Arch Computat Methods Eng (2018) 25:195–271
https://doi.org/10.1007/s11831-017-9244-1

ORIGINAL PAPER

FEMPAR: An Object-Oriented Parallel Finite Element Framework

Santiago Badia[1,2] · Alberto F. Martín[1,2] · Javier Principe[2,3]

Received: 5 August 2017 / Accepted: 15 September 2017 / Published online: 11 October 2017

Abstract FEMPAR is an open source object oriented Fortran200X scientific software library for the high-performance scalable simulation of complex multiphysics problems governed by partial differential equations at large scales, by exploiting state-of-the-art supercomputing resources. It is a highly modularized, flexible, and extensible library, that provides a set of modules that can be combined to carry out the different steps of the simulation pipeline. FEMPAR includes a rich set of algorithms for the discretization step, namely (arbitrary-order) grad, div, and curl-conforming finite element methods, discontinuous Galerkin methods, B-splines, and unfitted finite element techniques on cut cells, combined with *h*-adaptivity. The linear solver module relies on state-of-the-art bulk-asynchronous implementations of multilevel domain decomposition solvers for the different discretization alternatives and block-preconditioning techniques for multiphysics problems. FEMPAR is a framework that provides users with out-of-the-box state-of-the-art discretization techniques and highly scalable solvers for the simulation of complex applications, hiding the dramatic complexity of the underlying algorithms. But it is also a framework for researchers that want to experience with new algorithms and solvers, by providing a highly extensible framework. In this work, the first one in a series of articles about FEMPAR, we provide a detailed introduction to the software abstractions used in the discretization module and the related geometrical module. We also provide some ingredients about the assembly of linear systems arising from finite element discretizations, but the software design of complex scalable multilevel solvers is postponed to a subsequent work.

✉ Santiago Badia
sbadia@cimne.upc.edu

Alberto F. Martín
amartin@cimne.upc.edu

Javier Principe
principe@cimne.upc.edu

1 Department of Civil and Environmental Engineering, Universitat Politècnica de Catalunya, Jordi Girona 1-3, Edifici C1, 08034 Barcelona, Spain

2 CIMNE Centre Internacional de Mètodes Numèrics en Enginyeria, Parc Mediterrani de la Tecnologia, UPC, Esteve Terradas 5, 08860 Castelldefels, Spain

3 Department of Fluid Mechanics, Universitat Politècnica de Catalunya, Eduard Maristany, 10-14, 08019 Barcelona, Spain

1 Introduction

Even though the origins of the FE method trace back to the 50s, the field has drastically evolved during the last six decades, leading to increasingly complex algorithms to improve accuracy, stability, and performance. The use of the *p*-version of the FE method and its exponential convergence makes high-order approximations an excellent option in many applications [1]. Adaptive mesh refinement driven by *a posteriori* error estimates, i.e., *h*-adaptivity, is an essential ingredient to reduce computational cost in an automatic way [2]. For smooth solutions, *p*-adaptivity or hybrid *hp*-adaptivity can further reduce computational cost for a target level of accuracy [3]. Originally, FE methods were restricted to nodal Lagrangian bases for structural problems. The extension of FE methods to other applications, like porous media flow or electromagnetism, motivated the design of more complex bases and require different mappings from the reference to the physical space, complicating the implementation of these techniques in standard FE codes. Saddle-point problems also require particular mixed FE discretizations for stability purposes [4, 5]. More recently, novel FE formulations have been proposed within the frame of exterior calculus, e.g., for mixed linear elasticity problems [6]. Physics-compatible discretization are also gaining attention, e.g., in the field of incompressible fluid mechanics. Divergence-free mixed FEs satisfy mass conservation up to machine precision, but their implementation is certainly challenging [7]. During the last decade, a huge part of the computational mechanics community has embraced isogeometric analysis techniques [8], in which the discretization spaces are defined in terms of NURBS (or simply splines), leading to smoother global spaces. In the opposite direction, discontinuous galerkin (DG) methods have also been actively developed, and novel approaches, like hybridizable DG and Petrov-Galerkin DG methods, have been proposed [9, 10]. As the discretization methods become more and more complex, the efficient implementation of these techniques is more complicated. It also poses a challenge in the design of scientific software libraries, which should be extensible and provide a framework for the (easy) implementation of novel techniques, to be resilient to new algorithmic trends.

The hardware in which scientific codes run evolves even faster. During 40 years, core performance has been steadily increasing, as predicted by Moore's law. In some years, supercomputers will reach 1 exaflop/s, a dramatic improvement in computational power that will not only affect the extreme scale machines but radically transform the whole range of platforms, from desktops to high performance computing (HPC) clouds. The ability to efficiently exploit the forthcoming 100x boost of computational performance will have a tremendous impact on scientific discoveries/economic benefits based on computational science, reaching almost every field of research. However, all the foreseen exascale growth in computational power will be delivered by increasing hardware parallelism (in distinct forms), and the efficient exploitation of these resources will not be a simple task. HPC architectures will combine general-purpose fat cores, fine-grain many-cores accelerators (GPUs, DSPs, FPGAs, Intel MIC, etc.), and multiple-level disruptive-technology memories, with high non-uniformity as common denominator [11]. This (inevitable) trend challenges algorithm/software design. Traditional bulk-synchronous message passing interface (MPI) approaches are likely to face significant performance obstacles. Significant progress is already being made by MPI+X [12] (with X=OpenMP, CUDA, OpenCL, OmpSs, Kokkos, etc.) hybrid execution models. Going a step further, asynchronous many-task execution models (e.g., Charm++[13], Legion [14], or HPX [15]) and their supporting run-time systems hold great promise [16].

Traditionally, researchers in the field of scientific computing used to develop codes with a very reduced number of developers, e.g., a university department, and a limited life span. The software engineering behind scientific codes was poor. Codes were rigid and non-extensible, and developed for a target application and a specific numerical method. However, the increasing levels of complexity both in terms of algorithms and hardware make the development of scientific software that can efficiently run state-of-the-art numerical algorithms on HPC resources a real challenge. Considering to start from scratch a project of this kind has an ever increasing level of complexity. Furthermore, due to the huge resources required to carry out such a project, it is natural to develop a framework that will be resilient to new algorithmic and hardware trends, in order to maximize life time, and to be applicable to a broad range of applications. In this sense, object-oriented (OO) programming, which provides modularity of codes and data-hiding, is the key for the software design of flexible and scalable (in terms of developers) projects.

There is a number of open source OO FE libraries available through the Internet, e.g., deal.II [17, 18], FEniCS [19], GRINS [20], Nektar++ [21], MOOSE [22], MFEM [23], FreeFem++ [24], and DUNE [25]. In general, these libraries aim to provide all the machinery required to simulate complex problems governed by partial differential equations (PDE) using FE techniques. In any case, every library has its main goal and distinctive features. Some libraries, like FreeFem++ or FEniCS, have extremely simple user interfaces. FEniCS has its own domain specific language for weak forms to automatically generate the corresponding FE code (preventing p-adaptivity) and includes a collection of Python wrappers to provide user-friendly access to the services of the library. Other sophisticated libraries like deal.II or DUNE have a slightly more demanding learning curve. In general, parallel adaptivity is at most partially supported; as far as we know, none of the libraries above have support for parallel hp-adaptivity, unless DG methods are being used. Some libraries are restricted to a particular cell topology, e.g., deal.II is limited to hexahedral/quadrilateral (n-cubes) meshes, while FEniCS only supports simulations on triangular/tetrahedral (n-simplices) meshes.

In general, these libraries provide modules for some of the different steps in the simulation pipeline, which involves the set-up of the mesh, the construction of the FE space, the integration and assembly of the weak form, the solution of the resulting linear system, and the visualization of the computed solution. The solution of the linear system is clearly segregated from the discretization step in all the scientific software libraries described above (for parallel computations); the linear system is transferred to a general-purpose sparse linear algebra library, mainly PETSc [26–28], Hypre [29], and Trilinos [30, 31]. As a result, the coupling between the discretization step and the linear solver step is somehow weak, since they rely on general purpose solvers, which usually involve simple interfaces. The strong point of these general purpose numerical linear algebra libraries is to be problem-independent, but it also limits their performance for specific applications, since they cannot fully exploit the underlying properties of the PDE operator and the numerical discretization.[1] This segregation has a clear impact on the type of methods to be used. This black-box approach to general-purpose linear solvers has favoured the use of algebraic multigrid methods, the *de facto* linear solver [29]. On the other hand, geometric multigrid methods and domain decomposition (DD) methods, which are very specific to mesh-based PDE solvers, are not common, even though they can be superior to algebraic methods in many cases. A geometric multigrid method that exploits the *hp*-adaptive structure of the FE space is included in deal.II, but it can only be used in the serial case. In parallel scenarios, DD methods have certainly evolved during the last decade. Modern DD methods do not (necessarily) rely on a static condensation of the internal variables, which requires sparse direct methods for the local subdomain problems. Instead, *inexact* solvers can be used, e.g., multigrid methods, and linear complexity DD preconditioners can be defined (see [33, 34]). The definition of two-level DD methods resembles the one of FE methods, by exchanging the FE and subdomain concepts, and their definition is strongly related to the one of multiscale FEs [35]. Furthermore, multilevel extensions can be naturally defined. In short, state-of-the-art multilevel DD methods can be understood (in their *inexact* version) as a non-conforming multigrid method. Even though the mathematical theory of the DD methods is very sound, high performance implementations are quite recent (see [36–38]). On the other hand, we are not aware of any general purpose FE code that integrates a DD algorithm in the solution workflow. DD methods require sub-assembled matrices to be used, and are not supported by the majority of the existing advanced OO FE libraries. Analogously, the use of block-preconditioning is in general poorly supported, because it involves the discretization of additional operators to define the approximated Schur complement, and the corresponding block-based assembly of matrices.

On the other hand, based on the supercomputing trends, the segregation between time discretization, linearization, space discretization, and linear system solve, will progressively blur. As an example, nonlinear preconditioning and parallel-in-time solvers are two natural ways to attain the higher levels of concurrency of the forthcoming exascale supercomputers [36, 39]. These facts will complicate even more the rigid workflow of current advanced FE libraries. In this sense, current efforts in PETSc to provide nonlinear preconditioning interfaces can be found in [40], relying on call-back functions, and the XBraid solver [41] aims to provide time-parallelism in a non-intrusive way.

2 The FEMPAR Project

In this work, we present FEMPAR, an OO FE framework for the solution of PDEs, designed from inception to be highly scalable on supercomputers and to easily handle complex multiphysics problems. The first public release of FEMPAR has almost 300K lines of code written in (mostly) OO Fortran and makes intensive use of the features defined in the 2003 and 2008 standards of the language. The source code that is complementary to this work corresponds to the first public release of FEMPAR, i.e., version 1.0.0. It is available at a git repository [42]. In particular, the first public release was assigned the git tag FEMPAR-1.0.0, in accordance with the "Semantic Versioning" system.[2]

FEMPAR is very rich in terms of FE technology. In particular, it includes not only Lagrangian FEs, but also curl- and div-conforming ones, e.g., Nédélec (edge) and Raviart-Thomas FEs. The library supports n-cube and n-simplex meshes, and arbitrary high-order bases for all the FEs included. Continuous and discontinuous spaces can be used, providing all the machinery for the integration of DG facet (i.e., edges in 2D and faces in 3D) terms. Recently, in a beta version of the code, B-splines have also been added, together with the support for cut cell methods (using XFEM-type techniques) and *hp*-adaptivity, but we will not discuss these developments for the sake of brevity.

Moreover, FEMPAR has been developed with the aim to provide a framework that will allow developers to implement complex techniques that are not well-suited in the traditional segregated workflow commented above. FEMPAR also provides a highly scalable built-in numerical linear algebra module based on state-of-the-art domain decomposition solvers. FEMPAR can provide partially assembled matrices, required for DD solvers; the multilevel BDDC solver in FEMPAR has scaled up to almost half a million cores and 1.75 million MPI tasks (subdomains) in the JUQUEEN Supercomputer [34, 37]. It includes an abstract framework to construct applications and preconditioners based on multilevel nonoverlapping partitions.

[1] A paradigmatic example is the design of scalable solvers for the discretization of the Maxwell equations using edge elements, which involve the discretization of additional operators (discrete gradients) and changes of basis at the reference FE level [32].

[2] Available at http://semver.org/.

Even though every block within the library preserves modularity, the interface between discretization and numerical linear algebra modules within FEMPAR is very rich and focused on PDE-based linear systems. In the path to the exascale, FEMPAR has been designed to permit an asynchronous implementation of multilevel methods, both in terms of multiphysics FEs and multilevel solvers, which have been exploited, e.g., in [37]. It is a unique feature that is not available in other similar libraries. The library also allows the user to define blocks in multiphysics applications, that can be used to easily implement complex block preconditioners [43–45]. All these blocks are very customizable, which has already been used to develop scalable DD solvers for electromagnetics problems and block preconditioners for multiphysics problems, e.g., magnetohydrodynamics [44]. These distinctive features of FEMPAR, however, are not discussed in this article but in a forthcoming one. A general discussion of the main ingredients of our implementation of the discretization step using FE-like approximations is first necessary, which is the purpose of this work.

FEMPAR has already been successfully used in a wide set of applications by the authors of the library: simulation of turbulent flows and stabilized FE methods [46–49], magnetohydrodynamics [50–54], monotonic FEs [55–59], unfitted FEs and embedded boundary methods [60], and additive manufacturing simulations [61]. It has also been used for the highly efficient implementation of DD solvers [34, 37, 39, 62–66] and block preconditioning techniques [44].

This work is more than an overview article with the main features of the library. It is a detailed description of the software abstractions being used within FEMPAR to develop an efficient, modular, and extensible implementation of FE methods and supporting modules in a broad sense. To this end, we enrich the discussion with code snippets that describe data structures, bindings, and examples of use.[3] This document is intended to be used as a guide for new FEMPAR *developers* that want to get familiarized with its software abstractions. But it can also be a useful tool for developers of FE codes that want to learn how to implement FE methods in an advanced OO framework. In any case, due to the size of the library itself, many details cannot be exposed, to keep a reasonable article length. The article can be read in different ways, since it is not necessary to fully understand all the preceding sections to grasp the main ideas of a section. For instance, the section about the abstract implementation of polytopes in arbitrary dimensions and its related algorithms is quite technical and a reader that is not particularly interested in the internal design of this type and its bindings implementations can skip it. Experienced FE researchers can skip the short section with the basics of FE methods, and only look at this one (if needed) when referred in subsequent sections.

The article is organized as follows. In Sect. 3 we present a concise mathematical description of the FE framework. The main mathematical abstractions are expressed in software by means of a set of derived data types and their associated TBPs, which are described in subsequent sections. In particular, the main software abstractions in FEMPAR and their roles in the solution of the problem are:

- The polytope, which describes a set of admissible geometries and permits the automatic, dimension-independent generation of reference cells and structured domains. The mathematics underlying the polytope are presented in Sect. 3.14, while its software implementation in Sect. 4.
- The polynomial abstraction and related data types, which are presented in Sects. 3.4 and 5, respectively. These sections describe how shape functions bases can be generated for arbitrary orders and for n-cube and n-simplex topologies.
- The reference FE in Sect. 6, which describes the reference cell and defines a set of basis functions and degrees of freedom on each cell.
- The triangulation in Sect. 7, which represents a discrete approximation of the physical domain Ω.
- A set of tools required to perform numerical integration (e.g., quadratures and geometrical maps) produced by the reference FE and described in Sect. 8 for cell integrals and in Sect. 9 for facet integrals.
- The FE space described in Sect. 10, built from a triangulation and a set of reference FEs, which represents a global space of FE functions.
- The discrete integration, an abstract class to be extended by the user to define an affine FE operator, which describes the numerical integration of the weak form of the problem to be solved, described in Sect. 11.2.
- The linear (affine) operator in Sect. 11, whose root is the solution of the problem at hand, constructed using the FE space and a discrete integration.

[3] The code snippets are written in advanced OO Fortran 200X [67]. There is a close relationship between these language features and those available in the C++ language [68] and we established some code style rules to emphasize it. In particular, Fortran modules in FEMPAR are always named with the suffix _names, to indicate the analogy with namespaces in C++. Derived types, analog to C structs or C++ classes, are always named with _t to distinguish them from instances. However it should be kept in mind that, whereas structs in C++ are passive data containers and classes are used to carry also methods, Fortran derived data types are used in both cases since the introduction in the 2003 standard of the so called type-bound procedures (TBPs).

- An example of a user driver in Sect. 12, in which the different ingredients previously described are used to simulate a problem governed by PDEs, the Stokes system.

A (very simplified) graphical overview of the main software abstractions in FEMPAR and some of their relationships is shown in Fig. 1.

3 The FE Framework

In this section, we briefly introduce all the mathematical abstractions behind the FE method for the discretization of problems governed by PDEs. For a more detailed exposition of the topics, we refer to [69–72]. The FEs described below (and many other not covered herein) can be formulated and analyzed using the *finite element exterior calculus* framework [6], which makes use of exterior algebra and exterior calculus concepts. In this framework, one can define FEs, e.g., div and curl-conforming ones, in arbitrary space dimensions, using the concept of differential k-forms. However, we have decided not to use such presentation of FE methods to simplify the exposition for readers not familiar with these abstractions.

3.1 The Boundary Value Problem in Weak Form

We are interested in problems governed by PDEs posed in a physical domain $\Omega \subset \mathbb{R}^d$ with boundary $\Gamma \doteq \partial\Omega$. In practice $d = 2, 3$ but we are also interested in $d > 3$ for some particular applications (see Sect. 3.14). Let us consider a differential operator A, e.g., the Laplace operator $-\Delta$, and a force term $f : \Omega \to \mathbb{R}$. Let us also consider a partition of Γ into a Dirichlet boundary Γ_{D} and a Neumann boundary Γ_{N}, and the corresponding boundary data $u_{\mathrm{D}} : \Gamma_{\mathrm{D}} \to \mathbb{R}$ and $g_{\mathrm{N}} : \Gamma_{\mathrm{N}} \to \mathbb{R}$. The boundary value problem reads as follows: find $u(\boldsymbol{x})$ such that

$$Au(\boldsymbol{x}) = f(\boldsymbol{x}) \quad \text{in } \Omega, \qquad B_{\mathrm{D}}u(\boldsymbol{x}) = u_{\mathrm{D}}(\boldsymbol{x}) \quad \text{on } \Gamma_{\mathrm{D}}, \qquad B_{\mathrm{N}}u(\boldsymbol{x}) = g_{\mathrm{N}}(\boldsymbol{x}) \quad \text{on } \Gamma_{\mathrm{N}}. \tag{1}$$

The operator B_{D} is a trace operator and B_{N} is the flux operator. Other boundary conditions, e.g., Robin (mixed) conditions can also be considered. We assume that the unknown $u(\boldsymbol{x})$ in (1) can be a scalar, vector, or tensor field. (The case of multi-field problems is considered in Sect. 3.11.)

For FE analysis, we must consider the weak form of (1). The weak formulation can be stated in an abstract setting as follows. Let us consider an abstract problem determined by a Banach space $\mathcal{X}$ (*trial space*), a reflexive Banach space $\mathcal{Y}$ (*test space*), a continuous bilinear form $a : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$, and a continuous linear form $\ell : \mathcal{Y} \to \mathbb{R}$. The abstract problem is stated as: find $u \in \mathcal{X}$ such that

$$a(u, v) = \ell(v), \qquad \text{for any } v \in \mathcal{Y}. \tag{2}$$

The link between the two formulations is the following. Let $\mathcal{D}(\Omega)$ be the space of $\mathcal{C}^\infty$ functions with compact support in Ω; the dual space $\mathcal{D}(\Omega)'$ is the space of distributions. We have that:

$$a(u, \varphi) \doteq \langle Au, \varphi \rangle_\Omega, \qquad \ell(\varphi) \doteq \langle g_{\mathrm{N}}, \varphi \rangle_{\Gamma_{\mathrm{N}}} + \langle f, \varphi \rangle_\Omega, \qquad \text{for any } \varphi \in \mathcal{D}(\Omega),$$

where the derivatives are understood in distributional sense. e.g., For the Laplace operator, the bilinear form reads $a(u, v) \doteq \int_\Omega \nabla u \cdot \nabla v \mathrm{d}\Omega$. Furthermore, homogeneous Dirichlet boundary conditions, i.e., $u = 0$ on Γ_{D}, are usually enforced in a strong way; the functions in $\mathcal{Y}$ satisfy these boundary conditions. The extension to non-homogeneous boundary conditions is straightforward. One can define an arbitrary extension Eu_{D} of the Dirichlet data, i.e., $Eu_{\mathrm{D}} = u_{\mathrm{D}}$ on Γ_{D}. Next, we define the function $u_0 \doteq u - Eu_{\mathrm{D}}$ with zero trace on Γ_{D} and solve (2) for u_0 with the right-hand side

$$\ell(v) - a(Eu_{\mathrm{D}}, v). \tag{3}$$

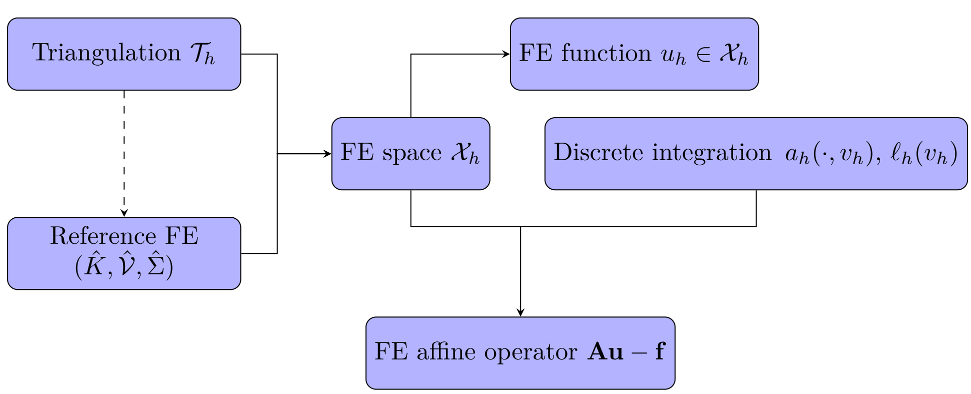


Fig. 1 Main software abstractions in FEMPAR and some of their relationships

Let us consider two classical examples.

Example 3.1 (Heat equation) Let us consider the Poisson problem $-\nabla \cdot \boldsymbol{\kappa}\nabla u = f$ with $u = u_{\mathrm{D}}$ on Γ_{D} and $\partial_{\boldsymbol{n}} u = g_{\mathrm{N}}$; $\boldsymbol{n}$ is the outward normal. Let us assume that $\boldsymbol{\kappa} \in L^{\infty}(\Omega)^{d\times d}$, $f \in H^{-1}(\Omega)$, $g_{\mathrm{N}} \in H^{-\frac{1}{2}}(\Gamma_{\mathrm{N}})$, and $u_{\mathrm{D}} \in H^{\frac{1}{2}}(\Gamma_{\mathrm{D}})$. Let us also consider an extension $Eu_{\mathrm{D}} \in H^1(\Omega)$ such that $Eu_{\mathrm{D}} = u_{\mathrm{D}}$ on Γ_{D}. The weak form of the problem reads as: find $u_0 \in H_0^1(\Omega)$ such that

$$\int_{\Omega} \boldsymbol{\kappa}\nabla u_0 \cdot \nabla v \mathrm{d}\Omega = \int_{\Omega} fv \mathrm{d}\Omega + \int_{\Gamma_{\mathrm{N}}} gv \mathrm{d}\Gamma \\ - \int_{\Omega} \boldsymbol{\kappa}\nabla Eu_{\mathrm{D}} \cdot \nabla v \mathrm{d}\Omega, \qquad \text{for any } v \in H_0^1(\Omega).$$

The solution is $u \doteq u_0 + Eu_{\mathrm{D}}$.

Example 3.2 (Stokes problem) The Stokes problem consists on finding a velocity field $\boldsymbol{u}$ and a pressure field p such that

$$-\nabla \cdot (\mu \boldsymbol{\epsilon}(\boldsymbol{u})) + \nabla p = \boldsymbol{f}, \qquad \nabla \cdot \boldsymbol{u} = 0,$$

and (for example) $\boldsymbol{u} = \boldsymbol{u}_{\mathrm{D}}$ on Γ, where $\boldsymbol{\epsilon}(\boldsymbol{u}) = \frac{1}{2}(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T)$ is the strain tensor. The weak form of the problem consists of finding $(\boldsymbol{u}_0, p) \in \mathcal{X} \doteq \left[H_0^1(\Omega)\right]^d \times L_0^2(\Omega)$ such that

$$\mu \int_{\Omega} \boldsymbol{\epsilon}(\boldsymbol{u}_0) : \boldsymbol{\epsilon}(\boldsymbol{v}) - \int_{\Omega} \nabla \cdot \boldsymbol{v} p + \int_{\Omega} q \nabla \cdot \boldsymbol{u}_0 \\ = \int_{\Omega} \boldsymbol{v} \cdot \boldsymbol{f} - \mu \int_{\Omega} \boldsymbol{\epsilon}(\boldsymbol{E}\boldsymbol{u}_{\mathrm{D}}) : \boldsymbol{\epsilon}(\boldsymbol{v}) - \int_{\Omega} q \nabla \cdot \boldsymbol{E}\boldsymbol{u}_{\mathrm{D}},$$

for any $(\boldsymbol{v}, q) \in \mathcal{X}$, where $\boldsymbol{E}\boldsymbol{u}_{\mathrm{D}} \in \left[H_0^1(\Omega)\right]^d$ is an extension of the Dirichlet data, i.e., $\boldsymbol{E}\boldsymbol{u}_{\mathrm{D}} = \boldsymbol{u}_{\mathrm{D}}$ on Γ. The solution is $\boldsymbol{u} \doteq \boldsymbol{u}_0 + \boldsymbol{E}\boldsymbol{u}_{\mathrm{D}}$.

3.2 Space Discretization with FEs

Problem (2) is an infinite-dimensional problem. In order to end up with a computable one, we must introduce finite-dimensional subspaces with some approximability properties. We restrict ourselves to FE schemes in a broad sense that involve conforming and non-conforming spaces. Thus, our aim is to explicitly build spaces $\mathcal{X}_h$ (and $\mathcal{Y}_h$) with some approximability properties. If the discrete spaces are subspaces of the original ones (conforming), i.e., $\mathcal{X}_h \subset \mathcal{X}$ and $\mathcal{Y}_h \subset \mathcal{Y}$, the discrete problem reads as: find $u_h \in \mathcal{X}_h$ such that

$$a(u_h, v_h) = \ell(v_h), \qquad \text{for any } v_h \in \mathcal{Y}_h.$$

This is the *Petrov-Galerkin* problem. In the particular case when $\mathcal{X}_h = \mathcal{Y}_h$, we have a *Galerkin* problem. The previous problem can be ill-posed for some choices of the FE spaces, e.g., using discrete spaces that do not satisfy the inf-sup condition for indefinite problems [5]. In some cases, judiciously chosen perturbations of $a(\cdot,\cdot)$ and $\ell(\cdot)$, represented with $a_h(\cdot,\cdot)$ and $\ell_h(\cdot)$ respectively, can *stabilize* the problem and make it stable and optimally convergent, circumventing the inf-sup condition restriction. In the most general case, we can describe any FE space as: find $u_h \in \mathcal{X}_h$ such that

$$a_h(u_h, v_h) = \ell_h(v_h), \qquad \text{for any } v_h \in \mathcal{Y}_h, \tag{4}$$

replacing the continuous bilinear form by a general *discrete* bilinear form. One can also define the affine operator

$$\mathcal{F}_h(u_h) = a_h(u_h, \cdot) - \ell_h(\cdot) \in \mathcal{Y}_h', \tag{5}$$

and state (4) as: find $u_h \in \mathcal{X}_h$ such that $\mathcal{F}_h(u_h) = 0$. This statement is the one being used for the practical implementation of FE operators in `FEMPAR` (see Sect. 11).

In order to define FE spaces, we require a triangulation $\mathcal{T}_h$ of the domain Ω into a set $\{K\}$ of *cells*. This triangulation is assumed to be conforming, i.e., for two neighbour cells $K^+, K^- \in \mathcal{T}_h$, its intersection $K^+ \cap K^-$ is a *whole k*-face ($k<d$) of both cells (note that k-face refers to a geometrical entity, e.g. cells, faces, edges and vertices for $d = 3$, see Sect. 3.14). In practice, the cells must be expressed as a particular type of mapping over a set of admissible geometries (polytopes, see Sect. 3.14). Thus, for every element $K \in \mathcal{T}_h$, we assume that there is a reference cell $\hat{K}_K$ and a diffeomorphism $\boldsymbol{\Phi}_K : \hat{K} \to K$. In what follows, we usually use the notation $\hat{\boldsymbol{x}} \doteq \boldsymbol{\Phi}_K^{-1}(\boldsymbol{x})$.

The definition of the functional space also relies on a reference functional space as follows: (1) we define a functional space in the reference cell $\hat{K}$; (2) we define a set of functions in the physical cell K via a function mapping; (3) we define the global space as the assemble of cell-based spaces plus continuity constraints between cells. In order to present this process, we introduce the concept of reference FE, FE, and FE space, respectively.

3.3 The FE Concept in the Reference and Physical Spaces

Using the abstract definition of Ciarlet, a FE is represented by the triplet $\{K, \mathcal{V}, \Sigma\}$, where K is a compact, connected, Lipschitz subset of $\mathbb{R}^d$, $\mathcal{V}$ is a vector space of functions, and Σ is a set of linear functionals that form a basis for the dual space $\mathcal{V}'$. The elements of Σ are the so-called DOFs of the FE. We denote the number of moments as n_Σ. The moments can be written as σ_a for $a \in \mathcal{N}_\Sigma \doteq \{1,\ldots,n_\Sigma\}$. We can also define the basis $\{\phi^a\}_{a\in\mathcal{N}_\Sigma}$ for $\mathcal{V}$ such that $\sigma_a(\phi^b) = \delta_{ab}$ for $a, b \in \mathcal{N}_\Sigma$. These functions are the so-called *shape functions* of the FE, and there is a one-to-one mapping between shape functions and DOFs. Given a function v, we define the *local interpolator* for the FE at hand as

$$\pi_K(v) \doteq \sum_{a\in\mathcal{N}_\Sigma} \sigma_a(v)\phi^a. \tag{6}$$

It is easy to check that the interpolation operator is in fact a projection.

In the reference space, we build *reference* FEs $(\hat{K},\hat{\mathcal{V}},\hat{\Sigma})$ as follows. First, we consider a bounded set of possible cell geometries, denoted by $\hat{K}$; see the definition of polytopes in Sect. 3.14. On $\hat{K}$, we build a functional space $\hat{\mathcal{V}}$ and a set of DOFs $\hat{\Sigma}$. We consider some examples of reference FEs in Sects. 3.8, 3.9, and 3.10.

In the physical space, the FE triplet $(K,\mathcal{V},\Sigma)$ on a mesh cell $K\in\mathcal{T}_h$ relies on: (1) a reference FE $(\hat{K},\hat{\mathcal{V}},\hat{\Sigma})$, (2) a geometrical mapping $\mathbf{\Phi}_K$ such that $K\doteq\mathbf{\Phi}_K(\hat{K})$, and (3) a linear bijective function mapping $\hat{\Psi}_K:\hat{\mathcal{V}}\to\hat{\mathcal{V}}$. The functional space in the physical space is defined as $\mathcal{V}\doteq\{\hat{\Psi}_K(\hat{v})\circ\mathbf{\Phi}_K^{-1} : \hat{v}\in\hat{\mathcal{V}}\}$; we will also use $\Psi_K:\hat{\mathcal{V}}\to\mathcal{V}$ defined as $\Psi_K(\hat{v})\doteq\hat{\Psi}_K(\hat{v})\circ\mathbf{\Phi}_K^{-1}$. The set of DOFs in the physical space is defined as $\Sigma\doteq\{\hat{\sigma}\circ\Psi_K^{-1} : \hat{\sigma}\in\hat{\Sigma}\}$. Given the set of shape functions $\{\hat{\phi}^a : a\in\mathcal{N}_{\hat{\Sigma}}\}$ in the reference FE, it is easy to check that $\{\phi_K^a\doteq\Psi_K(\hat{\phi}^a) : a\in\mathcal{N}_{\hat{\Sigma}}\}$ are the set of shape functions of the FE in the physical space.

The reference FE space $\hat{\mathcal{V}}$ is usually a polynomial space. Thus, the first ingredient is to define bases of polynomials; see Sect. 3.4. The analytical expression of the basis of shape functions is not straightforward for complicated definitions of moments; this topic is covered in Sect. 3.5. After that, we will consider how to build global (and conforming) FE spaces in Sect. 3.6, and how to integrate the bilinear forms in the corresponding weak formulation in Sect. 3.7. We finally provide three examples of FEs in Sects. 3.8, 3.9, and 3.10.

3.4 Construction of Polynomial Spaces

Local FE spaces are usually polynomial spaces. Given an order $k\in\mathbb{N}$ and a set $\mathcal{N}_k$ of distinct points (nodes) in $\mathbb{R}$ (we will indistinctly represent nodes by their index i or position x_i), we define the corresponding set of Lagrangian polynomials $\{\ell_0^k,\ldots,\ell_k^k\}$ as:

$$\ell_m^k(x)\doteq\frac{\Pi_{n\in\mathcal{N}_k\backslash\{m\}}(x-x_s)}{\Pi_{n\in\mathcal{N}_k\backslash\{m\}}(x_m-x_s)}. \tag{7}$$

We can also define the Lagrangian basis $\mathcal{L}^k=\{\ell_i^k : 0\le i\le k\}$. This set of polynomials are a basis for k-th order polynomials. We note that $\ell_m^k(x_l)=\delta_{ml}$, for $0\le m,\ l\le k$.

For multi-dimensional spaces, we can define the set of nodes as the Cartesian product of 1D nodes. Given a d-tuple order $\boldsymbol{k}$, we define the corresponding set of nodes for n-cubes as: $\mathcal{N}^{\boldsymbol{k}}\doteq\mathcal{N}^{k_1}\times\cdots\times\mathcal{N}^{k_d}$. Analogously, we define the multi-dimensional Lagrange basis

$$\mathcal{L}^{\boldsymbol{k}}=\{\ell_{\boldsymbol{m}}^{\boldsymbol{k}} : \boldsymbol{m}\in\mathcal{N}^{\boldsymbol{k}}\},\qquad\text{where}\qquad \ell_{\boldsymbol{m}}^{\boldsymbol{k}}(\boldsymbol{x})\doteq\Pi_{i=1}^d\ell_{m_i}^{k_i}(x_i). \tag{8}$$

Clearly, $\ell_{\boldsymbol{t}}^{\boldsymbol{k}}(\boldsymbol{x_s})=\delta_{\boldsymbol{st}}$, for $\boldsymbol{s},\boldsymbol{t}\in\mathcal{N}^{\boldsymbol{k}}$.

This Cartesian product construction leads to a basis for the local FE spaces usually used on n-cubes, i.e., the space of polynomials that are of degree less or equal to k with respect to each variable $x_1,\ldots,x_d$. We can define monomials by a d-tuple $\boldsymbol{\alpha}$ as $p_{\boldsymbol{\alpha}}(\boldsymbol{x})\doteq\Pi_{i=1}^d x_i^{\alpha_i}$, and the polynomial space of order $\boldsymbol{k}$ as $\mathcal{Q}_{\boldsymbol{k}}=\text{span}\{p_{\boldsymbol{\alpha}}(\boldsymbol{x}) : 0\le\alpha_i\le k_i,\ i=1,\ldots,d\}$. We have $\mathcal{Q}_{\boldsymbol{k}}=\text{span}\{\ell : \ell\in\mathcal{L}^{\boldsymbol{k}}\}$.

The definition of polynomial spaces on n-simplices is slightly different. It requires the definition of the space of polynomials of degree equal or less than k in the variables $x_1,\ldots,x_d$. It does not involve a full Cartesian product of 1D Lagrange polynomials (or monomials) but a truncated space, i.e., the corresponding polynomial space of order k is $\mathcal{P}_k=\text{span}\{p_{\boldsymbol{\alpha}}(\boldsymbol{x}) : |\boldsymbol{\alpha}|\le k\}$, with $|\boldsymbol{\alpha}|\doteq\sum_{i=1}^d\alpha_i$. Analogously as for n-cubes, a basis for the dual space of $\mathcal{P}_k$ are the values at the set of nodes $\tilde{\mathcal{N}^k}\doteq\{\boldsymbol{s}\in\mathcal{N}^{k\boldsymbol{1}} : |\boldsymbol{s}|\le k\}$. It generates the typical grad-conforming FEs on n-simplices.

3.5 Construction of the Shape Functions Basis

The analytical expression of shape functions can become very complicated for high order FEs and non-trivial definitions of DOFs, e.g., for electromagnetic applications. Furthermore, to have a code that provides a basis for an arbitrary high order, an automatic generator of shape functions must be implemented. When the explicit construction of the shape functions is not obvious, we proceed as follows.

Let us consider a FE defined by $\{K,\mathcal{V},\Sigma\}$.[4] First, we generate a *pre-basis* $\{\psi^b\}_{b\in\Sigma}$ that spans the local FE space $\mathcal{V}$, e.g., a Lagrangian polynomial basis (see Sect. 3.4). On the other hand, given the set of local DOFs, we proceed as follows. The shape functions can be written as $\phi^a=\sum_{b\in\mathcal{N}_\Sigma}\mathbf{\Phi}_{ab}\psi^b$, where ψ^b are the elements of the pre-basis. By definition, the shape functions must satisfy $\sigma_a(\phi^b)=\delta_{ab}$ for $a,b\in\mathcal{N}_\Sigma$. As a result, let us define $\mathbf{C}_{ab}\doteq\sigma_a(\psi^b)$. We have (using Einstein's notation):

[4] In this section, we do not make difference between reference and physical spaces, e.g., using the $\hat{}$ symbol. In any case, all the following developments are usually performed at the reference FE level.

$$\sigma_a(\phi^b) = \sigma_a(\mathbf{\Phi}_{bc}\psi^c) = \sigma_a(\psi^c)\mathbf{\Phi}_{bc} = \delta_{ab},$$

or in compact form, $\mathbf{C}\mathbf{\Phi}^T = I$, and thus $\mathbf{\Phi}^T = \mathbf{C}^{-1}$. As a result, $\mathbf{\Phi}_{ab} = \mathbf{C}^{-1}_{ba}$. The shape functions are computed as a linear combination of the pre-basis functions.

3.6 Global FE Space and Conformity

Finally, we must define the *global* FE space. Conforming FE spaces are defined as: $\mathcal{X}_h \doteq \{v \in \mathcal{X} : v|_K \in \mathcal{V}\}$. The main complication in this definition is to enforce the conformity of the FE space, i.e., $\mathcal{X}_h \subset \mathcal{X}$. In fact, the conformity constraint is the one that motivates the choice of $\hat{\Sigma}$ and Ψ, and as a consequence, Σ. In practice, the conformity constraint must be re-stated as a continuity constraint over FE DOFs. In general, these constraints are implicitly enforced via a global DOF numbering, even though it is not possible in general for adaptive schemes with non-conforming meshes and/or variable order cells, which require more involved constraints.

Let us define by $\mathcal{M}_h \doteq \{(b,K) : b \in \mathcal{N}_{\Sigma_K},\ K \in \mathcal{T}_h\}$ the Cartesian product of local DOFs for all cells. We define the global DOFs as the quotient space of $\mathcal{M}_h$ by an equivalence relation $\sim$. Using standard notation, given $\sim$, the equivalence class of $a \in \mathcal{M}_h$ with respect to $\sim$ is represented with $[a] \doteq \{b \in \mathcal{M}_h : a \sim b\}$, and the corresponding quotient set is $\mathcal{N}_h \doteq \{[a] : a \in \mathcal{M}_h\}$. The set $\mathcal{N}_h$ is the set of global DOF and $[\cdot]$ represents the local-to-global DOF map. We assume that the equivalence relation is such that if two elements (b,K), $(b',K') \in \mathcal{M}_h$ are such that $(b,K) \sim (b',K')$, then $K \neq K'$.[5] Using the one-to-one mapping between moments and shape functions, the same operator allows one to define global shape functions $\phi^a = \sum_{(b,K)\sim a} \phi^b_K$. We assume that the choices above are such that they satisfy the conformity constraint, i.e., $\mathcal{X}_h = \mathrm{span}\{\phi^a\}_{a\in\mathcal{N}_h} \subset \mathcal{X}$.

Let us consider an infinite-dimensional space $\tilde{\mathcal{X}}$ such that (1) $\mathcal{X}_h \subset \tilde{\mathcal{X}} \subset \mathcal{X}$ and (2) for every function $v \in \tilde{\mathcal{X}}$ and global DOF $a \in \mathcal{N}_h$, all the local DOFs b, $b' \in [a]$ are such that $\sigma_b(v) = \sigma_{b'}(v)$, i.e., local DOF related to the same global DOF are continuous among cells. The *global interpolator* is defined as:

$$\pi_{\mathcal{X}_h}(v) \doteq \sum_{K\in\mathcal{T}_h} \pi_K(v) = \sum_{K\in\mathcal{T}_h}\sum_{b\in\mathcal{N}_{\Sigma_K}} \sigma_b(v)\phi^b_K, \qquad \text{for } v \in \tilde{\mathcal{X}}. \tag{9}$$

It is easy to check that it is in fact a projector. In any case, we use *projection operator* to refer to other projectors that involve the solution of a global FE system, e.g., based on the minimization of the L^2 or H^1 norm.

Below, we provide details about how to choose the local DOFs $\hat{\Sigma}$, the function map Ψ, and the equivalence relation $\sim$ such that the conformity property is satisfied for grad, div, and curl-conforming FE spaces. The case of non-conforming methods, e.g., DG methods, can readily be considered. In this case, the conformity constraint is not required, which leads to much more flexibility in the definition of DOFs. On the other side, these schemes require numerical perturbations of the continuous bilinear and linear forms in (4) that involve integrals over the facets of FEs to *weakly* enforce the conformity. (Facets are $(d-1)$-faces, e.g., faces in 3D and edges in 2D).

Once we have defined a basis for the FE spaces $\mathcal{X}_h$ and $\mathcal{Y}_h$ using the FE machinery presented above, every FE function u_h can be uniquely represented by a vector $\mathbf{u} \in \mathbb{R}^{|\mathcal{N}_h|}$ as $u_h = \sum_{b\in\mathcal{N}_h} \phi^b \mathbf{u}_b$. In fact, problem (4) can be re-stated as: find $\mathbf{u} \in \mathbb{R}^{|\mathcal{N}_h|}$ such that

$$a_h(\phi^b,\psi^a)\mathbf{u}_b = \ell_h(\psi^a), \qquad \text{for any } a \in \mathcal{N}_h.$$

We have ended up with a finite-dimensional linear problem, i.e., a linear system. We note that in general, the trial space moments can be different from the ones of the test space, as soon as the cardinality is the same. In matrix form, the problem can be stated as:

$$\text{Solve} \quad \mathbf{A}\mathbf{u} = \mathbf{f}, \qquad \text{with} \quad \mathbf{A}_{ab} \doteq a_h(\phi^b,\psi^a), \quad \mathbf{f}_a \doteq \ell_h(\psi^a). \tag{10}$$

Assuming that the bilinear form can be split into cell contributions as $a_h(\cdot,\cdot) = \sum_{K\in\mathcal{T}_h} a_K(\cdot,\cdot)$, e.g., by replacing $\int_\Omega$ by $\sum_{K\in\mathcal{T}_h}\int_K$, the construction of the matrix is implemented through a cell-wise assembly process, as follows:

$$\mathbf{A}_{[a][b]} = \sum_{K\in\mathcal{T}_h}\sum_{a,b\in\mathcal{N}_{\Sigma_K}} \mathbf{A}^K_{ab} \doteq \sum_{K\in\mathcal{T}_h}\sum_{a,b\in\mathcal{N}_{\Sigma_K}} a_K(\phi^b_K,\psi^a_K). \tag{11}$$

The FE affine operator (5) can be represented as $\mathcal{F}_h(u_h) \doteq \mathbf{A}\mathbf{u} - \mathbf{f}$, i.e., it can be represented with a matrix and a vector of size $|\mathcal{N}_h|$.

