discussed in paragraph **??** necessitated the development of a formal `C++` design and implementation approach; the details can be found in a specific document [1].

The main ideas are as follows: using the `C++` complex potential is deliberately reduced to what is strictly necessary[4]; in particular, special care has been taken so that the programmer and the PELICANS user are only confronted by the object methodology via the traditional principles of procedural programming increased by the notion of *inheritance*, with the related tools: *polymorphism* and *dynamic binding*. In this way, the PELICANS software uses an object language for its implementation (`C++`), but depends very little on it, for it does not base itself on its specific features but only on the primal notions that exist in all object languages. On the other hand, these primal notions are formalised in terms of Design by Contract to make them far more rigorous in use than provided for naturally by `C++`.

▷ It must be underlined that the implementation technique for object concepts proposed by PELICANS is conveyed in reality by a `C++` language use methodology: the programmer and the PELICANS user write the text of their respective programs in standard `C++`, defined by the 1 September 1998 standard, so that the software programs developed only require one compiler complying with this standard for their use (like the free compiler gcc GNU in versions 3.2 and later).

▷ In a desire to comply with the requirements of purely object designs, we have insisted that the only modules (*i.e.* basic units of software decomposition) of the PELICANS architecture are the classes. Distinction is made between their respective responsibilities according to the three following points.

- The classes are arranged in layers, so that the user is only confronted by a small number of them which form a *facade*. The internal part of PELICANS is encapsulated (but not cached). It can be accessed but this is neither necessary nor mandatory. This layer arrangement reduces the amount of learning for the user considerably (figure I.3).
- The PELICANS *facade* contains abstract classes from which the user can derive concrete classes to implement his application. In addition to these abstract classes, the PELICANS facade contains a set of concrete classes that the user can use as supplies of elementary functionalities to implement the classes specific to his application.

The involvement of these last two points is tricky to convey other than by example; this is the objective of chapters **??** and IV in this document. We shall restrict ourselves here to the main ideas.

In software engineering terms, PELICANS is a *framework* and a set of re-usable and adaptable *software*

[3]What we present here as a weakness, others consider as an advantage. Memory management problems are so frequent and so penalising that the standard `C++` deliberately offers numerous possible solutions without imposing any of them. This wealth can however result in risks to which we do not wish to be exposed. It is in addition symptomatic to note that even the introductory works to `C++` language have to devote a significant section to memory management techniques[10].

[4]These notions of "complexity" and "strictly necessary" are of course arbitrary: what is tricky for some is elementary for others. The strategy for using `C++` in PELICANS arises from the deliberate choice to promote *re-usability* and *extendibility*, whilst guaranteeing *ease of use* for all diversely-qualified players (with absolutely no value judgment). No initial training in software engineering is required to use or even programme PELICANS. Thus, certain aspects of C++ have been deliberately left to one side like multiple inheritance or generic programming, although great authorities in object programming consider these notions essential [14, 21]. Under no circumstances do we question these judgements. But, after all, quality software programs have been developed in purely procedural language. Therefore, why not select the minimum required to manage the abstractions (§**??**) by omitting the rest, which demands wide practical experience in particular, and thus exonerate the user and the PELICANS programmer from acquiring a skill in a tool to the obvious benefit of a skill in the underlying business field.

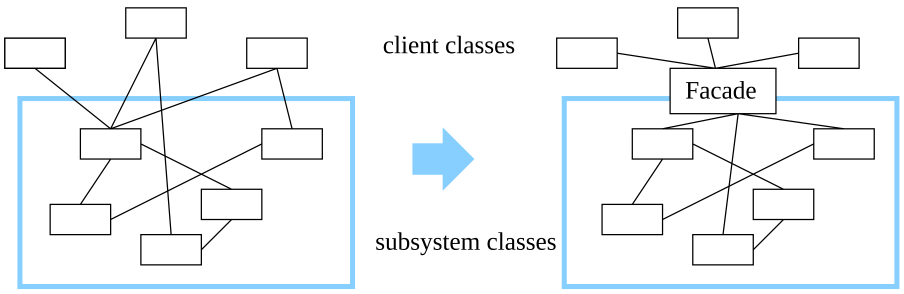


FIGURE I.3: Design pattern called “facade” [6]: the façade provides a unified interface for all the interfaces in a sub-system, making it easier to use.

components.

A *framework* is a collection of classes, some concrete and others abstract, designed to be used with each other. Each one contributes to the operation of the whole with its own responsibilities. The services rendered by a class and its interactions with the other classes in the *framework* are totally formalised in terms of software contracts, but some of these services may not be implemented. In others words, we know who does what, but we do not necessarily know how. The “how” is dependent on every time the *framework* is applied specifically to a special case. Thus, a *framework* defines the structure common to a family of applications, formalised in terms of abstractions. Developing an application involves supplying the missing pieces.

A software program based on PELICANS is constructed in two stages.

Design: this stage consists of identifying one or more abstract classes in the PELICANS *framework* which specify certain services required by the current application, without for all that implementing them in an adapted way (possibly without implementing them at all).

Implementation: this stage consists of programming the concrete classes derived from abstractions identified during the design phase. To achieve this, the off-the-shelf software components supplied by PELICANS are used.

To summarise, developing a software program based on PELICANS (figure I.4) involves implementing certain abstract classes (PELICANS as a *framework*) by calling on off-the-shelf, elementary functionalities (PELICANS as a *software component* library). This approach is sometimes called *framework instantiation*, with the application obtained *framework instance*.

▷ The PELICANS *framework* can be supplement at several levels. The highest, and least restrictive, level is normally intended for any program seeking to take advantage of the object concept implementation technique in `C++` described previously. In this context, the role of the user is to derive a concrete class from the *plug-point* called **`PEL_Application`** and to implement the member function **`run`**. In this way, any software can be programmed as an instance of the PELICANS framework: all that is needed is to transpose the instructions that would hase appeared in the main program to the member function run. The constraint granted is to link with the compiled PELICANS library, but the profits are many and immediate. The most significant are listed below.

- The mechanisms relating to the Design by Contract are available, in particular the systematic method set up by PELICANS to satisfy the Liskov Substitution Principle which is used to avoid hierarchies of conceptually incorrect classes [12, 1].
- Memory is managed under the methodology defined by PELICANS [1].
- The data packets transferred between objects or externally (code inputs-outputs) are structured. For this purpose, PELICANS defines a structure format for hierarchical data (called HDS for Hierarchical Data Structures) so that the information is classified as a tree structure where each

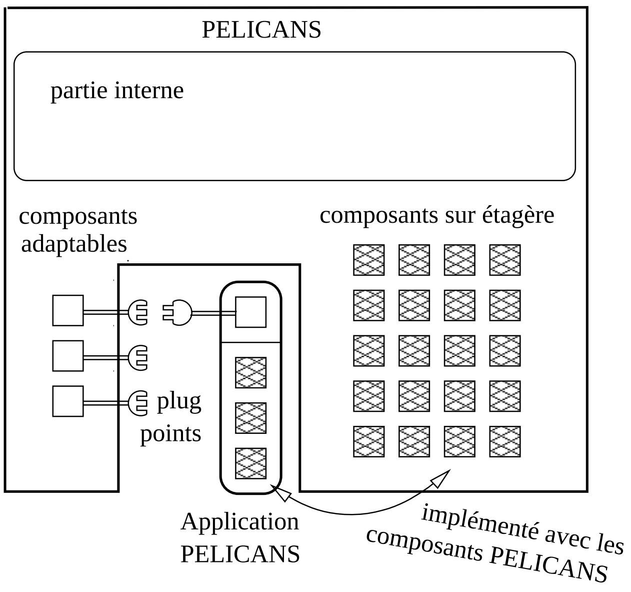


FIGURE I.4: *Frameworks* and Adaptable and re-usable and *software components*.

node (which we call **MODULE**) can contain other nodes or single inputs which are pairs { keyword ; data }, the data being typed[5]. All the management techniques for these data structures are available (creation, modification, navigation, qualitative and quantitative comparison, control, reading/writing in ascii or binary format).

- A persistence mechanism is available, which can be used to back-up onto disc the dynamic execution state of an application at an instant chosen by the user. A calculation can thus be interrupted then recovered later exactly where it was left off. This potential is especially useful in safeguarding an execution against accidental stoppages of the hardware platform (and thus avoiding losing hours if not days of calculation).
- The application generated benefits from the management of the PELICANS unit tests. Thus, any class developed can be associated with a test class which verifies the functionalities implemented. The test classes are part of the application in the same way as the classes they are testing and are in particular subjected to the same configuration management. In addition, these unit tests can be executed an any instant, without recompiling the application, by simply adapting its dataset.
- The PELICANS self-documentation tool is available. It is used to generate and put into HTML format the visible part of each class, as defined in paragraph **??**, by grouping the name, result type and parameters as well as the description of each member function of the class in question and all its ancestors (information extracted from header files) and the preconditions and postconditions which define the exact specifications of all the services rendered (information extracted from implementation files).

The three last points bring complete applications into play: a documentation generator, a unit test launcher and an application launcher from a previous system state back-up. These three applications are themselves framework instances which are an integral part of the PELICANS platform. The executable of a PELICANS application therefore contains, in addition to its specific features, everything required to manage the unit tests, the calculation back-up-recoveries and reference documentation generation[6].

[5]By way of example, below are listed a few recognised types: Boolean, integers, reals, strings, typed expressions bringing variables into play, vectors and tables of one of these simple types.

[6]We illustrate here one of the basic principles which leads to object architectures: "real systems have no top" [14]. A software program is not intended to carry out a task (the highest-level task, the "top") which can be broken down

I.6 Agile Development Process

▷ The appropriation of the preceding methodology by the developers remains the fundamental key to success of the entire approach. For this reason, a model of this particular life cycle, belonging to the family of “agile methods” was adopted when developing PELICANS. We present it succinctly here, for we believe it particularly well suited to scientific programming in general.

In the early 2000s, in reaction to the failures in applying traditional development methods (like the so-called predictive approaches) and in contrast to their litigious complexity, a group of specialists from the world of software met to work towards promoting iterative methods, with flexible planning, suiting both the change in context and in project specification. The word “agile” was chosen as overall qualifier and the community was christened “Agile Alliance” (**http://www.agilealliance.org**). Its work culminated in a manifesto (**http://www.agilemanifesto.org**) which highlights four values, described below, set out in twelve principles [13] and finally development practices, presented in the following paragraphs.

The agile methods promote an iterative development style centred on people and which focuses on customer satisfaction through continuous integration of a fully functional software program. They promote:

1. interactions and individuals over processes and tools;
2. delivery of a functional software program over exhaustive documentation;
3. communication with the customer over contractualisation;
4. taking change into account over respecting a fixed schedule.

Whilst recognising the value of elements on the right of the statements above, they are more in line with those on the left.

The priority is to satisfy the customer by delivering operational versions to him very early and regularly (a functioning application is the first indicator of project progress), in intervals of no more than one to a few weeks. The approach is incremental and aims for a compromise between the initial specifications (on which the planning is based) and the end result which can frequently be poles apart, by absorbing the modifications throughout the development cycle. The project owner can thus:

- refine the specifications, change their priorities during construction based on what has already been constructed and a better understanding of the need;
- adjust the plan (content, delivery delay) during construction based on the measured rapidity of the development team.

Two operational versions differ in a set of functionalities (or functional units), called increments. An increment is created as follows:

- increment planning: the project manager and the project owner negotiate the functional scope of the increment;
- increment: the project manager constructs what has been planned (design, encoding and tests);
- increment review: the project owner assesses and validates (or not) the functionalities implemented during the increment.

This approach offers an experience recovery point at each increment end, which thus minimises the risks. Customers and developers therefore work hand-in-hand throughout the project (ideally every day).

into sub-tasks, which themselves can be broken down into sub-sub-tasks and so on (top-down design). A software program is rather intended to accomplish a set of tasks around a certain theme (in our case everything seemingly necessary to administer, in the widest sense, numeric simulation software), without necessarily imposing a hierarchical relationship (what relationship can be imagined between resolving a partial differential equation system and editing HTML documentation?).

I.6.1 Software modifications

Whether motivated by a development or maintenance need, any software modification (*i.e.* of the source but also of the administration tools, use examples and documentation) corresponds to an increment to which is associated a micro-task (term which in PELICANS project jargon designates the work related to an increment).

All micro-tasks are managed by the support software called “Tracker”. This internally-developed tool is used to submit increment requests, classify them (Fault, Improvement, Development, Administration), allocate them a priority level and organise taking them into account and monitoring related actions. The full history is kept and may be consulted.

In computer terms, the software program is made up of a set of files (for example `C++` files with the extension **`.hh`**, **`.cc`** or **`.icc`**, perl files with the extension **`.pl`**, makefile files with the extension **`.mak`**, documentation files with the extension **`.tex`** or **`.html`**, *etc.*), for which the change is managed by the support software CVS (Concurrent Versions System). This “open-source” configuration management tool is integrated into the quality process by promoting team work firstly, because access is possible to the files making up the project concurrently, and secondly, because the full history can be kept.

Tracker and CVS are closely linked in the PELICANS project.

- A constituent software file can only be modified in a version increment and therefore has to be associated with a micro-task.
- Tracker allocates an identification number to each micro-task.
- A file is modified exclusively via CVS (known as “commit” in the specific jargon). It is only accepted by CVS if it is indeed associated with a valid micro-task identification number; this number is archived in the file history.

As regards the two configuration management (CVS) and micro-task management (Tracker) tools, the difference between maintenance and development lies exclusively in the motivation of the changing increment. The conceptually-relevant abstraction is therefore more one of “modification”, which we will adopt later on.

I.6.2 Increment Integrity and Continuous Integration

By its very definition, a version is operational and deliverable, in the sense that it satisfies a set of criteria, themselves defined in the version. Let us mention the most important:

- The class reference documentation can be constructed automatically and without error by the specific tool within PELICANS;
- The programming rules, which may be verified by this same tool, are respected;
- The units tests, which are an integral part of the software, are executable without error
- The sample application tests, which are an integral part of the version, are executable without error.

These criteria are verified every night by a “computer robot”, on a given machine, with additional tests for the contract of member functions called. Every Monday, the version called DEV is delivered to the users at IRSN, in the sense that it includes the increments of the previous week (provided the last session of the daily integrity tests was successful).

Two notions essential to the agile methods are appearing here:

- **continuous integration** , a consequence of the sub-division of tasks into micro-tasks and guarantee of stability;
- **importance of tests** which characterise the immediately-operational nature of an increment.

Software modifications are driven by tests wherever possible: the first stage of a micro-task is to imagine the associated tests (which, in the case of a maintenance action, come directly from the fault detected) before attempting to revise the source code. The benefit is instant:

- the programmer adopts the customer's viewpoint;
- the dependencies are minimised so that the code modules can be tested separately;
- the tests act as internal documentation: they are the original customers which illustrate the desired use;
- the increment is tested by construction.

This approach, without being stipulated because idealised, is recommended for all micro-tasks. The related tests (unit tests or application sample tests) thus are an integral part of the increment and will be added to the version, just like the sources.

I.6.3 Continuous Design and Refactoring

The traditional, predictive-type development processes attach a central importance to the initial design of new software programs and major developments. Finished product of an intellectual work preceding the encoding, its elegance and formal beauty rarely lasts (not to say never) beyond the very first versions implemented. The progressive degradation of the software quality as the initial design phase is left behind has several deep and inevitable causes:

- the requirements change in a way that the initial design had not anticipated;
- these changes introduce new, unscheduled dependencies between the system modules;
- as the dependency structure degrades, in nature and intensity, the system becomes increasingly rigid, fragile and immobile.

The temptation is to think that only the "bad" designs degrade and therefore to devote even more effort and resources to this initial design phase. This is refusing to see the intrinsic volatility of requirements, particularly in scientific programming.

Conversely, the agile methods recommend:

- not insisting unduly on a detailed initial design;
- moving forward by increments, as described previously, with a reduced yet relevant design in terms of fundamental principles like the Dependency Inversion Principle, the Open-Closed Principle and the Liskov Substitution Principle;
- maintaining and changing this design during the entire software life cycle.

Thus the emphasis is on **on-going improvement** of the design rather than its initial quality ("continuous care versus initial design"). Even if this proves slight, it will be improved gradually, increment by increment, whilst remaining reactive to change. Note that the same techniques are used as those applied to developing exhaustive designs, but throughout the life cycle:

- undesirable dependencies are broken and diverted;
- dependency firewalls are erected to prevent changes propagating;
- design patterns are applied to solve classic problems.

Refactoring is a direct result of this approach. This is a practice aiming to improve the design of a system without changing its behaviour (automatic tests and a facade architecture, as introduced in PELICANS, are thus extremely useful). The agile methods are based on refactoring so that the design emerges as a result of the incremental approach.

Chapter II

PELICANS Use Example

In computing, for a rapid demonstration of software development tools, it is customary to write a short program, as simple as possible, with the single goal of displaying the words “Hello world!” on the screen.

We are conforming to this tradition here by producing a slightly more ambitious PELICANS application which emits a name greeting on the screen and in a file.

II.1 Design

PELICANS is the structure common to a family of applications, formalised in terms of abstractions, in other words in terms of classes with declared but non-implemented member functions (i.e. abstract classes).

The main program is part of PELICANS and not accessible to us[1]. Besides, we have no need for exact knowledge of the operations it performs, as the only information required by a PELICANS user is described below.

▷ A PELICANS application is executed by typing an instruction in the command line such as:

```
exe data.pel
```

where **exe** designates the executable obtained by compiling implementation files specific to solving the problem in question then by linking with the PELICANS library. The **data.pel** file represents a hierarchical data structure in PELICANS format, whose main role is to serve as dataset to a *developed application.*

Listing II.1 gives an example of a **data.pel** file. It contains the text of a data structure in PELICANS format, where the information is organised as a hierarchy with nodes called **MODULE**s. Thus, each **MODULE** can contain other **MODULE**s or single entries which are pairs {keyword; data}, where the data are typed.

```
MODULE PEL_Application
   concrete_name = "HC_Greeting"
   name = "PELICANS"
   MODULE greeting_card
      institution = "Institut de Radioprotection et de Sureté Nucléaire"
      town = "Saint Paul Lez Durance"
      zip = 13115
   END MODULE  greeting_card
END MODULE PEL_Application
```

Listing II.1: sample **data.pel** file.

