Alma Mater Studiorum
Università di Bologna

Dottorato di Ricerca in Ingegneria delle Macchine e dei Sistemi Energetici – XVIII Ciclo

Development of a New Parallel Code for Computational Continuum Mechanics Using Object–Oriented Techniques

Stefano Toninel

Tutor Scientifico:
Prof. Piero Pelloni

Coordinatore di Dottorato:
Prof. Davide Moro

Aprile 2006

Settore Scientifico Disciplinare: ING–IND/08

E–mail: stoninel@gmail.com

Abstract

The physics of continuum domains, such as a fluid region or a solid body, is described by means of Partial Differential Equations (PDEs), solved according to different algorithms, using numerical methods.

The aim of this work is to describe the framework and the main features of NEMO (Numerical Engine for Multiphysics Operators), a new software package designed for solving PDEs by means of a finite volume method, running on distributed memory parallel computers.

In order to implement a single multiphysics code, able to solve the PDEs governing different continuum mechanics phenomena (e.g. heat transfer, fluid dynamics, etc.), or even their interaction (e.g fluid–structure interaction, magneto–fluid dynamics, etc.), an innovative object–oriented approach has been followed.

This is based on the definition of classes, representing different entities, building blocks of a PDE problem (e.g. "PDE", "Field", "Mesh", "Boundary Condition", etc.), and of differential operators, numerical procedures which hide, behind their interface, the specific implementation of the chosen finite–volume method and discretization schemes.

The code is written in Fortran 95 instead of C++, the latter being considered by many as the most suitable language for scientific, object–oriented, applications. This choice has been driven by the fact that many features required by object–oriented programming, such as data hiding, encapsulation, inheritance and polymorphism, are fully or partially supported by the Fortran 95 standard, and the widely proven, better efficiency of this language has been preferred.

The software is based on a co–located approach, with problem unknowns stored at cell–centers, and it supports hybrid meshes, with generic polygonal (2D) and polyhedral (3D) cells.

Special care has been devoted to the overall computational efficiency, thanks to the integration with the PSBLAS library, which implements a set of highly efficient preconditioners and iterative methods for solving linear systems with large sparse matrices, according to a message–passing paradigm.

Dedicated to Daria
April 2006

Acknowledgements

First of all, I would like to express my sincere and deep gratitude to my supervisors, Prof. P. Pelloni and Prof. G. M. Bianchi; they've always supported my PhD project with trust and enthusiasm, believing in my capabilities, doing all that they could to improve my knowledge and sharing their passion for academic research and teaching.

Thanks also to other Professors at DIEM, in particular to Prof. A. Peretto and Prof. D. Moro, former and current coordinator of the PhD program.

Thanks to Prof. G. Bella and Dr. S. Filippone from the University of Tor Vergata – Rome. The former introduced me to the world of complex CFD simulations; the latter taught me practically everything I know about parallel computation and gave me many new references for improving my capabilities as developer. The parallel framework of my code couldn't exist without the outstanding, first class, work of the PSBLAS team.

Thanks to Prof. D. P. Schmidt who hosted me in his lab as a visiting student at the University of Massachusetts – Amherst during year 2005; I'll never forget his teachings about numerics and CFD, his passion in working with students and his desire to face new scientific challenges, sharing his knowledge with new people like me. My life has changed after the year spent at UMASS and I'll always carry nice memories of that time.

Thanks to Linus Torvalds, the Open–Source community and all people spread all around the world, who decided to share their knowledge and software for free, making my job much easier and more fun.

Thanks to my old friends in Ferrara and the new ones met along the way, in Europe as well as in America, during the PhD years.

Thanks to my parents who always encouraged me and supported my choice of being PhD student, not an easy one, nowadays, in Italy.

Last but not least, I'll be forever in debt with my lovely Daria. How could I forget her endless patience, the strength she gave to me in hard times, and how she waited for my return while I was in the USA, far from home. I couldn't have done it without her trust in me, her support, her love.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

ADT	Abstract Data Type
BC	Boundary Condition
BiCGSTAB	BiConjugate Gradient Stabilized method
CCM	Computational Continuum Mechanics
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient method
COO	Coordinates (format)
CSM	Computational Solid Mechanics
CSR	Compressed Sparse Row (format)
CV	Control Volume
DDT	Derived Data Type
DDT	Derived Data Type
DIAGSC	Diagonal Scaling
FDM	Finite Differences Method
FEM	Finite Elements Method
FVM	Finite Volume Method
HPC	High Performance Computation
HPF	High Performance Fortran

ILU	Incomplete LU factorization
IT	Information Technology
JAD	Jagged Diagonal (format)
MPI	Message Passing Interface
OOA	Object–Oriented Analysis
OOD	Object–Oriented Design
OOP	Object–Oriented Programming
PDE	Partial Differential Equation
RHS	Right Hand Side
SE	Software Engineering
SPMD	Single Program Multiple Data
UML	Unified Modeling Language

Greek Characters

$\boldsymbol{\sigma}$	stress tensor
Γ	diffusivity
λ	second Lamè's coefficient
μ	viscosity
μ_l	first Lamè's coefficient
ν	Poisson's modulus
ϕ	generic scalar
ρ	density
ε	rate of strain tensor

Latin Characters

$\boldsymbol{b}$	body force

$\boldsymbol{I}$	unit tensor
$\boldsymbol{S}$	viscous part of the stress tensor
$\boldsymbol{u}$	displacement
$\boldsymbol{v}$	velocity vector
A	coefficients matrix of a linear system
b	RHS of a linear system
c	specific heat
h	enthalpy
k	thermal conductivity
p	pressure
Pr	Prandtl number
T	total temperature
t	time
x	unknown
E	Young's modulus

Chapter 1

Introduction

1.1 CCM and PDEs

A *continuum* is an amount of matter whose physics can be investigated ignoring the discontinuities at microscopic length scales related to the molecular structure and molecular motions. When a fluid or a solid is approached as a continuum, we describe its behavior in terms of macroscopic properties, such as velocity, pressure, density, temperature and their space and time derivatives. These may be thought of as averages over suitably large numbers of molecules. A point in a continuum is then the smallest possible element of matter whose macroscopic properties are not influenced by individual molecules.

Computational Continuum Mechanics (CCM) is the science which investigates continua by means of numerical simulations. It includes, among others, *Computational Fluid Dynamics* (CFD) and *Computational Solid Mechanics*, the former related to the estimation of flow fields, the latter to the evaluation of stresses and strains acting on deformable bodies.

Partial Differential Equations (PDEs) are the obvious mathematical tools needed for modeling the equilibrium or the evolution in space and time of a continuum system. Moreover it is easy to see that even different phenomena, such as fluid dynamic and structural ones, may be classified in a unified way, thanks to similarities in their governing PDEs.

1.1.1 Fluid Dynamics

The governing equations for unsteady, Newtonian, compressible flows are the well–known set of the Navier–Stokes Equations. According to what is reported in many classic CFD books, (e.g. Ferziger and Perić, 2002; Patankar,

1980; Versteeg and Malalasekera, 1995), one may write:

$$\frac{\partial \rho}{\partial t} + \operatorname{div}(\rho \mathbf{v}) = 0 \tag{1.1}$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \operatorname{div}(\rho \mathbf{v} \otimes \mathbf{v}) = \operatorname{div} \boldsymbol{\sigma} + \rho \mathbf{b} \tag{1.2}$$

$$\frac{\partial \rho h}{\partial t} + \operatorname{div}(\rho h \mathbf{v}) = \operatorname{div}(k \operatorname{grad} T) + \mathbf{v} \cdot \operatorname{grad} p + \mathbf{S} : \operatorname{grad} \mathbf{v} + \frac{\partial p}{\partial t} \tag{1.3}$$

Equations (1.1), (1.2) express, respectively, the conservation of mass and momentum where $\mathbf{b}$ stands for all body forces, $\boldsymbol{\sigma}$ is the stress tensor:

$$\boldsymbol{\sigma} = -\left(p + \frac{2}{3}\mu \operatorname{div} \mathbf{v}\right) \mathbf{I} + 2\mu \boldsymbol{\varepsilon} \tag{1.4}$$

and $\boldsymbol{\varepsilon}$ is the rate of strain (deformation) tensor:

$$\boldsymbol{\varepsilon} = \frac{1}{2}\left[\operatorname{grad} \mathbf{v} + (\operatorname{grad} \mathbf{v})^T\right] \tag{1.5}$$

Equation (1.3) models the conservation of internal energy, where h is the enthalpy, T is the temperature, k is the thermal conductivity and $\mathbf{S}$ is the viscous part of the stress tensor:

$$\mathbf{S} = \boldsymbol{\sigma} + p\mathbf{I} \tag{1.6}$$

Moreover, ρ and h are linked to other thermodynamic variables p and T by means of the equations of state:

$$\rho = f_\rho(p, T) \tag{1.7}$$

$$h = f_h(p, T) \tag{1.8}$$

which eventually close the problem with the number of variables matching the number of unknowns.

By substituting equations (1.4) and (1.5) in (1.2), after some rearrangements we obtain the momentum equation in the compact form:

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \operatorname{div}(\rho \mathbf{v} \otimes \mathbf{v}) = -\operatorname{grad} p + \operatorname{div}(\mu \operatorname{grad} \mathbf{v}) + S_{\mathbf{v}} \tag{1.9}$$

where the main viscous effects are accounted for by the Laplacian term $\operatorname{div}(\mu \operatorname{grad} \mathbf{v})$ and the remaining viscosity–related contributions, together with body and external forces, are included in a single source term $S_{\mathbf{v}}$.

Similarly, the energy equation (1.3) can be rearranged as:

$$\frac{\partial \rho T}{\partial t} + \operatorname{div}(\rho T \mathbf{v}) = \operatorname{div}\left(\frac{\mu}{Pr} \operatorname{grad} T\right) + S_T \tag{1.10}$$

where Pr is the Prandtl number and S_T is a source term which collects all extra contributions.

The conservation laws for mass (1.1), momentum (1.9) and energy (1.10), for an unsteady, Newtonian, compressible flow can be summarized in a single generic conservation equation, often reported (e.g. Versteeg and Malalasekera, 1995) as *the transport equation for the generic scalar* ϕ :

$$\frac{\partial \rho \phi}{\partial t} + \operatorname{div} \rho \phi \mathbf{v} = \operatorname{div} \Gamma \operatorname{grad} \phi + S_\phi \tag{1.11}$$

.

In conclusion, the complex physics of such a flow can be modeled by a combination of three differential operators ($\partial/\partial t$, div, div(grad) = ∇^2) and a source term (S_ϕ), acting on scalar and vector fields (ϕ, $\mathbf{v}$), according to proper coefficients ($\mathbf{\Gamma}$) and source terms (S_ϕ).

1.1.2 Heat Transfer

Fourier's equation describes the unsteady thermal diffusion in continua and can be considered as a specialization of the generic conservation equation (1.11) in absence of convection.

$$\frac{\partial \rho c T}{\partial t} = \operatorname{div} k \operatorname{grad} T + S_T \tag{1.12}$$

where c is the specific heat and k, as before, the thermal conductivity.

1.1.3 Stress Analysis

As explained by Jasak and Weller (2000) the mathematical model for a linear elastic solid can be summarized as follows. The differential form of the force equilibrium for a solid body element states:

$$\frac{\partial^2 (\rho \mathbf{u})}{\partial t^2} = \operatorname{div} \boldsymbol{\sigma} + \rho \mathbf{b} \tag{1.13}$$

where $\mathbf{u}$ is the displacement, $\mathbf{b}$ is the body force and $\boldsymbol{\sigma}$ is the stress tensor. The strain tensor $\boldsymbol{\varepsilon}$ is defined in terms of $\mathbf{u}$:

$$\boldsymbol{\varepsilon} = \frac{1}{2}\left[grad\boldsymbol{u} + (grad\boldsymbol{u})^T\right] \tag{1.14}$$

The system of equations is closed by the Hooke's law, relating the stress and strain tensors:

$$\boldsymbol{\sigma} = 2\mu_l \boldsymbol{\varepsilon} + \lambda \operatorname{tr}(\boldsymbol{\varepsilon})\mathbf{I} \tag{1.15}$$

where μ_l and λ are Lamé's coefficients, related to Young's modulus of elasticity E and Poisson's ratio ν as

$$\mu_l = \frac{E}{2(1+\nu)} \tag{1.16}$$

and

$$\lambda = \begin{cases} \dfrac{\nu E}{(1+\nu)(1-\nu)} & \text{for plane stress} \\ \dfrac{\nu E}{(1+\nu)(1-2\nu)} & \text{for plain strain and 3–D} \end{cases} \tag{1.17}$$

By substituting (1.15) and (1.14), the governing equation (1.13) for linear elastic solids can be rewritten with the displacement $\mathbf{u}$ as a primitive variable:

$$\frac{\partial^2(\rho \mathbf{u})}{\partial t^2} = \operatorname{div}\left[\mu \operatorname{grad} \boldsymbol{u} + \mu(\operatorname{grad} \boldsymbol{u})^T + \lambda \boldsymbol{I} \operatorname{tr}(\operatorname{grad} \boldsymbol{u})\right] = \rho \boldsymbol{f} \tag{1.18}$$

In summary, also the stress analysis of a continuum, although under the simple hypothesis of infinitesimal deformation and linear, elastic body, is modeled by means of a PDE. A reduced set of differential operators, such as $\partial^2/\partial t^2$, grad, div, acts on the field $\boldsymbol{u}$, according to coefficients λ and μ_l provided by constitutive relationships.

1.2 Numerical Methods for Solving PDEs

Once a physical phenomenon has been mathematically described by means of a PDE, one has to apply a specific method in order to find a numerical solution of that problem. Many methods have been developed so far, each having its peculiarities and preferred applications fields. In order to cite a variety one might use:

- Finite Difference Methods (FDM) (e.g. Hoffmann and Chiang, 2000; Tannehill et al., 1997);
- Finite Elements Methods (FEM) (e.g. Zienkiewicz et al., 2005; Löhner, 2001);

- Finite Volume Methods (FVM) (e.g. Patankar, 1980; Versteeg and Malalasekera, 1995);
- Spectral Methods (e.g. Gottlieb and Orszag, 1977).

As explained by Versteeg and Malalasekera (1995), every method performs three basic steps:

1. approximation of the unknown variables by means of simple functions;
2. discretization by substitution of the approximations into the governing equations and subsequent mathematical manipulations;
3. solution of the algebraic equations derived in the previous step.

The main differences between the four schemes above are associated with the way the variables are approximated in the discretization process. With particular reference to the Finite Volume Method, the most important method applied in CFD and implemented in many commercial codes such as FLUENT, STAR, FIRE, etc., we could recognize the following steps:

- formal integration of the governing PDEs over all control volumes (CV) into which the solution domain has been subdivided;
- possible application of Gauss' theorem for converting volume integrals into surface integrals;
- discretization in the integral equations of the terms representing flow processes, such as convection, diffusion and sources, by means of a variety of finite–difference–type approximations. This converts the integral equations into a system of algebraic equations.
- solution of the algebraic equations by means of an iterative method; implicit phases of the main algorithm involves the solution of linear systems with sparse matrix coefficients.

In summary, the process of modeling a physical phenomenon and solving it by means of one of the methods above, can be represented by the following problem sequence:

1. **Physical Problem**: a phenomenon of interest to a scientist or an engineer. In our case involves the modification of the state of continua.
2. **Mathematical Problem**: the modeling by means of PDEs with proper boundary and initial conditions.

3. **Numerical Problem**: the application of the method which converts the PDE(s) into algebraic equation(s), i.e. FDM, FEM, FVM, etc.

4. **Programming Problem**: implementation of the chosen numerical method, according to specific paradigm(s) (serial or parallel, procedural or object–oriented), using a proper programming language (C, C++, Fortran 95, etc.).

1.3 CCM and Object–Oriented Programming

1.3.1 Interface vs. Implementation

Consider the design of a brand new CCM code for solving PDEs applied to continua, not restricted to the specific field of CFD, but rather suitable, for instance, in structural analysis.

One of the first things that the developer should be aware of is that there are two completely distinct layers of interest:

1. user's layer;

2. developer's layer.

The user will claim the possibility of solving any PDEs, by simply "building" the equation(s) governing a particular problem by means of high level programming tools, which could reproduce the sequence of differential operators applied to some fields, as written in the PDE itself.

The developer instead should care that behind a user–friendly interface, the code consists of a modular implementation of the chosen solution method, better hiding any detail somehow related to the specific discretization scheme or algorithm in use.

This approach, i.e. the *separation of interface from implementation* has many advantages:

- the user does not have to care about too many implementation details;

- if the implementation is completely *transparent* for the user, and the interfaces are kept unchanged, the developer may apply any modification to the source code, without affecting the user's applications;

- depending on the current specific case, it would be possible to activate the best numerical method for that particular problem (e.g. finite volume for CFD, finite element for CSM), always using the same interface.

This separation reflects also the solution procedure for PDEs in CCM codes: there is a 1–1 correspondence between each differential operator applied to a variable and the contribution of this term to the algebraic equation which derives from the discretization of the PDE. The border line between differential operators (in the PDE) and algebraic equations (in the numerical method) should correspond to the separation between the user's and developer's layers, as well as between interface and implementation.

1.3.2 Procedural vs. OO Programming

Classic CFD applications, such as the well known KIVA code (see Amsden et al., 1989; Amsden, 1993, 1997), are based on a *procedural* approach, consisting of the *cascade* of calls to procedures devoted to a specific task. The main advantage of this programming style consists in developing applications which reproduce the flowchart of the chosen algorithm and its solution formulas.[1] Moreover a procedural application is supposed to be designed for a specific task, and hopefully should be optimized for that goal. Modularity and re–usability are not issues in a procedural program, because maybe it is often easier and faster, for a different application, to develop a new code starting from the scratch.

On the other hand, the most popular paradigm in modern software engineering (SE) is the *Object–Oriented Programming* (OOP), which is based on the concept of a *class*, defined as a *model* of all *objects* which share the same attributes, operations, relationships and meanings.

With particular reference to the solution of PDEs in CCM, one could think of a Pde as a class, which interacts with the objects of the class Field, modeling the behavior of a continuum, according to instances of the class BoundaryCondition. Differential operators could be implemented as procedures which modify the content of a Pde object, hiding the translation of the corresponding PDE term in the discretized algebraic equation behind a high level interface.

Procedural programming, which characterizes Fortran 77 scientific applications, is focused on the *actions*, while OOP is centered on *data*. Although this simplification could seem too strong, it is true that the OO approach starts from the organization of groups of data, and requires the definition of what they contain, how can they constructed, destroyed, accessed and modified.

In the past, massive scientific computing[2] had being accomplished only by

[1] The name FORTRAN comes from FORmula TRANslator, an explicit reference to its suitability for implementing numerical procedures.

[2] also known as "number crunching"

means of procedural programming languages such as Fortran, which ensured to the careful developer, highly efficient codes. Nowadays needs have changed and the necessity for developing multipurpose applications, improving the re–usability and the flexibility of the codes, is requiring the employment of OOP also in high performance computation. However, it is better to state right now two considerations, which should partially redefine the capabilities usually attributed to the OO practice.

First of all, OOP is not the solution for every kind of numerical application; moreover the gain in terms of flexibility has its own price in terms of pure performance. That is why a good developer should be "moderately object–oriented", differentiating the areas of his code where he may use OO abstraction and where, instead, it is better to prioritize the computational efficiency using a procedural approach.

Second, OOP is only the last step of a bigger approach for the general analysis of complex systems, such as a CCM code; in fact, as explained by Booch (1993), it should be better to refer to *Object–Oriented Techniques*, which include:

1. Object–Oriented Analysis (OOA)

2. Object–Oriented Design (OOD)

3. Object–Oriented Programming (OOP)

It is the set of all these three phases which eventually leads to advantages in mastering the complexity of large scale projects. Similarly, the choice of a language which fully supports object–oriented features is useless without a sound construction of data structures and operators.

1.4 Subject of the Present Work: NEMO

1.4.1 Motivations for a New CCM Code

Originally this project was born with the goal of implementing a parallel CFD code with hybrid meshes, able to replace in a few years the KIVA code, used for complex internal combustion engine simulations. The aim was to develop a procedural code, highly efficient thanks to a parallel framework suitable for running on distributed memory systems (e.g. Linux clusters), written in Fortran 95, implementing many of the best CFD algorithms available in literature.

However, in the last few years the experience of the OpenFOAM[3] code (see Weller et al., 1998; Jasak and Weller, 2000; Jasak et al., 2004) has clearly showed that the usage of Object–Oriented techniques could give a significant advantage in the development of large scale, CCM applications.

Being confident about the object–oriented capabilities of Fortran 95[4], and the fact that this language seems still to guarantee better performances for scientific computing, we decided to develop a new application merging the flexibility of OpenFOAM together with the efficiency of Fortran 95; that is a parallel Object–Oriented Code for Computational Continuum Mechanics written in Fortran 95. Its name is **NEMO** which stands for ***N****umerical* ***E****ngine for* ***M****ultiphysics* ***O****perators*.

1.4.2 NEMO: Main Features

The main features of NEMO, which will be widely exposed in the remaining chapters, are:

- object–oriented framework for the solution of CCM–related PDEs;
- written in standard Fortran 95;
- finite volume approach;
- support for hybrid meshes with generic polyhedral cells;
- co–located (cell–centered) arrangement of unknowns;
- discretization schemes of second order accuracy;
- message–passing parallelism for distributed memory systems;
- efficient set of parallel preconditioners and solvers;
- high portability on Unix–like platforms.

1.5 Structure of the Work

This PhD thesis is structured according to the following subdivisions. Each chapter is introduced with a brief overview of its contents.

[3] `http://www.opencfd.co.uk/openfoam`

[4] In chapter 2 it is explained why Fortran 95 can be considered an OO language.

Chapter 1 : preliminary introduction to continua modeling by means of PDEs and to OOP applied to CCM; aim of the work: NEMO, Multiphysics PArallel Computational Tool.

Chapter 2 : basic concepts of object–oriented techniques and discussion about how Fortran 95 may be used for developing object–oriented applications.

Chapter 3 : description of the PSBLAS library, numerical kernel of NEMO, which provides a parallel implementation of powerful preconditioners and iterative methods for solving sparse linear systems.

Chapter 4 : partitioning strategies for decomposing data between the processes of the parallel job and reordering algorithms for reducing the bandwidth of the sparse matrix associated to a computational mesh.

Chapter 5 : mesh capabilities and object–oriented formulation for topology and geometry–related data structures.

Chapter 6 : analysis of the object–oriented framework of the PDE solver; description of the most important classes representative of fields, PDEs and boundary conditions. Definition of differential operators acting respectively on fields and on PDEs.

Chapter 7 : conclusions, current status and future work.

Chapter 2

OO Programming in Fortran 95

2.1 Object–Oriented Analysis and Design

As explained by Booch (1993) and previously mentioned in Sect. 1.3.2, an object–oriented formulation consists first of all in modeling a system by identifying its basic elements and the roles they play. This is achieved by means of Object–Oriented Analysis (OOA) and Object–Oriented Design (OOD), whose main steps are listed respectively in Tabs. 2.1 and 2.2 (Akin, 2003).

Only after proper Object–Oriented Analysis (OOA) and Design (OOD), the developer should come to the final step, that is, the choice of a specific programming language and the implementation of a particular approach to the problem solution.

2.2 Object–Oriented Languages for Scientific Computing

2.2.1 Required Features

The programming language chosen for supporting an object–oriented formulation should satisfy the following requisites:

Data abstraction: ability to represent conceptual constructs in the program, using new, user defined, data types, *hiding* implementation details behind an interface. Such an entity is an *Abstract Data Type* (ADT); a *class* is obtained as an extension of an ADT by providing additional operations which serve as *constructors* or *destructors*.

Encapsulation: ability to contain and *protect* the data that make up the *object*. This is usually achieved by assigning *private* access to class

Find classes and objects: • Create an abstraction of the problem domain. • Give classes and objects meaningful names. • Identify structures pertinent to the system's complexity and responsibilities. • Observe I/O with the systems as well as information to be stored. • Look for reuse: are there multiple structures? Can subsystems be inherited?
Define the attributes: • Give attributes meaningful names. • Describe the attribute and any constraint. • What knowledge does it posses or communicate? • Put it in the type or class that best describes it. • Select accessibility as public or private. • Identify the default lower and upper bounds. • Identify the different states it may hold. • Note items that can either be stored or recomputed.
Define the behavior: • Give behaviors meaningful names. • What questions should each behavior be able to answer? • What services should it provide? • Which attributes should it access? • Select accessibility as public or private. • Define its interface prototype. • Identify a default constructor with error checking. • Identify a default destructor.
Diagram the system: • Employ an OO graphical representation such as UML (Fowler and Scott, 2003).

Table 2.1 Object–Oriented Analysis (OOA) summary.

- Improve the OOA during OOD.
- Divide the member functions into constructors, accessors, agents and servers.
- Design the human interaction components.
- Design the task management components.
- Design the data management components.
- Identify operators to be overloaded.
- Identify operators to be defined.
- Design the interface prototypes for member functions and for operators.
- Design code for reuse through *"is–a"*, *"has–a"* hierarchies.
- Identify base classes from which other classes are derived.
- Establish the exception–handling procedures for possible errors.

Table 2.2 Object–Oriented Design (OOD) summary.

members, permitting their manipulation only through a well–defined *public* interface.

Inheritance: ability to define new classes by extending (or limiting) attributes and operations of existing ones, achieving code re–usability and semantic relationships of "*is–a*" type.

Polymorphism: ability to provide the same interface to objects of different types, thus obtaining a conceptual equivalence between classes with distinct implementation.

Operator overloading: ability to assign natural syntax (e.g. algebraic operations) to newly defined classes. Although not properly a requirement for object–orientation, it can help in increasing the level of abstraction.

Besides this, one should account also for the following constraints:

- computational efficiency;
- debugging and maintaining possibilities;
- cross–platform portability.

Usually, it is impossible to find a programming language which provides all the above requirements. Therefore the final choice of the developer will be an unavoidable compromise among them.

For instance, if we had to implement a graphical user interface (GUI), our choice would be Java, or a scripting language such as Python or Perl. Their high level of abstraction permits a relatively easy access to graphical libraries, which would be dramatically complicated, practically impossible, in Fortran 95, developed and optimized in more than forty years for the specific target of scientific applications.

On the other hand, one could try to implement a complex CFD code in Java, using its powerful object–orientation, but the result would be ridiculous in terms of performance when compared to a corresponding Fortran program.

When we have to deal with CCM, the efficiency is a primary requisite; this constraint restricts the group of OO languages suitable for *High Performance Computation* (HPC) only to: C++ and Fortran 95.

2.2.2 C++

The leading, most popular, object–oriented language is C++ (Stroustrup, 2000). Written as a "better C", it provides not only the support for object–orientation and *generic programming*, but also offers full portability, free availability[1], and the efficiency needed for scientific computation.

Actually, the argument continues whether C++ could ever reach the performances of Fortran, which has been constantly improved in the last four decades. Some authors, like Decyk et al. (1997) and Norton et al. (1995), state that when they wrote the same application in both languages, the Fortran version was always faster than the C++ one. Others, like Weller et al. (1998), Cary et al. (1997) and Dubois-Pèlerin and Zimmermann (1993), claim that presently there are no significant differences in efficiency between Fortran and C group languages.