3.7 Numerical Integration

In general, the local bilinear form can be stated as:

$$a_K(\phi^b_K,\psi^a_K) = \int_K \boldsymbol{\mathcal{F}}(\boldsymbol{x})\mathrm{d}\Omega,$$

where the evaluation of $\boldsymbol{\mathcal{F}}(\boldsymbol{x})$ involves the evaluation of shape function derivatives. Let us represent the Jacobian of the geometrical mapping with $\boldsymbol{J}_K \doteq \frac{\partial\mathbf{\Phi}_K}{\partial\boldsymbol{x}}$. We can rewrite the cell integration in the reference cell, and next consider a

[5] This assumption in fact applies for FEs of any kind, since the local functional spaces are already conforming and do not require an equivalence class at the cell level.

quadrature rule Q defined by a set of points/weights $(\hat{\boldsymbol{x}}_{\mathrm{gp}}, \mathrm{w}_{\mathrm{gp}})$, as follows:

$$\int_K \mathcal{F}(\boldsymbol{x})\mathrm{d}\Omega = \int_{\hat{K}} \mathcal{F}\circ\boldsymbol{\Phi}(\boldsymbol{x})|\boldsymbol{J}_K|\mathrm{d}\Omega = \sum_{\hat{\boldsymbol{x}}_{\mathrm{gp}}\in\mathrm{Q}} \mathcal{F}\circ\boldsymbol{\Phi}(\hat{\boldsymbol{x}}_{\mathrm{gp}})\mathrm{w}(\hat{\boldsymbol{x}}_{\mathrm{gp}})|\boldsymbol{J}_K(\hat{\boldsymbol{x}}_{\mathrm{gp}})|. \tag{12}$$

Here, the main complication is the evaluation of $\mathcal{F}\circ\boldsymbol{\Phi}(\hat{\boldsymbol{x}}_{\mathrm{gp}})$. By construction, the evaluation of this functional only requires the evaluation of $\partial_{\boldsymbol{\alpha}}\phi_K^b\circ\boldsymbol{\Phi}(\hat{\boldsymbol{x}}_{\mathrm{gp}})$ for some values of the multi-index $\boldsymbol{\alpha}$ (idem for the test functions). Usually, $|\boldsymbol{\alpha}|\leq 2$ in $\mathcal{C}^0$ FEs, since higher-order derivatives would require higher inter-cell continuity. The second derivatives, which only have sense for *broken* cell-wise integrals, are in fact only needed for some method based on *stabilization* techniques based on the pointwise evaluation of residuals in the interior of cells [46].

Let us consider the case of zero and first derivatives, i.e., the evaluation of $\phi_K^b\circ\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}_{\mathrm{gp}})$ and $\nabla\phi_K^b\circ\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}_{\mathrm{gp}})$. The values of the shape functions (times the geometrical mapping) on the quadrature points is determined as follows:

$$\phi_K^b\circ\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}_{\mathrm{gp}}) = \hat{\Psi}(\hat{\phi}^b)(\hat{\boldsymbol{x}}_{\mathrm{gp}}), \tag{13}$$

whereas shape function gradients are computed as:

$$\begin{aligned}\nabla\phi_K^b\circ\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}_{\mathrm{gp}}) &= \nabla(\hat{\Psi}(\hat{\phi}^b)\circ\boldsymbol{\Phi}_K^{-1})\circ\boldsymbol{\Phi}_K(\hat{\boldsymbol{x}}_{\mathrm{gp}})\\ &= \nabla_{\hat{\boldsymbol{x}}}\hat{\Psi}(\hat{\phi}^b)(\hat{\boldsymbol{x}}_{\mathrm{gp}})\boldsymbol{J}_K^{-1}(\hat{\boldsymbol{x}}_{\mathrm{gp}}),\end{aligned} \tag{14}$$

where we have used some elementary differentiation rules and the inverse function theorem in the last equality; $\nabla_{\hat{\boldsymbol{x}}}$ represents the gradient in the reference space. Thus, one only needs to provide the values of the Jacobian, its inverse, and its determinant, from one side, and the value of the shape functions $\Psi(\hat{\phi}^b)$ and their gradients $\nabla_{\hat{\boldsymbol{x}}}\Psi(\hat{\phi}^b)$ in the reference space, on the other side, at all quadrature points, to compute all the entries of the FE matrices; second order derivatives can be treated analogously.

Quadrature rules for $\hat{K}$ being an n-cube can readily be obtained as a tensor product of a 1D quadrature rule, e.g., the Gauss-Legendre quadrature. Symmetric quadrature rules on triangles and tetrahedra for different orders can be found, e.g., in [69]. In any case, to create arbitrarily large quadrature rules for n-simplices, one can consider the so-called Duffy transformation [73, 74].

As it is well known, considering n-cube topologies for $\hat{K}$, Gauss quadratures with n points per direction can integrate *exactly* $2n-1$ order polynomials. e.g., For a Lagrangian reference FE of order p and an affine geometrical map, we choose $n = p + \mathrm{ceiling}(1/2) = p+1$ per direction to integrate exactly a mass matrix. For n-simplex meshes, we use either symmetric quadratures (if available) or tensor product rules plus the Duffy transformation [73, 74]. The latter case is based on introducing a change of variables that transform our n-simplex integration domain into an n-cube, and integrate on the n-cube using tensor product quadratures. It is worth noting that this change of variables introduces a non-constant Jacobian. The determinant of the Jacobian is of order at most $d-1$ with respect to each variable. To integrate a mass matrix exactly, we must be able to integrate exactly polynomials of order $2p+d-1$. Therefore, we need to take $n = p + \mathrm{ceiling}(d/2)$ to exactly integrate mass matrices.

3.8 Grad-Conforming FEs: Lagrangian (Nodal) Elements

In this section, we consider one characterization of the abstract FE technology above. First, we are interested in the so-called nodal FEs, based on Lagrange polynomials and DOFs based on nodal values.

Let us consider the same order for all components, i.e., $k\boldsymbol{1} \doteq (k,\ldots,k)$. When the reference geometry $\hat{K}$ is an n-cube, we define the reference FE space as $\mathcal{V}_k \doteq \mathcal{Q}_{k\boldsymbol{1}}$. The set of nodes $\mathcal{N}^{k\boldsymbol{1}}$ can be generated, e.g., from the equidistant Lagrangian nodes. Let us define the bijective mapping $\mathtt{i}(\cdot)$ from the set of nodes $\mathcal{N}^{k\boldsymbol{1}}$ to $\{1,\ldots,|\mathcal{N}^{k\boldsymbol{1}}|\}\equiv\mathcal{N}_\Sigma$, i.e., the local node numbering. The set of local DOFs $\mathcal{N}_{\Sigma_K}$ are the nodal values, i.e., $\sigma_{\mathtt{i}(\boldsymbol{s})} \doteq v(\boldsymbol{x}_{\boldsymbol{s}})$, for $\boldsymbol{s}\in\mathcal{N}^{\boldsymbol{k}}$. Clearly, the reference FE shape functions related to these DOFs are $\phi^{\mathtt{i}(\boldsymbol{s})} \doteq \ell_{\boldsymbol{s}}^{k\boldsymbol{1}}$. On the other hand, we simply take $\hat{\Psi}(v)\doteq v$.

For n-simplices, we consider the reference FE space $\mathcal{P}_k$ spanned by the pre-basis $\{p_{\boldsymbol{\alpha}}(\boldsymbol{x}) \,:\, 0\leq\alpha_i\leq k,\ i=1,\ldots,d\}$ and the set of nodes $\tilde{\mathcal{N}}^k$ (see Sect. 3.4). The set of local DOFs $\mathcal{N}_{\Sigma_K}$ are the nodal values. Since the pre-basis elements are not shape functions, we proceed as in Sect. 3.5 to generate the expression of the shape functions basis for arbitrary order reference FEs on n-simplices.

The global FE space is determined by the following equivalence relation. The set of local DOFs for n-cubes is $\mathcal{M}_h \doteq \{(\boldsymbol{s},K) \,:\, \boldsymbol{s}\in\mathcal{N}^{k\boldsymbol{1}}, K\in\mathcal{T}_h\}$ due to the one-to-one mapping between DOFs and nodes; we replace the set of nodes by $\tilde{\mathcal{N}}^k$ for n-simplices. Furthermore, we say that $(\boldsymbol{s},K)\sim(\boldsymbol{s}',K')$ iff $\boldsymbol{x}_{\boldsymbol{s}} = \boldsymbol{x}_{\boldsymbol{s}'}$. The implementation of this equivalence relation, and thus, the global numbering, relies on the ownership relation between n-faces and DOFs (e.g., in 3D we can say whether a DOF belongs to a vertex, edge, or face) and a permutation between the local node numbering in K^+ to the one in K^- for nodes on F. See Sect. 3.14 for more details. With such global DOF definition, it is easy to check that the global FE space functions are $\mathcal{C}^0$ and thus grad-conforming.

Since Lagrangian moments involve point-wise evaluations of functions and $H_0^1(\Omega)\not\subset\mathcal{C}^0(\Omega)$ for $d>1$, the interpolator (9) is not defined in such space. Instead, we consider that functions to be interpolated belong, e.g., to the space $\tilde{\mathcal{X}} \doteq \mathcal{C}^0(\Omega)$.

When one has to deal with vector or tensor fields, we can generate them as a Cartesian product of scalar spaces as follows. We define the local FE space $\boldsymbol{\mathcal{V}}_k \doteq [\mathcal{Q}_{k\mathbf{1}}]^d$ and the function map $\hat{\Psi}(\boldsymbol{v}) \doteq \boldsymbol{v}$. In the vector case, the local DOFs set is represented with $\mathcal{M}_h \doteq \{(i, \boldsymbol{s}, K) : 1 \le i \le d, \boldsymbol{s} \in \mathcal{N}^{k\mathbf{1}}, K \in \mathcal{T}_h\}$, and $(i, \boldsymbol{s}, K) \sim (i', \boldsymbol{s}', K')$ iff $i = i'$ and $\boldsymbol{x}_{\boldsymbol{s}} = \boldsymbol{x}_{\boldsymbol{s}'}$. Analogously, shape functions are computed as $\phi^a \doteq \sum_{(i,\boldsymbol{s},K)\sim a} \ell_{\boldsymbol{s}}^{k\mathbf{1}} \boldsymbol{e}_i$; $\boldsymbol{e}_i$ represents the i-th canonical basis vector of $\mathbb{R}^d$. We proceed analogously for n-simplices.

The verification that two nodes are in the same position is not straightforward. First, for every node $\boldsymbol{s}$ in K, we can assign an n-face owner F (e.g., a vertex, edge, face, or cell); cell DOFs are not replicated. Given a node $\boldsymbol{s} \in \mathcal{N}^{k\mathbf{1}}$ of cell K that belongs to the n-face F, it can be determined by an index $\boldsymbol{s}_F$ with respect to F and K. Analogously, another node that belongs to the same n-face but cell K', is represented by $\boldsymbol{s}'_F$. On the other hand, one can define a permutation mapping

$$\mathrm{p}_F(F, K, K'; \cdot), \tag{15}$$

that, given the local index of a node within the n-face F with respect to K, it provides the index in the n-face F with respect to K' (see Sects. 3.13 and 3.16 for more details). Thus, $\boldsymbol{x}_{\boldsymbol{s}} = \boldsymbol{x}_{\boldsymbol{s}'}$ iff $\mathrm{p}_F(F, K, K'; \boldsymbol{s}_F) = \boldsymbol{s}'_F$.

3.9 Div-Conforming FEs

We present the so-called Raviart-Thomas FEs for vector fields [5]; the implementation of Brezzi-Douglas-Marini FEs is analogous. In this case, the order being used is different at every space dimension. Let us start with Raviart-Thomas FEs on n-cubes. In 2D, the space reads as $\boldsymbol{\mathcal{V}}_k \doteq \mathcal{Q}_{(k+1,k)} \times \mathcal{Q}_{(k,k+1)}$, whereas in 3D it reads as $\boldsymbol{\mathcal{V}}_k \doteq \mathcal{Q}_{(k+1,k,k)} \times \mathcal{Q}_{(k,k+1,k)} \times \mathcal{Q}_{(k,k,k+1)}$; the Raviart-Thomas element can in fact be considered for any dimension. The basis for Σ in 3D is composed of two types of DOFs, boundary and interior DOFs, defined as

$$\begin{aligned}
&\frac{1}{\|\hat{F}_0\|}\int_{\hat{F}_0} \boldsymbol{v}\cdot\boldsymbol{n}\circ\boldsymbol{\Phi}_{\hat{F}}\, q \mathrm{d}\Gamma, \quad q \in \mathcal{P}_k,\\
&\quad\frac{1}{\|\hat{K}\|}\int_{\hat{K}} \boldsymbol{v}\cdot\, \boldsymbol{q}\mathrm{d}\Omega, \quad \boldsymbol{q} \in \mathcal{Q}_{(k-1,k,k)}\\
&\quad\times \mathcal{Q}_{(k,k-1,k)} \quad \times \mathcal{Q}_{(k,k,k-1)},
\end{aligned} \tag{16}$$

respectively[6]; the 2D case is straightforward, replacing the space of shape functions for the interior moments by $\mathcal{Q}_{(k-1,k)} \times \mathcal{Q}_{(k,k-1)}$. The definition of the boundary facets involves mappings from a reference facet $\hat{F}_0$ to all facets $\hat{F}$ of the FE K, i.e., $\boldsymbol{\Phi}_{\hat{F}} : \hat{F}_0 \to \hat{F}$. Every boundary moment can be associated to a function in a Lagrangian space, and thus, a node index. As a result, the boundary DOFs can be indexed with a node in $\mathcal{N}^{k\mathbf{1}}$ (for $d = 2$) on the corresponding facet F, i.e., $\mathcal{M}_h^{\partial} \doteq \{(F, \boldsymbol{s}, K) : F$ are facets of $K, \boldsymbol{s} \in \mathcal{N}^{k\mathbf{1}}, K \in \mathcal{T}_h\}$. We say that $(F, \boldsymbol{s}, K) \sim (F', \boldsymbol{s}', K')$ iff $F = F'$ and $\boldsymbol{x}_{\boldsymbol{s}} = \boldsymbol{x}_{\boldsymbol{s}'}$. To check whether $\boldsymbol{x}_{\boldsymbol{s}} = \boldsymbol{x}_{\boldsymbol{s}'}$ holds, we can proceed similarly as for Lagrangian elements. The shape functions are built as in Sect. 3.5. We consider a Lagrangian pre-basis for $\mathcal{V}$, and compute the shape functions via a change-of-basis. The function mapping reads as follows:

$$\hat{\Psi}_K(\boldsymbol{v}) \doteq \frac{1}{|\boldsymbol{J}_K|}\boldsymbol{J}_K \boldsymbol{v}; \tag{17}$$

the mapping $\hat{\Psi}_K \circ \boldsymbol{\Phi}_K^{-1}$ is the so-called contravariant Piola transformation. One can check that the definition of this mapping together with the assembly defined above leads to a global FE space that is div-conforming; i.e., its functions have continuous normal component across inter-cell facets. Thus, $\mathcal{X}_h \subset H(\mathrm{div}, \Omega)$ [5].

On n-simplices, the reference FE space is $\boldsymbol{\mathcal{V}}_k \doteq [\mathcal{P}_k]^d \times \boldsymbol{x}\mathcal{P}_k$, for $k = 0, 1, 2, \ldots$, and the basis for Σ is composed of the following boundary and interior DOFs:

$$\begin{aligned}
&\frac{1}{\|\hat{F}_0\|}\int_{\hat{F}_0} \boldsymbol{v}\cdot\boldsymbol{n}\circ\boldsymbol{\Phi}_{\hat{F}}\, q \mathrm{d}\Gamma, \quad q \in \mathcal{P}_k,\\
&\frac{1}{\|\hat{K}\|}\int_{\hat{K}} \boldsymbol{v}\cdot\, \boldsymbol{q}\mathrm{d}\Omega, \quad \boldsymbol{q} \in [\mathcal{P}_{k-1}]^d.
\end{aligned}$$

In this case, the generation of the pre-basis is not a Lagrangian FE space of functions, but it can easily be expressed as the span of vector functions with components in a selected subset of $\mathcal{P}_{k+1}$.

3.10 Curl-Conforming FEs

The weak formulation of electromagnetic problems involve the functional space $H(\mathbf{curl}, \Omega)$. Conforming FE spaces for $H(\mathbf{curl}, \Omega)$ must preserve the continuity of the tangential component of the field. The so-called edge elements (or Nédélec elements) are curl-conforming FEs [72]. As Raviart-Thomas elements, the edge elements pre-basis on n-cubes involves different orders per dimension and per component. In 2D, the space reads as $\boldsymbol{\mathcal{V}}_k \doteq \mathcal{Q}_{(k-1,k)} \times \mathcal{Q}_{(k,k-1)}$, whereas in 3D it reads as $\boldsymbol{\mathcal{V}}_k \doteq \mathcal{Q}_{(k-1,k,k)} \times \mathcal{Q}_{(k,k-1,k)} \times \mathcal{Q}_{(k,k,k-1)}$. The basis for Σ is composed of three types of DOFs (in 3D), namely edge, face, and interior DOFs, defined as:

[6] The test function spaces in the definition of the moments are always considered with respect to the corresponding domain of integration.

$$\frac{1}{\|\hat{E}_0\|}\int_{\hat{E}_0}(\boldsymbol{v}\cdot\boldsymbol{\tau})\circ\boldsymbol{\Phi}_{\hat{E}}q\,\mathrm{d}\Lambda,\ \ \forall q\in\mathcal{P}_{k-1},$$
$$\frac{1}{\|\hat{F}_0\|}\int_{\hat{F}_0}(\boldsymbol{J}_{\hat{F}}^T(\boldsymbol{v}\times\boldsymbol{n}))\circ\boldsymbol{\Phi}_{\hat{F}}\cdot\boldsymbol{q}\,\mathrm{d}\Gamma,\ \ \forall\boldsymbol{q}\in\mathcal{Q}_{(k-2,k-1)}\times\mathcal{Q}_{(k-1,k-2)},$$
$$\frac{1}{\|\hat{K}\|}\int_{\hat{K}}\boldsymbol{v}\cdot\boldsymbol{q}\,\mathrm{d}\Omega,\ \ \forall\boldsymbol{q}\in\mathcal{Q}_{(k-1,k-2,k-2)}\times\mathcal{Q}_{(k-2,k-1,k-2)}\times\mathcal{Q}_{(k-2,k-2,k-1)},$$

respectively, where the edge map $\boldsymbol{\Phi}_{\hat{E}}$ is defined as the one for the face. The boundary DOFs can be indexed by a triplet $(F,\boldsymbol{s},K)$, where F can be an edge or a face in 3D, following the same ideas as for Raviart-Thomas elements. In this case, the function mapping reads as follows:

$$\hat{\Psi}_K(\boldsymbol{v})\doteq\boldsymbol{J}_K^{-T}\boldsymbol{v}; \tag{18}$$

the mapping $\hat{\Psi}_K\circ\boldsymbol{\Phi}_K^{-1}$ is the so-called covariant Piola transformation, which leads to a global FE space that is curl-conforming [72], i.e., its functions have continuous tangential component across inter-cell facets.

On n-simplices, the space reads as:

$$\boldsymbol{\mathcal{V}}_k\doteq[\mathcal{P}_k]^d+\boldsymbol{\mathcal{S}}_k,\ \ \text{where}\ \ \boldsymbol{\mathcal{S}}_k\doteq\{\boldsymbol{v}\in[\mathcal{P}_{k+1}]^d\ :\ \boldsymbol{v}(\boldsymbol{x})\cdot\boldsymbol{x}=0\,\forall\boldsymbol{x}\in\hat{K}\}. \tag{19}$$

The basis for Σ in 3D is composed of the following boundary and interior DOFs:[7]

$$\frac{1}{\|\hat{E}_0\|}\int_{\hat{E}_0}(\boldsymbol{v}\cdot\boldsymbol{\tau})\circ\boldsymbol{\Phi}_{\hat{E}}q\,\mathrm{d}\Lambda,\quad\forall q\in\mathcal{P}_{k-1},$$
$$\frac{1}{\|\hat{F}_0\|}\int_{\hat{F}_0}(\boldsymbol{J}_{\hat{F}}^T(\boldsymbol{v}\times\boldsymbol{n}))\circ\boldsymbol{\Phi}_{\hat{F}}\cdot\boldsymbol{q}\,\mathrm{d}\Gamma,\quad\forall\boldsymbol{q}\in[\mathcal{P}_{k-2}]^2$$
$$\frac{1}{\|\hat{K}\|}\int_{\hat{K}}\boldsymbol{v}\cdot\boldsymbol{q}\,\mathrm{d}\Omega,\quad\forall\boldsymbol{q}\in[\mathcal{P}_{k-3}]^3.$$

In 2D, only the first two types of DOFs are required, where the first one is now related to facets (edges in 2D) and the second one are interior DOFs owned by the cell. As for Raviart-Thomas elements, the pre-basis functions are not Lagrangian shape functions, but they can again be expressed as the span of vector functions with components in a selected subset of $\mathcal{P}_{k+1}$. We refer to [75] for a discussion about the actual generation of a pre-basis for the space (19) in FEMPAR.

3.11 Cartesian Product of FEs for Multi-field Problems

Many problems governed by PDEs involve more than one field, e.g., the Navier-Stokes equations or any multi-physics problem. Let us consider a PDE that involves a set of unknown fields $(\boldsymbol{u}_1,\ldots,\boldsymbol{u}_n)\in\mathcal{X}^1\times\ldots\times\mathcal{X}^n$, defined as the Cartesian product of functional spaces. We can proceed as above, and define a FE space for every field space separately, leading to a global FE space $\mathcal{X}_h^1\times\ldots\times\mathcal{X}_h^n$ defined by composition of FE spaces. To define the global numbering of DOFs in the multi-field case, we consider that two DOFs are equivalent if they are related to the same field and satisfy the equivalence relation of the FE space of this field.

The Cartesian product of FE spaces is enough to define volume-coupling multi-physics problems governed on the same physical domain, i.e., the different physics are defined on the whole domain and coupled through volume terms in the formulation. However, many multi-physics problems are interface-based, i.e., the coupling between different physics that are defined on different subdomains is through transmission conditions on the interface. This is the case, e.g., of fluid-structure problems (see, e.g., [76–79]). In these cases, different FE spaces could be defined on different parts of the global mesh, i.e., one must describe the set of subdomains $(\Omega_1,\ldots,\Omega_n)$ of the whole domain Ω in which the corresponding FE spaces are defined.

3.12 Non-conforming Methods

Up to now, we have considered a global FE space that is conforming, i.e., $\mathcal{X}_h\subset\mathcal{X}$. Alternatively, one can consider FE schemes that are not conforming. Since the original bilinear form has no sense in general for a non-conforming FE space $\mathcal{X}_h$, one shall consider a *stabilized* bilinear form a_h that is well-posed (stable and continuous) in the discrete setting. In general, these schemes replace the required inter-cell continuity for conformity by a weak imposition of such continuity. Thus, the inter-cell continuity is imposed weakly through penalty-like terms. DG methods are schemes of this type [71].

In one sense, non-conforming FE spaces are simpler than conforming ones, since the conformity is not required; one has more flexibility in the definition of local DOFs and the equivalence class concept is not needed, since a DOF never belongs to more than one element. However, the bilinear form usually requires the integration of facet terms, i.e., terms of the type:

$$\sum_{F\in\mathcal{F}_h}\int_F\boldsymbol{\mathcal{F}}(\boldsymbol{x})\mathrm{d}\Omega.$$

The integration of facet terms is far more complicated than cell terms.

Let us first briefly illustrate a simple application of non-conforming methods, namely the FE discretization of the Poisson problem using the so-called interior penalty (IP)

[7] We note that we can take $\boldsymbol{J}_{\hat{F}}^T\boldsymbol{v}$ instead of $\boldsymbol{J}_{\hat{F}}^T(\boldsymbol{v}\times\boldsymbol{n})$ in the definition of the face moments, since the rows of the Jacobian matrix are the transformation of the axes in the reference face $\hat{F}_0$ to the actual face $\hat{F}$ of the reference cell and the space of test functions is invariant to rotations.

family of DG formulations [71]. Dirichlet boundary conditions constraints, say $u(x) = u_{\mathrm{D}}(x)$ on the whole boundary Γ of the domain Ω, are to be weakly imposed, as it is natural in such kind of formulations. The global discrete trial space $\mathcal{X}_h$ is composed of functions that are continuous within each cell, but discontinuous across cells, i.e., $\mathcal{X}_h = \{u_h \in L_2(\Omega) : u_h|_K \in \mathcal{X}_h|_K \subset H^1(K),\ K \in \mathcal{T}_h\}$, and the discrete test space $\mathcal{Y}_h = \mathcal{X}_h$. If we denote $\mathcal{F}_h^{\Omega}$ and $\mathcal{F}_h^{\Gamma}$ as the set of interior and boundary facets of $\mathcal{T}_h$, respectively, the discrete weak form underlying this family of methods reads as: find $u_h \in \mathcal{X}_h$ such that

$$\begin{aligned}
&\sum_{K\in\mathcal{T}_h}\int_K \nabla u_h\cdot\nabla v_h - \sum_{F\in\mathcal{F}_h^{\Omega}}\int_F [\![v_h]\!]\cdot\{\!\{\nabla u_h\}\!\}\\
&\quad - \tau\sum_{F\in\mathcal{F}_h^{\Omega}}\int_F [\![u_h]\!]\cdot\{\!\{\nabla v_h\}\!\}\\
&\quad + \sum_{F\in\mathcal{F}_h^{\Omega}}\gamma|F|^{-1}\int_F [\![u_h]\!]\cdot[\![v_h]\!]\\
&\quad - \sum_{F\in\mathcal{F}_h^{\Gamma}}\int_F v_h\nabla u_h\cdot\boldsymbol{n} - \tau\sum_{F\in\mathcal{F}_h^{\Gamma}}\int_F u_h\nabla v_h\cdot\boldsymbol{n}\\
&\quad + \sum_{F\in\mathcal{F}_h^{\Gamma}}\gamma|F|^{-1}\int_F u_h v_h\\
&= \sum_{K\in\mathcal{T}_h}\int_K f v_h - \tau\sum_{F\in\mathcal{F}_h^{\Gamma}}\int_F u_{\mathrm{D}}\nabla v_h\cdot\boldsymbol{n}\\
&\quad + \sum_{F\in\mathcal{F}_h^{\Gamma}}\gamma|F|^{-1}\int_F u_{\mathrm{D}} v_h \qquad \forall v_h\in\mathcal{Y}_h,
\end{aligned} \tag{20}$$

where τ is a fixed constant that characterizes the particular method at hand, γ is a facet-wise positive constant referred to as penalty parameter, and $|F|$ denotes the surface of the facet; τ and γ should be suitably chosen such that the bilinear form $a_h(u_h, v_h)$ on the left-hand side of (20) is well-posed (stable and continuous) in the discrete setting, and the resulting FE formulation enjoys optimal rates of convergence [71]. Finally, if we denote as K^+ and K^- the two cells that share a given facet, then $\{\!\{w_h\}\!\}$ and $[\![w_h]\!]$ denote mean values and jumps of w_h across cells facets:

$$\{\!\{w_h\}\!\} = \frac{w_h^+ + w_h^-}{2}, \qquad [\![w_h]\!] = w_h^+\boldsymbol{n}^+ + w_h^-\boldsymbol{n}^-, \tag{21}$$

with $\boldsymbol{n}^+$, $\boldsymbol{n}^-$ being the facet outward unit normals, and w_h^+, w_h^- the restrictions of w_h to the facet, both from either the perspective of K^+ and K^-, respectively.

The computation and assembly of DOFs related to interior nodes is straightforward. With regard to the facet terms, assuming that we are sitting on an interior facet $F \in \mathcal{F}_h^{\Omega}$, four facet-wise matrices, namely $\mathbf{A}^F_{K^+K^+}$, $\mathbf{A}^F_{K^+K^-}$, $\mathbf{A}^F_{K^-K^+}$, and $\mathbf{A}^F_{K^-K^-}$, are computed. (The case of boundary facets $F \in \mathcal{F}_h^{\Gamma}$ is just a degenerated case of the one corresponding to interior facets where only a single facet-wise matrix $\mathbf{A}^F_{K^+K^+}$ has to be computed; we omit this sort of facets from the discussion in order to keep the presentation short.) These hold all partial contributions of the facet to the corresponding global entries of the coefficient matrix. The entries of, e.g., $\mathbf{A}^F_{K^+K^-}$, are defined (for our particular problem at hand) as:

$$\begin{aligned}
\left(\mathbf{A}^F_{K^+K^-}\right)_{ab} &= -\int_F [\![\phi^b_{K^-}]\!]\cdot\{\!\{\nabla\phi^a_{K^+}\}\!\}\\
&- \tau\int_F [\![\phi^a_{K^+}]\!]\cdot\{\!\{\nabla\phi^b_{K^-}\}\!\} + \gamma|F|^{-1}\int_F [\![\phi^a_{K^+}]\!]\cdot[\![\phi^b_{K^-}]\!],
\end{aligned} \tag{22}$$

with indices a and b ranging from 1 to the number of shape functions $\mathcal{N}_\Sigma$ of K^+ and K^-, respectively.

3.13 Facet Integration

As mentioned in Sect. 3.7 for the case of cell integrals, facet integrals involved in the computation of the facet-wise matrix (22) cannot be in general computed analytically. These are instead computed using quadrature rules. In general, the bilinear form that contains the facet terms can be stated as

$$a_F(\phi^b_{K^+}, \psi^a_{K^-}) = \int_F \boldsymbol{\mathcal{F}}(\boldsymbol{x})\mathrm{d}F.$$

We can consider a reference facet $\hat{F}$, and a mapping $\boldsymbol{\Phi}_F : \hat{F} \to F$ from the reference to the physical space. Let us represent the Jacobian of the geometrical mapping with $\boldsymbol{J}_F \doteq \frac{\partial\boldsymbol{\Phi}_F}{\partial\hat{\boldsymbol{x}}}$, which has values in $\mathbb{R}^{(d-1)\times d}$. We can rewrite the facet integral in the reference facet, and next consider a quadrature rule Q on $\hat{F}$ defined by a set of points/weights $(\hat{\boldsymbol{x}}_{\mathrm{gp}}, \mathrm{w}_{\mathrm{gp}})$, as follows:

$$\begin{aligned}
\left(\mathbf{A}^F_{K^+K^-}\right)_{ab} &= \int_F \boldsymbol{\mathcal{F}}(\boldsymbol{x})\mathrm{d}\Omega = \int_{\hat{F}} \boldsymbol{\mathcal{F}}\circ\boldsymbol{\Phi}_F(\boldsymbol{x})|\boldsymbol{J}_F|\mathrm{d}F\\
&= \sum_{\hat{\boldsymbol{x}}_{\mathrm{gp}}\in\mathrm{Q}} \boldsymbol{\mathcal{F}}\circ\boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}})\mathrm{w}(\hat{\boldsymbol{x}}_{\mathrm{gp}})|\boldsymbol{J}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}})|.
\end{aligned} \tag{23}$$

$|\boldsymbol{J}_F|$ is defined as:

$$|\boldsymbol{J}_F| = \left\|\frac{\mathrm{d}\boldsymbol{\Phi}_F}{\mathrm{d}x}\right\|_2 \quad\text{and}\quad |\boldsymbol{J}_F| = \left\|\frac{\partial\boldsymbol{\Phi}^1_F}{\partial\hat{\boldsymbol{x}}}\times\frac{\partial\boldsymbol{\Phi}^2_F}{\partial\hat{\boldsymbol{x}}}\right\|_2, \tag{24}$$

for $d = 2, 3$, respectively.

The expression of the shape functions and their gradients in the physical space in terms of the ones in the reference space are computed by using the cell-wise maps. Thus, two mappings $\boldsymbol{\Phi}_{K^+}$ and $\boldsymbol{\Phi}_{K^-}$ among the reference cell $\hat{K}$ and the cells K^+ and K^- in physical space, respectively, are involved in the numerical evaluation of interior facet integrals. We can also consider the reference facet $\hat{F}$ and a map $\boldsymbol{\Phi}_F$ from this reference facet to F (analogously as $\boldsymbol{\Phi}_K$ and K but in one dimension less in the reference space). We can define a quadrature rule $(\hat{\boldsymbol{x}}_{\mathrm{gp}}, \mathrm{w}_{\mathrm{gp}})$ in $\hat{F}$. We can also define the reference facet $\hat{F}^{\pm}$ of $\hat{K}$ such that $\boldsymbol{\Phi}_{K^{\pm}}(\hat{F}^{\pm}) = F$, and the map $\boldsymbol{\Phi}_{\hat{F}^{\pm}}$ from $\hat{F}$ to $\hat{F}^{\pm}$. With this map, we can define the quadrature $(\hat{\boldsymbol{x}}^{\pm}_{\mathrm{gp}} \doteq \boldsymbol{\Phi}_{\hat{F}^{\pm}}(\hat{\boldsymbol{x}}_{\mathrm{gp}}), \mathrm{w}_{\mathrm{gp}})$ with respect to the reference cell $\hat{K}$.

However, the same facet F has (in general) a different orientation depending on the cell used as reference, and so, a different index might be assigned to the same facet quadrature points from the perspective of either cell, i.e., $\boldsymbol{\Phi}_{K^+}(\hat{\boldsymbol{x}}^{+}_{\mathrm{gp}}) \neq \boldsymbol{\Phi}_{K^-}(\hat{\boldsymbol{x}}^{-}_{\mathrm{gp}})$ in general. We adopt the convention that facet quadrature points identifiers are in the local numbering space of K^+, and these local identifiers are translated into the local numbering space of K^-. This is represented by the permutation $\Pi(\mathrm{gp})$ such that

$$\boldsymbol{\Phi}_{K^-}(\hat{\boldsymbol{x}}^{-}_{\Pi(\mathrm{gp})}) = \boldsymbol{\Phi}_{K^+}(\hat{\boldsymbol{x}}^{+}_{\mathrm{gp}}) = \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}}).$$

The logic underlying this translation is equivalent to the one discussed in Sect. 3.16; see Fig. 2 for an explanatory illustration. As a result, we have

$$\boldsymbol{\Phi}^{-1}_{K^+} \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}}) = \hat{\boldsymbol{x}}^{+}_{\mathrm{gp}}, \qquad \text{and} \qquad \boldsymbol{\Phi}^{-1}_{K^-} \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}}) = \hat{\boldsymbol{x}}^{-}_{\Pi(\mathrm{gp})}.$$

Let us consider the evaluation of zero and first order derivatives on facets, i.e., the evaluation of $\phi^b_{K^\alpha} \circ \boldsymbol{\Phi}_{K^\alpha}(\hat{\boldsymbol{x}}_{\mathrm{gp}})$ and $\nabla\phi^b_{K^\alpha} \circ \boldsymbol{\Phi}_{K^\alpha}(\hat{\boldsymbol{x}}_{\mathrm{gp}})$ for $\alpha \in \{+,-\}$, where the quadrature points belong to a quadrature in the reference facet $\hat{F}$. We note that the introduction of $\hat{\Psi}$ is not needed for non-conforming methods, since there is no continuity to be enforced, and we will consider it to be the identity operator for simplicity. The values of the shape functions (times the geometrical mapping) on the facet quadrature points is evaluated as follows:

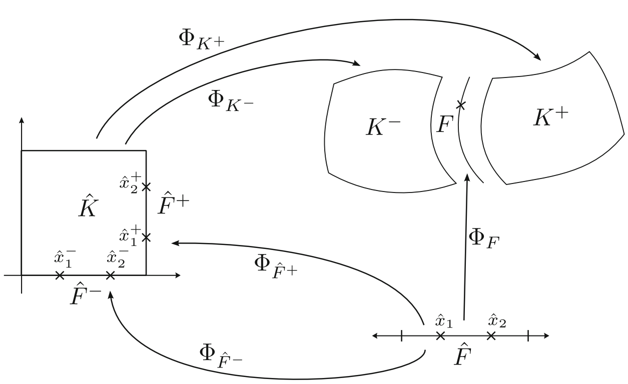


Fig. 2 Mappings required for facet integration. The (only) quadrature point shown in the physical space is located at $\boldsymbol{x} = \boldsymbol{\Phi}_F(\hat{x}_1) = \boldsymbol{\Phi}_{K^+}(\hat{x}^+_1) = \boldsymbol{\Phi}_{K^-}(\hat{x}^-_2)$, that is, $\Pi(1) = 2$ in this case

$$\phi^b_{K^\alpha} \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}}) = \hat{\phi}^b \circ \boldsymbol{\Phi}^{-1}_{K^\alpha} \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}}),$$

whereas shape function gradients are computed as:

$$\nabla\phi^b_{K^\alpha} \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}}) = \nabla_{\hat{\boldsymbol{x}}}\hat{\phi}^b(\hat{\boldsymbol{x}}_{\mathrm{gp}})\boldsymbol{J}^{-1}_K \circ \boldsymbol{\Phi}^{-1}_{K^\alpha} \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}}).$$

Without loss of generality, let us focus on the first integral in (22). Replacing the mean value and jump operators by their definition in (21), and taking into account that $\phi^b_{K^-}$ and $\nabla\phi^a_{K^+}$ vanish on K^+ and K^- (by construction of $\mathcal{X}_h$ and $\mathcal{Y}_h$), respectively, we end up with the following integral to be computed numerically:

$$-\frac{1}{2}\int_F \phi^b_{K^-}\boldsymbol{n}^- \cdot \nabla\phi^a_{K^+}\mathrm{d}F \ .$$

This integral is first mapped back to the reference facet $\hat{F} \subset \mathbb{R}^{\mathrm{d}-1}$, and then it is approximated by the following sum over quadrature points:

$$\begin{aligned}
-\frac{1}{2}\int_F & (\phi^b_{K^-}\boldsymbol{n}^-) \cdot \nabla\phi^a_{K^+}\mathrm{d}F \\
&= -\frac{1}{2}\int_{\hat{F}} \hat{\phi}^b_{K^-} \circ \boldsymbol{\Phi}^{-1}_{K^-} \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}})\boldsymbol{n}^- \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}}) \\
&\quad \cdot \nabla_{\hat{\boldsymbol{x}}}\hat{\phi}^b(\hat{\boldsymbol{x}}_{\mathrm{gp}})\boldsymbol{J}^{-1}_K \circ \boldsymbol{\Phi}^{-1}_{K^+} \circ \boldsymbol{\Phi}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}})|\boldsymbol{J}_F|\mathrm{d}\hat{F} \\
&\approx -\frac{1}{2}\sum_{gp\in\mathrm{Q}} \hat{\phi}^b_{K^-}(\hat{\boldsymbol{x}}^{-}_{\Pi(\mathrm{gp})})\boldsymbol{n}^-(\boldsymbol{x}_{\mathrm{gp}}) \\
&\quad \cdot \nabla_{\hat{\boldsymbol{x}}}\hat{\phi}^b(\hat{\boldsymbol{x}}^{+}_{\mathrm{gp}})\boldsymbol{J}^{-1}_K(\hat{\boldsymbol{x}}^{+}_q)|\boldsymbol{J}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}})|\mathrm{w}_{\mathrm{gp}}.
\end{aligned} \tag{25}$$

Using these ideas, we can compute all the terms related to facet integrals. Furthermore, outward normals on facets can be computed as:

$$\boldsymbol{n}^\alpha = (-1)^{o_\alpha}\frac{\frac{\mathrm{d}\boldsymbol{\Phi}_F}{\mathrm{d}x}}{\left\|\frac{\mathrm{d}\boldsymbol{\Phi}_F}{\mathrm{d}x}\right\|_2} \quad \text{and} \quad \boldsymbol{n}^\alpha = (-1)^{o_\alpha}\frac{\frac{\partial\boldsymbol{\Phi}^1_F}{\partial\hat{x}} \times \frac{\partial\boldsymbol{\Phi}^2_F}{\partial\hat{x}}}{\left\|\frac{\partial\boldsymbol{\Phi}^1_F}{\partial\hat{x}} \times \frac{\partial\boldsymbol{\Phi}^2_F}{\partial\hat{x}}\right\|_2}, \tag{26}$$

for $d = 2, 3$, respectively, and $\alpha \in \{+,-\}$; o is 0 or 1 and is used to enforce the normal to be outwards. Tangent vector(s) for a given facet can be easily computed out of the normal vector.

3.14 Polytopes

One of the motivations of FEMPAR is to develop a framework that can deal with arbitrary space dimensions. It permits to readily implement space-time formulations, which are posed in 4D. Other higher-dimensional applications include systems of PDEs posed in the phase space, e.g., the 7D (including time) Vlasov-Maxwell equations for the simulation of plasmas.

In this section, we provide the mathematical abstraction of cell topologies based on the concept of *polytope*. This abstract concept is of practical importance, because it allows us to develop algorithms and codes that can be applied to any topology that fits into the framework. The framework developed herein is very general and includes triangles and quadrilaterals in 2D, and tetrahedra, hexahedra, prysms, and pyramids in 3D. Furthermore, it can also be extended to arbitrary dimensions, to define not only n-cubes and n-simplices but many other topologies. A polytope is mathematically defined as the convex hull of a finite set of points. As a consequence, a polytope is a polyhedron. In the frame of `FEMPAR`, we consider polytopes that can be expressed as the image of the composition of two operators. The definition of topologies for reference FEs based on this idea can be found in [25].

The main topological information consumed by FE codes is the description of the d-dim polytope boundary as the assemble of $(d-1)$-dim polytopes, proceeding recursively till 0-dim objects are obtained (vertices); we use the contraction *k-dim* object to say object of dimension k. These lower-dimensional entities describing the polytope boundary are denoted herein as *n-faces*. Usually, the nomenclature used to describe n-faces in FEs is restricted to 3D problems. In `FEMPAR` and in the following exposition, we use a dimension-independent nomenclature in order to accommodate higher-dimensional problems. We consider the space dimension $d \in \mathbb{N}^+$ and a d-dim polytope. We define the d-face as the polytope itself. The set of $(d-1)$-dim polygons that compose the boundary of the polytope are its $(d-1)$-faces; $(d-1)$-faces are usually denoted as *facets*. We can proceed recursively, i.e., defining the $(k-1)$-faces of the polytope as the set of facets of its k-faces till reaching 0-faces. In 3D, 3-faces are called *cells*, 2-dim faces are *faces*, 1-dim faces are *edges*, and 0-dim faces are *vertices*. Herein, we use the term n-faces to denote all these objects. In this work, we denote by *vefs* the set of n-faces of dimension lower than the space dimension, e.g., it only includes vertices, edges, and faces in 3D.

Let us introduce some notation. We represent the set of bitmaps of size m with $\mathbb{B}^m$. The bitmaps $(1, 1, \ldots, 1)$ and $(0, 0, \ldots, 0)$ are represented with `1` and `0`, respectively. Given a domain $\square \subset \mathbb{R}^d$ we use the notation $\alpha\square + \boldsymbol{b}$, $\alpha \in \mathbb{R}$, $\boldsymbol{b} \in \mathbb{R}^d$ to denote the domain $\{\alpha\boldsymbol{x} + \boldsymbol{b} : \boldsymbol{x} \in \square\}$. $\boldsymbol{e}_j$ represents the j-th canonical basis vector of $\mathbb{R}^d$.

Let us define first the *directional extrusion* $\square_{(j;\alpha,\beta)}$ of $\square$ with respect to the direction $\boldsymbol{e}_j$ of type (α, β). α determines the topology of the extrusion, namely a prysm-type extrusion (1) or a pyramid-type extrusion (0) (see also [25]). β determines whether we want to perform the α-extrusion (1) or do-nothing (0). Based on this, we have the following definition.

Definition 3.3 (*Directional extrusion*) Given a domain $\square \subset \mathbb{R}^d$, we define $\square_{(j;\alpha,\beta)} \subset \mathbb{R}^d$, with $\beta, \alpha \in \{0, 1\}$ and $j = 1, \ldots, d$, as

$$\square_{(j;\alpha,0)} \doteq \square, \qquad \square_{(j;0,1)} \doteq \{(1-z)\square + z\boldsymbol{e}_j \,:\, z \in [0,1]\},$$
$$\square_{(j;1,1)} \doteq \{\square + z\boldsymbol{e}_j \,:\, z \in [0,1]\}.$$

The directional extrusion can be used recursively to define polytopes and their n-faces. An n-face is determined by a topology $\mathtt{t} \in \mathbb{B}^d$, an extrusion $\mathtt{e} \in \mathbb{B}^d$, and an anchor vertex $\boldsymbol{v} \in \mathbb{R}^d$, using a recursive procedure as follows. The use of directional extrusions to get different polytopes and n-faces is illustrated in Figs. 3 and 4. One can observe how all the lower dimensional n-faces after directional extrusion lead to one dimension larger n-faces for different values of α.