[1]We are being deliberately simplistic in describing here the most common situation [1].

To run through such a hierarchical data structure, PELICANS makes a class of explorers called **`PEL_ModuleExplorer`** available, where the instances, ones attached to a **`MODULE`** can access securely (verification of existence, consistency and type) the entries of this MODULE or create other instances attached to the **`MODULE`**s it contains.

▷ Executing a PELICANS application proceeds under a sequence with the following five stages.

Stage 1: Big-bang.

Stage 2: The data structure contained in **`data.pel`** is read and stored in the memory. The only constraint imposed on it is that its highest-level **`MODULE`** is called **`PEL_Application`** and that it contains a entry of keyword **`concrete_name`** with the argument a string identifying a class derived from **`PEL_Application`**.

Stage 3: An object of type the class derived from **`PEL_Application`**, designated by the **`concrete_name`** argument, is created.

Stage 4: The member function **`run`** associated with this object is called.

Stage 5: Finalisation.

The PELICANS user only needs concern himself with operations performed in stages 1 and 5.

▷ Thus the **`PEL_Application`** class is an abstract class which declares, but does not implement, the member function **`run`** (**`PEL_Application`** is a *plug-point*). This **`run`** function is supposed to perform the operations specific to the developed PELICANS application. We are therefore going to create a class derived from **`PEL_Application`**, called **`HC_Greeting`**, intended to display the greeting message.

▷ A precise approach must be followed to derive a class from a PELICANS *plug-point*. The various milestones in this process (named *plug-in* in the following) draw their basis from the `C++` techniques relating to the design pattern called “pluggable factory”. However, there is no need to understand them to use the functionality they produce. A PELICANS user simply has to follow the instructions in the **`FRAMEWORK INSTANTIATION`** heading in the reference documentation for the *plug-point* considered.

▷ Let us comment briefly on the implications of this approach in the **`HC_Greeting`** header and implementation files.

In listing II.2, line 1 means that **`HC_Greeting`** inherits from **`PEL_Application`**. The declarations of the destructor (line 11), the copy constructor (line 12) and the assignment operator (line 13) in the private zone stem from the PELICANS memory management technique [1].

The **`PROTOTYPE`** class attribute, declared in line 28 in the header file (listing II.2), forms the cornerstone of the *plug-in* mechanism. Its initialisation, in line 1 of the implementation file (Listing II.3) calls the constructor:

```
    PEL_Application( std::string const& name )
```

whose role is to christen the standard class: the litteral string **`"HC_Greeting"`** used as an argument in the constructor call (line 4 of listing II.3) is the value required for the data item associated with the keyword **`concrete_name`** appearing in the **`data.pel`** file (listing II.1) under the heading **`MODULE PEL_Application`**, so that the **`PEL_Application`** object created by the main program is of **`HC_Greeting`** type.

The virtual member function **`create_replica`** (declared in **`PEL_Application`** without being implemented) is implemented by the class **`HC_Greeting`** (listing II.3). Its basic role is to call the constructor:

```
    HC_Greeting( PEL_Object* a_owner, PEL_ModuleExplorer const* exp )
```

which is the one actually used to initialise an object of type **`HC_Greeting`**. The description of this constructor forms the actual start of the specific aspects of implementing the **`HC_Greeting`** class.

All this is in fact generic to the *plug-in* mechanism from **`PEL_Application`** and will be transposed as is for the classes developed in chapter IV.

```
class HC_Greeting : public PEL_Application
{
   public: //-----------------------------------

      virtual void run( void ) ;

   protected: //--------------------------------

   private: //----------------------------------

     ~HC_Greeting( void ) ;
      HC_Greeting( HC_Greeting const& other ) ;
      HC_Greeting& operator=( HC_Greeting const& other ) ;

      HC_Greeting( PEL_Object* a_owner,
                   PEL_ModuleExplorer const* exp ) ;

   //- Plug in

      HC_Greeting( void ) ;

      virtual HC_Greeting* create_replica(
                           PEL_Object* a_owner,
                           PEL_ModuleExplorer const* exp ) const ;

   //- Class attributes

      static HC_Greeting const* PROTOTYPE ;

   //- Attributes

      std::string NAME ;

      bool HAS_CARD ;
      std::string INST ;
      std::string TOWN ;
      int ZIP ;
} ;
```

Listing II.2: extract from the **HC_Greeting** class implementation file.

```
HC_Greeting const* HC_Greeting::PROTOTYPE = new HC_Greeting() ;

HC_Greeting:: HC_Greeting( void )
   : PEL_Application( "HC_Greeting" )
{}

HC_Greeting:: ~HC_Greeting( void )
{}

HC_Greeting*
HC_Greeting:: create_replica( PEL_Object* a_owner,
                              PEL_ModuleExplorer const* exp ) const
{
   PEL_LABEL( "HC_Greeting:: create_replica" ) ;
   PEL_CHECK( create_replica_PRE( a_owner, exp ) ) ;

   HC_Greeting* result = new HC_Greeting( a_owner, exp ) ;

   PEL_CHECK( create_replica_POST( result, a_owner, exp ) ) ;
   return( result ) ;
}
```

Listing II.3: extract from the **HC_Greeting** class implementation file.

II.2 Implementation of Specific Aspects of the Current Application

As stated previously, the execution of a PELICANS application starts by reading a file containing a hierarchical data structure; an object of the class derived from **PEL_Application** named in the data structure is then created and the member function **`run`** is called for this object. The programmer of the current PELICANS application is not responsible for triggering these three operations. His role in this sequence of operations is to implement the **`run`** function and to implement the constructor called by **`create_replica`**.

In `C++`, the basic creation mechanism for a class object automatically includes an initialisation phase; the role of constructors is to perform this initialisation. The constructor called by **`create_replica`** is declared in lines 15-16 of listing II.2. When called, the argument **`exp`** is attached to the **`PEL_Application`** module of the hierarchical data structure contained in the file **`data.pel`** (listing II.1).

```
HC_Greeting:: HC_Greeting( PEL_Object* a_owner,
                           PEL_ModuleExplorer const* exp )
   : PEL_Application( a_owner, exp )
   , NAME( exp->string_data( "name" ) )
   , HAS_CARD( false )
{
   if( exp->has_module( "id_card" ) )
   {
      PEL_ModuleExplorer const* se =
                        exp->create_subexplorer( 0, "greeting_card" ) ;
      HAS_CARD = true ;
      INST = se->string_data( "institution" ) ;
      TOWN = se->string_data( "town" ) ;
      ZIP  = se->int_data( "zip" ) ;
      se->destroy() ;
   }
}
```

Listing II.4: extract from the **`HC_Greeting`** class implementation file.

```
void HC_Greeting:: run( void )
{
   PEL_LABEL( "HC_Greeting:: run" ) ;

   std::cout « endl « "Hello " « NAME « std::endl ;
   if( HAS_CARD )
   {
      std::string ff = NAME + ".card" ;
      std::ofstream os( ff.c_str() ) ;
      if( !os )
         PEL_Error::object()->raise_plain( "file opening failed" ) ;
      std::cout « "Editing greeting card: " « ff « std::endl ;
      os « "TO: " « NAME « std::endl ;
      os « "    " « INST « std::endl ;
      os « "    " « ZIP  « " " « TOWN « std::endl ;
      os « "THANK YOU!" « std::endl ;
      os.close() ;
   }
}
```

Listing II.5: extract from the **`HC_Greeting`** class implementation file.

Chapter III

Implementation of PDEs Solvers

The PELICANS architecture has been designed to be as close as possible to the underlying business domain whilst remaining sufficiently flexible to satisfy the criteria of re-usability and extendibility. Thus, an effort of abstraction of the domain of numeric resolution of PDEs systems has highlighted concepts reflected by the PELICANS architecture.

III.1 Prerequisite to Using PELICANS

PELICANS is a PDEs solver implementation tool. Before any use, in other words before any development of a *PELICANS application*, the problem must be defined fully (mathematical model) and the numeric method to resolve this problem (numeric model) must be chosen from those that PELICANS can manage (§III): this is the formalisation phase for the application requirements.

III.1.1 Defining the Problem

The partial differential equation system for which a solution is sought must be clearly stated, in terms of:

1. spatial and time domains;
2. unknown fields;
3. equations verified by the unknown fields;
4. boundary conditions;
5. initial conditions.

This formalises what is called the "*continuous probleme*" (for the unknowns are time and space functions possessing certain regularity properties).

III.1.2 Numeric Method

The numeric scheme chosen to approximate the continuous problem must be established in all its details.

The continuous problem is a normally coupled and non-linear PDEs system. The choice is between approximating it "outright" or in sub-stages. This strategy must be formalised and the couplings between the various potential sub-stages established in advance. The discretization of each of these sub-stages and the techniques for resolving related algebraic systems must also be formalised.

III.2 Shaping the Mathematical Model

The first phase of the numeric simulation activity results in a mathematical model containing three fundamentally different ingredients which must be clearly distinguished:

1. the unknown fields, represented by $\mathbf{W}$;
2. the equations verified by the unknown fields:

$$\text{Find } \mathbf{W} \in \mathcal{S} \text{ such that } \begin{cases} R(\mathbf{W}) = 0 \quad \text{in } Q \text{ (or in a part of } Q) \\ \text{boundary conditions and initial conditions} \end{cases} \tag{3.1}$$

3. the parameters of the continuous problem, *i.e.* the data appearing in the equations verified by the unknown fields.

The nomenclature and notations used are as follows:

- Q is a spatio-temporal domain such that the spatial position is represented by a vector of $\mathbb{R}^d$, where d is the dimension of the geometric space.
- $\mathcal{S}$ is a space of functions defined in Q (or in a part of Q) formed by the product of p spaces of scalar or vectorial functions.
- R is a differential operator defined in $\mathcal{S}$ with values in $\mathcal{S}$.

▷ If the problem (3.1) is considered as a PDEs system, then p represents the number of equations in the system and the p components of $\mathbf{W}$, denoted W_i $(i = 1, \ldots, p)$, the unknown functions. Thus, there are as many unknown functions as there are PDEs.

▷ The unknown $\mathbf{W}$ depends on the time t and the geometric position $\mathbf{x}$. The name “unknown field” used subsequently results from this latter dependency. Physically, the parameters can be fields depending on time and space, just like $\mathbf{W}$, but unlike $\mathbf{W}$ they are assumed known in the mathematical model.

▷ The problem (3.1) is said to be 1D if $d = 1$, 2D if $d = 1$ and 3D if $d = 1$. It is said to be stationary if the time t does not intervene in either the boundary conditions or the PDEs system. PELICANS can be used to develop applications which are 1D, 2D and 3D, not as three different software versions, but as single version where the instructions are parametrized by the space dimension (which has tremendous advantages, particularly for the validation and maintenance activities).

III.3 Two Notions of Discretization

The aim of the numeric modelling phase is to fine tune a technique to obtain a discrete approximation of the solution to the continuous problem. Two stages should be considered.

- Initially, a discrete representation of the unknown $\mathbf{W}$ to the problem must be built.

  Let us consider one of the p components of $\mathbf{W}$, which we will denote $\mathbf{u}$. Let us fix the time t and consider the spatial dependency of $\mathbf{u}$.

  Discretising $\mathbf{u}$ involves choosing a representation close to $\mathbf{u}$ bringing into play a finite number $N^{\mathbf{u}}_{\text{dof}}$ of numeric values which are assembled in a vector $\mathcal{U} \in \mathbb{R}^{N^{\mathbf{u}}_{\text{dof}}}$. A component of $\mathcal{U}$ is called *degree of freedom* of $\mathbf{u}$ (abbreviated to `DOF`).

  The field $\mathbf{u}$ verifies certain hypotheses of regularity, which we express abstractly by writing that $\mathbf{u}$ belongs to a certain functional space $\mathcal{S}^{\mathbf{u}}$. The discretization of $\mathbf{u}$ comes down to establishing a formal link between $\mathbf{u}$ and $\mathcal{U}$, rendering a conceptual approximation operation.

$$\mathbf{u} \in \mathcal{S}^{\mathbf{u}} \underset{\text{approximation}}{\overset{\text{discretization}}{\rightleftarrows}} \mathcal{U} = \left(u_k\right)_{0 \le k < N^{\mathbf{u}}_{\text{dof}}} \in \mathbb{R}^{N^{\mathbf{u}}_{\text{dof}}} \tag{3.2}$$

The link between $\mathbf{u}$ and $\mathcal{U}$ is governed by the type of discretization. PELICANS has families of possibilities, all based on the notion of finite element meshing of the geometric domain.

The temporal dependency is taken into account by the possibility of managing several discrete representations $\mathcal{U}^{(\ell)}$ (all the same type) of $\mathbf{u}$, where, for example, $\ell = 0$ identifies the current time step and $\ell = 1$ the previous time step.

- Secondly, an approximate problem must be established, whose solution is the discrete representation of the unknown $\mathbf{W}$. The result of this stage is the numeric model. For its programming, PELICANS is restricted to providing functionalities relating to the techniques using a meshing of the geometric domain.

  It can happen that the unknown fields intervening in the partial differential equation systems have the value of certain of their components imposed in a part of Ω or its boundary (for example through Dirichlet boundary conditions). This constraint can be conveyed by the fact that the value of certain degrees of freedom of the discrete representation $\mathcal{U}$ of a component $\mathbf{u}$ of $\mathbf{W}$ must be considered as fixed *a priori*, to eliminate them algebraically from the discrete problem. Let us therefore call $\mathbb{U}^{\mathbf{u}}$ all the indices k of degrees of freedom u_k which are in fact unknowns of the discrete problem.

  In practice, the unknowns are indexed by a sequence of contiguous integers starting at 0:

$$k \in [0, N^{\mathbf{u}}_{\text{dof}}[\quad ; \quad k \in \mathbb{U}^{\mathbf{u}} \quad \xrightarrow[\text{indexing}]{\text{contiguous}} \quad I \in [0, N^{\mathbf{u}}_{\text{unk}}[\tag{3.3}$$

  This indexing is used to gather the reals u_k for $k \in \mathbb{U}^{\mathbf{u}}$ in the vector of discrete unknowns associated with $\mathbf{u}$ denoted $\mathbf{U} \in \mathbb{R}^{N^{\mathbf{u}}_{\text{unk}}}$.

  All these vectors $\mathbf{U}$ for all the components $\mathbf{u}$ of $\mathbf{W}$ are the unknown of the discrete problem. The responsibility for managing different indexings for the degrees of freedom and the discrete problem unknowns is endorsed by a dedicated component called **`PDE_LinkDOF2Unknown`**, which frees the PELICANS user from the related practical difficulties.

These two stages together are called *discretization* of the problem (3.1).

III.4 Finite Element Interpolation

▷ A finite element meshing (Figure III.1), as defined by PELICANS, is a partition of the closure $\overline{\Omega}$ of the geometric domain $\Omega \subset \mathbb{R}^d$ into a finite number of cells:

$$\overline{\Omega} = \bigcup_e \mathcal{C}_e$$

such that:

- each cell $\mathcal{C}_e$ is a polyhedron of $\mathbb{R}^d$;
- two distinct cells have non overlapping interiors;
- each face of a given cell is either the face of another cell or a part of the boundary Γ of Ω.

PELICANS can be used to manage 1D, 2D and 3D finite element meshings, where the cells can be segments, triangles, quadrilaterals, rectangles, hexahedra or tetrahedra.

The finite element meshings allow for the definition of *finite element discretizations* of fields.

▷ Let us consider one of the p components of $\mathbf{W}$, which we will denote $\mathbf{u}$ and let us fix the time t. The finite element method can be used to construct an approximation $\mathbf{u}_h$ of $\mathbf{u}$ as follows:

$$\mathbf{u}_h(\mathbf{x}) = \sum_{0 \le k < N^{\mathbf{u}}_{\text{dof}}} u_k \, \varphi^{\mathbf{u}}_k(\mathbf{x}) \quad \forall \mathbf{x} \in \Omega \tag{3.4}$$

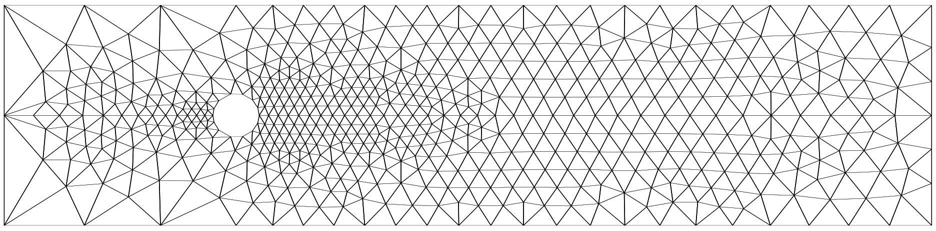


FIGURE III.1: Example of finite element meshing.

The functions $\varphi^{\mathbf{u}}_k$ which intervene in this approximation are defined in Ω totally and have $N^{\mathbf{u}}_{\mathrm{c}}$ components such that a single one of them is non null, according to the following relationship:

$$\varphi^{\mathbf{u}}_{n\cdot N^{\mathbf{u}}_{\mathrm{c}}}(\mathbf{x}) = \begin{bmatrix} \mathsf{N}^{\mathbf{u}}_n(\mathbf{x}) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \varphi^{\mathbf{u}}_{n\cdot N^{\mathbf{u}}_{\mathrm{c}}+1}(\mathbf{x}) = \begin{bmatrix} 0 \\ \mathsf{N}^{\mathbf{u}}_i(\mathbf{x}) \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \varphi^{\mathbf{u}}_{n\cdot N^{\mathbf{u}}_{\mathrm{c}}+N^{\mathbf{u}}_{\mathrm{c}}-1}(\mathbf{x}) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \mathsf{N}^{\mathbf{u}}_n(\mathbf{x}) \end{bmatrix} \quad \forall \mathbf{x} \in \Omega$$

The $\mathsf{N}^{\mathbf{u}}_n$ $(n = 1, \ldots, N)$ form a family of functions (called scalar basis functions) which generate a space of finite dimension scalar functions.

