GeoPDEs: a research tool for IsoGeometric Analysis of PDEs

C. de Falco[a], A. Reali[b,c,d], R. Vázquez[c]

[a] MOX-Modeling and Scientific Computing, Dipartimento di Matematica, Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133 Milano, Italy.
[b] Dipartimento di Meccanica Strutturale, Università degli Studi di Pavia, Via Ferrata, 1, 27100 Pavia, Italy.
[c] Istituto di Matematica Applicata e Tecnologie Informatiche del CNR, Via Ferrata, 1, 27100 Pavia, Italy.
[d] EUCENTRE, via Ferrata 1, 27100, Pavia, Italy.

Abstract

GeoPDEs (http://geopdes.sourceforge.net) is a suite of free software tools for applications on Isogeometric Analysis (IGA). Its main focus is on providing a common framework for the implementation of the many IGA methods for the discretization of partial differential equations currently studied, mainly based on B-Splines and Non-Uniform Rational B-Splines (NURBS), while being flexible enough to allow users to implement new and more general methods with a relatively small effort. This paper presents the philosophy at the basis of the design of GeoPDEs and its relation to a quite comprehensive, abstract definition of IGA.

Key words: Isogeometric Analysis, Finite Element Method, NURBS, B-Splines, Matlab, Octave

1. Introduction

IsoGeometric Analysis (IGA) is a (relatively) recent technique for the discretization of Partial Differential Equations (PDEs), introduced by Hughes et al. in [1]. The main feature of the method is the ability to maintain the same *exact* description of the computational domain geometry throughout the analysis process, including refinement. In the original presentation of IGA as described in [1] (where one of the main focuses is on structural analysis), the exact geometry representation is obtained by invoking the *isoparametric* concept, that is, by using the same class of functions most commonly used for geometry parameterization in Computer Aided Geometric Design (CAGD), namely Non-Uniform Rational B-Splines (NURBS), for the PDE solution space. As NURBS spaces include as a special case the piece-wise polynomial spaces commonly used in the Finite Element Method (FEM), IGA can be understood as a generalization of standard FEMs where more regular functions are

Email addresses: carlo.defalco@polimi.it (C. de Falco), alessandro.reali@unipv.it (A. Reali), vazquez@imati.cnr.it (R. Vázquez)

Preprint submitted to Elsevier *September 2, 2011*

employed. This higher regularity has been shown to lead to various advantages of IGA over FEM in addition to the better handling of CAGD geometries, e.g., better convergence on a per-degree-of-freedom basis, better approximation of the eigenspectrum of the Laplacian and biharmonic operators [2] or the ability to deal with higher order differential operators [3, 4].

Since its introduction, IGA has evolved along different directions. On one hand, function spaces other than NURBS or B-Splines have been considered, such as T-splines [5, 6], which allow for local refinement, or so-called generalized B-Splines [7, 8], that allow to better handle important classes of curves and surfaces. On the other hand, especially for applications where the properties descending from a strictly isoparametric approach are not of such paramount importance as for structural analysis, the isoparametric constraint has been relaxed in order to produce B-Spline generalizations of edge and face finite elements [9] which have been successfully applied to problems in electromagnetism [9, 10] and incompressible fluid dynamics [11]. A comprehensive reference on IGA advantages and successful applications is the recent book by Cottrell et al. [12].

IGA is indeed a powerful method, which has been shown to outperform FEM in every numerical test we have tried so far. The large number of papers, presentations and dedicated symposia at international conferences on the topic clearly indicate the great interest that IGA has been drawing from the PDE discretization and CAGD research communities. Nonetheless, researchers from both such areas often refrain from getting directly involved in IGA because of their reluctance to invest the time and effort required to get acquainted with the basics of each other's field. Furthermore, for those working on complex engineering applications, the amount of work required to adapt their existing codes to the IGA framework would need to be carefully estimated before undertaking such a task[1].