Definition 3.4 (*n-face*) Given $\mathtt{t}$, $\mathtt{e} \in \mathbb{B}^d$ and $\boldsymbol{v} \in \mathbb{R}^d$, we can define the n-face $\square$ in a recursive way as follows. Let $\square^0 \doteq \{\boldsymbol{v}\}$; we define $\square \doteq \square^d$ based on the following recursion:

$$\square^0 \to \square^1 \doteq \square^0{}_{(1;\mathtt{t}(0),\mathtt{e}(0))} \to \ldots \to \square^{i+1} \doteq \square^i{}_{(i+1;\mathtt{t}(i),\mathtt{e}(i))}$$
$$\to \ldots \to \square^d \doteq \square^{d-1}{}_{(d;\mathtt{t}(d-1),\mathtt{e}(d-1))}. \tag{27}$$

For our purposes, the anchor vertex $\boldsymbol{v}$ has only 0/1 entries, and thus, it can be represented as an element $\mathtt{v}$ of $\mathbb{B}^d$. As a result, an n-face can be uniquely represented with $(\mathtt{t};\mathtt{e},\mathtt{v})$. Based on this definition, we can define a set of d-dim polytopes by recursion. d-dim polytopes are given by $\mathtt{t}$, and represented as n-faces with $(\mathtt{t},\mathtt{1},\mathtt{0})$, i.e., using the origin $\boldsymbol{0}$ as anchor vertex and performing extrusions in all directions. On the other hand, a vertex $\boldsymbol{v}$ (with only 0/1 coordinates) is an n-face with $(\mathtt{t},\mathtt{0},\mathtt{v})$. Some examples of n-face constructions using this procedure can be found in Figs. 3 and 4. Furthermore, in these figures we show all n-faces of the 3-cube and 3-simplex, with all the $\mathtt{e}$ and $\mathtt{v}$ values. In our implementation of polytopes, we use Hasse diagrams based on the composition of extrusion and anchor vertex bitmaps to label the different n-faces of a polytope.

In codes, like in `FEMPAR` , the topology can be coded with the bitmap $\mathtt{t}$ (e.g., one 32-bit integer). `FEMPAR` can use any geometry that can be defined this way, for an arbitrary space dimension. This polytope definition leads to the following geometries. The 1-dim line segment topology is $\mathtt{t} = (0)$ or (1); this ambiguity in 1D is inherited to higher dimensions. In 2D, the triangle topology is $\mathtt{t} = (00)$ (or (01)) and the quadrilateral topology $\mathtt{t} = (10)$ (or (11)). In 3D, cubes are represented by $\mathtt{t} = (1, 1, 0)$ (or (1, 1, 1)),

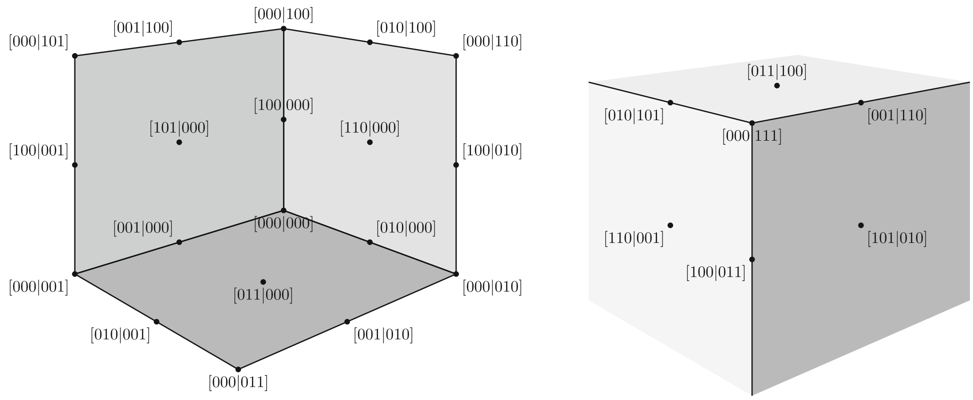


Fig. 3 e and v values for all the n-faces (with the exception of the volume) of the 3-cube, with topology $\mathtt{t} = (111)$

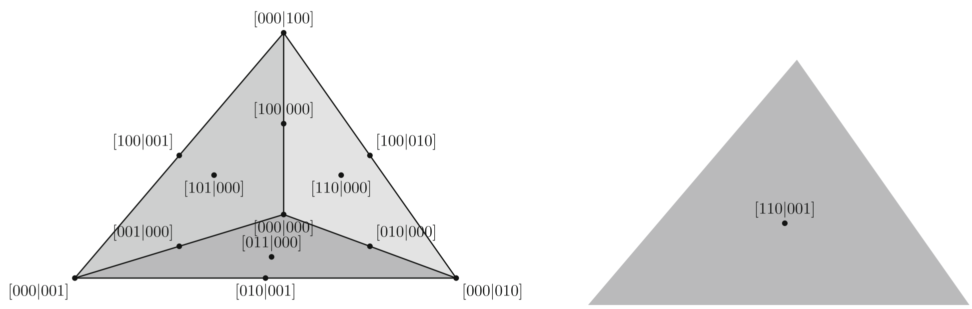


Fig. 4 e and v values for all the n-faces (with the exception of the volume) of the 3-simplex, with topology $\mathtt{t} = (000)$

tetrahedra $\mathtt{t} = (0,0,0)$ (or (0,0,1)), prysms by $\mathtt{t} = (1,0,0)$ (or (1,1,1)), and pyramids by $\mathtt{t} = (0,1,0)$ (or (0,1,1)). Cosserats in 4D are represented by $\mathtt{t} = (1,1,1,0)$ (or (1,1,1,1)). In general, 2^{k-1} types of k-dim topologies are possible. n-cubes are expressed by $\mathtt{t} = \mathtt{1}$ and n-simplices by $\mathtt{t} = \mathtt{0}$.

Given a bitmap $\mathtt{t}$ and a bit α, we define the bit operation that modifies the j bit of $\mathtt{t}$ to α with $\mathtt{t}.o_j(\alpha)$. Given the chain on n-faces (27), let us assume that $\square_{i-1}$ is represented by $(\mathtt{t},\mathtt{e}',\mathtt{v})$. The extrusion $\square_i \doteq \square_{i-1\,(i;*,\alpha)}$ is defined by $(\mathtt{t},\mathtt{e}'.o_{i-1}(\alpha),\mathtt{v})$. Thus, the chain (27) can be represented as follows. Given a topology $\mathtt{t}$, an extrusion $\mathtt{e}$, and an anchor vertex $\mathtt{v}$, we start with $(\mathtt{t},\mathtt{e}',\mathtt{v}) \doteq (\mathtt{t},\mathtt{0},\mathtt{v})$ and proceed recursively:

$$
\begin{aligned}
(\mathtt{t},\mathtt{e}',\mathtt{v}) &\rightarrow (\mathtt{t},\mathtt{e}'.o_0(\mathtt{e}(0)),\mathtt{v}) \rightarrow \ldots \\
&\rightarrow (\mathtt{t},\mathtt{e}'.o_i(\mathtt{e}(i)),\mathtt{v}) \rightarrow \ldots \\
&\rightarrow (\mathtt{t},\mathtt{e}'.o_{d-1}(\mathtt{e}(d-1)),\mathtt{v}) \equiv (\mathtt{t},\mathtt{e},\mathtt{v}).
\end{aligned} \tag{28}
$$

E.g., in 3D, the polytope itself (or 3-face) is determined by $\mathtt{t} = (1,1,1)$ and $(\mathtt{e},\mathtt{v}) = ((1,1,1),(0,0,0))$. The chain (28) in this case reads as follows: (We omit $\mathtt{t}$ in the chain since it is the same for all elements in the recursion.)

$$
\begin{aligned}
&((0,0,0),(0,0,0)) \rightarrow ((0,0,1),(0,0,0)) \\
&\quad \rightarrow ((0,1,1),(0,0,0)) \rightarrow ((1,1,1),(0,0,0)).
\end{aligned}
$$

Using the definition of the n-face, every element of the chain has a geometrical representation. We start with the vertex $\boldsymbol{0}$, next obtain the line segment $\{(x,0,0) : x \in [0,1]\}$, next the square $\{(x,y,0) : x, y \in [0,1]\}$, and finally the unit cube. The previous definition is not only useful to represent d-dim objects *but all its n-faces*. See Figs. 3 and 4.

For a given n-face $\square \equiv (\mathtt{t},\mathtt{e},\mathtt{v})$, we want to define the set $\mathcal{S}_\square$ of all n-faces of $\square$. In order to do so, we introduce the following concepts.

Definition 3.5 (*Oriented set extrusion*) Given a set $\mathcal{S} = \{\square : \square \in \mathbb{R}^d\}$, we define $\mathcal{S}_{(j;\alpha,\beta)}$, with $\beta, \alpha \in \{0,1\}$ and $j = 1,\ldots,d$ as:

$$
\begin{aligned}
&\mathcal{S}_{(j;\alpha,0)} \doteq \mathcal{S}, \qquad \mathcal{S}_{(j;0,1)} \doteq \{\square, \boldsymbol{0} + \boldsymbol{e}_j, \square_{(j;0,1)} : \square \in \mathcal{S}\}, \\
&\qquad \mathcal{S}_{(j;1,1)} \doteq \{\square, \square + \boldsymbol{e}_j, \square_{(j;1,1)} : \square \in \mathcal{S}\}.
\end{aligned}
$$

Definition 3.6 (*Set of n-faces*) Given an n-face $(\mathtt{t},\mathtt{e},\mathtt{v})$, we can obtain all its n-faces recursively as follows. Let $\mathcal{S}^0 \doteq \{\boldsymbol{v}\}$; we define $\mathcal{S} \doteq \mathcal{S}^d$ based on the following recursion:

$$
\begin{aligned}
\mathcal{S}^0 &\rightarrow \mathcal{S}^1 \doteq \mathcal{S}^0{}_{(1;\mathtt{t}(0),\mathtt{e}(0))} \rightarrow \ldots \rightarrow \mathcal{S}^{i+1} \\
&\doteq \mathcal{S}^i{}_{(i+1;\mathtt{t}(i),\mathtt{e}(i))} \rightarrow \ldots \rightarrow \mathcal{S}^d \doteq \mathcal{S}^{d-1}{}_{(d;\mathtt{t}(d-1),\mathtt{e}(d-1))}.
\end{aligned} \tag{29}
$$

All the resulting n-faces can also be written with the $(\mathtt{t}, \mathtt{e}, \mathtt{v})$ notation commented above. In order to define this chain as in (28) (i.e., only based on the bitmap notation), we note the following. Given the n-face $\square \equiv (\mathtt{t}, \mathtt{e}, \mathtt{v})$, the n-face $\square + \boldsymbol{e}_j \equiv (\mathtt{t}, \mathtt{e}, \mathtt{v}.o_j(1))$. With this ingredient, we can implement the generator of all n-faces of an n-face using the bitmap notation.

We also want to know the facets of an n-face. We use the following statement. Given an n-face $\square \equiv (\mathtt{t}, \mathtt{e}, \mathtt{v})$ and its corresponding chain (28), the i-th element boundary $\partial\square^i \doteq \partial\square^{i-1}{}_{(i;\mathtt{t}(i-1),\mathtt{e}(i-1))}$ is the following:

$$\begin{aligned}
&\partial\square^i = \partial\square^{i-1}, \text{ if } \mathtt{e}(i-1) = 0,\\
&\partial\square^i = \{\square^{i-1}, \partial\square^{i-1}{}_{(i;0,1)}\}, \text{ if } \mathtt{t}(i-1) = 0, \mathtt{e}(i-1) = 1,\\
&\partial\square^i = \{\square^{i-1}, \square^{i-1} + \hat{\mathbf{e}}_i, \partial\square^{i-1}{}_{(i;1,1)}\} \text{ if } \mathtt{t}(i-1) = 1, \mathtt{e}(i-1) = 1,
\end{aligned} \tag{30}$$

with $\partial\square^1 = \{\square^0, \square^0 + \hat{\mathbf{e}}_1\}$.

Using this definition of facets for the 3D cube, we get the following faces: ((1, 1, 0); (0, 0, 0)) and ((1, 1, 0); (0, 0, 1)) faces ($x = 0$ and $x = 1$ faces), ((1, 0, 1); (0, 0, 0)) and ((1, 0, 1); (0, 1, 0)) faces ($y = 0$ and $y = 1$ faces), ((0, 1, 1); (0, 0, 0)) and ((0, 1, 1); (1, 0, 0)) faces ($z = 0$ and $z = 1$ faces), having 6 faces in total. For every one of these faces, we can use the same definition above, to obtain the $(d-2)$-faces that are in the boundary of every $(d-1)$-face. All these ideas can be used for any polytope, not only n-cubes. The only difference is the type of extrusion being used in every case.

3.15 Node Generation and Indexing

FE spaces are polynomial spaces, e.g., Lagrangian polynomials. (Let us note that div- and curl-conforming FEs also rely on Lagrangian polynomials for the definition of the pre-bases and the definition of the equivalence classes.) In order to express these polynomials, one must define a set of points (nodes). In the following, we define a node generator for a given order on an arbitrary polytope, using lexicographical notation.[8]

Definition 3.7 (*Set of nodes*) Let us consider a polytope $\square \in \mathbb{R}^d$ represented by $(\mathtt{t}, \mathtt{1}, \mathtt{0})$. Its set $\mathcal{N}^k$ of equidistant Lagrangian nodes of order k, in lexicographical notation, are generated recursively as follows: $\mathcal{N}^k \doteq \mathcal{N}^k_d$, where

$$\begin{aligned}
&\mathcal{N}^p_{m+1} = \{(\boldsymbol{\alpha}, \beta) : \boldsymbol{\alpha} \in \mathcal{N}^{p-\beta(1-\mathtt{t}(m))}_m, \beta \in \mathcal{N}^p_1\},\\
&\quad\text{with } \mathcal{N}^q_1 = \{\alpha \in \mathbb{N}^+ : \alpha \leq q\}.
\end{aligned} \tag{31}$$

Given a node $\boldsymbol{\alpha} \in \mathbb{N}^d$ in lexicographical notation and assuming an equidistant distribution of nodes, its space coordinates $x_{\boldsymbol{\alpha}} \in \mathbb{R}^d$ can readily be obtained, $x_{\boldsymbol{\alpha}} \doteq \boldsymbol{\alpha}/k$. We note that for n-cubes we recover the typical tensor product definition of nodes and the corresponding truncated subset of nodes for n-simplices. Other node generators can also be considered, especially for very high-order elements (e.g., Fekete points).

It is basic in FE analysis to have an *ownership* relation between n-faces and nodes. In particular, it is basic to enforce continuity between FEs by enforcing continuity of nodal values. In order to generate the nodes of the polytope that belong to an n-face, we use the following construction.

First, we generate the local set of nodes, using the definition above, for the n-face. Given a k-face $(\mathtt{t}, \mathtt{e}, \mathtt{v})$ in $\mathbb{R}^d$, we consider the *reference* k-dim polytope $(\mathtt{t}', \mathtt{1}, \mathtt{0})$, where $\mathtt{t}'$ is the restriction of $\mathtt{t}$ to the components that are extruded, i.e., $\mathtt{t}' \doteq \mathtt{t} \circ m_{\ell g}$ with the mapping $m_{\ell g} : \{1, \ldots, k\} \to \{j \in \{1, \ldots, d\} : \mathtt{e}(j) = 1\}$. Next, we define the local nodes of the n-face as the nodes of the reference polytope. It defines the n-faces nodes and their local coordinates. Finally, we define the linear mapping from the reference k-dim polytope to the k-face. The map can be defined with $k + 1$ independent conditions. It can be defined by enforcing that the mapping maps the anchor vertex of the reference polytope to the one of the k-face and the same for the extrusion of the anchor vertex to all directions:

$$\begin{aligned}
&m(\boldsymbol{0}) = \boldsymbol{v}, \qquad m(\boldsymbol{e}_s) = \boldsymbol{e}_{m_{\ell g}(s)}, \text{ if } \mathtt{t}'(s) = 0,\\
&\qquad m(\boldsymbol{e}_s) = \boldsymbol{v} + \boldsymbol{e}_{m_{\ell g}(s)}, \text{ if } \mathtt{t}'(s) = 1.
\end{aligned}$$

Since the mapping is linear, it can be written as:

$$m(\boldsymbol{x}) = \boldsymbol{\alpha}_0 + x_1\boldsymbol{\alpha}_1 + \ldots + x_k\boldsymbol{\alpha}_k.$$

Form the first constraints we get that $\boldsymbol{\alpha}_0 = \boldsymbol{v}$. For the other constraints, we get:

$$m(\boldsymbol{e}_s) = \boldsymbol{v}\mathtt{t}'(s) + \boldsymbol{e}_{m_{\ell g}(s)} = \boldsymbol{v} + \boldsymbol{\alpha}_s \longrightarrow \boldsymbol{\alpha}_s = \boldsymbol{v}(\mathtt{t}'(s) - 1) + \boldsymbol{e}_{m_{\ell g}(s)}.$$

Thus, we get:

$$m(\boldsymbol{x}) = \boldsymbol{v} + \sum_{s=1}^{k} x_s \boldsymbol{v}(\mathtt{t}'(s) - 1) + x_s \boldsymbol{e}_{m_{\ell g}(s)} + x_s, \tag{32}$$

and thus:

[8] We note that in fact the order k is not a scalar but a vector $\boldsymbol{k} \in \mathbb{R}^d$. In principle, the use of a vector-valued order only has sense for n-cubes. The implementation in FEMPAR makes use of a vector-valued order, even though all entries should be the same for polytopes that are not n-cubes. We note that the use of different orders in different directions is basic to define high order Raviart-Thomas and Nédélec elements on n-cubes. In the following presentation, we consider the scalar order case for simplicity.

$$m(\boldsymbol{x})_i = v_i(1 - \sum_{\substack{\{s = 1,\ldots,k : \\ \mathtt{t}'(s) = 0\}}} x_s) + x_{m_{\ell g}^{-1}(i)}.$$

We could also obtain the expression for the inverse of the mapping m analogously. We can readily use the mapping for lexicographical coordinates. As a result, given a k-face, we can define its nodes with a local numbering based on the lexicographical label of the reference k-face. The local-to-global lexicographical label (where global is the label of the d-dim base polytope) is obtained by applying the mapping (32).

3.16 Global DOF Numbering and Conformity

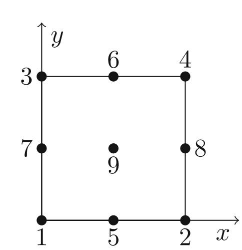
A basic ingredient in FE analysis is the imposition of continuity among FEs in order to build conforming global FE spaces. This process is mathematically defined with equivalence classes on DOFs (see Sect. 3.6). For example, functions in the Lagrangian FE space are related to geometrical nodes, and to impose continuity of a function among FEs is equivalent to impose continuity of nodal values in the same spatial position (see Sect. 3.8). In the following, we provide a mechanism to identify nodes in two different cells that share the same position to implement the required equivalence class. The situation is slightly more involved for div-conforming and curl-conforming FE spaces. In these cases, one can still determine a DOF with a node plus n-face ownership (see Sects. 3.9 and 3.10, respectively). Thus, the equivalence class in these situations can be formulated as in Lagrangian FEs (determine nodes with the same position) at every n-face separately.

Following Sect. 3.6, a node within a cell of our triangulation can be represented as (b, K), where b is the local cell-wise index of the node and K is the cell global index. Given an n-face F of the cell, the same node can be represented with (b', F, K), where b' is an n-face-based local index. For example, node 8 (cell-wise local index) in the cell of Fig. 6 can also be determined as the node 1 (facet-wise local index) of the n-face 8 (see Fig. 5). This facet-wise local index is determined by the coordinate system being used at the n-face. For example, the nodes of n-face 8 in Fig. 6 are ordered as (8, 12) (i.e., first 8 and then 12). On the other hand, node indices are represented with the coordinates in a lexicographical coordinate system, as presented in (31). For example, node with $b = 8$ ($b' = 1$ in n-face 8) is represented with the coordinates $\boldsymbol{s} = (4, 1)$ ($\boldsymbol{s}' = (1)$ in the n-face).

Let us consider an n-face F in our triangulation, two cells K^+ (source cell) and K^- (target cell) sharing the n-face, and nodes $(\boldsymbol{s}'_+, F, K^+)$ and $(\boldsymbol{s}'^-, F, K^-)$ (with n-face-wise local indices). The question that must be answered is: are nodes $(\boldsymbol{s}'_+, F, K^+)$ and $(\boldsymbol{s}'^-, F, K^-)$ in the same spatial position? This question can be answered with the map p_F in (15) that, given the position of the node in the coordinate system of F in K^+, provides the one in K^-.

We note that this mapping is trivial when using structured (possibly locally adapted) n-cube meshes, since the local ordering of nodes in an n-face based on increasing local index leads to the same ordering for all cells containing that n-face; we say that the mesh is *properly oriented* in this case. However, 2D or 3D unstructured mesh generators might not return properly oriented meshes, and thus the FE code has to deal with the explicit construction and application of permutations. We also note that one can always end up with oriented meshes for n-simplices by simple cell-wise permutations (see, e.g., [72, Sect. 5.5] and [80]). After reading n-simplex meshes, these meshes are always properly oriented in FEMPAR before proceeding to any computation. While this is also true for 2D n-cube meshes, 3D n-cube meshes cannot be properly oriented in general [81].

Let us consider the reference polytope $\hat{K}$ associated to K^+ and K^-. In general, the n-face F has a different n-face local index with respect to the two cells; its corresponding



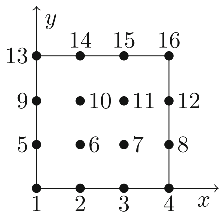
n	9															
p	1	2	3	4	5		7		9		11		13			17
l	1	2	3	4	1	2	3	4	1	3	2	4	1	2	3	4

`vertices_n_face`

n	9															
p	1	1	1	1	1		3		5		7		9			13
l					1	2	3	4	1	3	2	4	5	6	7	8

`facets_n_face`

Fig. 5 Numbering convention for n-faces with $\hat{K}$ a quadrilateral (left) and the status of `vertices_n_face` and `facets_n_face` corresponding to that numbering (right). `n`, `p(n+1)`, and `l(p(n+1)-1)` are private member variables of `type(list_t)` storing the number of entities, the start position in `l(:)` of the list associated to each entity, and the identifiers associated of all lists gathered in a single array, respectively



n	9																
p	1	2	3	4	5		7		9		11		13				17
l	1	4	13	16	2	3	14	15	5	9	8	12	6	7	10	11	

`own_dofs_n_face (conformity==.true.)`

n	9																
p	1	1	1	1	1	1	1	1	1								17
l									1	2	3	4	5	6	...	16	

`own_dofs_n_face (conformity==.false.)`

Fig. 6 Numbering convention for the DOFs of an (scalar-valued) bi-cubic Lagrangian FE on top of a quadrilateral (left) and the status of `own_dofs_n_face` for this `reference_fe_t` in its CG (right, top) and DG forms (right, bottom)

reference n-face is represented with $\hat{F}^{+}$ and $\hat{F}^{-}$ for K^{+} and K^{-}, respectively. In general, the map between nodes of these two n-faces can be defined by using (32), which is invertible (since it is linear and full rank). Using this approach, the map can be generated for arbitrary dimension and polytope topology. However, for the particular case of 2D/3D n-cube meshes, we have implemented this procedure in a more computationally efficient manner. In particular, the required permutations (mappings) are expressed in terms of a set of tables, which are stored and set up (filled) by the so-called `reference_fe_t` abstract data type in `FEMPAR`. We refer to Sect. 6.3 for detailed implementation details. (Recall that n-simplices meshes do not actually require this procedure as they can always be properly (re)oriented.)

Let us consider the case of 3D n-cube meshes. Vertices are trivial because there is only one node and no permutation is needed. For edges and faces, we rely on the three following concepts:

- Rotation index: Provides the local index of the anchor vertex of F^{-} with respect to the coordinate system of F_{+}. When FEs are sharing two edges, we have the following situations. The edge can have the same anchor vertex seen from both elements, or not. For faces, the anchor vertex can be in 4 positions. It is called *rotation* because it represents a map that keeps invariant the reference face $\hat{F}^{-}$ and makes the anchor vertices of the source and target cells coincide.
- Orientation index: Given two cells sharing an n-face with the same anchor vertex, the orientation index codes the map from the coordinate system of the n-face with respect to the first cell to the one with respect to the second one.[9] For edges, this map is always the identity, because two cells sharing an edge with the same anchor node provide the same edge-wise node coordinates to its nodes. For faces, the situation is more complex, because it involves 2 different possible situations. The orientation index is equal to 0 for the identity permutation and 1 when we have to swap indices. We denote the base face as the face with the lowest local index (face [011|000] in Fig. 3). Next, we consider two cubes that share a face, restricted to the following scenario: (1) the face is the base face in at least one of the cubes; (2) the face has the same anchor vertex in the two cubes. It is trivial to compute the orientation index in these cases. The orientation index in the more general case of two cubes sharing a face only restricted to (2), i.e., two arbitrary faces with the same anchor vertex, can be obtained by composition as follows. If two faces have the same orientation index with the base face, they have an orientation index equal to 1, and 0 otherwise.
- Permutation index: An index obtained by composition of the rotation and orientation indices (i.e., it ranges from 1 and 2, and 1 and 8 for edges and faces, respectively), that codifies the final mapping between coordinates of two cells as the composition of a rotation and a orientation map. We note that the composition of all possible rotations and orientations cover all the possible relative positions of cells for a conforming mesh.

4 Implementation of `polytope_t` and Related Data Types

In `FEMPAR`, the reference FE cell geometry is defined by the `polytope_t` data type; see Listing 1. The input needed to define the polytope is the space dimension `num_dimensions` and the topology `t` in the 32-bit integer `topology`.

[9] In the following, one can consider two unit cubes sharing a face. Since all the concepts are logical one does not have to take into account the real shape of the cells in the physical space. On the other hand, we note that the orientation index is invariant to which of the two cells sharing the face we select as first and second cell.

```
type polytope_t
 private
 integer(ip)                :: num_dimensions
 integer(ip)                :: topology
 integer(ip)                :: number_n_faces
 integer(ip), allocatable :: n_face_array(:)
 integer(ip), allocatable :: ijk_to_index(:)
contains
 procedure           :: create                => polytope_create
 procedure           :: create_facet_iterator => polytope_create_facet_iterator
 procedure           :: free                  => polytope_free
 procedure, private :: fill_polytope_chain
 ... ! Rest of getter TBPs that return the values of the member variables or
     ! processed data out of them
end type polytope_t
```

Listing 1. The `polytope_t` data type.

Using the ideas in (27), (28), and (29), we create the set of all n-faces of the polytope $(\mathtt{t}, \mathtt{e}, \mathtt{v})$ in the (private) `fill_polytope_chain` TBP, which is in turn invoked by the (public) `create` TBP. All n-faces of the polytope have the same topology, and can be uniquely determined by a 32-bit integer that represents the composition of $(\mathtt{e}, \mathtt{v})$. We note that the ordering of n-faces based on $(\mathtt{e}, \mathtt{v})$ mixes n-faces of different dimensions and it is non-consecutive in general. Thus, we consider an ordering based first on the n-face dimension, and next by $(\mathtt{e}, \mathtt{v})$. The set of all n-faces generated by the recursion (29) are stored in `n_face_array`, an array of size `number_n_faces`. This array in particular provides the $(\mathtt{e}, \mathtt{v})$ associated to each n-face. The inverse mapping (from $(\mathtt{e}, \mathtt{v})$ to the actual numbering) is stored in the `ijk_to_index` array.

It is also possible to iterate over facets of an n-face, based on (30). The `create_facet_iterator` TBP of `polytope_t` creates a `facet_iterator_t` instance for a given n-face. `facet_iterator_t` is defined in Listing 2. The n-face $(\mathtt{e}, \mathtt{v})$ is stored in `root`, the topology can be extracted from its `polytope` pointer member variable. The iterator over facets is described by two integers, `component` and `coordinate`, using the ideas in (30). The complexity of the traversal over facets is coded in `facet_iterator_next` and `facet_iterator_has_finished`.

```
type facet_iterator_t
 private
 type(polytope_t), pointer :: polytope
 integer(ip)                :: root
 integer(ip)                :: component
 integer(ip)                :: coordinate
contains
 ...
 procedure, non_overridable :: first        => facet_iterator_first
 procedure, non_overridable :: next         => facet_iterator_next
 procedure, non_overridable :: has_finished => facet_iterator_has_finished
 ...
end type facet_iterator_t
```

Listing 2. The `facet_iterator_t` data type.

With regard to the implementation of nodes within FEMPAR, we provide the node_array_t data type to represent the set of nodes defined in (31); see Listing 3. It is constructed from a polytope and the order. It provides a create TBP, where we perform (31) and fill all the resulting nodes in the node_array array member variable. We number the nodes using a consecutive numbering with increasing lexicographical index. The node array provides the lexicographical label in one integer. The inverse is stored in ijk_to_index. The total number of nodes is stored in num_nodes. Finally, the space coordinates of nodes are stored in coordinates.

```
type node_array_t
 private
 type(polytope_t), pointer      :: polytope
 integer(ip)                    :: order(SPACE_DIM)
 integer(ip)                    :: num_nodes
 integer(ip), allocatable       :: node_array(:)
 integer(ip), allocatable       :: ijk_to_index(:)
 integer(ip), allocatable       :: coordinates(:,:)
contains
  procedure                   :: create                    => node_array_create
  procedure                   :: free                      => node_array_free
  procedure                   :: create_node_iterator      => node_array_create_node_iterator
  procedure                   :: get_number_nodes          => node_array_get_number_nodes
  procedure         , private :: fill                      => node_array_fill
  procedure, nopass , private :: fill_permutations         => node_array_fill_permutations
  procedure, nopass , private :: compute_num_rot_and_perm  => node_array_compute_num_rot_and_perm
end type node_array_t
```

Listing 3. The node_array_t data type.

We also provide the node_iterator_t object (see Listing 4), which iterates over the nodes of an n-face (stored in n_face) using (31) and (32). It has a pointer to the node_array of the base polytope. Internally, it goes through the nodes of n_face (using (31)) (the current node being stored in displacement), which can be translated to the base polytope node numbering using (32) (stored in coordinate); the coordinate is computed on demand by calling the TBP node_iterator_current_ijk. The own_boundary logical allows one to iterate over the nodes considering the n-face as an open or closed set. We note that the create TBP of node_array_t relies on node_iterator_t.

```
type node_iterator_t
 private
 type(node_array_t), pointer :: node_array
 logical                     :: own_boundary
 integer(ip)                 :: n_face
 integer(ip)                 :: topology
 integer(ip)                 :: displacement(0:SPACE_DIM-1)
 integer(ip)                 :: coordinate(0:SPACE_DIM-1)
 ...
contains
 ...
end type node_iterator_t
```

Listing 4. The node_iterator_t data type.

5 The polynomial_t Abstraction

In FEMPAR, the definition of shape functions is not hard-coded, as usually done in most FE codes. Such approach has severe limitations: (1) it is not practical for high order discretizations, and the code cannot be written for an arbitrary order; (2) it involves a huge number of code lines with the analytical expression of shape functions for a given set of available orders (see the discussion in [82]); and (3) it does not allow for dimension-independent code. Instead, we consider a framework based on the concepts in Sect. 3.5, in which one considers a pre-basis, defines the moments, and performs a change of basis. The pre-basis is defined using the product of 1D functions (e.g., the Cartesian product), and the 1D function generator is written in terms of the (arbitrary) order. Our machinery for the generation of 1D functions has been restricted for the moment to polynomial functions in one variable, namely Lagrangian polynomials, monomials, and B-splines, but the implementation can be extended to other choices. The product of 1D functions can be a Cartesian product of 1D Lagrange polynomials (or monomials), to define $\mathcal{Q}_k$ spaces on n-cubes, or a reduced combination of monomials to define $\mathcal{P}_k$ spaces on n-simplices.

The definition of the reference FE functional space relies on the polynomial_t data type in Listing 5, which represents a polynomial in one variable, i.e., $p(x) = \sum_{i=0}^{k} a_i x^k$. Thus, a 1D polynomial is defined in terms of its order k and a set of $k+1$ coefficients $\{a_i\}_{i=0}^{k}$, stored in order and the coefficients array, respectively. Different type extensions of polynomial_t have been considered so far, namely lagrange_polynomial_t and monomial_t. The first one generates a Lagrangian polynomial as in Sect. 3.4, in which the coefficients array has in its first order entries the coordinates of the nodes and in the last entry the coefficient $\frac{1}{\Pi_{n\in\mathcal{N}_k\setminus\{m\}}(x_m-x_s)}$ in (7). The monomial_t extension represents x^k where k is its order. It is just a trivial case of polynomial_t for optimization purposes that is uniquely defined by the order (the coefficients array is not needed). We also consider the polynomial_basis_t data type, which is just a set (array) of (polymorphic) polynomials.

Up to this point, we have defined Lagrange polynomials and monomials in one variable. lagrange_polynomial_t and monomial_t also provide the binding generate_basis that generates a Lagrangian and monomial basis of polynomials, for a given order k. The result of this subroutine is a polynomial_basis_t that includes as many polynomials as the polynomial space dimension. In the case of the Lagrangian basis, it implements the basis $\mathcal{L}^k$ in Sect. 3.4, whereas the binding for monomials simply implements $\{x^i\}_{i=0}^{k}$.

The next step is to generate higher dimensional spaces. We consider two types of spaces. The first one is a space that can be generated as the Cartesian product of 1D spaces, implemented in the data type tensor_product_polynomial_space_t. This data type is defined through the number of space dimensions and as many polynomial_basis_t as space dimensions. This data type can be applied to any combination of 1D spaces. e.g., In the case of 1D Lagrange bases (possibly with different order and nodes per dimension), it leads to

```
type, abstract :: polynomial_t
   private
   integer(ip)              :: order
   real(rp), allocatable :: coefficients(:)
 contains
   ...
   procedure(...)         , deferred :: get_values
   procedure(...), nopass, deferred :: generate_basis
end type polynomial_t

type, extends(polynomial_t) :: lagrange_polynomial_t
 contains
   procedure           :: get_values      => lagrange_polynomial_get_values
   procedure, nopass   :: generate_basis  => lagrange_polynomial_generate_basis
end type lagrange_polynomial_t

type, extends(polynomial_t) :: monomial_t
 contains
   procedure           :: get_values      => monomial_get_values
   procedure, nopass   :: generate_basis  => monomial_generate_basis
end type monomial_t

type :: polynomial_basis_t
   private
   class(polynomial_t), allocatable :: polynomials(:)
 contains
   ...
end type polynomial_basis_t

type :: tensor_product_polynomial_space_t
   private
   integer(ip)                   :: num_dimensions
   integer(ip)                   :: num_pols_dim(SPACE_DIM)
   type(polynomial_basis_t)      :: polynomial_1D_basis(SPACE_DIM)
   type(allocatable_array_rp3_t) :: work_shape_data(SPACE_DIM)
 contains
   ...
   procedure :: evaluate => tensor_product_polynomial_space_evaluate
end type tensor_product_polynomial_space_t

type, extends(tensor_product_polynomial_space_t) :: truncated_tensor_product_polynomial_space_t
 contains
   ...
end type truncated_tensor_product_polynomial_space_t
```

Listing 5. The polynomial_t data type and related data types.

the multi-dimensional basis in (8). Thus, with this data type and Lagrangian 1D bases we generate the Lagrangian FE spaces on top of n-cube cells, i.e., the $\mathcal{Q}_k$ space of polynomials.[10]

Furthermore, we also consider the `truncated_tensor_product_polynomial_space_t` extension that generates Lagrangian FE spaces on n-simplices, i.e., the $\mathcal{P}_k$ space of polynomials. In this case, the `generate_basis` TBPs of `monomial_t` should be used to create the monomial 1D bases per direction and the order should also be the same for all directions. Otherwise, the resulting multi-variable function would have no sense. Next, the combination of 1D monomials only involves terms such that $|\boldsymbol{\alpha}| \leq k$ (see Sect. 3.4), to generate a pre-basis for FE spaces on tetrahedra, i.e., the $\mathcal{P}_k$ space of polynomials.

We note that with these abstract representations of polynomial spaces one can define the reference FE local space. However, unless one considers 1D Lagrangian basis and tensor product polynomials on n-cubes, the resulting basis is not the shape functions basis. Even in the case of Lagrangian n-simplices, a change-of-basis is needed, using the procedure in Sect. 3.5 taking nodal values as moments. In Sect. 9.5, we show how we can define the shape function basis for the case of div-conforming FEs of arbitrary order. The same ideas apply for grad-conforming Lagrangian FEs on n-simplices and curl-conforming FEs in general, but are not included for the sake of brevity.

6 The `reference_fe_t` Abstraction

In this section, we introduce the `reference_fe_t` data type. This data type is the OO representation of the standard mathematical definition of a reference FE presented in Sect. 3.3, namely, a reference cell geometry $\hat{K}$, a functional space $\hat{\mathcal{V}}$, and a set of DOFs $\hat{\Sigma}$ defined on top of it. The `reference_fe_t` is a central abstraction in a FE library and must be judiciously designed to be extensible and reusable. In particular, it must not only accommodate Lagrangian FEs, but also other (more involved/general) spaces like Raviart-Thomas or edge FEs, DG methods, and B-spline patches. An extensible and reusable design of `reference_fe_t` should allow one to, e.g., easily incorporate new local functional spaces that were not originally considered, and to do so without having to rewrite (and thus recompile) any code that is grounded on the set of methods provided by `reference_fe_t`. To this end, in FEMPAR, `reference_fe_t` is an *abstract* data type that serves as a template equipped with a set of member variables and deferred bindings that subclasses have to set up and implement (i.e., override), respectively, in order to complete the description of the concrete FE space at hand. The definition of the `reference_fe_t` data type, a classification of its member variables into three different categories (corresponding to the three ingredients in Ciarlet's definition), and an enumeration of its most relevant regular and deferred bindings, are shown in Listing 6.

[10] Analogously, one could generate serendipity elements only by changing the generation of the multi-dimensional space in terms of 1D ones.

```
type, abstract :: reference_fe_t
  private
  ! Member variables of reference_fe_t can be grouped into three different categories:
  ! 1. Description of topology of the reference cell K̂ (see Section 6.1)
  integer(ip)                              :: num_dimensions
  character(:)                 , allocatable :: topology
  integer(ip)                              :: num_n_faces
  integer(ip)                              :: ptr_n_faces_x_dim(SPACE_DIM+2)
  type(list_t)                             :: vertices_n_face
  type(list_t)                             :: facets_n_face

  ! 2. Description of finite element space on top of V̂ (see Section 6.2)
  character(:)                 , allocatable :: fe_type
  integer(ip)                              :: order
  character(:)                 , allocatable :: field_type
  integer(ip)                              :: num_shape_functions

  ! 3. Description of the set degrees of freedom Σ̂ (see Section 6.3)
  logical                                  :: conformity
  logical                                  :: continuity
  type(list_t)                             :: dofs_n_face
  type(list_t)                             :: own_dofs_n_face
  type(allocatable_array_ip2_t), allocatable :: own_dofs_permutations(:)
contains
  public
  ! ****Regular bindings****
  procedure, non_overridable :: get_num_dimensions
  procedure, non_overridable :: get_topology
  ... ! Rest of getter TBPs that return either the values of the member variables
      ! themselves, a reference (pointer) to them (to avoid the overhead associated
      ! to copies), or other processed data out of them
  procedure                  :: permute_dof_lid_n_face                 ! See Section 6.3

  ! ****Deferred bindings****
  procedure(...), deferred   :: create                                 ! See Section 6.4

  ! Cell-integration related bindings
  procedure(...), deferred   :: create_quadrature                      ! See Section 8.1
  procedure(...), deferred   :: create_interpolation                   ! See Section 8.2
  procedure(...), deferred   :: apply_cell_map                         ! See Section 8.4
  procedure(...), deferred   :: get_default_quadrature_degree          ! See Section 10

  ! Facet-integration related bindings
  procedure(...), deferred   :: create_facet_quadrature                 ! See Section 9.1
  procedure(...), deferred   :: create_facet_interpolation              ! See Section 9.2
  procedure(...), deferred   :: create_interpolation_restricted_to_facet ! See Section 9.3
  procedure(...), deferred   :: fill_qpoints_perm                       ! See Section 9.3

  ! Extract data out of interpolation_t into user-friendly data structures
  ! (See Section 8.4 and 9)
  procedure(...), deferred :: get_values_scalar => cell_integrator_get_values_scalar
  procedure(...), deferred :: get_values_vector => cell_integrator_get_values_vector
  procedure(...), deferred :: get_values_tensor => cell_integrator_get_values_tensor
  generic                  :: get_values => get_values_scalar, get_values_vector, get_values_tensor

  ... ! Extract shape functions gradients out of interpolation_t
  ... ! Extract shape functions hessians out of interpolation_t
  ... ! Extract shape functions curls out of interpolation_t
  ... ! Extract shape functions divergences out of interpolation_t

  ! Restriction of global FE functions to cells and facets
  ! (See Section 10.5)
  procedure(...), deferred :: evaluate_fe_function_scalar
  procedure(...), deferred :: evaluate_fe_function_vector
  procedure(...), deferred :: evaluate_fe_function_tensor
  generic                  :: evaluate_fe_function_ => evaluate_fe_function_scalar, ...vector, ...
      tensor

  ... ! Variants of the above three deferred bindings for gradients, hessians, etc.

  ! Generation of a global DOF numbering
  procedure(...), deferred   :: fill_own_dofs_permutations              ! See Section 6.3
end type reference_fe_t
```

Listing 6. The `reference_fe_t` abstract type, a classification of its member variables, and an enumeration of its most relevant regular and deferred bindings.

This section is structured as follows. The member variables in each of the three aforementioned categories are covered in detail in Sects. 6.1–6.3, respectively. In Sect. 6.4, we discuss the OO design pattern chosen in FEMPAR for the creation of `reference_fe_t` polymorphic instances, and describe the arguments that uniquely define a subclass of this data type; these are in line with its mathematical definition. In Sect. 6.5, we enumerate and briefly describe the subclasses of `reference_fe_t` currently available in FEMPAR. We note that the section is not self-contained as most of the deferred bindings of `reference_fe_t` are not covered here. These involve interactions with other data types in our OO design, and will be described in the sections in which these interactions are exposed. Code comments in Listing 6 serve as a table of contents with the article sections in which these deferred bindings are covered.

see Fig. 5 for an illustration of these member variables and the data type `list_t` used in FEMPAR to store and traverse lists.

The FEMPAR data type `list_t` stores a set of (variable-sized) lists of integer identifiers, one per each entity; in this particular scenario, entities are n-faces. As shown in Fig. 5, the current implementation of this data type uses a compressed storage layout as, e.g., in compressed storage formats for sparse graphs. In order to preserve encapsulation and data hiding, `list_t` offers a rich set of TBPs that lets users to set up (step by a step) a new `list_t` instance; this type also provides a `list_iterator_t` type that lets them to sequentially read/write each of the integer identifiers of the list associated to an entity. The code snippet in Listing 7 illustrates how to iterate and print the identifiers of those vertices belonging to the n-face with identifier `n_face_lid`.

```
subroutine print_vertex_lids_in_n_face(reference_fe, n_face_lid)
  class(reference_fe_t), intent(in) :: reference_fe
  integer(ip)          , intent(in) :: n_face_lid
  type(list_iterator_t) :: iterator
  iterator = reference%create_vertices_n_face_iterator(n_face_lid)
  do while (.not. iterator%has_finished())
     write(*,*) 'vertex␣LID␣(within␣cell)␣', iterator%current(), &
                'in␣n_face␣LID␣(within␣cell)␣', n_face_lid
     call iterator%next()
  end do
end subroutine print_vertex_lids_in_n_face
```

Listing 7. User-level code that illustrates how to print to screen those (local within cell) vertex identifiers belonging to n-face with (local within cell) identifier `n_face_lid`.

6.1 The Reference Cell Topology

The reference cell $\hat{K}$ is a polytope. Therefore, following Sect. 3.14, it can be described with the topology, coded as a set of d bits, where d is the dimension of the polytope. The reference cell topology is generated using `polytope_t` described in Sect. 4, which offers methods like composition and local numbering of n-faces. Polytope topologies include triangles and quadrilaterals in 2D, and tetrahedra, hexahedra, prysms, and pyramids in 3D. The member variables in charge of the description of the reference cell topology $\hat{K}$ are shown in Lines 5–10 of Listing 6. The user must provide the topology and dimension of the polytope to define $\hat{K}$, stored in the member variables `topology` and `num_dimensions`, respectively. A set of *getters* return this basic information, and other related data that can be generated out of them, e.g., the number of n-faces in the boundary of the cell is stored in the `num_n_faces` member variable. The list of vertex identifiers per each n-face and the list of facets (of dimension $n-1$) per each n-face are stored in `vertices_n_face` and `facets_n_face`, respectively;

The number of n-faces of any dimension can be easily computed from `ptr_n_faces_x_dim`. We note that `ptr_n_faces_x_dim` is not a `list_t` instance, since we adopt the convention that n-faces are numbered from the lowest to highest dimension, and thus only the `p` array of the list is actually needed (see Fig. 5). In the example in Fig. 5, the value of this array is $\{1, 5, 9, 10\}$, since we have 4 vertices (dimension 0), 4 facets or edges (dimension 1), and 1 cell (dimension 2).