The relationship (3.4) can therefore be written as follows:

$$\begin{aligned} &\mathbf{u}_h(\mathbf{x}) = \sum_{0\le n<N^{\mathbf{u}}_{\mathrm{node}}} \; \sum_{0\le i_c<N^{\mathbf{u}}_{\mathrm{c}}} u_{n,i_c}\, \mathsf{N}^{\mathbf{u}}_n(\mathbf{x})\, \mathbf{e}_{i_c} \quad \forall \mathbf{x} \in \Omega \\ &\mathbf{e}_{i_c} = \text{vector with } N^{\mathbf{u}}_{\mathrm{c}} \text{ components such that } \mathbf{e}_{i_c,j} = \delta_{i_c j} \end{aligned} \tag{3.5}$$

where i_c designates the component of $\mathbf{u}$ and where n is called *node index*.

▷ The function space $\mathcal{S}^{\mathbf{u}}$ to which $\mathbf{u}$ belongs is of infinite dimension. The relationship (3.5) defines a function $\mathbf{u}_h$ approximating $\mathbf{u}$ to an extent, which belongs to a functional finite dimensional space generated by the $N^{\mathbf{u}}_{\mathrm{dof}}$ functions $\mathbf{x} \mapsto \mathsf{N}^{\mathbf{u}}_n(\mathbf{x})\mathbf{e}_{i_c}$. Thus, the finite element method is naturally interpreted as an "approximation" method of functional spaces:

$$\mathcal{S}^{\mathbf{u}} \quad \text{“approximé” par} \quad \mathcal{S}^{\mathbf{u}}_h = \mathrm{span}\Big\{ \mathsf{N}^{\mathbf{u}}_n\mathbf{e}_{i_c} \;\Big|\; 0 \le n < N^{\mathbf{u}}_{\mathrm{node}} \;;\; 0 \le i_c < N^{\mathbf{u}}_{\mathrm{c}} \Big\}$$

with the terminology "approximation" formalised in the theory of finite elements through estimations of $(\mathbf{u} - \mathbf{u}_h)$ [4].

▷ A scalar base function $\mathsf{N}^{\mathbf{u}}_n$ has the fundamental property of being associated with a *geometric node* $\mathbf{a}^{\mathbf{u}}_n$, of having its support reduced to all the cells containing this node and of being defined by its restriction to each cell $\mathcal{C}_e$ as follows:

$$\forall \mathbf{x} \in \mathcal{C}_e \quad \begin{cases} \text{si } \mathbf{a}^{\mathbf{u}}_n \notin \mathcal{C}_e & \mathsf{N}^{\mathbf{u}}_n(\mathbf{x}) = 0 \\ \text{si } \mathbf{a}^{\mathbf{u}}_n \in \mathcal{C}_e & \mathsf{N}^{\mathbf{u}}_n(\mathbf{x}) = \widehat{\mathsf{N}}^{\mathbf{u}}_i(\widehat{\mathbf{x}}) \quad \text{où} \quad \mathbf{x} = F_e(\widehat{\mathbf{x}}) \quad \text{et} \quad \widehat{\mathbf{x}} \in \widehat{K} \end{cases}$$

In this formula, a random cell $\mathcal{C}_e$ is linked to the reference mesh $\widehat{K}$ *via* the change in coordinates (mapping) $F_e \colon \widehat{K} \to \mathcal{C}_e$. The basis functions $\mathsf{N}^{\mathbf{u}}_n$ with $\mathbf{a}^{\mathbf{u}}_n \in \mathcal{C}_e$ are defined from basis functions in the reference mesh, denoted $\widehat{\mathsf{N}}^{\mathbf{u}}_i$. Thus, to each node $\mathbf{a}^{\mathbf{u}}_n$ of $\mathcal{C}_e$ is associated a node $\widehat{\mathbf{a}}^{\mathbf{u}}_i$ of the reference mesh $\widehat{K}$ such that $\mathbf{a}^{\mathbf{u}}_n = F_e(\widehat{\mathbf{a}}^{\mathbf{u}}_i)$.

▷ Let us denote $N^{\mathbf{u}}$ the number of basis functions defined in the reference mesh $\widehat{K}$. For any mesh $\mathcal{C}_e$, the scalar basis functions non identically null in $\mathcal{C}_e$ are $N^{\mathbf{u}}$ in number. They can be indexed in two different ways:

Global indexing: $\mathbf{N}^{\mathbf{u}}_n : \mathcal{C}_e \to \mathbb{R}$ defined by $\begin{cases} \mathbf{N}^{\mathbf{u}}_n = \widehat{\mathbf{N}}^{\mathbf{u}}_i \circ F_e^{-1} \\ \mathbf{a}^{\mathbf{u}}_n = F_e(\widehat{\mathbf{a}}^{\mathbf{u}}_i) \\ 0 \leq i < N^{\mathbf{u}} \end{cases}$

Local indexing: $\mathbf{N}^{\mathbf{u}}_i : \mathcal{C}_e \to \mathbb{R}$ defined by $\begin{cases} \mathbf{N}^{\mathbf{u}}_i = \widehat{\mathbf{N}}^{\mathbf{u}}_i \circ F_e^{-1} \\ 0 \leq i < N^{\mathbf{u}} \end{cases}$

▷ One of the difficulties in implementing the finite element method relates to the multiple numberings (for example, local, global node index, index of related unknowns, *etc.*):

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{degree of freedom}\\ k\end{array}} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow, "k \in \mathbb{U}^{\mathbf{u}}"] & \boxed{\begin{array}{cc}\text{node} & n\\ \hline \text{component} & i_c\end{array}} \\
\boxed{\begin{array}{c}\text{discrete unknown}\\ I\end{array}} &
\end{tikzcd}
\qquad \varphi^{\mathbf{u}}_I = \varphi^{\mathbf{u}}_k = \mathbf{N}^{\mathbf{u}}_n \mathbf{e}_{i_c} \tag{3.6}$$

$$\boxed{\begin{array}{c}\text{global node}\\ n\end{array}} \longleftarrow \text{maille } K_e \longrightarrow \boxed{\begin{array}{c}\text{local node}\\ i\end{array}} \qquad \mathbf{N}^{\mathbf{u}}_n = \mathbf{N}^{\mathbf{u}}_i$$

This difficulty is managed entirely by PELICANS, so that it does not affect the user adversely.

▷ With the joint data

- of all geometric nodes $\{ \widehat{\mathbf{a}}^{\mathbf{u}}_i \mid 0 \leq i < N^{\mathbf{u}} \}$
- of all reference basis functions $\widehat{P} \stackrel{\text{def}}{=} \{ \widehat{\mathbf{N}}^{\mathbf{u}}_i \mid 0 \leq i < N^{\mathbf{u}} \}$

is associated in the PELICANS architecture an abstract class called **`PDE_ReferenceElement`**.

The set $\widehat{P}$ is often described in terms of polynomial functions of limited degree $\mathbb{P}_k$ when the reference mesh is a simplex (2-dimensional triangle, 3-dimensional tetrahedron) and $\mathbb{Q}_k$ when the reference mesh is a parallelotope (2-dimensional square, 3-dimensional cube):

$$\mathbb{P}_k = \text{span}\Big\{ (x_1, \ldots, x_d) \mapsto x_1^{\alpha_1} \cdots x_d^{\alpha_d} \quad \text{avec} \quad \alpha_i \in \mathbb{N} \quad \text{et} \quad \alpha_1 + \cdots + \alpha_d \leq k \Big\}$$
$$\mathbb{Q}_k = \text{span}\Big\{ (x_1, \ldots, x_d) \mapsto x_1^{\alpha_1} \cdots x_d^{\alpha_d} \quad \text{avec} \quad \alpha_i \in \mathbb{N} \quad \text{et} \quad \alpha_i \leq k \Big\}$$

The class **`PDE_ReferenceElement`**, which is a *framework plug-point* is the standard example of the adaptable software component:

- PELICANS supplies off-the-shelf a set of concrete classes derived from **`PDE_ReferenceElement`** which covers a wide range of elements (Table V.1)(Table V.1);
- where the numeric model brings a reference element into play which is not found in the previous list, the user simply has to derive a concrete class of **`PDE_ReferenceElement`** which implements the specific features of this new element.

Note that the geometric nodes are only used to distinguish the basis functions: two base functions associated with the same reference element are distinct if they are linked to two distinct geometric nodes. In particular, no hypothesis is applied to the value of basis functions at the geometric nodes.

III.5 Discretization of a Set of Fields

▷ In the most general case, the unknown **W** from problem (3.1) gathers several fields with one or more components which depend not only on space but possibly on time. The discrete representation of W is such that:

1. All the fields of **W** defined in a given geometric domain are discretised in the same meshing.
2. Two fields of **W** can be discretised with different finite elements.

▷ The very nature of the discretization is part of the definition of discrete fields. Two unknowns of the continuous problem can be seen as two components of a same field if it is decided to discretise them with the same finite element, otherwise they should be represented by two distinct discrete fields.

Thus, the numeric model influences retroactively the formulation of the mathematical model: for example, the choice of a discretization strategy for fields partly conditions their definition.

III.6 Discrete Problem or Algebraic Equation System

The discrete problem is at the heart of numeric simulation. It is the culmination of approximation techniques used to match the continuous problem with a set of algebraic equations. The terminology used until now has been deliberately simplistic and has basically tried for a clear separation between the numeric simulation activity, producing a discrete problem from an continuous problem, and the computer programming activity, producing a software program from a discrete problem. At this point in our account, we assume that this dichotomy, which is essential to understanding the business concepts of PELICANS, has been acquired and we are now going to refine certain notions.

▷ As much as it is possible to summarise the continuous problem as unknown fields, equations verified by unknown fields and parameters, it is normally impossible to formalise an abstract discrete problem concept (even by choosing a single, *a priori* given technique for discretising fields and equations). The discrete problem (in the singular) is an illusion which conceals a set of algebraic equation systems (in the plural) to be resolved in sequence. This sequence can be associated with a time marching, with iterations associated with non-linearities or with any other context. Certain numeric models of coupled problems can also interleave the sequences one after the other, process each sequence stage like another sequence[1],*etc.* At this level, PELICANS sets no constraints nor numeric context.

▷ It is precisely this notion of *algebraic equation system* and the multiplicity of its occurrences that forms the conceptual pivot between the numeric modelling and the choice of classes to be implemented with their respective responsibilities.

A PELICANS application is object-oriented software and therefore its structure is based on a decompostions into classes. The PELICANS business concepts are used to formulate recommendations to "find the classes"[2] associated with the numeric schemes we are interested in.

- Implement two classes for each of the sequences described previously:
  - the first represents algebraic equation systems which firstly, can be constructed from elementary contributions relating to the geometric entities of a mesh and secondly, can be solved;
  - the second represents objects capable of calculating the elementary contributions mentioned above associated with the continuous problem from its data (for an *a priori* fixed discretization technique) and of updating the discrete representation of unknown fields (we dare to introduce a neologism: a class of discretisers).
- Represent the suite of sequences as an instance of a dedicated class.

[1]this implies here a possible use of the design pattern known as "composite" [6].

[2]generic problem area, developed in detail for example in the reference [14] including one chapter entitled "How to Find the Classes".

▷ The time marching algorithms are so frequent that the PELICANS *framework* brings to the fore a set of abstractions defining an object model of coupled problems discretised over time, where the solution at time t_n is determined from solutions at the times $t_{n-1}, t_{n-2}, \ldots$ At each time step, the unknowns, which at this point depend only on space variables, are expressed as a solution to a suite of problems which may be coupled by explicit or implicit Gauss-Seidel-type strategies.

The related *plug-points* are used to manage the most popular situations. If the numeric model studied brings a temporal semi-discretization into play which does not enter this common framework, the user can programme his own dedicated classes from low-level components, themselves used by the PELICANS time marching framework classes.

III.7 Discretization of PDEs in a Finite Element Meshing

▷ Many numerical methods exists assigned to the discretization of PDEs systems where the unknowns are represented discretely in a finite element mesh, for example, and in disorder: the finite volume method, the Galerkin, Galerkin Least-Squares, Stabilised Galerkin, Petrov-Galerkin, Taylor-Galerkin and Galerkin-Characteristics methods and no doubt other methods about which the author (once more) professes his ignorance. The fundamental pragmatic observation - the basis for the PELICANS design - is as follows: the extent of essential functionalities required to implement related numeric models is clearly less than is suggested by their multiplicity.

▷ The general numeric discretization framework for PDEs in a finite element meshing involving PELICANS is that of the *variational approximation* and *finite volumes*. All the methods mentioned above can be interpreted - or re-interpreted - in these terms. In fact, on reaching the implementation phase, the theoretical justifications of the numeric model have little bearing, only the constraints placed on the programming count. The main thing for the implementation phase is that it can be written *formally* as a variational approximation or finite volume method and that this formal writing produces the same discrete problem as the one produced by the original mathematical definition. We can go even further, and why deny ourselves? Every numeric scheme where implementation requires the same elementary functionalities as a variational approximation or finite volume method can be implemented with PELICANS, whether or not such a method is involved. The example of the family of “Fluctuation Splitting” or “Residual Distribution” schemes illustrates this point [2].

▷ The responsibility for services offered for PDEs discretization in finite element meshings is allocated to a small number of classes (less than ten) which form the (adaptable) components of the facade through which the user can take full advantage of PELICANS. The description of these classes goes beyond the scope of this document (refer to on-line manuals). The functionalities they offer have two objectives: immediate programming of a certain number of actions and access to some fundamental information.

Let us consider a few essential actions.

- Iterate over the cells the meshing.
- Iterate over the external faces of the meshing.
- Iterate over the internal faces of the meshing.
- In a given mesh (cell or face) denoted K, calculate a numeric approximation of an integral using a formula such as:

$$\int_K f \approx \sum_k w_k f(\mathbf{x}_k)$$

where the pairs $(\mathbf{x}_k, w_k)$, with $\mathbf{x}_k \in K$ and $w_k \in \mathbb{R}$, define a quadrature rule in K. In PELICANS, these notions are associated with two abstract classes which are *framework plug-points*, typical examples of adaptable software components: concrete classes derived from these *plug-points* are available off-the-shelf so that the user can access a range of integration methods covering the most common needs (and even a little more). If however the numeric model brings a quadrature rule into play that is not pre-defined, the associated concrete classes simply have to be derived to acquire its own specific features. The induced flexibility is precious considering that a number of numeric strategies can be interpreted as a modification to the integration method [18, 17, 9, 4].

Note that the quadrature rules are normally defined in the reference mesh, so that the computation of the w_k and $\mathbf{x}_k$, on the actual mesh, require the intervention of the change of coordinates between these two meshes. This technical machinery is managed entirely by PELICANS, which takes the user through the related difficulties (which are encapsulated).

- Calculate the value, at a random point in a mesh, of the finite element reconstruction relating to the discrete representation of a random field.
- Manage a discrete system local to a given mesh (cell or face).
- Assemble the coefficients of a local discrete system in the global discrete system by changing the numbering and eliminating the degrees of freedom with an imposed value.

Let us now move on to some essential information.

- Geometric characteristics of a given mesh (cell or face): vertex coordinates, measure, centre, diameter, roundness, *etc.*
- For each field, link between the local numbering in a given mesh of the base functions and the global numbering of its degrees of freedom.
- In a given mesh (cell or face) denoted K and for the finite element reconstruction of the discrete representation (3.5) of a given field $\mathbf{u}$:

$$\mathbf{u}_h(\mathbf{x}) = \sum_{0 \le n < N^{\mathbf{u}}_{\text{node}}} \ \sum_{0 \le i_c < N^{\mathbf{u}}_{\text{c}}} u_{n,i_c} \, \mathbf{N}^{\mathbf{u}}_n(\mathbf{x}) \, \mathbf{e}_{i_c}$$

  set $\mathbb{I}_K$ of indices n such that $\mathbf{N}^{\mathbf{u}}_n$ is non identically zero in K. For each of these indices, value of degrees of freedom u_{n,i_c}.
- With the notations of the previous point, value of:

$$\mathbf{N}^{\mathbf{u}}_n(\mathbf{x}) \ , \quad \nabla\mathbf{N}^{\mathbf{u}}_n(\mathbf{x}) \ , \quad \nabla\nabla\mathbf{N}^{\mathbf{u}}_n(\mathbf{x}) \quad \text{pour} \quad i \in \mathbb{I}_{\mathcal{D}} \quad \text{et} \quad \mathbf{x} \in \mathcal{D}$$

  As the finite element base functions are defined in the reference mesh, calculating derivatives requires a certain number of algebraic operations associated with the change of coordinates between the reference mesh and the actual mesh.

  This technical machinery is managed entirely by PELICANS, which takes the user through the related difficulties (which are encapsulated).

In practice, the actions and information listed below are provided by the member functions of a few dedicated classes.

III.8 Algebraic Resolutions

The discrete problem generated by the numeric modelling requires the resolution of algebraic equations to which PELICANS devotes a set of software components, some being *framework plug-points*.

▷ The matrix systems produced by discretization of PDEs in meshings through “local” methods have the essential characteristic of being large with a sparse structure: only a few coefficients are non zero on each line.

In addition to "natural" linear algebraic classes[3], PELICANS defines abstract data types relating to the sparse matrices and linear solvers of sparse systems. The fundamental specific aspects are described below.

- The data structures are such that:
  1. only the non zero coefficients are stored;
  2. the usual operations are implemented efficiently.
- All the interfaces have been implemented by default, but some of them form *framework plug-points*. Thus, the PELICANS applications can be coupled with external libraries with minimum effort, by creating classes derived from these plug-points which have the role of adapting the interfaces (which constitutes an example of the design pattern called "adapter" [6]).
- The elementary operations are BLAS (Basic Linear Algebra Subprograms [7]) which define a set of low-level interfaces implemented in optimised fashion in numerous hardware platforms (PELICANS provides portable implementation for the rest). Table III.1 lists a few of these elementary operations.

▷ Some well-known algorithms for solving large, sparse linear systems are based on pre-conditioned Krylov methods [20, 8]. Two adaptive components are associated with them: one representing the iterative solvers and the other the preconditionners.

Three use levels are therefore possible.

1. Concrete classes derived from these *plug-points* are implemented by PELICANS so that the user can access a range of solvers and preconditionners delivered by PELICANS.
2. If the user wishes to implement his own solver and his own preconditionner, he simply has to programme the related concrete classes derived from *plug-points*, by using the available elementary functionalities such as BLAS.
3. The *plug-points* can serve as adapters for the external libraries so that their interface conforms with the PELICANS requirements (in this case the design pattern called "adapter" is used [6]).