Another issue is that the great flexibility in providing full object–orientation has its counterpart in longer debug sessions for codes written in C++ than in Fortran 95. One reason is that Fortran 95 is more restricted in its features and more strict in its type checking (no automatic conversion across arguments, for example).

One should also remark that while Fortran has been designed expressly for scientific computing, C++ has been developed for more general purposes. For example, Fortran 95 has powerful array classes which must be explicitly implemented in C++.

According to C++ supporters, its full support for object–orientation and the improvements in compilers performances achieved in the last years take over possible deficiencies, so that C++ becomes the natural, ideal, candidate

[1] by means of the `gcc` compilers suite: see `http://www.gcc.org`

language for developing object–oriented CCM applications. The worldwide success of OpenFOAM code confirms this fact.

2.2.3 Fortran 95

According to Metcalf et al. (2004) Fortran 90 is a major revision of Fortran 77; Fortran 95 is a minor revision, which has been further integrated with official extensions.[2]

The common opinion shared by many developers is that Fortran 90/95 is simply a "*Fortran 77 with dynamic memory allocation*", actually referring to its most obvious change with respect to the older dialect. A relatively few people know that Fortran 95, actually, is a *potential* object–oriented language, thanks to many other features, which often are unknown or heavily underestimated by the majority of programmers .

The most frequent statement about Fortran from C++ developers is: "Fortran 90 is NOT an object–oriented language". But let's try, first of all, to find a fair definition for an OO language. In fact if it means "*It does everything that C++ does*" one should draw the absurd conclusion that "*Java is NOT object–oriented*" because it has no pointers![3].

A better definition could be the one suggested by Stroustrup (2000), which considers a language as object–oriented if it supports:

- abstraction (encapsulation);
- inheritance;
- polymorphism.

According to these requirements Fortran 77 was, of course, definitely not object–oriented, while Fortran 90/95 is, because:

- it provides full support to data abstraction, by means of derived data type definition and private/public access of variables and procedures defined in modules;
- it implements inheritance of data types and permits "has–a" relationships between classes;
- it provides with *static* polymorphism by means of generic interfaces.

[2]See also `http://www.j3-fortran.org`. J3 (former X3J3) is the committee responsible for Fortran standards.

[3]In the way they're defined in C++

These features are enough for considering Fortran 95 as a *potential* OO language. In the last decades many authors have begun to rewrite their C++ scientific applications in Fortran 95, obtaining nice results in terms of data abstraction, and often improving the overall efficiency. This has generated new attention toward Fortran 95's capabilities and the reconsideration of what is really the best OO language for scientific computing.

It is obvious that the past trend is in favor of C++, exactly as it is in favor of Fortran with regards to performance issues, but authors like Akin (2003), Norton et al. (1995), Decyk et al. (1997, 1998) [4] and Machiels and Deville (1997) demonstrate an advanced use of Fortran 95 as a software tool suitable for OOP.

In particular, Decyk et al. (1998) report how to implement the concepts of *inheritance*, by means of delegation of attributes and operations of a class, and how to realize *run–time polymorphism*[5]. The latter is often remarked as a lack of Fortran 90/95 with respect to C++, but, as explained by the authors, it can be *emulated* by using pointers. Moreover, the experience of developing NEMO showed that the circumstances where this feature is *absolutely* necessary and cannot be worked around, in order to preserve the abstraction level, are rare.

The lack of some OO features is compensated by the better performance and computational efficiency that Fortran 95 *seems* to still guarantee against other languages used in scientific field.

For this reason, the goal of the present work is to develop a new CCM code that aggressively utilizes the Fortran 95 capabilities for object–orientation. NEMO will hopefully have similar flexibility and modularity to OpenFOAM and even better performance. The initial experiences with Fortran 95 are very encouraging. A further speed–up of the code will be possible when it will benefit also from the official ISO/IEC 15581 Fortran 95 extensions, related to the use of allocatable arrays in derived data type declaration and dummy arguments. They are way more urgent than the compilers support for some sophisticated object–oriented features which tries, maybe in vain, to replicate C++ abilities.

2.2.4 Fortran 2003

During the last five years the Fortran standard passed through another major revision, even bigger than the transition from Fortran 77 to Fortran 90: the final result is Fortran 2003.

[4] `http://www.cs.rpi.edu/~szymansk/oof90.html`

[5] Also known as *dynamic dispatching* or *dynamic binding*

This is, of course, a superset of Fortran 95, and backward compatibility towards legacy Fortran 77 codes is guaranteed, except that some features, considered obsolescent in the standard 95, are now illegal. The list of new features covers different areas like:

- floating–point exception handling;
- allocatable array extensions;
- enhanced module facilities;
- interoperability with C;
- type parameters and procedure pointers;
- object–oriented programming;
- establishing and moving data;
- input/output enhancements;

A detailed review of all new capabilities of Fortran 2003 is reported by Metcalf et al. (2004) and Reid (2003).

The first two entries in the list above are actually official Fortran 95 *extensions*. However their support is not guaranteed by all compilers, therefore it is likely that they will have absolute priority for accomplishing the migration to standard 2003.

Obviously the most interesting feature for our application are the capabilities directly or indirectly related to object–oriented programming, like: differentiation between *modules* and *submodules*, *procedures pointers*, support of type–extension for *dynamic inheritance*, *runtime polymorphism*, etc.

With these powerful tools, there would no longer be a need for emulation of inheritance and dynamic binding; therefore, according to the definition provided in Sect. 2.2.3, Fortran 2003 is a fully object–oriented programming language. There is still no support for *templates* though, in contrast to C++.

Currently only a reduced set of standard 2003 features are supported in the most advanced commercial Fortran compilers and the majority of them are not related to the enhanced OO capabilities. Reasonably, all inclusive Fortran 2003 compilers will not be ready in less than five years; therefore the development of OO applications must rely on the capabilities of Fortran 95, trying whenever possible to work around its limitations, and never forgetting that Fortran remains a language for scientific, high performance, computing.

2.3 Fortran 95 Compilers

NEMO has been, so far, successfully tested with many different Fortran 95 compilers for Unix–like platforms, such as: Intel's ifort 9.0, PGI's pgf95 6.0, Pathscsale's pathf95 2.2.1 and GCC's gfortran 4.2.

Until year 2004, C++ would have held an advantage over Fortran in the choice of the best language for developing a new application such as NEMO. In fact, at that time there was no stable Fortran 95 compiler, available for free, while for C++ one could use the well known cpp from gcc suite[6].

In the last two years (2005–2006) gfortran[7], the new Fortran 95 compiler included in GCC suite, has been constantly improved by a group of volunteers, and tested by the Open–Source Community spread all around the world. Although it is still a beta release, it compensates, at last, for historical lack in the scenario of free–available compilers. Moreover gfortran, as well as any other compilers in GCC suite, is cross–platform: this ensures a *full portability* of all applications which support gfortran.

That's also the reason why gfortran was chosen as a reference compiler, instead of other, maybe more advanced and faster compilers, which, for instance, already implement Fortran 95 extensions for allocatable arrays or even have some features of the standard 2003: this is the only way for allowing that everyone in the future may build NEMO on their platform, with no additional costs for the Fortran compiler.

Last but not least, we've already encountered some circumstances where gfortran acts even better than some commercial competitors, in terms of computational efficiency. In this way the user will always have the possibility to choose between commercial compilers which actually give peak performance and a free one that is functional.

2.4 UML – Unified Modeling Language

The development of NEMO has been accomplished using many tools derived from Software Engineering (SE) and specifically designed for large–scale, object–oriented applications: among them the UML (Unified Modeling Language) is worth a few words more.

As explained by Fowler and Scott (2003) UML consists in a set of well–defined diagrams and graphical elements used for modeling and describing complex systems such as a CCM code. In particular, it is extremely appropriate for representing classes and relationships between the building blocks

[6] `http://gcc.gnu.org/`
[7] `http://gcc.gnu.org/fortran/`

of an OO application. The main reason which led to its definition was, in extreme synthesis, the need for a *standard* visual language for depicting the framework of a code.

During the 90's many different methodologies for analysis and design of software systems were introduced in the Information Technology (IT), however each one had its specific set of notations, conventions and symbols and sometimes they differ significantly from each other. In particular, the most popular were the ones developed respectively by Jim Rumbaugh, Grady Booch and Ivar Jacobson.

Starting from 1995, once it was clear that a "methods war" was intolerable, the three authors, which in the meanwhile had joined Rational[8], now an IBM subsidiary, began to work together on the draft of UML, a new visual language specifically designed in order to replace the previous three, hopefully eliminating their respective drawbacks and weak points. In 1997 UML became a standard after the official approval of the OMG[9] (Object Management Group) and during the last decade its usage has spread very fast. As reported by Quatrani (2003), *"the UML is the standard language for specifying, visualizing, constructing, and documenting all the artifacts of a software system"* ; in other words UML diagrams can be considered as *blueprints* for Software Engineering.

During the development of NEMO their usage has been very helpful in designing the sections of the code, in particular classes and interfaces, involved in the different layers of the PDE solver. By assigning in advance proper attributes and operations to the classes, it has been possible to conceive the whole framework of the code, to anticipate possible deficiencies or inconsistency in the data structures, so that the final result has been a clearer comprehension of the whole system together with an accurate view of the interfaces and inter–relationships. Once this has been set, the main goal of the developer has been to write the implementations according to those prototypes, the whole process becoming faster and with a better final quality.

Nowadays there are many software tools specifically designed for drawing UML diagrams, extracting the corresponding code, or vice versa produce the diagram starting from available sources. Of course, this capability for "direct" and "indirect" engineering have been developed so far only for Object–Oriented languages such as C++ or Java. In our case, UML resources are valuable for OOA, OOD and last but not least for documentation purposes: if the diagrams are consistent with the standard notations, any developer which

[8] `http://www-306.ibm.com/software/rational/uml`
[9] `http://www.omg.org`

knows its syntax and symbols can "read" and understand the representation of the code without risk of misunderstandings.

The UML standard consists in seven different types of diagrams, each focused on a particular aspect of classes, objects and other components of a system. In the next chapters just one kind of diagram will be used, that is *"Class Diagrams"*, which have been conceived for representing the internal structure of a class (i.e. its *members* or *attributes*), how it is possible to create, destroy its instances, and access its content (i.e. the set of class *operations* or *methods*) and the relationships (if they exist) of *dependency*, *composition*, *aggregation*, *realization* and *generalization* involving more classes. These concepts will be reviewed and properly explained at their appearance in the next chapters of the thesis; for a complete description of all UML diagrams and symbols the interested reader may consult the book of Fowler and Scott (2003).

2.5 Abstraction vs. Efficiency

As explained in Sect. 2.2.2, it is possible to find in scientific literature some authors which have written their object–oriented application using both Fortran 95 and C++, in order to understand advantages and drawbacks of the two programming languages. From their results the better capabilities of C++ in terms of data abstraction are undoubtable, but the shorter execution time of the Fortran implementation highlights one basic concept: there is a trade–off between abstraction capability and computational efficiency.

This issue is particular clear when a developer has to choose the access policy of a class, that is whether its content is *public* and may be manipulated directly at any circumstance or instead must be kept *private*, hiding it behind an interface.

The former method (i.e. public access) is more direct and hence more efficient because the data are soon available for being elaborated in computational sections, while the latter (i.e private access) involves the use of *accessors*, (*setters* for assigning values to class members, *getters* for extracting information from an object) but separates interface from implementation, confining *that* specific programming choice in a restricted and well–defined area of the code, allowing easier changes to the class framework (if needed), and minimizing the need for knowledge about the source's details on the user's side.

The usage of getters, however, implies an additional overhead related to the call of the procedure which can be particularly heavy and slow down dramatically the computation if the object is accessed repeatedly, for example

when it is instantiated inside a *do loop* with an extremely high iteration rate. Another reason why private access can harm the performance is that the content of class attributes must be copied out of the object, manipulated, and then copied in back, with two obvious consequences: the longer execution time related to the slowness of the copy operations and the greater memory needed for the temporary buffer which stores the data outside the object during the computation.

Since all these effects can be amplified, depending on the specific application, a careful developer should evaluate whether to privilege the abstraction on a case by case basis, maybe reproducing on a smaller scale the same situations.

The use of private access rules the practice of object–oriented programming in C++, mainly because of an "*historical*" background that is definitely in favor of data abstraction. When somebody highlights that getter functions imply a remarkable overhead the most common answer is:"It can be eliminated by forcing the compiler to *inline* the procedures." This is a sort of misconception about the capabilities of *Inlining*, the incorporation of the called subprograms in the calling unit. In fact, as explained in Norton et al. (1995), C++ programmers should note that compilers are *free to ignore* the `inline` directive, which is therefore *indicative* but not *prescriptive*. This implies that basing the efficiency of a code on the inlining capabilities of the compiler could produce very heterogeneous results among different platforms and often an overestimation of the code potential efficiency. Moreover in a fair approach to the optimization problem, first one should try to have better code design and only on a second level get the best from the available hardware and the compiler features.

In order to maximize the data abstraction without harming the overall efficiency too much, the particular strategy chosen for NEMO is a mix between public and private access. This can be summarized as follows:

- Whenever possible, private access is preferred to public in order to separate interface from implementation and confine class internal details in the corresponding module.

- Data hiding, encapsulation, and other OO practices, are used especially at high level, in order to enhance the abstraction.

- At low levels intrinsic types are preferred to abstract data types, public access is better than private, in order to maintain a good computational efficiency.

- Class attributes are kept private if there is really an advantage in doing

it, otherwise it is contradictory to hide a datum and extract it "*as it is*", only in order to be OO at any cost.

- If there is no need for private access, because, for instance, there is just one plausible implementation of that class, it is better to have a simpler and cleaner code and use public access, even if it is at high level.

- If the operations which act on a class are particularly long, it could be convenient to implement them in distinct procedures, out of the main class module. This requires that the access is kept public, in order to permit the manipulation in the external subprogram. With Fortran 2003 it would be possible to declare the members as private and collect the related operations in different "submodules"[10].

2.6 Example: The Connectivity Class

In this section, the implementation of the Connectivity class is considered, in order to give the reader a first example of OOP using Fortran 95 and the consequence of choosing either a public or a private access.

As will be explained in Sec. 5.1.2, the topology of an unstructured mesh is described by means of connectivity relationships between different element types, such as vertices, faces, cells. For example, one should know which vertices are assigned to a specific cell, that is the *vertex–to–cell* connectivity, also tagged as v2c.

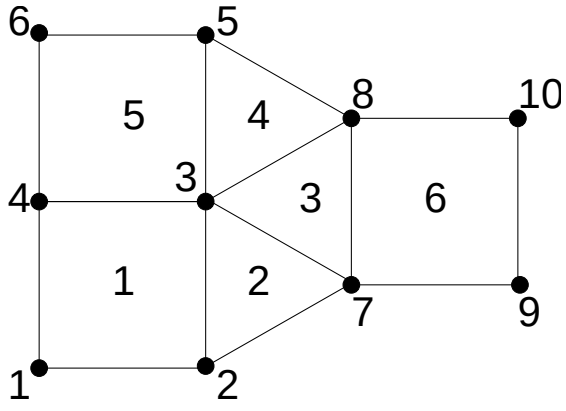


Figure 2.1 Vertex and cell numbering of a 2D hybrid mesh.

[10]See *modules* and *submodules* in Metcalf et al. (2004)

With explicit reference to the simple mesh depicted in Fig. 2.1, fixed an arbitrary numbering for vertices and cells, one can build the connectivities reported in Tab. 2.3.

Cell	Vertices
1	1 2 3 4
2	2 7 3
3	3 7 8
4	8 5 3
5	6 4 3 5
6	7 9 10 8

Table 2.3 Vertex–to–Cell connectivity of the mesh depicted in Fig. 2.1.

The problem can be generalized by considering two arrays of indices, let's say `a` and `b`. The i–th element of `b`, namely `b(i)`, is linked by a particular association to a set of elements of `a`, namely `a(j)`, where the subscript `j`, in general, does not assume consecutive values. Thus one has to define the `a2b` connectivity.

This kind of relationship can be implemented in different ways, for instance using either a *CSR*–like storage (Compressed Sparse Row format, see Saad (2003) and Karypis and Kumar (1998)) or a *linked list* (e.g. Akin, 2003). Let's examine the former method and implement two versions of the corresponding class: ConnPub, with *public* access, and ConnPri, with *private* access. The UML diagram for both of them is depicted in Fig. 2.2.

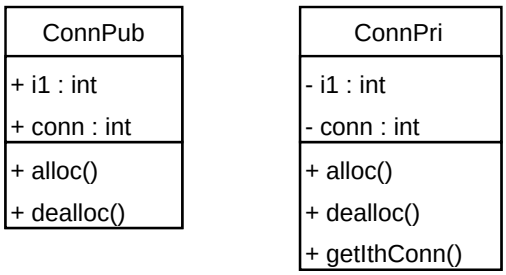


Figure 2.2 UML diagram of the ConnPub and ConnPri classes.

The two classes consist of a new data structure made of two arrays `i1` and `conn`. By referring again to the example of Fig. 2.1 and Tab. 2.3, one can stick together, sequentially, all vertex–to–cell connectivities, and store them in the `v2c%conn` array. In order to get the list of vertices which build

the i^{th} cell, one has hust to know where its first element is stored . This information is supplied by the `v2c%i1` array, such that:

- `i1 = v2c%i1(i)` points to the first vertex of the i^{th} cell;
- `i2 = v2c%i1(i+1) - 1` points to the last vertex of the i^{th} cell;
- `v2c%conn(i1:i2)` contains the desired set of vertices;
- `v2c%i1(ncells+1) - 1` is equal to the total number of connectivities.

Figure 2.3 represents the CSR–like data structure for v2c connectivity object.

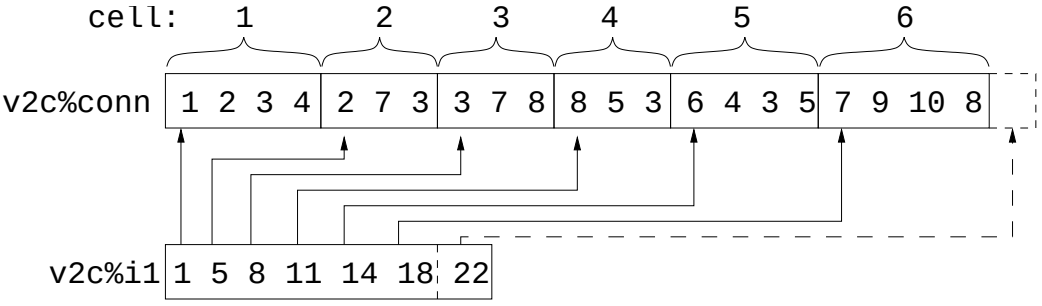


Figure 2.3 CSR–like implementation of v2c connectivity.

The operations `alloc` and `dealloc` serve as *constructor* and *destructor*, simply by allocating and deallocating the memory space for arrays `i1` and `conn`.

The main differences between the two implementations consist in the visibility of class attributes `i1` and `conn`, which are public (UML notation "+") in ConnPub and private (UML notation "−") in ConnPri, and in the presence of the `getIthConn` *getter* procedure in the latter class, needed for extracting the indices sequence for the generic element of structure `b`.

The Fortran 95 sources of the two classes are listed in Figs. 2.4 and 2.5–2.6 respectively. [11] Each class consists in a module: the class itself is defined in the upper section as a new Derived Data Type (DDT), whose components are the class members. In the ConnPri class private access is enforced by specifying the `private` attribute, right after the `Type ConnPri` statement, prohibiting procedures not included in the module from manipulating the content of an object of that class. The lower section of the module contains the class *operations*. Generic interfaces for the module procedures are specified in the declarative part, so that the same name may be used

[11] The implementation of the ConnPri class has been split in two figures for readability's sake.

for different subroutines or functions having distinct argument lists. This features provides what has been called in Sect. 2.2.3 *static polymorphism*.

The method `getIthConn` in the class ConnPri is of particular interest: instead of allocating new memory for the array `ithConn`, whose size can vary depending on the number of connectivities of the element `b(i)`, and copy out the content of the corresponding fragment of `a2b%conn(i1:i2)`, the class member is accessed directly by means of a *pointer*. This approach has three advantages:

- it does not requires additional memory storage;
- pointer dereferencing is faster than memory allocation/deallocation;
- it avoids copy operations, which are quite expensive;

Table 2.4 shows the results of a simple speed test consisting of accessing randomly the data stored in two ConnPub and ConnPri objects; the number of events has been set to 10 million. The private version has been tested with two variants of the getter method, reported in Fig. 2.6:

1. `getIthConn_a`: dynamic memory allocation and local copying;
2. `getIthConn_p`: pointer dereferencing.

Of course private access is slower than public, but it allows the code to confine the implementation details of the class ConnPri inside the module `class_conn_pri`, while the usage of ConnPub implies that everywhere an object of this class has to be accessed, one must use expressions such as `a2b%conn`, or `a2b%i1` which is specific to *that* programming choice.

Class	Access Type	ΔTime[%]
ConnPub	Public	0%
ConnPri	Private + Allocation	+330%
ConnPri	Private + Pointing	+26%

Table 2.4 Public vs. Private access of the Connectivity class.

If, in the future, one decides to change the implementation of the ConnPub, for instance using linked lists instead of CSR–like format, then he will have to revisit and change all parts where the class members have been explicitly manipulated.

On the contrary the class ConnPri forces the user to access connectivity data only by means of the the *public operations*, such as `getIthConn`, fixed

```
module class_conn_pub

  type ConnPub
     integer, pointer :: i1(:) => null()
     integer, pointer :: val(:) => null()
  end type ConnPub

  interface alloc
     module procedure allocConnPub
  end interface

  interface dealloc
     module procedure deallocConnPub
  end interface

contains

  subroutine allocConnPub(a2b, nel, nconn)
    type(ConnPub), intent(inout) :: a2b
    integer, intent(in) :: nel, nconn
    integer :: info

    allocate(a2b%i1(nel+1),a2b%conn(nconn),stat=info)
    call checkError(info)

  end subroutine allocConnPub

  subroutine deallocConnPub(a2b)
    type(ConnPub), intent(inout) :: a2b
    integer :: info

    deallocate(a2b%i1,a2b%val,stat=info)
    call checkError(info)

  end subroutine deallocConnPub

end module class_conn_pub
```

Figure 2.4 Source code of the ConnPub class.

```
module class_conn_pri

  type ConnPri
     private ! Private Access
     integer, pointer :: i1(:) => null()
     integer, pointer :: val(:) => null()
  end type ConnPri

  interface alloc
     module procedure allocConnPri
  end interface

  interface dealloc
     module procedure deallocConnPri
  end interface

contains

  ! Operations...

end module class_conn_pri
```

Figure 2.5 Source code of the ConnPri class: attributes.

```
module class_conn_pri

  ! Attributes...

contains

  ! Subroutines allocConnPri and deallocConnPri are equal
  ! to their respective public counterparts.

  subroutine getIthConn_a(ithConn,a2b,i)
    integer, pointer :: ithConn(:)
    type(Connectivity), intent(in) :: a2b
    integer, intent(in) :: i
    integer :: i1, i2, n

    if(associated(ithConn)) deallocate(ithConn)
    i1 = a2b%i1(i)
    i2 = a2b%i1(i+1) - 1
    n = i2 - i1 + 1
    allocate(ithConn(n))        ! Allocation
    ithConn(:) = a2b%val(i1:i2) ! Copy

  end subroutine getIthConn_a

  subroutine getIthConn_p(ithConn,a2b,i)
    integer, pointer :: ithConn(:)
    type(Connectivity), intent(in) :: a2b
    integer, intent(in) :: i
    integer :: i1, i2

    i1 = a2b%i1(i)
    i2 = a2b%i1(i+1) - 1
    ithConn => a2b%conn(i1:i2) ! Pointer dereferencing

  end subroutine getIthConn_p

end module class_conn_pri
```

Figure 2.6 Source code of the ConnPri class: operations.

in the class prototype. If their interfaces are kept unchanged, possible future modification to the internal structure of the class can be managed without affecting other sections of the code out of `class_conn_pri`, with an enormous saving in terms of maintaining and development efforts.

Moreover Tab. 2.4 shows clearly that the private solution with pointer dereferencing works much faster than the one with additional memory allocation followed by copying, and how a proper use of Fortran 95 tools, such as pointers in this case, can provide a good compromise between data abstraction and computational efficiency.

Eventually, according to the previously discussed arguments, ConnPri with pointer dereferncing (`getIthConn_p` getter), has been chosen as ultimate solution for the NEMO code.

2.7 Naming Conventions

Wherever possible we have tried to respect several naming conventions for classes, attributes, and operations in NEMO. The following is a brief summary for a better comprehension of the source excerpts which will be listed in the next chapters, and, as an example, it refers to the Vector class described in Sect. 5.3.1.

Class: the implementation of the Vector class is contained in the Fortran 95 module called `class_vector` . The class itself consists in a new DDT `Vector`: capital letters are used at the beginning of the class name. For a composed name, such as ScalarPde, the DDT name is `ScalarPde` and is contained in the `class_scalar_pde` module. [12]

Default public constructor: as explained by Akin (2003), when the components of the class DDT are kept private it is illegal, although tolerated by many compilers, to use the *default constructor* outside of the `class_vector` module, consisting of a function of type `Vector` with the DDT members as input arguments, such as:

```
Type(Vector) :: a
real(kind(1.d0)) :: x, y, z
! ...
a = vector(x,y,z) ! Default constructor
```

In order to make the constructor available outside the module, it is necessary to wrap it in the *public default constructor* `vector_` (lower case and underscored suffix) like:

[12] This is purely a convention because Fortran 95 is *case insensitive*.

```
function vector_(x,y,z)
  type(Vector) :: vector_
  real(kind(1.d0)), intent(in) :: x, y, z
  vector_ = vector(x,y,z)
end function vector_
```

Getters: if they just give public access to a private member, they keep the name of the member itself but with final underscore. For instance, in order to get the component in x direction of an object `v` of the Vector class one will use the function `x_` defined as:

```
elemental function x_(v)
  real(kind(1.d0)) :: x_
  type(Vector), intent(in) :: v
  x_ = v%x
end function x_
```

Generic Interfaces: there is a set of operations, such as allocation, deallocation, re–allocation and parallel broadcasting, which are defined for different classes. In order to simplify their call, each class provides a proper routine, such as `allocVector`, and a generic interface, such as

```
interface alloc
  module procedure allocVector
end interface
```

which permits use of the same procedure name with different data types.

Chapter 3

The PSBLAS Library

3.1 NEMO and PSBLAS

As anticipated in Chap. 1, NEMO is designed for running on distributed memory parallel systems, according to the *Message Passing* paradigm. This solution ensures portability on a wide set of machines: from multi–processor workstations, to the most complex supercomputers[1], or Beowulf[2] clusters based on commodity hardware.