6.2 The Reference FE Space

For a given cell topology, different definitions of functional spaces and sets of DOFs are possible, e.g., the ones of the nodal Lagrangian grad-conforming reference FE in Sect. 3.8, the Raviart-Thomas div-conforming reference FE in Sect. 3.9, or the curl-conforming Nédélec reference FE in Sect. 3.10. The member variables of `reference_fe_t` required to describe the functional space $\hat{\mathcal{V}}$ with support on $\hat{K}$ are encompassed within Lines 13–16 of Listing 6.

The local FE space $\hat{\mathcal{V}}$ is determined by the member variables fe_type, (in some cases) field_type, and order. fe_type uniquely identifies the concrete FE space at hand. Possible values are provided by means of the public parameter constants fe_type_lagrangian, fe_type_raviart_thomas, and fe_type_nedelec corresponding to the reference_fe_t implementors currently supported in FEMPAR; see Sect. 6.5 for additional details on those. field_type identifies the "type" of physical field being discretized, i.e., whether it is scalar, vector-valued, etc. There are FE spaces that are inherently vector-valued such as, e.g., Raviart-Thomas and edge FEs. However, Lagrangian FEs can be either used to discretize scalar, vector, or tensor-valued fields, and field_type must be provided. We assume that $\hat{\mathcal{V}}$ can be parameterized with respect to an order, which is stored in order. Out of these values, we can generate additional data, e.g., the number of shape functions is stored in num_shape_functions. For example, for (scalar-valued) bi-quadratic (2D) and tri-quadratic (3D) Lagrangian FEs, the field_type is scalar, num_components is equal to 1, order is equal to 2, and num_shape_functions is equal to 9 and 27, respectively.

6.3 The Set of Local DOFs

Additional data is required to describe the set of DOFs $\hat{\Sigma}$ for $\hat{\mathcal{V}}$. In particular, the member variables encompassed within Lines 19–23 of Listing 6 serve this purpose.

The conformity member variable determines whether the global FE space $\mathcal{X}_h$ is conforming with respect to the infinite-dimensional space $\mathcal{X}$, i.e., whether $\mathcal{X}_h \subset \mathcal{X}$ or not. It is used to describe the n-face that owns every DOF, which is required to enforce conformity of the global FE space through equivalence classes (see Sect. 3). e.g., For Lagrangian FEs, setting it to .true. results in a grad-conforming global FE space, whereas setting it to .false. it results in a discontinuous space for DG methods. It is conceptually possible to set it to .true. on some cells and false on others, leading to the CDG method in [83]. On the other hand, the continuity member variable is only determined by $\mathcal{X}$, and tells us whether $\mathcal{X}$ admits a trace operator. Roughly speaking, it tells us whether we must enforce some type of continuity at the discrete level to preserve conformity, e.g., full, tangential, or normal traces for $H^1(\Omega)$, $H(\mathbf{curl},\Omega)$, and $H(\mathrm{div},\Omega)$, respectively. The value of continuity is .false. when $\mathcal{X} = L^2(\Omega)$, since no continuity is required. When continuity is .false., conformity must be .true.. continuity is barely used (see discussion in next paragraph).

The value of conformity is used to generate the own_dofs_n_face member variable of type list_t. This member variable stores, for every n-face, the DOFs it owns; see Fig. 6. For CG methods, the notion of ownership is related to the geometrical location. For DG FEs, although node functionals are still geometrically located on the boundary of the cell, they are nevertheless owned by the cell, and considered as interior DOFs, since there is no global conformity to be enforced. This array is heavily used to generate the global DOF numbering.[11] On the other hand, the dofs_n_face member variable, determines, for a given n-face, the set of DOFs such that their respective shape functions are non-zero on the n-face. The continuity member variable is (currently) only used for DG methods in parallel distributed-memory environments. In particular, in order to decide whether to associate or not a global DOF identifier to nodes on the interface facets of ghost cells (and thus to be able to define non-singular sub-assembled matrices for the DD methods in [84] for DG discretizations). The dofs_n_face member variable is used when continuity is .true. and a global DOF numbering is to be generated, and also might be used by triangulation subclasses (see Sect. 7) in order to extract the coordinates of those nodes on top of a vertex, edge, or face (using the dofs_n_face member variable of the reference_fe_t instance that describes the geometry of the cell). For example, in Fig. 6, the list corresponding to n-face with identifier 8 in dofs_n_face is {4,8,12,16}.

The reference_fe_t data type plays a crucial role in the algorithm in charge of assigning global DOF identifiers to node functionals distributed over the interior of the triangulation cells and their boundary n-faces. (This algorithm, which is is covered in detail in Sect. 10, is grounded on the notion of equivalence classes introduced in Sect. 3.) In particular, the function-like (regular) binding referred to as permute_dof_lid_n_face (see Line 32 of Listing 6) implements the mapping p_F in (15). This function takes as input the so-called permutation index in Sect. 3.16, the local index of a node within an n-face of given dimension (e.g., in 3D, either 0 for vertices, 1 for edges, and 2 for faces) from the perspective of a source cell, and returns the local index of a node within that n-face from the perspective of the target cell.[12] This is in

[11] We can consider three levels of DOF numbering: the cell-wise DOF numbering (referred to as local DOFs), the subdomain-wise DOF numbering (referred to as global DOFs), and a full domain global DOFs. The latter numbering is never created/required in FEMPAR. In serial environments, the latter two match.

[12] We note that the responsibility of determining the permutation index does not lay on reference_fe_t, but on the abstraction of FEMPAR that represents the mesh of the computational domain; see Sect. 7.

particular the transformation that we have to apply when global DOF identifiers have been already assigned to n-face nodes in the source cell, and we want to transfer them to n-face nodes in the target cell; see Sect. 10.3. This binding, implemented in `reference_fe_t`, ultimately relies on its `own_dof_permutations(:)` member variable; see Line 23 in Listing 6. This allocatable array is indexed with the n-face dimension (i.e., 1 for edges, and 2 for faces). For each n-face dimension larger than 0, it contains a rank-2 allocatable array (i.e., `type(allocatable_array_ip2_t)` is the base type of the array), of the polymorphic instance to be returned at runtime based on the values of its dummy arguments `topology` and `fe_type`. (For example, assuming the topology of an hexahedron and `fe_type_lagrangian`, then it will select its dynamic type to be `hex_lagrangian_reference_fe_t`, i.e., the concrete data type implementing Lagrangian-type FE spaces on top of n-cubes.) Before returning, it calls a deferred binding of `reference_fe_t`, called `create`, which is responsible to leave the `reference_fe_t` in a fully functional state. The interface of this deferred binding is shown in Listing 8.

```
abstract interface
 subroutine reference_fe_create(this, topology, number_dimensions, order, &
      &                    field_type, conformity, continuity)
  class(reference_fe_t), intent(inout) :: this
  integer(ip)          , intent(in)    :: number_dimensions
  character(*)         , intent(in)    :: topology
  integer(ip)          , intent(in)    :: order
  character(*)         , intent(in)    :: field_type
  logical              , intent(in)    :: conformity
  logical              , intent(in)    :: continuity
 end subroutine reference_fe_create
...
end interface
```

Listing 8. The signature of the `create` binding of `reference_fe_t`.

which serves as a lookup table for the implementation of the aforementioned transformation. In particular, the rows are indexed with the local index of the node identifier on top of the n-face from the perspective of the source cell, and the columns with the permutation index; see Sect. 3.16. The entry in the corresponding row and column of the table provides the local index of the node within the n-face from the perspective of the target cell. These lookup tables are filled within the `fill_own_dofs_permutations` deferred binding of `reference_fe_t`. We note that this latter binding, and `permute_dof_lid_n_face`, are declared as overridable bindings in Listing 6 on purpose. This lets, e.g., subclasses of `reference_fe_t` to be used in conjunction with (*properly oriented*; see Sect. 3.16) n-simplex meshes to implement the former such that the `own_dof_permutations(:)` member variable is not allocated nor filled, and the latter such that always returns the identity transformation.

6.4 Creating `reference_fe_t` Polymorphic Instances

Central to any OO software system relying on abstract data types is the approach chosen to create polymorphic instances at runtime. For simplicity, `FEMPAR` follows the so-called simple factory design pattern [85]. It takes the form of a single stand-alone function, called `make_reference_fe`, which selects the dynamic type

We remark that `field_type` is only a free parameter for Lagrangian FEs (i.e., for a particular `reference_fe_t` subclass). In other words, it must be `field_type_vector` for Raviart-Thomas and edge elements. We note that despite its fix set of dummy arguments interface, it has been proven to be sufficient to fully describe all subclasses currently available in `FEMPAR`; see Sect. 6.5. However, in the event that it is needed, and with extensibility in mind, a single parameter dictionary of $<key, value>$ pairs might have been used instead; `FEMPAR` indeed relies on an implementation of this data type where *key* is a string (typically denoting the name of the parameter), and *value* a scalar or arbitrary rank array of intrinsic or even user-defined types.[13]

6.5 Enumeration of `reference_fe_t` Subclasses

There is a rather complex data type hierarchy rooted at `reference_fe_t` in `FEMPAR`, which has been judiciously designed with code re-use as the main driver. (For example, Lagrangian FE spaces on top of n-cubes and n-simplices share member variables and code that can be gathered into a common base data type.) For the sake of brevity, in this work we do not cover in full detail the implementation of the data types in this hierarchy (except those details given in Sects. 5 and 9.5). However, for

[13] This data type is implemented within the `FPL` software package [86].

completeness, it is convenient to enumerate those `reference_fe_t` subclasses that, at present, are available in this hierarchy. These subclasses, which lay at the leaves of the hierarchy, are the following ones:

- `hex` and `tet_lagrangian_reference_fe_t`. Space of polynomials of arbitrary degree k on top of n-cubes (i.e., tensor-product like spaces $\mathcal{Q}_k$) and n-simplices (i.e., $\mathcal{P}_k$), respectively, for the discretization of either scalar-valued, vector-valued or tensor-valued fields; see Sect. 3.8. By selecting the ownership relationship among node functionals and n-faces appropriately (see Sect. 6.3), this FE space can be either globally continuous, or entirely discontinuous across cell boundaries.
- `hex` and `tet_raviart_thomas_reference_fe_t`. The vector-valued Raviart-Thomas FE of arbitrary degree k on top of n-cubes, and n-simplices, resp., suitable for the mixed Laplacian problem and some fluid flow problems. Global FE functions of this space (in its conformal variant) have continuous normal components across cell faces; see Sect. 3.9 for details.
- `hex` and `tet_nedelec_reference_fe_t`. The vector-valued curl-conforming Nédélec FE of arbitrary degree k on top of n-cubes, and n-simplices, resp., suitable for electromagnetic problems. Global FE functions of this space (in its conformal variant) have continuous tangential components across cell faces; see Sect. 3.9 for details.
- `void_reference_fe_t`. A software artifact that represents a FE space with no DOFs at all, neither at the cell interiors, nor at their boundary n-faces. This sort of software resource has been proven extremely efficient for: (1) the numerical solution of a PDE on a subdomain of our original discretized domain (which thus has to be aligned with the cells boundaries); (2) the numerical solution of a PDE using XFEM-like discretization techniques (which are grounded on FE spaces that do not assign DOFs to cells exterior to the embedded domain); (3) to simplify the implementation of discretization methods for PDE problems that involve coupling at the interface level, e.g., fluid-structure interaction.

Apart from these `reference_fe_t` subclasses, there are already concluded developments within this hierarchy in a beta version of the code, such as B-splines [8], and other scheduled developments, such as div-conforming FEs [7].

7 The Description of the Physical Domain: The `triangulation_t` Abstraction

A central abstraction in all FE numerical simulation codes is the one that describes the triangulation/mesh $\mathcal{T}_h$ of the physical domain $\Omega \subset \mathbb{R}^d$ in which our problem is posed. (In practice, the mesh generation for Ω introduces a geometrical error, and the mesh is in fact over an approximated domain Ω_h). In FEMPAR, this abstraction is called `triangulation_t`. With flexibility, and code reuse in mind, this is an abstract data type. In Sect. 7.1, we introduce `triangulation_t`, and the mechanism that it provides to its subclasses in order to preserve encapsulation and data hiding, while still letting subclasses to store and access to data efficiently. For completeness, in Sect. 7.2, we introduce details underlying the implementation of a particular concrete subclass of `triangulation_t`.

7.1 An Abstract Triangulation Representation and Its Software Implementation

In this section, we present an abstract (conceptual) representation of a triangulation that FEMPAR exposes to user-level applications and other library software abstractions that are grounded on it (see, e.g., Sect. 10). This conceptual representation is provided by a set of abstract derived data types (and the methods bounded to them) to which we have converged as a result of our experience in accommodating a wide range of state-of-the-art FE discretizations and solver techniques within a single framework, from desktops/laptops, to high-end distributed-memory supercomputers (see Sect. 2).

For the sake of brevity, in this work we restrict ourselves to a subset of this representation that only provides support to the implementation of high-order conforming and non-conforming FE discretizations grounded on *conforming meshes* in a serial computing environment. We stress, however, that the actual (complete) representation also incorporates concepts to express the mesh in a distributed-memory environment (e.g., the set of cells of a subdomain is divided into local cells and a layer of cells owned by remote subdomains, which we denote as *ghost cells*). It also provides support to the implementation of high-order *hp*-adaptive (i.e., on locally refined, non-conforming meshes) conforming and non-conforming FEs (using hanging node constraints [82] and subface integration over a facet between cells of different refinement level, respectively) and to the implementation of XFEM-type techniques (see [60] and references therein); provided an implicit representation of the geometry of the domain, a background mesh is able to know whether a cell is interior, exterior or cut by the domain, and in the latter case, to provide the coordinates of the intersection points. This extra expressivity comes in the form of additional data types and an extended set of methods for those data types that are covered in this section. We stress, however, that neither the former nor the latter ones will be covered in this section.

Although our abstract representation of a triangulation has been proven to have high expressivity, we do not claim, however, that our triangulation representation is universally applicable to the implementation of arbitrary numerical discretization and solver techniques. It indeed has been designed such that extra extensions are foreseen to satisfy further requirements.

The triangulation representation encompasses both topological and geometric data. A triangulation is conceived as a partition of Ω into a set of cells (d-faces). Each cell is uniquely identified by a global identifier in the range $\texttt{cell\_gid} = 1, \ldots, \texttt{num\_cells}$.[14] Apart from the cells, a triangulation is also composed by a set of lower dimensional objects, i.e., a set of k-faces, for $k = 0, \ldots, d-1$. We will also refer to elements in this set as "vefs", provided that in the $d = 3$ case, it is composed of vertices, edges, and faces. Each of the objects in this set is uniquely identified by a global identifier in the range $\texttt{vef\_gid} = 1, \ldots, \texttt{num\_vefs}$.[15]

Apart from the cells and vefs, a triangulation also encompasses adjacency data. This sort of data describes how n-faces in a mesh are related to each other. We denote by F the set of all n-faces in the mesh, by F^k the set of all k-faces, and by F_i and F_i^k the i-th n-face (of arbitrary dimension) and the i-th k-face (of fixed dimension k), respectively. In conforming meshes, there are mainly two relevant types of adjacency relationships, namely *composition* (m-faces that are part of a k-face for $m < k$) and *neighbourhood* (m-faces around a given k-face for $m > k$). Following [87], the set of m-faces adjacent to F_i^k, is denoted by $F_i^k\langle F^m \rangle$ (i.e., the operator $\langle \cdot \rangle$ selects from the set the m-faces adjacent to the one in the left). A triangulation conforming with `FEMPAR` abstract representation should be able to provide the composition data $F_i^3\langle F \rangle$, and the neighbourship data $F_i\langle F^3 \rangle$, that is, n-faces that compose each cell and cells around n-faces.

A triangulation also includes geometry data. Cell geometries are represented by a map $\boldsymbol{\Phi}_K$ of a polytope $\hat{K}$ in the reference space to the physical space (see Sect. 3). This map is represented as a function of a *scalar* FE space (e.g., grounded on high-order Lagrangian FEs or B-splines), with its DOF values being the vectors of node coordinates (i.e., `point_t` instances) in the physical space.

At the core of the software design in charge of providing the triangulation-related data covered so far is an abstract data type named `triangulation_t`. (The rationale behind this data type being abstract will be made clear in the course of this section.) This data type is defined as shown in Listing 9. `triangulation_t` is conceived as a template to which all subclasses have to conform. On the one hand, it is composed by a (*minimal*) set of member variables encompassing data common to any triangulation. In particular, any triangulation is embedded in a `num_dimensions`-dimensional space, and is composed of a total number of `num_cells` (`num_dimensions`-dimensional) cells and `num_vefs` vefs, respectively; see Lines 3–5 of Listing 9, respectively. On the other hand, `triangulation_t` is equipped with a set of *deferred* methods that the subclasses of `triangulation_t` must implement; see Lines 11–18. The rationale underlying these methods requires further elaboration, to be discussed in the sequel.

[14] We note that the actual conceptual representation of the triangulation in `FEMPAR` differences among local (to subdomain) cell identifiers and global cell identifiers (among the whole triangulation of the domain) in a distributed-memory context. The second sort of identifiers are coded as long precision integers, i.e., `integer(igp)`, in order to accommodate simulations with more than $2^{31} - 1$ global cells.

[15] As mentioned in the case of cells, the actual conceptual representation of the triangulation in `FEMPAR` differences among local (to a subdomain) vef identifiers and global vef identifiers (among the whole triangulation of the domain) in a distributed-memory context. Again the latter ones are long precision integers.

```
1  type, abstract :: triangulation_t
2   private
3   integer(ip) :: num_dimensions
4   integer(ip) :: num_cells
5   integer(ip) :: num_vefs
6   ... ! Rest of member variables of the full conceptual triangulation representation
7  contains
8   procedure :: get_num_dimensions => triangulation_get_num_dimensions
9   procedure :: get_num_cells      => triangulation_get_num_cells
10  procedure :: get_num_vefs       => triangulation_get_num_vefs
11  !subroutine (this(in)::class(triangulation_t),cell(inout)::class(cell_iterator_t),allocatable)
12  procedure(...), deferred :: create_cell_iterator
13  !subroutine (this(in)::class(triangulation_t),cell(inout)::class(cell_iterator_t),allocatable)
14  procedure(...)  , deferred :: free_cell_iterator
15  !subroutine (this(in)::class(triangulation_t),vef(inout)::class(vef_iterator_t),allocatable)
16  procedure(...) , deferred :: create_vef_iterator
17  !subroutine (this(in)::class(triangulation_t),vef(inout)::class(vef_iterator_t),allocatable)
18  procedure(...) , deferred :: free_vef_iterator
19  ... ! Rest of regular and deferred bindings of the full conceptual triangulation representation
20 end type triangulation_t
```

Listing 9. The `triangulation_t` abstract data type.

In order to construct a conceptual view of `triangulation_t` suitable for the user (and library) code needs, FEMPAR relies on the so-called *iterator* OO design pattern [88]. Iterators are data types that provide sequential traversals over the *full sets of objects* that all together (conceptually) comprise `triangulation_t` as a mesh-like container. There are several different iterators available, each one related to a different set of objects to be traversed. For example, `cell_iterator_t` provides traversals over the set composed of all cells, while `vef_iterator_t` over the one composed of all vefs.[16] In our software design, iterators are created and freed by a set of public TBPs provided by `triangulation_t`; see Lines 11–18 of Listing 9. Thus, for example, the expression `call triangulation%create_cell_iterator(cell)` creates an iterator on the `cell` client-space instance, while `call triangulation%free_cell_iterator(cell)` frees it. Iterators sequentially traverse objects in increasing order by their global identifiers. However, we note that `triangulation_t` subclasses are completely free to decide how to internally label these objects.[17]

As the reader might have already noted from the minimal set of member variables in Listing 9 (among others), our software design is such that we want to provide *complete flexibility* to concrete subclasses of `triangulation_t` with respect to how do they internally layout the (topology and geometry) data to be provided. To this end, `triangulation_t` is an abstract class that defers this decision to its subclasses. There is a clear separation among how the data is handled (i.e., stored and accessed) by the *private data structures* (member variables) underlying `triangulation_t` subclasses, and the conceptual/abstract view of `triangulation_t` exposed to FEMPAR users. This view renders `triangulation_t` easily accessible and understandable. Whereas the public interface of `triangulation_t` being used by client codes is designed to be stable over time, the internals of `triangulation_t` subclasses, however, are allowed to (and are subject to) change over time (e.g., in order to accommodate further optimizations, additional requirements, etc.). At the price of dynamic run-time polymorphism, `triangulation_t` subclasses might be designed such that they strongly strive to preserve encapsulation and data hiding while *still storing and accessing to data efficiently*. Thus, e.g., a `triangulation_t` subclass in charge of handling structured/uniform meshes of simple domains may decide to not explicitly store the cell-wise global vef identifiers, nor the vertex coordinates of the mesh, but instead to provide them implicitly on demand as a function of the global cell identifier.

Apart from encompassing the logic underlying the actual traversal over objects of the set at hand, iterators also have the following crucial responsibility. Following the software concept of "accessors" presented in [17], they are able to tease out the data related to the current object on which they are seated from the global arrays and rest of private data structures that comprise the internals of the corresponding `triangulation_t` subclass. They therefore do not explicitly store, e.g., the global vef identifiers of the current cell. Instead, they know how to fetch

[16] For completeness, let us mention that `triangulation_t` also offers traversals over *subsets of objects* conveniently selected for acceleration purposes. For example, `triangulation_t` provides an iterator over vertices, edges, and faces that lay on the interface among subdomains, called `itfc_vef_iterator_t` (i.e., a subset of the set of objects traversed by `vef_iterator_t`) for those subclasses suitable for parallel distributed-memory environments.

[17] Thus, e.g., a `triangulation_t` subclass that internally labels the global identifiers of vefs by their dimension in increasing order would result in a traversal with such an order. This is however a potentially changing over time low-level implementation detail that user programs relying on `triangulation_t` and its associated iterators should not assume nor rely on.

```
subroutine set_..._create(this,triang)
 class(set_iterator_t), inout  :: this
 class(trian..._t), target, in :: triang
 call this%free()
 call this%triangulation=>triang
 call this%first()
end subroutine set_iterator_create

subroutine set_iterator_first(this)
  class(set_iterator_t), inout  :: this
  this%gid = 1
end subroutine set_iterator_first
```

```
function set_iterator_has_finished(this)
 class(set_iterator_t), in :: this
 logical :: set_iterator_has_finished
 assert (associated(this%triangulation))
 set_iterator_has_finished = &
  this%gid > this%triang...%get_num_sets()
end function set_iterator_has_finished

subroutine set_iterator_next(this)
  class(set_iterator_t), inout  :: this
  this%gid = this%gid+1
end subroutine set_iterator_nest
```

Fig. 7 Implementation of a partial (selected) subset of the bindings of `set_iterator` (see Listing 10)

them from the corresponding `triangulation_t` subclass into data structures suitable for the user needs. Provided that it is the responsibility of `triangulation_t` subclasses to decide how to internally layout data, iterators are abstract data types as well, and most of its TBPs are deferred/virtual. This also justifies why the methods in the Lines 11–18 of Listing 9 are deferred, and why the corresponding iterator dummy arguments, polymorphic allocatable. It is ultimately the responsibility of the concrete subclass of `triangulation_t` to decide on execution time the dynamic type of the polymorphic variable being created.

Let us next discuss the rationale underlying the design of iterators over cells and vefs. These data types are defined in Listing 10, where `set` must be actually replaced by the corresponding name uniquely identifying the set of objects to be traversed by the iterator at hand, i.e., either `cell` or `vef`. In Fig. 7, we illustrate the implementation of a partial (selected) subset of the bindings of these data types.

The `create` binding of `set_iterator_t` takes as input a polymorphic `triangulation_t` instance to be traversed, and leaves the iterator positioned in the first object of the `set`, i.e., in a state ready to start the sequential traversal over all of its objects; see Fig. 7. This method (like `free`) is not intended to be directly called by the user. Instead, `triangulation_t` clients should rely on the deferred bindings of `triangulation_t` presented in Listing 9. The `init`, `next`, and `has_finished` bindings let clients to position the iterator on the first object of the `set`, move to its next object, and check whether all of its objects have been already traversed or not, respectively; see Fig. 7.

The actual set of (deferred) TBPs of a `triangulation_t` iterator highly depends on the type of object being pointed. We now briefly discuss those TBPs in the set corresponding to cell and vef iterators that provide support to the subset of the triangulation conceptual representation we are focusing on. These are in particular enumerated in Listing 11.

```
type, abstract :: set_iterator_t
  private
  class(triangulation_t), pointer :: triangulation => NULL()
  integer(ip)                     :: gid
contains
  public
  procedure :: create            => set_iterator_create
  procedure :: free              => set_iterator_free
  procedure :: first             => set_iterator_first
  procedure :: next              => set_iterator_next
  procedure :: has_finished      => set_iterator_has_finished
  procedure :: get_gid           => set_iterator_get_gid
  procedure :: set_gid           => set_iterator_set_gid
  procedure :: get_triangulation => set_iterator_get_triangulation
  ... ! Set of deferred TBPs providing access to data items in the corresponding set
end type set_iterator_t
```

Listing 10. `triangulation_t` “set” (either cell or vef) iterators.

```
1  type, abstract :: cell_iterator_t
2    ... ! Member variables (see Listing 10)
3  contains
4    public
5    ... ! Create/free/traversal TBPs (see Listing 10)
6
7    ! Topology-data related TBPs (i.e., F_i^3⟨F⟩)
8    procedure(...), deferred :: get_num_vefs
9    procedure(...), deferred :: get_vef
10   procedure(...), deferred :: get_vef_lid
11   procedure(...), deferred :: get_vef_gid
12   procedure(...), deferred :: get_vefs_gid
13
14   ! Transformation among n-face-wise node local numberings (see Section 3.16)
15   procedure(...), deferred :: get_permutation_index
16
17   ! Geometry-related data
18   procedure(...), deferred :: get_reference_fe
19   procedure(...), deferred :: get_num_nodes
20   procedure(...), deferred :: get_nodes_coordinates
21
22   ! Set IDs-related
23   procedure(...), deferred :: get_set_id
24   procedure(...), deferred :: set_set_id
25
26   ... ! Rest of TBPs of full conceptual triangulation presentation (e.g., is_local() vs is_ghost())
27 end type cell_iterator_t
28
29 type, abstract :: vef_iterator_t
30    ... ! Member variables (see Listing 10)
31 contains
32   public
33   ... ! Create/free/traversal TBPs (see Listing 10)
34
35   ! Topology-data related TBPs (i.e., F_i⟨F^3⟩)
36   procedure(...), deferred :: get_num_cells_around
37   procedure(...), deferred :: get_cell_around
38
39   ! Geometry related-data TBPs
40   procedure(...), deferred :: get_num_nodes
41   procedure(...), deferred :: get_nodes_coordinates
42
43   ! Misc TBPs
44   procedure(...), deferred :: is_at_interior
45   procedure(...), deferred :: is_at_boundary
46   procedure(...), deferred :: get_dimension
47   procedure(...), deferred :: get_set_id
48   procedure(...), deferred :: set_set_id
49
50   ... ! Rest of TBPs  of the full conceptual triangulation presentation (e.g., is_at_interface())
51 end type vef_iterator_t
```

Listing 11. A subset of the deferred TBPs of the `cell_iterator_t` and `vef_iterator_t` data types (follow-up to Listing 10).

The TBPs in Lines 8–12 of Listing 11 are in charge of providing data related to the composition relationship $F_i^3\langle F\rangle$. In particular, the `get_num_vefs` binding returns the number of vefs on the boundary of the mesh (i.e., the cardinality of the composition relationship). Given the local index of a vef in a cell (within the range $1,\ldots,$`num_vefs`), `get_vef` positions the `vef_iterator_t` instance on input such that it points to this vef, while `get_vef_gid`, returns its global identifier; `get_vef_lid` performs the inverse translation to the one of `get_vef_gid`. Finally, `get_vefs_gid` let the client obtain the global identifier of all vefs of the current cell in one shot provided a user-space pointer to integer array. The semantics of this last TBP are such that subclasses of `cell_iterator_t` are not allowed to allocate the provided pointer, but to associate it with existing (internal) memory (for increased performance and memory leaks avoidance).

The TBP in Line 15 of Listing 11 provides support to the implementation of the transformation procedure described in Sect. 3.16. In particular, this binding has to be invoked on a `cell_iterator_t` instance positioned in the source cell, and given a `cell_iterator_t` positioned on the target cell, and the n-face local identifier within the former and latter cells, returns the permutation index; see Sect. 3.16. We stress that both the rotation and orientation indices can be always computed using the TBPs in the previous paragraph. For example, in order to determine the rotation index, one can extract the global id of the anchor vertex of the n-face in the target cell (by calling get_vef_gid), and then searching for this global id in the set of vertices that comprise the n-face in the target cell (using an iterator over the corresponding sublist in vertices_n_face; see Sect. 6.1). However, we preferred to provide a specialized deferred binding for such purpose in order to leave room for optimizations in triangulation_t subclasses. For example, in the case of a subclass that works with oriented meshes, then get_permutation_index may be implemented such that it always returns the permutation index corresponding to the identity transformation. In the case of a subclass of triangulation_t that is intended to remain static (or to be adapted very infrequently) during the course of the simulation process (see, e.g., Sect. 7.2), then it might be beneficial for performance to precalculate all possible permutation indices during set up into lookup tables, and re-use them all the way through without having to perform the aforementioned searches over and over again.

The TBPs in Lines 18–20 are in charge of providing the cell geometry related-data. In particular, get_reference_fe returns a polymorphic pointer to the reference_fe_t instance that describes the space of functions to which the mapping $\mathbf{\Phi}_K$ belongs. get_num_nodes and get_nodes_coordinates return the number of nodes describing the geometry of the cell, and its associated coordinates in physical space, respectively. Instead of a pointer to an user-space array to be associated with internal storage (as get_vef_gids), get_nodes_coordinates takes a user-space (pre-allocated) array of type point_t instances, and fills it (because of reasons made clear in Sect. 8.3). Assuming that reference_fe_t is a bi-linear Lagrangian FE on a quadrilateral, then get_num_nodes would return 4 (one node per cell-vertex), while get_nodes_coordinates the coordinates in physical space of its vertices.

Any triangulation_t subclass should let its clients to classify the cells into sets. Each set is globally identified by an integer number, named set_id. The methods get_set_id and set_set_id let the caller to associate a set to the current cell, or to retrieve the set to which the cell is currently associated. Cells set identifiers are primarily (although not only) used by fe_space_t during its set-up; see Sect. 10. In particular, they instruct the latter to determine which reference_fe_t instances to use on top of the cells belonging to the same set. For example, assuming that we want to solve a scalar, single-field PDE problem on a subdomain of our original domain (that we assume to be aligned with the cells boundaries), we would use two different sets. The first for the cells that are interior to the subdomain, and the second for those that are exterior. Then we could associate e.g., a linear Lagrangian reference FE to cells in the first set, and void_reference_fe_t on those cells of the second set; see Sect. 6.5.

Sitting on a given vef, the TBPs in Lines 36–37 are in charge of providing data related to the adjacency relationship $F_i\langle F^3\rangle$. In particular, get_num_cells_around returns its cardinality, while get_cell_around returns a cell in this set. To be more precise, the latter TBP positions the instance of cell_iterator_t on input such that it points to a cell in this set identified with an index within the range $1,\ldots,\texttt{get\_num\_cells\_around}()$. The order in which the cells around a vef are listed can be arbitrary, so that codes relying on triangulation_t should not assume, e.g., that they are ordered increasingly by their global cell identifiers. On the other hand, get_num_nodes and get_nodes_coordinates return the number of points on top of the vef (including those on top of the lower-dimensional ones on its boundary), and its associated coordinates in physical space, respectively; see Lines 40–41. We adopt the convention that these nodes are (locally) labeled (within the input/output array of point coordinates to be filled) according to the reference coordinate system of the *first cell* around the vef, i.e., the cell obtained as vef%get_cell_around(1,cell).

The TBPs in Lines 44–48 let the client to determine whether the vef is at the interior of the domain or on its boundary, the vef dimension (e.g., in 3D, it would return 0, 1, and 2 for vertices, edges, and faces, respectively) and to retrieve the set to which the vef is currently associated, or associate a new set to it, respectively. Sets in the case of vefs are primarily used to codify the boundary conditions of the PDE problem at hand, as discussed in Sect. 10.4.

At this point we are already in position to show user-level code that exploits the software design covered so far. In particular, Listing 12 splits the whole set of triangulation cells into two disjoint sets, those that are in contact to the boundary of the domain, and those that are in its interior.

```
type(my_triangulation_t) :: triangulation
class(cell_iterator_t), allocatable :: cell
class(vef_iterator_t) , allocatable :: vef
integer(ip), parameter :: interior_cell_set_id = 1
integer(ip), parameter :: boundary_cell_set_id = 2

... ! Set up triangulation using the TBPs of my_triangulation_t
    ! (my_triangulation_t extends triangulation_t)
call triangulation%create_cell_iterator(cell)
call triangulation%create_vef_iterator(vef)
do while ( .not. cell%has_finished() )
  call cell%set_set_id(interior_cell_set_id)
  do vef_lid=1,cell%get_num_vefs()
    call cell%get_vef(vef_lid,vef)
    if (vef%is_at_boundary()) then
       call cell%set_set_id(boundary_cell_set_id)
       exit
    end if
  end do
  call cell%next()
end do
... ! Execute code which consumes set IDs for whatever purpose
call triangulation%free_cell_iterator(cell)
call triangulation%free_vef_iterator(vef)
... ! Free triangulation using the TBPs of my_triangulation_t
```

Listing 12. User-level code illustrating the usage of the data types and its associated TBPs supporting FEMPAR conceptual triangulation representation.

7.2 An Example `triangulation_t` Subclass and Rationale

In this section, we discuss how a particular subclass of `triangulation_t` is internally organized in order to efficiently provide triangulation-related data by means of the software abstractions presented in Sect. 7.1. This subclass is `static_triangulation_t`. A `static_triangulation_t` codifies a *conforming mesh*, which is set up from scratch at the beginning of the simulation, and remains unaltered during the whole process. On the other hand, `static_cell_iterator_t` and `static_vef_iterator_t` are two non-abstract data type extensions of `cell_iterator_t` and `vef_iterator_t`, respectively. By overriding the set of deferred methods of the former ones, the latter ones tease out the data related to the current object on which they are seated from the global arrays and rest of private data structures that comprise the internals of `static_triangulation_t`.

There is no single approach to layout the data within a given triangulation subclass. The seek of an acceptable trade-off among memory consumption, computational time required to set up, update (if it applies), access to triangulation data, and the frequency on which these operations are performed should guide its internal organization. For example, in [87], two storage layouts are presented, and its memory and computational cost for the computation of any possible adjacency relationship is evaluated in 3D. The first one, called *one-level* representation, is defined by $F_i^1\langle F^0\rangle$, $F_i^2\langle F^1\rangle$, and $F_i^3\langle F^2\rangle$, and by $F_i^0\langle F^1\rangle$, $F_i^1\langle F^2\rangle$, and $F_i^2\langle F^3\rangle$ (neighbourhood information). In other words, it stores vertices of each edge, edges of each face, and faces of each cell, together with edges around vertices, faces around edges, and cells around faces. The second one, called *circular* representation, is defined by the composition information $F_i^1\langle F^0\rangle$, $F_i^2\langle F^1\rangle$, $F_i^3\langle F^2\rangle$ (as above), together with the neighbourhood information $F_i^0\langle F^3\rangle$ (cells around vertices). An important property of these two storage layouts is their *completeness*, i.e., the possibility to determine any adjacency without a loop over the entire mesh. The storage requirements for a uniform mesh of a cube domain with N_c cells are $48N_c$ (for hexahedra) and $24N_c$ (for tetrahedra) in the former, and $32N_c$ (for hexahedra) and $16N_c$ (for tetrahedra) in the latter. However, the operation count for determining some adjacencies, although independent of N_c, is high. For example, in the case of the one-level representation, to obtain the cells around a vertex requires 48 (for hexahedra) and 140 (for tetrahedra) operations, whereas only one operation is needed to obtain cells around facets. In the case of the circular representation, these queries involve one and 148 (for hexahedra) or 299 (for tetrahedra) operations, respectively [87]. (We recall that both kind of adjacencies are required by FEMPAR as presented in Sect. 7.1.)

Another quite different storage data layout is the one followed by the triangulation in the deal.II library [17], essentially defined by the composition data $F_i^1\langle F^0\rangle$, $F_i^2\langle F^1\rangle$, and $F_i^3\langle F^2\rangle$ (referred as hierarchical cell representation by the authors of the library), and the neighbourhood data $F_i^3\langle F^2\rangle$ *stored cell-wise* (i.e., a given cell stores the identifiers of its cell neighbours across each face within the cell). Besides, the (potentially non-conforming) triangulation in this library is conceived (and explicitly represented) as a collection of trees, where the cells of a coarsest conforming mesh (generated by deal.II itself for simple domains, or read from a file from several file formats) form the roots, and the children branch off their parent cells, thus forming binary-trees, quad-trees and octtrees in $d = 1, 2,$ and 3 spatial dimensions, respectively [17]. While both the ancestors (i.e., the so-called "inactive" cells) and leaf cells of the tree (i.e., the so-called "active" cells) are stored, only the latter ones actually form the partition of the domain. Apart from a hierarchy of cells, the deal.II triangulation also maintains a hierarchy of k-faces for $k = 1, \ldots, d-1$. Such quite complex data structure is justified by the authors for two reasons. First, it allows for an efficient implementation of adaptive mesh adaptation (including coarsening and refinement). The hierarchy of n-faces aids in the process of handling the so-called hanging node constraints required to build conforming FE spaces on top of non-conforming meshes. The second reason is the implementation of (geometric) multigrid preconditioners grounded on the adaptivity tree. In particular, such preconditioners require that DOFs are also associated to inactive cells. Thus, also inactive n-faces have to explicitly exist in the triangulation. In any case, such structure is hard to generate and maintain, and does not fit well when integrated with parallel octree libraries [89], like `p4est` [90]. The whole hierarchy must be generated from scratch on each mesh adaptivity step. However, based on our own experience, *such hierarchy is not really needed for an efficient implementation of adaptive refinement*. The second reason, i.e., the implementation of a serial hierarchical multigrid solver in deal.II, would probably be more complicated without such a hierarchical representation of the mesh.

While the hierarchical cell representation in deal.II has been proven to be successful in the implementation of highly complex *hp*-adaptive FE discretization [82] and reduces memory consumption over $F_i^3\langle F\rangle$, the restriction of the global vef identifiers to a cell (a very frequent operation in FE codes), becomes significantly more expensive in this storage layout as this operation requires permutations among the reference coordinate system of the cell that owns the vef to the one to which we are restricting to; the same applies to the restriction of global DOF identifiers to a cell when the DOFs are stored n-face-wise. Furthermore, it is a *non-complete* storage layout. In particular, neighbourship data $F_i\langle F^3\rangle$ has to be computed by the user by means of a loop over all cells. Besides, it prevents library support to loops over the facets of the mesh, and access to the neighbouring cells, a natural operation in the implementation of DG methods. In our experience, facet-loop based integration of DG terms (versus cell-loop based) leads to a software that is significantly easier to use, as it might be designed such that most of the complexity underlying facet integration can be hidden to the user (see Sect. 9). Finally, although it is very efficient for hierarchical and local mesh adaptation (within each subdomain), the most severe drawback is its costly set up (from scratch) for a given initial conforming coarse mesh (this can be mitigated by reducing the coarse mesh resolution, at the price of potentially losing geometry modelling accuracy), and, in a distributed-memory environment, the even more costly regeneration of an adapted non-conforming forest of trees after a re-distribution step among processes for dynamic load-balancing [90]. Indeed, in [89], the latter is reported as the second more costly operation in the simulation pipeline, only below the linear solver step.

The `static_triangulation_t` data type *explicitly* stores the composition data $F_i^3\langle F\rangle$, and the neighbourship data $F_i\langle F^3\rangle$ within its internal (private) member variables.[18] The memory consumption of such *complete* storage layout is $52N_c$ (hexahedra) and $28N_c$ (tetrahedra), which is less than twice the one of the one-sided and circular representations [87]. At the price of this increased memory consumption, `static_triangulation_t` is able to provide the required adjacency data with $\mathcal{O}(1)$ arithmetic complexity. Besides, the cell-based storage of the composition relationship is perfectly suited for its migration in parallel distributed-memory environments. On the other hand, the amount of *permanent storage* of this data layout can be reduced if one exploits the fact that neighbourship data is only required in very specific parts of the code. For example, unstructured mesh generators usually provide only the composition data $F_i^3\langle F^0\rangle$. In such a case, `static_triangulation_t` requires the neighbourship data $F_i^0\langle F^3\rangle$ (plus the reference cell topology data encompassed within the `reference_fe_t` instance mapped to each cell; see Sect. 6.1) in order to set up the composition data $F_i^3\langle F^1\rangle$ and $F_i^3\langle F^2\rangle$. It is also needed in `triangulation_t` subclasses suitable for distributed-memory computers, among others, to set up the data structures required to perform nearest neighbour exchanges of DOFs nodal values among subdomains. (We stress that this process requires to globally identify interface DOFs consistently among subdomains sharing such DOFs .) In this latter scenario, this adjacency data is only required for n-faces that lay on the inter-subdomain interface (and not for those on the interior). The evaluation of facet integrals (as designed in `FEMPAR`, see Sect. 9) also requires at least $F_i^2\langle F^3\rangle$ and $F_i^1\langle F^2\rangle$, in 2D and 3D, respectively. The use of the full adjacency data can be needed for the implementation of advanced numerical discretization schemes, e.g.,

18 We note that $F_i^3\langle F^3\rangle$ is simply F_i^3 and is not stored.

for the implementation of nodal-based *shock detectors* for monotonic FEs [58, 59]. Due to the aforementioned reasons, we decided to design `static_triangulation_t` such that it permanently stores such data, but we stress that our software design is such that a triangulation subclass is always free to offer methods that set up and destroy these data on demand to reduce the amount of permanent data storage.

```
1  type, extends(triangulation_t) :: static_triangulation_t
2    private
3    ! Container of polymorphic reference_fe_t instances (See Section 6)
4    type(p_reference_fe_t)   , allocatable :: reference_fes(:)
5
6    ! Mapping of cell GiDs to reference_fes(:)
7    integer(ip)              , allocatable :: cell_to_ref_fes(:)
8
9    ! Composition data $F_i^3\langle F\rangle$
10   integer(ip)              , allocatable :: ptr_vefs_x_cell(:)
11   integer(ip)              , allocatable :: lst_vefs_gids(:)
12
13   ! Neighborship data $F_i\langle F^3\rangle$
14   integer(ip)              , allocatable :: ptr_cells_around(:)
15   integer(ip)              , allocatable :: lst_cells_around(:)
16
17   ! Geometry interpolatory nodes global numbering + nodes coordinates in physical space
18   integer(ip)              , allocatable :: ptr_nodes_x_cell(:)
19   integer(ip)              , allocatable :: lst_nodes_gids(:)
20   type(point_t)            , allocatable :: nodes_coordinates(:)
21
22   ! Cell and vef set IDs, vefs at boundary?
23   integer(ip)              , allocatable :: cells_set_ids(:)
24   integer(ip)              , allocatable :: vefs_set_ids(:)
25   logical                  , allocatable :: vefs_at_boundary(:)
26
27  contains
28   public
29   procedure :: create  => static_triangulation_create
30   procedure :: free    => static_triangulation_free
31   ... ! TBPs overriding those which are deferred in triangulation_t (See Listing 9)
32   ... ! Comprehensive set of private TBPs providing support to the
33       ! implementation of the public TBPs above. These are hierarchically organized into
34       ! short, auxiliary re-usable subroutines/functions for code readability purposes
35 end type triangulation_t
```

Listing 13. The internals of `static_triangulation_t` and a selected set of its bindings.

The `static_triangulation_t` data type, together with a selected set of its bindings, is defined as shown in Listing 13. Before going into more detail, there are two main points to remark with respect to how this type internally layouts its data. First, it relies all the way through on intrinsic Fortran allocatable arrays. These sort of data structures are perfectly suited for the particular case of `static_triangulation_t`, due to its static nature. We stress, however, that more efficient data structures (i.e., able to mitigate the effect of frequent/costly allocatable array re-allocations) would be convenient if it also had to support mesh adaptation (e.g., a linked list, or even better for data locality, a data structure with semantics close to `std:vector` of the C++ standard template library, which in fact is already in FEMPAR but not included for brevity). Second, for increased data locality during cell and vef sequential traversals (and thus a more efficient on the memory hierarchy of modern computer architectures) the data is not stored into cell-wise or vef-wise local arrays, but into global arrays that are indexed either by the global cell or vef identifiers.