An adaptation of this type has already been performed to provide access to several outside libraries (Table V.1).

▷ The iterative methods are not the only ones suitable for resolving discrete problems arising from discretization of PDEs. PELICANS does not impose their use and leaves open the use of other approaches such as the direct methods (although no PELICANS application has explored this path to date to our knowledge). ??

III.9 Inputs-Outputs

▷ PELICANS is a PDEs solver implementation tool. It concentrates only on the generation and discrete problem resolution phase and does not (or barely) include a pre-processing (meshing generator) and post-processing (graphic display) tool.

PELICANS applications must therefore rely on external tools to make up a complete system including pre- and post-processing.

[3]The term "natural classes" is employed here to signify "that the reader will imagine naturally", but in no way reflects the reality of the object oriented programming: as much as it is possible to obtain a consensus on the algebraic definition of a matrix and the link between all square matrices and all rectangular matrices for example, it is unrealistic to hope to associate these mathematical notions with a single set of so-called "natural" classes. An object model is an abstraction of the reality (here the mathematical theory of the linear algebra) which underlines certain details whilst leaving others aside, depending on how much light you wish to throw on this reality. This depends on the behaviour sought for the class objects to be designed. Thus, it is theoretically impossible to answer the following question: does an inheritance relationship exist between a class of square matrices and a class of rectangular matrices and if so, which one inherits from the other? Regardless of the answer, it is both true and false: everything depends on the behaviour sought. Any inheritance relationship is determined from this behaviour and by applying the Liskov Substitution Principle [12].

BLAS niveau 1 opérations vecteur-vecteur	BLAS niveau 2 opérations matrice-vecteur	BLAS niveau 3 opérations matrice-matrice
$x \leftrightarrow y$	$x \leftarrow \alpha A x + \beta y$	$C \leftarrow \alpha A B + \beta C$
$x \leftarrow \alpha x$	$x \leftarrow \alpha A^{\mathrm{T}} x + \beta y$	$C \leftarrow \alpha A^{\mathrm{T}} B + \beta C$
$y \leftarrow x$	$A \leftarrow \alpha x y^{\mathrm{T}} + A$	$C \leftarrow \alpha A B^{\mathrm{T}} + \beta C$
$y \leftarrow \alpha x + y$		$C \leftarrow \alpha A^{\mathrm{T}} B^{\mathrm{T}} + \beta C$
$\mathrm{dot} \leftarrow x^{\mathrm{T}} y$		

TABLE III.1: Elementary algebraic operations defined by BLAS [7] and integrated with the PELICANS component interfaces (Greek letters designate scalars, small Latin letters vectors and capital Latin letters matrices).

▷ A certain number of tools, all free, exist which have already been interfaced with PELICANS (Table V.1). In addition, PELICANS includes a simplified (and simple) meshing generation tool for domains with normal borders at the coordinate system axes (like the parallelepipeds).

▷ Two adaptable components are devoted to interfacing the PELICANS applications with the pre- and post-processing tools. They constitute two *framework plug-points* which are used to convert the services rendered by meshing generators or external viewers to comply with the interfaces required by PELICANS (the design pattern called "adapter" is used here [6]).

III.10 Advanced Functionalities

We have described above the essential concepts that are the foundation of the PELICANS architecture and structure the functionalities relating to PDEs numeric resolution.

Some points outside the scope of this work have not been addressed. We quote them below without expanding on them.

III.10.1 Characteristics Method

Support for the characteristics method is offered in the finite element meshings, to be able to discretise the special derivatives as follows (immediate notations):

$$\left\{\frac{\partial u}{\partial t} + \mathbf{a}\cdot\nabla u\right\}(\mathbf{x}, t_{n+1}) \simeq \frac{u(\mathbf{x}, t_{n+1}) - u\big(\mathbf{X}(\mathbf{x}, t_{n+1}; t_n), t_n\big)}{\Delta t}$$

$$\text{où} \quad \left\{ \begin{array}{l} \dfrac{d\mathbf{X}(\mathbf{x}, t_{n+1}; \tau)}{d\tau} = \mathbf{a}\big[\mathbf{X}(\mathbf{x}, t_{n+1}; \tau)\big] \\ \mathbf{X}(\mathbf{x}, t_{n+1}; t_{n+1}) = \mathbf{x} \quad \text{final condition} \end{array} \right.$$

III.10.2 Deformable Meshes

The finite element meshes can be deformed to allow implementation of ALE (Adaptative Lagrange Euler) methods, particularly for interfaces tracking.

III.10.3 Free Surface Immersed in a Fixed Meshing

For two-dimensional flows characterised by major movements of a free surface, PELICANS allows a variant of the Eulerian finite element method to be implemented; here a mobile polygonal boundary is immersed in a fixed meshing [11, 3]. There are three distinct mesh families in this method: full meshes, empty meshes and meshes containing the free surface. In the latter, the unknowns are represented by the restriction to the fluid domain of the usual linear combination of shape functions of the Eulerian finite element mesh, which defines fictitious degrees of freedom at the nodes located outside the fluid domain. The balance equations are thus discretised by a Galerkin-Characteristics method where the variational formulation is stated in terms of integrals over the fluid domain at each time step. Thus, the assembly process is essentially similar to that of an ordinary finite element method, apart from the fact that the numeric integrations are performed at the intersection of each mesh with the fluid domain. The effort of designing adaptive software components for the supply of quadrature rules is here richly rewarded.

III.10.4 Multi-Physic, Multi-Domain Application

PELICANS can be used to manage PDEs systems stated in distinct, adjacent geometric domains triangulated by finite element meshings which share part of their boundary not necessarily in conforming fashion. The discrete problems established in each domain are coupled via mixed boundary conditions or by the mortar finite element technique [19].

III.10.5 Parallelism

III.10.6 Multilevel Methods

III.10.7 Adaptive Local Refinement

Chapter IV

PELICANS Use Example

Suppose we wanted to develop a software program to calculate the stationary state of the temperature field in a flat plate subjected to miscellaneous thermal loadings on its walls.

The physical system represented by IV.1 is considered especially. A flat plane with conductivity $\lambda = 236\,\mathrm{W/m/K}$ is subjected to a uniform heat flux of $50\,\mathrm{kW/m^2}$ on its western face, whilst its southern and eastern faces are insulated thermally. In addition, its northern face is maintained at a temperature of $100\,°\mathrm{C}$. The idea is to determine the temperature at all points at the stationary state.

IV.1 Prerequisite to using PELICANS

Before using PELICANS to develop software, it is important to establish a mathematical model and to choose a numeric method that suits it.

IV.1.1 Mathematical Model

The physical situation in question can be encompassed in the following mathematical model.

Let Ω be an open polyhedral domain of $\mathbb{R}^d$. The boundary of Ω is denoted Γ and the unit [—]

- Champ inconnu :

$$T : \Omega \to \mathbb{R}$$

- Équation vérifiée par le champ inconnu :

$$\begin{cases} \nabla \cdot (-\lambda \nabla T) = 0 & \text{dans } \Omega \\ T = T_0 & \text{sur } \Gamma_{\mathrm{D}} \\ \lambda \nabla T \cdot \mathbf{n} = q_0 & \text{sur } \Gamma_{\mathrm{N}} \end{cases} \tag{4.1}$$

- Paramètres du modèle : T_{D} et q_0

We are going to use two numeric methods to resolve this problem: a Galerkin method and a finite volume method.

IV.1.2 Modèle Numérique : Méthode de Galerkin Éléments Finis

▷ On suppose ici que le champ inconnu T est discrétisé par la méthode des éléments finis.

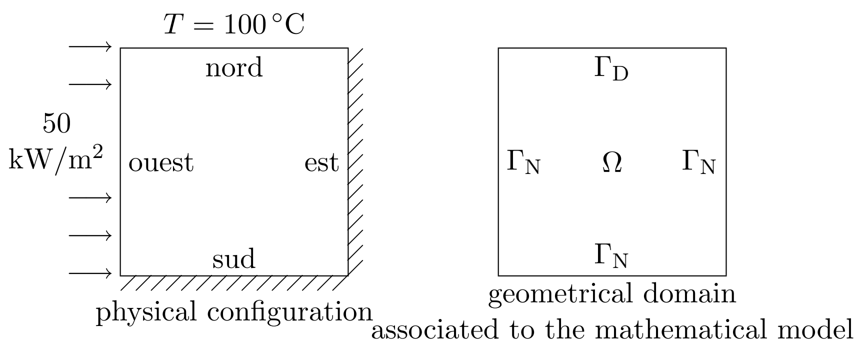


FIGURE IV.1: physical environment and geometry that the simulation software must handle.

▷ La formulation variationnelle du problème (4.1) s'écrit :

$$\text{Trouver } T \in \mathcal{S}^T \text{ tq } \forall v \in \mathcal{V}^T \qquad \int_\Omega \lambda \nabla v \cdot \nabla T = \int_{\Gamma_\mathrm{N}} q_0 v$$

où les espaces fonctionnels $\mathcal{S}^T$ et $\mathcal{V}^T$ sont définis comme suit :

$$\mathcal{S}^T = \left\{ v \in \mathrm{H}^1(\Omega) \;\middle|\; v = T_0 \text{ sur } \Gamma_\mathrm{D} \right\}$$
$$\mathcal{V}^T = \left\{ v \in \mathrm{H}^1(\Omega) \;\middle|\; v = 0 \text{ sur } \Gamma_\mathrm{D} \right\}$$

▷ Supposons que $T_0 \in \mathrm{H}^{1/2}(\Gamma_\mathrm{D})$. Alors il existe $T_\mathrm{D} \in \mathrm{H}^1(\Omega)$ tel que $T_\mathrm{D}|_{\Gamma_\mathrm{D}} = u_0$ (au sens d'un théorème de trace), ce qui va nous permettre d'introduire une nouvelle inconnue u au problème (4.1) définie comme suit :

$$u = T - T_\mathrm{D} \tag{4.2}$$

La nouvelle inconnue u est solution de :

$$\text{Trouver } u \in \mathcal{S}^T \text{ tel que : } \quad a(u,v) = f(v) \quad \forall v \in \mathcal{V}^T \tag{4.3}$$

Les fonctionnelles bilinéaire a et linéaire f s'écrivent :

$$\begin{cases} a(u,v) \overset{\text{def}}{=} \displaystyle\int_\Omega \lambda \nabla v \cdot \nabla u \\ \ell(v) \overset{\text{def}}{=} \displaystyle\int_{\Gamma_\mathrm{N}} q_0 v \\ f(v) \overset{\text{def}}{=} \ell(v) - a(T_\mathrm{D}, v) \end{cases}$$

▷ Introduisons $\mathcal{T}_h$ une triangulation régulière de Ω, composées de cellules $\mathcal{C}_e$ ($\overline{\Omega} = \cup_e \mathcal{C}_e$), et X_h^T une approximation éléments finis de $\mathrm{H}^1(\Omega)$ construite à partir de $\mathcal{T}_h$:

$$X_h^T = \mathrm{span}\{\ \varphi_k^T \ \big|\ 0 \le k < N_\mathrm{dof}^T\ \}$$

Une approximation de Galerkin du problème variationnel (4.3) s'obtient simplement en remplaçant les espaces $\mathcal{S}^T$ et $\mathcal{V}^T$ par des espaces de dimension finie $\mathcal{S}_h^T$ et $\mathcal{V}_h^T$ déduits de X_h^T :

$$\text{Trouver } \ u_h \in \mathcal{V}_h^T \ \text{ tq } \ \forall v_h \in \mathcal{V}_h^T \ : \ a(u_h, v_h) = \ell(v_h) - a(T_{\mathrm{D}h}, v_h)$$

où

$$T_{\mathrm{D}h}(\mathbf{x}) = \sum_{k \in \mathbb{I}^T} T_{\mathrm{D}k}\, \varphi_k^T(\mathbf{x}) \qquad \mathbb{I}^T = \left\{\ k \in [0, N_\mathrm{dof}^T[\ \middle|\ \varphi_k^T|_{\Gamma_\mathrm{D}} \not\equiv 0\ \right\}$$

$$\mathcal{S}_h^T \;=\; T_{\mathrm{D}h} + \mathcal{V}_h^T$$

$$\mathcal{V}_h^T \;=\; \mathrm{span}\{\ \varphi_k^T \ \big|\ 0 \le k < N_\mathrm{dof}^T \ \ k \notin \mathbb{I}^T\ \} \;=\; \mathrm{span}\{\ \varphi_I^T \ \big|\ 0 \le I < N_\mathrm{unk}^T\ \}$$

▷ Le problème discret peut alors être reformulé sous forme matricielle comme suit :

$$\text{Trouver } \mathbf{U} \in \mathbb{R}^{N^T_{\text{unk}}} \quad \text{tel que :} \quad \mathbf{A} \cdot \mathbf{U} = \mathbf{F} \tag{4.4}$$

où

$$\begin{aligned}
\mathbf{A}_{IJ} &= a(\varphi^T_J, \varphi^T_I) && 0 \le I, J < N^T_{\text{unk}} \\
\mathbf{F}_I &= \ell(\varphi^T_I) - a(T_{\mathrm{D}h}, \varphi^T_I) && 0 \le I < N^T_{\text{unk}} \\
T(\mathbf{x}) &\simeq u_h(\mathbf{x}) = \sum_{0 \le I < N^T_{\text{unk}}} \mathbf{U}_I \varphi^T_I(\mathbf{x}) + \underbrace{\sum_{k \in \mathbb{I}^T} T_{\mathrm{D}k}\, \varphi^T_k(\mathbf{x})}_{T_{\mathrm{D}h}(\mathbf{x})} && \forall \mathbf{x} \in \Omega
\end{aligned}$$

▷ Le calcul des éléments de la matrice **A** et du vecteur **F** repose sur une décomposition des intégrales de volume et de surface en sommes d'intégrales, respectivement sur les cellules et sur les faces externes du maillage :

$$\begin{aligned}
\mathbf{A}_{IJ} &= \int_\Omega \lambda \nabla\varphi^T_J \cdot \nabla\varphi^T_I &&= \sum_{\mathcal{C}_e} \int_{\mathcal{C}_e} \lambda \nabla\varphi^T_J \cdot \nabla\varphi^T_I \\
\mathbf{F}_I &= \int_{\Gamma_\mathrm{N}} q_0\, \varphi^T_I - a(T_{\mathrm{D}h}, \varphi^T_I) &&= \sum_{\mathcal{F}_e \subset \Gamma_\mathrm{N}} \int_{\mathcal{F}_e} q_0\, \varphi^T_I - a(T_{\mathrm{D}h}, \varphi^T_I) \\
a(T_{\mathrm{D}h}, \varphi^T_I) &= \sum_{k \in \mathbb{I}^T} \underbrace{\left\{ \int_\Omega \lambda \nabla\varphi^T_k \cdot \nabla\varphi^T_I \right\}}_{\text{ressemble à ``}\mathbf{A}_{Ik}\text{''}} T_{\mathrm{D}k}
\end{aligned}$$

En utilisant la localité des fonctions de base éléments finis ainsi que leur indexation locale (3.6), on construit le système algébrique (4.4) par ajouts successifs des contributions de chaque maille :

$$\begin{bmatrix} \mathbf{K} \\ \mathbf{F} \end{bmatrix} = \sum_{e\,:\,\text{cellules}} \mathsf{A}^{\text{mat}}_{\text{vec}} \begin{bmatrix} a^e \\ 0 \end{bmatrix} + \sum_{\substack{e\,:\,\text{faces} \\ \text{de } \Gamma_\mathrm{N}}} \mathsf{A}^{\text{mat}}_{\text{vec}} \begin{bmatrix} 0 \\ \ell^e \end{bmatrix} \tag{4.5}$$

où :

- Les contributions élémentaires sont données par :

$$\begin{aligned}
& a^e = \left[a^e(i,j) \right] \qquad \ell^e = \left[\ell^e(i) \right] \qquad 0 \le i, j < N^u_e \\
& a^e(i,j) = \int_{\mathcal{C}_e} \lambda\, \nabla\varphi^T_J \cdot \nabla\varphi^T_I = \int_{\mathcal{C}_e} \kappa\, \nabla\mathsf{N}^T_j \cdot \nabla\mathsf{N}^T_i \\
& \ell^e(i) = \int_{\mathcal{F}_e} q_0\, \varphi^T_I = \int_{\mathcal{F}_e} q_0\, \mathsf{N}^T_i \qquad \mathcal{F}_e \subset \Gamma_\mathrm{N}
\end{aligned} \tag{4.6}$$

- $\mathsf{A}^{\text{mat}}_{\text{vec}}$ est l'opérateur d'assemblage qui ajoute les contributions élémentaires à l'endroit approprié dans la matrice **A** et le vecteur **F**, en prenant soin d'éliminer, dans les inconnues du système global, les degrés de liberté d'indice appartenant $\mathbb{U}^T$ et de reporter les contributions associées au second membre :

$$\begin{bmatrix}\mathbf{A}\\\mathbf{F}\end{bmatrix} = \sum_{\text{mailles } K_e} \mathsf{A}^{\text{mat}}_{\text{vec}} \begin{bmatrix}a^e\\\ell^e\end{bmatrix} \qquad \begin{array}{r}\text{sur une maille } K_e \text{ donnée}\\ \text{(face ou cellule) :}\end{array} \left\{ \begin{array}{l} \text{Pour chaque } \mathsf{N}^T_i \text{ tq } i \notin \mathbb{I}^T \\ \quad \mathbf{F}_I = \mathbf{F}_I + \ell^e(\,i\,) \\ \quad \text{Pour chaque fonction de base } \mathsf{N}^T_j \\ \qquad \text{si } j \notin \mathbb{I}^T \text{ alors} \\ \qquad\quad \mathbf{A}_{IJ} = \mathbf{A}_{IJ} + a^e(\,i,j\,) \\ \qquad \text{sinon} \\ \qquad\quad \mathbf{F}_I = \mathbf{F}_I - a^e(\,i,j\,)\, T_{\text{D}j} \end{array} \right. \tag{4.7}$$

IV.1.3 Modèle Numérique : Méthode des Volumes Finis

▷ On considère ici un maillage partitionnant l'adhérence $\overline{\Omega}$ de Ω en un nombre finis de cellules, et une discrétization du champ inconnu T mettant en jeux exactement un nœud par cellule (ce qui définit autant de degrés de liberté que de nœuds puisque T n'a qu'une composante).