This kind of approach is much more complicated than an OpenMP solution, which consists of the application of compiling directives in the sections of the code where the work load can be split among the processors of the shared memory machine. In particular the numerical algorithms used in a CCM code, which require great effort in order to be parallelized, are the iterative methods for the solution of the linear systems, arising from the discretization of a PDE.

In NEMO this task is accomplished by means of the Parallel Sparse BLAS (PSBLAS[3]) library, whose development has been addressed by Filippone and Colajanni (2000). In particular, PSBLAS provides a large set of very efficient preconditioners and non–stationary, iterative, methods from the Krylov subspace family (Saad, 2003). A parallel implementation of BLAS[4] (Basic Linear Algebra Subprograms) has been used for supplying the building blocks (see Barrett et al., 1994) of these complex, high level, algorithms.

Preconditioners and iterative methods are invoked by calling single subroutines in a manner completely transparent to the user. However, it would be impossible to switch NEMO's numerical kernel from PSBLAS to another

[1] `http://www.top500.org/`
[2] `http://www.beowulf.org/`
[3] `http://www.ce.uniroma2.it/psblas`
[4] `http://www.netlib.org/blas`

library, because NEMO has been designed *around* PSBLAS. For instance, the way PSBLAS handles the *halos*, i.e. the buffers at the numerical boundary of a process, which play a key–role for the parallel solution of a linear system, influences the strategy for the storage of local data, related to the mesh geometry and topology.

In summary, the current framework of NEMO wouldn't exist without PSBLAS.

3.2 Library Structure

The PSBLAS library is written in a mix of Fortran 95, Fortran 77 and C (for low level operations). The component hierarchy is depicted in Fig. 3.1 while Fig. 3.2 shows the UML diagram for the PSBLAS framework.

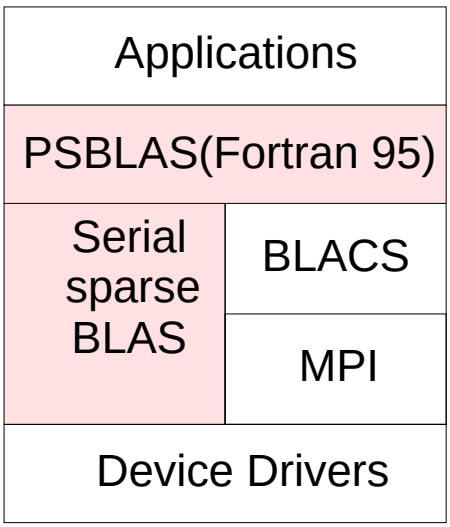


Figure 3.1 Component hierarchy of the PSBLAS library.

The library communicates at the highest level with the user's application by means of a Fortran 95 interface. Also the subroutines for iterative methods and preconditioners are written in this language. Under this cover, serial sparse BLAS routines (Duff et al., 1997), written in Fortran 77, are used for local basic linear algebra computation, while the memory management, low level operations and the interaction with the operating system is performed by means of C procedures.

As depicted in Fig. 3.1, the inter–processor communication in the message–passing paradigm is delegated to the BLACS[5] (Basic Linear Algebra Com-

[5] `http://www.netlib.org/blacs`

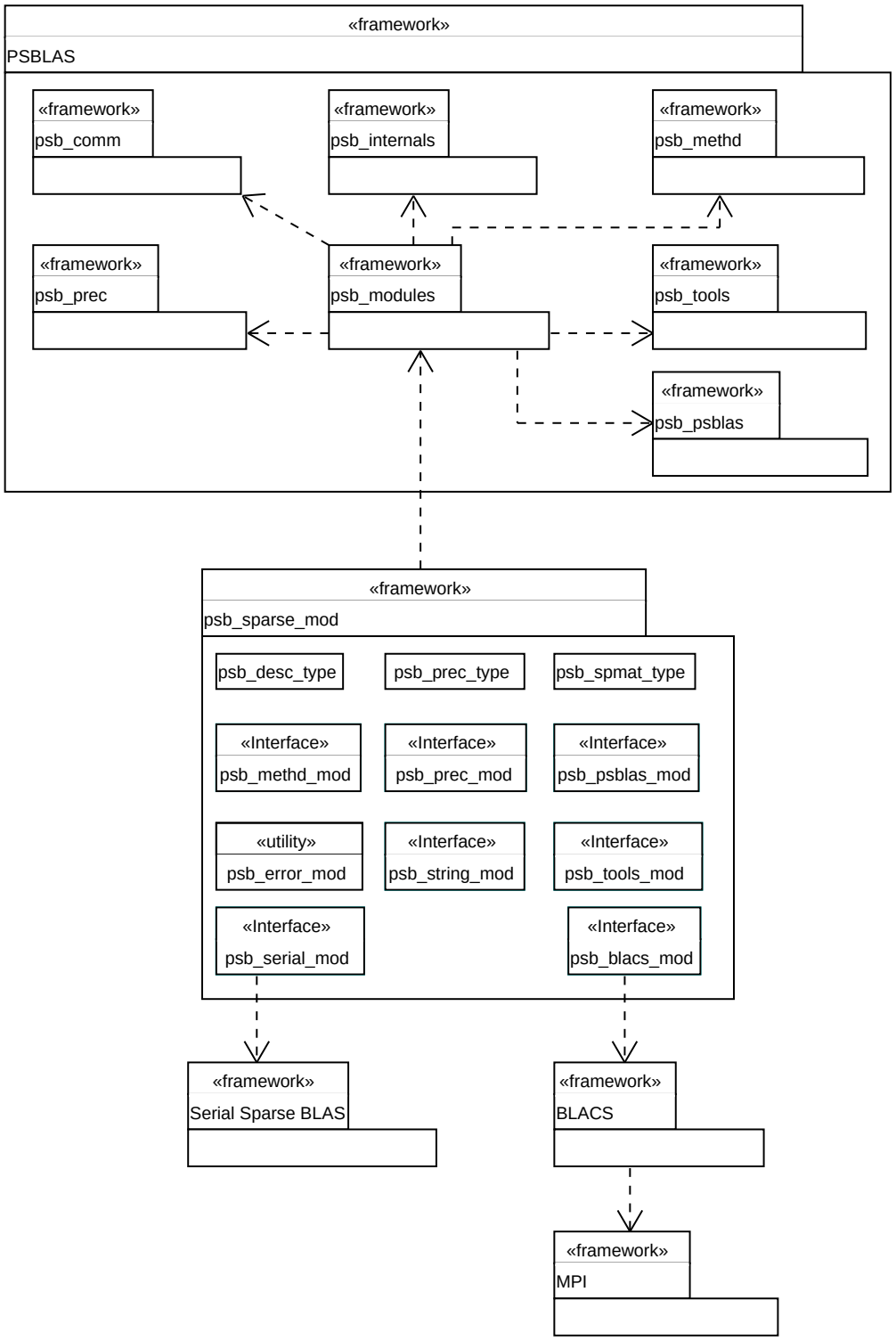


Figure 3.2 UML diagram for the PSBLAS framework.

munication Subprograms) library (see Dongarra and Whaley, 1995), which consists in a set of message–passing operations specialized for linear algebra applications. BLACS does not perform directly the message–passing between processes of a parallel job, rather it is built on top of an already existing communication library for distributed memory machines, such as MPI (Message Passing Interface), or PVM (Parallel Virtual Machine). PSBLAS requires the MPI–based version of BLACS. That is the reason why in Fig. 3.1 BLACS is sketched on the top of MPI.

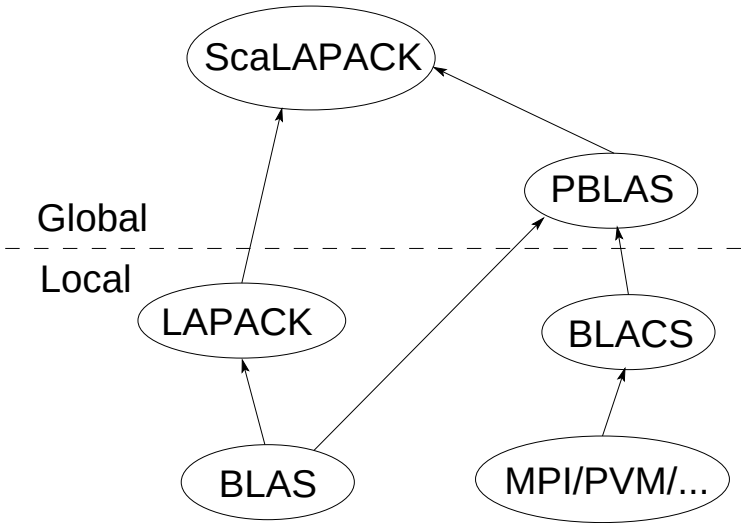


Figure 3.3 Component hierarchy of the ScaLAPACK library.

The advanced user could find some analogies between ScaLAPACK[6] (Scalable Linear Algebra PACKages), whose structure is depicted in Fig. 3.3, and PSBLAS: the former is an implementation of dense algebra algorithms, while the latter deals with sparse matrices; both libraries have been designed for running on distributed memory systems. Lastly they both use for parallel communication the BLACS package, which encapsulates the Message Passing Interface in a portable and efficient layer, expressly designed for parallel operations on dense linear algebra structures. However, in some cases, MPI routines are directly used either to improve efficiency or to implement communication patterns for which the BLACS package does not provide any method, such as *gathering or scattering* operations.

In the latest release of the PSBLAS library an interesting feature has been added. A Fortran 95 wrapper for BLACS routines applies a generic interface to the procedures which perform the same task on different intrinsic types,

[6] `http://www.netlib.org/scalapack/`

such as integer, real, etc. This allows more compact calls to the BLACS set, which is used not just internally to PSBLAS, but in NEMO too, for:

- broadcasting input parameters;
- broadcasting geometry and topology–related data;
- parallel reduction operations (sum, maximum evaluation, etc.);
- enforcing synchronization among processes;
- aborting a parallel job.

The PSBLAS library consists of two classes of subroutines that is, the *computational routines* and the *auxiliary routines*. The computational routine set includes:

- sparse matrix by dense matrix product[7];
- sparse triangular systems solution for block diagonal matrices;
- vector and matrix norms;
- dense matrix sums;
- dot products.

The auxiliary routines set includes:

- communication descriptor allocation;
- dense and sparse matrix allocation;
- dense and sparse matrix build and update;
- sparse matrix and data distribution preprocessing.

[7]This includes sparse matrix by vector product.

3.3 Iterative Methods

The application of the FVM leads to the discretization of PDEs; if the resulting algebraic equations are written in implicit form for every node of the computational mesh, one has to solve a linear system

$$Ax = b \tag{3.1}$$

where A is a sparse matrix. As explained by Saad (2003) direct methods, like Gaussian elimination, are unfit to this kind of systems, first of all because of their excessive fill–in. Hence only iterative methods should be used. In this superset, stationary methods, like Jacobi or Gauss–Seidel, are not the best choice, due to their very low convergency speed. Therefore, one should use non–stationary methods, like the ones belonging to the Krylov subspace family. PSBLAS contains a parallel implementation of the following methods:

- Conjugate Gradient (CG);
- Generalized Minimal Residual (GMRES);
- BiConjugate Gradient (BICG);
- Conjugate Gradient Stabilized (CGS);
- BiConjugate Gradient Stabilized (BiCGSTAB).

3.4 Preconditioners

Lack of robustness is a widely recognized weakness of iterative solvers when compared to direct ones. This drawback hampers the acceptance of iterative methods in industrial applications despite their intrinsic appeal for very large linear systems. Both the *efficiency* and *robustness* of iterative solvers can be improved by the usage of *preconditioning*.

The convergence rate of an iterative method depends on the *spectral* properties of the coefficient matrix. Hence one may attempt to transform the linear system into one that is equivalent, that is having the same solution, but that has more favorable spectral properties.

A *preconditioner* is a matrix that performs such a transformation. For instance, if the preconditioning matrix M approximates somehow the coefficient matrix A, the transformed system:

$$M^{-1}Ax = M^{-1}b \tag{3.2}$$

has the same solution of the original system 3.1, but the spectral properties of its coefficient matrix $M^{-1}A$ could make it more suitable for being solved by means of an iterative method. In general, the reliability of iterative techniques depends much more on the quality of the preconditioner than on the particular Krylov subspace method used.

The general problem of finding a preconditioner for a linear system $Ax = b$ (see Barrett et al., 1994; Saad, 2003) is to find a matrix M (the "preconditioner" or "preconditioning matrix") with the following properties:

1. M is somehow a good approximation of A;

2. the cost of the construction of M is not too high;

3. the system $Mx = b$ is much easier to solve than the original system.

As explained by Filippone and Colajanni (2000) and Buttari (2006), PSBLAS contains a parallel implementation of the following preconditioners:

- Diagonal Scaling;
- Incomplete LU (ILU) factorization;
- Additive Schwarz;
- Multilevel Schwarz.

The best couple preconditioner/solver is not known in advance, but is somehow related to the problem itself. For example, convergence can be guaranteed and analytically proven for the CG method only with symmetric A matrices, like those deriving from the application of the central difference scheme to elliptic problems (e.g. steady diffusion). On the contrary, hyperbolic problems and upwind–like discretization schemes produce non–symmetric systems, which can converge only with certain methods, like BiCGSTAB. The issue is even more complex for CFD solvers like SIMPLE or PISO for subsonic flows, where, as explained by Patankar (1980) or by Ferziger and Perić (2002), the predictor step for momentum equation is generally non–symmetric, while the corrector step(s) for pressure is a Poisson's equation with symmetric discretization. In this case two different solvers are needed for the respective linear systems. Similarly, several preconditioners are compatible with a reduced set of solvers; for instance, some Multilevel Schwarz preconditioners may be coupled only with the BiCGSTAB method and cannot run with the CG, even if the problem is symmetric.

3.5 Main PSBLAS Data Structures

In this section the main data structures of the PSBLAS library are addressed. For further references about its framework and functionalities the interested reader is redirected to the works of Filippone and Colajanni (2000) and Buttari (2006); the "PSBLAS–2.0 User's Guide" by Filippone and Buttari (2006) contains an exhaustive overview of the most important derived data types, modules, and high level procedures, with their respective interfaces and argument lists. The following naming scheme has been adopted in the PSBLAS software package:

- all the symbols (i.e. subroutine names, data types...) are prefixed by `psb_`;
- all the data type names are suffixed by `_type`;
- all the constant values are suffixed by `_`;
- all the subroutine names follow the rule `psb_xxname` where `xx` can be either:
  - `ds`: the routine is related to dense data;
  - `sp`: the routine is related to sparse data;
  - `cd`: the routine is related to communication descriptor (see Sect. 3.5.2).

For example, `psb_dsins`, `psb_spins` and `psb_cdins` perform the same action respectively on dense matrices, sparse matrices and communication descriptors. Interface overloading allows the usage of the same subroutine name for both real and complex data.

3.5.1 Library Design Choices

In any distributed memory application, the data structures that represent the partition of the computational problem are essential to the viability and efficiency of the entire parallel implementation. The criteria guiding the decomposition choices are:

1. maximizing load balancing,
2. minimizing communication costs,
3. optimizing the efficiency of the serial computation parts.

For *dense* linear algebra algorithms the ScaLAPACK (Blackford et al., 1997) library has demonstrated that a *block–cyclic* distribution of the row and column index spaces is general and powerful enough to achieve a good compromise among the various factors that affect parallel computation performance. PSBLAS library addresses parallel *sparse* iterative solvers typically arising in the numerical solution of PDEs. In these instances, it is necessary to pay special attention to the structure of the problem from which the application originates. The nonzero pattern of a matrix arising from the discretization of a PDE is influenced by various factors, such as the shape of the domain, the discretization strategy, and the equation/unknown ordering. The matrix pattern itself can be interpreted as the connectivity table of the adjacency graph associated with the discretization mesh; this characteristic will be discussed in Sect. 4.2.

The allocation of the coefficient matrix for the linear system is based on the "owner computes" rule: the associated variable of each mesh point is assigned to a process that will own the corresponding row in the coefficient matrix and will carry out all related computations. This allocation strategy is equivalent to a partition of the discretization mesh into *sub–domains*. PSBLAS routines support any distribution that keeps together the coefficients of each matrix row; there are no other constraints on the variable assignment. The available distribution strategies include data distributions commonly used in ScaLAPACK such as `Cyclic(N)` and `Block`, as well as completely arbitrary assignments of equation indices to processes. Dense vectors conform to sparse matrices, that is, the entries of a vector follow the same distribution of the matrix rows. It is never required that the entire system matrix is available on a single node and efficiency is obviously improved in the case where each node generates its own portion of the system. However, it is possible to hold the entire matrix in one process and distribute it explicitly, even though the resulting bottleneck would make this option unattractive in most cases.

The storage scheme used for the computational parts pertaining to each process conforms to the storage formats defined in the serial sparse BLAS proposal (Duff et al., 1997). The data structures that describe the local matrix storage are kept separated from those used for representing the communication pattern. This choice satisfies the encapsulation relations among the PSBLAS layers as described previously.

3.5.2 Communication Descriptor

Once the distributed sparse matrix has been allocated and built, it is necessary to arrange the data structures that will be used to perform inter-process

communications during the execution of routines such as matrix-vector products, preconditioners and scalar products. The detailed contents of these data structures depend on the sparsity pattern of the coefficient matrix. This index space is classified according to the user defined assignment of its elements to the processes of the parallel machine. The PSBLAS computational model implies that the data allocation on the parallel distributed memory machine is guided by the structure of the physical model, and specifically by the discretization mesh of the PDE. Each point of the discretization mesh will have (at least) one associated equation/variable, and therefore one index. Point i depends on point j if the equation for a variable associated with i contains a term in j, or equivalently if $a_{ij} \neq 0$. After the partition of the discretization mesh into *sub–domains* assigned to the parallel processes, the points of a given sub–domain can be classified as following:

Internal. An internal point of a given subdomain depends only on points of the same subdomain. If all points of a domain are assigned to one process, then a computational step (e.g., a matrix–vector product) of the equations associated with the internal points requires no data items from other subdomains and no communications.

Boundary. A point of a given subdomain is a boundary point if it depends on points belonging to other subdomains.

Halo. A halo point for a given subdomain is a point belonging to another subdomain such that there is a boundary point which depends on it. Whenever performing a computational step, such as a matrix–vector product, the values associated with halo points are requested from other subdomains. A boundary point of a given subdomain is a halo point for (at least) another subdomain; therefore the cardinality of the boundary points set denotes the amount of data sent to other subdomains.

Overlap. An overlap point is a boundary point assigned to multiple subdomains. Any operation that involves an overlap point has to be replicated for each assignment. This may be acceptable, for example to accelerate the convergence rate of the overall iterative method through an improvement of the preconditioning task (Filippone et al., 1992).

The sets of *internal*, *boundary* and *halo* points for a given subdomain are denoted by $\mathcal{I}$, $\mathcal{B}$ and $\mathcal{H}$ respectively. Each subdomain is assigned to one process; each process usually owns one subdomain in which case the number of rows in the local sparse matrix is $|\mathcal{I}_i| + |\mathcal{B}_i|$, and the number of local columns (i.e. those for which there exists at least one non-zero entry

in the local rows) is $|\mathcal{I}_i| + |\mathcal{B}_i| + |\mathcal{H}_i|$. The representation of the points of a subdomain assigned to a process is stored into integer arrays; the internal format is described in more detail in Filippone and Buttari (2006). Upon each invocation of a PSBLAS subroutine, it is assumed that only one set of communication descriptors is active. This assumption is not too restrictive in view of commonly addressed applications. Indeed, if the library is used in the context of a PDE solver, often multiple equations are defined on the same discretization mesh, so that the coefficient matrices of the linear systems have the same sparsity pattern.

The definition of the communication descriptor is the following.

MATRIX_DATA A vector containing some general information, such as the descriptor status, the size of the global matrix, the number of local rows and columns and the BLACS communication context.

HALO_INDEX The local list of the indices of halo points that have to be exchanged with other processes. For each process, this list contains the process identifier, the number and indices of points to be received, the number and indices of points to be sent.

OVERLAP_INDEX A (local) list of the overlap points, organized in groups like the halo index descriptor; the format is identical to that of the halo descriptor.

OVERLAP_ELEM A (local) list containing for each overlap point its local index and the number of processes sharing it. This information is implicitly available in `OVERLAP_INDEX`; it is computed and stored separately at initialization time for efficiency reasons.

Two auxiliary arrays are used to keep track of the mapping between local and global indices. The Fortran 95 language allows a convenient packing of the necessary data structures inside a single DDT as reported in Fig. 3.4.

```
type psb_desc_type
   integer, pointer :: matrix_data(:), halo_index(:)
   integer, pointer :: overlap_elem(:), overlap_index(:)
   integer, pointer :: loc_to_glob(:), glob_to_loc(:)
end type psb_desc_type
```

Figure 3.4 PSBLAS derived data type storing the communication descriptor.

3.5.3 Sparse Matrix Storage

Assembling a sparse matrix requires that the user defines matrix entries in terms of the global equation numbering. Then, each process in the parallel machine builds the sparse matrix rows that are assigned to it by the user by means of the `psb_spins` subroutine. Once the build step is completed, the local part of the matrix undergoes a preprocessing operation. During this step, performed through the `psb_spasb` subroutine, the global numbering scheme is converted into the local numbering scheme, and the local sparse matrix representation is converted to a format suitable for subsequent computations.

The serial sparse BLAS routines (Duff et al., 1997) are used to carry out the internal storage conversion step and all other operations involving local sparse matrix computations. The paper contains a detailed discussion about the rationale for the sparse matrix representation shown below. In particular, it describes the format of the permutation vectors `pr` and `pl`. These permutations arise in various sparse storage format conversions that may impose a renumbering of the local equations and variables to achieve the desired runtime efficiency. In PSBLAS, the sparse matrix storage conforms to the Fortran 95 implementation of the serial sparse data structures reported in Fig. 3.5.

```
type psb_dspmat_type
   integer :: m, k
   character :: fida(5)
   character :: descra(10)
   integer :: infoa(10)
   real(kind(1.d0)), pointer:: aspk(:)
   integer, pointer :: ia1(:), ia2(:), pr(:), pl(:)
end type psb_dspmat_type
```

Figure 3.5 PSBLAS derived data type storing a sparse matrix.

Complex matrices have a similar structure, with the appropriate type declaration for member `aspk`. At the moment the PSBLAS library provides the possibility to store sparse matrices in the Compressed Sparse Row (CSR), Coordinate (COO) and Jagged Diagonal (JAD) formats. However the `psb_spmat_type` data structure has been designed to be flexible enough to easily allow the implementation of other storage formats. In this sense, the content of the `ia1`, `ia2` and `aspk` arrays is interpreted according to the content of the `fida` and `infoa` records. The following two cases are among

the most commonly used:

`fida='CSR'` → Compressed Sparse Row format:

1. `ia2(i)` contains the index of the first element of row `i`; the last element of the sparse matrix is thus stored at index $ia2(m+1)-1$. It should contain `m+1` entries in non–decreasing order (strictly increasing, if there are no empty rows).
2. `ia1(j)` contains the column index and `aspk(j)` contains the corresponding coefficient value, for all $ia2(1) \leq j \leq ia2(m+1)-1$.

`fida='COO'` → COOrdinate format:

1. `infoa(1)` contains the number of non–zero elements in the matrix.
2. For all $1 \leq j \leq infoa(1)$, the coefficient, row index and column index are stored into `apsk(j)`, `ia1(j)` and `ia2(j)` respectively.

3.5.4 PSBLAS Data Structures Building Steps

During their existence, a sparse matrix and a communication descriptor may be in different states. A sparse matrix may be in the "build" (`bld`), "assembled" (`asb`) or "update" (`upd`) states while a descriptor may be in the `bld` or `asb` ones. Provided that all the computational subroutines may be invoked only when both of them are in the `asb` state, there are different paths to reach these states by means of the insert and assembly routines to provide the user the greatest flexibility. Namely there are two paths: one where the matrix and descriptor constructions are handled separately and another one to build both of them at once.

Figure 3.6 reports the typical layout of a PSBLAS application and shows how the sparse matrix and descriptor data structures can be built and which state transitions are determined by the invocation of some auxiliary subroutines (note that either step 2 or 3 is performed):

1. Initialize the communication descriptors with `psb_cdall`; initialize the sparse matrix with `psb_spall` and RHS with `psb_dsall`. After this step, both the sparse matrix and the communication descriptor are in the `bld` state.
2. Assemble the communication descriptor and the sparse matrix: two different paths can be followed.
   2.1. This step is associated with the right branch on Fig. 3.6:

a. Loop over all mesh points owned by the current process, build their equations and insert them into the communication descriptor through calls to the `psb_cdins` subroutine.

b. Assemble the communication descriptor through a call to the `psb_cdasb` subroutine. At the end of this step the descriptor is in the `asb` state.

c. Loop on all mesh points owned by the current process (not necessarily in the same order as in step a), build their equations and insert them into the sparse matrix and right hand side through calls to the `psb_spins` and `psb_dsins` subroutines.

d. Assemble the sparse matrix and the right hand side through calls to the `psb_spasb` and `psb_dsasb` subroutine. At the end of this step the sparse matrix is in the `asb` state.

2.2. This step is associated with the left branch on Fig. 3.6:

a. Loop over all mesh points owned by the current process, build their equations and insert them into the communication descriptor, sparse matrix and right hand side through calls to the procedures `psb_spins` and `psb_dsins`. If the descriptor is in the `bld` state instead of the `asb` one (which is the case of the left branch in Fig. 3.6), the `psb_spins` also takes care of inserting the points into it.

b. Assemble the communication descriptor, sparse matrix and right hand side through calls to the `psb_cdasb`, the `psb_spasb` and the `psb_dsasb`. At the end of this step both the descriptor and the sparse matrix are in the `asb` state;

3. Compute the preconditioner and call the iterative method.

4. If a system with the same sparsity pattern must be solved, the system matrix may be reinitialized through a call to the `psb_spreinit` subroutine. At the end of this step the matrix is in the `upd` state.

5. Loop on all the mesh points owned by the current process in the same order as in step 2.1.c or 2.2.a, build their equations and insert them into the sparse matrix and right hand side through calls to the `psb_spins` and `psb_dsins` subroutines.

6. Assemble the sparse matrix and the right hand side through calls to the `psb_spasb` and `psb_dsasb` subroutine. At the end of this step the sparse matrix is in the `asb` state; go to step 3.