Bearing in mind the above-mentioned considerations, we have decided to implement and distribute the `GeoPDEs` software suite [15] with multiple objectives. First, it is meant to serve as an entry point for researchers who wish to get acquainted with the practical issues that implementing an IGA code involves. Furthermore, by decoupling as much as possible the various aspects of IGA-related algorithms (e.g., basis function definition and evaluation, differential operator discretization, choice of function spaces, numerical quadrature, etc.), it intends to allow the users mainly interested in one of such aspects to test their ideas in a complete solver environment while having to deal as little as possible with issues that fall outside their area of expertise. Finally, it is meant to be used as a rapid prototyping and

[1]Techniques meant to ease the adaptation of existing FEM software to the IGA setting have been presented in [13, 14]

testing tool for new IGA algorithms.

The design of the GeoPDEs suite descends directly from the objectives stated above, in particular the decision to implement it as an open and free software platform is driven by the intention to use it as a way to communicate and share ideas related to IGA among researchers from different areas; similarly, to allow its use as a fast prototyping tool, it has been implemented mainly in an interpreted language, the particular language of choice being Matlab which is a *de facto* standard for prototyping of numerical algorithms. Finally, to maximize its accessibility and availability, GeoPDEs has been especially optimized to work in the free GNU/Octave interpreter [16].

The present paper is not expected to be a full user guide to GeoPDEs. Rather, it is intended to explain its architecture, its design and its main features, and to provide various examples of how to use and extend it. Furthermore, as in the implementation of GeoPDEs, given its design objectives, properties such as code clarity, generality and extensibility have been often favored over efficiency and scalability, we will, at times, explain how some of the current speed or memory bottlenecks may be avoided.

The paper is structured as follows. In Section 2 we present an overview on IGA, and in Section 3 we focus on its application to the particular case of Poisson's problem. The backbone of the code is presented in Section 4, with a detailed description of its main data structures and the help of a simple example. The way to modify the code for developing new methods is explained in Section 5, and in particular we show modified versions to solve linear elasticity, Stokes, and Maxwell's equations. Finally, in Section 6 we show the extension of the code to problems on NURBS multipatch geometries.

2. A brief overview on Isogeometric Analysis

As stated in the introduction, the initial concept of IGA has been extended and generalized in many ways. In this paper, we intend to demonstrate the ability of GeoPDEs to accommodate within its framework various different IGA formulations. To this end, we find it convenient to briefly present, in the next section, the main concepts of IGA in a general framework. To introduce some useful notation, we will as well introduce, in Section 2.2 NURBS and B-Splines, which have been the first and simplest (and, so far, most successful) functions adopted in IGA.

2.1. Isogeometric Analysis: a general framework

The goal of IGA, as it is also for FEM, is the numerical approximation of the solution of PDEs. In both approaches the PDEs are numerically solved using a Galerkin procedure, i.e.,

the equations are written in their equivalent variational formulations, and a solution is sought in a finite dimensional space with good approximation properties. The main difference between the two methodologies is that in FEM the basis functions and the computational geometry (i.e., the mesh) are defined using piecewise polynomials, whereas in IGA the computational geometry is defined exactly from the information and the basis functions (e.g., NURBS, T-splines, or generalized B-Splines) given by CAD.

Let us consider a three-dimensional case, where we assume that the physical domain $\Omega \subset \mathbb{R}^3$ is open, bounded and Lipschitz. We also assume that such a domain can be exactly described through a parameterization of the form

$$\mathbf{F} : \widehat{\Omega} \longrightarrow \Omega, \tag{1}$$

where $\widehat{\Omega}$ is some parametric domain (e.g., the unit cube), and the value of the parameterization can be computed with the information given by the CAD software. The parameterization $\mathbf{F}$ is assumed to be smooth with piecewise smooth inverse.