On suppose qu'il est possible d'associer un point géométrique $\mathbf{x}_K$ à chaque cellule K de telle façon que l'ensemble de ces points vérifie des propriétés agréables à l'aune de la méthode des volumes finis [**?**].

La valeur affectée au degré de liberté d'un nœud donné, associé à une cellule K, sera interprétée comme une approximation de la température au point $\mathbf{x}_K$ et sera notée T_K.

▷ La formulation intégrale du problème (4.1) sur chaque cellule s'écrit :

$$\left\{ \begin{array}{ll} \displaystyle\int_{\partial\mathcal{C}_e} -\lambda\nabla T\cdot\mathbf{n} = 0 & \\ T = T_{\text{D}} & \text{sur } \Gamma_{\text{D}} \\ \lambda\nabla T\cdot\mathbf{n} = q_0 & \text{sur } \Gamma_{\text{N}} \end{array} \right. \tag{4.8}$$

où $\mathbf{n}$ désigne le vecteur unitaire normal à $\partial\mathcal{C}_e$ orienté vers l'extérieur de $\mathcal{C}_e$.

▷ L'intégrale surfacique sur la frontière $\partial\mathcal{C}$ d'une cellule $\mathcal{C}$ donnée peut être développée en une somme d'intégrales sur les faces de $\mathcal{C}$.

Considérons donc l'une des faces $\mathcal{F}$ de $\mathcal{C}$.

En fonction de la localisation géométrique de la face $\mathcal{F}$, l'approximation de l'intégrale de surface :

$$\int_{\mathcal{F}} -\lambda\nabla T\cdot\mathbf{n} \tag{4.9}$$

est donnée ci-dessous ($|\mathcal{F}|$ désigne la surface de $\mathcal{F}$).

- Si $\mathcal{F}$ est une face interne :

$$\int_{\mathcal{F}} -\lambda\nabla T\cdot\mathbf{n} \approx -|\mathcal{F}|\,\lambda\,\frac{T_L - T_K}{d_{KL}} \quad \text{avec} \quad d_{KL} \text{ distance entre } \mathbf{x}_K \text{ et } \mathbf{x}_L \tag{4.10}$$

  où K et L désignent les deux cellules adjacentes à $\mathcal{F}$ discernées de façon que la normale $\mathbf{n}$ soit dirigée de K vers L.

- Si $\mathcal{F}$ est une face frontière située sur Γ_{D} :

$$\int_{\mathcal{F}} -\lambda\nabla T\cdot\mathbf{n} \approx -|\mathcal{F}|\,\lambda\,\frac{T_{\text{D}} - T_K}{d_{K\Gamma}} \quad \text{avec} \quad d_{K\Gamma} \text{ distance entre } \mathbf{x}_K \text{ et } \Gamma_{\text{D}} \tag{4.11}$$

  où K désigne la cellule adjacente à $\mathcal{F}$ et où $\mathbf{n}$ est orientée vers l'extérieur de K.

- Si $\mathcal{F}$ est une face frontière située sur $\Gamma_{\rm N}$:

$$\int_{\mathcal{F}} -\lambda \nabla T \cdot \mathbf{n} \approx -|\mathcal{F}|\, q_0 \tag{4.12}$$

  où $\mathbf{n}$ est orientée vers l'extérieur de Ω.

▷ Les équations intégrales (4.8) sur toutes les cellules $\mathcal{C}_e$, dans lesquelles les intégrales surfaciques sur toutes les faces sont approximées comme décrit précédemment, conduisent à un système linéaire :

$$\mathbf{A} \cdot \mathbf{U} = \mathbf{F}$$

où l'inconnue $\mathbf{U}$ est le vecteur des valeurs des degrés de liberté de température.

La matrice $\mathbf{A}$ et le vecteur $\mathbf{F}$ sont obtenus par assemblage des intégrales surfaciques sur les faces internes et les faces-fontières.

IV.2 Software Development

Once the prerequisites to the use of PELICANS have been formalised, in other words the mathematical and numeric models have been listed, PELICANS may be used to programme the required software.

Two stages are then necessary:

1. design stage, where certain classes abstracted from PELICANS - called *plug-points* - are identified, from which the concrete classes specific to the standard application are going to be derived;
2. implementation stage, in which these concrete classes are implemented using PELICANS components.

IV.2.1 Design

We are going to create two classes derived from **`PEL_Application`**, one called **`HC_GalerkinFE`**, assigned to resolving the problem (4.1) by the finite element Galerkin method and the other called **`HC_FiniteVolume`**, assigned to resolving the problem (4.1) by the finite volume method.

These two classes derive from **`PEL_Application`**, as the class **`HC_Greeting`** described in chapter **??**. The *plug-in* mechanism is identical, as shown in Listings IV.6, IV.7, IV.13, IV.14.

IV.2.2 Implementation of the Galerkin Finite Element Method

The purpose of this section is to describe the source code of the class **`HC_GalerkinFE`** and the format of the dataset it defines.

Header File

There are two stages to resolving the problem using the finite element Galerkin method: constructing the discrete system (4.4) and resolving it.

▷ The unknown vector of the discrete system $\mathbf{U}$ contains the value of degrees of freedom of the temperature which are not associated to nodes belonging to the boundary $\Gamma_{\rm D}$. The degrees of freedom (**`DOF`** for short) of the temperature are represented by an object of the class **`PDE_DiscreteField`**ÿ referenced by the pointer **`TT`**; the correspondence between these **`DOF`**s and the components of unknown vector $\mathbf{T}$ (4.5) is managed by an object of the class **`PDE_LinkDOF2Unknown`**, referenced by the pointer **`TT_link`** (lines 40 and 41 of Listing IV.6).

▷ The continuous problem contains two types of boundary condition.

The Dirichlet boundary conditions, in the part $\Gamma_{\rm D}$ of Γ, induce for the discrete representation of T, in other words the object referenced by **`TT`**, **`DOF`**s with an imposed value. The object referenced by **TT_link** will be used to eliminate these **DOF**s of unknowns from the discrete system during assembly. Thus, the Dirichlet boundary conditions of the continuous problem, which are conveyed when PELICANS is implemented by imposed value DOF conditions, are managed automatically during assembly.

The boundary conditions in the part Γ_N of Γ must be managed explicitly by **HC_GalerkinFE** as they intervene in the discrete system (4.5) *via* the elementary contribution ℓ^e which must be calculated then assembled. To achieve this, an object of class **PDE_SetOfBCs**, referenced by the attribute **BCs** (line 43 of Listing IV.6), is required to identify the boundary faces belonging to Γ_N then to access for each of them all the information relating to the boundary conditions considered.

▷ The matrix **A** and the second member **F** intervening in the discrete system (4.4) are constructed step by step during iterations over the elementary geometric entities of the meshing explained by the numeric model: the cells and the boundary faces in the case we are interested in here. The elementary contributions a^e and ℓ^e are expressed as integrals, over the geometric entity considered, where the integrand brings the scalar base functions into play in local numbering. These integrals are calculated using numeric quadrature, so that the calculation of components of a^e and ℓ^e require, for a given mesh, the value of these basis functions and their derivatives at the chosen integration points. These functionalities are made available by two objects of classes **PDE_LocalFEcell** and **PDE_LocalFEbound**, referenced by the pointers **cFE** and **bFE** (lines 46 and 47 of Listing IV.6).

In a given mesh, the components of a^e and ℓ^e will all be calculated then temporarily stored by an object of the class **PDE_LocalEquation**, referenced by the pointer **ELEMENT_EQ** (line 45 of Listing IV.6), before being assembled in matrix **A** and vector **F**.

```
 1 class HC_GalerkinFE : public PEL_Application
 2 {
 3    public: //-----------------------------------------
 4
 5    //- Program core execution
 6
 7       virtual void run( void ) ;
 8
 9    protected: //--------------------------------------
10
11    private: //----------------------------------------
12
13      ~HC_GalerkinFE( void ) ;
14       HC_GalerkinFE( HC_GalerkinFE const& other ) ;
15       HC_GalerkinFE& operator=( HC_GalerkinFE const& other ) ;
16
17       HC_GalerkinFE( PEL_Object* a_owner,
18                      PEL_ModuleExplorer const* exp ) ;
19
20    //- Plug in
21
22       HC_GalerkinFE( void ) ;
23
24       virtual HC_GalerkinFE* create_replica(
25                              PEL_Object* a_owner,
26                              PEL_ModuleExplorer const* exp ) const ;
27
28    //- Discrete problem building
29
30       void loop_on_cells( void ) const ;
31
32       void loop_on_bounds( void ) const ;
33
34    //- Class attributes
35
36       static HC_GalerkinFE const* PROTOTYPE ;
37
38    //- Attributes
39
40       PDE_DiscreteField* TT ;
41       PDE_LinkDOF2Unknown* TT_link ;
42       double CONDUCTIVITY ;
43       PDE_SetOfBCs const* BCs ;
44
45       PDE_LocalEquation* ELEMENT_EQ ;
46       PDE_LocalFEcell* cFE ;
47       PDE_LocalFEbound* bFE ;
48
49       PDE_ResultSaver* SAVER ;
50
51       PDE_BlockAssembledSystem* GLOBAL_EQ ;
52       PDE_BlockAssembledSystem::LHSindex A ;
53       PDE_BlockAssembledSystem::RHSindex F ;
54       PDE_BlockAssembledSystem::UNKindex X ;
55 } ;
```

LISTING IV.6: extract from the header file of the class **HC_GalerkinFE**.

▷ The global discrete system (4.5) is a linear system; it is constructed by an assembly procedure of systems local to the meshes. In the PELICANS framework and in a finite element context, it must be represented by an object of a class derived from the plug-point **PDE_AssembledSystem**. The discrete problems expressed as block linear systems are so frequent that PELICANS supplies a class likely to represent them off-the-shelf: **PDE_BlockAssembledSystem** (itself a derivative of **PDE_AssembledSystem**) and **HC_GalerkinFE** possesses the attribute **GLOBAL_EQ** referencing an object of this class (line 51 of Listing IV.6).

▷ *Lastly, the results obtained must be saved for the purposes of graphic post-processing, a task entrusted to an object of the class **PDE_ResultSaver** referenced by the attribute **SAVER** (line 49 of Listing

IV.6).

Plug-In Mechanism

▷ The discretization of a PDE system on a meshing brings a set of operations into play, for which PELICANS allocates responsibility to a few dedicated classes like **PDE_DiscreteFields**, **PDE_SetOfBCs**, **PDE_LocalFEcell**, **PDE_LocalFEbound** or **PDE_ResultSaver**. As we have said earlier, **HC_GalerkinFE** has pointer-type attributes on these classes, which must be assigned in the constructor called by **create_replica**. The class **PDE_DomainAndFields** provides access to objects of these classes, initialised in the user's preferred configuration. This configuration is formalised in a hierarchical data structure with a format pre-defined by **PDE_DomainAndFields**, *via* the type and keyword of the expected inputs.

Having said this, let us consider implementation of the constructor:

HC_GalerkinFE(PEL_Object* a_owner, PEL_ModuleExplorer const* exp)

for which the text is given by Listing IV.8.

```
HC_GalerkinFE const* HC_GalerkinFE::PROTOTYPE = new HC_GalerkinFE() ;

HC_GalerkinFE:: HC_GalerkinFE( void )
   : PEL_Application( "HC_GalerkinFE" )
{}

HC_GalerkinFE:: ~HC_GalerkinFE( void )
{}

HC_GalerkinFE*
HC_GalerkinFE:: create_replica( PEL_Object* a_owner,
                                PEL_ModuleExplorer const* exp ) const
{
   PEL_LABEL( "HC_GalerkinFE:: create_replica" ) ;
   PEL_CHECK( create_replica_PRE( a_owner, exp ) ) ;

   HC_GalerkinFE* result = new HC_GalerkinFE( a_owner, exp ) ;

   PEL_CHECK( create_replica_POST( result, a_owner, exp ) ) ;
   return( result ) ;
}
```

Listing IV.7: extract from the **HC_GalerkinFE** class implementation file.

An explorer is created and attached to the **MODULE** named **PDE_DomainAndFields** in the dataset then used *via* the argument **ee** to instantiate an object of the class **PDE_DomainAndFields** (lines 7–9). In its turn, this object is used to assign five pointers **TT**, **cFE**, **bFE**, **BCs** and **SAVER** (lines 12, 17, 21, 24 and 38) so that the state of the five objects they reference is determined by the inputs accessed by the argument **ee**. The other class attributes are assigned to the result of *nstantiation functions* (*factory methods*) of related classes (lines 13, 15 and 31) or by interrogating the dataset directly (line 27).

The objects referenced by **cFE** and **bFE** are used to iterate in the respective meshes (cells and boundary faces) then, for the discretization of every unknown field, provide access to the values of base functions and their derivatives at the integration points. The underlying calculations can be costly: for example, the gradient of a base function is obtained by resolving a linear system of size d, dimension of the geometric space, for each integration point. To reduce the load on CPU resources as much as possible, only strictly necessary calculations are performed. For this, the classes **PDE_LocalFEcell** and **PDE_LocalFEbound** define an operating mode via the member function **require_field_calculation**. Thus, the instructions of lines 18, 19 and 22 mean that only the base function values and their first derivatives in the cells and the base function values in the boundary faces are required.

▷ The common features in the *instantiation functions* of PELICANS classes are as follows:

- Their name starts with **`create`** or **`make`**.
- The object created is *dynamic* and its lifetime is managed by the proprietary method [1]. Thus, the first parameter of an *instantation function* is a pointer towards **`PEL_Object`** designating the owner of the instance created [1]. When this point is **`0`**, the object created must be destroyed explicitly by calling the member function **`destroy`** (as for example line 32). Otherwise, it will be destroyed automatically when its owner is destroyed.

```
 1 HC_GalerkinFE:: HC_GalerkinFE( PEL_Object* a_owner,
 2                               PEL_ModuleExplorer const* exp )
 3    : PEL_Application( a_owner, exp )
 4 {
 5    PEL_LABEL( "HC_GalerkinFE:: HC_GalerkinFE" ) ;
 6
 7    PEL_ModuleExplorer* ee =
 8              exp->create_subexplorer( 0, "PDE_DomainAndFields" ) ;
 9    PDE_DomainAndFields* dom = PDE_DomainAndFields::create( this, ee ) ;
10    ee->destroy() ; ee = 0 ;
11
12    TT = dom->set_of_discrete_fields()->item( "temperature" ) ;
13    TT_link = PDE_LinkDOF2Unknown::create( this, TT, true ) ;
14
15    ELEMENT_EQ = PDE_LocalEquation::create( this ) ;
16
17    cFE = dom->create_LocalFEcell( this ) ;
18    cFE->require_field_calculation( TT, PDE_LocalFE::N ) ;
19    cFE->require_field_calculation( TT, PDE_LocalFE::dN ) ;
20
21    bFE = dom->create_LocalFEbound( this ) ;
22    bFE->require_field_calculation( TT, PDE_LocalFE::N ) ;
23
24    BCs = dom->set_of_boundary_conditions() ;
25
26    ee = exp->create_subexplorer( 0, "HC_GalerkinFE" ) ;
27    CONDUCTIVITY = ee->double_data( "conductivity" ) ;
28
29    PEL_ModuleExplorer* eee =
30              ee->create_subexplorer( ee, "PDE_BlockAssembledSystem" ) ;
31    GLOBAL_EQ = PDE_BlockAssembledSystem::make( this, eee, TT_link, 1 ) ;
32    ee->destroy() ; ee = 0 ;
33
34    A = GLOBAL_EQ->LHS_index( 0, 0, 0 ) ;
35    F = GLOBAL_EQ->RHS_index( 0, 0 ) ;
36    X = GLOBAL_EQ->UNK_index( 0, 0 ) ;
37
38    SAVER = dom->result_saver() ;
39 }
```

Listing IV.8: extract from the **`HC_GalerkinFE`** class implementation file.

- In a good number of cases, the following parameter is a pointer towards an explorer attached to a hierarchical data structure containing the information required to initialise the desired instance, which the instantiation function can interrogate thanks to this explorer.

The *instantiation function* of the class **`PDE_DomainAndFields`**, called line 9 in Listing IV.8, illustrates this point perfectly. Its status of class function explains the call syntax where its name is preceded by the name of the class and the scope operator **`::`** . As the first argument is **`this`**, the object created is owned by the current instance of the class **`HC_GalerkinFE`**. From the reference documentation of the class **`PDE_DomainAndFields`**, we learn that the null value of the third argument means that the problem considered brings a single meshing with the index 0 into play. The second argument is an explorer over the hierarchical database contained in the **`MODULE`** named **`PDE_DomainAndFields`**; we are going to describe succinctly its implications for the objects devoted to the discretization to which **`PDE_DomainAndFields`** gives access.

Datasets

A sample dataset of the application **`"HC_GalerkinFE"`** is given in Listing IV.9. In the remainder of this document we shall consider the **`MODULE`** named **`PDE_DomainAndFields`**.

▷ The geometric domain Ω and its meshing are described in the **`MODULE`** named **`GE_Meshing`**. PELICANS is able to manage meshings from several sources; the argument of the keyword **`concrete_name`** identifies this source. Given the simplicity of the geometry considered here, there is no need to use a sophisticated external tool. The functionalities provided by the PELICANS component **`GE_BoxWithBoxes`**[1] are used to triangulate the domains into boxes (*i.e.* such that the normal vector at the boundary is parallel to one of the axes) by cells which are themselves box-shaped. In this case, we have ten cells in the direction **`0`**, here the x axis, between the abscissa 0.0 m and 0.3 m and ten cells in the direction **`1`**, here the y axis, between the ordinates 0.0 m and 0.4 m.

PELICANS can mark clusters of geometric meshes by allocating them “colors”, with a color basically defined by a string. Thus **`GE_BoxWithBoxes`** marks the faces belonging to the boundary of the geometric domain as **`"left"`**, **`"bottom"`**, **`"right"`** or **`"top"`** depending on their location (the names speak for themselves). This marking is used especially in defining boundary conditions.

▷ The discrete representation of unknown fields in the mathematical model can be defined in the geometric meshing described previously.