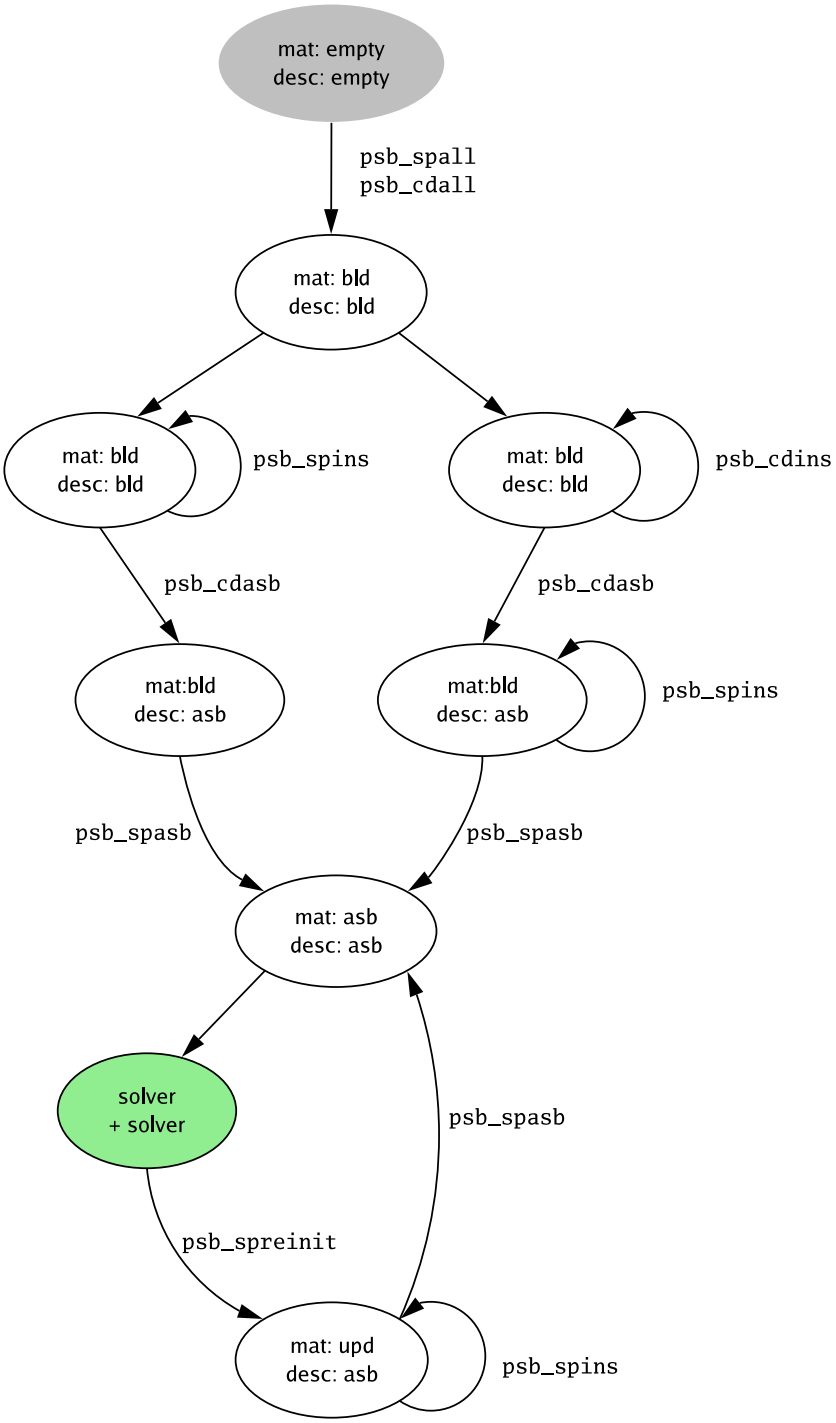


Figure 3.6 PSBLAS data structures building steps.

The order in which the mesh points are visited in step 2.1.c or 2.2.a is arbitrary. This is useful for finite element applications, where it may be desirable to loop over the elements, rather than over the equations.

Substantial savings in the time needed to build a sparse matrix may be achieved by considering that for many applications the same problem has to be solved repeatedly, for example in successive time steps. If the topology of the discretization mesh doesn't change from one time step to another, the system matrices will have the same sparsity pattern. In this case, extra information may be stored during the insert and assembly steps of the first problem solution to be reused in the next ones, allowing the code to perform the same steps in a shorter time. In particular, this requires that the user reinitializes the sparse matrix at the beginning of each new iteration (except for the first one) through a call to the `psb_spreinit` subroutine. This operation allows the regeneration of a matrix on the base of information stored in a previous matrix building.

If the data distribution is independent of the discretization mesh structure, for example when using a Block distribution, the above application structure has no serial bottlenecks. If a graph partitioning package is used, the cost of the setup phase depends on the graph partitioning routine. If the subroutine is serial, there is a serial bottleneck both in terms of processing time and memory space, because one of the processes will hold the structure of the entire discretization mesh. For many applications this would not be a serious drawback, because the linear solver itself is a single step in an outer solution algorithm, and often many (if not all) consecutive steps share the same mesh topology. The serial graph partitioning approach is not very suitable to applications that use adaptive meshes with fast rates of change; extensions to PSBLAS for such applications are currently being investigated (see Filippone et al., 1999).

3.6 The Blacs class

Every class and external procedure defined in NEMO loads a module called `class_blacs` that contains the global variables, related to the BLACS environment, which must be set at the beginning of a parallel job. This represents the only case of common variables shared by the different parts of the code. Although this practice should be avoided, in order to improve the readability of the code, it can be tolerated in this single circumstance; the corresponding data are really of common interest for all procedures, because they describe the parallel framework, where the message passing happens. The declarative section of the module is depicted in Fig. 3.7.

```
module class_blacs
  use psb_sparse_mod
  logical, private :: blacsOn = .false.
  integer, private :: icontxt, mypnum, nprocs
  integer, private :: myprow, mypcol, nprows, npcols
  real(kind(1.d0)), private :: tic

  interface gebs2d
     module procedure cgebs2ds, cgebs2dv, lgebs2ds, lgebs2dv
  end interface

  interface gebr2d
     module procedure cgebr2ds, cgebr2dv, lgebr2ds, lgebr2dv
  end interface
contains
  ! ... class operations...
end module class_blacs
```

Figure 3.7 Source code of the Blacs class.

The first statement, `use psb_sparse_mod`, loads all PSBLAS–related modules, containing derived data type declaration and procedures interfaces, including the Fortran 95 wrapper of BLACS library. Wherever in NEMO a PSBLAS routine must be invoked, it is therefore sufficient to insert the `use class_blacs` statement, in order to provide implicitly also the corresponding needed interface.

The global variables declared in the upper section of the module have the attribute `private`, in order to protect them from accidental access and modification. They can be accessed in read–only mode by using proper getter functions, defined in the `contains` section of the module. The following is a brief list of what they mean:

- `blacsOn`: it is used in different sections of the code in order to check if the BLACS environment has been initialized.
- `icontxt`: stores the value of the BLACS `context`, analogous to the `mpi_comm_world` descriptor used in MPI.
- `mypnum`: ID of the local process.
- `nprocs`: number of processes of the parallel job.

- `myprow`, `mypcol`: row and column coordinates of the local process in the BLACS `nprows`×`npcols` grid of processes.
- `tic`: stores the wall–clock time of the run beginning.

The generic interfaces `gebs2d` and `gebr2d` extends the Fortran 95 interfaces for the BLACS library, provided by PSBLAS, to `logical` and `character` types, according to the listed module procedures, whose implementation is not included in Fig. 3.7.

The information stored in this module does not build a class in the traditional way of meaning, first of all because there are no new ADTs defined in it. Second, there is just a single instance of these data during the execution of the program. However, OOP and UML provides a particular *stereotype* for classes which contains only *static* members (functions and data) and cannot be instantiated: they are defined as *utility* classes. Libraries and modules, like `class_blacs`, can be classified with this attribute and find a proper collocation in the UML description of the code. Figure 3.8 depicts the UML diagram for the Blacs class and its `<<use>>` dependency from the `psb_sparse_mod` module supplied by the PSBLAS framework, as previously seen in Fig. 3.2.

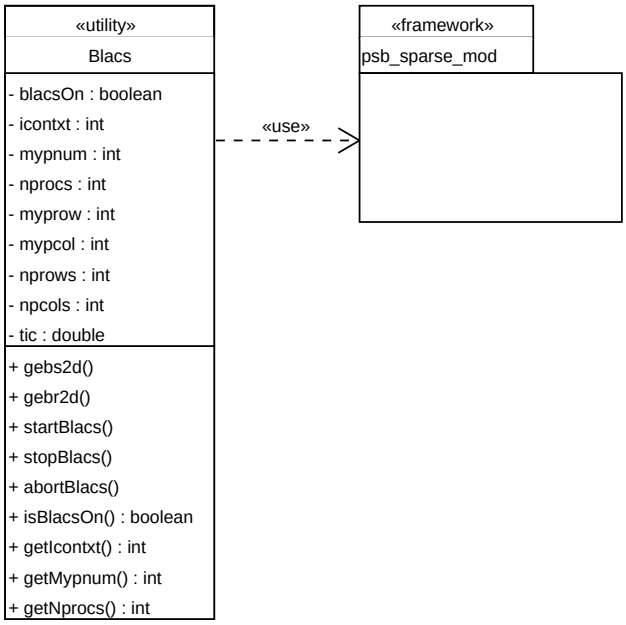


Figure 3.8 UML diagram of the Blacs class.

The procedures `startBlacs`, `stopBlacs` and `abortBlacs` respectively initialize, close and make abort the BLACS environment; they are wrappers

of the BLACS support routines, described by Dongarra and Whaley (1995), which provide controls over the BLACS grid of processes. The remaining functions listed in the operation compartment of the Blacs class are getters which give access to the corresponding private members.

Chapter 4

Partitioning and Reordering

4.1 Partitioning Strategies

The data decomposition strategy is fundamental in every parallel application, in particular in a SPMD (Single Program Multiple Data) code, such as NEMO, where the parallelism consists in performing operations on subsets of a main data set. Every partitioning scheme should accomplish two tasks:

1. balancing as much as possible the amount of operations performed by each process of the parallel job;

2. minimizing the amount of data which has to be exchanged according to the Message Passing paradigm.

In PSBLAS the criterium for the assignment of data to the different processes, coherently with the paradigm *"owner computes"*, is based on the correspondency between the *adjacency graph* of the discretization mesh and the *sparsity pattern* of the coefficients matrix of the linear system $Ax = b$ to be solved. In particular, the index space of the state vector x is distributed among the processes so that each index i belongs to a process p; this means that p owns both the i^{th} row of matrix A and the i^{th} node of the adjacency graph, or, equivalently, the i^{th} element of the mesh.

The subdivision of the computational grid among the available processes consists therefore in the redistribution of the rows of the matrix A. Concerning this issue, the main feature of the PSBLAS project is to let the user freely decide the partitioning of the simulation domain. Any assignment of the row indices is logically acceptable and produces results functionally correct[1]; however, this does not mean that all the redistribution policies are equivalent in

[1] A testing strategy consists in assigning *randomly* the indices to the processes!

terms of efficiency. The choice of a particular data decomposition influences the parallel performances according to two factors:

1. the resources needed for achieving the desired partitioning;

2. the influence of the decomposition on the efficiency of elemental operations in iterative methods, especially the matrix–vector product.

If one considers the sequence of computational steps performed in the iterative solutions of a linear system, we may state that the amount of floating–point operations in each process is proportional to the number of cells assigned to the process itself, while the amount of data to be passed between the processes is proportional to the number of cells lying on the numerical boundary between two or more adjacent processes. In other words, if we cut into "*slices*" the discretization domain, the computational load is proportional to the *volume* of every "*slice*", while the communication is proportional to the interface *surface* between processes. In summary, the optimum choice for data allocation is substantially equivalent to the problem of *partitioning a graph*, that is how to assign the nodes of a graph to distinct subsets (with no intersections) so that:

- the assignment of nodes to subdomains is homogeneous and well balanced;

- the number of arcs which connect nodes in two distinct subsets, crossing their interface, is minimized.

This kind of problem is $\mathcal{NP}$–complete (Garey and Johnson, 1978), i.e. it belongs to a class of problems for which it is substantially impossible to find the optimum solution. Therefore one has to use heuristic strategies which should lead to a good compromise between the time necessary to find a solution and its quality; the possible options can be very different, so that it would have been too restrictive, in the definition of the PSBLAS protocol, to force the user to adopt a specific one.

A further consideration is related to the dynamic behavior of the discretization mesh. If the grid is *static* and the simulation lasts a large number of time–steps, the usage of a very expensive partitioning strategy is then acceptable, because it has to be done just once, at the beginning of the simulation; the result can be re–used and its cost is amortized. On the opposite extreme one could have a *strongly adaptive* application, such as the one developed by Dai and Schmidt (2005), where at each time–step a topology change occurs; in this circumstance on should choose either an extremely

simple and light strategy or a more sophisticated one that permits updating the partitioning by means of successive small adjustments in the graph distribution.

Lastly, while in the PSBLAS library there are no assumptions about the application where the problem originates from, it is still possible that the knowledge of the physics underlying the numerical problem could guide the user to the most suitable partitioning strategy.

Having stated that the choice of the particular data decomposition is completely left to the user, the only object that one has to create, in order to interface correctly with PSBLAS, is a subroutine which fixes a correspondence between the index of every mesh element and the process which that index/unknown/row is assigned to. The availability of this user–defined procedure permits the code to allocate the *descriptor* of the sparse matrix A by means of a simple call to the PSBLAS auxiliary routine `psb_cdall`, as showed in Fig. 4.1.

```
if(ipart > 0) then
   call getConnCSR(msh%c2c,xadj_glob,adjncy_glob)
end if

select case(ipart)
case(0)
   ! HPF Block partitioning
   call bldPartBlock(ncells,nprocs,ivg)
case(1)
   ! ParMETIS Partitioning
   call bldPartGraph(xadj_glob,adjncy_glob,ipart=1)
   call getPartGraph(ivg)
case default
   write(*,*) 'ERROR! Unknown partitioning method'
   call abortBlacs
end select

call psb_cdall(ncells,ivg,icontxt,msh%desc,info)
```

Figure 4.1 Source code of the mesh partitioning subroutine.

The switch `ipart` assumes different values according to the partitioning method selected by the user. It is processed inside a `select case` construct whose different branches cause the invoking of distinct decompositions. In

all cases the partitioning result is stored in the allocatable array `ivg`, such that `ivg(i)` stores the ID of the process which the i cell belongs to. Lastly, `ivg` is passed as input to the PSBLAS routine `psb_cdall` which allocates the descriptor `msh%desc` of type `psb_desc_type`.

4.1.1 Block and Cyclic Decomposition

A first kind of data decomposition comes from the partitioning strategies typical of dense linear algebra, as implemented in ScaLAPACK and supported by the programming language HPF (High Performance Fortran).[2] They are based simply on the size of the index space and on the number of available parallel processes; hence the sparsity pattern of the coefficient matrix is completely neglected . The decompositions defined in this context are:

Block: the space of index $i \in 1 \dots N$ is split in np blocks made of contiguous indices where np is the number of processes.

$$i \mapsto \lceil i/L \rceil \quad L = \lfloor N/np \rfloor \tag{4.1}$$

Cyclic: the indices are assigned cyclically to the processes:

$$i \mapsto i \bmod np \tag{4.2}$$

Block–cyclic: the indices are cyclically assigned to processes in blocks with size equal to r:

$$i \mapsto \lfloor (i \bmod T) \rfloor \quad T = r \cdot np \tag{4.3}$$

The Block–cyclic distribution includes the Cyclic (for $r = 1$) and the Block (for $r = L$). Figure 4.2 represents the Block partitioning on 3 blocks of a vector with 8 elements.

The load balancing criterium consists essentially of creating blocks as homogeneous as possible; therefore it is a matter of managing the rest of the integer division between the number of elements n and the number of blocks/processes np. The block partitioning is for sure a very straightforward subdivision, even if it does not guarantee a working efficiency because it doesn't consider the topology of the discretization mesh. Nevertheless good results can be achieved when it is used with structured grids (see Sect. 5.1.1), having the same topology of a matrix, with rows, columns, and a progressive natural numbering (row–major or column–major) of the cells.

Figures 4.3 and 4.4 show the partitioning on 4 processes of the same cubic solid discretized, respectively, with a structured and an unstructured mesh.

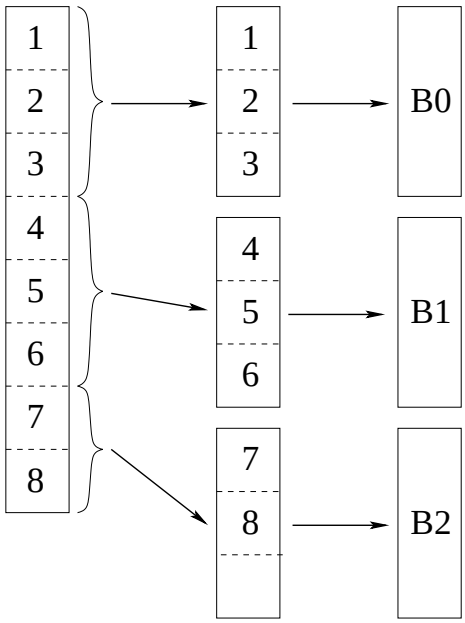


Figure 4.2 Block partitioning into 3 subsets of a vector with 8 elements.

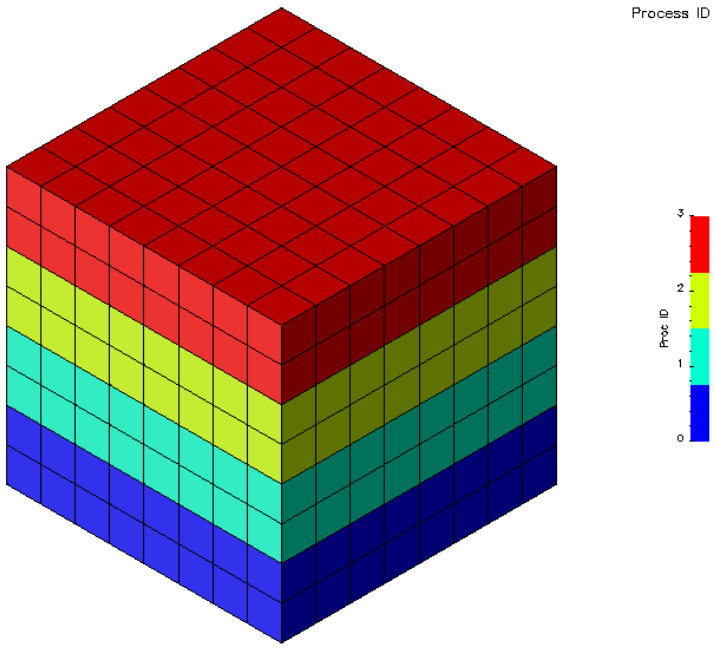


Figure 4.3 Block partitioning into 4 subsets of a structured mesh.

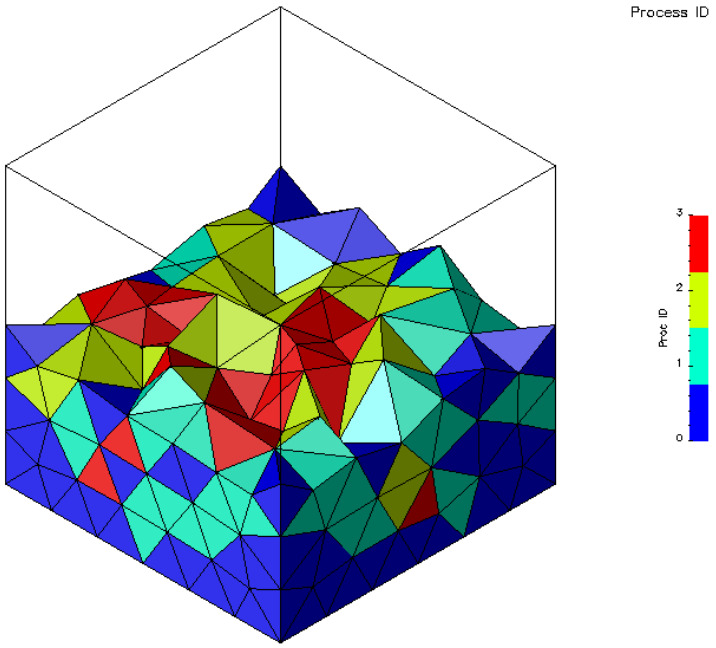


Figure 4.4 Block partitioning into 4 subsets of an unstructured mesh.

The relative good quality of the partitioning in Fig. 4.3 is related to the "surface/volume" effect previously mentioned in Sect. 4.1; vice versa it is clear that if the numbering is irregular, as for the unstructured mesh in Fig. 4.4, one can get a "checkerboard" allocation effect which, of course, is exactly the opposite of an optimum solution in terms of minimizing communication, although the load is well balanced among the processes.

4.1.2 METIS and ParMETIS

When the mesh is particularly complex and, most of all, is unstructured, i.e. it does not show the connectivity stencil typical of a 2D or 3D matrix (left–right, top–bottom, front–derrière), contains arbitrary coupling between different blocks, or geometrical elements that are not quadrangular (2D) or hexahedral(3D), it is then necessary to perform the data decomposition by means of more complex techniques, such as the algorithms for the partitioning of graphs.

[2]HPF consists of a set of extensions to Fortran 90 which provide access to high–performance architecture features while maintaining portability across platforms. See also `http://www.netlib.org/hpf`

As previously mentioned, there is a 1–1 correspondence between the sparsity pattern of A matrix and the discretization mesh; with reference to finite volume modeling, one can build its adjacency graph by associating one node at each control volume and one arc at each, namely $N1$ and $N2$ nodes such that the balance equation for the $N1$ contains also the unknown variable of $N2$. In this case the concept of the adjacency graph is numerical, rather than geometrical, because the use of high order discretization schemes, such as QUICK (Versteeg and Malalasekera, 1995), involves into the same equation also nodes which do not share cell faces. The non–zero coefficient a_{ij} corresponds to the contribution of the variable stored at node j to the balance expressed by the discretized equation written for the node i. It is therefore clear that the adjacency graph coincides exactly with the sparsity pattern of the coefficient matrix of the linear system associated to the finite volume discretization. For this reason, it is straightforward to use any algorithm for graph partitioning in order to solve the problem of data decomposition.

Presently, one of the most popular tools for such purposes is the *METIS*[3] package, developed by Karypis and Kumar (1998); Fig. 4.5 depicts a scheme of the decomposition process according to this approach.

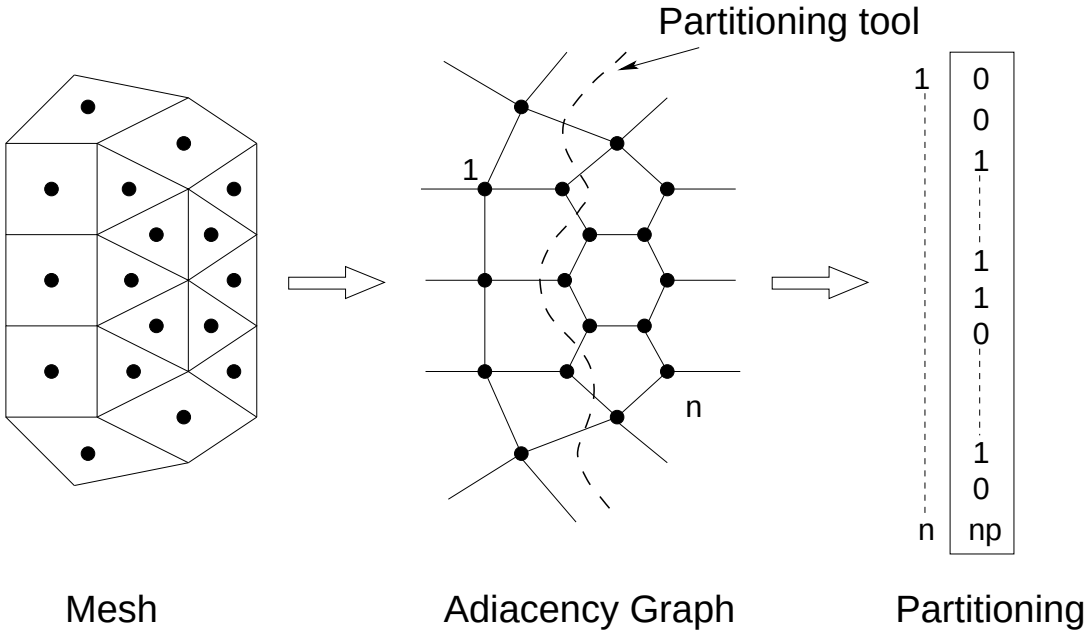


Figure 4.5 Scheme of graph partitioning by means of the METIS software.

As previously stated the partitioning software has to assign every graph

[3]http://glaros.dtc.umn.edu/gkhome/views/metis

node to a subdomain, trying to satisfy two requisites:

- **load balancing**: the single blocks must have more or less the same number of elements;

- **communication minimization**: the number of graph arcs which cross the boundary between two subdomains, representing the inter-processor communication, must be minimized.

At the end of the partitioning in np blocks of a mesh with n cells, one obtains a vector of n elements that indicates for each cell the corresponding ownership block. This is exactly the information carried by the array `ivg` in Fig. 4.1 as input argument of the PSBLAS routine `psb_cdall` which allocates the communication descriptor.

In the above formulation one has used exclusively information pertaining the topology of the computational domain; in particular, one is assuming that all graph nodes (that is all control volumes) are equivalent. This could not be true; therefore the majority of software tools for graph partitioning permits to specify explicitly a weight for each node. The goal of load balancing will be then accomplished by having a total sum of the weights associated to the nodes of each subdomain as homogeneous as possible; usually, this is the only way for introducing in this kind of scheme extra information arising from the knowledge of the physical behavior of the problem.

Figures 4.6 and 4.7 show respectively the Block and the METIS partitioning of the computational grid used for parallel heat transfer simulations on a Diesel engine piston. The mesh is made of about $686,000$ tetrahedral cells. The Block partitioning produces a very disordered partitioning, with a large *edgecut*, i.e a large number of arcs of the adjacency graph which cross the boundary between two adjacent subdomains. This is due to the particular numbering of the cells, which in its turn comes from the specific meshing algorithm used in the discretization of the three–dimensional geometry, such that each cell is surrounded by other ones which are very "distant" in the index space. This disuniformity, or "checkerboard" effect, also reflects into the sparsity pattern of the associated matrix, as depicted in Fig. 4.8. Such a scattered pattern originates from the fact that the non–zero elements of each row are very far from the main–diagonal element, which is, again, another way for saying that each cell is associated with other ones with very different indices.

On the contrary, the METIS partitioning can master the complexity of the adjacency graph by clustering the cells assigned to a certain process in a single homogeneous subdomain. The main effect of this strategy is that the

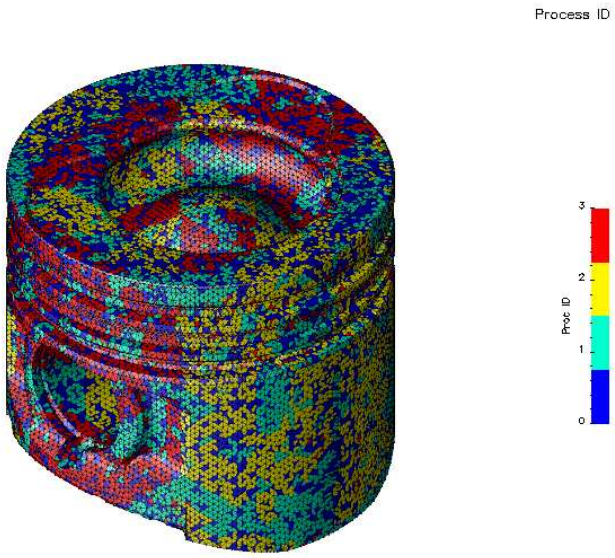


Figure 4.6 Block partitioning of a Diesel engine piston mesh.