Now, let V be a Hilbert space of functions defined in Ω, and let V' be its dual. We denote by $(\cdot, \cdot)$ the scalar product in V, and by $\langle \cdot, \cdot \rangle$ the duality pairing between V' and V. We study now the following source and eigenvalue problems. Given the bilinear form $a : V \times V \to \mathbb{R}$ and the linear functional $l \in V'$, the variational formulation of the source problem reads:

Find $u \in V$ such that

$$a(u, v) = \langle l, v \rangle \quad \forall v \in V, \tag{2}$$

whereas the variational formulation of the eigenvalue problem is:

Find $\lambda \in \mathbb{R} \setminus \{0\}$, and $u \in V$, $u \neq 0$, such that

$$a(u, v) = \lambda(u, v) \quad \forall v \in V. \tag{3}$$

The Galerkin procedure, then, consists of approximating the infinite-dimensional space V by a finite-dimensional space V_h, and to solve the corresponding discrete source problem:

Find $u_h \in V_h$ such that

$$a(u_h, v_h) = \langle l, v_h \rangle \quad \forall v_h \in V_h, \tag{4}$$

or discrete eigenvalue problem:

Find $\lambda \in \mathbb{R} \setminus \{0\}$, and $u_h \in V$, $u_h \neq 0$, such that

$$a(u_h, v_h) = \lambda(u_h, v_h) \quad \forall v_h \in V_h. \tag{5}$$

In standard FEM, the space V_h is a space of piecewise polynomials. In an IGA context, as introduced in [1], this space is formed by, e.g., NURBS functions. In our framework we prefer to define this space in the following general way

$$V_h := \{v_h \in V : \widehat{v}_h = \iota(v_h) \in \widehat{V}_h\} \equiv \{v_h \in V : v_h = \iota^{-1}(\widehat{v}_h),\, \widehat{v}_h \in \widehat{V}_h\},$$

where ι is a proper pull-back, defined from the parameterization (1) (see [9] and references therein), and $\widehat{V}_h$ is a discrete space defined in the parametric domain $\widehat{\Omega}$.

Finally, it is worth to remind how problem (4) is solved. Let $\{\widehat{v}_j\}_{j\in\mathcal{J}}$ be a basis for $\widehat{V}_h$, with $\mathcal{J}$ a proper set of indices. With the assumptions made on $\mathbf{F}$, the set $\{\iota^{-1}(\widehat{v}_j)\}_{j\in\mathcal{J}} \equiv \{v_j\}_{j\in\mathcal{J}}$ is a basis for V_h. Hence, the discrete solution of problem (4) can be written as

$$u_h = \sum_{j\in\mathcal{J}} \alpha_j v_j = \sum_{j\in\mathcal{J}} \alpha_j \iota^{-1}(\widehat{v}_j).$$

Substituting this expression into (4), and testing against every basis function $v_i \in V_h$, we obtain a linear system of equations where the coefficients α_j are the unknowns, and the entries of the matrix and the right-hand side are $a(v_i, v_j)$ and $\langle l, v_i\rangle$, respectively. These terms have to be computed using suitable quadrature rules for numerical integration.

2.2. Definition of B-Splines and NURBS

We now propose a brief introduction on B-Splines and NURBS, with the only aim of showing some notations and the most basic concepts. A more detailed treatment of this topic can be found, for instance, in [17, 18] for B-Splines, and in [19] for NURBS functions and geometric entities.

Given two positive integers p and n, we introduce the (non-decreasing) knot vector $\Xi := \{0 = \xi_1, \xi_2, \dots, \xi_{n+p+1} = 1\}$. We also introduce the vector $\{\zeta_1, \dots, \zeta_m\}$ of knots without repetitions, and the vector $\{r_1, \dots, r_m\}$ of their corresponding multiplicities, such that

$$\Xi = \{\underbrace{\zeta_1, \dots, \zeta_1}_{r_1 \text{ times}}, \underbrace{\zeta_2, \dots, \zeta_2}_{r_2 \text{ times}}, \dots, \underbrace{\zeta_m, \dots, \zeta_m}_{r_m \text{ times}}\},$$

with $\sum_{i=1}^m r_i = n + p + 1$. Univariate B-Spline basis functions are defined from Ξ starting from piecewise constants as:

$$B_{i,0}(\widehat{x}) = \begin{cases} 1, & \text{if } \xi_i \le \widehat{x} < \xi_{i+1}, \\ 0, & \text{otherwise,} \end{cases} \tag{6}$$

and then, for a degree $p > 0$, they are defined recursively as follows ([17, Ch. IX]):

$$B_{i,p}(\widehat{x}) = \frac{\widehat{x} - \xi_i}{\xi_{i+p} - \xi_i} B_{i,p-1}(\widehat{x}) + \frac{\xi_{i+p+1} - \widehat{x}}{\xi_{i+p+1} - \xi_{i+1}} B_{i+1,p-1}(\widehat{x}). \tag{7}$$

We remark that, in the expression above, whenever one of the denominators is zero, the corresponding function $B_{i,p-1}$, given by (6) or (7), is also zero, and no contribution is added. These n B-Spline functions form a partition of unity and they are linearly independent. We will denote the space they span by $S^p_{\boldsymbol{\alpha}}(\Xi)$, or simply $S^p_{\boldsymbol{\alpha}}$, with $\boldsymbol{\alpha} = \{\alpha_1, \ldots, \alpha_m\}$, with $\alpha_i := p - r_i$. This is the space of piecewise polynomials of degree p with α_i continuous derivatives at the breakpoints.

In the following, we always assume that the knot vector Ξ is "open", that is, the first and last knots appear exactly $p+1$ times. In this case the first and last basis functions are interpolatory at the parametric coordinates 0 and 1, respectively. Moreover, we highlight that the maximum allowed multiplicity for the internal knots is $r = p+1$, corresponding to the case of discontinuous functions, that is, $\alpha_i = -1$.

Remark 2.1. *The use of discontinuous B-Splines (or NURBS) may not be very interesting from the point of view of CAD, but, in fact, they have been already successfully used in the simulation of geometries with cracks [20].*

The previous definition is easily generalized to the two- and three-dimensional cases by means of tensor products. For instance, in the trivariate case, given the degrees p_d, the integers n_d and the knot vectors Ξ_d $(d = 1, 2, 3)$, the B-Spline basis functions are defined as

$$B_{\mathbf{i}}(\widehat{\mathbf{x}}) \equiv B_{i_1 i_2 i_3}(\widehat{\mathbf{x}}) = B_{i_1,p_1}(\widehat{x}) B_{i_2,p_2}(\widehat{y}) B_{i_3,p_3}(\widehat{z}),$$

where $\mathbf{i} \equiv (i_1, i_2, i_3)$ is a multi-index that belongs to the set

$$\mathcal{J} := \{\mathbf{j} = (j_1, j_2, j_3) : 1 \leq j_d \leq n_d,\ d = 1, 2, 3\}.$$

Notice that the knot vectors Ξ_d define a Cartesian partition $\mathcal{Q}_h$ of the unit cube $\widehat{\Omega} = (0,1)^3$, and the multiplicity of the knots defines the regularity across the knot spans. This space of B-Splines will be denoted by $S^{p_1,p_2,p_3}_{\boldsymbol{\alpha}_1,\boldsymbol{\alpha}_2,\boldsymbol{\alpha}_3}(\mathcal{Q}_h)$, or simply $S^{p_1,p_2,p_3}_{\boldsymbol{\alpha}_1,\boldsymbol{\alpha}_2,\boldsymbol{\alpha}_3}$. We refer the reader to [17, 18] for exhaustive studies on B-Splines and their approximation properties.

NURBS basis functions and geometric entities are then immediately obtained from the previous B-Spline spaces. In brief, a positive weight $w_{\mathbf{i}}$ can be associated to each B-Spline basis function $B_{\mathbf{i}}$, and the corresponding NURBS basis function is defined as

$$N_{\mathbf{i}}(\widehat{\mathbf{x}}) = \frac{w_{\mathbf{i}} B_{\mathbf{i}}(\widehat{\mathbf{x}})}{w}, \quad \text{with} \quad w = \sum_{\mathbf{j} \in \mathcal{J}} w_{\mathbf{j}} B_{\mathbf{j}},$$

both $\mathbf{i}$ and $\mathbf{j}$ being multi-indices. The space of NURBS is denoted by $N^{p_1,p_2,p_3}(\mathcal{Q}_h; w)$, or just N^{p_1,p_2,p_3} for simplicity. Notice that this space depends on the weight function w, and,

when all the weights $w_{\mathbf{j}}$ are equal, it simply reduces to a B-Spline space, due to the partition of unity property.