The object of the **`MODULE`** name **`interior_fields`** is assumed to contain a succession of **`MODULE`**s describing these discrete representations. In the case of interest to us here, the finite element method is used (**`type="finite_element"`**) and the field T is interpolated in each cell based on an element $\mathbb{Q}_1$ (**`element name="PDE_2D Q1_4nodes"`**[2]). The geometric meshing and the reference element are used to define the discrete representation of T, noted $\mathcal{T} = (T_i)_{0 \leq i < N^T_{\text{dof}}}$ where the index i designates a geometric node (or a scalar base function in equivalent fashion) and defines totally a **`DOF`** (if the unknown field had several components, the component index would have to be added to define a **`DOF`**). The discrete representation of the field T is determined entirely by the values of its **`DOF`**s, which are stored in the object referenced by **`TT`**.

▷ The numeric model considered here does not require an initial value to be given to the **`DOF`**s, but PELICANS imposes one despite everything, hence the presence of the **`MODULE`** named **`DOFs_values`**.

▷ The unknown field T has its value imposed at 100 K by the mathematical model in the part Γ_{D} of the boundary of Ω. The **`MODULE`** named **`DOFs_imposed_value`** passes on this constraint by allocating an imposed value 100.0 (**`type="uniformly_defined"`**, **`value=<100.0>`**) to the **`DOF`**s of $\mathcal{T}$ associated with geometric nodes located at the boundary faces (**`location = "on bounds"`**) of color **`"top"`** (**`color = "top"`**).

[1]`GE_BoxWithBoxes` is a PELICANS class derived from the class `GE_Meshing`, itself a plug-point. The `GE_Meshing` interface formalises what PELICANS understands by geometric meshing. In other words, every class derived from `GE_Meshing` represents a method of defining a meshing; therefore, following this reasoning, interfacing PELICANS with a meshing software consists of creating a class derived from `GE_Meshing` which implements its virtual member functions from outputs from this meshing software. The data of keyword `concrete_name` in the `GE_Meshing` `MODULE` of the data structure is a string naming the concrete class derived from `GE_Meshing` used to define the meshing and the other inputs from this `MODULE` represent the data required to instantiate an object of this class. This type of construction appears frequently in hierarchical data structures, where the words prefixed by one of the PELICANS library identifiers (namely `PEL`, `LA`, `GE`, `PDE` or `FE`) always represent a PELICANS component or *plug-point*.

[2]`PDE_2D_Q1_4nodes` is the name of a class derived from the class `PDE_ReferenceElement`. This is a PELICANS *plug-point*, therefore the native PELICANS functions can be extended by deriving new classes of `PDE_ReferenceElement` and discretising the unknown fields on the basis of newly-created elements by simply changing the data of keyword `"element_name"`.

```
MODULE PEL_Application
   concrete_name = "HC_GalerkinFE"
   MODULE PDE_DomainAndFields
      verbose_level = 1
      nb_space_dimensions = 2
      type = "finite_element"
      MODULE GE_Meshing
         concrete_name = "GE_BoxWithBoxes"
         vertices_coordinate_0 = regular_vector( 0.0, 10, 0.3 )
         vertices_coordinate_1 = regular_vector( 0.0, 10, 0.4 )
         mesh_polyhedron = < "GE_Segment" "GE_Rectangle" >
      END MODULE GE_Meshing
      MODULE interior_fields
         MODULE temperature
            name = "temperature"
            nb_components = 1
            element_name = "PDE_2D_Q1_4nodes"
            storage_depth = 1
            MODULE DOFs_values
               type = "uniformly_defined"
               value = < 0.0 >
            END MODULE DOFs_values
            MODULE DOFs_imposed_value
               MODULE xxx
                  location = "on_bounds"
                  type = "uniformly_defined"
                  color = "top"
                  value = < 100.0 >
               END MODULE xxx
            END MODULE DOFs_imposed_value
         END MODULE temperature
      END MODULE interior_fields
      MODULE boundary_conditions
         MODULE xxx
            field = "temperature"
            color = "left"
            type = "imposed_flux"
            flux_value = 50.E+3
         END MODULE xxx
      END MODULE boundary_conditions
      MODULE PDE_ResultSaver
         ...
      END MODULE PDE_ResultSaver
   END MODULE PDE_DomainAndFields
   MODULE HC_GalerkinFE
      conductivity = 236.
      MODULE PDE_BlockAssembledSystem
         concrete_name = "PDE_BlockAssembledSystemLA"
         ...
      END MODULE PDE_BlockAssembledSystem
   END MODULE HC_GalerkinFE
END MODULE PEL_Application
```

Listing IV.9: **`data.pel`** file associated with the class **`HC_GalerkinFE`**.

Member Function run

The member function **`run`** (Listing IV.10) performs the operations specific to the application considered here. The various stages of the numeric model are found.

1. Construction of the discrete problem (lines 5 and 7), by calling functions **`loop_on_cells`** and **`loop_on_bounds`**, with the respective responsibilities described schematically as follows:

$$\begin{bmatrix}\mathbf{A}\\ \mathbf{F}\end{bmatrix} = \underbrace{\sum_{e\,:\,\text{cellules}} \mathsf{A}^{\text{mat}}_{\text{vec}} \begin{bmatrix}a^e\\ 0\end{bmatrix}}_{\texttt{loop\_on\_cells}} + \underbrace{\sum_{\substack{e\,:\,\text{faces}\\ \text{de }\Gamma_{\text{N}}}} \mathsf{A}^{\text{vec}} \begin{bmatrix}0\\ \ell^e\end{bmatrix}}_{\texttt{loop\_on\_bounds}}$$

2. Resolution of the discrete system (line 9) with possible emission of a fatal error message if the object referenced by **GLOBAL_EQ** has been incapable of accomplishing this task (lines 11 and 12).
3. Updating of the unknown field from the solution to the discrete problem (line 14). The services of the object of type **PDE_LinkDOF2Unknown** referenced by the attribute **TT_link** are required to connect the indexing of **DOF**s of the discrete temperature representation and the indexing of the components of the discrete system solution.
4. Saving for graphic post-processing purposes (lines 16–19).

```
 1 void HC_GalerkinFE:: run( void )
 2 {
 3    PEL_LABEL( "HC_GalerkinFE:: run" ) ;
 4
 5    loop_on_cells() ;
 6
 7    loop_on_bounds() ;
 8
 9    GLOBAL_EQ->estimate_unknown( X, A, F, true ) ;
10
11    if( !GLOBAL_EQ->unknown_is_solution( X ) )
12       PEL_Error::object()->raise_plain( "convergence failure" ) ;
13
14    TT->update_free_DOFs_value( 0, GLOBAL_EQ->unknown_vector( X ), TT_link ) ;
15
16    SAVER->start_cycle() ;
17    SAVER->save_grid() ;
18    SAVER->save_fields( 0 ) ;
19    SAVER->terminate_cycle() ;
20 }
```

Listing IV.10: extract from the **HC_GalerkinFE** class implementation file.

Calculation of Local Systems and Assembly

▷ The integrals intervening in the expressions of a^e and ℓ^e are approximated numerically:

$$\int_K f \simeq \sum_k w_k f(\mathbf{x}_k)$$

where the pairs $(\mathbf{x}_k, w_k)$, with $\mathbf{x}_k \in K$ and $w_k \in \mathbb{R}$ define a quadrature rule [5] over the geometric element K (which represents here either a cell or a boundary face). The PELICANS equivalent of this notion is the class **GE_QuadratureRule**.

As the temperature is discretised by finite elements $\mathbb{Q}_1$ and that λ and q_0 are uniform, the respective integrands of a^e and ℓ^e are polynomials of a degree less than three.

The user must explicitly choose the quadrature rules used from those available[3]. As frequently in similar situations, this choice is expressed by using an object from a concrete class, which represents the choice made, via a pointer to the abstract parent class, which represents all the possible variants of this choice. Thus the class **GE_QRprovider** describes providers of quadrature rules (*i.e.* of instances of **GE_QuadratureRule**) which are exact, in all the geometric meshes that PELICANS can manage, for all the polynomials of a certain degree. It is an abstract class which formalises the interface used by **PDE_LocalFEcell** and **PDE_LocalFEbound** to activate an iteration in the integration points of a given mesh. We have little interest in what is exactly the class **GE_QRprovider** and its relationship with **GE_QuadratureRule**. Choosing an object of a concrete class derived from **GE_QRprovider** comes down to choosing a family of quadrature rules used by **PDE_LocalFEcell** and **PDE_LocalFEbound**: this is the relevant information.

[3]In fact, the class `GE_QuadratureRule` is a *plug-point* of the PELICANS framework, so that the user can add new quadrature rules by creating concrete classes derived from `GE_QuadratureRule`.

Thus, the local variables **`qrp`** (line 8 of Listing IV.11 and line 7 of Listing IV.12) reference an object of the class **`GE_QRprovider`** which can ultimately be used to create exact numeric quadratures for polynomials of a degree less than or equal to three. This choice is communicated to the objects referenced by **`cFE`** and **`bFE`** at lines 18 and 25 of Listings IV.11 and IV.12.

▷ The member function **`loop_on_cells`**, for which the text is given by Listing IV.11, implements the construction of local systems at each cell and their assembly in the global discrete problem.

One of the classic difficulties in programming the finite element method relates to the three families of indices intervening in the assembly phase. PELICANS copes entirely with this difficulty, so that the user is not faced with the related technical expertise. On the other hand, it is a good idea to understand the respective responsibilities allocated to the various classes.

The contribution of a given cell $\mathcal{C}_e$ to the discrete problem is obtained by the relationships (4.5) and (4.6) and by the algorithm (4.7).

In the text of the function **`loop_on_cells`**, the integer variables **`i`** and **`j`** represent i and j respectively.

The implementation of the assembly algorithm using PELICANS components brings to the fore five given objects with clearly-defined responsibilities.

- The object referenced by **`qrp`**, for which the standard is derived from **`GE_QRprovider`**, shows the expression of the choice of a certain quadrature rule in the mesh considered.

- The object referenced by **`cFE`**, of type **`PDE_LocalFEcell`**, is used to perform the iterations on the cells and on the chosen integration points. It also provides access to all the information required to construct the local discrete system: connectivity linking the local and global node numbering, value of base functions and their derivatives at the integration points (and other services that are not used here and which are described in the reference documentation.

- The object referenced by **`ELEMENT_EQ`**, of type **`PDE_LocalEquation`**, has two essential characteristics. For all pairs (i, j) in a given mesh, it knows the global numbers of nodes of related **`DOF`**s (information communicated during its configuration at lines 15 and 16). In addition, it can store the values $a^e ij$ and $\ell^e(i)$.

- The object referenced by **`TT_link`**, with the type **`PDE_LinkDOF2Unknown`**, provides access to the cross-referencing between the indexing of **`DOF`** nodes and the numbering of the components of discrete unknowns (indices I).

- The object referenced by **`GLOBAL_EQ`**, with the type **`PDE_BlockAssembledSystem`**, is used to assemble the global discrete problem from the **`ELEMENT_EQ`** and **`TT_link`** data (line 36).

Once the respective responsibilities of the various PELICANS components brought into play have been assimilated, the text of the function **`loop_on_cells`** can be understood easily (Listing IV.11).

```
void HC_GalerkinFE:: loop_on_cells( void ) const
{
   PEL_LABEL( "HC_GalerkinFE:: loop_on_cells" ) ;

   PDE_LocalFE::field_id const row = PDE_LocalFE::row ;
   PDE_LocalFE::field_id const col = PDE_LocalFE::col ;

   GE_QRprovider const* qrp = GE_QRprovider::object( "GE_QRprovider_3" ) ;

   size_t nb_dims = cFE->nb_space_dimensions() ;
   for( cFE->start() ; cFE->is_valid() ; cFE->go_next() )
   {
      cFE->set_row_and_col_fields( TT, TT ) ;

      ELEMENT_EQ->initialize( cFE->row_field_node_connectivity(), 1,
                              cFE->col_field_node_connectivity(), 1 ) ;

      cFE->start_IP_iterator( qrp ) ;
      for( ; cFE->valid_IP() ; cFE->go_next_IP() )
      {
         for( size_t i=0 ; i<cFE->nb_basis_functions( row ) ; ++i )
         {
            for( size_t j=0 ; j<cFE->nb_basis_functions( row ) ; ++j )
            {
               double xx = 0.0 ;
               for( size_t d=0 ; d<nb_dims ; ++d )
               {
                  xx += cFE->dN_at_IP( col, j, d ) *
                        cFE->dN_at_IP( row, i, d ) ;
               }
               xx = xx * cFE->weight_of_IP() * CONDUCTIVITY ;
               ELEMENT_EQ->add_to_matrix( xx, i, j ) ;
            }
         }
      }
      GLOBAL_EQ->assemble_in_LHS_and_RHS( ELEMENT_EQ, A, F ) ;
   }
}
```

Listing IV.11: extract from the **HC_GalerkinFE** class implementation file.

▷ The principles governing the implementation of the function **loop_on_cells** are valid for the function **loop on_bounds**; its text is given in Listing IV.12. The objects referenced by **cFE** and **bFE** have similar behaviour in many points, which is explained by the fact that the classes **PDE_LocalFEcell** and **PDE_LocalFEbound** both derive from the class **PDE_LocalFE**, thus factoring the aspects common to discrete system constructions local to cells and boundary faces.

The only difference worthy of note between the implementation of **loop_on_bounds** and **loop_on_cells** is the appearance of the object of type **PDE_SetOfBCs** referenced by **BCs**. This basically gives access to the information in the **MODULE** named **boundary_conditions** found in the hierarchical data structure of the **data.pel** file (Listing IV.9).

```
void HC_GalerkinFE:: loop_on_bounds( void ) const
{
   PEL_LABEL( "HC_GalerkinFE:: loop_on_bounds" ) ;

   PDE_LocalFE::field_id const row = PDE_LocalFE::row ;

   GE_QRprovider const* qrp = GE_QRprovider::object( "GE_QRprovider_3" ) ;

   for( bFE->start() ; bFE->is_valid() ; bFE->go_next() )
   {
      bFE->set_row_and_col_fields( TT, TT ) ;

      ELEMENT_EQ->initialize( bFE->row_field_node_connectivity(), 1,
                              bFE->col_field_node_connectivity(), 1 ) ;

      GE_Color const* color = bFE->color() ;
      if( BCs->has_BC( color, TT ) )
      {
         PEL_ModuleExplorer const* ee = BCs->BC_explorer( color, TT ) ;
         string bc_type = ee->string_data( "type" ) ;
         if( bc_type=="imposed_flux" )
         {
            double flux = ee->double_data( "flux_value" ) ;

            bFE->start_IP_iterator( qrp ) ;
            for( ; bFE->valid_IP() ; bFE->go_next_IP() )
            {
               for( size_t i=0 ; i<bFE->nb_basis_functions( row ) ; ++i )
               {
                  double xx = bFE->weight_of_IP() *
                              flux * bFE->N_at_IP( row, i ) ;
                  ELEMENT_EQ->add_to_vector( xx, i ) ;
               }
            }
         }
      }
      GLOBAL_EQ->assemble_in_LHS_and_RHS( ELEMENT_EQ, A, F ) ;
   }
}
```

Listing IV.12: extract from the **HC_GalerkinFE** class implementation file.

IV.2.3 Implementation of the Finite Volume Method

The classes **HC_FiniteVolume** and **HC_GalerkinFE** share many characteristics which have been discussed in the previous paragraph. The structure of the header file (Listing IV.13), the use of the *plug-in* mechanism (Listings IV.13 and IV.14), the implementation of the constructor called by the member function **create_replica** (Listing IV.15) and the hierarchical data structure of the **data.pel** file (Listing IV.16) are similar in their equivalence in the class **HC_GalerkinFE**.

We shall limit the discussion here to the specific features induced by the finite volume method.

```
class HC_FiniteVolume : public PEL_Application
{
   public: //--------------------------------------

   //- Program core execution

      virtual void run( void ) ;

   protected: //-----------------------------------

   private: //-------------------------------------

     ~HC_FiniteVolume( void ) ;
      HC_FiniteVolume( HC_FiniteVolume const& other ) ;
      HC_FiniteVolume& operator=( HC_FiniteVolume const& other ) ;

      HC_FiniteVolume( PEL_Object* a_owner,
                       PEL_ModuleExplorer const* exp ) ;

   //- Plug in

      HC_FiniteVolume( void ) ;

      virtual HC_FiniteVolume* create_replica(
                               PEL_Object* a_owner,
                               PEL_ModuleExplorer const* exp ) const ;

   //- Discrete system building

      void loop_on_sides( void ) ;

      void loop_on_bounds( void ) ;

   //- Class attributes

      static HC_FiniteVolume const* PROTOTYPE ;

   //- Attributes

      PDE_DiscreteField* TT ;
      PDE_LinkDOF2Unknown* TT_link ;
      double CONDUCTIVITY ;
      PDE_SetOfBCs const* BCs ;

      PDE_CursorFEside* sFE ;
      PDE_LocalFEbound* bFE ;

      PDE_ResultSaver* SAVER ;

      PDE_BlockAssembledSystem* GLOBAL_EQ ;
      PDE_BlockAssembledSystem::LHSindex A ;
      PDE_BlockAssembledSystem::RHSindex F ;
      PDE_BlockAssembledSystem::UNKindex X ;
} ;
```

Listing IV.13: extract of the class **HC_FiniteVolume** header file.

```
HC_FiniteVolume const* HC_FiniteVolume::PROTOTYPE = new HC_FiniteVolume() ;

HC_FiniteVolume:: HC_FiniteVolume( void )
   : PEL_Application( "HC_FiniteVolume" )
{}

HC_FiniteVolume:: ~HC_FiniteVolume( void )
{}

HC_FiniteVolume*
HC_FiniteVolume:: create_replica( PEL_Object* a_owner,
                                  PEL_ModuleExplorer const* exp ) const
{
   PEL_LABEL( "HC_FiniteVolume:: create_replica" ) ;
   PEL_CHECK( create_replica_PRE( a_owner, exp ) ) ;

   HC_FiniteVolume* result = new HC_FiniteVolume( a_owner, exp ) ;

   PEL_CHECK( create_replica_POST( result, a_owner, exp ) ) ;
   return( result ) ;
}
```

Listing IV.14: extract from the **HC_FiniteVolume** class implementation file.

```
HC_FiniteVolume:: HC_FiniteVolume( PEL_Object* a_owner,
                                   PEL_ModuleExplorer const* exp )
   : PEL_Application( a_owner, exp )
{
   PEL_LABEL( "HC_FiniteVolume:: HC_FiniteVolume" ) ;

   PEL_ModuleExplorer* ee =
                       exp->create_subexplorer( 0, "PDE_DomainAndFields" ) ;
   PDE_DomainAndFields* dom = PDE_DomainAndFields::create( this, ee ) ;
   ee->destroy() ; ee = 0 ;

   TT = dom->set_of_discrete_fields()->item( "temperature" ) ;
   TT_link = PDE_LinkDOF2Unknown::create( this, TT, true ) ;

   sFE = dom->create_CursorFEside( this ) ;
   sFE->require_field_calculation( TT, PDE_LocalFE::node ) ;

   bFE = dom->create_LocalFEbound( this ) ;
   bFE->require_field_calculation( TT, PDE_LocalFE::node ) ;

   BCs = dom->set_of_boundary_conditions() ;

   ee = exp->create_subexplorer( 0, "HC_FiniteVolume" ) ;
   CONDUCTIVITY = ee->double_data( "conductivity" ) ;

   PEL_ModuleExplorer* eee =
             ee->create_subexplorer( ee, "PDE_BlockAssembledSystem" ) ;
   GLOBAL_EQ = PDE_BlockAssembledSystem::make( this, eee, TT_link, 1 ) ;
   ee->destroy() ; ee = 0 ;

   A = GLOBAL_EQ->LHS_index( 0, 0, 0 ) ;
   F = GLOBAL_EQ->RHS_index( 0, 0 ) ;
   X = GLOBAL_EQ->UNK_index( 0, 0 ) ;

   SAVER = dom->result_saver() ;
}
```

Listing IV.15: extract from the **HC_FiniteVolume** class implementation file.