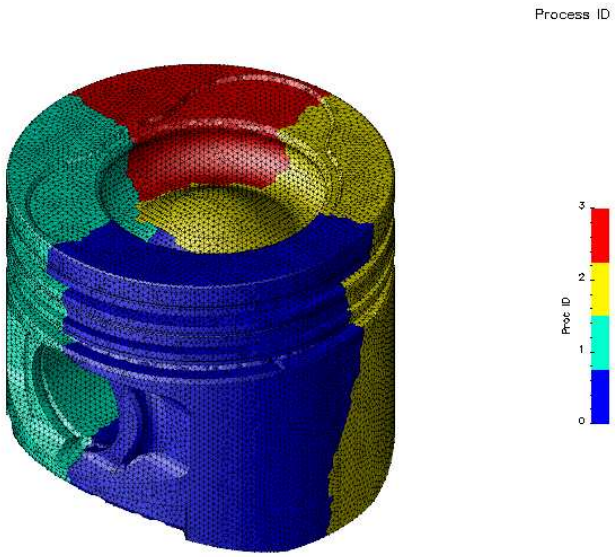


Figure 4.7 METIS partitioning of a Diesel engine piston mesh.

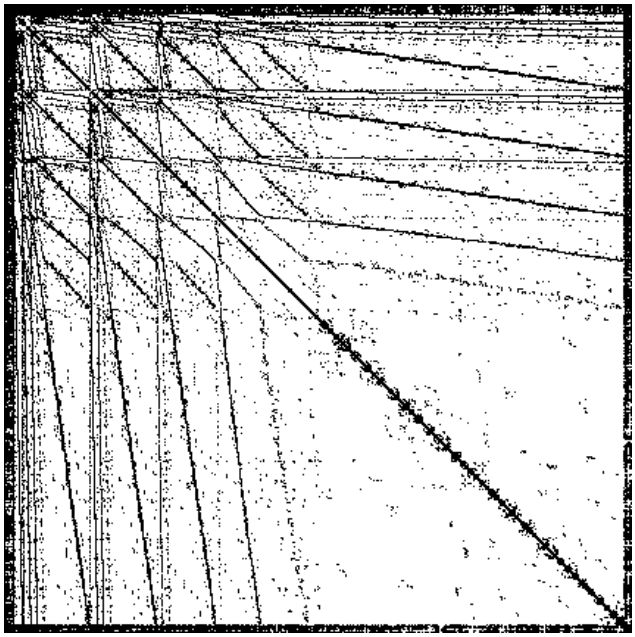


Figure 4.8 Sparsity pattern associated with the mesh of Fig. 4.6.

communication interfaces is dramatically reduced with respect to the Block decomposition.

4.1.3 ParMETIS

NEMO supports Block and ParMETIS partitioning; the latter is a partitioning tool of the METIS family. As explained by (Karypis et al., 2003), ParMETIS performs the same kind of data decomposition as METIS but with some interesting extra features. First and most important, the partitioning program is not serial, as in METIS, but rather parallel, and uses the MPI library for communication tasks. When the cost of the partitioning is very high, it can be conveniently split between the different processes of the parallel job, with consequent reduction of the load per CPU.

Second, the user can set an arbitrary number of sweeps of an additional algorithm, which can improve the quality of the decomposition. According to author's experience, METIS is better, in terms of edgecuts, than ParMETIS when no extra sweeps are applied; however, two to five iterations of the ParMETIS refining scheme are enough in order to have an equivalent interface extension in the serial and the parallel version, without a significant difference in the elapsed time.

The most attractive feature of ParMETIS is its capability to efficiently

partition adaptively refined meshes, i.e. computational grids whose topology changes a high frequency. At each call a proper routine elaborates a new decomposition, which tries to satisfy the usual criteria of graph partitioning, plus another one, consisting of minimizing the migration of nodes across processes boundaries, as required by the new subdivision.

Lastly, the ParMETIS project is more recent and better supported than METIS; because of all these reasons it represents the best choice for a complex multipurpose parallel framework such as NEMO. The support for mesh adaptation make it a powerful resource in the perspective of a possible implementation in NEMO of moving mesh schemes, like the one of Dai and Schmidt (2005).

4.2 Reordering

The *sparsity* level of a $n \times n$ square matrix A can be measured by means of two quantities: the *bandwidth* and the *profile* of A. In particular, according to Gibbs et al. (1976b) one may give the following definitions:

$$\text{Bandwidth} = \max\{|i-j| : a_{ij} \neq 0\} \tag{4.4}$$

$$\text{Profile} = \sum_{i=1}^{N} d_i, \quad d_i = i - f_i, \quad f_i = \min\{j : a_{ij} \neq 0\} \tag{4.5}$$

As previously seen in Sects. 4.1.1 and 4.1.2, the cells numbering influences the adjacency graph of the mesh, hence the pattern of the associated sparse matrix A. In the particular case of unstructured grids, commercial grid generators, such as Gambit or ICEM–CFD, produce very disordered numberings as a collateral effect of the particular meshing schemes being used. This situation can be described in terms of large bandwidth and profile of the matrix A and can harm the performance; in particular one should distinguish between serial and parallel consequences:

Serial runs: the scattered numbering is responsible for a large "cache trashing" during iterative loops on the cell index space. Cells which are geometrically near to each other do not show a corresponding adjacency in the memory storage. Therefore cache registers must be repeatedly loaded/unloaded without exploiting the presence of data already available there, stored in the same line of memory.

Parallel runs: in the case of Block partitioning, as seen in Figs. 4.4 and 4.6, a disordered numbering generates a poor homogeneity in the subdomains, with consequent increasing of interprocessor communication.

> In case of graph partitioning with METIS or ParMETIS the final quality of the decomposition is influenced by the initial node numbering.

In these circumstances it could be useful to apply a *reordering algorithm*. In particular, the one developed by Gibbs et al. (1976a) has been implemented in NEMO.[4] The results are quite impressive. For example, considering the highly sparse pattern depicted in Fig. 4.8, the application of the GPS[5] algorithm produces an outstanding reduction of the matrix bandwidth, as illustrated in Fig. 4.9.

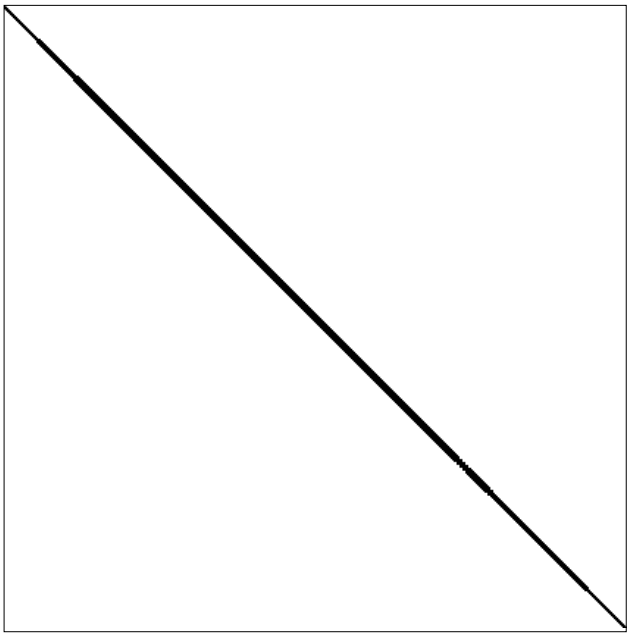


Figure 4.9 Sparsity pattern associated with the mesh of Fig. 4.6 after Gibbs–Poole–Stockmeyer's reordering.

The reordering has a positive effect also on a subsequent Block partitioning, as showed in Fig. 4.10, where the domain decomposition is much more uniform than in Fig. 4.6 and now comparable to the application of METIS, depicted in Fig. 4.7.

According to the author's experience, the GPS reordering can be useful also in case of (Par)METIS usage, because the resulting numbering works as a better starting point for the graph partitioning engine, producing a data decomposition with a better quality. In summary, the use of such a numerical

[4] An implementation of the Gibbs–Poole–Stockmeyer's algorithm is freely available at `http://www.netlib.org/toms/508`.

[5] Gibbs–Poole–Stockmeyer

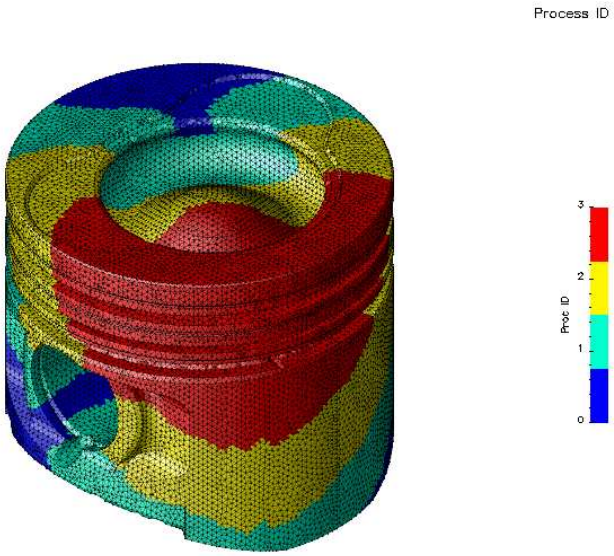


Figure 4.10 Block partitioning of a Diesel engine piston mesh after Gibbs–Poole–Stockmeyer's reordering.

tool is always recommended, in serial as well as in parallel runs. As further reference, the reader may find an interesting comparison of several reordering algorithms in the paper of Gibbs et al. (1976b).

Chapter 5

Geometry and Mesh Capabilities

5.1 Main Features

During the design of NEMO it was clear that a brand new code, written from the scratch, should be able to handle different kinds of meshes, in order to deal with the complex geometries typically encountered in engineering applications. In particular, previous experiences with the KIVA code (see Amsden et al., 1989; Amsden, 1993), had enforced the awareness that simple *structured*, *multi–block* grids would have been insufficient for challenging CCM problems.

Hence, the geometrical section of NEMO has been designed and implemented in order to support, *unstructured mesh with generic polyhedral cells*; in the next sections they will often called as *hybrid meshes*, with explicit reference to the their feature of being built with elements of different geometrical shape. Currently the grids can be made of *mixed*:

- triangles, quadrilaterals, or generic polygons (2D);
- tetrahedra, pyramids, prisms, hexahedra or generic polyhedra (3D).

Figure 5.1 shows a hybrid, two–dimensional, mesh, where quadrilaterals are used for the discretization of the boundary layer around a circular obstacle and of the far field, while the transition region in between is filled with triangles.

Figure 5.2 depicts a hybrid, three–dimensional, mesh, where one half of the body is discretized with unstructured hexahedra, while the second one with tetrahedra. Boundary layers on the cylinder surfaces are made

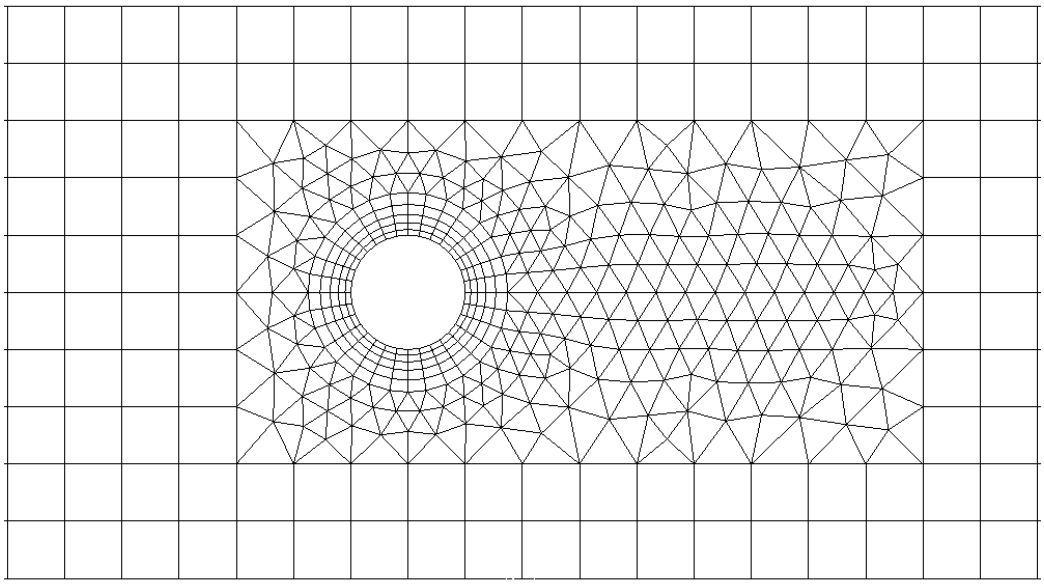


Figure 5.1 2D hybrid mesh with mixed triangles and qudrilaterals.

respectively of hexahedra and prisms; pyramids are used for the transition from the hexahedral to the tetrahedral region.

This generic approach, combining many different cell types, is based on two key features:

1. *co–located* arrangement;
2. *master–slave*, face–based approach.

The first aspect is related to the location of problems unknowns. As explained by Ferziger and Perić (2002), in a *co–located* arrangement all variables to be solved are stored on the same location, usually the center of a cell. For instance, in CFD problems with coupled variables, such as pressure and velocity, there is no need for *staggered* grids, whose building and indexing can be particularly complicated, especially with three–dimensional unstructured mesh with polyhedral cells. A co–located grid implies simpler mesh management, in terms of topology definition and metrics evaluation; the price to pay in CFD application consists in accuracy losses in the pressure–velocity coupling, which are amplified, for instance, under the action of a "checkerboard" pressure field (Patankar, 1980; Versteeg and Malalasekera, 1995). However, a remedy to this problem are ad–hoc interpolation practices, like the one proposed by Rhie and Chow (1983), universally adopted in many commercial

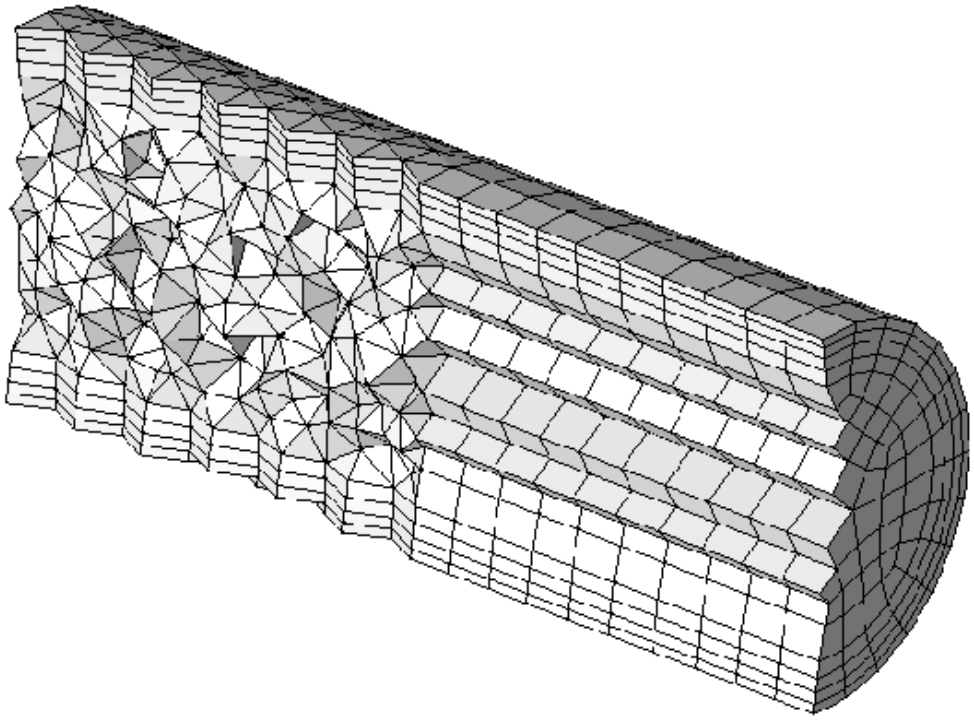


Figure 5.2 3D hybrid mesh with mixed tetrahedra, pyramids, prisms and hexahedra.

CFD software like FLUENT, STAR-CD, FIRE, which all share the co–located arrangement. Another advantage deriving by this approach is that the implementation of a moving mesh algorithm is simpler, thanks to the lack of a secondary staggered mesh which complicates the definition of the grid kinematics.

The second aspect, namely the *master–slave* approach, consists of uniquely defining, for each face, the orientation of normal vector $\boldsymbol{n}$; then, as depicted in Fig. 5.1 it is possible to identify a *master* cell (upstream to the face, according to the orientation of $\boldsymbol{n}$) and a *slave* cell (downstream).

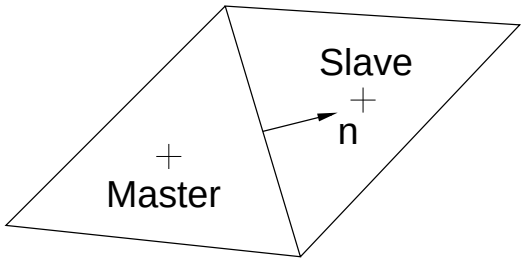


Figure 5.3 Master–slave approach for unstructured meshes.

Any computational loop involved in the discretization of a PDE operator is performed by sweeping the space of face index; then, for each face, proper contributions from the master and slave cells are accounted for and added to the corresponding coefficients on the sparse matrix associated to that PDE. This approach simplifies the computational loops and permits a general approach, independent from the particular shape and the number of faces of the polyhedral cell. This is also a consequence of the fact, that in a co–located, cell–centered, arrangement the "atomic" entity is a cell, which communicates with its neighbors through its shared faces. Information between control–volumes is exchanged through faces, therefore it is necessary to focus the attention in looping phases on faces. The validity of the master–slave approach was addressed also by Andreassi et al. (1999) in a first attempt of implementing an unstructured version of the KIVA code.

5.1.1 Structured Meshes

In a single–block *structured* mesh the relative position of a cell can be located by just just a couple of indices (i, j) (2D) or a triplet (i, j, k), because scanning a mesh for a particular cell is exactly equivalent to moving through columns and rows of matrix or a 3D array. When one must deal with more complex block structured grids, one only has to add another index, which identifies

the specific block. Lastly, the indexing can be simplified and reduced to just one subscript, according to the approach followed in the KIVA–3 (Amsden, 1993) code. Thanks to the fact that a cell in a *hexahedral–structured* meshes is always surrounded by its right, left, front, derrière, top, bottom neighbors, one can identify it just using the index of the left–bottom–front corner, as depicted in Fig. 5.1.1. In this way in the KIVA code every cell is referenced

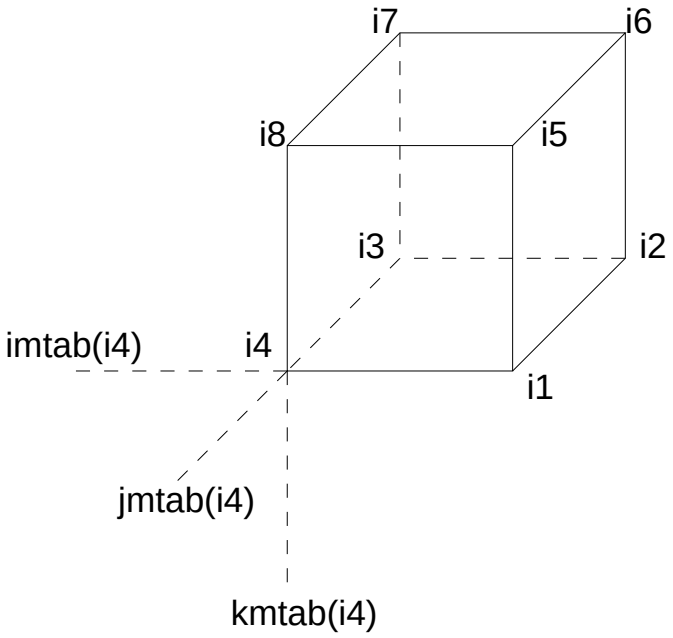


Figure 5.4 KIVA–3 connectivity stencil.

simply by the index of its $I4$ vertex. Figure 5.5 shows a typical block–structured mesh of a Diesel piston bowl used in engine cycle simulations performed with KIVA–3.

Structured or *block–structured* grids with quadrilaterals or hexahedra are handled in NEMO *as unstructured meshes*, which is quite obvious, because the data classes required for structured grids are much simpler than those for unstructured ones. If one is able to deal with the topology of generic unstructured meshes, the application to structured ones is straightforward. The opposite, of course, is not true.

The only drawback of this solution is that structured meshes in NEMO require much more information, and hence memory storage, than in KIVA, but it would have been too expensive to set up two different indexing strategies, one for structured meshes and another for unstructured ones. Moreover it is clear that block–structured meshes are not the best solution in terms of efficient meshing strategy and local refining capabilities: unstructured grids

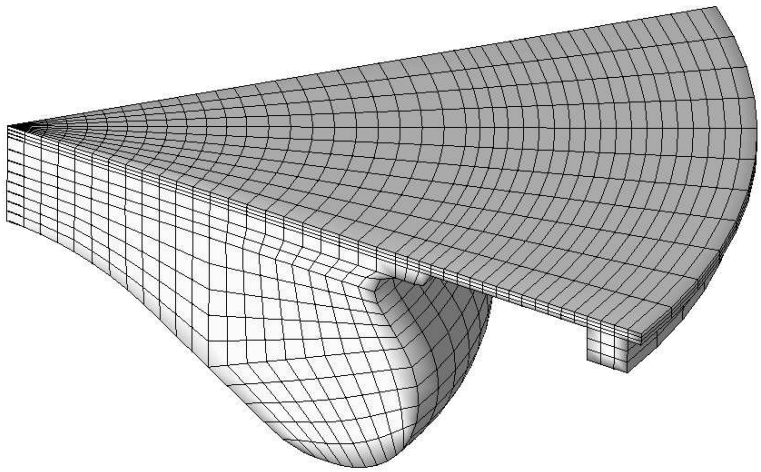


Figure 5.5 Multiblock structured grid of a Diesel piston bowl.

offer much better resources in managing the transition from coarse to fine mesh areas.

5.1.2 Unstructured Meshes

Unstructured meshes are the most flexible type of grid which can be supported by a CCM code. Their name comes from the main topology–related feature that cells and nodes cannot be addressed using the indexing of the coefficients in a matrix.

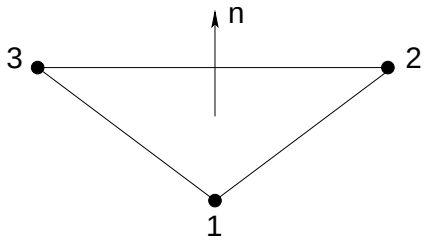
As described by Ferziger and Perić (2002), in an *unstructured* mesh, which could be made either of hexahedra, or tetrahedra, or any combination of polyhedral elements, it is impossible to recognize a single, constant, stencil connecting neighbor cells, therefore one needs many more data structures for reconstructing the topology of the mesh. In NEMO a grid is described by almost all possible connectivity relationships, that is:

- vertices to faces or v2f[1];
- faces to cells or f2c;

[1] v = vertex; e = edge; f = face; c = cell; b = boundary; v2f = *vertex to face*; f2c = *face to cell*

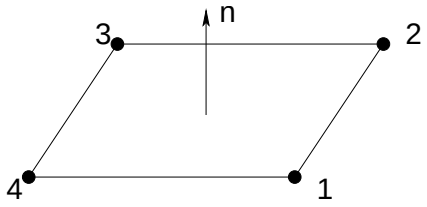
- cells to cells or c2c;
- faces to boundaries or or f2b.

The first two, namely v2f and f2c are essential and describe the sequence of vertices in every face, and of faces in every cell; the order in which vertices and faces are listed is not random, but rather uniquely defined by some conventions. Because of the lack of an official standard, the most authoritative document that one can take as reference, is the SIDS (Standard Interface Data Structures) guide of the CGNS (CFD General Notation System) [2], which defines a protocol for indexing vertices, faces, and cells in a generic mesh. Figures 5.6–5.11 depict the CGNS numbering conventions for nodes and faces of triangular, quadrilateral, tetrahedral, pyramid, prismatic and hexahedral elements, as it has been assumed in NEMO.



Face	Corner Nodes
F1	N1, N2
F2	N2, N3
F3	N3, N1

Figure 5.6 CGNS numbering convention for triangular elements.

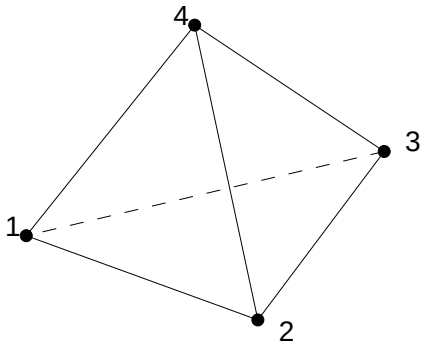


Face	Corner Nodes
F1	N1, N2
F2	N2, N3
F3	N3, N4
F4	N4, N1

Figure 5.7 CGNS numbering convention for quadrilateral elements.

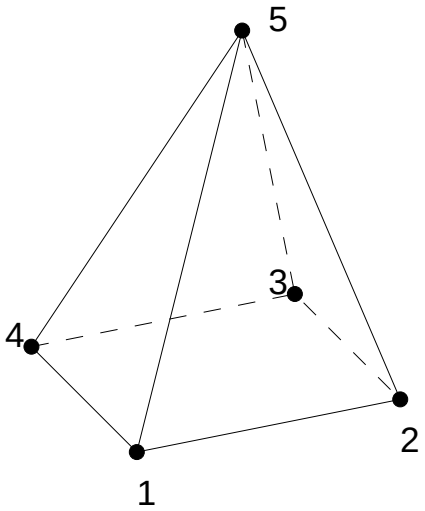
The connectivity structure c2c, i.e. "cell–to–cell", can be built starting from v2f and f2c and represents the *adjacency graph* of the mesh, i.e. the way the cells are connected with each other as previously explained in Sects. 4.1 and 4.2. This is a fundamental data structure, which reflects the sparsity pattern of the matrices arising from the discretization of the PDEs on the computational mesh, therefore, although not involved in the evaluation of

[2] `http://www.cgns.org`



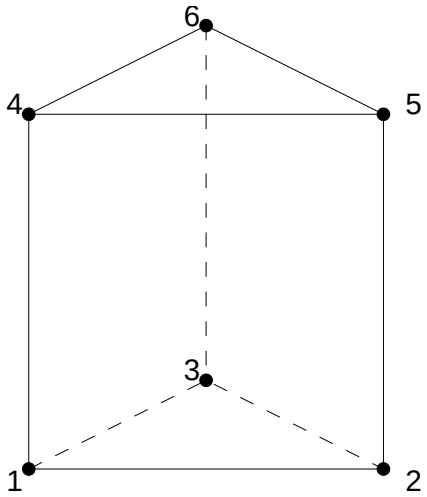
Face	Corner Nodes
F1	N1, N2, N3
F2	N1, N2, N4
F3	N2, N3, N4
F4	N3, N1, N4

Figure 5.8 CGNS numbering convention for tetrahedral elements.