A collection of `reference_fe_t` polymorphic instances is stored in the `reference_fes(:)` array (see Line 4 of Listing 13). These instances are uniquely identified (within the local scope of `static_triangulation_t`) by their position in this array. For a given cell with global identifier `cell_gid`, the FE space of functions to which the cell mapping $\mathbf{\Phi}_K$ belongs, is described by the `reference_fe_t` instance with identifier `cell_to_ref_fes(cell_gid)` in the collection; see Line 7. The member variables used to store the composition data $F_i^3\langle F\rangle$ are encompassed within Lines 10–11 of Listing 13. As stated above, the global vef identifiers are stored cell-wise, in the `lst_vefs_gids(:)` array, which is in turn (indirectly) addressed by the `ptr_vefs_x_cell(:)` array. In particular, the ones assigned to the vefs on cell `cell_gid` start and end in position `ptr_vefs_x_cell(cell_id)` and `ptr_vefs_x_cell(cell_id+1)-1` of `lst_vefs_gids(:)`, respectively. Thus, e.g., the implementation of the (overridden) `get_num_vefs` TBP in `static_cell_accessor` (see Listing 12), just determines the number of vefs on the boundary of the current cell as `ptr_vefs_x_cell(cell_id+1)-ptr_vefs_x_cell(cell_id)`. On the other hand, the member variables used to store the adjacency data $F_i\langle F^3\rangle$ are encompassed within Lines 14–15 of Listing 13. The global identifiers of the cells around a vef `vef_gid` start and end in position `ptr_cells_around(vef_gid)` and `ptr_cells_around(vef_gid+1)-1` of `lst_cells_around(:)`, respectively.

The geometry-related data is handled by the member variables in Lines 18–20. In particular, during the set up of `static_triangulation_t` a global numbering of the nodes of the global FE space describing the geometry of the mesh is internally built. (The process that generates such numbering is identical to the one described in Sect. 10.3, so that we omit it here to keep the presentation short.) In particular, the global node identifiers restricted to cell `cell_gid` start and end in position `ptr_nodes_gids(cell_id)` and `ptr_nodes_gids(cell_id+1)-1` of `lst_nodes_gids(:)`, respectively. These global node identifiers are used to (indirectly) address the global array of nodes coordinates in Line 20. The `cells_set_ids(:)` and `vefs_set_ids(:)` arrays are used to store the user-provided cell and vef set identifiers (see Sect. 7.1), respectively, while `vefs_at_boundary(:)`, whether the corresponding vef lays on the boundary of the domain or not.

Finally, the `static_triangulation_create` binding sets up a new `static_triangulation_t` instance. There are two options for creating a `static_triangulation_t` in `FEMPAR`, depending on whether the mesh is structured or unstructured. In the first case, `FEMPAR` provides the machinery for the automatic generation of a triangulation on simple domains (e.g., a unit cube), currently of brick (quadrilateral or hexahedral) cells. This function is implemented exploiting a tensor product structure of the space, numbering cells and vefs using lexicographical order. The second way to create a `static_triangulation_t` instance is from a mesh data file, e.g., using the GiD mesh generator [91].

8 Evaluation of Cell Integrals

In this section, we describe the data structures required to perform the numerical integration of the local matrices. In order to compute cell integrals (12), one needs (among others) functionality to evaluate the shape functions and their derivatives at the quadrature points in the physical cell and the determinant of the Jacobian at the quadrature points in the reference cell. In turn, the evaluation of the shape functions and derivatives in the physical cell rely on their evaluation (and possibly the evaluation of the Jacobian) in the reference cell (see, e.g., (13) and (14)). We note that the evaluation of $\hat{\boldsymbol{\Psi}}$ does not require any additional information; it is the identity for Lagrangian elements and only requires the Jacobian in the reference cell for vector-valued shape functions (see (17) and (18)). In the following, we present a set of data types that contain all this information.

The evaluation of cell integrals involves the data type `quadrature_t` that represents the quadrature Q, `interpolation_t`, that stores the values of the shape functions and its first derivatives (either in the reference or physical space) at the quadrature points of Q, and a `cell_map_t` that describes the mapping from a reference to a physical cell $\boldsymbol{\Phi}_K$ (e.g., Jacobian-related data). Additionally, the data type `cell_integrator_t` provides the machinery to compute the `interpolation_t` corresponding to the physical space from the one at the reference space and the `cell_map_t` at every cell of the triangulation. In the following sections, we cover in detail these software abstractions.

8.1 Numerical Quadrature

The data type that in `FEMPAR` represents an arbitrary quadrature rule is called `quadrature_t` and is defined as shown in Listing 14.

```
type quadrature_t
 private
 integer(ip)              :: num_dimensions
 integer(ip)              :: num_quad_points
 real(rp), allocatable :: coordinates(:,:)
 real(rp), allocatable :: weights(:)
contains
 public
 ! subroutine(this(inout)::class(quadrature_t),num_dimensions(in),num_quad_points(in)::integer(ip))
 procedure, non_overridable :: create => quadrature_create
 ! subroutine(this(inout)::class(quadrature_t))
 procedure, non_overridable :: free   => quadrature_free
 ... ! Rest of getter TBPs to preserve encapsulation
end type quadrature_t
```

Listing 14. The `quadrature_t` data type.

In Listing 14, `coordinates(:,gp)` and `weights(gp)` store, respectively, $\hat{\boldsymbol{x}}_{gp} \in \mathbb{R}^{\text{num\_dims}}$ and w_{gp}, for $gp = 1, \ldots,$ `num_quadrature_points`. It might readily be observed from the interface of its `create` binding that `quadrature_t` is designed to be simply a placeholder for the quadrature points coordinates and its associated weights. Indeed, this binding essentially allocates `coordinates(:,:)` and `weights(:)`. The code that ultimately decides how to distribute the quadrature points over $\hat{K}$ and set up its associated weights is actually bounded to the `reference_fe_t` implementors through the deferred binding with interface shown in Listing 15.

```
abstract interface
 ...
 subroutine reference_fe_create_quadrature ( this, quadrature, degree )
  class(reference_fe_t)          , intent(in)    :: this
  type(quadrature_t)             , intent(inout) :: quadrature
  integer(ip)          optional , intent(in)    :: degree
 end subroutine reference_fe_create_quadrature
 ...
end interface
```

Listing 15. The interface of the `create_quadrature` deferred binding of `reference_fe_t`.

All `reference_fe_t` subclasses currently available in FEMPAR select by default a Gaussian quadrature that exactly integrates mass matrix terms (within their implementation of the binding in Listing 15) by invoking `fill_*_gauss_legendre` methods at lines Lines 13 and 14 in Listing 14. This quadrature can be solely determined from the attributes of the `reference_fe_t` implementor at hand (its topology and order).[19] However, in other more demanding situations, e.g., the integration of a trilinear weak form, the user can provide the desired quadrature degree through the `degree` optional dummy argument. If more general scenarios to the ones currently covered (e.g., a non-Gaussian quadrature) are to be addressed, then the interface might be modified such that an optional parameter dictionary is passed instead.

8.2 Evaluation of Reference Cell Shape Functions

As commented in the introduction of this section, to compute cell integrals (12), one needs to evaluate shape functions and their derivatives in the physical cell, which in turn rely on their evaluation in the reference cell (see, e.g., (13) and (14)). The values of the shape functions and their first derivatives at a set of quadrature points provided by a `quadrature_t` instance are stored in the `interpolation_t` data type presented below. The same data type can be used to store this data in the reference or physical space.

Let us start with the evaluation of shape function in the reference space. The local FE space on top of $\hat{K}$ actually depends on the particular `reference_fe_t` implementor at hand. Consequently, this functionality has to be offered through a deferred binding of this abstract type. The interface of this binding is declared in Listing 16. The subroutine overriding it in concrete subclasses is in charge of computing the shape functions values and derivatives at quadrature points in the reference space and stores them in a raw-data container of type `interpolation_t` (to be discussed later in this section).

```
abstract interface
 ...
 subroutine reference_fe_create_interpolation ( this, quadrature, reference_cell_interpolation )
   class(reference_fe_t), intent(in)    :: this
   type(quadrature_t)   , intent(in)    :: quadrature
   type(interpolation_t), intent(inout) :: reference_cell_interpolation
 end subroutine reference_fe_create_interpolation
 ...
end interface
```

Listing 16. The interface of the `create_interpolation` deferred binding of `reference_fe_t`.

Let us remark several points related to this interface. First, this binding is typically called only once, and the data pre-computed and stored within the passed `interpolation_t` dummy argument is repeatedly re-used when transforming these values to an actual cell; see Sect. 8.4.

[19] As it is well known, considering n-cube topologies for $\hat{K}$, for a Lagrangian reference FE of order p and an affine geometrical map, we need a 1D Gaussian quadrature with $p+1$ points. For tetrahedral meshes with the Duffy transformation, we need to take $n = p + \text{ceiling}(d/2)$ to integrate exactly mass matrices (see Sect. 3.5 for more details).

Second, this binding is designed such that all functions are evaluated at all quadrature points within a single call, instead of following a (much) finer granularity approach in which only one function is evaluated at a quadrature point per call.[20] Third, we stress that the actual implementation of this deferred binding in FEMPAR computes shape functions values and first derivatives in the reference space, whereas it lets the caller to selectively decide whether to compute or not the second derivatives of the shape functions, provided that they are expensive to compute and only required in very particular scenarios; see Sect. 3.7. Indeed, the code implementation of this feature is of cross-cutting nature, being reflected in several interfaces and data types in which the cell (and face) integration functionality is split. We will nevertheless omit here (and in the rest of sections) details regarding second derivatives (and its optional computation) in order to keep the presentation simple.

Let us now discuss on the rationale underlying `interpolation_t`. This data type is not exposed at all to the user of FEMPAR. It is instead used as an internal low-level container that lets the data types involved in the implementation of cell integrals exchange the sort of data subject to consideration. It is ultimately the responsibility of the concrete `reference_fe_t` subclass to decide how the data is actually laid out within the member variables of `interpolation_t`. Thus, `reference_fe_t` is the only data type that can access or modify `interpolation_t`. In its current flavour, `interpolation_t` is a concrete (i.e., non-abstract) data type with a fixed set of multi-rank allocatable array member variables for storing shape function values and derivatives. For example, the one storing shape function values is a 3-rank array, where a `reference_fe_t` implementor may choose its indices, from left to right, to refer to the component of the shape function, the shape function, and the quadrature point, respectively. The `reference_fe_t` subclass is, however, completely free to lay out the data in these arrays, and it is in this flexibility where the extensibility of the software design to accommodate several FE space realizations resides. This, indeed has been proven to be sufficient to (efficiently) implement all FE spaces currently available in FEMPAR, including scalar, vector, and tensor-valued Lagrangian FEs (where higher-rank spaces are determined as the tensor product of the scalar spaces, and shape functions have only one non-zero component), and genuinely vector-valued FE spaces (where more than one component of the shape function may be non-zero).

8.3 Geometrical Mapping

A basic building block is the mapping $\mathbf{\Phi}_K$ among the reference cell $\hat{K}$ coordinate system and the one corresponding to an actual cell K of the triangulation in the physical space; see Sects. 3.2 and 3.3. For example, we are able to pull back the gradients of the shape functions from the reference to the physical space in (14) using the Jacobian of the transformation evaluated at quadrature points, or to evaluate the source term at quadrature points in real space. The Jacobian is also required to the transform the integral from the physical to the reference space in (12) and to compute the Piola transformations in div and curl-conforming FE spaces (see (17) and (18)). The derived type `cell_map_t` in FEMPAR is designed to be a placeholder for the data required to provide this sort of services. It is declared as shown in Listing 17. The rationale underlying the inheritance relationship among `cell_map_t` and `base_map_t` will be made clear in Sect. 9.

[20] Here (and in many other places) we try to maximize the granularity of each call to a deferred binding for efficiency reasons. The reader should be aware that calling to deferred bindings with the granularity of the latter approach would be very expensive, apart from preventing a number of potential compiler optimizations enabled by the former.

```
type base_map_t
 private
 integer(ip)                  :: num_dimensions
 integer(ip)                  :: num_quad_points
 real(rp)     , allocatable   :: jacobian(:,:,:)
 real(rp)     , allocatable   :: det_jacobian(:)
 type(interpolation_t)        :: interpolation
 type(point_t), allocatable   :: coordinates_quadrature_points(:)
 type(point_t), allocatable   :: coordinates_nodes(:)
contains
  public
  ! subroutine(this(inout)::class(base_map_t))
  procedure                   :: free => base_map_free
  ... ! Getter TBPs to preserve encapsulation;
end type base_map_t

type, extends(base_map_t) :: cell_map_t
 private
 real(rp)     , allocatable   :: inv_jacobian(:,:,:)
 ... ! Member variables related to the hessian (2nd derivatives) of the mapping
     ! (omitted to keep the presentation simple)
contains
 public
 ! subroutine(this(inout)      :: class(cell_map_t),
 !            quadrature(in)   :: type(quadrature_t),
 !            reference_fe(in) :: class(reference_fe_t))
 procedure, non_overridable :: create => cell_map_create
 ! subroutine(this(inout)::class(cell_map_t),quadrature(in)::type(quadrature_t))
 procedure, non_overridable :: update => cell_map_update
 ! subroutine(this(inout)::class(cell_map_t))
 procedure, non_overridable :: free   => cell_map_free
 ... ! Rest of getter TBPs to preserve encapsulation;
end type cell_map_t
```

Listing 17. The `cell_map_t` data type.

The `create` binding of `cell_map_t` takes as input a `quadrature_t` instance with a set of integration points where $\boldsymbol{J}_K(\hat{\boldsymbol{x}}_{gp})$, $\boldsymbol{J}_K^{-1}(\hat{\boldsymbol{x}}_{gp})$, and $|\boldsymbol{J}_K(\hat{\boldsymbol{x}}_{gp})|$ are to be evaluated (see Listing 17). These geometry-related data are stored in the `jacobian(:,:,gp)`, `inv_jacobian(:,:,gp)`, and `det_jacobian(gp)` allocatable array member variables of `cell_map_t`, respectively, and allocated during a call to this binding. Apart from a `quadrature_t` instance, `cell_map_t` also requires a description of the (discrete) space of functions to which $\boldsymbol{\Phi}_K$ belongs. FEMPAR supports mappings $\boldsymbol{\Phi}_K$ belonging to abstract FE spaces (e.g., high-order polynomial FE spaces or spline-based spaces). The `reference_fe` dummy argument of polymorphic type `reference_fe_t` serves the purpose. (We note that dynamic run-time polymorphism in this particular context let us re-use `cell_map_t`, e.g., with an arbitrary cell topology.) It turns out that the only information that `reference_fe_t` has to provide to `cell_map_t` are its shape functions, first derivatives, and (on demand) second order derivatives at the quadrature points (in the reference space). The `interpolation` member variable (see Listing 17) is used by `reference_fe` to exchange this sort of data with `cell_map_t` via a call to the `create_interpolation` binding of the former (see Listing 16) during a call to the `create` binding of the latter.

While the `create` TBP of `cell_map_t` is designed to be called once, the `update` TBP of `cell_map_t` is, however, designed to be called multiple times, once per every cell K of the triangulation. A pre-condition of `update` is that the `nodes_coordinates(:)` scratch member variable (see Listing 17) has been loaded with the coordinates in real space of the nodes describing the geometry of K (stored into `point_t` instances). Once this pre-condition is fulfilled, $\boldsymbol{\Phi}_K$ can be expressed as a linear combination of the `reference_fe_t` shape functions with `nodes_coordinates(:)` being the corresponding coefficients in the expansion. At this stage, `coordinates_quadrature_points(:)`, which stores the coordinates of quadrature points in real space, and `jacobian(:,:,:)`, can be easily computed. Finally, `inv_jacobian(:,:,:)` and `det_jacobian(:)` can be computed from `jacobian(:,:,:)` using straightforward numerical algorithms.

8.4 Evaluation of Shape Functions in the Physical Space

The user code that evaluates cell integrals in (12), may need the value, gradient, curl, and divergence of the shape functions at the integration points in the physical space, provided that we want to unburden FEMPAR users from the complexity of having to explicitly apply mapping transformations. As commented in Sect. 3, the mapping that transforms a shape function $\hat{\phi}^a(\hat{\boldsymbol{x}})$ in the reference FE space into the one in the physical space $\phi^a(\boldsymbol{x}) = \hat{\Psi}_K(\hat{\phi}^a) \circ \boldsymbol{\Phi}_K^{-1}$, depends on the particular FE space at hand; see Sects. 3.8, 3.9, and 3.10 for details. For this reason, the actual code that performs these transformations is not actually bounded to `cell_map_t`, but to `reference_fe_t`, through the deferred binding with interface declared in Listing 18.

```
abstract interface
 ...
 subroutine reference_fe_apply_cell_map ( this, cell_map, &
                               interpolation_reference_cell, interpolation_real_cell )
  class(reference_fe_t), intent(in)    :: this
  type(cell_map_t)     , intent(in)    :: cell_map
  type(interpolation_t), intent(in)    :: interpolation_reference_cell
  type(interpolation_t), intent(inout) :: interpolation_real_cell
 end subroutine reference_apply_cell_map
 ...
end interface
```

Listing 18. The interface of the `apply_cell_map` deferred binding of `reference_fe_t`.

The `interpolation_reference_cell` input dummy argument of `apply_cell_map` (see Listing 18) must have been obtained from a call to the binding in Listing 16 invoked on the same `reference_fe_t` instance. The output dummy argument `interpolation_real_cell` holds the shape functions and their derivatives evaluated at quadrature points in physical space (see (13) and (14)). It is also assumed that, on input, `interpolation_real_cell` already contains the data that does not have to be re-computed on each mesh cell, e.g., the value of the shape functions on integration points for Lagrangian FEs; see the discussion related to the `update` binding below for the strategy that we follow in order to fulfill this requirement. This leaves room for optimization in the implementation of this deferred binding (on subclasses), since these quantities do not have to be re-computed on each cell. The `reference_fe_t` subclass uses the `cell_map_t` instance (passed to the `apply_cell_map` binding, see Listing 18) as a placeholder for the data required to provide the mapping transformations required.

We stress, however, that `interpolation_t` is a low level structure that is not designed as a data type that FEMPAR users have to interact with, for reasons made clear in Sect. 8.2. Therefore, we need to introduce an additional data type in our software design, called `cell_integrator_t`, that, among other services, is able to fetch raw data from `interpolation_t` into field data types (i.e., scalars, vectors, and tensors) the user can be easily familiarized with. This data type is declared as shown in Listing 19.

```
type cell_integrator_t
 private
 ... ! num_dimensions, num_quad_points, num_shape_functions
 class(reference_fe_t), pointer :: reference_fe
 type(interpolation_t)          :: interpolation_reference_cell
 type(interpolation_t)          :: interpolation_real_cell
contains
 public
 ! subroutine(this(inout)      :: class(cell_integrator_t),
 !            quadrature(in)   :: type(quadrature_t),
 !            reference_fe(in) :: class(reference_fe_t))
 procedure, non_overridable :: create => cell_integrator_create
 ! subroutine(this(inout) :: class(cell_integrator_t),
 !            cell_map(in)  :: type(cell_map_t))
 procedure, non_overridable :: update => cell_integrator_update
 ! subroutine(this(inout) :: class(cell_integrator_t))
 procedure, non_overridable :: free   => cell_integrator_free

 ! Evaluation of shape functions values
 procedure, non_overridable, private :: get_values_scalar => cell_integrator_get_values_scalar
 procedure, non_overridable, private :: get_values_vector => cell_integrator_get_values_vector
 procedure, non_overridable, private :: get_values_tensor => cell_integrator_get_values_tensor
 generic :: get_values => get_values_scalar, get_values_vector, get_values_tensor

 ... ! Evaluation of shape functions gradients
 ... ! Evaluation of shape functions hessians
 ... ! Evaluation of shape functions divergences
 ... ! Evaluation of shape functions curls
 ... ! Rest of TPBs to preserve encapsulation
end type cell_integrator_t
```

Listing 19. The `cell_integrator_t` data type.

An instance of cell_integrator_t is created from a quadrature rule (where the shape functions and their derivatives are to be evaluated) and a polymorphic reference_fe_t instance describing the reference FE space at hand; see interface of the create binding in Listing 19. During this stage, reference_fe creates the interpolation_reference_cell member variable of cell_integrator_t via create_interpolation; see Listing 16. It also clones interpolation_reference_cell into interpolation_real_cell, and copies the contents of the former into the latter. This lets cell_integrator_t to fulfill later on the pre-condition on the last dummy argument of apply_cell_map. The create binding also associates its polymorphic pointer reference_fe member variable to the reference_fe_t instance provided to it on input. This pointer is required later on by the update and get_* bindings (see discussion in the sequel).

```
1 subroutine cell_integrator_get_gradients_scalar (this, gradients)
2  implicit none
3  class(cell_integrator_t)                , intent(in)    :: this
4  type(vector_field_t),      allocatable, intent(inout) :: gradients(:,:)
5  call this%reference_fe%get_gradients(this%interpolation_real_cell,gradients)
6 end subroutine cell_integrator_get_gradients_scalar
```

Listing 20. The code implementing the get_gradients_scalar binding of cell_integrator_t ultimately relies on a deferred binding of reference_fe_t with the same name.

The update binding of cell_integrator_t simply invokes apply_cell_map on its polymorphic reference_fe member variable, using the instance of cell_map_t provided on input to update, and the two interpolation_t member variables as actual arguments, respectively; see Listings 18 and 19. It leaves the cell_integrator_t instance on which it is invoked in a state such that it is able to provide the services it was primarily designed for. These are offered through the get_values, get_gradients, get_divergences, get_curls, etc., *generic bindings*. We note that cell_integrator_t is designed such that it can handle either scalar, vector, or tensor-valued reference_fe_t instances (see Sect. 6.2). With this purpose in mind, each of the aforementioned generic bindings are overloaded with subroutines that have appropriate interfaces for these three types of FEs. For example, the subroutine overloading get_gradients in the case of scalar-valued FEs is declared and implemented as shown in Listing 20, with vector_field_t representing a d-dimensional rank-1 tensor; the interface of the one corresponding to vector-valued FEs only differs from the one above on the base type of the gradients allocatable array dummy argument, which is of base type tensor_field_t (i.e., data type representing a d-dimensional rank-2 tensor).

Let us remark some important points with respect to the subroutines overloading the generic bindings of cell_integrator_t. First, we note that the actual argument passed in place of, e.g., the gradients(:,:) dummy argument in Listing 20, is intended to be actually declared in code written by the user of FEMPAR. Provided that FEMPAR can support variable degree FEs on top of different triangulation cells (see Sect. 10), the allocatable attribute of the gradients(:,:) dummy argument not only unburdens the user from the complexity of having to (pre)allocate this array, but even from the one associated to variable degree FEs. For example, if on input, the size of gradients(:,:) is not sufficient to hold the data to be provided by the cell_integrator_t instance corresponding to the reference_fe_t on top of the current triangulation cell, then it can be re-allocated to the appropriate size. Second, this binding is designed such that all functions are evaluated at all quadrature points within a single call, justifying why the dummy argument has to be a rank-2 allocatable array.[21] At this point, let us note that all subroutines subject to consideration ultimately rely on (deferred bindings of) reference_fe_t; see, e.g., line 5 in Listing 20. We recall that reference_fe_t must mediate in any process that requires retrieving data from interpolation_t; see Sect. 8.2.

8.5 Cell Integration User Code Example

At this point of the discussion, we are already in position to show user code that evaluates the entries of the (current cell) local matrix for the Example 3.1 presented in Sect. 3.1. This code is sketched in Listing 21. This code would be bounded to a subclass of the discrete_integration_t abstract data type presented in Sect. 11.2 suitable for the Galerkin discretization of the Poisson problem.

[21] This represents another design decision in the seek of maximizing the granularity of the calls to deferred bindings for code efficiency reasons.

```
type(vector_field_t), allocatable :: shape_gradients(:,:)
real(rp)            , allocatable :: element_matrix(:,:)
... ! Declaration + variable initialization
! Loop over all cells K
... ! Update cell integration data structures to reflect the status corresponding to current cell
element_matrix = 0.0_rp
call cell_integrator%get_gradients(shape_gradients)
do q = 1, quadrature%get_num_quad_points()
   detJ_x_wq = cell_map%get_det_jacobian(q) * quadrature%get_weight(q)
   do b = 1, num_shape_functions
      do a = 1, num_shape_functions
         element_matrix(a,b) = element_matrix(a,b) + &
                               shape_gradients(a,q) * shape_gradients(b,q) * detJ_x_wq
      end do
   end do
end do
... ! Assemble element_matrix into global linear system coefficient matrix
! End Loop over all cells K
```

Listing 21. User-level code illustrating the usage of cell integration data structures in order to compute the element matrix for the Example 3.1 presented in Sect. 3.1.

The reader may note from Listing 21 that FEMPAR also offers an expression syntax that lets its users code weak forms in a way that resembles their mathematical expression. The user is in charge of explicitly writing the expression of the numerical integration in the reference cell, i.e., of explicitly implementing the quadrature point summation (loop) and handling the determinant of the Jacobian and the quadrature point weighting in (12). However, the evaluation of the shape function and their gradients, curls, etc., at the quadrature points in the physical space (e.g., expressions (13) and (14)) are completely hidden to the user. This can be achieved using a feature of modern programming languages called *operator overloading*. (We refer to [67] for a detailed exposition of this mechanism in Fortran2003.) Common (contraction) operations among tensors are provided by means of overloaded intrinsic and library-defined operators. For example, the `operator(*)` generic interface (corresponding to the * intrinsic operator) has to be overloaded with the single contraction of rank-1 tensors, and the multiplication of a rank-1 tensor by a scalar to let our code compile. A crucial design requirement in the seek of code efficiency is that no dynamic memory allocation/deallocation is involved as the partial evaluation of sub-expressions proceeds (in the order dictated by operator associativity and priority rules in Fortran). In order to fulfill this requirement, the data types representing vectors and tensors are declared such that their entries are stored in an array member variable *of size known at compilation time*. This size is stored in the library-level parameter constant SPACE_DIM, defined as the maximum number of space dimensions of the physical space in which the physical problem is posed. By default, FEMPAR is prepared to deal with 3D simulations, but the code is written such that a 2D simulation might also be performed if SPACE_DIM is equal to 3, at the price of extra storage and computation.[22] Higher dimensional problems could be considered by compiling FEMPAR with a larger value for SPACE_DIM. Apart from avoiding dynamic memory allocation/deallocation during the evaluation of weak forms, this solution has the following advantages: (1) there is no need to explicitly have the number of dimensions as a member variable of the data types representing vectors and tensors; (2) the limits of the loops implementing tensor contraction operations are known at compilation time, enabling compiler optimizations. We finally stress that we preferred this solution over the usage of Fortran2003 parameterized data types [67] due to the lack of support of this feature in some of the most popular compilers widely available on high-end computing environments.

9 Evaluation of Facet Integrals

This section covers the data types (and their interactions) in which the evaluation of integrals over the facets of the triangulation is grounded on. The integration of facet-wise matrices and vectors (see, e.g., (23)) involves the evaluation of shape functions and gradients of the neighbouring cells at the quadrature points within the facet in the physical space and the Jacobian of the facet map at the reference space. As described in Sect. 8, the former quantities are computed at every neighbouring cell from their values at the reference space and the Jacobian of the cell mapping. The evaluation of interior facet also requires the computation of the permutation $\Pi(\text{gp})$ (see (25)) provided that the coordinate systems of the cells surrounding the facet might not be aligned in physical space.

[22] In fact, 2D problems for PDEs that involve curl operators require SPACE_DIM to be equal to 3.

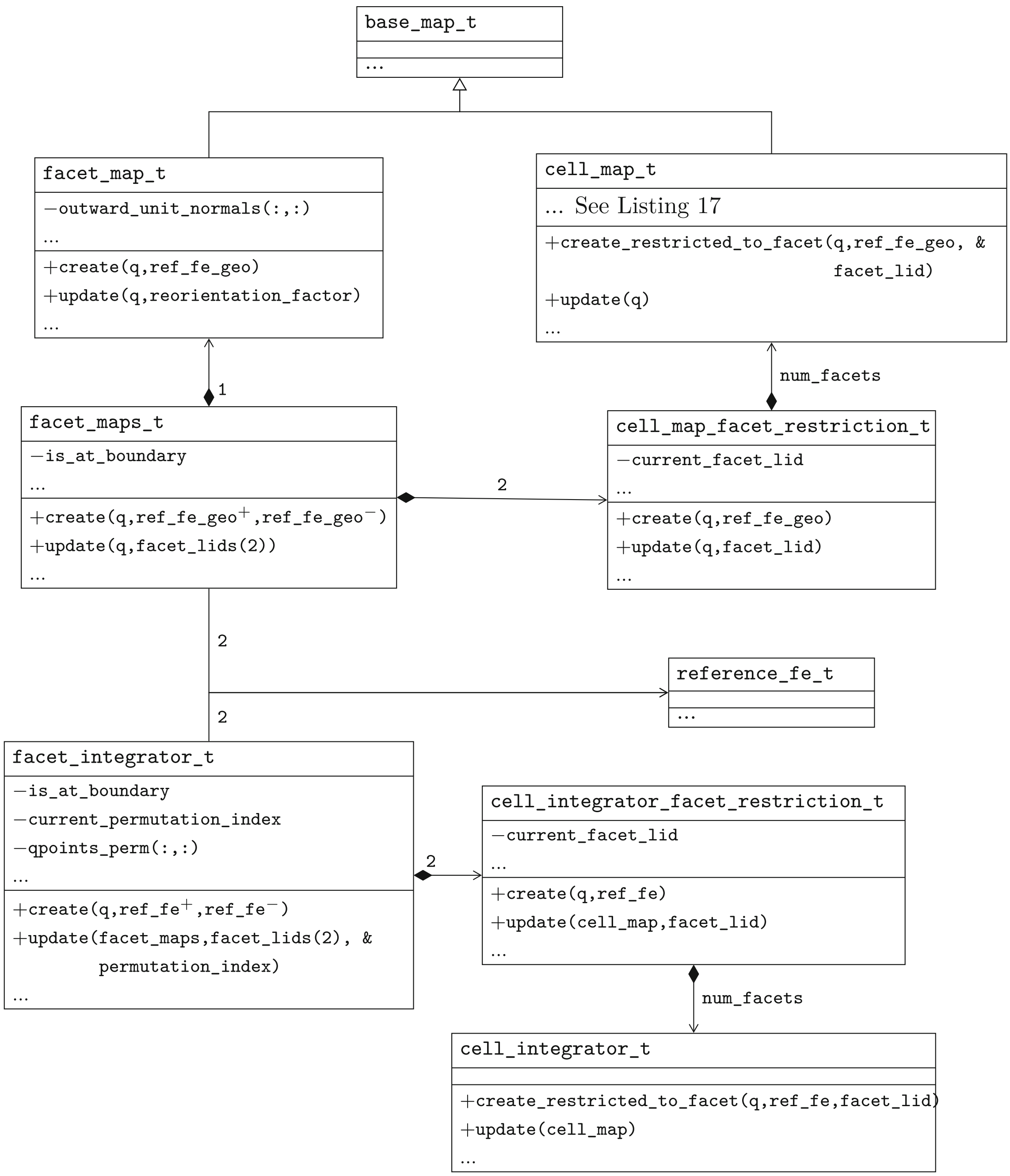


Fig. 8 UML class diagram of the data types on which the numerical evaluation of facet integrals is grounded on

In FEMPAR the assembly process of the global linear system underlying the discrete weak problem (20) involves two loops, over all cells and facets, respectively. In the former loop, a cell-wise matrix $\mathbf{A}^K$ and vector $\mathbf{f}^K$ are computed per each cell. These hold the partial contributions of the cell to the corresponding entries of the global coefficient matrix and right-hand side vector, respectively. The data structures involved in their efficient computation have been already covered in Sect. 8. In the latter loop, and assuming that we are sitting on an interior facet $F \in \mathcal{F}_h^{\Omega}$, four facet-wise matrices, namely $\mathbf{A}^F_{K^+K^+}$, $\mathbf{A}^F_{K^+K^-}$, $\mathbf{A}^F_{K^-K^+}$, and $\mathbf{A}^F_{K^-K^-}$ are computed (see Sect. 3.12).

We depict in Fig. 8 a complete UML class diagram of the data types involved in the evaluation of facet integrals and their relationships. The data types the user has to ultimately interact with are `quadrature_t`, which holds the facet quadrature points and weights, `facet_maps_t`, which handles (i.e., stores, updates, provides) all the geometrical related data of the facet and neighbouring cells K^+ and K^-, and, finally, `facet_integrator_t`, which stores and updates shape function values and first derivatives, and provides shape function values, gradients, curls, etc., of K^+ and K^- evaluated at facet quadrature points in real space. The rest of data types in Fig. 8 are auxiliary data types, not exposed to the user, which aid the latter two in the implementation of their corresponding services. The reader might readily observe in Fig. 8 that our software design is such that the data types that provide support to the evaluation of cell integrals, i.e., `quadrature_t`, `cell_map_t`, and `cell_integrator_t` (see Sect. 8), can be re-used to a large extent for the evaluation of facet integrals. As we will see in the rest of the section, some of the methods to be invoked in order to control their respective life cycles in the context of facet integrals are nevertheless different from the ones to be invoked in the context of cell integrals; see, e.g., the signature of the `create_restricted_to_facet` binding of `cell_integrator_t` in Fig. 8 compared to that of its `create` binding in Listing 19.

9.1 Numerical Quadrature

The data type `quadrature_t` is designed to be a placeholder for the *facet quadrature* points $\hat{\boldsymbol{x}}_q$ and its associated weights w_q. However, the code that ultimately decides how to distribute $\hat{\boldsymbol{x}}_q$ over the reference facet $\hat{F}$ coordinate system, and set up w_q, is bounded to `reference_fe_t`, in particular through the deferred binding with interface shown in Listing 22. We refer to Sect. 8.1 for the rationale underlying the `degree` optional dummy argument of this deferred binding.

9.2 Geometrical Mappings

The `facet_maps_t` data type in Fig. 8 handles the geometrical facet mapping and the two geometrical cell mappings. The facet mapping is represented by `facet_map_t`, whereas the cell mappings by `cell_map_t`; see Sects. 9.2.1 and 9.2.2, respectively.

9.2.1 Facet Mapping

As illustrated in Fig. 8, `facet_maps_t` is composed, among others, of a single instance of type `facet_map_t`. The member variables (and associated code) that are common to `facet_map_t` and `cell_map_t` are factored into a superclass `base_map_t` (see Listing 17). `facet_map_t` handles all data related to the facet map $\boldsymbol{\Phi}_F$, including the facet outward unit normals (see Fig. 8). An extra 2-rank real allocatable array member variable, `outward_unit_normals(:,:)`, stores the facet outward unit normals (with respect to K^+ by convention) evaluated at facet quadrature points in real space, as required by (25); $\boldsymbol{n}^-(\boldsymbol{x}_{\mathrm{gp}})$ can be simply obtained as $\boldsymbol{n}^-(\boldsymbol{x}_{\mathrm{gp}}) = -\boldsymbol{n}^+(\boldsymbol{x}_{\mathrm{gp}})$.

Let us now see how `facet_maps_t` controls the life cycle of its `facet_map_t` instance. The `create` binding of `facet_map_t` takes a `quadrature_t` instance with the facet quadrature points. $\boldsymbol{J}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}})$ and $|\boldsymbol{J}_F(\hat{\boldsymbol{x}}_{\mathrm{gp}})|$ are evaluated at these quadrature points and stored in the `jacobian` and `det_jacobian` member variables, which are allocated during a call to this binding together with `outward_unit_normals(:,:)`. Apart from a `quadrature_t` instance, `facet_map_t` also requires a description of the discrete, lower dimensional space of functions on top of the reference facet $\hat{F}$ to which $\boldsymbol{\Phi}_F$ belongs. The `ref_fe_geo` dummy argument of `create`, of polymorphic type `reference_fe_t`, is provided for this purpose; in particular, `facet_maps_t` sends the `reference_fe_t` on top of K^+ as an actual argument to the `ref_fe_geo` dummy argument in order to comply with the above described convention for the normals. The `interpolation_t` member variable of

```
abstract interface
 ...
 subroutine reference_fe_create_facet_quadrature ( this, quadrature, degree  )
  class(reference_fe_t)          , intent(in)    :: this
  type(quadrature_t)             , intent(inout) :: quadrature
  integer(ip)          optional , intent(in)    :: degree
 end subroutine reference_fe_create_facet_quadrature
 ...
end interface
```

Listing 22. The interface of the `create_facet_quadrature` deferred binding of `reference_fe_t`.

`facet_map_t` (see Listing 17) is used by `ref_fe_geo` to exchange with `facet_map_t` the shape function values and their derivatives. To this end, `reference_fe_t` is equipped with the `create_facet_interpolation` deferred binding (see its signature in Listing 23) that computes these quantities on top of the reference facet $\hat{F}$. with α being either $+$ or $-$. Let us thus refer to these instances as `cell_map_facet_restriction`$^\alpha$, and to the polymorphic `reference_fe_t` instances on top of K^α as `ref_fe_geo`$^\alpha$. In turn, `cell_map_facet_restriction`$^\alpha$ are composed by as many `cell_map_t` instances as facets in K^α. Provided that an actual facet F can

```fortran
abstract interface
 ...
 subroutine reference_fe_create_facet_interpolation ( this, quadrature, &
                                                       reference_facet_interpolation )
   class(reference_fe_t) , intent(in)    :: this
   type(quadrature_t)    , intent(in)    :: quadrature
   type(interpolation_t) , intent(inout) :: reference_facet_interpolation
 end subroutine reference_fe_create_facet_interpolation
 ...
end interface
```

Listing 23. The signature of the `create_facet_interpolation` deferred binding of `reference_fe_t`.

The `update` binding of `facet_map_t` is intended to be called once per facet loop iteration, i.e., once per each facet of the triangulation. A pre-condition of this binding is that the `nodes_coordinates(:)` scratch member array of `facet_map_t` (see Listing 17) has been loaded with the coordinates in real space of the nodes that lay on the the facet.[23] The `update` binding takes as input dummy arguments a `quadrature_t` instance and the real parameter `reorientation_factor` in order to adjust the sign of the facet normals (see (26)). Within `update`, `quadrature_points_coordinates(:)` and `jacobian(:,:,:)` can be easily computed from the basis shape functions and their first derivatives, respectively. On the other hand, `det_jacobian(:)` and `outward_unit_normals(:,:)` can be computed from `jacobian(:,:,:)`. The former as stated in (24), while the latter as in (26).

9.2.2 Neighbouring Cells Mappings

The `facet_maps_t` data type is also composed by two instances of type `cell_map_facet_restriction_t`; see Fig. 8. These instances handle all data related to $\mathbf{\Phi}_{K^\alpha}$, *potentially* have local identifier F^α in K^α within the range $F^\alpha = 1, \ldots, \text{num\_ facets}(\text{K}^\alpha)$, having as many `cell_map_t` instances as facets per surrounding cell let us hold and (pre)calculate within these instances the result of evaluating the $\hat{K}^\alpha$ shape functions and their derivatives at the facet quadrature points for all facets in the reference system. To this end, the `create` binding of `cell_map_facet_restriction`$^\alpha$ is invoked (from the one corresponding to `facet_maps_t`) with the facet quadrature `q` and `ref_fe_geo`$^\alpha$ as input actual arguments. It then walks over all possible local facet identifiers in the corresponding cell, and for each local facet identifier, invokes a specialized version of the `create` binding of the corresponding `cell_map_t` instance, named `create_restricted_to_facet` (that additionally requires the local facet identifier); see Fig. 8. The `reference_fe_t` is ultimately responsible to exchange this sort of data with `cell_map_t`. This service is in particular provided by the `create_interpolation_restricted_to_facet` deferred binding of `reference_fe_t`, with signature defined in Listing 24.

```fortran
abstract interface
 ...
 subroutine ..._create_interpolation_restricted_to_facet ( this, quadrature, facet_lid, &
                                                            ref_cell_interp_restricted_to_facet )
   class(reference_fe_t), intent(in)    :: this
   type(quadrature_t)   , intent(in)    :: quadrature
   integer(ip)          , intent(in)    :: facet_lid
   type(interpolation_t), intent(inout) :: ref_cell_interp_restricted_to_facet
 end subroutine ..._create_interpolation_restricted_to_facet
 ...
end interface
```

Listing 24. The signature of the `create_interpolation_restricted_to_facet` deferred binding of `reference_fe_t`.

[23] This can be easily fulfilled by calling the `get_nodes_coordinates` binding of `vef_iterator_t` in Listing 11.

As seen so far, the create binding of facet_maps_t is designed to be called right before the actual loop over all triangulation facets, and it sets up all the scratch data. It does so by covering all possible scenarios corresponding to potential values of local facet identifiers within the two surrounding cells (even if some of these scenarios are not actually exposed in the triangulation). The update binding of facet_maps_t, however, is intended to be called sitting on a particular facet F of the triangulation, and it has to only update those two cell_map_t instances within cell_map_facet_restriction$^\alpha$ corresponding to the particular scenario at hand, i.e., to the particular combination of local facet identifiers F^+ and F^- of the facet on which it is being updated. To this end, the update binding of facet_maps_t receives these local identifiers in facet_lids (see Fig. 8) and then calls the update binding of cell_map_facet_restriction$^+$ and cell_map_facet_restriction$^-$ with facet_lid=facet_lid(1) and facet_lid=facet_lid(2), respectively. The update binding of cell_map_facet_restriction_t picks up the cell_map_t corresponding to facet_lid and invokes the update binding of the latter. We stress that no specialized version of this binding is required in the context of facet integration, i.e., the same version discussed in Sect. 8.4 for cell integration can be re-used here.[24] During the update process, cell_map_facet_restriction_t also registers in its current_facet_lid private member variable, the value supplied to the facet_lid dummy argument. This lets facet_maps_t to extract later on from cell_map_facet_restriction $^\alpha$ the *updated* cell_map_t instances; see discussion of facet_integrator_t in the sequel.

9.3 Evaluation of Shape Functions in the Physical Space

The last data type that remains to be covered is facet_integrator_t; see Fig. 8. This data type is the counterpart of cell_integrator_t (see Sect. 8.4) for the case of facet integrals. In particular, it stores and updates shape function values and derivatives, and provides the values, gradients, curls, and divergences of the respective fields for both K^+ and K^- evaluated at facet quadrature points in real space. As can be observed from Fig. 8, its overall design is very close to the one of facet_maps_t, with cell_integrator_facet_restriction_t and the cell_integrator_t instances it is composed of, playing the role of its counterparts in the scope of facet_maps_t (i.e., cell_map_facet_restriction_t and cell_map_t, respectively). There are, however, two major differences among these two. First, facet_integrator_t deals with (e.g., it is created from) the two polymorphic reference_fe_t instances (see ref_fe$^\alpha$ dummy arguments of its create binding in Fig. 8) on which the global FE spaces of functions $\mathcal{X}_h, \mathcal{Y}_h$ are grounded on. For example, the create binding of cell_integration_facet_restriction$^+$ invokes the create_restricted_to_facet binding of the cell_integrator_t for all facets F^+ within K^+. The latter computes at a given facet $\hat{\phi}^a_{K^+}(\hat{\boldsymbol{x}}^+_{\mathrm{gp}})$, $\nabla\hat{\phi}^a_{K^+}(\hat{\boldsymbol{x}}^+_{\mathrm{gp}})$ through the deferred binding create_interpolation...to_facet of reference_fe_t presented in Listing 24. Second, facet_integrator_t has to unburden the user from the complexity underlying the fact that the coordinate systems of K^+ and K^- might not be aligned in real space. To this end, it is equipped with a private lookup permutation table, called qpoints_perm(:,:) in Fig. 8, that lets it translate facet quadrature points identifiers from the local numbering space of K^+ into the one of K^-. This table is allocated and filled during the create binding of facet_integrator_t, in particular by reference_fe_t through a deferred binding called fill_qpoints_permutations. Given the facet quadrature identifier gp and the facet permutation index pi (see Sect. 3.16), qpoints_perm(gp,pi) stores the value of $\Pi(\mathrm{gp})$ (see (25)). The permutation index is stored within the current_permutation_index of facet_integrator_t, extracted from the permutation_index dummy argument of the update binding. In turn, this parameter is extracted from the array facet_permutation_indices(:) of fe_space_t in Listing 27 (see Sect. 10). We note that for n-simplices, we consider a renumbering such that all facets have the same orientation on both cells that share it, as commented in Sect. 3.16. In this case, fill_qpoints_permutations fills the table with the identity permutation in all columns. We note that the re-orientation of the n-simplices can lead to mappings $\boldsymbol{\Phi}_K$ such that $|\boldsymbol{J}_K| < 0$, but this is not a problem as soon as one takes its absolute value, e.g., in (12).