▷ The geometrical domain Ω is triangulated by a mesh with rectangular cells (lines 7–12 in Listing IV.16), in which the unknown field T is discretised constantly in each cell (line 17 of Listing IV.16).

Thus is defined the discrete representation of T, noted $\mathcal{T} = (T_K)$, where the index K designates the centre $\mathbf{x}_K$ of a cell and characterises entirely a degree of freedom, or **DOF** for short (if the unknown field had several components, the component index would have to be added to define a **DOF**). The discrete representation of the field T is determined entirely by the values of its **DOF**s, which are stored in the object referenced by the attribute **TT**.

```
MODULE PEL_Application
   concrete_name="HC_FiniteVolume"
   MODULE PDE_DomainAndFields
      verbose_level = 1
      nb_space_dimensions = 2
      type = "finite_element"
      MODULE GE_Meshing
         concrete_name = "GE_BoxWithBoxes"
         vertices_coordinate_0 = regular_vector( 0.0, 10, 0.3 )
         vertices_coordinate_1 = regular_vector( 0.0, 10, 0.4 )
         mesh_polyhedron = < "GE_Segment" "GE_Rectangle" >
      END MODULE GE_Meshing
      MODULE interior_fields
         MODULE temperature
            name = "temperature"
            nb_components = 1
            element_name = "PDE_2D_Q0_1node"
            storage_depth = 1
            MODULE DOFs_values
               type = "uniformly_defined"
               value = < 0. >
            END MODULE DOFs_values
         END MODULE temperature
      END MODULE interior_fields
      MODULE boundary_conditions
         MODULE xxx
            field = "temperature"
            color = "left"
            type = "imposed_flux"
            flux_value = 50.E+3
         END MODULE xxx
         MODULE yyy
            field = "temperature"
            color = "top"
            type = "Dirichlet"
            value = 100.
         END MODULE yyy
      END MODULE boundary_conditions
      MODULE PDE_ResultSaver
         ...
      END MODULE PDE_ResultSaver
   END MODULE PDE_DomainAndFields
   MODULE HC_FiniteVolume
      conductivity = 236.
      MODULE PDE_BlockAssembledSystem
         concrete_name = "PDE_BlockAssembledSystemLA"
         ...
      END MODULE PDE_BlockAssembledSystem
   END MODULE HC_FiniteVolume
END MODULE PEL_Application
```

Listing IV.16: **data.pel** ile associated with the class **HC_FiniteVolume**.

The class **PDE_DiscreteField** contains no information relating to the type of discretization of the underlying continuous field. Its role is simply to organise all the numeric values T_K of the discrete field with a formalised indexing. It is the numeric model which allocates a conceptual meaning to the value of **DOF**s. Here it has been chosen for T_K to represent an approximation of the temperature in the centre of the cell K.

▷ None of the points $\mathbf{x}_K$ are located on the boundary Γ_{D}, therefore the Dirichlet boundary condition found in the mathematical model cannot be taken into account by allocating an imposed value to certain **DOF**s as was the case for the Finite Element Galerkin method. For this reason, the data structure in Listing IV.16 has no **`MODULE`** named **`DOFs_imposed_value`**, unlike that of Listing IV.9. It does on the other hand have an additional **`MODULE`** in the heading **`boundary_conditions`** (Lines 32 to 37), which provides the information required to calculate flux at the boundary faces belonging to Γ_{D}.

▷ The global discrete system is a linear system constructed during an iteration at the internal faces of the meshing, with responsibility allocated to the object of type **`PDE_CursorFEside`** referenced by the attribute **`sFE`**, then during an iteration at the boundary faces of the meshing, with responsibility allocated to the object of type **`PDE_LocalFEbound`** referenced by the attribute **`bFE`**.

▷ The text of the member function **`run`** is given in Listing IV.17.

The discrete problem is constructed by calling the function **`loop_on_sides`** (line 5) which performs an iteration over the internal faces and adds the contributions relating to the approximation (4.10) relating to the current face, then by calling the function **`loop_on_bounds`** (line 7), which performs an iteration over the boundary faces and adds the contributions relating to the approximations (4.11) and (4.12) depending on the location of the current boundary face.

```
 1  void HC_FiniteVolume:: run( void )
 2  {
 3     PEL_LABEL( "HC_FiniteVolume:: run" ) ;
 4
 5     loop_on_sides() ;
 6
 7     loop_on_bounds() ;
 8
 9     GLOBAL_EQ->estimate_unknown( X, A, F, true ) ;
10
11     if( !GLOBAL_EQ->unknown_is_solution( X ) )
12        PEL_Error::object()->raise_plain( "convergence failure" ) ;
13
14     TT->update_free_DOFs_value( 0, GLOBAL_EQ->unknown_vector( X ), TT_link ) ;
15
16     SAVER->start_cycle() ;
17     SAVER->save_grid() ;
18     SAVER->save_fields( 0 ) ;
19     SAVER->terminate_cycle() ;
20  }
```

Listing IV.17: extract from the **`HC_FiniteVolume`** class implementation file.

▷ Integrals contribute to the discrete problem at the internal faces of the meshing during execution of the member function **`loop_on_sides`**, for which the text is given in Listing IV.18.

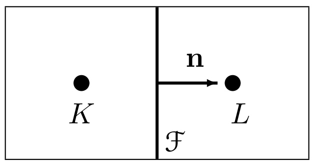
The instructions in lines 21 to 25 add to the block the contributions induced by the flux, at the current face noted $\mathcal{F}$, approximated according to the formula (4.10):

$$\int_{\mathcal{F}} -\lambda \nabla T \cdot \mathbf{n} \;\approx\; -\frac{|\mathcal{F}|\;\lambda}{\ell}\, T_L + \frac{|\mathcal{F}|\;\lambda}{\ell}\, T_K$$

This flux intervenes in the integral formulation relating to the two cells adjacent to $\mathcal{F}$ and therefore contributes to:

- two coefficients of the line defined by the **`DOF`** at the centre $\mathbf{x}_K$ of the first adjacent cell;
- two coefficients of the line defined by the **`DOF`** at the centre $\mathbf{x}_L$ of the second adjacent cell.

Figure IV.2 illustrates and justifies lines 21 to 25 of Listing IV.18.



colonne→ ligne↓	K	L
K	$\frac{\lvert\mathfrak{F}\rvert\,\lambda}{\ell}$	$-\frac{\lvert\mathfrak{F}\rvert\,\lambda}{\ell}$
L	$-\frac{\lvert\mathfrak{F}\rvert\,\lambda}{\ell}$	$\frac{\lvert\mathfrak{F}\rvert\,\lambda}{\ell}$

FIGURE IV.2: elementary coefficients produced by the approximation of the flux at the face $\mathfrak{F}$.

```
void HC_FiniteVolume:: loop_on_sides( void )
{
   PEL_LABEL( "HC_FiniteVolume:: loop_on_sides" ) ;

   PDE_LocalFEcell const* fe_K = sFE->adjacent_localFEcell( 0 ) ;
   PDE_LocalFEcell const* fe_L = sFE->adjacent_localFEcell( 1 ) ;

   for( sFE->start() ; sFE->is_valid() ; sFE->go_next() )
   {
      PDE_DiscreteField::Node n_K = fe_K->global_node( TT, 0 ) ;
      PDE_DiscreteField::Node n_L = fe_L->global_node( TT, 0 ) ;

      double d_KL = sFE->distance_to_adjacent_finite_volume_center( 0 ) +
                    sFE->distance_to_adjacent_finite_volume_center( 1 ) ;

      double xx = CONDUCTIVITY * sFE->polyhedron()->measure() / d_KL ;

      size_t u_K = TT_link->unknown_linked_to_DOF( n_K, 0 ) ;
      size_t u_L = TT_link->unknown_linked_to_DOF( n_L, 0 ) ;

      GLOBAL_EQ->add_to_LHS_submatrix_item( A, u_K, u_K,  xx ) ;
      GLOBAL_EQ->add_to_LHS_submatrix_item( A, u_L, u_K, -xx ) ;

      GLOBAL_EQ->add_to_LHS_submatrix_item( A, u_K, u_L, -xx ) ;
      GLOBAL_EQ->add_to_LHS_submatrix_item( A, u_L, u_L,  xx ) ;
   }
}
```

Listing IV.18: extract from the **HC_FiniteVolume** class implementation file.

▷ The principles governing the implementation of the function **loop_on_sides** are valid for the function **loop_on_bounds**; its text is given in Listing IV.19.

During the iteration over the boundary faces, the object of type **PDE_SetOfBCs** referenced by the attribute **BCs** is used to test the possible presence of a boundary condition at the current face (line 8) and its type (lines 16 and 24) as defined in the **MODULE** named **boundary_conditions** found in the hierarchical data structure of the **data.pel** file (Listing IV.16).

The flux at a boundary face intervene in the integral formulation over its adjacent cell and therefore contribute to the coefficients of the discrete system associated with the line defined by the **DOF** located in the centre of this cell.

Given these geometric constraints, the instructions in lines 21, 22 and 28 bring to the discrete problem the contributions relating to the approximations (4.11) and (4.12) of flux at the current boundary face.

```
void HC_FiniteVolume:: loop_on_bounds( void )
{
   PEL_LABEL( "HC_FiniteVolume:: loop_on_bounds" ) ;

   for( bFE->start() ; bFE->is_valid() ; bFE->go_next() )
   {
      GE_Color const* color = bFE->color() ;
      if( BCs->has_BC( color, TT ) )
      {
         PEL_ModuleExplorer const* ee = BCs->BC_explorer( color, TT ) ;

         PDE_DiscreteField::Node n_K = bFE->global_node( TT, 0 ) ;
         size_t u_K = TT_link->unknown_linked_to_DOF( n_K, 0 ) ;

         string const& type = ee->string_data( "type" ) ;
         if( type == "Dirichlet" )
         {
            double val = ee->double_data( "value" ) ;
            double xx = CONDUCTIVITY * bFE->polyhedron()->measure() /
                        bFE->distance_to_adjacent_finite_volume_center() ;
            GLOBAL_EQ->add_to_LHS_submatrix_item( A, u_K, u_K, xx ) ;
            GLOBAL_EQ->add_to_RHS_subvector_item( F, u_K, xx * val ) ;
         }
         else if( type == "imposed_flux" )
         {
            double xx = bFE->polyhedron()->measure() *
                        ee->double_data( "flux_value" ) ;
            GLOBAL_EQ->add_to_RHS_subvector_item( F, u_K, xx ) ;
         }
      }
   }
}
```

Listing IV.19: extract from the **HC_FiniteVolume** class implementation file.

Chapter V

Assessment and Conclusion

The numeric simulation activity in which is inserted the use of PELICANS is presented schematically in Figure V.1.

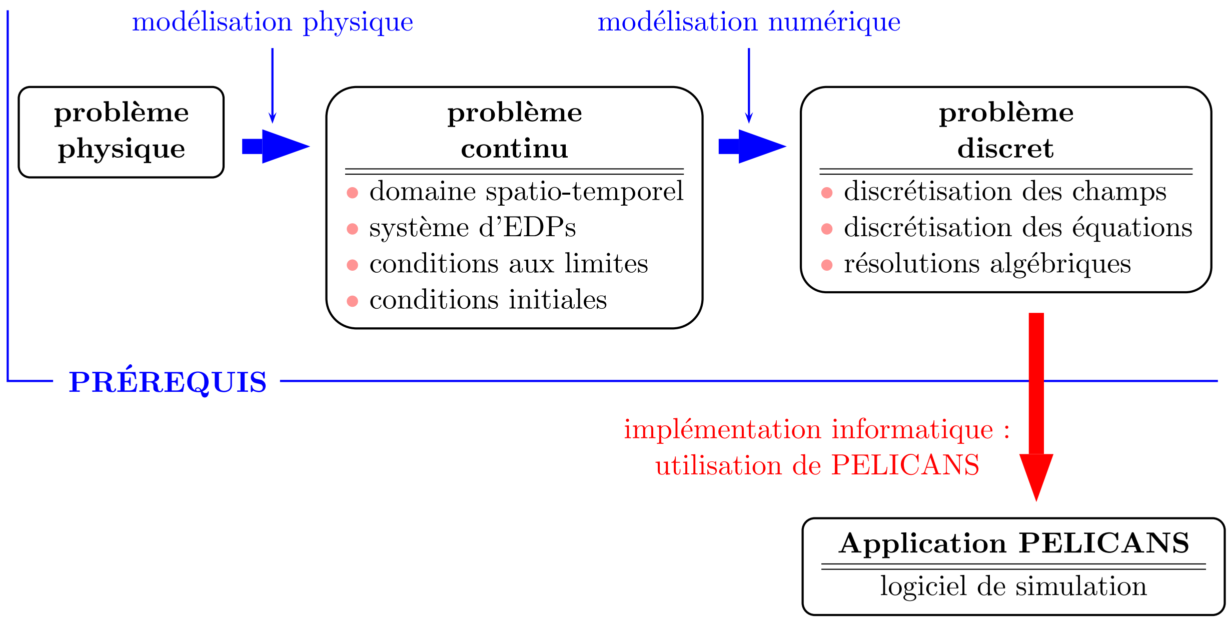


FIGURE V.1: numeric simulation and use of PELICANS.

▷ PELICANS appears clearly as a computer implementation tool.

▷ Its software architecture has been designed to be as close as possible to its business domain whilst remaining sufficiently flexible to enable the essential extensions linked to the changes in mathematical and numeric modelling. To build an architecture of this type with two properties sometimes (and erroneously) considered antagonistic, namely rigour and flexibility, the object principles have been used and extended by strengthening the role of interfaces by adding the notion of *specification* of *software components*.

Thus, a set of tools and programming techniques have been developed especially to implement in `C++` language development principles based on software components (Component-Based Development or CBD for short) such as Programming by Contract, separation of commands and queries, inheritance, naming and self-documentation. As such this all constitutes one of the finished products of the PELICANS project and forms a consistent, validated whole dedicated to the construction of scientific calculation applications.

▷ These tools and techniques have been applied to the numeric resolution of PDE systems in meshed geometries. The domain analysis has resulted in the design and partial implementation of a collection of abstractions and interaction modes between these abstractions, the whole defining the re-usable core of a family of PDE solvers, in other words, in terms of software engineering, a *framework*. Developing an application from this *framework* consists of completing its implementation (by definition, the implementation of an abstraction is partial, therefore deferred) by providing the missing parts as inherited classes, which are programmed by using the elementary functionalities of the range of software components available "off-the-shelf" within the *framework*.

▷ To summarise, PELICANS is a *framework* of *adaptable software components* for the implementation of PDE solvers by numeric methods for which the global specific aspects are given in Table V.1.

Mailleur –

- fonctionnalités intégrées pour des domaines simples
- couplage avec des outils externes: Méfisto (U Paris VI), GAMBIT (FLUENT), EMC2 (INRIA), Triangle (UC Berkeley), GMSH (U Louvain)

Post-traitement –

couplage avec des outils externes: GMV (LANL), TIC (IRSN), OpenDX (IBM), Paraview (Sandia), MeshTV (LLNL), gnuplot

Types de Maille –

1D (segment) 2D (triangle, quadrilatère) 3D (hexaèdre, tétraèdre)

Représentation Discrète des Champs –

éléments finis $\mathbb{P}_0$, $\mathbb{P}_1$, $\mathbb{P}_1$bulle, $\mathbb{P}_1$iso$\mathbb{P}_2$, $\mathbb{P}_1$non-conforme, $\mathbb{P}_2$, $\mathbb{Q}_0$, $\mathbb{Q}_1$, $\mathbb{Q}_1$bulles, $\mathbb{Q}_1$non-conforme, $\mathbb{Q}_2$, $\mathbb{Q}_2$incomplet

Frontières Mobiles –

- domaine fluide variable sur un maillage Eulérien
- méthode Lagrange Euler Arbitraire

Multi-domaines –

éléments finis joints (mortar)

Discrétisation Spatiale des EDPs – LIBRE, par exemple :

- volumes finis
- méthodes d'approximation variationnelle, *eg* :
  - Galerkin standard et stabilisée (SUPG, GLS, PSPG, ...)
  - Galerkin-Caractéristique
  - Galerkin-Discontinue
- Méthodes mixtes volumes finis / éléments finis

Discrétisation Temporelles des EDPs – LIBRE, par exemple :

- Differentiation Rétrogrades (BDF)
- Theta methodes
- Un framework de marche en temps est fourni

Couplage de Problèmes – LIBRE

un framework de couplage explicite/implicite est fourni

Algèbre Linéaire –

- structures de données : vecteurs, matrices (pleines, diagonales, symétriques, creuses, blocs)
- méthodes de Krylov (CG, GMRES) préconditionnées (SSOR, ILU(p), ILUT, MILU)
- couplage avec les librairies externes PETSc (ANL), Aztec (Sandia), SPARSKIT (Y. Saad), UMFPACK (U. Florida)

Parallélisme –

machines à mémoire distribuée

Méthodes Multiniveaux – en cours de développement

- méthodes multi-grilles géométriques
- raffinement local adaptatif

TABLE V.1: functionalities available for the numeric resolution of PDE systems.