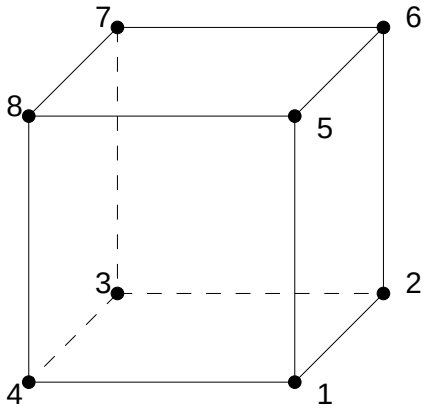
Face	Corner Nodes
F1	N1, N4, N3, N2
F2	N1, N2, N5
F3	N2, N3, N5
F4	N3, N4, N5
F5	N4, N1, N5

Figure 5.9 CGNS numbering convention for pyramid elements.



Face	Corner Nodes
F1	N1, N2, N5, N4
F2	N2, N3, N6, N5
F3	N3, N1, N4, N6
F4	N1, N3, N2
F5	N4, N5, N6

Figure 5.10 CGNS numbering convention for prismatic elements.



Face	Corner Nodes
F1	N1, N4, N3, N2
F2	N1, N2, N6, N5
F3	N2, N3, N7, N6
F4	N3, N4, N8, N7
F5	N1, N5, N8, N4
F6	N5, N6, N7, N8

Figure 5.11 CGNS numbering convention for hexahedral elements.

grid metrics, c2c results very important for other purposes such as reordering and partitioning.

Lastly, f2b permits finding the external faces lying on a specific boundary. This data structure is widely used in the application of the boundary conditions, because it is possible to define a "do-loop" on the physical boundaries and by means of f2b iterate on the respective faces.

The connectivities v2e ("vertex–to–edge") and e2f ("edge–to–face") are presently neither needed nor supported by the geometrical section of NEMO.

5.1.3 Supported Mesh Formats

Presently NEMO can import meshes in two formats:

1. Gambit Neutral Format;
2. CGNS format

Gambit is the grid generator of FLUENT; it is very powerful and can produce high quality hybrid meshes, using a user–friendly CAD interface. Meshes can be exported to a wide variety of formats, specifically tailored for FLUENT or other third–partly solvers. The most useful one for new users who want to exploit Gambit as grid generator for new applications, is the *Neutral Format*, described in Gambit User's Guide (Fluent, Inc., 2004). This is a textual format, consisting of different sections which refer respectively to: locations of vertices, v2f connectivities, subdivision of external faces into respective boundary groups. The Neutral format is well documented and easily accessible beacause of its use of ASCII. Thus it can be easily manipulated in order to extract mesh–related information. The only drawback is that the size of files can increase remarkably: for example a mesh with about 1.7 M elements uses 117 MB of disk space.

The second format supported by NEMO is the *CGNS* format. As previously discussed in Sect. 5.1.2, the CFD General Notation System (CGNS) can be considered as a standard format. It has been promoted by research centers (e.g. NASA) and private companies, which either develop (e.g. FLUENT, CD–Adapco, etc.) or use (e.g. Boeing, Pratt & Whitney, etc.) CAD/CAE applications, in order to have an easier I/O of geometrical data and solution results produced by different CCM codes and grid generators. As explained by Rumsey et al. (2005) CGNS is a binary, cross–platform format. The first feature ensures smaller sizes of the resulting files; portability is achieved by means of a proper API (Application Programming Interface), that automatically resolves also endianism issues. The development of applications able to read or write CCM data in CGNS format is possible by

means of a library of ad–hoc routines, with a Fortran and a C interface. In this way NEMO can import meshes saved in the CGNS format from every grid generator (e.g. ICEM–CFD) that supports exporting in this format. In the future it is likely that also the results of the numerical simulations carried out with NEMO will be output in CGNS format, in order to permit the compatibility with third–partly post–processing applications.

5.1.4 Current Limitations

Although NEMO's mesh capabilities can be considered advanced and innovative, especially if compared to the ones of structured codes like KIVA, one could list some possible future improvements, which would bring NEMO to the same level of many prestigious commercial competitors, in particular:

1. *non–conformal*, or *arbitrary* interfaces (Ferziger and Perić, 2002) are not yet fully supported;

2. generic *polyhedral* cells can be processed by the code, but actually there are still no *stand–alone* grid generators for such usage;

3. *moving meshes* algorithm are not yet implemented: *only static meshes* can be used.

5.2 Parallel Mesh Management

5.2.1 Mesh Broadcasting

As explained in the previous sections, the description of a unstructured grid requires different data structures and a considerable amount of memory. The first problem related to the parallel management of a mesh in CCM code involves the reading of the mesh file and the allocation of the corresponding objects representative of vertices, faces, cells, connectivities, etc. In order to make this information available on all processes one could adopt one of the following strategies:

1. the mesh file is propagated from the master node of the parallel system to all other ones, by means of a shared filesystem, such NFS[3]. All nodes execute the same set of instructions, importing the grid simultaneously.

[3] `http://nfs.sourceforge.net/`

2. the mesh file is imported by just one process, for example the process 0; then the corresponding data structures are propagated to all other processes according to the Message–Passing paradigm by means of proper *broadcast–send/broadcast–receive* procedures.

The first solution is the most straightforward because it simply requires that every process calls the procedure for mesh reading. On the other hand, it assumes the mandatory presence of a working shared filesystem. Moreover, when a parallel machine with NFS (or similar ones) is accessed simultaneously by a large number of users, the propagation of the filesystem could be excessively slow and harm the overall network performance.

The second solution requires more coding effort but, once set, is definitely safer, because the propagation of the mesh is now a part of the program execution itself, avoiding all drawbacks of the first scenario. Hence, it has been preferred in the input interface of NEMO, coherently with all other I/O tasks in the code, which do not require a shared filesystem, but perform message passing.

In the UML diagrams of the mesh–related classes analyzed in the next sections there is always a `broadcast` operation, which refers exactly to this task: the propagation of the corresponding object after being read on process 0.

5.2.2 Global to Local Reallocation

One of the most important advantages of parallel computing is that the total amount of data required by the solution of a problem can be split among the processes, so that one can save memory storage and avoid a dramatic loss of performance due to memory *swapping*. In other words, by means of a parallel code it should be possible not only to get results *more quickly* but also to solve problems with *larger* sizes.

The management of unstructured grids involves large data structures. In order to increase the dimension of solvable problems, one must save on the single node of the distributed memory machine just the topological and geometrical data related to the *local* portion of the mesh, i.e. the part of the whole grid which has been assigned to that node, according to the domain decomposition. However, the mesh–file usually refers to the whole grid, so that, after importing, all CPUs store the *global* mesh. A *global–to–local* reallocation is hence needed, in order to resize the vertex–, face– and cell–based arrays, according to the vertices, faces and cells which have been assigned to the *local* process. The identification of the local sizes is possible by means of

the PSBLAS communication descriptor, which is allocated using the partitioning results, as explained in Sect. 4.1.

It is worth remarking that in NEMO the list of *local cells* is comprehensive of the ones assigned to the current process by the data decomposition, *plus* the *halo* elements tagged during the assembling of the PSBLAS descriptor. This extension has been introduced in order to compute more easily the discretization coefficients of the *strictly local* cells, lying at a process boundary, which requires also the contribution of the cells assigned to the adjacent subdomain. If the data of the overlapped cells layer(s) on the adjacent processes are consistent, there is a redundancy which permits only local computing sections, avoiding the need of inter–processor communication.

The classes described in the next sections often contain a `glob2loc` method which performs the global to local reallocation of the corresponding objects; this task frees a large fraction of memory, allowing handling larger problems.

5.3 Object–Oriented Implementation

In the following sections the object–oriented implementation of the geometrical elements used in the definition of an unstructured mesh are analyzed. The first class devoted to this task is the Connectivity class that has already been addressed in Sect. 2.6. Figure 5.12 shows the UML diagrams of three new classes, namely Vector, Face and Cell, which are the building blocks of the class Mesh, represented in 5.16. The first three classes are absolutely independent from each other; the link to the utility class Blacs, discussed in Sect. 3.6 has been deliberately omitted, because it is common to all classes, modules and procedures of the code.

5.3.1 The Vector Class

This class has been designed for handling vector quantities such as face normals or point coordinates. The Fortran 95 module containing an implementation of the class is represented in three figures, which respectively refer to:

1. the definition of the derived data type `Vector`, Fig. 5.13;
2. generic interfaces and operator overloading, Fig. 5.14;
3. the implementation of some class operations, Fig. 5.15.

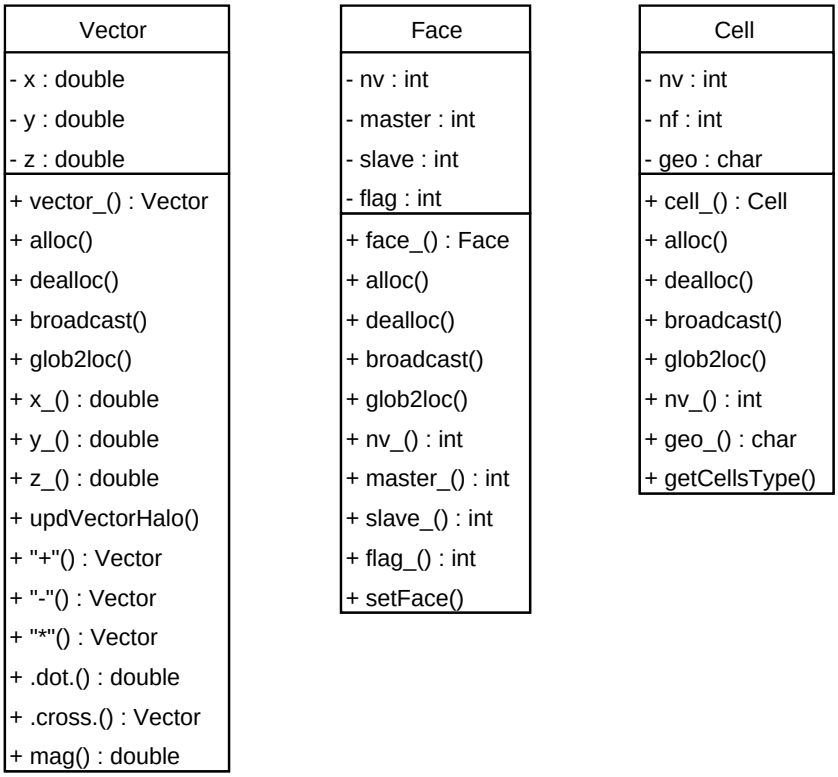


Figure 5.12 UML diagram of the Vector, Face and Cell classes.

Figure 5.13 contains the first part of the source code of the class. Attributes and methods are included in the module `class_vector`, whose access is reset from `public`, the default value for a Fortran module, to `private`, such that one has to define explicitly the elements (derived data types, as well as procedure, overloaded operators and generic interfaces) with external visibility, by means of lists beginning with the statement `public ::` . The triplet of real, double precision, variables `x`, `y` and `z` defines a new abstract data type with private access for its members.

```
module class_vector
  use class_blacs

  private ! Default
  public :: Vector
  public :: vector_, alloc, dealloc, broadcast, glob2loc
  public :: x_, y_, z_, mag
  public :: operator(+), operator(-), operator(*),&
       &    operator(.dot.), operator(.cross.)

  type Vector
     private
     real(kind(1.d0)) :: x
     real(kind(1.d0)) :: y
     real(kind(1.d0)) :: z
  end type Vector

  ! ... Generic interfaces ..

contains

  ! ... Class operations ...

end module class_vector
```

Figure 5.13 Source code of the Vector class: attributes

Figure 5.14 shows the final part of the module declarative section that ends with a list of generic interfaces and overloaded operators. The formers allow to use the same name for routines acting on different data types, according to distinct arguments lists. For example, each of the three classes

depicted in Fig. 5.12 has a constructor called `alloc`; actually this is only a generic interface for a specific routine contained in the module, such as `allocVector` for Vector class, `allocCell` for Cell and so on. The overloading of operators allows applying some intrinsic operators, such as `+`, `-` and `*`, to the vector algebra functions of sum, difference and scalar–vector product. Similarly new ones, such as `.dot.` and `.cross.`, are defined in order to use the notation of binary operators, `c = a .op. b` for dot–product and cross–product. This provides a high level notation which allows the programmer to manipulate Vector objects, avoiding the traditional Fortran syntax for functions call.

Figure 5.15 depicts the source code for a selection of the class procedures; the full list is included in the UML diagram of Vector class in Fig. 5.12. According to the naming conventions explained in Sect. 2.7, `vector_` is the default public constructor while `x_`, `y_` and `z_` are the elemental (see Akin, 2003) getter functions for the class members.

The operation `alloc` allocates the memory needed for a pointer of type `Vector` with rank 1, storing for instance the coordinates of all mesh nodes; `dealloc` provides the corresponding deallocation; `Broadcast` transmits an object from process 0, where it has been previously imported/created, to all the other ones of the parallel job; `glob2loc` resizes a *global* array of type `Vector` according to the global–to–local mapping.

Together with the definition of the `Vector` data type, the module provides also an implementation of classic vector algebra functions. Considering the vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$, and the scalar r the following operations are supported:

- $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$
- $\boldsymbol{c} = \boldsymbol{a} - \boldsymbol{b}$
- $\boldsymbol{c} = r\boldsymbol{a}$
- $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b}$
- $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$
- $r = |\boldsymbol{a}|$

The functions `dotProd` and `crossProd` are invoked throughout the code by means of the operators `.dot.` and a`.cross.` previously defined. In this way, in order to compute the cross product one has to use:

```
c = a .cross. b
```

instead of

```
module class_vector

  ! ... Type Vector Declaration

  interface alloc
     module procedure allocVector
  end interface
  interface dealloc
     module procedure deallocVector
  end interface
  interface broadcast
     module procedure broadcastVector
  end interface
  interface glob2loc
     module procedure glob2locVector
  end interface
  interface operator(+)
     module procedure sumVect
  end interface
  interface operator(-)
     module procedure difVect
  end interface
  interface operator(*)
     module procedure scalVectProd
  end interface
  interface operator(.dot.)
     module procedure dotProd
  end interface
  interface operator(.cross.)
     module procedure crossProd
  end interface
contains
  !...
end module class_vector
```

Figure 5.14 Source code of the Vector class: interfaces and operator overloading.

```
elemental function vector_(x,y,z)
  type(Vector) :: vector_
  real(kind(1.d0)), intent(in) :: x, y, z
  vector_ = vector(x,y,z)
end function vector_

elemental function x_(vect)
  real(kind(1.d0)) :: x_
  type(Vector), intent(in) :: vect
  x_ = vect%x
end function x_

function sumVect(a,b)
  type(Vector) :: sumVect
  type(Vector), intent(in) :: a, b
  sumVect%x = a%x + b%x
  sumVect%y = a%y + b%y
  sumVect%z = a%z + b%z
end function sumVect

function dotProd(a,b)
  real(kind(1.d0)) :: dotProd
  type(Vector), intent(in) :: a, b
  dotProd = a%x * b%x + a%y * b%y + a%z * b%z
end function dotProd

function crossProd(a,b)
  type(Vector) :: crossProd
  type(Vector), intent(in) :: a, b
  crossProd%x = a%y * b%z - a%z * b%y
  crossProd%y = a%z * b%x - a%x * b%z
  crossProd%z = a%x * b%y - a%y * b%x
end function crossProd
```

Figure 5.15 Source code of the Vector class: vectorial operations

```
c = crossProd(a,b)
```

All operations defined in this module are widely used in the sections of the code where the metrics of the mesh (distances, face areas, cell volumes, etc.) is computed. By means of the operator overloading, complex expressions can be written according to a code syntax very similar to the corresponding hand–written formulas.

5.3.2 The Face Class

This class describes a cell face; attributes and operations are listed in the UML diagram of Fig. 5.12 and one can easily figure out their meaning by their names.

Attributes

- `nv` represents the number of vertices of a face.
- `master` and `slave` store, respectively, the index of the master and the slave cell, uniquely defined by the orientation of the face normal, as depicted in Fig. 5.1; by default boundary faces have normals pointing outwards of the domain and their `slave` index is set to 0. If a face is shared by a halo and an internal cell, of the adjacent process `master` is set to the halo index, while `slave` is set to a negative value.
- `flag` identifies the boundary surface which a face belongs to, ranging from 1 to the number of applied BCs. Internal faces (often referred as "fluid" faces) have `flag` set to 0.

Operations

- `face_` is the default public constructor.
- `alloc` and `dealloc` are the rank–1 constructor and destructor.
- `broadcast` and `glob2loc` implements the broadcasting and the global–to–local reallocation of `Face` objects.
- `nv_`, `master_`, `slave` and `flag_` are getters, defined as `elemental` functions in order to be applied to both rank–0 and rank–1 objects.
- `setFace` is a setter with optional arguments, used for modify some attributes, for instance during the re–assignment of `master` and `slave` indices in the `glob2loc` procedure.

5.3.3 The Cell Class

This class models the properties of a "cell" object. Its UML diagram is depicted in Fig. 5.12, together with the one for Vector and Face classes. The meaning of attributes and operations is quite immediate.

Attributes

- `nv` represents the number of vertices of a cell.
- `nf` indicated the number of faces of a cell.
- `geo` is a string of three characters, which describes the geometrical shape of a cell. Supported values are `tri` (triangle), `qua` (quadrilater), `tet` (tetrahedron), `pyr` (pyramid), `pri` (prism), `hex` (hexahedron).

Operations

- `cell_` is the default constructor.
- `alloc`, `dealloc`, `broadcast`, `glob2loc` have the same meaning of the corresponding methods in Face class.
- `nv_` and `geo_` are getter functions. A generic interface for `nv_` is also defined in order to use the same name with objects of both Face and Cell classes.

5.3.4 The Mesh Class

The Mesh class is the most important class among the ones involved in the definition of the geometry and topology of the computational grid. Its UML diagram is depicted in Fig. 5.16.

A first remarkable feature is that the class attributes have *public* visibility, instead of being kept private. Coherently with the issues addressed in Sect. 2.5, this strategy has been adopted for performance reasons, in order to have a straightforward access to the class data structures, avoiding the need of implementing getter functions and making local copies of the information carried by a Mesh object.

Geometrical information such as face normals, areas, cell volumes, among others, which is stored in proper arrays easily recognizable in the list of the class attributes, must be accessed repeatedly at each time–step or big–iteration of the code. This is a classic circumstance where the level of abstraction must be carefully considered, in order to avoid performance losses.

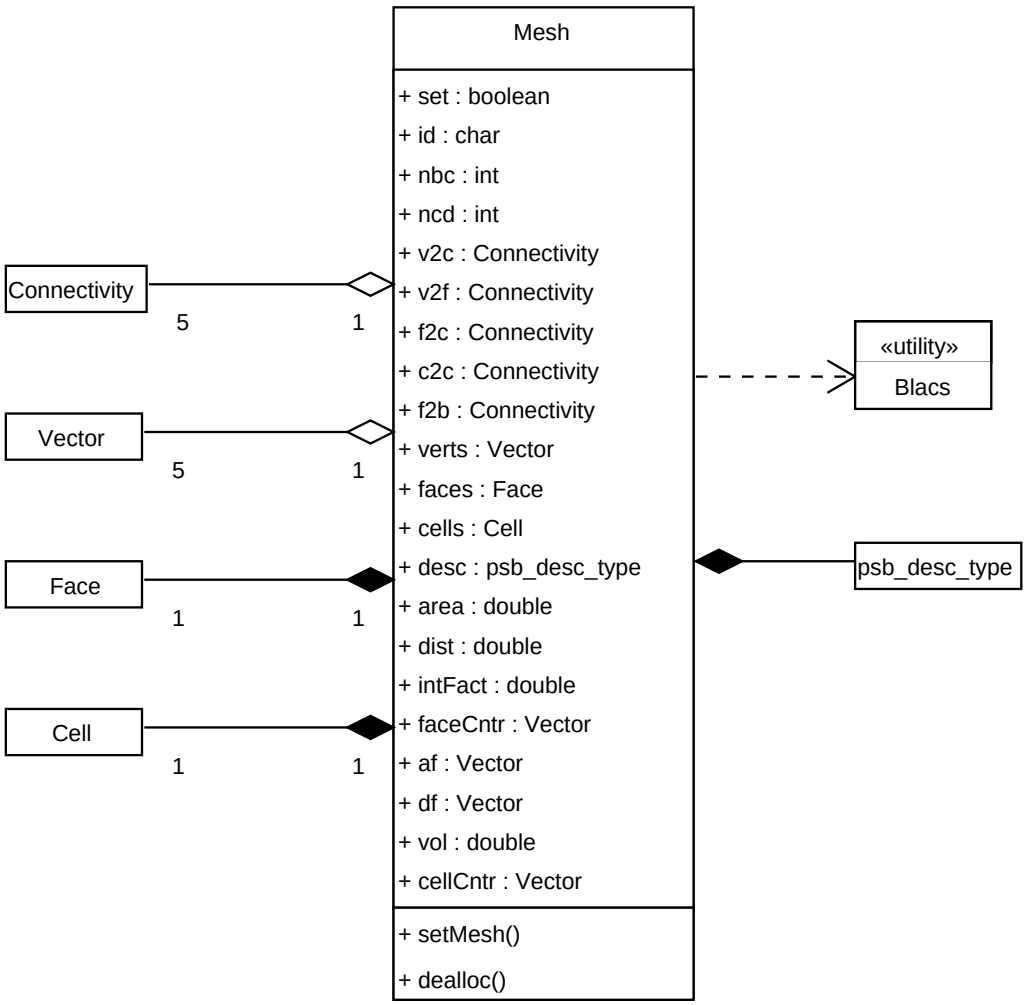


Figure 5.16 UML diagram of the Mesh class.

Moreover the object–oriented formulation should always lead to a rationalization of the code, not to useless complications, as it would have been in the case of private access.

Attributes

- `set` is a boolean variable which assumes value `.true.` right before the return from the subroutine `setMesh`.
- `id` contains the name of the mesh.
- `nbc` is the number of boundary surfaces flagged during the mesh generation.
- `ncd` express the dimensions of the problem (2D or 3D). This variable is used by many other procedures, where an algorithm can distinguish between 2D and 3D mesh.
- `v2c`, `v2f`, `f2c`, `c2c` and `f2b` are attributes of Connectivity class.
- `verts`, `faces` and `cells` are rank–1 arrays respectively of Vector, Face and Cell classes and store the vertex coordinates and topological information of faces and cells.
- `desc` is the PSBLAS descriptor; it is allocated right after the domain decomposition and is shared by all sparse matrices which arise from the discretization of PDEs on the same mesh.
- the remaining class members contains the metrics respectively of the faces (`area`, `dist`, `intFact`, `faceCentr`, `af`, `df`) and the cells (`vol`, `cellCentr`).

Operations

- `setMesh` performs a complex series of tasks:
  - reads mesh–related parameters from input file;
  - imports the mesh in `.neu` or `.cgns` format;
  - broadcasts grid components previously read such as vertices coordinates, Face, Cell and Connectivity objects;
  - builds the adjacency graph `c2c`;
  - renumbers the cells, if requested from the user;

  - performs the domain decomposition;
  - executes the global–to–local reallocation;
  - evaluates geometrical quantities (distances, areas, volumes).

- `dealloc` frees the memory storage used by a Mesh object.

It is worth noting that the UML diagram of Mesh class in Fig. 5.16 introduces two kinds of relationships not yet encountered: *aggregation* and *composition*. They both express an "has–a" link between Mesh and another class, so that one may say that "Mesh has a Face" (meaning a rank–1, `Face` type, array). As reported by Fowler and Scott (2003), the main difference between aggregation (the line ends with a white diamond) and composition (the line ends with a black diamond) is that an aggregated class (like Vector) can have an independent and self–standing instance, while a Face object can be instantiated only as a part of a Mesh.

Chapter 6

PDE Solver

6.1 High Level Interface

In Chap. 5 the mesh capabilities of NEMO have been addressed, and the basic classes related to topology and geometry, such as Connectivity, Vector, Face, Cell and Mesh have been discussed and analyzed in detail.

This chapter will consider the most important classes related to the main task of the code: the solution of generic PDEs by means of a finite volume method.

As previously discussed in Chap. 1, the object–oriented approach to a CCM problem is focused on the description of the most important entities involved in the mathematical modeling of the physical phenomenon. The operations which perform the PDEs discretization and the solution of the corresponding algebraic equations are hidden behind a high–level front–end, that splits the external interface from the internal code implementation.

The basic idea behind NEMO, inspired by the OpenFOAM code, is to reproduce the process of "*writing*" a PDE, by algebraically adding many differential operators acting on scalar or vector fields. Each operator represents an independent contribution to the algebraic tensor (i.e. the sparse matrix A) corresponding to the physical system described by the PDE; the application of sources and sinks in the PDE is translated into the sum of terms in the RHS. The solution of the equation is accomplished by preconditioning and solving the resulting linear system $Ax = b$. In summary, following an object–oriented formulation, the following statement:

> "Solve the PDE Eqn in the unknown field Fld, according to the set of the boundary conditions Bc"

is translated into an algorithm and the corresponding source code, according to the sequence depicted in Fig. 6.1.

1. Proposition:

"Solve the PDE Eqn in the unknown field Fld, according to the set of the boundary conditions Bc"

2. Algorithm:

$$
\begin{aligned}
&Eqn \Leftarrow \pm \mathrm{op}_1(Fld)\\
&Eqn \Leftarrow \pm \mathrm{op}_2(Fld)\\
&\ldots\\
&Eqn \Leftarrow\!= \mathrm{op}_n(Fld)\\
&\ldots\\
&Eqn \Leftarrow + \mathrm{op}_{Src}(Src)\\
&\ldots\\
&\text{Solve } Eqn \text{ in } Fld, \text{ using } bc.
\end{aligned}
$$

3. Code:

```
call pde<Operator-1>('+',...,Fld,Eqn)
call pde<Operator-2>('-',...,Fld,Eqn)
!...
call pde<Operator-n>('=',...,Fld,Eqn)
!...
call pdeSource('+',Src,Eqn)
!...
call pdeSolve(Eqn,Fld,Bc)
```

Figure 6.1 Prototype of the high–level interface of the PDE solver.

6.2 Classes Implementation

From the previous considerations it is clear that the main role in the kernel of the code is played by the following entities:

- *Field*;
- *Pde*;
- *Boundary Condition*;
- *Differential Operators*.

The first three items are implemented in NEMO as classes and they will be discussed in the next sections, together with the differential operators presently supported in the code.

6.2.1 The Field Class

The UML diagram of the Field Class is depicted in Fig. 6.2. It has been designed as an *abstract class* (see Fowler and Scott, 2003), i.e. a class which cannot be instantiated directly, but only by means of its subclasses. One could also say that ScalarField and VectorField *realize* the parent class Field; in practice this means that only the interfaces of the class operations are defined in the superclass, while the specific implementation is delegated to the subclasses. An *abstract class* defines not only a set of interfaces but also[1] some attributes, such as `dim`, `material` and `mesh`, which are transmitted by inheritance to all subclasses. The UML symbol for the *realization* relationship is a dashed line that ends with a triangular white arrow. Italics are used in the parent class name for specifying the *abstract* stereotype.