9.4 Facet Integration User Code Example

In order to grasp how the data structures covered so far are actually used together in practice, the Fortran pseudocode snippet at Listing 25 shows user's space code in charge of

[24] We note that, as in Sect. 8.3, the nodes_coordinates(:) member variable of these two cell_map_t instances has to be loaded with the coordinates in physical space of the geometry nodes of the two cells surrounding the facet.

```
1  real(rp)              , allocatable :: shape_values_K^+(:,:),shape_values_K^-(:,:)
2  type(vector_field_t), allocatable :: shape_gradients_K^+(:,:), shape_gradients_K^-(:,:)
3  real(rp)              , allocatable :: facet_matrix_K^+_K^-(:,:)
4  ... ! Declaration of the rest of facet_matrices
5  type(vector_field_t)                :: outward_unit_normals(2)
6  integer(ip)           , parameter   :: K^+ = 1, K^- = 2
7  ... ! Declaration + variable initialization
8  ! Loop over all facets F
9  ... ! Update facet integration data structures to reflect the status corresponding to current facet
10 if (current_facet_interior) then
11  facet_matrix_K^+_K^- = 0.0_rp
12  ... ! Initialize the rest of facet_matrices
13  call facet_map%get_outward_unit_normals(outward_unit_normals)
14  call facet_integrator%get_values(K^-,shape_values_K^-)
15  call facet_integrator%get_gradients(K^+,shape_gradients_K^+)
16  ... ! Get values and gradients required to evaluate the rest of facet_matrices
17  do q = 1, facet_quadrature%get_num_quad_points()
18   |J_F|_x_wq = facet_map%get_|J_F|(q) * facet_quadrature%get_weight(q)
19   ... ! Compute facet_matrix_K^+_K^+(:,:)
20   ! Compute facet_matrix_K^+_K^-(:,:)
21   do b = 1, num_shape_functions_K^-
22    do a = 1, num_shape_functions_K^+
23     facet_matrix_K^+_K^-(a,b) = facet_matrix_K^+_K^-(a,b) + &
24     (0.5*(shape_values_K^-(b,q)*outward_unit_normals(K^-)*shape_gradients_K^+(a,q))+...)*|J_F|_x_wq
25    end do
26   end do
27   ... ! Compute facet_matrix_K^-_K^+(:,:)
28   ... ! Compute facet_matrix_K^-_K^-(:,:)
29  end do
30 else ! Current facet lays on the boundary
31    ...
32 end if
33 ... ! Assemble facet_matrices into global linear system coefficient matrix
34 ! End Loop over all facets F
```

Listing 25. User-level pseudocode illustrating the usage of facet integration data structures in order to compute the first integral in (22) for each interior facet in a loop over all facets.

evaluating the first integral in (22) for each interior facet in a loop over all facets. It would be bounded to a subclass of the `discrete_integration_t` abstract data type presented in Sect. 11.2 suitable for the non-conforming DG discretization of the Poisson problem.

There are a pair of worth noting remarks about Listing 25. First, the call to the `get_values()` binding of `facet_integrator_t` in Line 14 already returns the permuted K^- shape function values, i.e., `shape_values_` K^-`(b,gp)` actually stores $\phi^b_{K^-}(\boldsymbol{x}^-_{\Pi(\mathrm{gp})})$. Second, it is the so-called `fe_space_t` abstraction (to be covered in Sect. 10) the one in charge of creating the facet integration data structures on loop initialization and to update them at each facet loop iteration (see Line 9). Therefore, the user does not actually directly deals with all the data types bindings and their interactions illustrated in Fig. 8. In this example, it becomes evident that facet-loop based integration is very convenient for the implementation of DG methods, since it very much resembles the blackboard expressions (see, e.g., (20)).

9.5 Change-of-Basis Implementation in a `reference_fe_t` Subclass

In this section, we provide a detailed presentation of how the change-of-basis required to compute the shape functions basis is implemented in a `reference_fe_t` subclass. In particular, we show the implementation for the Raviart-Thomas div-conforming FE on n-cubes in Sect. 3.5 (see also Sect. 3.9 for details). The pre-basis, e.g., $\mathcal{Q}_{(k+1,k,k)} \times \mathcal{Q}_{(k,k+1,k)} \times \mathcal{Q}_{(k,k,k+1)}$ in 3D, has to be generated before this subroutine is called; see, e.g., the evaluation of the pre-basis in Line 31 of Listing 26.

```
1  subroutine hex_raviart_thomas_reference_fe_change_basis(this)
2    class(hex_raviart_thomas_reference_fe_t), intent(inout) ::  this
3    type(hex_lagrangian_reference_fe_t) :: d_1_fe
4    type(quadrature_t)                  :: d_1_quadrature
5    type(interpolation_t)               :: d_1_interpolation, facet_interpolation
6    real(rp)                            :: shape_test
7    type(vector_field_t)                :: normal
8    type(hex_lagrangian_reference_fe_t) :: d_fe_geo
9    type(vector_field_t)                :: v_shape_trial
10   integer(ip)                         :: ishape, jshape, qpoint
11   real(rp)                            :: factor
12   integer(ip)                         :: facet_id
13
14   ... ! Allocate this%change_basis_matrix (this%num_shape_functions,this%num_shape_functions)
15   ! Create a d-1-dim scalar lagrangian reference FE of order k
16   call d_1_fe%create(topology          = this%get_topology(), &
17        number_dimensions = this%number_dimensions-1, order = this%order, &
18        field_type        = field_type_scalar, conformity = .true. )
19   ! Create a d-dim scalar lagrangian reference FE of order 1 (geometry)
20   call d_fe_geo%create(topology          = this%get_topology(), &
21        number_dimensions = this%number_dimensions, order = 1, &
22        field_type        = field_type_scalar, conformity = .true. )
23   ! Create a d-1 dimension quadrature from RT reference FE
24   call this%create_facet_quadrature( d_1_quadrature )
25   call d_1_fe%create_interpolation( d_1_quadrature, d_1_interpolation )
26   ! Initialize change of basis matrix
27   this%change_basis_matrix = 0.0_rp
28   d = 0
29   do facet_id = this%get_first_facet_id(), this%get_first_facet_id() + this%get_number_facets()-1
30      ! Map quadrature points in reference facet to cell facet and evaluate shape functions
31      call this%create_facet_interpolation ( facet_id - d_fe_geo%get_first_facet_id()+1, &
32                                             d_1_quadrature, facet_interpolation )
33      ! Integrate boundary moments int_Facet(u.n q), q \in Q_k
34      do qpoint = 1, d_1_quadrature%number_quadrature_points
35         factor = d_1_quadrature%get_weight(qpoint) / ( 2.0 ** d_1_fe%get_number_dimensions() )
36         ... ! Compute normal such that (for oriented meshes) two elements sharing a facet have a
37         ! moment determined with the same normal, in order to have conformity
38         ! Integrate moments and assemble in this%change_basis_matrix
39         do ishape=1, d_1_interpolation%number_shape_functions
40            call d_1_fe%get_value(d_1_interpolation, ishape, qpoint, shape_test)
41            do jshape=1, facet_interpolation%number_shape_functions
42               call this%get_value(facet_interpolation, jshape, qpoint, v_shape_trial)
43               this%change_basis_matrix(d+ishape,jshape) += &
44                    shape_test * v_shape_trial * normal * factor
45            end do
46         end do
47      end do
48      d = d + d_1_interpolation%number_shape_functions
49   end do
50   ... ! Compute number interior moments shape functions: Qk-1,k,k x Qk,k-1,k x Qk,k,k-1
51   ! Compute interior moments (using same approach as above for facet moments)
52   ! and assemble them in this%change_basis_matrix
53   ! Invert change_basis_matrix
54   call this%invert_change_basis_matrix()
55   ... ! Transform type(list_t) member variables of this to reflect change of basis
56   ... ! Deallocate dynamic memory
57 end subroutine hex_raviart_thomas_reference_fe_change_basis
```

Listing 26. Implementation of the change-of-basis required for Raviart-Thomas divconforming FEs on n-cubes, following the procedure presented in Sect. 3.5.

We also present how to compute the boundary moments in (16) in Listing 26; interior moments are simpler and omitted for the sake of brevity. The implementation of the boundary moments requires: (1) to create the `reference_fe_t` that implements $[\mathcal{Q}_{kI}]^{d-1}$ in Line 16, (2) a facet quadrature on the reference facet in Line 24, and (3) the evaluation of the reference FE in the quadrature points in the `interpolation_t` in Line 25. We also require a Lagrangian (first order) FE that represents the geometry in Line 20. Next, we loop over all the facets of the cell and compute the values of the shape functions of the cell in the facet quadrature, stored in the `interpolation_t` instance in Line 31. With all these ingredients, we can compute the boundary moments for the pre-basis functions (see line 43) and assemble them in the change-of-basis matrix. After doing the same for interior moments, we just need to invert the change-of-basis matrix in Line 54. At this point, we have the shape functions basis as a linear combination of pre-basis functions. Thus, when one calls the `fill_interpolation` binding of the corresponding reference FE, it creates the pre-basis `interpolation_t` instance and next applies the change-of-basis matrix to compute the one for the shape functions basis, i.e., the placeholder where the evaluation of the shape functions and its derivatives (at the set of quadrature points for which the interpolation has been created) are stored. We note that the ownership of DOFs also changes in this process. The boundary moments (integrals of functions on facets) belong to the corresponding facet, whereas interior moments belong to the cell. Vertices and edges do not have DOFs in this case. The definition of the ownership is skipped for brevity.

10 Integration and Global DOF Handling: The `fe_space_t` Abstraction

In this section, we introduce a software abstraction, referred to as `fe_space_t`, which represents (in the most general scenario) the mathematical concept of a global FE space $\mathcal{X}_h = \mathcal{X}_h^1 \times \ldots \times \mathcal{X}_h^n$ obtained by means of the Cartesian product of global FE spaces $\mathcal{X}_h^i$ corresponding to each of the $i = 1, \ldots, n_{\text{field}}$ field unknowns involved in a system of PDEs; see Sects. 3.6 and 3.11. Each $\mathcal{X}_h^i$ is described as a combination of: (1) an approximation Ω_h of the physical domain Ω provided by `triangulation_t`, i.e., a mesh-like container for the cells on which Ω_h is partitioned, their boundary lower-dimensional objects, and their adjacency relationships; see Sect. 7; (2) a description of the n_{field} reference FEs associated to each triangulation cell grounded on `reference_fe_t`; see Sect. 6.

These two basic building blocks equip `fe_space_t` with the tools required to provide the following two crucial services.[25] On the one hand, it is in charge of handling (i.e., generating, storing, fetching) a *global* enumeration of the DOFs corresponding to each $\mathcal{X}_h^i$ taking into account the notion of conformity; see e.g., Sects. 3.6 and 6.2. On the other hand, it handles the data structures that are required to evaluate integrals over cells and facets (see Sects. 8 and 9, respectively). In particular, it judiciously sets up them, and orchestrates their respective life cycles and interactions, while unburdening the user (to a large extent) from the complexity (among others) inherent to high order FEs.

The OO design of `fe_space_t` (as the one of many other data types in FEMPAR, e.g., `triangulation_t`) strongly strives to preserve encapsulation and data hiding while still storing and accessing data efficiently (i.e., in a way that leverages data locality for the efficient exploitation of modern computer memory architectures). The user-friendly view of `fe_space_t` is implicitly (re)constructed by the data types (associated interfaces and interactions) that will be covered in Sect. 10.2. We now move on the approach that we follow for the internals of `fe_space_t`.

10.1 The Internal Organization of `fe_space_t`

In this section, we *sketch* how the internals of `fe_space_t` are organized in order to efficiently deliver the two services outlined above. For simplicity, we restrict ourselves to a simplified version of `fe_space_t` that, to a large extent, captures the spirit of its actual counterpart in FEMPAR. The declaration of this simplified data type is shown in Listing 27.[26]

[25] We stress, however, that the full set of services provided by `fe_space_t` is not actually restricted to only these two.

[26] We note that `fe_space_t` is not actually in FEMPAR. It is a whole data type hierarchy rooted at `base_fe_space_t`, not included here for simplicity. Within this hierarchy, we have, e.g., FE space concretizations suitable for either serial or parallel distributed-memory environments. The one shown in the listing very much resembles `serial_fe_space_t`.

```
type :: fe_space_t
  private
  integer(ip)                            :: num_fields

  ! Container of polymorphic reference_fe_t instances (See Section 6)
  type(p_reference_fe_t)    , allocatable :: reference_fes(:)

  ! Mapping of (field_id,cell_id) pairs to reference_fes(:)
  integer(ip)               , allocatable :: field_cell_to_ref_fes(:,:)

  ! (Polymorphic) pointer to a triangulation it was created from (See Section 7)
  class(triangulation_t)    , pointer     :: triangulation => NULL()

  ! Descriptor of the block layout selected for the PDE system at hand (See Section 10.3)
  type(block_layout_t)      , pointer     :: block_layout => NULL()

  ! Global DoF enumeration-related data
  integer(ip)               , allocatable :: ptr_dofs_x_field_cell(:,:)
  integer(ip)               , allocatable :: lst_dofs_gids(:)
  integer(ip)               , allocatable :: num_dofs_x_field(:)

  ! Cell-related integration member variables (See Section 8)
  integer(ip)               , allocatable :: cell_quadratures_degree(:)
  type(quadrature_t)        , allocatable :: cell_quadratures(:)
  type(hash_table_t)                      :: cell_quadratures_position
  type(cell_map_t)          , allocatable :: cell_maps(:)
  type(hash_table_t)                      :: cell_maps_position
  type(cell_integrator_t)   , allocatable :: cell_integrators(:)
  type(hash_table_t)                      :: cell_integrators_position

  ! Facet-related integration member variables (See Section 9)
  integer(ip)               , allocatable  :: facet_quadratures_degree(:)
  type(quadrature_t)        , allocatable  :: facet_quadratures(:)
  type(hash_table_t)                       :: facet_quadratures_position
  type(facet_maps_t)         , allocatable :: facet_maps(:)
  type(hash_table_t)                       :: facet_maps_position
  type(facet_integrator_t)   , allocatable :: facet_integrators(:)
  type(hash_table_t)                       :: facet_integrators_position

  ! Member variables to provide support to implementation of fe_facet_iterator_t (See Section 10.2)
  integer(ip)               , allocatable :: facet_gids(:)
  integer(ip)               , allocatable :: facet_permutation_index(:) ! (See Section 3.16)


  ! Strong imposition of boundary conditions-related data (See Section 10.4)
  integer(ip)                             :: num_fixed_dof_values

  ... ! Remaining accelerator look-up tables, scratch data, other member variables
      ! reflecting low-level implementations details, member variables providing support
      ! to other services of fe_space_t, etc.
contains
  public
  ! A specialized overload of the generic create binding where a single cell topology is assumed,
  ! and each global space is ground on the same reference_fe_t on all cells. The latter might be
  ! different among fields, as e.g., in Q2xQ1 inf-sup FEs for the Navier-Stokes problem
  procedure, private :: fe_space_create_same_reference_fes_on_all_cells
  ! The most general overload of the generic create binding, i.e., mixed cell topologies, and
  ! (potentially) different per-field and per-cell reference_fe_t instances
  procedure, private :: fe_space_create_arbitrary_per_cell_reference_fes
  generic   :: create => fe_space_create_same_reference_fes_on_all_cells, &
                         fe_space_create_arbitrary_per_cell_reference_fes, ...
  procedure :: generate_global_dof_numbering => fe_space_generate_global_dof_numbering
  procedure :: set_up_cell_integration       => fe_space_set_up_cell_integration
  procedure :: set_up_facet_integration      => fe_space_set_up_facet_integration
  procedure :: free                          => fe_space_free
  ... ! TBPs in charge of providing other fe_space_t services
  ... ! TBPs in charge of creating iterators on fe_space_t objects (See Section 10.2)
  ... ! Full set of getter TBPs to preserve data hiding and encapsulation
  ... ! Comprehensive set of private TBPs providing support to the
      ! implementation of the public TBPs above. These are hierarchically organized into
      ! short, auxiliary re-usable subroutines/functions for code readability purposes
end type fe_space_t
```

Listing 27. The internals of `fe_space_t` and a selected set of its bindings.

A collection of `reference_fe_t` polymorphic instances is stored in the `reference_fes(:)` array. These instances are uniquely identified (within the local scope of `fe_space_t`) by their position in this array. The global FE space corresponding to a given field, with identifier `f_id` in the range $1, \ldots,$ `num_fields` (with num_fields equal to n_{field} above), is described by: (1) the triangulation member variable (the rationale underlying it being polymorphic is made clear in Sect. 10.2; (2) its restriction to each cell provided by the reference FE space defined by the reference_fe_t instance with identifier field_cell_to_ref_fes(f_id,c_id) in the collection; c_id is assumed to be a positive integer in $1,\ldots,$ **triangulation%get_num_cells**() that uniquely identifies each cell.

The member variables used to handle the global DOF numbering are encompassed within Lines 18–27 of Listing 27. The global DOF identifiers are stored cell-wise, and field-wise within each cell, in the lst_dofs_gids(:) array, which is in turn (indirectly) addressed by the ptr_dofs_x_fe(:,:) array. In particular, the ones assigned to the local nodes related to field f_id on cell c_id start and end in position ptr_dofs_x_fe(f_id,c_id) and ptr_dofs_x_fe(f_id+1,c_id)-1 of lst_dofs_gids(:), respectively, if **f_id** < **num_fields**, and in position ptr_dofs_x_fe(f_id,c_id) and ptr_dofs_x_fe(1,c_id+1), respectively, if **f_id** = **num_fields**. The number of DOFs of the global FE space corresponding to each field (excluding those that are subject to strong boundary conditions) is stored in the num_dofs_x_field(:) array.

The member variable in Line 15 stores a reference to a data type that describes the block layout *currently selected* (i.e., it can be changed on demand) for the global matrix and right-hand side vector of the linear system (or a sequence of them) required for the solution of the PDE system at hand. The role of block_layout_t in the global DOF numbering generation process will be illustrated in Sect. 10.3.

The data structures that let fe_space_t handle the evaluation of cell integrals are declared in Lines 23–29 of Listing 27. The set_up_cell_integration binding sets up them. The method is intended to be called by the user's program right before any cell integration loop. It ensures that any (scratch) data that can be computed on its final form in the reference cell is pre-computed *for any of the triangulation cells* while minimizing the number of integration data structures required for the particular scenario at hand. To this end, fe_space_t is equipped with three array containers of quadrature_t, cell_map_t and cell_integrator_t objects (see Lines 24, 26, and 28, respectively), which are indirectly addressed by the hash_table_t member variables with corresponding names.[27] This is required because fe_space_t supports, e.g., non-conforming FE spaces with variable order per cell. A *unique identifier* (dynamically generated within the scope of fe_space_t) is assigned to each of the integration objects that must be created. The hash_table_t instances let fe_space_t transform these unique identifiers into container array positions from which the integration objects can be fetched.

The set_up_cell_integration method loops over all cells. Sitting on a cell, it determines an appropriate quadrature to be used on that cell and its associated unique identifier. (See discussion in the next paragraph for more details.) If this quadrature has not been generated yet (i.e., if the hash table lookup fails), then a new quadrature is created on the next free position of the cells_quadratures(:) array container, and a new identifier-position pair is inserted into the hash table. Otherwise, the quadrature is fetched from this array. The same process is repeated for the cell_map_t and cell_integrator_t instances. The former ones are uniquely determined by the combination of the unique identifier quadrature_t just created/fetched and that of the reference_fe_t instance on top of the current cell (see Sect. 7). On the other hand, a cell_integrator_t instance has to be associated to each field within the current cell; the cell_integrator_t instance corresponding to a field is uniquely determined by the unique identifier of the quadrature_t just created/fetched and the one of the reference_fe_t associated to that field (see Sect. 8.4). Therefore, the unique identifiers of the cell_map_t and cell_integrator_t instances required for the evaluation of cell integrals over the current cell can be easily determined combining the ones corresponding to the instances from which they are created. We recall that the unique identifier of the reference_fe_t instance on top of the current cell, c_id, for a given field, f_id, can be retrieved from reference_fe_id=field_cell_to_ref_fes(f_id, c_id), while the reference_fe_t instance itself from reference_fes(reference_fe_id).

The allocatable array member variable in line 23 (with as many entries as triangulation cells) can be used by the user in order to (optionally) determine the degree of the quadrature to be used on each triangulation cell. This member variable is allocated and initialized (during fe_space_t creation) to a reserved flag that instructs set_up_cell_integration to use an automatic (default) strategy to decide the degree of the quadrature to be used on each cell. This default strategy relies on a deferred binding of reference_fe_t, named get_default_quadrature_degree, which typically returns the quadrature degree for which mass matrix terms are integrated exactly (see Sect. 8.1).[28] The strategy,

[27] The term hash table here reflects its usual meaning, i.e., an associative array that maps keys to values.

[28] We stress, however, that each particular reference_fe_t subclass at hand has the freedom to implement a different strategy if required.

in particular, walks over all `reference_fe_t` instances on top of the cell, and the one for which its (polynomial) reference cell functional space is of maximum order becomes ultimately responsible of creating the quadrature via an invocation to its `create_quadrature` deferred binding. Alternatively, the user may explicitly select the quadrature degree to be used on each cell. In such a case, `create_quadrature` is invoked to create a quadrature with the degree given by the corresponding entry in the `cell_quadratures_degree(:)`; see Sect. 8.1. In any case (i.e., default or explicit quadrature degree), both the unique identifier of the `reference_fe_t` instance on top of the current cell and the quadrature degree are used to generate a unique identifier of the quadrature to be created/fetched.

On the other hand, Lines 32–38 of Listing 27 encompass those data structures required for the evaluation of (both boundary and interior) facet integrals; see Sect. 9. A very close rationale to the one underlying their cell counterparts is followed to set up these data structures. The `set_up_facet_integration` binding loops over all facets. Sitting on a facet, it determines an appropriate facet `quadrature_t` rule. The quadrature degree is either the default or a user-defined one (via the allocatable array member variable in Line 32). It also determines the unique identifier of the quadrature and of the rest of the facet-integration data structures, which are created as necessary, while handling their interactions. Both the topology of the two cells sharing the facet and the quadrature degree are used to generate a unique identifier of facet quadratures. The member variables in Lines 41–42 provide support to the implementation on the so-called `fe_facet_iterator_t` data type and will be covered in detail in Sect. 10.2. Finally, the member variable `num_fixed_dofs` in Listing 27 is used by `fe_space_t` to count how many DOFs are subject to strong boundary conditions; see Sect. 10.4.

10.2 A Conceptual View of `fe_space_t`

Following the ideas presented in Sect. 7.1, `fe_space_t` offers a number of iterators to provide traversals over its objects, and uniform data access to its internals. Apart from iterators over cells and vefs, `fe_space_t` also provides traversals over facets by means of the so-called `fe_facet_iterator_t` data type. This iterator is essentially required to implement the evaluation of jump terms in, e.g., error estimators or DG methods in a user-friendly manner. For reasons made clear in the course of this section, a design goal to be fulfilled by `fe_space_t` iterators is that they are able to provide access to the same data as their counterpart `triangulation_t` iterators (see Sect. 7.1), and that they are able to do so *efficiently* while avoiding duplication of code bounded to the latter ones. For example, `fe_cell_iterator_t` should be designed such that it is also able to provide the coordinates (in physical space) of the nodes describing the geometry of the cell, apart from the global DOF identifiers on top of it.

Let us first discuss the design of iterators over cells and vefs (as the one of both follows the same lines). These data types are defined in Listing 28, where `set` must be actually replaced by either `cell` or `vef`.

```
type fe_set_iterator_t
  private
  class(set_iterator_t), allocatable :: set_iterator
  class(fe_space_t)    , pointer     :: fe_space
contains
  public
  procedure :: create       => fe_set_iterator_create
  procedure :: free         => fe_set_iterator_free
  procedure :: first        => fe_set_iterator_first
  procedure :: next         => fe_set_iterator_next
  procedure :: has_finished => fe_set_iterator_has_finished
  ... ! TBPs providing access to data items in the corresponding set (either cell or vef)
end type fe_set_iterator_t
```

Listing 28. `fe_space_t` ``set'' (either `cell` or `vef`) iterators and the composition relationship with their counterpart `triangulation_t` iterators (`set_iterator_t`).

As shown in Listing 28, `fe_set_iterator_t` holds a *polymorphic* pointer to the `fe_space_t` instance to which it has to provide data access. Dynamic polymorphism is exploited here with extensibility and code reuse in mind. Any type extension of `fe_space_t` (e.g., the one suitable for distributed-memory environments), can also become the target of this polymorphic pointer, thus enabling reuse of data and code bounded to fe_set_iterator_t with these extensions. Of special relevance in Listing 28 is the composition relationship among the data type being defined and set_iterator_t, i.e., its triangulation_t iterator counterpart (see Sect. 7.1). This lets fe_set_iterator_t to fulfill the aforementioned design goal, i.e., to provide a superset of data over the class it is composed of, while still being able to access to any data stored within the triangulation scope. fe_set_iterator_t also reuses from set_iterator_t the code underlying the sequential traversal over all objects of the set. Indeed, as many other TBPs of fe_set_iterator_t, init, next, and has_finished TBPs of fe_set_iterator_t are simply implemented as wrappers of their counterparts in set_iterator_t. (We remark that this is possible provided that fe_space_t is deliberately set up such that it shares with triangulation_t a consistent global numbering for cells and lower-dimensional objects.)

At this point it is important to remark that the set_iterator_t instance that fe_set_iterator_t aggregates is also *polymorphic* (see Line 3 in Listing 28). As stated in Sect. 10.1 (in particular, see Line 12 of Listing 27), a fe_space_t instance is created from a polymorphic triangulation_t instance. The create binding of fe_set_iterator_t extracts the latter from fe_space_t, and then calls its create_cell_iterator binding (see Sect. 7.1), which becomes ultimately in charge of determining the dynamic type of the set_iterator_t member variable of fe_set_iterator_t (apart from leaving the iterator positioned in the first object of the set). This lets fe_space_t (and its associated iterators) to be re-used with any type extension of triangulation_t (e.g., the one suitable for distributed-memory computers and/or *h*-adaptivity). Likewise, the free binding of fe_set_iterator_t relies on the free_cell_iterator binding of triangulation_t in order to safely deallocate any dynamic memory allocation performed during creation. We stress that, as in the case of triangulation_t iterators, both the create and free TBPs are not intended to be directly called by the user. Instead, triangulation_t provides a set of (public) TBPs (as many as different iterators) for this purpose. For example, the expression call fe_space%create_fe_cell_iterator(fe_cell_iterator) creates an iterator on the *polymorphic* fe_cell_iterator client-space instance, while call fe_space%free_fe_cell_iterator(fe...) is in charge of safely deallocating this polymorphic instance.

The implementation of fe_facet_iterator_t is based on a very close rationale to the one of cell and vefs iterators, with subtle differences though; see Listing 29. Provided that fe_facet_iterator_t is a kind of fe_vef_iterator_t, it should provide the same set of data access methods of the latter (e.g., the cells sharing the facet). However, it should restrict the traversal to those vefs that are actually facets, and to be able to provide all data required for the implementation of jump terms over facets. As shown in Listing 29, fe_facet_iterator_t extends fe_vef_iterator_t. This automatically equips the former with the data access methods of the latter. On the other hand, it overrides those methods controlling the sequential traversals over the items in the set such that it restricts to facets, i.e., create/free/first/next/has_finished in Listing 29. The implementation of these methods relies on its member variable facet_gid, and the facet_gids(:) member variable of fe_space_t; see Line 41 of Listing 27. For a given facet with global identifier **facet_gid**, facet_gids(facet_gid) holds the global vef identifier corresponding to the facet.

```
type, extends(fe_vef_iterator_t) :: fe_facet_iterator_t
  private
  integer(ip) :: facet_gid
contains
  public
  ! create/free/first/next/has_finished TBPs of fe_facet_iterator_t override
  ! the ones in fe_vef_iterator_t. This lets fe_facet_iterator_t to restrict the
  ! traversal to the set of facets of the triangulation
  procedure :: create       => fe_facet_iterator_create
  procedure :: free         => fe_facet_iterator_free
  procedure :: first        => fe_facet_iterator_first
  procedure :: next         => fe_facet_iterator_next
  procedure :: has_finished => fe_facet_iterator_has_finished
  ... ! TBPs providing access to facet-related data in fe_space_t (See Listing 30)
end type fe_facet_iterator_t
```

Listing 29. The fe_facet_iterator_t data type.

```
1  type :: fe_cell_iterator_t
2    ... ! Member variables (see Listing 28)
3  contains
4    public
5    ... ! Create/free/traversal TBPs (see Listing 28)
6    procedure :: update_integration    => fe_cell_iterator_update_integration
7  
8    ! Helpers for the generation of a global DoF numbering
9    procedure, private :: count_dofs_on_cell_interior
10   procedure, private :: count_dofs_on_vef
11   procedure, private :: generate_dofs_on_cell_interior_numbering
12   procedure, private :: generate_dofs_on_vef_numbering
13   procedure, private :: fetch_dofs_on_vef_numbering_from_source_cell
14 
15   ! Getter TBPs
16   procedure :: get_cell2dof          => fe_cell_iterator_get_cell2dof
17   procedure :: get_quadrature        => fe_cell_iterator_get_quadrature
18   procedure :: get_cell_map          => fe_cell_iterator_get_cell_map
19   procedure :: get_cell_integrator   => fe_cell_iterator_get_cell_integrator
20   procedure :: get_vef               => fe_cell_iterator_get_vef
21 
22   ! Assembly TBPs (See Section  11.1)
23   procedure, private :: fe_cell_iterator_assembly_array
24   procedure, private :: fe_cell_iterator_assembly_matrix
25   procedure, private :: fe_cell_iterator_assembly_matrix_array
26   procedure, private :: fe_cell_iterator_assembly_matrix_array_with_strong_bcs
27   generic  :: assembly => fe_cell_iterator_assembly_array,                        &
28        &                  fe_cell_iterator_assembly_matrix,                       &
29        &                  fe_cell_iterator_assembly_matrix_array,                 &
30        &                  fe_cell_iterator_assembly_matrix_array_with_strong_bcs
31   ... ! Rest of getter TBPs
32 end type fe_cell_iterator_t
33 
34 type, extends(fe_vef_iterator_t) :: fe_facet_iterator_t
35    ... ! Member variables (see Listing 29)
36 contains
37   public
38   ... ! Create/free/traversal TBPs (see Listing 29)
39   procedure :: update_integration    => fe_facet_iterator_update_integration
40   procedure :: get_quadrature        => fe_facet_iterator_get_quadrature
41   procedure :: get_facet_maps        => fe_facet_iterator_get_facet_maps
42   procedure :: get_facet_integrator  => fe_facet_iterator_get_facet_integrator
43   procedure :: get_permutation_index => fe_facet_iterator_get_facet_integrator
44   ! Assembly TBPs (See Section  11.1)
45   procedure, private :: fe_facet_iterator_assembly_array
46   procedure, private :: fe_facet_iterator_assembly_matrix
47   procedure, private :: fe_facet_iterator_assembly_matrix_array
48   procedure, private :: fe_facet_iterator_assembly_matrix_array_with_strong_bcs
49   generic  :: assembly => fe_facet_iterator_assembly_array,                       &
50        &                  fe_facet_iterator_assembly_matrix,                      &
51        &                  fe_facet_iterator_assembly_matrix_array,                &
52        &                  fe_facet_iterator_assembly_matrix_array_with_strong_bcs
53   ... ! Rest of getter TBPs
54 end type fe_facet_iterator_t
```

Listing 30. The `fe_cell_iterator_t` and its facet counterpart.

The actual set of TBPs of a `fe_space_t` iterator highly depends on the type of object being pointed to. For completeness, we now briefly discuss those TBPs in the set corresponding to cell and facet iterators, which provide support for the implementation of the two services of `fe_space_t` we are focusing on. These are in particular shown in Listing 30. This listing also includes the generic TBPs in Lines 35 and 68, although they will be discussed in Sect. 11.1.

The TBPs in Lines 18–28, and 50–61 of Listing 30 let the user fetch from `fe_space_t` the integration data associated to the current cell and facet being pointed to, respectively. On the other hand, the `update_integration` bindings in Lines 6 and 47 perform those computations required to update these data structures such that they hold shape function values and derivatives evaluated at (current) cell and facet (quadrature points) in the physical space. The former binding is implemented as shown in Listing 31. Finally, the `get_permutation_index` TBP of `fe_facet_iterator_t` lets the caller to obtain the permutation index (see Sects. 3.16 and 9.3 for further details). The implementation of this method relies on the `facet_permutation_indices(:)` member variable of `fe_space_t`; see Line 42 of Listing 27. For a given facet with global identifier `facet_gid`, `facet_permutation_indices(facet_gid)` holds the permutation index corresponding to the facet. We have decided to permanently store facet permutation indices for performance reasons. These can be reused over and over again (e.g., in a transient and/or nonlinear PDE problem) without the overhead associated to its computation on each traversal over the facets of the triangulation.

update the `cell_integrator_t` for every field in Line 17. The update of the former requires that its `nodes_coordinates(:)` scratch member variable has been loaded with the coordinates in the physical space of the nodes describing the geometry of the cell at hand (see Sect. 8.3). This is in particular fulfilled in Line 10. The coordinates fetched by this call are actually stored within the triangulation. However, `fe_cell_iterator_t` can satisfy this query provided that it is composed of a `cell_iterator_t` instance; see Listing 28 and accompanying discussion. At this point, the reader should be already capable to grasp how the `fe_facet_iterator_t` counterpart of this subroutine is implemented, so that it is omitted here in order to keep the presentation short.

Going back to Listing 30, the binding in Line 16 lets the user fetch the field-wise global DOF identifiers that `fe_space_t` has associated to the node functionals on

```
1  subroutine fe_cell_iterator_update_integration( this )
2    class(fe_cell_iterator_t), intent(inout) :: this
3    type(cell_map_t), pointer :: cell_map
4    type(point_t) , pointer :: coordinates_nodes(:)
5    ... ! Declaration of the rest of local variables
6
7    ! Update the cell_map_t instance associated to current cell
8    cell_map => this%get_cell_map()
9    coordinates_nodes => cell_map%get_coordinates_nodes()
10   call this%get_coordinates(coordinates_nodes)
11   quadrature => this%get_quadrature()
12   call cell_map%update(quadrature)
13
14   ! Update field-wise cell_integrator_t instances associated to current cell
15   do field_id = 1, this%get_num_fields()
16      cell_integrator => this%get_cell_integrator(field_id)
17      call cell_integrator%update(cell_map)
18   end do
19 end subroutine fe_cell_iterator_update_integration
```

Listing 31. Implementation of the `update_integration` binding of `fe_cell_iterator_t`.

An update of the `cell_map_t` instance (associated to the cell pointed by the `fe_cell_iterator_t` instance on which this subroutine is invoked) is performed in Line 12 of Listing 31. It is followed by a loop over the number of fields of the PDE system at hand in order to the current cell interior and its vefs. (The bindings in Lines 9–13 of Listing 30, however, assist `fe_space_t` on the generation of the global DOF numbering and their usage will be illustrated in Sect. 10.3.) This binding is implemented in Listing 32.

```
subroutine fe_cell_iterator_get_cell2dof(this,cell2dof)
  class(fe_cell_iterator_t) , intent(in)    :: this
  type(p_1D_ip_array_t)      , intent(inout) :: cell2dof(:)
  ... ! Declaration of local variables
  assert (size(cell2dof) == this%get_num_fields())
  do field_id = 1, this%get_num_fields()-1
    spos = this%fe_space%ptr_dofs_per_fe(field_id   ,this%get_id())
    epos = this%fe_space%ptr_dofs_per_fe(field_id+1,this%get_id())-1
    cell2dof(field_id)%p => this%fe_space%lst_dofs_gids(spos:epos)
  end do
  ... ! Treat the last field_id special case
end subroutine fe_cell_iterator_get_cell2dof
```

Listing 32. Implementation of the `get_fe_dofs` binding of `fe_cell_iterator_t`.

In Listing 32, p_1D_ip_array_t is assumed to be a data type with a single member variable, called p, declared as a pointer to a rank-1 integer(ip) array. For each field, the subroutine locates the region within the lst_dofs_gids(:) member variable corresponding to that field within the current cell, and then it associates to it the corresponding pointer in fe_dofs(:). At the expense of sacrificing type safety (in Fortran there is no mechanism to declare a pointer to be read-only), we avoid the costly re-allocation of user-level allocatable arrays that would be needed in the case of non-conforming FE spaces with highly varying degree polynomial spaces among cells.

To end up, the get_vef binding in Listing 30 sets up a fe_vef_iterator_t instance to point to the corresponding vef within the cell. As a consequence, one may navigate over the cells, its vefs, cells around these vefs, etc., using fe_space_t iterators all the way round.

10.3 Global DOF Numbering Generation

In this section, we discuss how fe_space_t coordinates the building blocks covered so far in order to generate a *global* enumeration of the DOFs describing the global FE space $\mathcal{X}_h \doteq \mathcal{X}_h^1 \times \ldots \times \mathcal{X}_h^n$ for general multi-field systems of PDEs. This process is encompassed within the generate_global_dof_numbering binding of fe_space_t (see Listing 27). The code of this method is shown in Listing 33. The block_layout dummy argument lets the caller to customize the global DOF numbering to be generated.[29] On the one hand, this data type specifies in how many blocks the user wants to split the (discrete) PDE system at hand. In particular, the user may select to generate a DOF numbering suitable for monolithic or blocked storage linear algebra data structures, with block_layout%get_num_blocks() returning one and a number larger than one, respectively. On the other hand, block_layout_t specifies the mapping of fields into blocks, with block_layout%get_block_id(field_id) returning the block identifier the field with identifier field_id is mapped to. Provided that blocked linear algebra data structures in FEMPAR are addressed using row/column identifiers that are local to each block, block_layout equips the subroutine with the input necessary to generate a block-aware global DOF numbering, in which the DOFs belonging to fields of the first block are numbered first, followed by the ones of the second, and so on. We note that block_layout_t also holds inside how many DOFs are there per block (see Sect. 11.3). These latter quantities are computed within generate_global_dof_numbering (see discussion in the sequel).

The subroutine in Listing 33 starts checking whether it has to actually generate a global DOF numbering. It has to do so if there is no global DOF numbering available yet (see predicate in Line 9), or if the one available is not suitable for the input block_layout (see predicate in Line 10). The bulk of generate_global_dof_numbering is concentrated in the private helper TBPs of fe_space_t called fe_space_count_dofs and fe_space_list_dofs; see Lines 14 and 15 of Listing 33, respectively. The code of these bindings is shown in Listings 34 and 35, respectively. While the former computes the number of DOFs per field and block, the latter is in charge of the actual generation of the global DOF identifiers.

```
 1 subroutine fe_space_generate_global_dof_numbering(this,block_layout)
 2  class(fe_space_t)            , intent(inout) :: this
 3  type(block_layout_t) , target, intent(inout) :: block_layout
 4  logical :: perform_numbering
 5
 6  ! Is block_layout compatible with fe_space_t?
 7  assert ( this%num_fields == block_layout%get_num_fields() )
 8
 9  perform_numbering = .not. associated(this%block_layout)
10  if (.not. perform_numbering) perform_numbering = .not. (this%block_layout == block_layout)
11  if (perform_numbering) then ! Generate block_layout-conforming global DoF numbering
12     this%block_layout => block_layout
13     ... ! Re-allocate this%num_dofs_x_field(:) to have size this%num_fields (see Listing 27)
14     call this%count_dofs() ! Count how many DoFs are there per field and block
15     call this%list_dofs()  ! Actually generate global DoF identifiers
16  else
17     call block_layout%copy_num_dofs_x_block(this%block_layout)
18  end if
19 end subroutine fe_space_generate_global_dof_numbering
```

Listing 33. The generate_global_dof_numbering binding of fe_space_t.

[29] We refer to Listing 43 and its accompanying text in Sect. 11 for a full description of the member variables and TBPs of block_layout_t. In this section, we restrict ourselves to those that are relevant for the global DOF numbering process.

```
1 subroutine fe_space_count_dofs ( this )
2   class(serial_fe_space_t), intent(inout) :: this
3   ... ! Declaration of local variables
4 
5   ! Count #DoFs per field
6   this%num_dofs_x_field = 0
7 
8   ... ! Allocate owner_cell_per_vef_visited(:,:) to size num_fields x triangulation%get_num_vefs()
9   owner_cell_per_vef_visited = .false.
10 
11   call this%create_fe_iterator(fe)
12   do while ( .not. fe%has_finished())
13     do field_id=1, this%number_fields
14       this%num_dofs_x_field(field_id) += fe%count_dofs_on_cell_interior(field_id)
15       if ( this%conforming_fe_space(field_id) .and. this%continuous_fe_space(field_id) ) then
16          do vef_lid = 1, fe%get_num_vefs()
17             vef_gid = fe%get_vef_gid(vef_lid)
18             if ( .not. owner_cell_per_vef_visited(field_id,vef_gid) ) then
19                num_own_dofs_on_vef = fe%count_own_dofs_on_vef(vef_lid,field_id)
20                if (num_own_dofs_on_vef>0) then
21                   owner_cell_per_vef_visited (field_id,vef_gid) = .true.
22                   this%num_dofs_x_field(field_id) += num_own_dofs_on_vef
23                end if
24             end if
25          end do
26       end if
27     end do
28     call fe%next()
29   end do
30   call this%free_fe_iterator(fe)
31   ... ! Free owner_cell_per_vef_visited(:,:)
32 
33   ! Count #DoFs per block
34   call this%block_layout%clear_num_dofs_x_block()
35   do field_id=1, this%get_number_fields()
36     iblock = this%block_layout%get_block_id(field_id)
37     call this%block_layout%add_to_block_num_dofs(iblock,this%num_dofs_x_field(field_id))
38   end do
39 
40 end subroutine fe_space_count_dofs
```

Listing 34. The `count_dofs` binding of `fe_space_t`.

Lines 6–31 of Listing 34 are in charge of computing the number of DOFs per field, while those in Lines-34–38, those per block. The latter lines just determine the number of DOFs per block by accumulating those corresponding to fields mapped to the block (computed in the former lines). The former lines are grounded on the notion of *owner cell of a vef*; a cell is the owner of a vef if (1) the latter lays on the boundary of the former, (2) it is the *first cell* for which (1) holds in the order in which the iterator over all cells presents them, and (3) the vef owns at least one DOF of the global FE space subject to consideration.[30] The (logical) work array `owner_cell_per_vef_visited(:)` keeps track whether the owner cell of the vefs have been already visited (or not) as these are traversed in the nested loop over all cells (see outer loop in Line 12), and over all vefs within the current cell (see inner loop in Line 16). Sitting on a cell, the algorithm first counts those DOFs associated to node functionals logically placed in the interior of the current cell (see line 14). It then loops over the vefs of the current cell. If the owner cell of the current vef has not been visited yet, and the current cell is its owner, then the current cell is registered as the owner of the cell, and the DOFs associated to node functionals logically placed on this vef within the current cell are counted in Line 22. Provided that non-conforming FE spaces do not have DOFs on vefs, we can skip the loop over the vefs of a cell and accelerate the process in this case (see the if clause in Line 15 of Listing 34).