In order to describe the domain geometry, a control point $\mathbf{C_i} \in \mathbb{R}^3$ is then associated to each NURBS (or B-Spline) basis function and the domain is defined by the parameterization

$$\begin{aligned} \mathbf{F} : \widehat{\Omega} &\longrightarrow \Omega \\ \widehat{\mathbf{x}} &\longmapsto \mathbf{x} = \mathbf{F}(\widehat{\mathbf{x}}) := \sum_{\mathbf{j}\in\mathcal{J}} N_{\mathbf{j}}(\widehat{\mathbf{x}})\mathbf{C_j}. \end{aligned} \tag{8}$$

Finally, B-Splines and NURBS allow to easily reproduce both FEM typical refinement strategy, namely, h- and p-refinements, by means of knot insertion and degree elevation procedures, respectively. Moreover, a third particularly effective option is offered by the so-called k-refinement, and consists of an high-regularity refinement strategy (see [1, 21]).

Further details on NURBS and on their use in CAD can be found in [19], along with several algorithms to handle basis functions and geometric entities. We also refer the reader to [1, 12] for more details and examples on the subject.

3. A model problem: Poisson

We now specialize the general framework of Section 2.1 to the particular case of the Poisson's problem, defined in a physical domain described with NURBS and discretized either with NURBS or B-Splines. This constitutes the model problem on which we show in detail the basic features and possibilities of the code.

Let us assume that the computational domain is constructed as a single NURBS patch, such that the parameterization $\mathbf{F}$ is given by (8). The assumptions on $\mathbf{F}$ of Section 2.1 are here supposed to be valid. A Poisson's problem with mixed boundary conditions is then considered. Therefore, the boundary $\partial\Omega$ is split into two disjoint parts, $\partial\Omega = \Gamma_N \cup \Gamma_D$ (with $\Gamma_N \cap \Gamma_D = \emptyset$, and $\Gamma_D \neq \emptyset$), and the equations of the source problem read

$$\begin{cases} -\operatorname{div}\left(k(\mathbf{x})\,\mathbf{grad}\,u\right) = f & \text{in } \Omega, \\ k(\mathbf{x})\dfrac{\partial u}{\partial \mathbf{n}} = g & \text{on } \Gamma_N, \\ u = 0 & \text{on } \Gamma_D, \end{cases} \tag{9}$$

where $\mathbf{n}$ is the unit normal vector exterior to Ω, and, for simplicity, $f \in L^2(\Omega)$ and $g \in L^2(\Gamma_N)$. Again for the sake of simplicity, homogeneous Dirichlet boundary conditions are assumed.

The eigenvalue problem consists instead of finding $u \neq 0$ and λ such that

$$\begin{cases} -\operatorname{div}\left(k(\mathbf{x})\,\mathbf{grad}\,u\right) = \lambda \epsilon(\mathbf{x}) u & \text{in } \Omega, \\ k(\mathbf{x}) \dfrac{\partial u}{\partial \mathbf{n}} = 0 & \text{on } \Gamma_N, \\ u = 0 & \text{on } \Gamma_D. \end{cases}$$

These problems in their variational formulation, as in (2) and (3), read as:

Find $u \in H^1_{0,\Gamma_D}(\Omega)$ *such that*

$$\int_\Omega k(\mathbf{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,d\mathbf{x} = \int_\Omega f\,v\,d\mathbf{x} + \int_{\Gamma_N} g\,v\,d\Gamma \quad \forall v \in H^1_{0,\Gamma_D}(\Omega),$$

and:

Find $\lambda \in \mathbb{R} \setminus \{0\}$*, and* $u \in H^1_{0,\Gamma_D}(\Omega)$*,* $u \neq 0$*, such that*

$$\int_\Omega k(\mathbf{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,d\mathbf{x} = \lambda \int_\Omega \epsilon(\mathbf{x})\,u\,v\,d\mathbf{x} \quad \forall v \in H^1_{0,\Gamma_D}(\Omega),$$

with $H^1_{0,\Gamma_D} := \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\}$ the space of functions with vanishing trace on Γ_D. The variational formulations of the discrete problems are:

Find $u_h \in V_h$ *such that*

$$\int_\Omega k(\mathbf{x})\,\mathbf{grad}\,u_h \cdot \mathbf{grad}\,v_h\,d\mathbf{x} = \int_\Omega f\,v_h\,d\mathbf{x} + \int_{\Gamma_N} g\,v_h\,d\Gamma \quad \forall v_h \in V_h, \tag{10}$$

and:

Find $\lambda \in \mathbb{R} \setminus \{0\}$*, and* $u_h \in V_h$*,* $u_h \neq 0$ *such that*

$$\int_\Omega k(\mathbf{x})\,\mathbf{grad}\,u_h \cdot \mathbf{grad}\,v_h\,d\mathbf{x} = \int_\Omega \epsilon(\mathbf{x})\,u_h\,v_h\,d\mathbf{x} \quad \forall v_h \in V_h, \tag{11}$$

where the discrete space V_h is defined as

$$V_h = \{v_h \in H^1_{0,\Gamma_D} : v_h = \widehat{v}_h \circ \mathbf{F}^{-1},\, \widehat{v}_h \in \widehat{V}_h\},$$

and $\widehat{V}_h$ is the discrete space in the parametric domain, that has to be chosen.

Let us denote by

$$N_h = \dim(\widehat{V}_h) = \dim(V_h),$$

the dimension of our finite dimensional spaces, and let

$$\{\widehat{v}_i\}_{i=1}^{N_h},$$

be a basis for $\widehat{V}_h$. Then, due to the assumptions on the parameterization $\mathbf{F}$, we can define a basis for V_h as follows

$$\left\{v_i = \widehat{v}_i \circ \mathbf{F}^{-1}\right\}_{i=1}^{N_h}. \tag{12}$$

Having introduced these bases, we can rewrite equations (10) and (11), where the trial functions can now be expressed as

$$u_h = \sum_{j=1}^{N_h} \alpha_j v_j = \sum_{j=1}^{N_h} \alpha_j (\widehat{v}_j \circ \mathbf{F}^{-1}) \tag{13}$$

and their gradients as

$$\mathbf{grad}\, u_h = \sum_{j=1}^{N_h} \alpha_j \, \mathbf{grad}\, v_j = \sum_{j=1}^{N_h} \alpha_j (D\mathbf{F})^{-T} (\, \mathbf{grad}\, \widehat{v}_j \circ \mathbf{F}^{-1}), \tag{14}$$

where $D\mathbf{F}$ is the Jacobian matrix of the parameterization $\mathbf{F}$, and $(D\mathbf{F})^{-T}$ denotes its inverse transposed. It is sufficient that the equations are verified for any test function of the basis (12), which yields the source problem

$$\begin{gathered}\sum_{j=1}^{N_h} A_{ij}\, \alpha_j = \int_\Omega k(\mathbf{x}) \sum_{j=1}^{N_h} \alpha_j \, \mathbf{grad}\, v_j \cdot \, \mathbf{grad}\, v_i \, d\mathbf{x} = \\ \int_\Omega f\, v_i \, d\mathbf{x} + \int_{\Gamma_N} g\, v_i \, d\Gamma = f_i + g_i, \qquad \text{for } i = 1, \ldots, N_h,\end{gathered} \tag{15}$$

and the eigenvalue problem

$$\sum_{j=1}^{N_h} A_{ij}\, \alpha_j = \lambda \int_\Omega \epsilon(\mathbf{x}) \sum_{j=1}^{N_h} \alpha_j v_j v_i \, d\mathbf{x} = \lambda \sum_{j=1}^{N_h} M_{ij}\, \alpha_j, \qquad \text{for } i = 1, \ldots, N_h, \tag{16}$$

where A_{ij} and M_{ij} are the coefficients of the stiffness and mass matrices, and f_i and g_i are the coefficients of the right-hand side contributions from the source and the boundary terms, respectively.