Attributes

- `dim` is a member of the Dimensions class (also depicted in Fig. 6.2), and it stores the physical dimensions of the quantity represented by a Field object. It is used for the dimensional analysis carried out during the application of the differential operators.
- `material` is a pointer to a Material object.
- `msh` is a pointer to the Mesh object describing the computational domain where a Field has been defined.

[1] This additional feature represents the main difference between abstract classes and interfaces in OOP.

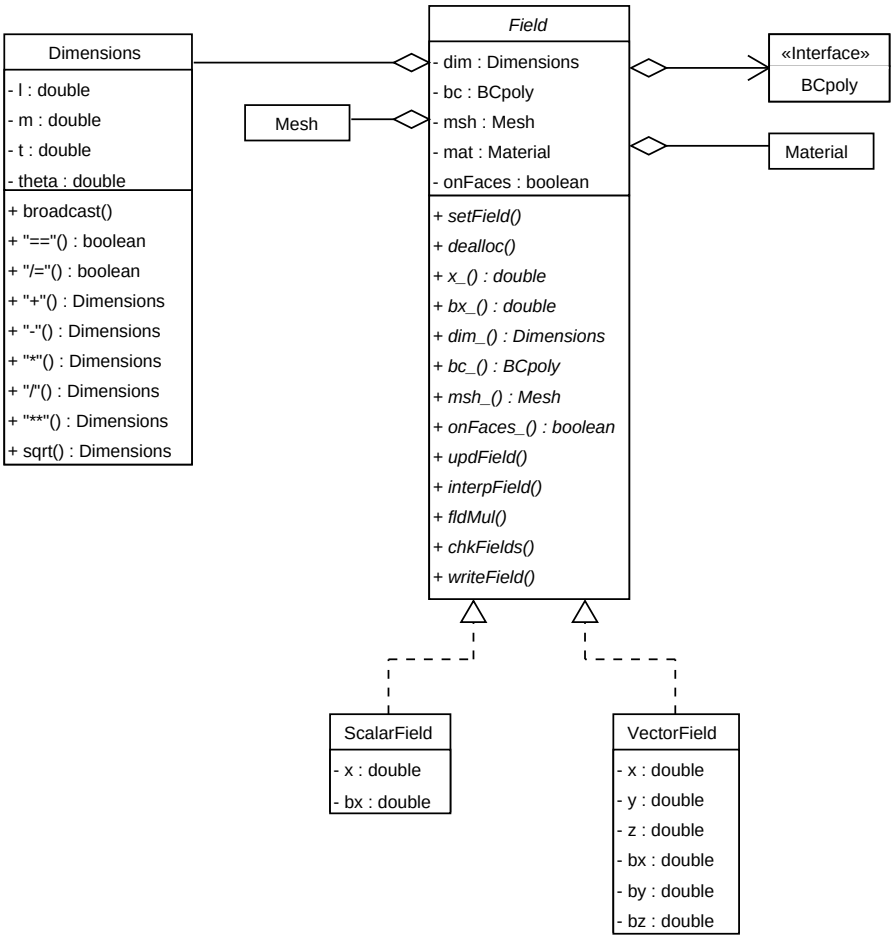


Figure 6.2 UML diagram of the Field class.

- `onFaces` is a boolean flag which specifies whether the field values are either cell–centered (`.false.`) or face–centered (`true`).
- In FieldScalar, `x` and `bx` store the values assumed by the field on the interior domain and on the boundary faces respectively. Similarly, in FieldVector, `x`, `y`, `z` and `bx`, `by`, `bz` contain the same kind of information, referring to the three cartesian components.

Operations

- `setField` and `dealloc` are, respectively, the constructor and the destructor of the class.
- `updFld` is called for enforcing the consistency between the field values on halo elements of adjacent processes and for updating the values on boundary faces, according to the prescribed BCs and the interior solution.

6.2.2 The Pde Class

The UML diagram for the Pde Class is depicted in Fig. 6.3. Similar to the Field class, it has been designed as an abstract class, whose realizations are respectively the ScalarPde and the VectorPde classes. The former is used for scalar instances, while the latter for vector ones. A Pde object encapsulates the linear system arising from the discretization of a specific governing equation. The finite volume method has been applied such that vector equations are translated to multiple linear systems, having the same coefficient matrix A, but different RHSs for each cartesian component in the x, y and z directions. When a differential operator is applied to a Field in a Pde object, the coefficients corresponding to the FV discretization of that operator are evaluated and inserted into the sparse matrix A and the RHS(s) b_i, members of the Pde object, according to the PSBLAS building steps described in Sect. 3.5.4.

Attributes

The following attributes are common to both ScalarPde and VectorPde subclasses:

- `dim` is a member of the Dimensions class, represented in Fig. 6.2, and stores the physical dimensions of a Pde object. When a differential operator is applied to a Field, the resulting term must dimensionally match the Pde.

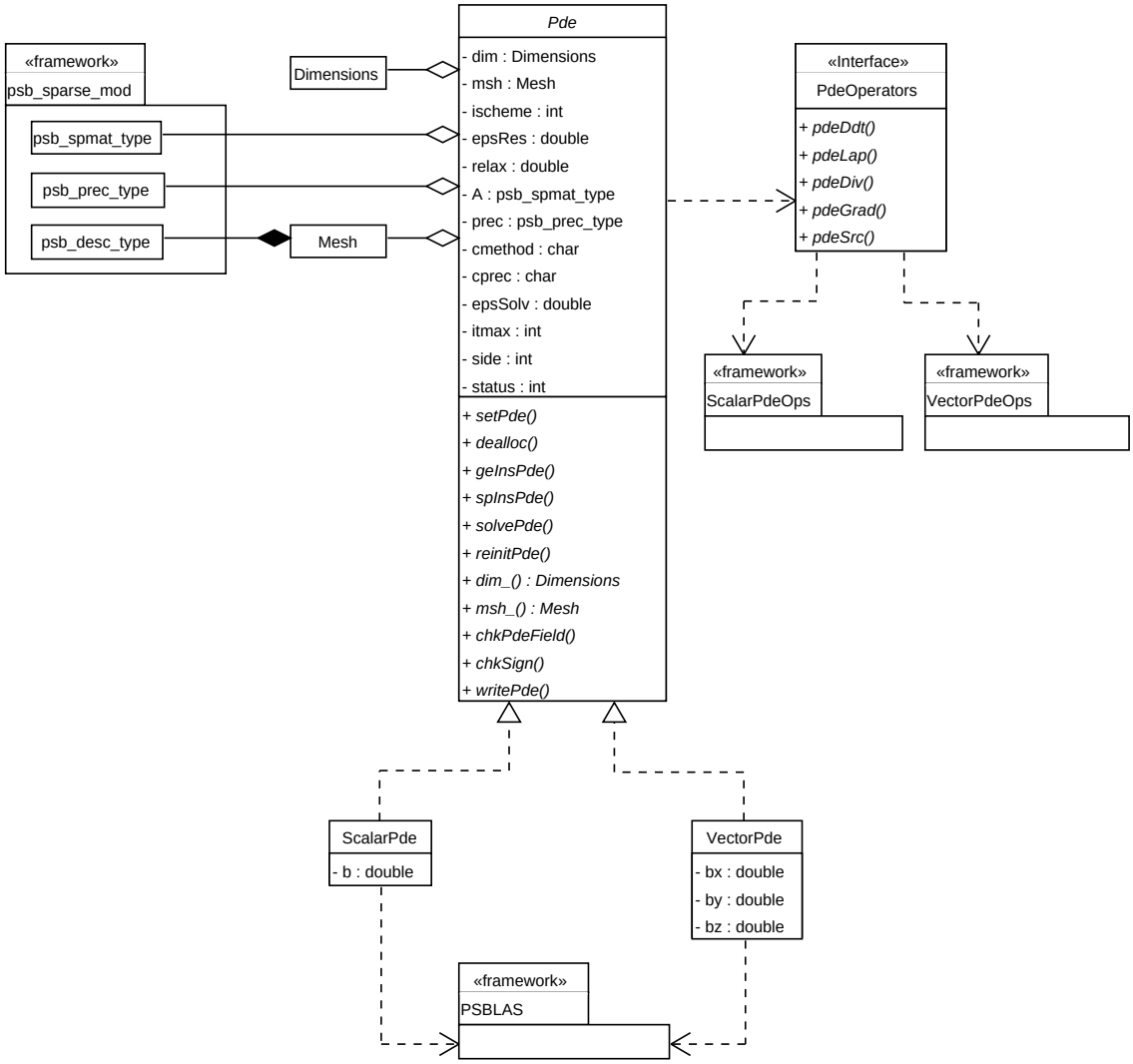


Figure 6.3 UML diagram of the Pde class.

- `msh` is a pointer to the Mesh object, which contains all data related to the computational domain.
- `ischeme` is an integer, which permits switching between different discretization schemes, such as "upwind", "central difference", etc.
- `epsRes` is a tolerance used for checking the convergence of an iterative refinement process.
- `relax` represents the under–relaxation factor.
- `A` is a member of type `psb_spmat_type` (see 3.5.3) and stores the PDE sparse matrix in PSBLAS format.
- `prec` is a member of type `psb_prec_type` and stores the preconditioner of matrix A computed by the PSBLAS library.
- `cmethod` and `cprec` are character variables which defines the solver and the preconditioner to be used.
- `epsSolv` and `itmax` contain the tolerance and the maximum number of iterations fixed by the user for the PSBLAS iterative methods.
- `side` is an integer flag which states whether the Pde building process is occuring at the LHS or RHS of the equation.
- `status` is an integer flag which specifies whether the Pde, and thus the `pde%A` member, are in "build", "assembled" or "update" status (see 3.5.4).

As depicted in the attribute compartments of the two subclasses ScalarPde and VectorPde, the RHSs are stored respectively in `b` for scalar objects, and in `bx`, `by` and `bz` for vector ones.

Operations

- `setPde` and `dealloc` are respectively the class constructor and destructor.
- `geInsPde` and `spInsPde` are wrappers to the corresponding PSBLAS procedures `psb_geins` and `psb_spins`. The use of such high–level "shells" is necessary in order to allow the PSBLAS routines to access the private class members `pde%A` and `pde%b`.
- `solvePde` performs the following steps:

  - PSBLAS assembling of the `pde%A` sparse matrix and the `pde%b` dense vector;
  - preconditioning of `pde%A`;
  - solution of the linear system $Ax = b$;
  - updating of the unknown Field associated with the Pde object.

- `reinitPde` reinitializes the flags `pde%side` and `pde%status` for successive iterations and works as wrapper for PSBLAS `psb_sprn` routine that regenerates a sparse matrix (see 3.5.4).
- `dim_` and `msh_` are gettters for the corresponding attributes.
- `chkPdeField` checks for the consistency of a Pde and a Field objects (same mesh, same variable location, same material, etc.)
- `chkSign` returns the proper sign of the coefficients to be inserted into the sparse matrix `pde%A` and the dense vector `pde%b`.
- `writePde` writes `pde%A` and `pde%b` in COO (Coordinates) format, for possible post–processing of the linear system $Ax = b$.

As depicted in Fig. 6.3 Pde objects are modified by means of the methods defined in the PdeOperators interface. This one collects the corresponding procedures for scalar and vector differential operators under the same name, as implemented in the ScalarPdeOps and VectorPdeOps frameworks respectively.

6.2.3 The BC Class

The UML diagram of the BC class is depicted in Fig. 6.4. According to the UML classification this class is an *interface* class, i.e. it declares a set operations with a well–defined syntax and I/O parameter list. Actually, the definition of the class attributes and the effective implementation of its operations are delegated to the subclasses which realize the different types of boundary conditions, such as: “wall”, “inlet”, “outlet”, etc.

The main difference, with respect to the abstract classes Pde and Field, is that in these cases the behavior as either a scalar or a vector object is defined at compile time. For instance, if the programmer defines `Fld` as a `type(ScalarField)` variable, it will be impossible to change the features of this entity and use it as a `type(VectorField)` variable at runtime. On the contrary, at compile time it is impossible to know in advance whether, for

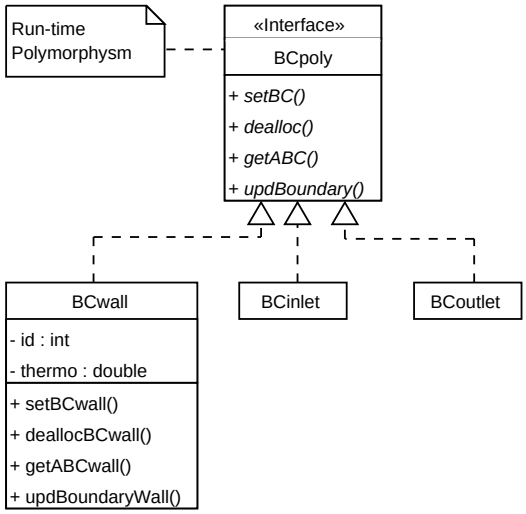


Figure 6.4 UML diagram of the BCpoly class.

example, the boundary condition[2] no. 1 will be a "wall", or an "inlet", etc. One way to handle this degree of freedom is to allow a generic, *polymorphic*, data structure to assume the features and support the actions of a particular boundary condition just after its assignment, at runtime. Depending on the problem, each BC object will then become a BCwall, or a BCinlet, etc.

In this way the class BC realizes what is called *run–time polymorphism* or *dynamic binding*. As previously discussed in Sect. 2.2.3 this feature is not natively supported by the Fortran 95 dialect (it will be available with Fortran 2003), but it can be emulated, as explained by Decyk et al. (1998) and Akin (2003). First of all, one has to implement the subclasses BCwall, BCinlet, etc. This implies the writing of different Fortran 95 modules, called `class_bc_wall`, `class_bc_inlet`, defining the abstract data types `BCwall`, `BCinlet` and their respective methods. Then one can build the BC superclass consisting of an abstract data type `type(BCpoly)` whose components are pointers to possible instances of all subclasses.

Figure 6.5 shows the upper section of the BCwall class implemented in the `class_bc_wall` module. Attributes and operations are consistent with the corresponding UML diagram reported in Fig. 6.4.

[2]In NEMO the boundary conditions are numbered from `1` to `nbc`.

```
module class_bc_wall

  private ! Default
  public :: BCwall                    ! Class
  public :: setBCwall, deallocBCwall ! Constructor/Destructor
  public :: getABCwall                ! Getter
  public :: updBoundaryWall           ! Updater

  type BCwall
     private
     integer :: id(1)
     ! THERMAL section
     real(kind(1.d0)) :: thermo(2) = 0.d0
  end type BCwall

  ! THERMAL section
  ! id | Description | thermo(1)      | thermo(2)
  !  1 | const temp  | temp  [K]      |    --
  !  2 | adiabatic   |    --          |    --
  !  3 | const flux  | flux  [W/m2]   |    --
  !  4 | convection  | coeff [W/m2 K] | temp [K]

contains

  ! Operations...

end module class_bc_wall
```

Figure 6.5 Source code of the BCwall class: attributes.

```
module class_bc

  use class_bc_inlet
  use class_bc_wall
  ! Other BC classes...

  private ! Default
  public :: BCpoly          ! Class
  public :: setBC, dealloc ! Constructor/Destructor
  public :: getABC          ! Getter
  public :: updBoundary     ! Updater

  type BCpoly
     private
     type(BCpoly), pointer :: inlet => null()
     type(BCwall), pointer :: wall => null()
  end type BCpoly

contains

  ! Operations...

end module class_bc
```

Figure 6.6 Source code of the BC class: attribute polymorphism.

The source code of the polymorphic class is depicted in Fig. 6.6. Attributes and methods of all boundary conditions subclasses (i.e. BCwall, BCinlet, etc.) are accessed by means of the statements:

```
use class_bc_inlet
use class_bc_wall
! Other BC classes...
```

The new polymorphic abstract data type is defined in the `type(BCpoly)` section. In particular there is one pointer to each BC–specific data type and by default they are all nullified.

Usually, in a NEMO–based application all boundary conditions are stored in a `BC` variable of rank 1 and type `BCpoly`, dynamically sized according to the actual number of boundary conditions.

```
use class_bc
[...]
type(BCpoly), pointer :: bc(:) => null()
[...]
allocate(bc(nbc)) ! nbc is the number of boundary conditions
```

During the boundary condition assignement the code simply allocates a proper BC–specific variable and then makes the corrisponding element in the polymorphic object point to it. For instance, if the boundary condition no. 1, stored in the variable `bc(1)` has to be a BCwall, the program first creates a `BCwall` object and then executes the statement:

```
bc(1)%wall => BCwall_object
```

Since all attributes of the polymorphic class point initially to `null()` (see Fig. 6.6), the application of the Fortran 95 intrinsic function `associated()` returns value `.false.` for every members of `bc(1)`, except for `bc(1)%wall`. This feature allows emulating the dynamic binding, not only for the class attributes, but also for the methods. The `BC` interface represented in the UML diagram of Fig. 6.4 prescribes some operations such as `setBC`, `getABC`, etc., whose corresponding realizations in the subclasses BCwall and BCinlet are respectively `setBCwall`, `getABCwall`, and `setBCinlet`, `getABCinlet`. In order to automatically call the right operations, according to the actual status of the polymorphic object, every method in the `class_bc` module consists of a selector that switches to the corresponding proper BC–specific subroutine on the basis of the results returned by the `associated` function (see Fig. 6.7).

```
subroutine setBC(bc,...)
  type(BCpoly), intent(inout) :: bc

  if(     associated(bc%wall)   ) then
     call setBCwall(bc%wall,...)
  elseif( associated(bc%inlet)  ) then
     call setBCinlet(bc%inlet,...)
  end if

end subroutine setBC
```

Figure 6.7 Source code of the BC class: method polymorphism.

As explained by Decyk et al. (1998) this strategy also permits a flexible extension of the specific subclasses grouped by the polymorphic class. For instance, if one has to add the support to another kind of boundary condition, namely `BCnew`, one has to perform the following steps:

- defining the specific abstract data type `type(BCnew)` (attributes and operations) in a new module `class_bc_new`;
- adding the statement `use class_bc_new` into the module `class_bc`;
- adding the member `type(BCnew), pointer :: new => null()` to the definition of `BCpoly`;
- adding to every class operation the conditional statements

```
elseif( associated(bc%new)) then
   call <operation>BCnew(bc%new)
```

6.2.4 Physical and Numerical Boundary Conditions

In the previous section, discussing the object–oriented implementation of the boundary conditions, there are multiple references to the `getABC` operation. `ABC` stands for "Analytic Boundary Condition" and for the triplet of real values, namely a, b, c, returned by the subroutine itself. This procedure (the polymorphyc version, as well as its BC–specific realizations) is strictly connected to the design issue of translating a physical boundary condition to its numerical counterpart.

For sake of simplicity one can consider a boundary condition of type "wall" in a thermal problem. In particular, one can set one of the following configuration:

- prescribed temperature;
- adiabatic wall;
- prescribed heat flux;
- prescribed heat exchange.

As showed in Fig. 6.5 this is done by assigning a proper value to the `BCwall%id` member and storing into the `Bcwall%thermo` array the possible values of temperature, flux, heat–exchange coefficient. In summary, this is how a *physical* boundary condition is stored in memory.

However, in the source code of the differential operators the application of the boundary conditions should be implemented following a generic interface, not constrained by the their actual physical essence. At this point, one should note that fixing a boundary condition for a generic scalar ϕ it is equivalent to specifying that a combination of the quantity and its normal derivative must be equal to a given value. In other words, one can always translate a *physical* constraint into a *numerical* one such as:

$$a\phi + b\frac{\partial \phi}{\partial n} = c \tag{6.1}$$

where n refers to the direction normal to the external surface. In literature (e.g. see Tannehill et al., 1997) this is called a *Robin* boundary condition, which reverts to *Dirichlet* and *Neumann* type respectively for $b = 0$ and $a = 0$. Every boundary condition can be described by means of a particular set of the coefficients a, b and c.

Using this mutual correspondence between *physics* and *numerics*, each differential operator has been implemented by separating the two layers; in particular, the code performs the following steps:

1. for every boundary condition it calls the `getABC` polymorphic operation;
2. trough the corresponding actual realization, for instance `getABCwall`, it gets the triplet (a, b, c) which describes *numerically* the boundary condition;
3. it generates proper, operator–dependent, source terms which are function of a, b, c and completely unaware of the underlying boundary condition physics.

6.2.5 The Source Class

The UML diagram of the Source class is depicted in Fig. 6.8. Source objects are used for modeling linear sources of a scalar quantity ϕ such as:

$$S_\phi = S_c + S_P\phi \tag{6.2}$$

The implementation is quite simple and consists of the following attributes and operations.

Attributes

- `dim` is a member of the Dimensions class and represents the physical dimensions of the source term.
- `sc` and `sp` contain the two coefficients of the linear approximation 6.2.

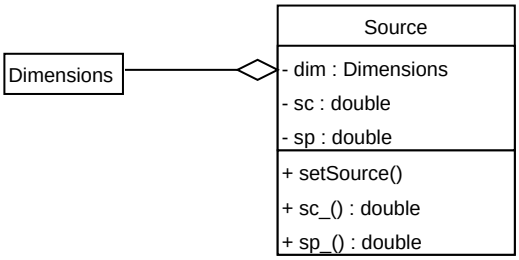


Figure 6.8 UML diagram of the Source class.

Operations

- `setSource` is the class constructor.
- `sc_` and `sp_` are getter functions for the corresponding class attribnutes.

The application of a source term to a Pde object, in terms of contributions to the `pde%A` and `pde%b` members is discussed in Sect. 6.4.3.

6.3 Operators Acting on Field Objects

The following section deals with the implementation of several operators acting on Field objects. Each one can be described by means of an *interface* and an *implementation*, the latter being related to a particular solution method. Presently, NEMO supports only a finite volume approach, but it is possible that in the future also different discretization techniques, such as the finite element method, will be implemented and encapsulated by using a common high level interface, shared by all approaches (e.g. FVM, FEM).

6.3.1 Gradient

As explained by Ferziger and Perić (2002) and by Versteeg and Malalasekera (1995), the application of the finite volume method to the term $\operatorname{grad}\phi$, where ϕ is a generic scalar, involves the integration of the differential term over a generic control volume.

One can assume a *linear* [3] variation of $\operatorname{grad}\phi$ over the CV and set it equal to $(\operatorname{grad}\phi)_P$, where P is the centroid of the corresponding cell, thus

[3] We define the centroid $\boldsymbol{x_P}$ of a cell with volume V_P as its center of mass, so that there is no first–order moment around it:

$$\boldsymbol{x_P} = \frac{1}{V_P}\int_{V_P} \boldsymbol{x}\,\mathrm{dV} \Rightarrow \int_{V_P} (\boldsymbol{x} - \boldsymbol{x_P})\,\mathrm{dV} = \boldsymbol{0}$$

obtaining:

$$\int_V \operatorname{grad}\phi \,\mathrm{dV} = (\operatorname{grad}\phi)_P V_P \tag{6.3}$$

The volume V_P is evaluated by means of the `setMesh` method, acting on the Mesh object, as described in Sect. 5.3.4, and the results is stored in the `Mesh%vol` attribute. The term $(\operatorname{grad}\phi)_P$ can be evaluated by means of different techniques; in NEMO a linear least–squares approximation (e.g. see Chapra and Canale, 2002) is implemented. This method consists of computing the coefficients of the linear expression:

$$\phi = a_0 + a_x x + a_y y + a_z z \tag{6.4}$$

that should interpolate the values ϕ_P and ϕ_{nb}, which have been estimated in the last iteration on P and its neighbors nb. Once the linear regression has been solved, using a direct method for the associated linear system, such as Gaussian elimination with LU factorization, one can evaluate the term $(\operatorname{grad}\phi)_P$ as:

$$(\operatorname{grad}\phi)_P = \left(\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right) = (a_x, a_y, a_z) \tag{6.5}$$

This method for evaluating the spatial derivatives of a field has been demonstrated to be more accurate and stable than others, based on Gauss' theorem (see Ferziger and Perić, 2002). However, denoting n as the number of cells, in order to compute the gradient over the whole mesh it is necessary to solve n linear systems with a 4×4 matrix A. It is worth noting that A contains only metric data, therefore, if the geometry and the topology do not change, as for instance with static grids, it is better to perform just once the LU factorization and store the results. Another important detail is that one has to solve a large number of small linear systems, therefore packages like LAPACK, specifically designed for solving large dense algebra problems are not the best choice.

If one assumes a *linear* variation of a generic property ϕ over the CV

$$\phi = \phi_P + \boldsymbol{b}\cdot(\boldsymbol{x} - \boldsymbol{x_P})$$

then one obtains:

$$\int_{V_P} \phi \,\mathrm{dV} = \int_{V_P} [\phi_P + \boldsymbol{b}\cdot(\boldsymbol{x} - \boldsymbol{x_P})] \,\mathrm{dV} = \int_{V_P} \phi_P \,\mathrm{dV} + \boldsymbol{b}\cdot \int_{V_P} (\boldsymbol{x} - \boldsymbol{x_P}) \,\mathrm{dV}$$

By substituting in the last equation the definition of centroid, one eventually gets:

$$\int_{V_P} \phi \,\mathrm{dV} = \phi_P V_P + \boldsymbol{b}\cdot\boldsymbol{0} = \phi_P V_P$$

The previous considerations refers to a scalar field. The extension to a vector field is straightforward, because it just consists of the same procedure, applied to each cartesian component in the x, y and z directions.

6.3.2 Divergence

The finite volume method is particularly well suited for the discretization of the PDE terms containing the divergence of a vector field. By means of the Gauss' theorem it is possible to convert a volume integral into a surface one, being:

$$\int_V \operatorname{div} \boldsymbol{v} \, \mathrm{dV} = \int_A \boldsymbol{v} \cdot \boldsymbol{n} \, \mathrm{dA} \tag{6.6}$$

One can then convert the surface integral to a finite sum:

$$\operatorname{div} \boldsymbol{v} \Rightarrow \sum_{f=1}^{nf} \boldsymbol{v}_f \cdot \boldsymbol{A}_f \tag{6.7}$$

where the upper bound nf is equal to the number of faces of the CV. $\boldsymbol{A}_f$ is stored in the Vector object `af`, an attribute of the Mesh class, as depicted in Fig. 5.16, while $\boldsymbol{v}_f$ can be evaluated by linearly interpolating the values at the centers of master and slave cells.

By observing that this computation requires the field values at the face–centers, it can be understood why NEMO has been designed to support both cell–centered and face–centered fields, as mentioned in Sect. 6.2.1.

6.4 Operators Acting on PDEs

For the sake of simplicity, in this section only *scalar* fields will be considered. As depicted in Fig. 6.3, the `ToolsPde` interface class groups corresponding scalar and vector operators under the same name and provides automatic switching to the right procedure depending on the actual data type, either `ScalarField` or `VectorField`.