License for PELICANS

This Agreement may apply to any or all software for which the holder of the economic rights decides to submit the use thereof to its provisions.

Article 1 - DEFINITIONS

For the purpose of this Agreement, when the following expressions commence with a capital letter, they shall have the following meaning:

Agreement: means this license agreement, and its possible subsequent versions and annexes.

Software: means the software in its Object Code and/or Source Code form and, where applicable, its documentation, "as is" when the Licensee accepts the Agreement.

Initial Software: means the Software in its Source Code and possibly its Object Code form and, where applicable, its documentation, "as is" when it is first distributed under the terms and conditions of the Agreement.

Modified Software: means the Software modified by at least one Integrated Contribution.

Source Code: means all the Software's instructions and program lines to which access is required so as to modify the Software.

Object Code: means the binary files originating from the compilation of the Source Code.

Holder: means the holder(s) of the economic rights over the Initial Software.

Licensee: means the Software user(s) having accepted the Agreement.

Contributor: means a Licensee having made at least one Integrated Contribution.

Licensor: means the Holder, or any other individual or legal entity, who distributes the Software under the Agreement.

Integrated Contribution: means any or all modifications, corrections, translations, adaptations and/or new functions integrated into the Source Code by any or all Contributors.

Related Module: means a set of sources files including their documentation that, without modification to the Source Code, enables supplementary functions or services in addition to those offered by the Software.

Derivative Software: means any combination of the Software, modified or not, and of a Related Module.

Parties: mean both the Licensee and the Licensor.

These expressions may be used both in singular and plural form.

Article 2 - PURPOSE

The purpose of the Agreement is the grant by the Licensor to the Licensee of a non-exclusive, transferable and worldwide license for the Software as set forth in Article 5 hereinafter for the whole term of the protection granted by the rights over said Software.

Article 3 - ACCEPTANCE

3.1 – The Licensee shall be deemed as having accepted the terms and conditions of this Agreement upon the occurrence of the first of the following events:

(i) loading the Software by any or all means, notably, by downloading from a remote server, or by loading from a physical medium;

(ii) the first time the Licensee exercises any of the rights granted hereunder.

3.2 – One copy of the Agreement, containing a notice relating to the characteristics of the Software, to the limited warranty, and to the fact that its use is restricted to experienced users has been provided to the Licensee prior to its acceptance as set forth in Article 3.1 hereinabove, and the Licensee hereby acknowledges that it has read and understood it.

Article 4 - EFFECTIVE DATE AND TERM

4.1 EFFECTIVE DATE – The Agreement shall become effective on the date when it is accepted by the Licensee as set forth in Article 3.1.

4.2 TERM – The Agreement shall remain in force for the entire legal term of protection of the economic rights over the Software.

Article 5 - SCOPE OF RIGHTS GRANTED

The Licensor hereby grants to the Licensee, who accepts, the following rights over the Software for any or all use, and for the term of the Agreement, on the basis of the terms and conditions set forth hereinafter.

Besides, if the Licensor owns or comes to own one or more patents protecting all or part of the functions of the Software or of its components, the Licensor undertakes not to enforce the rights granted by these patents against successive Licensees using, exploiting or modifying the Software. If these patents are transferred, the Licensor undertakes to have the transferees subscribe to the obligations set forth in this paragraph.

5.1 RIGHT OF USE – The Licensee is authorized to use the Software, without any limitation as to its fields of application, with it being hereinafter specified that this comprises:

1. permanent or temporary reproduction of all or part of the Software by any or all means and in any or all form.
2. loading, displaying, running, or storing the Software on any or all medium.
3. entitlement to observe, study or test its operation so as to determine the ideas and principles behind any or all constituent elements of said Software. This shall apply when the Licensee carries out any or all loading, displaying, running, transmission or storage operation as regards the Software, that it is entitled to carry out hereunder.

5.2 RIGHT OF MODIFICATION – The right of modification includes the right to translate, adapt, arrange, or make any or all modifications to the Software, and the right to reproduce the resulting software. It includes, in particular, the right to create a Derivative Software.

The Licensee is authorized to make any or all modification to the Software provided that it includes an explicit notice that it is the author of said modification and indicates the date of the creation thereof.

5.3 RIGHT OF DISTRIBUTION – In particular, the right of distribution includes the right to publish, transmit and communicate the Software to the general public on any or all medium, and by any or all means, and the right to market, either in consideration of a fee, or free of charge, one or more copies of the Software by any means.

The Licensee is further authorized to distribute copies of the modified or unmodified Software to third parties according to the terms and conditions set forth hereinafter.

5.3.1 DISTRIBUTION OF SOFTWARE WITHOUT MODIFICATION – The Licensee is authorized to distribute true copies of the Software in Source Code or Object Code form, provided that said distribution complies with all the provisions of the Agreement and is accompanied by:

1. a copy of the Agreement,
2. a notice relating to the limitation of both the Licensor's warranty and liability as set forth in Articles 8 and 9,

and that, in the event that only the Object Code of the Software is redistributed, the Licensee allows effective access to the full Source Code of the Software at a minimum during the entire period of its distribution of the Software, it being understood that the additional cost of acquiring the Source Code shall not exceed the cost of transferring the data.

5.3.2 DISTRIBUTION OF MODIFIED SOFTWARE – When the Licensee makes an Integrated Contribution to the Software, the terms and conditions for the distribution of the resulting Modified Software become subject to all the provisions of this Agreement.

The Licensee is authorized to distribute the Modified Software, in source code or object code form, provided that said distribution complies with all the provisions of the Agreement and is accompanied by:

1. a copy of the Agreement,
2. a notice relating to the limitation of both the Licensor's warranty and liability as set forth in Articles 8 and 9,

and that, in the event that only the object code of the Modified Software is redistributed, the Licensee allows effective access to the full source code of the Modified Software at a minimum during the entire period of its distribution of the Modified Software, it being understood that the additional cost of acquiring the source code shall not exceed the cost of transferring the data.

5.3.3 DISTRIBUTION OF DERIVATIVE SOFTWARE – When the Licensee creates Derivative Software, this Derivative Software may be distributed under a license agreement other than this Agreement, subject to compliance with the requirement to include a notice concerning the rights over the Software as defined in Article 6.4. In the event the creation of the Derivative Software required modification of the Source Code, the Licensee undertakes that:

1. the resulting Modified Software will be governed by this Agreement,
2. the Integrated Contributions in the resulting Modified Software will be clearly identified and documented,
3. the Licensee will allow effective access to the source code of the Modified Software, at a minimum during the entire period of distribution of the Derivative Software, such that such modifications may be carried over in a subsequent version of the Software; it being understood that the additional cost of purchasing the source code of the Modified Software shall not exceed the cost of transferring the data.

5.3.4 COMPATIBILITY WITH THE CeCILL LICENSE – When a Modified Software contains an Integrated Contribution subject to the CeCILL license agreement, or when a Derivative Software contains a Related Module subject to the CeCILL license agreement, the provisions set forth in the third item of Article 6.4 are optional.

Article 6 - INTELLECTUAL PROPERTY

6.1 OVER THE INITIAL SOFTWARE – The Holder owns the economic rights over the Initial Software. Any or all use of the Initial Software is subject to compliance with the terms and conditions under which the Holder has elected to distribute its work and no one shall be entitled to modify the terms and conditions for the distribution of said Initial Software.

The Holder undertakes that the Initial Software will remain ruled at least by this Agreement, for the duration set forth in Article 4.2.

6.2 OVER THE INTEGRATED CONTRIBUTIONS – The Licensee who develops an Integrated Contribution is the owner of the intellectual property rights over this Contribution as defined by applicable law.

6.3 OVER THE RELATED MODULES – The Licensee who develops a Related Module is the owner of the intellectual property rights over this Related Module as defined by applicable law and is free to choose the type of agreement that shall govern its distribution under the conditions defined in Article 5.3.3.

6.4 NOTICE OF RIGHTS – The Licensee expressly undertakes:

1. not to remove, or modify, in any manner, the intellectual property notices attached to the Software;
2. to reproduce said notices, in an identical manner, in the copies of the Software modified or not;
3. to ensure that use of the Software, its intellectual property notices and the fact that it is governed by the Agreement is indicated in a text that is easily accessible, specifically from the interface of any Derivative Software.

The Licensee undertakes not to directly or indirectly infringe the intellectual property rights of the Holder and/or Contributors on the Software and to take, where applicable, vis-à-vis its staff, any and all measures required to ensure respect of said intellectual property rights of the Holder and/or Contributors.

Article 7 - RELATED SERVICES

7.1 – Under no circumstances shall the Agreement oblige the Licensor to provide technical assistance or maintenance services for the Software.

However, the Licensor is entitled to offer this type of services. The terms and conditions of such technical assistance, and/or such maintenance, shall be set forth in a separate instrument. Only the Licensor offering said maintenance and/or technical assistance services shall incur liability therefor.

7.2 – Similarly, any Licensor is entitled to offer to its licensees, under its sole responsibility, a warranty, that shall only be binding upon itself, for the redistribution of the Software and/or the Modified Software, under terms and conditions that it is free to decide. Said warranty, and the financial terms and conditions of its application, shall be subject of a separate instrument executed between the Licensor and the Licensee.

Article 8 - LIABILITY

8.1 – Subject to the provisions of Article 8.2, the Licensee shall be entitled to claim compensation for any direct loss it may have suffered from the Software as a result of a fault on the part of the relevant Licensor, subject to providing evidence thereof.

8.2 – The Licensor's liability is limited to the commitments made under this Agreement and shall not be incurred as a result of in particular: (i) loss due the Licensee's total or partial failure to fulfill its obligations, (ii) direct or consequential loss that is suffered by the Licensee due to the use or performance of the Software, and (iii) more generally, any consequential loss. In particular the Parties expressly agree that any or all pecuniary or business loss (i.e. loss of data, loss of profits, operating loss, loss of customers or orders, opportunity cost, any disturbance to business activities) or any or all legal proceedings instituted against the Licensee by a third party, shall constitute consequential loss and shall not provide entitlement to any or all compensation from the Licensor.

Article 9 - WARRANTY

9.1 – The Licensee acknowledges that the scientific and technical state-of-the-art when the Software was distributed did not enable all possible uses to be tested and verified, nor for the presence of possible defects to be detected. In this respect, the Licensee's attention has been drawn to the risks associated with loading, using, modifying and/or developing and reproducing the Software which are reserved for experienced users.

The Licensee shall be responsible for verifying, by any or all means, the suitability of the product for its requirements, its good working order, and for ensuring that it shall not cause damage to either persons or properties.

9.2 – The Licensor hereby represents, in good faith, that it is entitled to grant all the rights over the Software (including in particular the rights set forth in Article 5).

9.3 – The Licensee acknowledges that the Software is supplied ”as is” by the Licensor without any other express or tacit warranty, other than that provided for in Article 9.2 and, in particular, without any warranty as to its commercial value, its secured, safe, innovative or relevant nature.

Specifically, the Licensor does not warrant that the Software is free from any error, that it will operate without interruption, that it will be compatible with the Licensee's own equipment and software configuration, nor that it will meet the Licensee's requirements.

9.4 – The Licensor does not either expressly or tacitly warrant that the Software does not infringe any third party intellectual property right relating to a patent, software or any other property right. Therefore, the Licensor disclaims any and all liability towards the Licensee arising out of any or all proceedings for infringement that may be instituted in respect of the use, modification and redistribution of the Software. Nevertheless, should such proceedings be instituted against the Licensee, the Licensor shall provide it with technical and legal assistance for its defense. Such technical and legal assistance shall be decided on a case-by-case basis between the relevant Licensor and the Licensee pursuant to a memorandum of understanding. The Licensor disclaims any and all liability as regards the Licensee's use of the name of the Software. No warranty is given as regards the existence of prior rights over the name of the Software or as regards the existence of a trademark.

Article 10 - TERMINATION

10.1 – In the event of a breach by the Licensee of its obligations hereunder, the Licensor may automatically terminate this Agreement thirty (30) days after notice has been sent to the Licensee and has remained ineffective.

10.2 – A Licensee whose Agreement is terminated shall no longer be authorized to use, modify or distribute the Software. However, any licenses that it may have granted prior to termination of the Agreement shall remain valid subject to their having been granted in compliance with the terms and conditions hereof.

Article 11 - MISCELLANEOUS

11.1 EXCUSABLE EVENTS – Neither Party shall be liable for any or all delay, or failure to perform the Agreement, that may be attributable to an event of force majeure, an act of God or an outside cause, such as defective functioning or interruptions of the electricity or telecommunications networks, network paralysis following a virus attack, intervention by government authorities, natural disasters, water damage, earthquakes, fire, explosions, strikes and labor unrest, war, etc.

11.2 – Any failure by either Party, on one or more occasions, to invoke one or more of the provisions hereof, shall under no circumstances be interpreted as being a waiver by the interested Party of its right to invoke said provision(s) subsequently.

11.3 – The Agreement cancels and replaces any or all previous agreements, whether written or oral, between the Parties and having the same purpose, and constitutes the entirety of the agreement between said Parties concerning said purpose. No supplement or modification to the terms and conditions hereof shall be effective as between the Parties unless it is made in writing and signed by their duly authorized representatives.

11.4 – In the event that one or more of the provisions hereof were to conflict with a current or future applicable act or legislative text, said act or legislative text shall prevail, and the Parties shall make the necessary amendments so as to comply with said act or legislative text. All other provisions shall remain effective. Similarly, invalidity of a provision of the Agreement, for any reason whatsoever, shall not cause the Agreement as a whole to be invalid.

11.5 LANGUAGE – The Agreement is drafted in both French and English and both versions are deemed authentic.

Article 12 - NEW VERSIONS OF THE AGREEMENT

12.1 – Any person is authorized to duplicate and distribute copies of this Agreement.

12.2 – So as to ensure coherence, the wording of this Agreement is protected and may only be modified by the authors of the License, who reserve the right to periodically publish updates or new versions of the Agreement, each with a separate number. These subsequent versions may address new issues encountered by Free Software.

12.3 – Any Software distributed under a given version of the Agreement may only be subsequently distributed under the same version of the Agreement or a subsequent version.

Article 13 - GOVERNING LAW AND JURISDICTION

13.1 – The Agreement is governed by French law. The Parties agree to endeavor to seek an amicable solution to any disagreements or disputes that may arise during the performance of the Agreement.

13.2 – Failing an amicable solution within two (2) months as from their occurrence, and unless emergency proceedings are necessary, the disagreements or disputes shall be referred to the Paris Courts having jurisdiction, by the more diligent Party.

Version 1.0 dated 2006-09-05.

Bibliography

[1] Object-oriented methodology for software development with PELICANS. Reference Documentation of PELICANS.

[2] Rémi Abgrall and Mohamad Mezine. Construction of second order accurate monotone and stable residual distribution schemes for unsteady flow problems. *Journal of Computational Physics*, 188(1):16–55, jun 2003.

[3] F. Babik, J.-C. Latché, and D. Vola. On a numerical strategy to compute non-newtonian fluids gravity currents. In *XIIIth International Workshop on Numerical Methods for non-Newtonian Flows, Lausanne*, 2003.

[4] P.G. Ciarlet. *Handbook of Numerical Analysis Volume II : Finite Elements Methods – Basic Error Estimates for Elliptic Problems.* North-Holland, 1991.

[5] Alexandre Ern and Jean-Luc Guermond. *Éléments Finis: Théorie, Applications, mise en œuvre*, volume 36 of *Mathématiques & Applications.* Springer, 2002.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, second edition, 1989.

[8] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, volume 17 of *Frontiers in Applied Mathematics.* SIAM, 1997.

[9] Thomas J.R. Hughes. *The Finite Element Method : Linear Static and Dynamic Finite Element Analysis.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1987.

[10] Andrew Koenig and Barbara E. Moo. *Accelerated C++. Practical Programming by Example.* C++ In-Depth Series. Addison-Wesley, 2000.

[11] J.-C. Latché. A fictitious degrees of freedom finite element method for free surface flows. In Preparation, 2003.

[12] Robert C. Martin. The Liskov substitution principle. *C++ Report*, 1996.

[13] Robert Cecil Martin. *Agile Software Development. Principles, Patterns and Practices.* Alan Apt Series. Prentice Hall, Pearson Education, Inc. Upper Saddle Reaver, New Jersey 07458., 2003. 013594445.

[14] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, second edition, 1997.

[15] Scott Meyers. *More Effective C++ : 35 New Ways to Improve Your Programs and Designs.* Professional Computing Series. Addison-Wesley, 1996.

[16] Scott Meyers. *Effective C++ : 50 Specific Ways to Improve Your Programs and Designs.* Professional Computing Series. Addison-Wesley, second edition, 1998.

[17] B. Piar, B.D. Michel, F. Babik, J.-C. Latché, G. Guillard, and J.-M. Ruggiéri. CROCO : a computer code for corium spreading. In *Ninth International Topical Meeting on Nuclear Reactor Thermal Hydraulics (NURETH-9)*, 1999. San Francisco, California, October 3-8.

[18] Alfio Quarteroni and Alberto Valli. *Numerical Approximation of Partial Differential Equations*, volume 23 of *Springer Series in Computational Mathematics.* Springer, 1997.

[19] Alfio Quarteroni and Alberto Valli. *Domain Decomposition Methods for Partial Differential Equations.* Numerical Mathematics and Scientific Computation. Oxford University Press, 1999.

[20] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, 1996.

[21] Bjarne Stroustrup. *The C++ Programming Language, Special Edition.* Addison-Wesley, 2000.

Contents