The algorithm shown in Listing 35 is in charge of the actual generation of the global DOF identifiers. The work array `owner_cell_gid_per_vef(:,:)` is used to store the owner cell global identifier of the vefs. On the other hand, `vef_lid_in_owner_cell(:,:)` array is used as an accelerator lookup table that stores the vef local identifiers (i.e., `vef_lid`) within their corresponding owner cells if they have been already visited, and `-1` otherwise. Both arrays are indexed using vef global identifiers (i.e., `vef_gid`). Sitting on a cell, the algorithm first allocates global DOF identifiers for all node functionals associated to the interior of the current cell starting from `fields_current_dof(field_id)`, i.e., the next freely available global identifier; see Line 27. It then loops over

[30] The last requirement has been introduced to include the concept of void FEs for multi-field problems in which some fields are not defined on the whole domain (see Sect. 6.5).

```
1  subroutine fe_space_list_dofs ( this )
2    class(serial_fe_space_t), intent(inout) :: this
3    ... ! Declare all local variables
4
5    ... ! Allocate owner_cell_gid_per_vef(:,:) to size number_fields x triangulation%num_vefs()
6    owner_cell_gid_per_vef = -1
7
8    ... ! Allocate vef_lid_in_owner_cell(:,:) to size number_fields x triangulation%num_vefs()
9    vef_lid_in_owner_cell = -1
10
11   ... ! Allocate blocks_current_num_dofs(:) work array to block_layout%get_num_blocks()
12   blocks_current_num_dofs = 0
13
14   ... ! Allocate fields_current_dof(:) work array to num_fields
15   fields_current_dof = 0
16
17   do field_id=1, this%number_fields
18     block_id = this%block_layout%get_block_id(field_id)
19     fields_current_dof(field_id) += blocks_current_num_dofs(block_id)
20     blocks_current_num_dofs(block_id) += this%number_dofs_per_field(field_id)
21   end do
22
23   call this%create_fe_iterator(owner_fe)
24   call this%create_fe_iterator(fe)
25   do while ( .not. fe%has_finished())
26    do field_id=1, this%get_number_fields()
27      call fe%generate_dofs_on_cell_interior_numbering(field_id, fields_current_dof(field_id))
28      do vef_lid = 1, fe%get_num_vefs()
29        vef_gid = fe%get_vef_gid(vef_lid)
30        if ( owner_cell_gid_per_vef(field_id,vef_gid) == -1) then
31          previous_dof_block = fields_current_dof(field_id)
32          call fe%generate_dofs_on_vef_numbering(vef_lid, field_id, fields_current_dof(field_id))
33          if (previous_dof_block < fields_current_dof(field_id)) then
34            owner_cell_gid_per_vef(field_id, vef_gid) = fe%get_gid()
35            vef_lid_in_owner_cell (field_id, vef_gid) = vef_lid
36          end if
37        else
38          call owner_fe%set_gid(owner_cell_gid_per_vef(field_id, vef_gid))
39          call fe%fetch_dofs_on_vef_numbering_from_source_cell(vef_lid, &
40                                                               owner_fe, &
41                                                               vef_lid_in_owner_cell(field_id,
                                                                     vef_gid), &
42                                                               field_id)
43        end if
44      end do
45    end do
46    call fe%next()
47   end do
48   call this%free_fe_iterator(owner_fe)
49   call this%free_fe_iterator(fe)
50
51   ... ! Deallocate all work arrays
52 subroutine fe_space_list_dofs ( this )
```

Listing 35. The `list_dofs` binding of `fe_space_t`.

the vefs of the current cell. If the current vef has not been visited yet, then the current cell becomes its owner, and both the cell and the local identifier of this vef within the cell are registered in the corresponding work arrays. The global DOF identifiers associated to node functionals on this vef within the owner cell are allocated in Line 32 (as above starting from `fields_current_dof(field_id)`). On the other hand, if the current vef has been visited, then the global DOF identifiers associated to node functionals on this vef within the current cell are fetched from the corresponding ones within the owner cell in Line 39. The binding called in this line encodes the permutations described in Sect. 3.16.

As the reader might observe, Listing 35 is grounded on several (private) helper bindings of `fe_cell_iterator_t` that, at the cell level, aid in the generation of a global DOF numbering; see Lines 9–13 of Listing 30. These bindings ultimately rely on the `reference_fe_t` instances mapped to the cells of the triangulation; see Sect. 10.1. In particular, sitting on a cell, `reference_fe_t` instructs `fe_cell_iterator_t` with the association of its node functionals to the interior of the cell, and its lower-dimensional boundary objects according to the notion of conformity underlying the FE space at hand; see Sects. 3.6 and 6.2. For example, the implementation of the `generate_own_dofs_vef_numbering` binding is implemented as shown in Listing 36.

The code in Listing 36 extracts a `list_iterator_t` from the `own_dofs_n_face` member variable of the

```
1  subroutine fe_cell_iterator_generate_dofs_on_vef_numbering (this, vef_lid, field_id, current_dof)
2    class(fe_cell_iterator_t) , intent(inout) :: this
3    integer(ip)               , intent(in)    :: vef_lid
4    integer(ip)               , intent(in)    :: field_id
5    integer(ip)               , intent(inout) :: current_dof
6
7    class(reference_fe_t), pointer :: reference_fe
8    type(list_iterator_t)          :: own_dofs_on_vef_iterator
9    ... ! Declaration of the rest of local variables
10
11   base_pos = this%fe_space%ptr_dofs_per_fe(field_id, this%get_gid())-1
12   reference_fe => this%get_reference_fe(field_id)
13   own_dofs_on_vef_iterator = reference_fe%create_own_dofs_on_n_face_iterator(vef_lid)
14   do while (.not. own_dofs_on_vef_iterator%has_finished())
15     dof_pos = base_pos+own_dofs_on_vef_iterator%current()
16     if (this%fe_space%lst_dofs_gids(dof_pos)==0) then
17       this%fe_space%lst_dofs_gids(dof_pos) = current_dof
18       current_dof = current_dof + 1
19     end if
20     call own_dofs_on_vef_iterator%next()
21   end do
22 end subroutine fe_cell_iterator_generate_dofs_on_vef_numbering
```

Listing 36. Implementation of the `generate_own_dofs_vef_numbering` binding of `fe_cell_iterator_t`.

`reference_fe_t` instance used in the current cell for `field_id`. This iterator lets it to traverse those node functionals owned by the vef with local identifier `vef_lid` (see Sect. 6.2), and thus determine the (relative) position in `lst_dofs_gids(:)` of the global DOF identifiers to be allocated for such node functionals. We note that the logical predicate in Line 16 is evaluated to `.true.` if the DOF at hand is actually free, i.e., not subject to boundary conditions imposed in strong form; see Sect. 10.4.

Finally, we would like to stress that error checking statements and a major optimization that can be applied for the single-field single-block case are not shown in the code listings of this section in order to keep the presentation as simple as possible. Both are present in FEMPAR. In particular, for the aforementioned case, the global DOF numbering can be generated with a single loop over all cells (instead of two). The call in Line 14 of Listing 33 can be avoided, deferring the computation of the number of DOFs per field and block to the call in Line 15.

On the other hand, there is no need to generate a global DOF numbering from scratch when there is already one available, a permutation from the old to the new numbering could be computed and applied to `lst_dofs_gids(:)` by a single sweep over all cells. This optimization, however, is not present in FEMPAR, as indeed we did not find frequent the case where an application requires to change on-the-fly the block-layout of the system of PDEs at hand.

10.4 Strong Imposition of Boundary Conditions

In this section, we discuss the mechanisms that `fe_space_t` provides in order to support the strong imposition of boundary conditions. In order to grasp why these mechanisms are needed and how `fe_space_t` is designed to provide them, we must first briefly introduce the approach chosen by FEMPAR in order to handle this type of boundary conditions. We will use the term "fixed DOFs" to refer to those DOFs sitting on the boundary whose values are constrained (i.e., subject to strong boundary conditions), and the term "free DOFs" to refer to the remaining ones. For simplicity, let us restrict ourselves to the Laplacian problem with inhomogeneous Dirichlet boundary conditions $u(x) = u_{\mathrm{D}}(x)$ on Γ_{D} discretized with grad-conforming FEs.[31] The discrete solution $u_h \in \mathcal{X}_h$ can be split into two parts as $u_h = \bar{u}_h + E_h u_{\mathrm{D}}$, where:

$$\bar{u}_h = \sum_{a \in \{\text{free DOFs}\}} \bar{u}_a \phi^a \;+\; \sum_{a \in \{\text{fixed DOFs}\}} 0 \phi^a \quad \text{and} \quad E_h u_{\mathrm{D}} = \sum_{a \in \{\text{free DOFs}\}} 0 \phi^a + \sum_{a \in \{\text{fixed DOFs}\}} u_a^{\mathrm{D}} \phi^a.$$

The nodal values $\bar{u}_a$ are the actual unknowns of the problem at hand. $E_h u_{\mathrm{D}}$ is a discrete Dirichlet data extension, which can be understood as the projection of a Dirichlet data extension $E u_{\mathrm{D}}(x)$ introduced in Sect. 3.1. Its nodal values u_a^{D} are selected such that $E_h u_{\mathrm{D}}$ becomes a suitable boundary FE approximation of $u_{\mathrm{D}}(x)$ (e.g., a boundary FE interpolation).[32] The linear system to be solved in order to compute the nodal values of $\bar{u}_h$ can be written as:

[31] We stress, however, that the approach discussed in the sequel to handle the strong imposition of boundary conditions is applicable to more complex problems and discretizations, e.g., the Maxwell equations discretized with curl-conforming FE spaces.

[32] It is assumed that the discrete Dirichlet data extension is zero on free DOFs, but other more general situations can also be accommodated.

$$\sum_{b\in\{\text{free DOFs}\}} a(\phi^a,\phi^b)\bar{u}_b = (\phi^a,f) - \sum_{c\in\{\text{fixed DOFs}\}} a(\phi^a,\phi^c)u_c^{\mathrm{D}} \quad \forall a\in\{\text{free DOFs}\}, \tag{33}$$

where *its coefficient matrix has as many rows as free DOFs*, and its right-hand side is the FE discretization of the linear form in (3); see Sect. 3.1.

In order to assemble (33), the process described in Sect. 8 has to be slightly modified. A sweep over all triangulation cells is still required. Sitting on a given cell K, the element matrix $\mathbf{A}^K$ and vector $\mathbf{f}^K$ are computed as usual. However, the rows/columns corresponding to fixed DOFs in $\mathbf{A}^K$ are not assembled into the global matrix. The same applies to the entries of $\mathbf{f}^K$. However, $\mathbf{f}^K$ has to be updated before assembly in order to reflect the contributions of strong boundary conditions (see the right-hand side of (33)). Fortunately, the users of FEMPAR are unburdened from these subtleties. These are hidden within the `assembly` generic binding of `fe_cell_iterator_t`; see Listing 30 and 39. Apart from adding the contributions of the current cell to the global coefficient linear system and right-hand side, this binding is in charge of computing the contribution to $\mathbf{f}^K$ from strong Dirichlet boundary conditions. This poses two additional requirements on `fe_space_t`. In particular, (1) it should handle a global enumeration of free and *fixed* DOFs, while being able to distinguish among both kinds of DOFs; and (2) it should offer a suitable set of bindings to project/interpolate $u_{\mathrm{D}}(x)$ on the boundary to get $E_h u_{\mathrm{D}}$.

In order to satisfy (1), `fe_space_t` splits the whole set of DOFs into free and fixed DOFs, and the DOFs within each subset are labeled separately from each other as $\{1,2,\ldots,|\{\text{free DOFs}\}|\}$, and $\{-1,-2,\ldots,-|\{\text{fixed DOFs}\}|\}$, respectively. (This is nevertheless an implementation detail that is never exposed to FEMPAR users.) In turn, free and fixed DOF values are actually stored into different arrays, so that they can be addressed separately using the corresponding global identifiers in the former and latter set, respectively; see Sect. 10.5.

The process that associates global identifiers to free DOFs has been already covered in Sect. 10.3. The one corresponding to fixed DOFs very much resembles the one for free DOFs. It is, however, restricted to vertices, edges, and faces of the triangulation that lay at the boundary, and it has to be equipped with support from the user that lets the process become aware of which DOFs sitting on the boundary are actually fixed. The fixed DOFs global enumeration process occurs during the initial set-up of `fe_space_t`; see `create` generic binding in Listing 27. This process is grounded on two different ingredients. On the one hand, the user can determine Γ sub-regions through the sets associated to vefs sitting on the boundary (see Sect. 7.1). For example, the user may decide to use set identifier 1 and 2 to split the vefs in Γ into those which belong to Γ_{D} and Γ_{N}, respectively. On the other hand, an abstract data type, called `conditions_t`, to be extended by FEMPAR users, lets users to customize the strong imposition of boundary conditions. In particular, with regard to the fixed DOFs global enumeration process, this data type offers a deferred binding that given a set identifier, provides a logical component mask. For each component of the PDE system, this mask provides whether the DOFs associated to vefs marked with this set identifier are fixed or free. For those FE spaces for which there is no DOF-to-component association (e.g., Raviart-Thomas or Nédélec FEs), only the first component in the mask is taken into account, and the rest neglected.

On the other hand, for 2), `fe_space_t` provides a set of methods that let the user interpolate/project $u_{\mathrm{D}}(x)$ on the boundary to get $E_h u_{\mathrm{D}}$ in a number of suitable ways. $E_h u_{\mathrm{D}}$ is ultimately stored within an instance of the `fe_function_t` data type; see Sect. 10.5. Boundary projectors involve the solution of a boundary mass matrix problem where integrals over boundary facets have to be evaluated; see Sect. 9. Again, all these bindings rely on the `conditions_t` abstract data type. In particular, given a boundary vef set identifier, a deferred binding of this data type returns a user-defined (scalar-valued) function to be imposed for each component of the PDE system at hand. In the case of Raviart-Thomas or Nédélec FEs, the d scalar-valued functions corresponding to its components are used to reconstruct the vector-valued function, whose tangential or normal component, respectively is to be imposed.

10.5 Global FE Functions and Their Restriction to Triangulation Cells/Facets

In this section, we introduce a convenient software abstraction in our OO design, referred to as `fe_function_t`, which represents a global FE function $u_h \in \mathcal{X} \doteq \mathcal{X}_h^1 \times \ldots \times \mathcal{X}_h^n$. This data type and a subset of its TBPs (in particular, those that are relevant for the present section) are presented in Listing 37.

In Listing 37, the `free_dof_values` and `fixed_dof_values` are used to store $\bar{u}_h$ and $E_h u_{\mathrm{D}}$, respectively; see Sect. 10.4. The former is a *polymorphic* member variable of type `array_t`; see Sect. 11.1. Relying on the set of deferred bindings offered by `array_t`, the code bounded to `fe_function_t` can be written independently of how the entries within the concrete implementation of `array_t` are laid out in memory, enabling code re-use to a large extent. For example, `scalar_array_t` is a concrete realization of `array_t` that uses monolithic storage, while `block_array_t`

```
type fe_function_t
  private
  class(array_t), allocatable :: free_dof_values
  type(scalar_array_t)        :: fixed_dof_values
contains
  ! subroutine(this(inout)::class(fe_function_t),fe_space(in)::class(fe_space_t))
  procedure, non_overridable  :: create   => fe_function_create
  ! subroutine(this(inout)::class(fe_function_t))
  procedure, non_overridable  :: free     => fe_function_free

  ! Restrict free_dof_values + fixed_dof_values to given cell and field_id
  ! subroutine(this(in)::class(fe_function_t),cell(in)::type(fe_cell_accessor_t),
  !              field_id(in)::integer(ip),nodal_values(:)(inout)::real(rp),allocatable)
  procedure, non_overridable  :: gather_nodal_values        => fe_function_gather_nodal_values
  ... ! Rest of TBPs (getters, operator overloading for fe_function_t expression syntax, etc.)
end type fe_function_t
```

Listing 37. The `fe_function_t` data type.

stores the entries organized into blocks (see Sect. 11.1 for more details). On the other hand, `fixed_dof_values` is a member variable of *static* type `scalar_array_t`; see Sect. 11.1.[33] Fixed DOFs belonging to different fields might be indeed assigned intermixed global identifiers, significantly simplifying the enumeration process. In particular, a single sweep over all boundary objects suffices, in contrast to Listing 33, where two sweeps over all cells are required in order to generate a block-aware global numbering. From our experience, it turns out that neither blocked storage nor a data structure suitable for distributed-memory environments are strictly required to store $E_h u_{\mathrm{D}}$, so that we can prevent the overhead associated to run-time polymorphism when dealing with `fixed_dof_values`.[34]

A `fe_function_t` instance is created from a `fe_space_t` instance (to which it belongs); see signature of the `create` binding in Listing 37. This binding selects the dynamic type of `free_dof_values`, and therefore its storage layout, according to the one currently selected for the PDE system at hand; see `block_layout` member variable in Listing 27. The entries of `free_dof_values` can be determined in a number of ways. They might become the unknowns of a problem to be solved (e.g., by a preconditioned iterative linear solver or sparse direct solver), or computed from an expression involving other `fe_function_t` instances, e.g., $u_h = v_h$, or $u_h = v_h + w_h$, with $u_h, v_h, w_h \in \mathcal{X}_h$. (Indeed, FEMPAR offers an expression syntax for global FE functions grounded on overloaded operators.) Apart from these, `fe_space_t` offers a pair of generic bindings, referred to as `interpolate` and `project`, to compute the DOFs nodal values of u_h by either interpolation (using the expression in (9)) or projection (e.g., a global L^2 projection) into the FE space of a user-defined function $u(x)$.[35] Each of these generic bindings is overloaded with three different regular bindings suitable for scalar, vector, and tensor-valued functions, respectively. The `interpolate` bindings in `fe_space_t` can be written independently of the reference FE by using a TBP associated to `reference_fe_t` that computes the local interpolator in (6).

Apart from the software representation of a global FE function, FE codes typically need a mechanism that, sitting on a cell or facet of the triangulation, provides the values, gradients, etc. of a global FE function $u_h = u_h^1 \times \ldots \times u_h^n$ evaluated at quadrature points in the physical space. To this end, FEMPAR offers a set of data types, referred to as `cell_fe_function_type_t` and `facet_fe_function_type_t`, with `type=scalar,vector,tensor`, that represent the restriction of u_h^i to a given triangulation cell and facet, respectively. The two code snippets in Fig. 9 illustrate the usage of these data

[33] In parallel environments, every processor only stores the fixed DOF values that belong to its associated subdomain.

[34] Some of the algorithms in charge of computing $E_h u_{\mathrm{D}}$ may require a different storage layout from the one of `scalar_array_t` (e.g., blocked storage and/or suitable for distributed-memory computers), and/or restrict themselves to those fixed DOFs of $E_h u_{\mathrm{D}}$ corresponding to a given field (or set of fields). In such a case, $E_h u_{\mathrm{D}}$ is scattered in place back and forth into temporary work space with the appropriate layout for the algorithm at hand in charge of computing its entries (e.g., a serial or parallel distributed-memory boundary mass problem iterative solver). It turns out that it is not such a high performance penalty provided that such algorithms already require to perform a sweep over boundary facets (e.g., in order to assemble a boundary mass matrix). During this sweep, the fixed DOFs in question can be already counted and identified.

[35] Analytical scalar, vector, and tensor-valued functions are also supported in FEMPAR through the classes `scalar_function_t`, `vector_function_t`, and `tensor_function_t`, respectively. To implement an analytical scalar function $f(\boldsymbol{x})$ in FEMPAR, the user has to extend `scalar_function_t` methods `get_value`, `get_gradient` (if used), etc., with the analytical expression, for a given `point_t` that represents $\boldsymbol{x}$. We proceed analogously for vector and tensor fields. These data types are very simple and we omit their description here.

```
type(fe_space_t)                 :: fe_space
type(fe_function_t)              :: u_h
type(cell_fe_function_vector_t)  :: u_i_h_K
type(tensor_field_t)             :: gradient
...
call u_i_h_K%create(fe_space,field_id=i)
! Loop over all cells
call fe_space%create_fe_cell_iterator(fe)
do while (.not.fe%has_finished())
  call fe%update_integration()
  ! Restrict u_i_h to current cell K
  call u_i_h_K%update(fe,u_h)
  ! Fetch u_i_h gradients evaluated on K
  ! quadrature points
  do q=1, u_i_h_K%get_num_quad_points()
   call u_i_h_K%get_gradient(q,gradient)
  end do
  call fe%next()
end do
call fe_space%free_fe_cell_iterator(fe)
```

```
...
type(facet_fe_function_vector_t) :: u_i_h_F
type(tensor_field_t)             :: gradient
...
call u_i_h_F%create(fe_space,field_id=i)
call fe_space%create_..._iterator(fe_facet)
do while (.not.fe_facet%has_finished())
 call fe_facet%update_integration()
 ! Restrict u_i_h to current facet F
 call u_i_h_F%update(fe_facet,u_h)
 ! Fetch u_i_h gradients evaluated on F
 ! quadrature points from both perspective
 ! of K-(K==1) and K+(K==2)
 do K=1,fe_facet%get_num_cells_around()
  do q_point=1,u_1_h_F%get_num_quad_points()
   call u_i_h_F%get_gradient(K,q_point,&
                            gradient)
  end do
 end do
 call fe_facet%next()
end do
call fe_space%free_..._iterator(fe_facet)
```

Fig. 9 User-level code snippets illustrating the usage of the `cell_fe_function_type_t` (left) and `facet_fe_function_type_t` (right) data types

types, where we are assuming that u_h^i belongs to a global FE space of vector-valued functions.

There are three worth noting remarks in these two code snippets. First, the `update` binding of both data types rely on the `gather_nodal_values` binding of `fe_function_t`; see Listing 37. The latter equips cell/facet FE functions with the ability to restrict (gather) the nodal values of u_h^i from global to local arrays (stored as private scratch data within cell/facet FE function data types), *while taking care of strong boundary conditions*. Second, the `update` bindings require a procedure that, given the shape functions, first derivatives, etc., evaluated at quadrature points in physical space, and the nodal values u_h^i restricted to the current cell, provides the shape function values, gradients, curls, etc., of the FE function at these quadrature points. This service is provided by `reference_fe_t` by the set of `evaluate_fe_function...` deferred bindings in Lines 63–68 of Listing 6. We note that `fe_function_t` can extract the first set of data from the `cell_integrator_t` and `facet_integrator_t` instances accessible through `fe_cell_iterator_t` and `fe_facet_iterator_t` (provided on input to `update`), respectively. Third, facet FE functions provide u_h^i values, gradients, etc., at facet quadrature points from the perspective of its two surrounding cells. This make sense for functions u_h^i belonging to non-conforming FE spaces, which might be discontinuous across cell boundaries. Facet FE functions should also cope with the fact that the coordinate systems of its surrounding cells might not be aligned in physical space, so that a different local numbering might be assigned to facet quadrature points from the perspective of either cell; see Sect. 9.3 for an exposition of the strategy followed to solve this issue.

11 Building FE Affine Operators

In this section, we introduce the software abstractions on which the construction of the algebraic problem (10) in Sect. 3 relies. These software abstractions, and their relationship, are depicted in Fig. 10. The main design goal underlying the proposed software architecture is as follows. In the seek of code reusability and extensibility, `FEMPAR` users should have at their disposal *a unique entry point data type and associated bindings* in order to build their FE linear system, no matter whether a scalar or a system of PDEs, no matter whether the linear algebra data structures holding the linear system entries are either scalar (monolithic) or blocked, and no matter how they are laid out in memory (centralized, distributed-memory). In `FEMPAR`, this unique entry point data type is referred to as `fe_affine_operator_t`. Mathematically, `fe_affine_operator_t` represents the affine operator in (5), obtained from the discrete weak formulation of the linear(ized) problem (4). As introduced in Sect. 3.6, the operator can be represented (after defining bases for trial and test FE spaces) with $\mathbf{A}$ and $\mathbf{f}$ defined in (10). The solution of the FE problem is the only root of this operator (as soon as the FE problem is nonsingular).

In order to seek the aforementioned goal, `fe_affine_operator_t` relies on an abstract data type, referred to as `assembler_t` (see Fig. 10). In a nutshell, `assembler_t` offers a set of *FE-assembly tailored*, data structure neutral, deferred TBPs, e.g., to assemble the contributions of a cell or facet integral into the linear system coefficient matrix $\mathbf{A}$ and/or right-hand side $\mathbf{f}$. The subclasses of `assembler_t` are the ones ultimately responsible to deal with the details underlying the

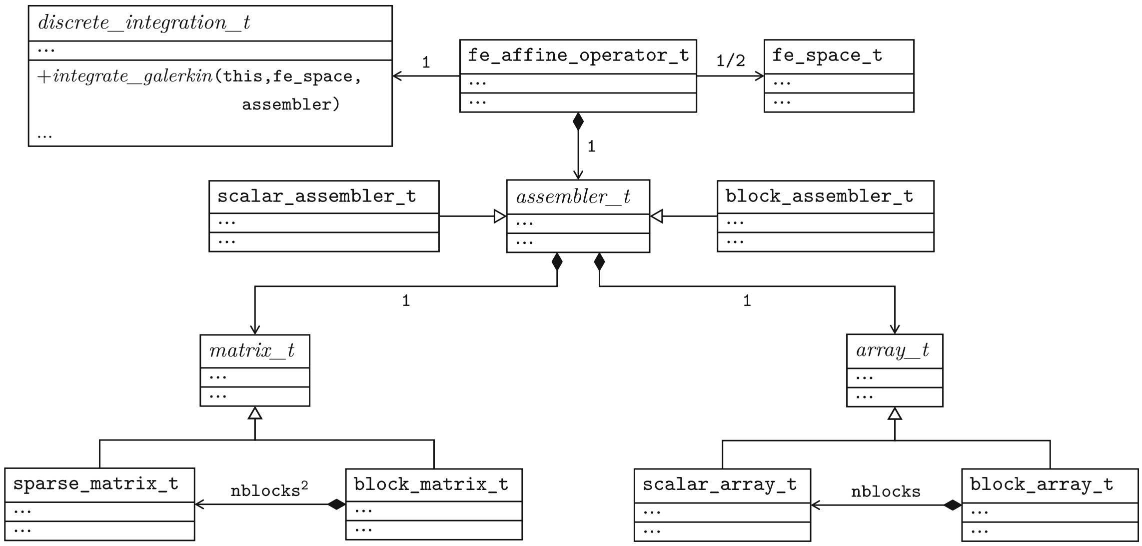


Fig. 10 UML class diagram of the fe_affine_operator_t abstraction and its relationship with other FEMPAR classes

particular linear algebra data structures at hand. The latter ones offer *FE-assembly neutral* interfaces to inject new entries or add contributions to them, such that this software piece becomes reusable and separable, e.g., to be used in third party software projects (not necessarily FE-oriented) as a standalone software subsystem. FEMPAR offers a rich set of linear algebra data structures, e.g., data structures organized by blocks, which enable the implementation of block preconditioners for multiphysics problems (see, e.g., [43–45]). Apart from those required to handle the linear coefficient matrix and right-hand side of the system, fe_affine_operator_t also interacts with other data types required to deliver its life cycle (i.e., its auto-generation). In particular, **A** and **f** entries are computed according to the expressions in (10). These expressions involve a FE space (fe_space_t) and the discrete (bi)-linear forms of the problem at hand. To express in software this second ingredient, we introduce the discrete_integration_t abstraction; see Fig. 10.

We have structured this section as follows. In Sect. 11.1, we first present the assembler_t abstract data type, and the rationale underlying the design of the linear algebra structures it is grounded on. Next, in Sect. 11.2, we introduce the discrete_integration_t abstract data type that ultimately is in charge of performing the integration of the (bi)linear forms and assembly of the discrete affine operator. We show a particular implementation of this data type (i.e., a subclass) for the Galerkin approximation of the Stokes problem. Finally, the fe_affine_operator_t data type is described in Sect. 11.3.

11.1 Linear Algebra Data Structures and Associated Assemblers

Linear algebra in FEMPAR relies on a pair of data type hierarchies rooted at the mathematical abstractions of a linear algebra operator and a vector, and represented in software by means of the linear_operator_t and vector_t abstract data types, respectively. These abstract data types let a number of linear algebra algorithms within FEMPAR (e.g., iterative linear solvers and block preconditioners for PDE systems) to be expressed independently from the actual implementation of the concrete matrix and vector data structures being used, such as block layout (if any), storage (e.g., dense or sparse storage format) or memory layout (e.g., local or distributed-memory), enabling code re-use and extensibility to a large extent. An abstract expression syntax that allows the construction of complex expressions involving operations among operators and/or vectors is also provided. This enables the implementation of new algorithms in a compact manner. However, because these linear algebra algorithms are not discussed herein but postponed to a further work, the description of the data types and associated methods in these hierarchies will be restricted to what is necessary to describe the assembly of the FE affine operator.

The sparse_matrix_t data type can be found at an intermediate level in the hierarchy rooted at linear_operator_t. This is a crucial data type in FEMPAR, which represents a scalar, non-distributed, sparse matrix. Its design follows the ideas presented in [92]. This design (re)uses the *state* OO design pattern [88] to hide the actual sparse matrix storage format to the user. Following this pattern, `sparse_matrix_t` is composed of a polymorphic member variable of (declared) type `base_sparse_matrix_t`. Its dynamic type can be thus changed at runtime (via re-allocation). This dynamic type represents the storage at hand being used. Current subclasses of `base_sparse_matrix_t` include `coo_sparse_matrix_t`, `csr_sparse_matrix_t`, `csc_sparse_matrix_t`, corresponding to the coordinate list (COO), the compressed sparse row (CSR), and the compressed sparse column (CSC) sparse matrix storage formats [93], respectively.

The life cycle of a `sparse_matrix_t` instance is as follows. The user first invokes its `create` TBP, in which one solely specifies the size of the matrix, i.e., the number of rows and columns. This method, however, triggers a number of subsequent actions. In particular, *it allocates its dynamic type to the one corresponding to the COO format*, and leaves it ready for the injection or addition of contributions to the entries of the matrix. Although not compressed, this format is ideally shaped for the injection or addition of contributions to the entries of the matrix. These are simply pushed back into member arrays that can grow dynamically during the integration/assembly loop (via a judiciously reallocation strategy to trade off cost and memory). Other sparse storage formats, as the CSR storage implemented in the `csr_sparse_matrix_t` data type (also a type extension of `base_sparse_matrix_t`), although more memory efficient, require a predefined sparsity pattern, which has to be precomputed. They are not thus well suited for the dynamic build up process of the matrix. At this point the reader should note that, for such inflexible storage formats, *one typically needs an accurate estimation of the maximum number of nonzeros per each row (or column) to be memory efficient*. This estimation, however, can only be a quite large upper bound in complex scenarios (e.g., *hp*-adaptive methods in 3D, among others).

Once the build up process finishes, the user can call a method specially designed to leave the `sparse_matrix_t` instance ready for being used (e.g., to perform operations with it). This involves a compression process, in which duplicated entries are either summed up, or filtered (as selected by the user) and a transformation of the COO storage format into the storage format that the user actually requires (e.g., CSR). For simplicity, we refer to this stage as the "compression" of the matrix. Once the `sparse_matrix_t` instance is in this final state, it is still possible to insert or add contributions to its entries, as far as they belong to the sparsity pattern resulting from the first build up process. Thus, e.g., if a transient and/or nonlinear problem is to be solved and the triangulation of the domain does not change, the assembly in COO format will only be performed at the first nonlinear iteration of the first time step.

As shown so far, the software architecture of `sparse_matrix_t` is such that several (current and future) storage formats are possible within a single framework. This flexibility is convenient for two main reasons. First, no given storage format is likely to be uniformly better in performance across all possible operations and computer architectures. Second, `FEMPAR` interoperability with external software dramatically increases. If a new library, that uses its own storage format, is to be integrated, only a new extension of `base_sparse_matrix_t` has to be added, while leveraging dozens of thousands of lines of code already written. Apart from `sparse_matrix_t`, there are other sparse matrix data types available, suitable to handle blocks and/or distributed-memory computers. All these data types are essentially composed in some way or another of `sparse_matrix_t` instances. For example, `block_sparse_matrix_t` is composed of $\texttt{nblocks}^2$ `sparse_matrix_t` instances; see Fig. 10. It, however, provides a set of specialized TBPs that only apply in the blocked case, e.g., the `get_block` TBP that lets a client to retrieve the `sparse_matrix_t` instance corresponding to a given block of the matrix.

The counterpart of `sparse_matrix_t` in the vector case is referred to as `scalar_array_t`. It represents a scalar, non-distributed, linear algebra vector, with its entries stored explicitly in a simple (Fortran intrinsic) allocatable array. However, provided that it does not have to exploit sparsity, the code bounded to this data type is significantly simpler to the one bounded to `sparse_matrix_t`. It is equipped with a pair of generic bindings, with signatures coming in different flavours, in order to insert or add contributions to the vector. Likewise, there are other vector-like data types available suitable to handle blocks and/or distributed-memory computers. For example, `block_array_t` is composed of `nblocks` `scalar_array_t` instances; see Fig. 10.

Apart from the linear algebra data structures so far, we need the additional data type `assembler_t`, which offers FE-assembly tailored signatures to `fe_affine_operator_t`. The interface of its deferred TBPs, which its extensions, e.g., `scalar_assembler_t` and `block...assembler_t`, implement, are shown in Listing 38. `assembler_t` has to be "general enough" to handle many storage layouts and it is in charge to isolate `fe_affine_operator_t` from implementation details. With that purpose in mind, it is composed of a (polymorphic) `matrix_t` and a (polymorphic) `array_t` instance. These are in turn abstract data types rooted at all the matrix and array data types seen so far, respectively. The set of deferred TBPs of these two abstract data types is designed (on purpose) to be not sufficiently rich to handle the whole life cycle of the concrete matrix and array instances. The high heterogeneity of the concrete subclasses of matrix_t and array_t precludes it. This set of TBPs is, in particular, restricted to allocation of memory for its entries, initialization of its entries to a given value (e.g., initialization to zero), and deallocation of any internal memory. These three operations are required by fe_affine_operator_t during the deployment of its life cycle. The bulk of the life cycle of the concrete subclasses of matrix_t and array_t is handled by the subclasses of assembler_t. This is how it should be, provided that assembler_t subclasses are the ones aware of the concrete details of the corresponding matrix_t and array_t subclasses. Besides, by doing this, we can overcome the overhead associated to dynamic run-time polymorphism, provided that the binding of fine-grain calls to those TBPs injecting or adding contributions to the matrix or the array *can be determined at compilation time*.

Going back to Listing 38, observe that assembly_array (resp., assembly_matrix) takes an intrinsic Fortran array (resp., rank-2 array) as dummy argument for the element vector (resp., matrix). Besides, it also gets the global DOFs identifiers on top a single cell, or those corresponding to cells surrounding the facet (see Lines 23, 35 and 36 in Listing 38). In the case of scalar_assembler_t, the implementation is made using the TBPs provided by scalar_array_t in order to add contributions to its entries and the corresponding TBPs of sparse_matrix_t. In the case of block_assembler_t, the implementation is made by looping through the blocks, obtaining a reference to the current block with the get_block TBP, and using the corresponding TBPs as in the previous case. The assembly_array and assembly_matrix TBPs are used by the fe_cell_iterator_t and fe_facet_iterator_t data types to implement their assembly TBPs (see Lines 35 and 68 in Listing 30 of Sect. 10.2). For completeness, in Listing 39 we show the

```fortran
1  type, abstract :: assembler_t
2     private
3     class(matrix_t), pointer :: matrix
4     class(array_t) , pointer :: array
5   contains
6     procedure (assembly_array_interface)   , deferred :: assembly_array
7     procedure (assembly_matrix_interface) , deferred :: assembly_matrix
8     procedure (compress_storage_interface), deferred :: compress_storage
9     ...! TBPs to allocate, init and free matrix and array
10    ...! Getter and Setter TBPs
11 end type assembler_t
12
13 abstract interface
14    subroutine assembly_array_interface( this, number_fields, field_blocks, field_coupling, &
15         &                               number_dofs, cell2dof, elvec)
16      class(assembler_t) , intent(inout) :: this
17      integer(ip)        , intent(in)    :: number_fields
18      integer(ip)        , intent(in)    :: field_blocks(number_fields)
19      logical            , intent(in)    :: field_coupling(number_fields,number_fields)
20      ! Number of DOFs per field on the current cell (See Section  10)
21      integer(ip)        , intent(in)    :: number_dofs(number_fields)
22      ! List of DOFs GIDs for each field at the current cell (See Section  10)
23      type(i1p_t)        , intent(in)    :: cell2dof(number_fields)
24      real(rp)           , intent(in)    :: elvec(:)
25    end subroutine assembly_array_interface
26
27    subroutine assembly_matrix_interface(this, number_fields, field_blocks, field_coupling, &
28         &            number_row_dofs, number_col_dofs, cell2row_dofs, cell2col_dofs, elmat)
29      class(assembler_t), intent(inout) :: this
30      integer(ip)       , intent(in)    :: number_fields
31      integer(ip)       , intent(in)    :: field_blocks(number_fields)
32      logical           , intent(in)    :: field_coupling(number_fields,number_fields)
33      integer(ip)       , intent(in)    :: number_row_dofs(number_fields)
34      integer(ip)       , intent(in)    :: number_col_dofs(number_fields)
35      type(i1p_t)       , intent(in)    :: cell2row_dofs(number_fields)
36      type(i1p_t)       , intent(in)    :: cell2col_dofs(number_fields)
37      real(rp)          , intent(in)    :: elmat(:,:)
38    end subroutine assembly_matrix_interface
39
40    subroutine compress_storage_interface( this, sparse_matrix_storage_format )
41      class(assembler_t) , intent(inout) :: this
42      character(*)       , intent(in)    :: sparse_matrix_storage_format
43    end subroutine compress_storage_interface
44 end interface
```

Listing 38. The assembler_t abstract data type and its deferred TBPs.

signature of the latter TBPs. These are the ones actually used by the user in the type extension of discrete_integration_t, as described in Sect. 11.2.

Finally, the compress_storage deferred TBP of assembler_t lets fe_affine_operator_t to signal that the build up process of the linear algebra data structures has already finished and that they can already be "compressed" into its final stage.

We stress that the software architecture presented in this section provides uniform assembly interfaces to the client that are completely independent of the underlying implementation of linear algebra data structures. The subclasses of assembler_t are in charge of the management of blocks (if any), whereas sparse_matrix_t is in charge of the management of the storage schemes.

11.2 Discrete Integration of FE Operators

In this section, we introduce the abstract data type discrete_integration_t (see Listing 40). It defines the generic integrate binding, which is overloaded by the integrate_galerkin and integrate_petrov_galerkin deferred TBPs, depending on the number of fe_space_t instances being passed to them (see, e.g., Line 8 of Listing 40 for the interface corresponding to the Galerkin case). A user that wants to implement a FE problem must extend this data type and overwrite the TBP to be used (Galerkin or Petrov-Galerkin) in the user-defined subclass. In the overridden method, the user must implement the evaluation of the entries of **A** and **f** as the numerical integration of the discrete bilinear and linear forms as in (10) (see Sect. 3).

Based on our experience, the integration part of a FE code must exhibit a huge level of flexibility. Every time one wants to consider a new set of PDEs or a new expression of the discrete bilinear form, this component must be modified. It must also have the ability to integrate general time integration schemes that can require functions in an arbitrary number of steps, deal with nonlinear problems that involve the need to evaluating FE functions in the integration of the discrete forms, or including variable physical coefficients of body force terms determined through analytical functions. As a result, any rigidity at this level must be eliminated. Indeed, the discrete_integration_t abstract data type only forces its subclasses to adhere to the signatures of the deferred TBPs overloading integrate, and has no member variables that subclasses are forced to handle. Using the design previously sketched, the user has absolute flexibility to design its own discrete_integration_t subclass, adding the attributes and methods that can be required to integrate and assemble the discrete forms, e.g., by adding an arbitrary number of fe_function_t and *_function_t instances (and corresponding setters to be used at the driver level) that can describe physical properties, previous time step values, the solution at the previous nonlinear iteration, etc.

The integration of cell-wise terms of the (bi)linear forms is accomplished by traversing through all the cells using a fe_cell_iterator instance (see Sect. 10.2), which

```
interface
   subroutine fe_set_iterator_assembly_array( this, elvec, assembler)
     class(fe_set_iterator_t) , intent(in)    :: this
     real(rp)                 , intent(in)    :: elvec(:)
     class(assembler_t)       , intent(inout) :: assembler
   end subroutine fe_set_iterator_assembly_array
   subroutine fe_set_iterator_assembly_matrix( this, elmat, assembler)
     class(fe_set_iterator_t) , intent(in)    :: this
     real(rp)                 , intent(in)    :: elmat(:,:)
     class(assembler_t)       , intent(inout) :: assembler
   end subroutine fe_set_iterator_assembly_matrix
   subroutine fe_set_iterator_assembly_matrix_array( this, elmat, elvec, assembler)
     class(fe_set_iterator_t), intent(in)    :: this
     real(rp)                , intent(in)    :: elmat(:,:)
     real(rp)                , intent(in)    :: elvec(:)
     class(assembler_t)      , intent(inout) :: assembler
   end subroutine fe_set_iterator_assembly_matrix_array
    subroutine fe_set_iterator_assembly_matrix_array_with_strong_bcs( this, elmat, elvec, &
                                                                      fe_function, assembler)
     class(fe_set_iterator_t), intent(in)    :: this
     real(rp)                , intent(in)    :: elmat(:,:)
     real(rp)                , intent(inout) :: elvec(:)
     type(fe_function_t)     , intent(in)    :: fe_function
     class(assembler_t)      , intent(inout) :: assembler
   end subroutine fe_set_iterator_assembly_matrix_array_with_strong_bcs
end interface
```

Listing 39. The interfaces of the assembly TBPs of "set" (either cell or facet) iterators.

```
1 type, abstract :: discrete_integration_t
2  contains
3    procedure (integrate_galerkin_interface)       , deferred :: integrate_galerkin
4    procedure (integrate_petrov_galerkin_interface), deferred :: integrate_petrov_galerkin
5    generic :: integrate => integrate_galerkin, integrate_petrov_galerkin
6 end type discrete_integration_t
7 abstract interface
8    subroutine integrate_galerkin_interface ( this, fe_space, assembler)
9      class(discrete_integration_t), intent(in)    :: this
10     class(fe_space_t)            , intent(inout) :: fe_space
11     class(assembler_t)           , intent(inout) :: assembler
12   end subroutine integrate_galerkin_interface
13   subroutine integrate_petrov_galerkin_interface ( this, test_fe_space, trial_fe_space, assembler)
14     class(discrete_integration_t), intent(in)    :: this
15     class(fe_space_t)            , intent(inout) :: test_fe_space
16     class(fe_space_t)            , intent(inout) :: trial_fe_space
17     class(assembler_t)           , intent(inout) :: assembler
18   end subroutine integrate_petrov_galerkin_interface
19 end interface
```

Listing 40. The abstract data type discrete_integration_t and its deferred TBPs.

has access to (1) all the cell integration data (see Sect. 8) required to compute the local cell contributions in (11) and (2) the local-to-global DOF numbering needed for the assembly in the global linear algebra data structures. Analogously, the integration of facet terms, e.g., the ones in (20) for DG formulations, requires the use of a fe_facet_iterator_t instance to traverse through the facets and integrate the corresponding facet terms. The method integrate is called during the execution of the numerical_setup TBP of fe_affine_operator_t. It is in fact the fe_affine_operator_t the one that decides whether to invoke the Galerkin or Petrov-Galerkin integration, depending on whether one or two FE spaces have been passed as actual arguments (the second one being optional) in its create binding (see Line 15 of Listing 42). Analogously, the FE space(s) are also passed as actual argument(s) to the integrate_* bindings, since they will be needed at any integration step (see Line 8 of Listing 40 for the Galerkin case).

For illustration purposes, we present in Listing 41 an example extension of discrete_integration_t. It shows the implementation of the deferred procedure integrate_galerkin for the approximation of the Stokes problem using a Galerkin method. This data types will be used in the example driver presented in Sect. 12 for the inf-sup stable Taylor-Hood mixed FE method (see Listing 41).[36]

As commented above, the integration of the (bi)linear forms requires the cell integration machinery, which is provided by fe_space_t through the creation of the fe_cell_iterator_t in Line 19 of Listing 41. Apart from controlling the loop over cells (Lines 24 and 62), fe_cell_iterator_t provides the numerical quadrature, which is in turn required to get the number of integration points (line 31), and its associated weights (line 32). It also provides the determinant of the Jacobian of the cell map (line 32), and the shape functions and gradients at Lines 28 to 30 (see (13) and (14)). The implementation of the (bi)linear forms is very close to the blackboard expression, making it compact, simple, and intuitive. This is possible through the definition of the vector_field_t, and tensor_field_t data types, together with their corresponding expression syntax available in FEMPAR. As it was carefully discussed in Sect. 8.5, it is achieved using operator overloading for different vector and tensor operations, e.g., the contraction and scaling operations. The symmetric_part (used at Lines 35 and 38), double_contract (used at line 40) and trace helper stand-alone functions (used at Lines 36 and 49) are also offered to make tensor operations easy. We also note that this implementation is also *efficient*, since all these operations are made without any dynamic memory allocation/deallocation.