All these coefficients are given by the values of the integrals in (15) and (16), that are numerically approximated by a suitable quadrature rule. In order to describe this rule, let us introduce $\widehat{\mathcal{K}}_h := \{\widehat{K}_k\}_{k=1}^{N_e}$, that is a partition of the parametric domain $\widehat{\Omega}$ into N_e non-overlapping subregions, that henceforth we refer to as *elements*. The assumptions on the parameterization $\mathbf{F}$ ensure that the physical domain Ω can be partitioned as

$$\overline{\Omega} = \bigcup_{k=1}^{N_e} \mathbf{F}(\widehat{K}_k),$$

and the corresponding *elements* $K_k := \mathbf{F}(\widehat{K}_k)$ are also non-overlapping. We denote this partition by $\mathcal{K}_h := \{K_k\}_{k=1}^{N_e}$.

For the sake of generality, let us assume that a quadrature rule is defined on every element $\widehat{K}_k$. Each of these quadrature rules is determined by a set of n_k *nodes*

$$\{\widehat{\mathbf{x}}_{l,k}\} \subset \widehat{K}_k, \; l = 1, \ldots, n_k$$

and by their corresponding *weights*

$$\{w_{l,k}\} \subset \mathbb{R},\ l = 1, \ldots, n_k.$$

After introducing a change of variables, the integral of a generic function $\phi \in \mathcal{L}^1(K_k)$ can be approximated as follows

$$\int_{K_k} \phi \, d\mathbf{x} = \int_{\widehat{K}_k} \phi(\mathbf{F}(\widehat{\mathbf{x}})) \, |\det(D\mathbf{F}(\widehat{\mathbf{x}}))| \, d\widehat{\mathbf{x}} \simeq \sum_{l=1}^{n_k} w_{l,k} \ \phi(\mathbf{x}_{l,k}) \ \ |\det(D\mathbf{F}(\widehat{\mathbf{x}}_{l,k}))| \, ,$$

where $\mathbf{x}_{l,k} := \mathbf{F}(\widehat{\mathbf{x}}_{l,k})$ are the images of the quadrature nodes in the physical domain.

Using the quadrature rule, the coefficients A_{ij} of the stiffness matrix are numerically computed as

$$A_{ij} \simeq \sum_{k=1}^{N_e} \sum_{l=1}^{n_k} k(\mathbf{x}_{l,k}) \ w_{l,k} \ \ \mathbf{grad}\, v_j(\mathbf{x}_{l,k}) \cdot \mathbf{grad}\, v_i(\mathbf{x}_{l,k}) \ \ |\det(D\mathbf{F}(\widehat{\mathbf{x}}_{l,k}))| \, , \tag{17}$$

while the coefficients f_i of the right-hand side vector are approximated as

$$f_i \simeq \sum_{k=1}^{N_e} \sum_{l=1}^{n_k} f(\mathbf{x}_{l,k}) \ w_{l,k} \ v_i(\mathbf{x}_{l,k}) \ \ |\det(D\mathbf{F}(\widehat{\mathbf{x}}_{l,k}))| \, . \tag{18}$$

Finally, for the eigenvalue problem the coefficients M_{ij} of the mass matrix are numerically computed as

$$M_{ij} \simeq \sum_{k=1}^{N_e} \sum_{l=1}^{n_k} \epsilon(\mathbf{x}_{l,k}) \ w_{l,k} \ v_j(\mathbf{x}_{l,k}) \, v_i(\mathbf{x}_{l,k}) \ \ |\det(D\mathbf{F}(\widehat{\mathbf{x}}_{l,k}))| \, . \tag{19}$$

In order to treat the boundary terms, let us first define the mapping $\mathbf{F}_b : (0,1) \longrightarrow \Gamma_N$ for 2D geometries, and $\mathbf{F}_b : (0,1)^2 \longrightarrow \Gamma_N$ for 3D geometries. Notice that for B-Splines and NURBS, assuming that each side of the parametric domain is completely mapped into Γ_N or Γ_D, this mapping could be taken (roughly speaking) as the restriction of $\mathbf{F}$ to the boundary.