6.4.1 Time Derivative

The simplest differential operator which can be encountered in a PDE for CCM applications is the time derivative $\partial/\partial t$; when applied to a scalar field

ϕ, using a finite volume discretization, one gets:

$$\frac{\partial \phi}{\partial t} \Rightarrow \int_V \frac{\partial \phi}{\partial t} \, \mathrm{dV} \tag{6.8}$$

$$\int_V \frac{\partial \phi}{\partial t} \, \mathrm{dV} \approx \left(\frac{\partial \phi}{\partial t} \right)_V^n V_P \tag{6.9}$$

$$= \frac{\phi_P^n - \phi_P^{n-1}}{\Delta t} V_P \tag{6.10}$$

This leads to the source term:

$$S_t = S_{ct} + S_{Pt} \phi_P^n \tag{6.11}$$

where

$$S_{ct} = \frac{V_P}{\Delta t} \phi_P^{n-1} \tag{6.12}$$

$$S_{Pt} = -\frac{V_P}{\Delta t} \tag{6.13}$$

S_t must be added to the RHS of the related discretized PDE by writing:

$$a_P \phi_P^n = \sum_{nb} a_{nb}^n \phi_{nb} + S_t \Rightarrow A\phi^n = b \tag{6.14}$$

In summary the application of the differential operator $\partial / \partial t$ involves the addition of:

1. $-S_{pt}$ to the corresponding main diagonal element of the matrix A;
2. S_{ct} to the corresponding element of the RHS.

Figure 6.9 shows the Fortran 95 interface of the subroutine `scalarPdeDdt` which implements the operator. In particular, the procedure, included in the framework `ScalarPdeOps` (see Fig. 6.3), is invoked by the main unit as `pdeDdt` thanks to the ToolsPde interface.

As a first input parameter, the user must supply the string `sign`, that specifies explicitly the algebraic sign of the operator, as it appears in the PDE; legal values are +, − and =. The last one tells the pde–solver that the PDE building process has "entered into the RHS". Next arguments are the Field object `fld`, the timestep `dt` and the Pde object `eqn`.

The procedure performs first a dimensional check, according to the Dimensions objects included by aggregation in the Field and Pde instances, then computes for each cell the corresponding source term S_t. Lastly, it invokes the method `insPde`, in order to execute the PSBLAS insertion of S_{ct} and $-S_{pt}$ coefficients into the proper location of the sparse matrix A and RHS b, which are encapsulated in the Pde object, as depicted in Fig. 6.3.

```
module PdeOperators

  interface pdeDdt
     subroutine scalarPdeDdt(sign,fld,dt,eqn)
       use class_field
       use class_pde
       character(len=1), intent(in) :: sign
       type(ScalarField), intent(in) :: fld
       real(kind(1.d0), intent(in) :: dt
       type(ScalarPde), intent(inout) :: eqn
     end subroutine scalarPdeDdt

     ! ... vectorPdeDdt operator ...

  end interface

end module PdeOperators
```

Figure 6.9 Interface of the `scalarPdeDdt` operator.

6.4.2 Laplacian

The Laplacian operator usually models the diffusive terms in a PDE; for instance, in the transport equation of a generic scalar one may have the term: (1.11):

$$\operatorname{div}(\Gamma \operatorname{grad} \phi) \tag{6.15}$$

In case of constant diffusivity Γ, the operator can be written in the form:

$$\nabla^2 = \Delta = \sum_i \frac{\partial^2}{\partial x_i^2} \tag{6.16}$$

However, in CCM applications it is common to have a spatially non–uniform Γ coefficient, like the viscosity in the Navier–Stokes Equations coupled to the $k-\varepsilon$ turbulence model. Hence, for the sake of generality, the Laplacian must be implemented as div(Γ grad); the specialization to the case of constant diffusivity is then straightforward.

As explained by Ferziger and Perić (2002), in a finite volume approach the application of the Laplacian to a field ϕ and the successive integration

over a generic CV leads to the following expression:

$$\int_{CV} \mathrm{div}\,(\Gamma\,\mathrm{grad}\,\phi)\,\mathrm{dV} \tag{6.17}$$

By means of the Gauss' theorem the volume integral is converted into a surface integral:

$$\int_{A} \Gamma\,\mathrm{grad}\,\phi \cdot \boldsymbol{n}\,\mathrm{dA} \tag{6.18}$$

Switching from integrals to discrete sums one gets:

$$\sum_{f=1}^{nf} \Gamma_f A_f (\mathrm{grad}\,\phi)_f \cdot \boldsymbol{n}_f = \sum_{f=1}^{nf} \Gamma_f A_f \left(\frac{\partial \phi}{\partial n}\right)_f \tag{6.19}$$

The derivative of ϕ with respect to the n direction can be computed using the *deferred correction approach*, widely discussed by Ferziger and Perić (2002). This consist in approximating this term as the sum of an *implicit* and an *explicit* contribution, the latter being function of values from the last solved iteration.

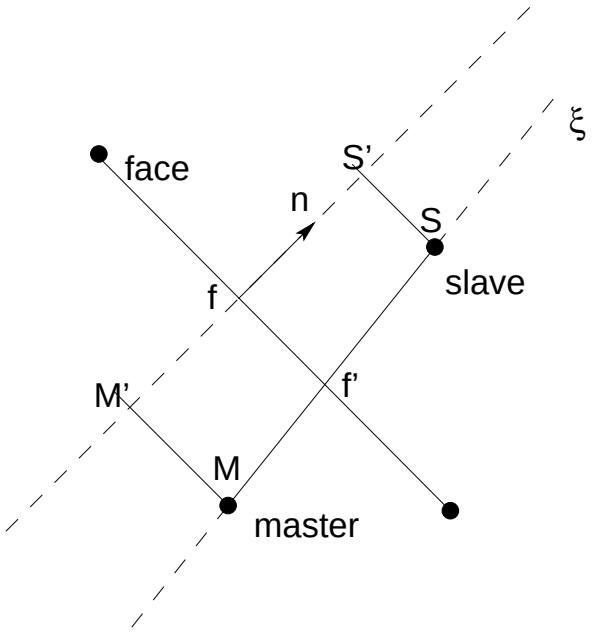


Figure 6.10 Non–orthogonality correction for the Laplacian operator.

In particular, according to the notation of the sketch in Fig. 6.10, one

may write:

$$\left(\frac{\partial\phi}{\partial n}\right)_f^{\text{new}} = \left(\frac{\partial\phi}{\partial\xi}\right)_{f'}^{\text{new}} + \left[\left(\frac{\partial\phi}{\partial n}\right)_f - \left(\frac{\partial\phi}{\partial\xi}\right)_{f'}\right]^{\text{old}} \tag{6.20}$$

where:

- f is the face center;
- M and S are the cell centers of the master and slave cells;
- ξ is the direction of the straight line passing through M and S;
- f' is the intersection between ξ and the face;
- n is the direction of the face normal, applied at the face center f;
- M' and S' are the projections of M and S on the n direction.

One can approximate the three derivatives at the RHS in the following way:

$$\left(\frac{\partial\phi}{\partial\xi}\right)_{f'}^{\text{new}} = \frac{\phi_S^n - \phi_M^n}{|\boldsymbol{x}_S - \boldsymbol{x}_M|} \tag{6.21}$$

$$\left(\frac{\partial\phi}{\partial n}\right)_f^{\text{old}} = \frac{\phi_{S'}^{n-1} - \phi_{M'}^{n-1}}{|\boldsymbol{x}_{S'} - \boldsymbol{x}_{M'}|} \tag{6.22}$$

$$\left(\frac{\partial\phi}{\partial\xi}\right)_{f'}^{\text{old}} = \frac{\phi_S^{n-1} - \phi_M^{n-1}}{|\boldsymbol{x}_S - \boldsymbol{x}_M|} \tag{6.23}$$

where

$$\phi_{M'} = \phi_M + (\nabla\phi)_M \cdot (\boldsymbol{x}_{M'} - \boldsymbol{x}_M) \tag{6.24}$$

$$\phi_{S'} = \phi_S + (\nabla\phi)_S \cdot (\boldsymbol{x}_{S'} - \boldsymbol{x}_S) \tag{6.25}$$

This non–orthogonality correction has been demonstrated to guarantee a second–order accuracy in the evaluation of the diffusive term. Its only drawback is that the lag between implicit and explicit part requires a loop of outer iterations in order to numerically converge.

In terms of implementation, the application of the laplacian operator consists of invoking the external procedure `scalarPdeLap`. It is possible to exploit the generic interface `pdeLap`, reported in Fig. 6.4.2, which provides static polymorphism for scalar and vector Field objects.

The procedure receives as an input argument the Field object `gamma`, representing the spatial distribution of the diffusion coefficient Γ, the Field

```
module PdeOperators

  interface pdeLap
     subroutine scalarPdeLap(sign,gamma,fld,eqn)
       use class_source
       use class_pde
       character(len=1), intent(in) :: sign
       type(ScalarField), intent(in) :: gamma
       type(ScalarField), intent(in) :: fld
       type(ScalarPde), intent(inout) :: eqn
     end subroutine scalarPdeLap

     ! ... vectorPdeLap operator ...

  end interface

end module PdeOperators
```

Figure 6.11 Interface of the `scalarPdeLap` operator.

object `fld`, which stores the scalar variable ϕ, and lastly the `Pde` object `eqn`. After a preliminary dimensional check, the code executes a loop on the local faces, applying to each one the deferred correction approach and evaluating the implicit and explicit contributions; they must be respectively added to the sparse matrix A and the RHS b encapsulated in the `Pde` object, as depicted in Fig. 6.3. This task is performed by calling the `spInsPde` and `geInsPde` methods, which wrap the PSBLAS routines `psb_spins` and `psb_geins` related to the insertion either of a block of rows into a sparse matrix, or of single elements into a dense vector.

6.4.3 Source Application

In the conservation equation for a generic scalar field ϕ, a source is often represented as a linear term:

$$s_\phi = s_c + s_P \phi \tag{6.26}$$

In a finite volume discretization one can assume that s_c and s_P are constant over a CV, thus obtaining the following source term in the discretized

equation:

$$S_\phi = S_c + S_P \phi_P \tag{6.27}$$

with

$$S_c = s_c V_P \tag{6.28}$$

$$S_p = s_P V_P \tag{6.29}$$

S_ϕ must be added to the RHS of the related discretized PDE by writing:

$$a_P \phi_P^n = \sum_{nb} a_{nb} \phi_{nb}^n + S_\phi \Rightarrow A\phi^n = b \tag{6.30}$$

In summary the application of the source term $s_\phi = s_c + s_P \phi$ involves the addition of:

1. $-S_P$ to the corresponding main diagonal element of the matrix A;
2. S_c to the corresponding element of the RHS.

```
module PdeOperators

  interface pdeSrc
     subroutine scalarPdeSrc(sign,src,eqn)
       use class_source
       use class_pde
       character(len=1), intent(in) :: sign
       type(ScalarSource), intent(in) :: src
       type(ScalarPde), intent(inout) :: eqn
     end subroutine scalarPdeSrc

     ! ... vectorPdeSrc operator ...

  end interface

end module PdeOperators
```

Figure 6.12 Interface of the `scalarPdeSrc` operator.

Figure 6.12 shows the Fortran 95 interface of the subroutine `scalarPdeSrc` which applies the Source object `src` to the Pde instance `eqn`. The procedure

receives in input `src`, checks its dimensions against the ones of `eqn`, extracts by means of getter functions the members `sc` and `sp` encapsulated in it and, lastly, invokes the `insPde` method, in order to perform the PSBLAS insertion of S_c and S_P terms.

6.5 Example: THERMO Main Program

The object–oriented framework of NEMO permits solving CCM problems by mimicking the writing of the governing PDEs. For example, one could study the temperature distribution in a complex–shaped solid body, under the effects of prescribed boundary conditions and of a uniformly–distributed heat source, assuming a temperature–dependent variation for the thermal conductivity. The governing equation for such phenomenon is the well known Fourier's equation, already discussed in Sect. 1.1.2, here written again as:

$$\frac{\partial \rho c T}{\partial t} = \operatorname{div} k \operatorname{grad} T + S_T \tag{6.31}$$

The corresponding source of the NEMO–based code which solves this problem has been reported in Figs. 6.13, 6.14, 6.15 and 6.16.

Figure 6.13 refers to the variable declaration; one should note the block of Fortran 95 modules loaded at the beginning of the program which provides access to the respective classes.

Figure 6.14 refers to the setup phase, consisting of the following steps:

- initialization of the BLACS parallel environment;
- importing of the mesh;
- reordering, partitioning, global–to–local reallocation and evaluation of geometric quantitities;
- initialization of material section;
- reading and setup of boundary conditions;
- allocation of Field, Pde, Source, Counter objects which will be used in the PDE–solver.

Figure 6.15 reports the kernel of the code, i.e. the application of the differential operators on Field and Pde object, by reproducing signs and term groups of the orginal governing equation 6.31. The PDE building and solving phases are inserted in an outer loop which performs the time advancing

```
program thermo

  use class_blacs
  use class_bc
  use class_counter
  use class_dimensions
  use class_field
  use class_material
  use class_mesh
  use class_pde
  use class_source
  use constants
  use tools_operators

  implicit none
  !
  type(Mesh) :: msh
  type(Material) :: mat
  type(BCpoly), pointer :: bc(:) => null()
  type(Counter) :: runTime
  type(Source) :: src
  type(ScalarField) :: T, k, rho, c, rho_c
  type(ScalarPde) :: eqnT
  !
  character(len=30) :: inputFile = 'thermo.inp'
  real(kind(1.d0)) :: dt
```

Figure 6.13 Source code of the Thermo program: variable declaration

```
call startBlacs

call setMesh(msh,inputFile)
call setMaterial(mat,inputFile)
call setBC(bc,inputFile,msh)

call setField(T,bc,msh,mat,temperature)
call setField(k,bc,msh,mat,conductivity,onFaces=.true.)

call setField(rho,bc,msh,mat,density)
call setField(c,bc,msh,mat,specificHeat)
call fldMul(rho,c,rho_c)

call setPDE(eqnT,inputFile,'PDE energy',msh,power)

call setCounter(runTime,inputFile)
call setSource(src,inputFile,'SOURCE',power/volume)
```

Figure 6.14 Source code of the Thermo program: variable setup

until the maximum number of timesteps has been reached: this is monitored by the `runTime` Counter object. The statement `call updField(k,T)` allows updating the thermal conductivity as a function of the temperature, according to the desidered dependency and the material properties stored in a database.

Lastly the memory used by all objects is deallocated as depicted in Fig. 6.16. As a final operation the Blacs parallel environment is closed, causing the normal termination of the program.

```
  dt = dt_(runTime)

  timeLoop: do
     call updCounter(runTime)

     call pdeDdt('+',rho_c,T,dt,eqnT)
     call pdeLap('=',k,T,eqnT)
     call pdeSrc('+',src,eqnT)

     call solvePde(eqnT,T)

     call updField(k,T)

     if(end_(runTime)) exit timeLoop
  end do timeLoop
```

Figure 6.15 Source code of the Thermo program: pde solving

```
  call dealloc(eqnT)
  call dealloc(k)
  call dealloc(rho)
  call dealloc(c)
  call dealloc(rho_c)
  call dealloc(T)

  call dealloc(bc)
  call dealloc(mat)
  call dealloc(msh)

  call stopBlacs

end program thermo
```

Figure 6.16 Source code of the Thermo program: memory deallocation

Chapter 7

Conclusions

7.1 Current Status

Presently (April 2006) the basic kernel of the code has been completed implementing several classes, shared by different CCM applications, such as:

- Mesh, Cell, Face, Vector and Connectivity: geometry and topology;
- Pde, Field, Source, Bc: PDE modeling;
- Material: constitutive relationships;
- Blacs: parallel environment and message–passing.

The following differential operators have been successfully implemented and tested:

$$\frac{\partial}{\partial t}, \quad \text{grad}, \quad \text{div(grad)}$$

Behind the user interface, a finite–volume method with central difference discretization has been adopted that shows second order accuracy, even with a non–orthogonal mesh, thanks to a special correction implemented in the Laplacian operator.

The combination between diffusion operators and central difference scheme makes the current code suitable for solving *elliptic* problems such as stationary conduction, or the irrotational flow of an incompressible inviscid fluid, governed respectively by Poisson's and Laplace's equation. The availability of the time derivative permits also the analysis of some *parabolic* problems, such as the unsteady thermal diffusion, which requires, with respect to the steady case, just one additional, high level, call to the procedure which encapsulates the finite volume approximation of $\partial/\partial t$.

In summary, every diffusion–like phenomena, involving the Laplacian operator, can be simulated by re–using the available classes, without requiring any additional effort from the developer.

It is worth noting that the PDE solver can run in parallel, by using the powerful preconditioners and iterative methods available in the PSBLAS library. Preliminary tests on Linux clusters, with high speed network (e.g. Infiniband), have showed encouraging scalability results.

The choice of Fortran 95 as programming language for implementing object–oriented features has been fully satisfactory. Those features, such as *is–a* inheritance and run–time polymorphism, which are not yet implemented in the standard Fortran 95, can be emulated. Up to now we needed just once to implement a dynamically polymorphic class; this confirms that in scientific applications run–time polymorphism is not such a fundamental feature for object–orientation.

7.2 Future Work

The first step of the future work will be to perform an extensive validation campaign for the existing set of operators. Immediately after, it will be expanded, by introducing also the divergence of a vector field, necessary for modeling convective phenomena.

After having accomplished this task, it will be possible to solve the transport equation for a generic scalar 1.11, which is fundamental for solving the momentum equation in the predictor step of many segregated CFD solvers, such as SIMPLE or PISO (see Versteeg and Malalasekera, 1995; Ferziger and Perić, 2002). It is worth noting that these algorithms are often discussed in literature using operators, instead of discretized finite–difference expressions, therefore it should be not too hard translating these relationships into calls to NEMO procedures. Following this strategy, we will try to implement a laminar, incompressible version of the PISO scheme, designed by Issa (1985).

In addition to the central–difference, the upwind scheme should be introduced as soon as possible. This is the only first–order accurate, bounded, discretization scheme, which still persists as the first option in CFD simulations when one has to avoid instabilities, without either refining the mesh, or using higher order, TVD, schemes.

With a quite stable code one should, at last, test and stress it, before starting the implementation of new features. This is doubly important for a correct development of NEMO. The traditional debugging work, should be, in fact, coupled with accurate timing sessions, in order to detect possible bottlenecks in the passage from the high–level object–oriented layer to the

low–level procedural areas. This analysis will be useful for understanding where data abstraction must be avoided in favor of efficiency and, on the other hand, where encapsulation and data hiding can improve the flexibility, the re–usability and make the maintaining process easier.

At that time, NEMO will be able to handle basic CFD simulations, heat transfer computations and even structural analysis. After having consolidated this know–how, it will be possible to start working on the refinement of the CFD section and on conjugate problems, involving the interaction of two phenomena, such as fluid dynamics and heat transfer, enhancing the actual *multiphysics* capabilities of the code.

Bibliography

Akin, E. 2003. *Object–Oriented Programming Via Fortran 90/95*, First ed. Cambridge University Press.

Amsden, A. A. 1993. KIVA–3: A KIVA Program with Block–Structured Mesh for Complex Geometries. Tech. Rep. LA–12503–MS, Los Alamos National Laboratory. March.

Amsden, A. A. 1997. KIVA–3V: A Block–Structured KIVA Program for Engines with Vertical or Canted Valves. Tech. Rep. LA–13313–MS. July.

Amsden, A. A., Rourke, P. J. O., and Butler, T. D. 1989. KIVA-II: A Computer Program for Chemically Reactive Flows with Sprays. Tech. Rep. LA–11560–MS, Los Alamos National Laboratory. May.

Andreassi, L., Maio, A. D., Bella, G., and Bernaschi, M. 1999. Internal combustion engines computational tools: Use and development. In *Proceedings of ICE99 Conference, Naples*. SAE Europe, 219–227.

Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. 1997. *ScaLAPACK User's Guide*. SIAM, Philadelphia, PA.

Booch, G. 1993. *Object–Oriented Analysis and Design with Applications*, Second ed. Professional Series. Addison–Wesley.

Buttari, A. 2006. Software tools for sparse linear algebra computations. Ph.D. thesis, University of Rom "Tor Vergata".

Cary, J. R., Shasharina, S. G., Cummings, J. C., Reynders, J. V. W., and Hinker, P. J. 1997. Comparison of C++ and Fortran 90 for object–oriented scientific programming. *Computer Physics Communications 105,* 1 (September), 20–36.

Chapra, S. C. and Canale, R. P. 2002. *Numerical Methods for Engineers with Software and Programming Applications*, Fourth ed. Higher Education Series. McGraw–Hill.

Dai, M. and Schmidt, D. P. 2005. Adaptive tetrahedral meshing in free–surface flow. *Journal of Computational Physics 208*, 228–252.

Decyk, V. K., Norton, C. D., and Szymanski, B. K. 1997. How to express C++ concepts in Fortran 90. *Scientific Programming 6,* 4 (Winter), 363–390. `http://www.cs.rpi.edu/~szymansk/oof90.html`.

Decyk, V. K., Norton, C. D., and Szymanski, B. K. 1998. How to support inheritance and run–time polymorphism in Fortran 90. *Computer Physics Communications 113*, 1–9.

Dongarra, J. J. and Whaley, R. C. 1995. LAPACK working note 94 - a user's guide to the blacs v1.1. URL: `www.netlib.org/blacs/`.

Dubois-Pèlerin, Y. and Zimmermann, T. 1993. Object–oriented finite element programming: III. an efficient implementation in C++. *Computer Methods in Applied Mechanics and Engineering 108*, 165–183.

Duff, I., Marrone, M., Radicati, G., and Vittoli, C. 1997. Level 3 basic linear algebra subprograms for sparse matrices: a user level interface. *ACM Transactions on Mathematical Software 23,* 3 (September), 379–401.

Ferziger, J. H. and Perić, M. 2002. *Computational Methods for Fluid Dynamics*, Third ed. Springer.

Filippone, S. and Buttari, A. 2006. *PSBLAS–2.0 User's Guide. A Reference Guide for Parallel Sparse BLAS library.*

Filippone, S. and Colajanni, M. 2000. PSBLAS: a library for parallel linear algebra computations on sparse matrices. *ACM Transactions on Mathematical Software 26,* 4, 527–550.

Filippone, S., Colajanni, M., and Pascucci, D. 1999. An object–oriented environment for sparse parallel computation on adaptive grids. In *Proceedings of the 13th International Symposium on Parallel Processing*

and the 10th Symposium on Parallel and Distributed Processing. IEEE Computer Society, Washington, DC, USA,.

Filippone, S., Marrone, M., and di Brozolo, G. R. 1992. Parallel preconditioned conjugate–gradient type algorithms for general sparsity structures. *International Journal of Computer Math. 40*, 159–167.

Fluent, Inc. 2004. *Gambit 2.2 – User's Guide*. Fluent, Inc.

Fowler, M. and Scott, K. 2003. *UML Distilled – A Brief Guide to the Standard Object Modeling language*, Third ed. Object Technology Series. Addison–Wesley.

Garey, M. R. and Johnson, D. S. 1978. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. H. Freeman, San Francisco.

Gibbs, N. E., William G. Poole, J., and Stockmeyer, P. K. 1976a. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal of Numerical Analysis 18,* 2, 235–251.

Gibbs, N. E., William G. Poole, J., and Stockmeyer, P. K. 1976b. A comparison of several bandwidth and profile reduction algorithms. *ACM Transactions on Mathematical Software 2,* 4, 322–330.

Gottlieb, D. and Orszag, S. A. 1977. *Numerical Analysis of Spectral Methods: Theory and Applications*, First ed. SIAM.

Hoffmann, K. A. and Chiang, S. T. 2000. *Computational Fluid Dynamics – Volumes I, II, III*, Fourth ed. Engineering Education System.

Issa, R. I. 1985. Solution of the Implicitly Discretized Fluid Flow Equations by Operator–Splitting. *Journal of Computational Physics 62*, 40–65.

Jasak, H. and Weller, H. G. 2000. Application of the finite volume method and unstructured meshes to linear elasticity. *International Journal for Numerical Methods in Engineering 48*, 267–287.

Jasak, H., Weller, H. G., and Nordin, N. 2004. In–cylinder cfd simulation using a C++ object–oriented toolkit. *SAE Technical Papers 2004–01–0110*.

Karypis, G. and Kumar, V. 1998. *METIS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*.

Karypis, G., Schloegel, K., and Kumar, V. 2003. *ParMETIS – Parallel Graph Partitioning and Sparse Matrix Ordering Library – Version 3.1.*

Löhner, R. 2001. *Applied CFD Techniques – An Introduction Based On Finite Element Methods*, First ed. John Wiley & Sons.

Machiels, L. and Deville, M. 1997. Fortran 90: An entry to object-oriented programming for the solution of partial differential equations. *ACM Transactions on Mathematical Software 23,* 1 (March), 32–49.

Metcalf, M., Reid, J., and Cohen, M. 2004. *Fortran 95/2003 Explained*, First ed. Oxford University Press.

Norton, C. D., Szymanski, B. K., and Decyk, V. K. 1995. Object–oriented parallel computation for plasma simulation. *Communications of the ACM 38,* 10 (October), 88–100. `http://www.cs.rpi.edu/~szymansk/papers.html`.

Patankar, S. V. 1980. *Numerical Heat Transfer and Fluid Flow*, First ed. Series in Computational Methods in Mechanics and Thermal Sciences. Taylor & Francis.

Quatrani, T. 2003. Introducction to the unified modeling language. Tech. rep., IBM – Rational Software. June.

Reid, J. 2003. The future of Fortran. *Computing in Science & Engineering 5,* 4, 59–67.

Rhie, C. M. and Chow, W. L. 1983. Numerical study of the turbulent flow past an airfoil with trailing edge separation. *AIAA Journal 21,* 11, 1525–1532.

Rumsey, C. L., Poirier, D. M. A., Bush, R. H., and Towne, C. E. 2005. Cfd general notation system: A users guide to cgns. Tech. Rep. Document Version 1.1.7, CGNS Version 2.4, CGNS.

Saad, Y. 2003. *Iterative Methods for Sparse Linear Systems*, Second ed. SIAM, Philadelphia, PA.

Stroustrup, B. 2000. *The C++ Programming Language*, Third ed. Professional Series. Addison–Wesley.

Tannehill, J. C., Anderson, D. A., and Pletcher, R. H. 1997. *Computational Fluid Mechanics And Heat Transfer*, Second ed. Series in Computational and Physical Processes in Mechanics and Thermal Sciences. Taylor and Francis.

Versteeg, H. K. and Malalasekera, W. 1995. *An Introduction to Computational Fluid Dynamics – The Finite Volume Method*, First ed. Prentice Hall.

Weller, H. G., G.Tabor, Jasak, H., and Fureby, C. 1998. A tensorial approach to computational continuum mechanics using object–oriented techniques. *Computers in Physics 12,* 6.

Zienkiewicz, O. C., Taylor, R. L., and Zhu, J. Z. 2005. *The Finite Element Method: Its Basis and Fundamentals*, Sixth ed. Butterworth–Heinemann.