Finally, the fe_cell_iterator_t also offers a TBP to assemble the element matrix and vector into the assembler and to impose strong Dirichlet conditions (line 66) using the perturbation in (3) (See Sect. 10.4). The Dirichlet data is extracted from a fe_function_t that represents $E_h u_D$, which must be an attribute of the concrete discrete_integration_t. For non-conforming FE spaces, the formulation requires also a loop over the facets to integrate DG terms. It can be written in a similar fashion using the tools described in Sect. 9. In this example, the stokes_galerkin_integration_t extension has the attribute force, which is used in Line 56 to integrate

[36] We note that the Stokes subclass of discrete_integration_t in Listing 41 implements the Galerkin approximation for this problem but it is independent of the FE space being used. It can be re-used for any conforming inf-sup stable mixed FE method, e.g., Taylor-Hood, conformal Crouzeix-Raviart, MINI element, etc. The choice of the mixed FE space will be determined by the user in the driver, when building the Cartesian two-field FE space.

```
subroutine stokes_integrate(this, fe_space, assembler)
  class(stokes_discrete_integration_t), intent(in)    :: this
  class(fe_space_t)                   , intent(inout) :: fe_space
  class(assembler_t)                  , intent(inout) :: assembler

  class(fe_cell_iterator_t), allocatable :: fe                      ! See Sect. 10.2
  type(quadrature_t)       , pointer     :: quad                    ! See Sect. 8.1
  type(cell_map_t)         , pointer     :: cell_map                ! See Sect. 8.3
  type(cell_integrator_t)  , pointer     :: cell_int_u              ! See Sect. 8.4
  type(cell_integrator_t)  , pointer     :: cell_int_p              ! See Sect. 8.4
  real(rp)                 , allocatable :: shape_p_values(:,:)     ! See Sect. 8.5
  type(tensor_field_t)     , allocatable :: shape_u_gradients(:,:) ! See Sect. 8.5
  type(vector_field_t)     , allocatable :: shape_u_values(:,:)     ! See Sect. 8.5

  real(rp)    , allocatable :: elmat(:,:), elvec(:)
  integer(ip), allocatable :: num_dofs_x_field(:)
  integer(ip)               :: qpoint, idof, jdof
  type(tensor_field_t)      :: eps_v, eps_u
  real(rp)                  :: dV, div_v, div_u

  call fe_space%create_fe_iterator(fe)
  ...! Allocate elmat, elvec, num_dofs_x_field
  call fe%get_num_dofs_per_field(num_dofs_x_field)
  quad           => fe%get_quadrature()
  cell_map       => fe%get_cell_map()
  cell_int_u     => fe%get_cell_integrator(1)
  cell_int_p     => fe%get_cell_integrator(2)
  ! See in Sect. 10.2 the discussion of Listing 28
  do while ( .not. fe%has_finished())
     ! See in Sect. 10.2 the discussion of Listing 30
     call fe%update_integration()
     elmat = 0.0_rp; elvec = 0.0_rp
     call cell_int_u%get_gradients(shape_u_gradients)
     call cell_int_u%get_values(shape_u_values)
     call cell_int_p%get_values(shape_p_values)
     do qpoint = 1, quad%get_num_quadrature_points()
        dV = cell_map%get_det_jacobian(qpoint) * quad%get_weight(qpoint)
        ! LHS
        do idof = 1, num_dofs_x_field(1)
           eps_v = symmetric_part(shape_u_gradients(idof,qpoint))
           div_v = trace(epsd_v)
           do jdof = 1, num_dofs_x_field(1)
              eps_u  = symmetric_part(shape_u_gradients(jdof,qpoint))
              elmat(idof,jdof) = elmat(idof,jdof) &
                   &           + dV * double_contract( eps_v , 2.0_rp * this%mu * eps_u)
           end do
           do jdof = 1, num_dofs_x_field(2)
              elmat(idof,num_dofs_x_field(1)+jdof) = elmat(idof,num_dofs_x_field(1)+jdof) &
                   &                                + dV * div_v * shape_p_values(jdof,qpoint)
           end do
        end do
        do idof = 1, num_dofs_x_field(2)
           do jdof = 1, num_dofs_x_field(1)
              div_u = trace(shape_u_gradients(jdof,qpoint))
              elmat(num_dofs_x_field(1)+idof,jdof) = elmat(num_dofs_x_field(1)+idof,jdof) &
                   &                                + dV * shape_p_values(idof,qpoint) * div_u
           end do
        end do
        ! RHS
        do idof = 1, num_dofs_x_field(1)
           elvec(idof) = elvec(idof) + dV * shape_u_values(idof,qpoint) * this%force
        end do
     end do
     ! Assemble elmat/elvec into assembler while taking care of strong
     ! Dirichlet BCs (in turn extracted from this%fe_function)
     call fe%assembly ( elmat, elvec, this%fe_function, assembler ) ! See Sect. 11.1
     call fe%next()
  end do
  call fe_space%free_fe_iterator(fe)
  ! Free shape_* arrays, elmat, elvec, num_dofs_x_field
end subroutine stokes_integrate
```

Listing 41. The implementation of a binding that overrrides the `integration_galerkin` TBP of `discrete_integration_t` for the Galerkin approximation to the Stokes problem.

```
1  type, extends(linear_operator_t):: fe_affine_operator_t
2   private
3   ...
4   ! Data type describing the block layout of the PDE system (See Listing 43)
5   type(block_layout_t)                     :: block_layout
6   ! Polymorphic pointers to fe_spaces it was created from (See Section 10)
7   class(fe_space_t)           , pointer    :: test_fe_space         => NULL()
8   class(fe_space_t)           , pointer    :: trial_fe_space        => NULL()
9
10  ! Polymorphic pointer to the instance defining the bilinear form (See Section  11.2)
11  class(discrete_integration_t), pointer   :: discrete_integration => NULL()
12  ! Polymorphic pointer to the instance storing linear algebra data structures (See Section  11.1)
13  class(assembler_t)          , pointer    :: assembler             => NULL()
14 contains
15  procedure           :: create => fe_affine_operator_create
16  procedure           :: free   => fe_affine_operator_free
17  procedure           :: setup  => fe_affine_operator_setup
18  ... ! Getter/setters TBPs
19  ... ! TBPs overriding the deferred TPBs of linear_operator_t
20  ... ! Misc. TBPs not relevant for the current section
21 end type fe_affine_operator_t
```

Listing 42. The `fe_affine_operator_t` data type.

the right-hand side. It is a vector field described by an instance of the `vector_function_t` data type.

11.3 The FE Affine Operator Abstraction

A (simplified) declaration of the `fe_affine_operator_t` data type is shown in Listing 42. The `fe_affine_operator_t` is created from a single `fe_space_t` instance, or even two for Petrov-Galerkin formulations; the second instance is optional and, when it is not passed, the Galerkin method is used, i.e., the same FE space is used for trial and test spaces. The user can (optionally) configure a desired block layout. Given a Cartesian product FE space $\mathcal{X}_h^1 \times \ldots \times \mathcal{X}_h^{n_{\text{field}}}$ for a multi-field problem with n_{field} fields (see Sect. 3.11), the block layout represents a partition of fields into subsets.[37] It is described through the argument array `field_blocks` of size `num_fields` equal to n_{field}, which indicates the block to which each field is assigned; by default, the one-block case is used. e.g., For the Stokes problem in Example 3.2, one can consider a monolithic block layout with only one block that includes both the velocity and pressure field (`field_blocks`=[1,1]), or two one-field blocks (`field_blocks`=[1,2] or [2,1]). Additionally, the user must provide additional information about the diagonal blocks, namely (1) whether the block is symmetric or not, (2) whether symmetric storage wants to be used for the block or not, and (3) whether the block is positive definite, semi-positive definite, or indefinite. The user can optionally provide the array of logicals `field_coupling` (of size `num_fields` $\times$ `num_fields`); the position $(\mathtt{i}, \mathtt{j})$ determines whether the matrix entries related to trial/test functions of the FE space `i` and FE space `j` are always zero (in this case, the coupling is false) or not. For the Stokes problem and the Galerkin method, the only entry that is false (no coupling) is the pressure-pressure entry. When this array is not provided, the case by default is that all fields are coupled. It only implies more memory consumption, e.g., to store the zero entries in the pressure-pressure block for the Stokes problem.

The block layout information is stored in the data type `block_layout_t`, sketched in Listing 43, which stores the arrays `field_blocks` and `field_coupling`. It is created in the binding that creates the `fe_affine_operator_t`. It also stores a block-wise DOF numbering generated by the `fe_space_t` instance, which is instructed to do so by passing the `block_layout_t`[38] when calling its TBP `generate_global_dof_numbering`, described in Sect. 10.3.

The `fe_affine_operator_t` also holds a polymorphic pointer to an `assembler_t` instance. Its dynamic type is selected during the creation phase depending on the number of blocks, the storage layout required, and the (parallel or serial) environment. Finally, a polymorphic pointer to an instance of declared type `discrete_integration_t` is also stored (see line 11 of Listing 42). After the creation phase, the `fe_affine_operator_t` is ready for its setup. Thanks to the design of the linear algebra data structures in FEMPAR, it does not require a symbolic setup, i.e., to precompute a (potential) sparsity

[37] The actual ordering of the fields in the Cartesian FE space is determined by the user in the creation of the multi-field FE space, which must be consistent with the implementation of the discrete weak form. See, e.g., the creation of the mixed Taylor-Hood FE space in Lines 11–21 of Listing 46, where the first field is the velocity field and the second one is the pressure field, and the integration of the weak form, e.g., in Lines 34, 37, and 42 of Listing 41, where this numbering is respected.

[38] The block-wise numbering creates independently the DOF numbering of every block. Thus, DOFs of different blocks can have the same block-wise DOF label.

```
type :: block_layout_t
   private
   integer(ip)                :: num_blocks = -1
   integer(ip)                :: num_fields = -1
   integer(ip), allocatable   :: field_blocks(:)
   integer(ip), allocatable   :: num_dofs_x_block(:)
   logical     , allocatable  :: field_coupling(:,:)
   logical     , allocatable  :: block_coupling(:,:)
 contains
   procedure, non_overridable :: create                    => block_layout_create
   procedure, non_overridable :: free                      => block_layout_free
   procedure, non_overridable :: fields_coupled            => block_layout_fields_coupled
   procedure, non_overridable :: blocks_coupled            => block_layout_blocks_coupled
   ... ! Full set of setter and getter TBPs to preserve data hiding and encapsulation
end type block_layout_t
```

Listing 43. The `block_layout_t` data type.

pattern. The `numerical_setup` TBP at line 17 of Listing 42 calls the `integrate_galerkin` TBP of `discrete_integration` when the pointer to `trial_fe_space` is not associated or `integrate_petrov_galerkin` otherwise, as discussed in Sect. 11.2.

12 Driver Example for the Stokes Problem

In this section, we describe the software architecture of a driver program that approximates the solution of the Stokes problem. To this end, it implements a Galerkin FE method grounded on a "static" (i.e., non-adaptable) conforming mesh and inf-sup stable FE spaces. In particular, we consider a conforming FE space $\mathcal{V}_h \times \mathcal{Q}_h$, where $\mathcal{V}_h$ is a grad-conforming Lagrangian space of order $k+1$, and $\mathcal{Q}_h$, a grad-conforming Lagrangian space of order k, i.e., the mixed Taylor-Hood FE [5].[39]

It is up to FEMPAR users to decide how to design the software architecture of their main driver program. Any driver program has nevertheless to follow the typical stages needed in a simulation pipeline based on FEs. In the seek of uniformity, the architecture presented in Listing 44 and 45 is recommended to FEMPAR users. The main program unit relies on a number of driver-level module units, which are not part of the FEMPAR library but developed by the user specifically for the problem at hand. Each of these modules defines a driver-level derived data type and its TBPs. A central derived data type, called `stokes_driver_t` in this example, is designed to drive all the necessary steps. In particular, it offers a public TBP, called `run_simulation`, on which the driver program relies to perform the actual simulation. The driver program is therefore as simple and concise as shown in Listing 44.

[39] The pressure field belongs to $L^2(\Omega)$. Thus, a discontinuous pressure FE space could have been also considered as well. It would still be $L^2(\Omega)$-conforming. This is the case of, e.g., the conformal Crouzeix-Raviart mixed FE.

The main data type of the driver, `stokes_driver_t`, is shown in Listing 45. It is equipped with a set of member variables of type already described in previous sections; see comments on the right-hand side of each member variable. The data type `solver_t` in Line 11 does not exist in FEMPAR as such. There is actually a complete set of data types that provide interfaces to high-end third party sparse direct solvers. Besides, we have developed our own abstract implementation of iterative linear solvers (including, e.g., the conjugate gradient or GMRES Krylov subspace solvers). The convergence of these solvers can be accelerated using advanced preconditioners grounded on the Multilevel Balancing Domain Decomposition by Constraints (MLBDDC) preconditioner [34, 37]. The description of the linear solvers software subsystem deserves considerable space and is postponed to a future work. In this example, it has to be understood as a data type that provides the necessary services required to implement the `solve_system` TBP at Line 20 of Listing 45. The data type `stokes_conditions_t` at Line 9 extends `conditions_t` in Sect. 10.4. It encodes the strong Dirichlet boundary conditions data for this particular operator. The member variable `parameter_list` (see Line 4) is a parameter dictionary of $<key, value>$ pairs. Its implementation is provided as a stand-alone external software library called FPL [86]. The member variable `stokes_parameters` (see Line 3) is a user-defined data type that encapsulates the interaction with a command line parser provided by the FLAP software package [94]. Both of them are used to implement the TBP in Line 14, which parses the arguments given by the user in the command line, and transfers them into the aforementioned `parameter_list` member variable.

The `run_simulation` TBP (called from the main program in Line 8 of Listing 44) is implemented with the help of the private TBPs in Lines 17–21 of Listing 45. The `setup_triangulation` TBP invokes the `create` TBP of `static_triangulation_t`. Depending on the command-line parameter values, the user may select to automatically generate a structured/uniform triangulation

```
program stokes_driver
  use fempar_names
  use stokes_driver_names
  implicit none
  type(stokes_driver_t) :: stokes_driver
  call fempar_init()
  call stokes_driver%parse_command_line_parameters()
  call stokes_driver%run_simulation()
  call stokes_driver%free()
  call fempar_finalize()
end program stokes_driver
```

Listing 44. The main program for the solution of the Stokes problem.

```
type stokes_driver_t
   private
   type(stokes_params_t)                    :: stokes_parameters
   type(parameterlist_t) , pointer          :: parameter_list
   type(static_triangulation_t)             :: triangulation       ! See Sect. 7
   type(fe_space_t)                         :: fe_space            ! See Sect. 10
   type(p_reference_fe_t), allocatable      :: reference_fes(:)    ! See Sect. 6
   type(stokes_discrete_integration_t)      :: stokes_integration  ! See Sect. 11.2
   type(stokes_conditions_t)                :: stokes_conditions   ! See Sect. 10.4
   type(fe_affine_operator_t)               :: fe_affine_operator  ! See Sect. 11.3
   type(solver_t)                           :: solver              ! To be covered in a future work
   type(fe_function_t)                      :: solution            ! See Sect. 10.5
 contains
   procedure :: parse_command_line_parameters
   procedure :: run_simulation
   procedure :: free
   procedure , private :: setup_triangulation
   procedure , private :: setup_fe_space
   procedure , private :: setup_fe_affine_operator
   procedure , private :: solve_system
   procedure , private :: write_solution
end type stokes_driver_t
```

Listing 45. The main data type of the Stokes driver.

for simple domains (e.g., a unit cube), currently of brick (quadrilateral or hexahedral) cells, or read it from a mesh data file, e.g., using the GiD unstructured mesh generator [91]. The FE space is built in `setup_fe_space` TBP, sketched in Listing 46.

An array with base type `p_reference_fe_t`, a data type that wraps a polymorphic pointer to a `reference_fe_t` instance, is allocated in Line 8 of Listing 46. The `reference_fe_t` instances for the velocity and pressure fields are created by calling `make_reference_fe` in Lines 11 and 21, respectively; see Sect. 6.4. The interpolation order of the numerical scheme is read from command-line in Line 5. We select `order` equal to $k+1$ and k in Lines 11 and 21, respectively. The dummy argument `continuity` determines whether $\mathcal{X}$ admits a trace operator. In this particular example, we could consider `continuity=.false.` if we wanted to use a discontinuous pressure space. The `create` TBP of `fe_space_t` (Line 35) performs the composition of the reference FEs to build the Cartesian product space $\mathcal{X}_h$. Finally, we call the `set_up_cell_integration` TBP of `fe_space_t` in Line 38 to set up all the data structures required to evaluate cell integrals in Listing 40.

The implementation of the `setup_fe_affine_operator` binding is shown in Listing 47. It first invokes the `create` TBP of `fe_affine_operator_t` in Line 6. We state monolithic storage for the global coefficient matrix (Line 13), that it is symmetric (Line 9), that we want symmetric storage, i.e., to only store its upper triangle (Line 8), and the fact that it is indefinite (Line 10). The definition of `field_coupling` in Line 14 reflects that the pressure diagonal block is null. We also pass an instance of `fe_space_t` in Line 11 and an instance of the subclass `stokes_integration_t` in Line 12.

Before we set up the operator in Line 27, we create a `fe_function_t` instance in Line 18. In Line 19, by means of the services provided by `fe_space_t`, we interpolate the analytical function to be prescribed on the boundary for the velocity field (retrieved from `stokes_conditions`), fixing the strong Dirichlet DOFs of the `fe_function_t` instance at hand. As a result, this FE function represents $E_h u_{\mathrm{D}}$, with the zero extension to free DOFs; see Sect. 10.4. This FE function is passed to the `stokes_integration_t` instance in Line 24. Finally, we trigger the operator auto-construction in Line 27.

The `solve_system` TBP (see Line 20 of Listing 45) invokes either a direct or preconditioned iterative solver to obtain the free DOFs nodal values of our FE function (see Sect. 10.5). Provided that `this%solution` on input to `solve_system` is such that it vanishes on free DOFs (see discussion in previous paragraph), a common practice used in `FEMPAR` drivers to save space is to re-use the space devoted for free DOFs in `this%solution` to store the free DOFs nodal values of the solution of the problem at hand. We stress that all solvers in `FEMPAR` are such that they only solve for free DOFs. In our experience, this decision dramatically simplifies the development of some preconditioners, provided that they can be developed without taking care of strong Dirichlet boundary conditions.

Finally, the `write_solution` TBP (see Line 21 of Listing 45) is in charge of the generation of simulation results in data files for later visualization using, e.g., VisIt [95] or Paraview [96]. To this end, `write_solution` relies on a format independent, extensible abstraction, referred to as `output_handler_t`. It lets the user to register an arbitrary number of FE functions (together with the corresponding FE space these functions were generated from) and cell data arrays (e.g., material properties or error estimator indicators), to be output in the appropriate format for later visualization. Among its responsibilities, this (abstract) data type generates the data to be written to the (potentially parallel-distributed) file system in neutral, cell-oriented data structures, dealing with (potentially) non-conforming (discontinuous), and variable degree FE spaces among cells. The user may also select to apply a differential operator to the FE function, such as divergence, gradient or curl, which involve further calculations to be performed on each cell, or to customize those cells to be output (e.g., only those that belong to the interior of the geometry in unfitted FE simulations) via their own implementation of cell iterators.

The generation of the actual data files in the appropriate format is in charge of the implementations (extensions) of `output_handler_t`. `FEMPAR` currently offers two implementations of `output_handler_t` (although many others could be implemented as well by the growing community of `FEMPAR` developers given the extensible software architecture designed). `vtk_output_handler_t` lets the user to generate their data in the standard-open model VTK [97]. It currently relies on `Lib_VTK_IO` [98], which (by now) does not actually exploit parallel MPI I/O but instead uses a naive single file per MPI task scheme. `vtk_output_handler_t` is

```fortran
subroutine setup_fe_space(this)
  class(stokes_driver_t), intent(inout) :: this
  ...! Local variables declaration

  interpolation_order = this%get_interpolation_order()
  num_dimensions = this%triangulation%get_num_dimensions()

  allocate ( this%reference_fes(2) )

  ! Velocity field reference FE
  this%reference_fes(1) = make_reference_fe(topology        = this%triangulation%get_topology(), &
                                            fe_type         = fe_type_lagrangian,    &
                                            num_dimensions  = num_dimensions,        &
                                            order           = interpolation_order+1, &
                                            field_type      = field_type_vector,     &
                                            conformity      = .true.,                &
                                            continuity      = .true. )


  ! Pressure field reference FE
  this%reference_fes(2) = make_reference_fe(topology        = this%triangulation%get_topology(), &
                                             fe_type        = fe_type_lagrangian,    &
                                             num_dimensions = num_dimensions,        &
                                             order          = interpolation_order,   &
                                             field_type     = field_type_scalar,     &
                                             conformity     = .true.,                &
                                             continuity     = .true. )

  num_components = this%reference_fes(1)%p%get_num_components()+ &
                   this%reference_fes(2)%p%get_num_components()

  call this%stokes_conditions%set_num_dimensions(num_dimensions)
  call this%stokes_conditions%set_num_components(num_components)

  call this%fe_space%create(triangulation  = this%triangulation,      &
       &                    conditions     = this%stokes_conditions, &
       &                    reference_fes  = this%reference_fes)
  call this%fe_space%set_up_cell_integration()
end subroutine setup_fe_space
```

Listing 46. The implementation of the `setup_fe_space` binding for the Stokes problem.

```
subroutine setup_fe_affine_operator (this)
  implicit none
  class(stokes_driver_t), intent(inout) :: this

  ! FE operator
  call this%fe_affine_operator%create ( &
            &    sparse_matrix_storage_format      = csr_format,              &
            &    diagonal_blocks_symmetric_storage = [ .true. ],              &
            &    diagonal_blocks_symmetric         = [ .true. ],              &
            &    diagonal_blocks_sign              = [ INDEFINITE ],          &
            &    fe_space                          = this%fe_space,           &
            &    discrete_integration              = this%stokes_integration,&
            &    field_blocks                      = [1,1],                   &
            &    field_coupling                    = &
                    reshape([.true., .true., .true., .false., [2,2]))

  ! Set up fe_function_t (this%solution), and interpolate strong Dirichlet data
  call this%solution%create(this%fe_space)
  call this%fe_space%interpolate_strong_dirichlet_values(this%solution,     &
                                                         this%stokes_conditions, &
                                                         fields_to_interpolate=[1])

  ! Pass fe_function_t with Dirichlet data, i.e., E_h u_D, to stokes_integration
  call this%stokes_integration%set_fe_function(this%solution)

  ! This call ultimately triggers the integrate generic TBP of discrete_integration_t
  call this%fe_affine_operator_setup()
end subroutine setup_fe_affine_operator
```

Listing 47. The implementation of the `setup_fe_affine_operator` binding for the Stokes problem.

therefore the recommended option for serial computations or parallel computations on a moderate number of processors. The second one, `xh5_output_handler_t`, lets the user generate their data in XDMF [99]. XDMF separates the description of the raw data, referred to as "light data", from the data itself, referred to as "heavy data". The light data is expressed using a set of XML-based constructs that are suited to represent the distributed-memory data structures in FEMPAR. XDMF in turn supports the heavy data to be stored using HDF5 [100]. HDF5 is, among others, a data model and file format designed with the parallel I/O data challenge in mind. By means of a set of supporting open source libraries, referred to as parallel HDF5 libraries, FEMPAR takes advantage of the underlying distributed file system without having to deal with the high complexity of other lower-level implementations, such as raw MPI I/O. In particular, the latter service is provided by XH5For [101], a stand-alone software library, which we developed from scratch, and lets the user to read/write parallel partitioned FEM meshes taking advantage of the Collective/Independent MPI-IO provided by the PHDF5 library for the efficient generation of the vast amount of data typically resulting from a large-scale scientific computing simulation.

13 Conclusions

In this work, we have thoroughly described the approach that we have followed in FEMPAR in order to abstract in software the numerical approximation of problems governed by PDEs using FE methods. The mathematical framework of FEs has been split into a number of (mathematically motivated) derived data types and their interaction, resulting into a well-separated, robust, and stable set of customizable software abstractions for the development of widely applicable FE solvers. These tools equip FEMPAR users with the machinery needed to perform all the steps in the simulation pipeline, including mesh import/generation, DOFs enumeration, evaluation/assembly of the algebraic system of linear equations via FE integration, solution of the linear system, and output of computational results in the appropriate format for later visualization. In order to achieve this goal, the software architecture of FEMPAR has been thoroughly designed by means of advanced OO software re-engineering techniques (including the recurrent application of OO design patterns [85, 88]) in order to increase its ease of use, extensibility, flexibility, and reusability. FEMPAR software architecture has been implemented using the latest OO features of the Fortran03/08 standard, namely, *information hiding and data encapsulation*, *inheritance via type extension*, and *dynamic run-time polymorphism*. This version of the Fortran standard is already widely (and robustly) supported by most of the compilers typically available on high-end computing environments. A judiciously set of programming techniques let us achieve a reasonable trade-off among extensibility and performance, while avoiding in most cases the computational overheads frequently associated with abstract OO software libraries.

The software abstractions covered in this work include:

- The definition of reference FEs, which relies on the concept of polytopes to define the cell topology in arbitrary dimensions, a machinery to define multi-dimensional polynomial functions of arbitrary order in an easy and automatic way, and a general procedure for the generation of the shape function bases and local DOFs.
- The global FE space abstraction, which relies on reference FE(s) and a triangulation of the physical domain. It is responsible to define the local-to-global DOF numbering, which must respect conformity (if needed). The FE space also provides tools for the numerical integration of (bi)linear forms, e.g., mappings from the reference to the physical space, etc., in cells and facets (for DG methods).
- The FE affine operator generated after the discretization of the original problem (probably after a linearization step). The FE solution is the only root (as soon as the problem is well-posed) of this operator. This operator, once the trial and test functions and the discrete (bi)linear forms of the problem at hand are defined, is represented through a matrix and a vector whose entries can be computed by numerical integration using the FE space.

`FEMPAR` has been used for more than 4 years now by a team of about 10 researchers of different research institutions and universities. During the initial OO re-design, derived data types (attributes and bindings) were gradually modified to accommodate new features that had not been considered, to fix expressivity limitations or even dependency knots of the original design. The software architecture to which we have converged, although certainly subject to future change, has been already proven to be capable to satisfy a number of users' software requirements, even when the application problems involved complex and advanced features (e.g., the development of growing geometries in 3D printing technology). We consider that this steady regime, which has been attained after years of development, and a tremendous man-month power effort, is the proof that the software abstraction in `FEMPAR` is of practical relevance not only for prospective users and developers, but also for researchers that want to learn about the OO implementation of FE methods. It has motivated the decision of the authors to promote the library as a community software project, to open it to external users and new collaborators, to publish the library in an public git repository [42], and to write this article. In particular, the architecture described here corresponds to the first public release of `FEMPAR`, to which we assigned the git tag `FEMPAR-1.0.0`.

The first public release of `FEMPAR` has almost 300K lines of (mostly) Fortran code. Thus, a document like this one, with a quite detailed description of the services provided by the library and the motivation underlying our software design, can be a very valuable resource to complement the source code, which can become overwhelming in itself. In this paper, we have restricted ourselves to the construction of FE operators for body-fitted FE spaces. However, a major (and unique compared to other FE scientific software packages available on the Internet) cornerstone of `FEMPAR` is an abstract OO framework for the implementation of widely applicable highly scalable multilevel DD solvers.[40] By letting this framework to be highly coupled with the numerical integration data structures of the application, on the one hand, and to be highly customizable, on the other, one can derive optimal preconditioners for the particular structure of the discrete operator at hand, and tackle new problems and challenges, while leveraging the distributed-memory implementation ideas [37] on which the framework is grounded on. Customizable building blocks in the framework include the fine-grid to coarse-grid DOFs aggregation, the constraint matrix underlying the imposition of continuity of coarse DOFs functionals across coarse objects, the weighting operator underlying the injection among the continuous and discontinuous spaces, and the kind of solvers to be used for the Dirichlet, Neumann constrained local problems, and the coarsest-grid global problem [103]. However, we postpone the discussion about solvers, preconditioners, data structures suitable for parallel distributed-memory computers, and other more exotic discretization techniques in `FEMPAR`, like B-splines and XFEM methods, to subsequent works.

Acknowledgements The authors want to thank Jesús Bonilla, Oriol Colomés, Eric Neiva, Hieu Nguyen, Marc Olm, Víctor Sande, and Francesc Verdugo (in alphabetical order) for their strong commitment to the `FEMPAR` project, the implementation of some of the software described in this work, and their thorough review of preliminary versions of this document. The resources needed to develop a scientific library library like `FEMPAR` would have not been feasible without excellent research funding. In this sense, SB sincerely thanks the support of the European Research Council through the Starting Grant No. 258443—*COMFUS: Computational Methods for Fusion Technology* under the the FP7 Program and the two related Proof of Concept Grant No. 640957—*FEXFEM: On a free open source extreme scale finite element software* and Proof of Concept Grant No. 737439—*NuWaSim: On a Nuclear Waste Deep Repository Simulator* under the H2020 Program. SB gratefully acknowledges the support received from the Catalan Government through the ICREA Acadèmia Research Program.

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no conflict of interest.

[40] Indeed, the multilevel DD solvers within `FEMPAR` are since 2014 in the High-Q club of the most scalable European codes, maintained by the Jülich supercomputing center [102].

References

1. Guo B, Babuska I (1986) The h-p version of the finite element method. Comput Mech 1(1):21–41
2. Ainsworth M, Oden JT (2011) A posteriori error estimation in finite element analysis. Wiley, New York
3. Melenk JM, Wohlmuth BI (2001) On residual-based a posteriori error estimation in hp-FEM. Adv Comput Math 15(1–4):311–331
4. Nedelec JC (1980) Mixed finite elements in $\mathcal{R}^{\ni}$. Numer Math 35(3):315–341
5. Brezzi F, Fortin M (1991) Mixed and hybrid finite element methods. Springer, Berlin
6. Arnold DN, Falk RS, Winther R (2006) Finite element exterior calculus, homological techniques, and applications. Acta Numer 15:1–155
7. Neilan M, Sap D (2016) Stokes elements on cubic meshes yielding divergence-free approximations. Calcolo 53(3):263–283
8. Hughes TJR, Cottrell JA, Bazilevs Y (2005) Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. Comput Methods Appl Mech Eng 194(39–41):4135–4195
9. Cockburn B, Gopalakrishnan J, Lazarov R (2009) Unified hybridization of discontinuous Galerkin, mixed, and continuous Galerkin methods for second order elliptic problems. SIAM J Numer Anal 47(2):1319–1365
10. Demkowicz L, Gopalakrishnan J (2010) A class of discontinuous Petrov-Galerkin methods. Part I: the transport equation. Comput Methods Appl Mech Eng 199(23–24):1558–1572
11. Ang J, Barrett R, Benner R, Burke D, Chan C, Cook J, Donofrio D, Hammond S, Hemmert K, Kelly S, Le H, Leung V, Resnick D, Rodrigues A, Shalf J, Stark D, Unat D, Wright N (2014) Abstract machine models and proxy architectures for exascale computing. In: Co-HPC'14 Proceedings of the 1st international workshop on hardware–software co-design for high performance computing. IEEE, pp 25–32
12. Gropp W (2015) Is MPI+X enough for exascale? Keynote for international high performance computing forum, Tianjin, China
13. Kale LV, Krishnan S (1993) CHARM++: a portable concurrent object oriented system based on C++. ACM SIGPLAN Not 28(10):91–108
14. Bauer M, Treichler S, Slaughter E, Aiken A (2012) Legion: expressing locality and independence with logical regions. In: Proceedings of the international conference on high performance computing, networking, storage and analysis, SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press, pp 66:1–66:11
15. Kaiser H, Heller T, Adelstein-Lelbach B, Serio A, Fey D (2014) Hpx: A task based programming model in a global address space. In Proceedings of the 8th international conference on partitioned global address space programming models, PGAS '14, New York, NY, USA, ACM, pp 6:1–6:11
16. Janine Bennett PI, Robert Clay PM, Baker G, Gamell M, Hollman D, Knight S, Kolla H, Sjaardema G, Slattengren N, Teranishi K et al (2015) ASC ATDM level 2 milestone# 5325: asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical Report SAND2015-8312, Sandia National Laboratories
17. Bangerth W, Hartmann R, Kanschat G (2007) deal. II–a general-purpose object-oriented finite element library. ACM Trans Math Softw 33(4):24
18. Bangerth W, Davydov D, Heister T, Heltai L, Kanschat G, Kronbichler M, Maier M, Turcksin B, Wells D (2016) The deal. II library, version 8.4. J Numer Math 24:135–141
19. Alnæs M, Blechta J, Hake J, Johansson A, Kehlet B, Logg A, Richardson C, Ring J, Rognes ME, Wells GN (2015) The FEniCS project version 1.5. Arch Numer Softw 3(100):9–23
20. Bauman P, Stogner R (2016) GRINS: a multiphysics framework based on the libmesh finite element library. SIAM J Sci Comput 38(5):S78–S100
21. Cantwell CD, Moxey D, Comerford A, Bolis A, Rocco G, Mengaldo G, De Grazia D, Yakovlev S, Lombard JE, Ekelschot D, Jordi B, Xu H, Mohamied Y, Eskilsson C, Nelson B, Vos P, Biotto C, Kirby RM, Sherwin SJ (2015) Nektar++: an open-source spectral/element framework. Comput Phys Commun 192:205–219
22. MOOSE (Multiphysics Object-Oriented Simulation Environment) Framework. http://mooseframework.org/
23. MFEM—a free, lightweight, scalable C++ library for finite element methods. http://mfem.org/
24. Hecht F (2012) New development in FreeFem++. J Numer Math 20(3–4):251–265
25. Dedner A, Nolte M (2012) Construction of local finite element spaces using the generic reference elements. In: Dedner A, Flemisch B, Klöfkorn R (eds) Advances in DUNE. Springer, Berlin, pp 3–16
26. Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, Dalcin L, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Rupp K, Smith BF, Zampini S, Zhang H, Zhang H (2016a) PETSc web page
27. Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, Dalcin L, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Rupp K, Smith BF, Zampini S, Zhang H, Zhang H (2016b) PETSc users manual. Technical report ANL-95/11—Revision 3.7, Argonne National Laboratory
28. Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP (eds) Modern software tools in scientific computing. Birkhäuser Press, Berlin, pp 163–202
29. Falgout RD, Yang UM (2002) hypre: a library of high performance preconditioners. In: Computational science—ICCS 2002. Springer, Berlin, pp 632–641
30. Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET, Salinger AG, Thornquist HK, Tuminaro RS, Willenbring JM, Williams A, Stanley KS (2005) An overview of the trilinos project. ACM Trans Math Softw 31(3):397–423
31. The Trilinos Project. https://trilinos.org
32. Toselli A (2006) Dual-primal FETI algorithms for edge finite-element approximations in 3D. IMA J Numer Anal 26(1):96–130
33. Dohrmann CR (2007) An approximate BDDC preconditioner. Numer Linear Algebra Appl 14(2):149–168
34. Badia S, Martín AF, Principe J (2015) On the scalability of inexact balancing domain decomposition by constraints with overlapped coarse/fine corrections. Parallel Comput 50:1–24
35. Efendiev Y, Hou TY (2009) Multiscale finite element methods: theory and applications. Springer, New York
36. Klawonn A, Lanser M, Rheinbach O (2015) Toward extremely scalable nonlinear domain decomposition methods for elliptic

partial differential equations. SIAM J Sci Comput 37(6):C667–C696
37. Badia S, Martín A, Principe J (2016) Multilevel balancing domain decomposition at extreme scales. SIAM J Sci Comput, pp C22–C52
38. Zampini S (2016) PCBDDC: a class of robust dual-primal methods in PETSc. SIAM J Sci Comput 38(5):S282–S306
39. Badia S, Olm M (2017) Space–time balancing domain decomposition. SIAM J Sci Comput 39(2):C194–C213
40. Brune P, Knepley M, Smith B, Tu X (2015) Composing scalable nonlinear algebraic solvers. SIAM Rev 57(4):535–565
41. Falgout R, Friedhoff S, Kolev T, MacLachlan S, Schroder J (2014) Parallel time integration with multigrid. SIAM J Sci Comput 36(6):C635–C661
42. FEMPAR: Finite Element Multiphysics PARallel solvers. https://gitlab.com/fempar/fempar
43. Elman HC, Silvester DJ, Wathen AJ (2005) Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics. Oxford University Press, Oxford
44. Badia S, Martín AF, Planas R (2014) Block recursive LU preconditioners for the thermally coupled incompressible inductionless MHD problem. J Comput Phys 274:562–591
45. Cyr E, Shadid J, Tuminaro R (2016) Teko: a block preconditioning capability with concrete example applications in Navier–Stokes and MHD. SIAM J Sci Comput 38(5):S307–S331
46. Colomés O, Badia S, Codina R, Principe J (2015) Assessment of variational multiscale models for the large eddy simulation of turbulent incompressible flows. Comput Methods Appl Mech Eng 285:32–63
47. Colomés O, Badia S (2016) Segregated Runge–Kutta methods for the incompressible Navier–Stokes equations. Int J Numer Methods Eng 105(5):372–400
48. Colomés O, Badia S, Principe J (2016) Mixed finite element methods with convection stabilization for the large eddy simulation of incompressible turbulent flows. Comput Methods Appl Mech Eng 304:294–318
49. Colomés O, Badia S (2017) Segregated Runge–Kutta time integration of convection-stabilized mixed finite element schemes for wall-unresolved LES of incompressible flows. Comput Methods Appl Mech Eng 313:189–215
50. Badia S, Codina R, Planas R (2013a) On an unconditionally convergent stabilized finite element approximation of resistive magnetohydrodynamics. J Comput Phys 234:399–416
51. Badia S, Planas R, Gutiérrez-Santacreu JV (2013b) Unconditionally stable operator splitting algorithms for the incompressible magnetohydrodynamics system discretized by a stabilized finite element formulation based on projections. Int J Numer Methods Eng 93(3):302–328
52. Planas R, Badia S, Codina R (2011) Approximation of the inductionless MHD problem using a stabilized finite element method. J Comput Phys 230(8):2977–2996
53. Smolentsev S, Badia S, Bhattacharyay R, Bühler L, Chen L, Huang Q, Jin HG, Krasnov D, Lee DW, de les Valls EM, Mistrangelo C, Munipalli R, Ni M-J, Pashkevich D, Patel A, Pulugundla G, Satyamurthy P, Snegirev A, Sviridov V, Swain P, Zhou T, Zikanov O (2015) An approach to verification and validation of MHD codes for fusion applications. Fusion Eng Des 100:65–72
54. Badia S, Codina R, Planas R (2015) Analysis of an unconditionally convergent stabilized finite element formulation for incompressible magnetohydrodynamics. Arch Comput Methods Eng 22(4):621–636
55. Badia S, Hierro A (2015) On discrete maximum principles for discontinuous Galerkin methods. Comput Methods Appl Mech Eng 286:107–122
56. Badia S, Hierro A (2014) On monotonicity-preserving stabilized finite element approximations of transport problems. SIAM J Sci Comput 36(6):A2673–A2697
57. Hierro A, Badia S, Kus P (2016) Shock capturing techniques for adaptive finite elements. Comput Methods Appl Mech Eng 309:532–553
58. Badia S, Bonilla J (2017) Monotonicity-preserving finite element schemes based on differentiable nonlinear stabilization. Comput Methods Appl Mech Eng 313:133–158
59. Badia S, Bonilla J, Hierro A (2017) Differentiable monotonicity-preserving schemes for discontinuous Galerkin methods on arbitrary meshes. Comput Methods Appl Mech Eng 320:582–605
60. Badia S, Verdugo F (2017) Robust and scalable domain decomposition solvers for unfitted finite element methods. arXiv:1703.06323 [math]
61. Chiumenti M, Neiva E, Salsi E, Cervera M, Badia S, Davies C, Chen Z, Lee C (2017) Numerical modelling and experimental validation in selective laser melting (submitted)
62. Badia S, Martín AF, Principe J (2013) Implementation and scalability analysis of balancing domain decomposition methods. Arch Comput Methods Eng 20(3):239–262
63. Badia S, Martín AF, Príncipe J (2013) Enhanced balancing Neumann–Neumann preconditioning in computational fluid and solid mechanics. Int J Numer Meth Eng 96(4):203–230
64. Badia S, Nguyen H (2016) Balancing domain decomposition by constraints and perturbation. SIAM J Numer Anal 54(6):3436–3464
65. Badia S, Martín AF, Nguyen H (2016) Physics-based balancing domain decomposition by constraints for heterogeneous problems. Working paper or preprint
66. Badia S, Martín A, Principe J (2014) A highly scalable parallel implementation of balancing domain decomposition by constraints. SIAM J Sci Comput 36(2):C190–C218
67. Adams JC, Brainerd WS, Hendrickson RA, Maine RE, Martin JT, Smith BT (2009) The Fortran 2003 handbook. Springer, London,
68. Rouson D, Xia J, Xu X (2011) Scientific software design: the object-oriented way, 1st edn. Cambridge University Press, New York
69. Ern A, Guermond J-L (2004) Theory and practice of finite elements. Springer, Berlin
70. Brenner SC, Scott R (2010) The mathematical theory of finite element methods. Springer, softcover reprint of hardcover 3rd ed. 2008 edn
71. Quarteroni A (2014) Numerical models for differential problems. Springer Milan, Milano
72. Monk P (2003) Finite element methods for Maxwell's equations. Clarendon Press, Oxford
73. Duffy M (1982) Quadrature over a pyramid or cube of integrands with a singularity at a vertex. SIAM J Numer Anal 19(6):1260–1262
74. Dunavant DA (1985) High degree efficient symmetrical Gaussian quadrature rules for the triangle. Int J Numer Meth Eng 21(6):1129–1148
75. Olm M, Badia S, Martín AF (2017) Simulation of high temperature superconductors and experimental validation. arXiv:1707.09783 [physics]
76. Badia S, Quaini A, Quarteroni A (2008a) Modular vs. non-modular preconditioners for fluid-structure systems with large added-mass effect. Comput Methods Appl Mech Eng 197(49–50):4216–4232
77. Badia S, Nobile F, Vergara C (2008b) Fluid-structure partitioned procedures based on Robin transmission conditions. J Comput Phys 227(14):7027–7051

78. Badia S, Quaini A, Quarteroni A (2008c) Splitting methods based on algebraic factorization for fluid–structure interaction. SIAM J Sci Comput 30(4):1778
79. Badia S, Nobile F, Vergara C (2009) Robin–Robin preconditioned Krylov methods for fluid-structure interaction problems. Comput Methods Appl Mech Eng 198(33–36):2768–2784
80. Rognes M, Kirby R, Logg A (2009) Efficient assembly of H(div) and H(curl) conforming finite elements. SIAM J Sci Comput 31(6):4130–4151
81. Agelek R, Anderson M, Bangerth W, Barth W (2017) On orienting edges of unstructured two- and three-dimensional meshes. ACM Trans Math Softw (to appear)
82. Bangerth W, Kayser-Herold O (2009) Data structures and requirements for hp finite element software. ACM Trans Math Softw 36(1):1–31
83. Badia S, Baiges J (2013) Adaptive finite element simulation of incompressible flows by hybrid continuous–discontinuous Galerkin formulations. SIAM J Sci Comput 35(1):A491–A516
84. Dryja M, Galvis J, Sarkis M (2007) BDDC methods for discontinuous Galerkin discretization of elliptic problems. J Complex 23(4–6):715–739
85. Freeman E, Robson E, Sierra K, Bates B (eds) (2004) Head first design patterns. O'Reilly, Sebastopol
86. FPL—Fortran parameter list. https://gitlab.com/fempar/FPL
87. Beall MW, Shephard MS (1997) A general topology-based mesh data structure. Int J Numer Meth Eng 40(9):1573–1596
88. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston
89. Bangerth W, Burstedde C, Heister T, Kronbichler M (2012) Algorithms and data structures for massively parallel generic adaptive finite element codes. ACM Trans Math Softw 38(2):14:1–14:28
90. Burstedde C, Wilcox LC, Ghattas O (2011) p4est : scalable algorithms for parallel adaptive mesh refinement on forests of octrees. SIAM J Sci Comput 33(3):1103–1133
91. GiD—the personal pre and post processor. http://www.gidhome.com
92. Filippone S, Buttari A (2012) Object-oriented techniques for sparse matrix computations in Fortran 2003. ACM Trans Math Softw 38(4):23:1–23:20
93. Saad Y (2003) Iterative methods for sparse linear systems, 2nd edn. Society for Industrial and Applied Mathematics
94. FLAP—Fortran command Line Arguments Parser for poor people. https://github.com/szaghi/FLAP
95. Childs H, Brugger E, Whitlock B, Meredith J, Ahern S, Pugmire D, Biagas K, Miller M, Harrison C, Weber GH, Krishnan H, Fogal T, Sanderson A, Garth C, Bethel EW, Camp D, Rübel O, Durant M, Favre JM, Navrátil P (2012) VisIt: an end-user tool for visualizing and analyzing very large data. In: High performance visualization-enabling extreme-scale scientific insight, pp 357–372
96. Ayachit U (2015) The paraview guide: a parallel visualization application. Kitware Inc, Clifton Park
97. Schroeder W, Martin KM, Lorensen WE (1998) The visualization toolkit: an object-oriented approach to 3D graphics, 2nd edn. Prentice-Hall, Inc., Upper Saddle River
98. Lib_VTK_IO—Pure Fortran (2003+) library to write and read data conforming the VTK standard. https://gitlab.com/fempar/Lib_VTK_IO
99. XDMF—eXtensible data model and format. http://www.xdmf.org/index.php/Main_Page
100. The HDF Group. Hierarchical data format version 5. http://www.hdfgroup.org/HDF5, 2000–2017
101. XH5For—XDMF parallel partitioned mesh Input/Output on top of HDF5. https://gitlab.com/fempar/XH5For
102. Brömmel D, Wylie BJN, Frings W (2015) JUQUEEN extreme scaling workshop 2015. Technical Report FZJ-2015-01645, Jülich Supercomputing Center
103. Dohrmann CR (2003) A preconditioner for substructuring based on constrained energy minimization. SIAM J Sci Comput 25(1):246–258