To numerically compute the boundary term in (15) a quadrature rule is defined on the boundaries (mostly, inherited from the one defined on the whole domain). We denote the quadrature nodes in the reference interval (or square) by their parametric coordinates $t_{l,k}$ (or $(s_{l,k}, t_{l,k})$), and for the rest we use the same notation as before, with a superscript b. The boundary line integrals, for 2D geometries, are then approximated as follows

$$\int_{K_k^b} \phi \, d\Gamma = \int_{\widehat{K}_k^b} \phi(\mathbf{F}_b(t)) \, |\mathbf{F}_b'(t)| \ dt \simeq \sum_{l=1}^{n_k^b} w_{l,k}^b \ \phi(\mathbf{x}_{l,k}^b) \ \ |\mathbf{F}_b'(t_{l,k})| \, ,$$

which gives the Neumann term in (15)

$$g_i \simeq \sum_{k=1}^{N_e} \sum_{l=1}^{n_k} g(\mathbf{x}_{l,k}^b) \ w_{l,k}^b \ v_i(\mathbf{x}_{l,k}^b) \ \ |\mathbf{F}_b'(t_{l,k})| \, . \tag{20}$$

In the 3D case, the surface integrals of the boundary terms are computed as

$$\int_{K_k^b} \phi \, d\Gamma = \int_{\widehat{K}_k^b} \phi(\mathbf{F}_b(s,t)) \left| \frac{\partial \mathbf{F}_b}{\partial s}(s,t) \times \frac{\partial \mathbf{F}_b}{\partial t}(s,t) \right| ds \, dt$$
$$\simeq \sum_{l=1}^{n_k^b} w_{l,k}^b \, \phi(\mathbf{x}_{l,k}^b) \, \left| \frac{\partial \mathbf{F}_b}{\partial s}(s_{l,k}, t_{l,k}) \times \frac{\partial \mathbf{F}_b}{\partial t}(s_{l,k}, t_{l,k}) \right|,$$

which yields the coefficients for the Neumann term in (15)

$$g_i \simeq \sum_{k=1}^{N_e} \sum_{l=1}^{n_k} g(\mathbf{x}_{l,k}^b) \, w_{l,k}^b \, v_i(\mathbf{x}_{l,k}^b) \, \left| \frac{\partial \mathbf{F}_b}{\partial s}(s_{l,k}, t_{l,k}) \times \frac{\partial \mathbf{F}_b}{\partial t}(s_{l,k}, t_{l,k}) \right|. \tag{21}$$

Remark 3.1. *We notice that, when discretizing with NURBS and using, e.g., standard Gauss quadrature rules, the partition $\widehat{\mathcal{K}}_h$ coincides with the partition $\mathcal{Q}_h$ defined in Section 2.2. However, such partitions may be different, for instance when the quadrature rules of [22] are used.*

4. The design of `GeoPDEs`

Remark 4.1. *The examples in the following sections are written for* `GeoPDEs` *1.1.0, or earlier versions. The examples are rewritten for* `GeoPDEs` *2.0.0 (or later) in Appendix C.*

As we explained in the introduction, `GeoPDEs` is intended to serve as a rapid prototyping tool for the implementation of new IGA methods and ideas, as well as to introduce other researchers to IGA coding. We remark once again that the efficiency has been sacrificed in order to implement a code that is general and easy to understand and to modify.

The implementation follows in some sense the ideas of [13]: i.e., the computations for the geometry, the discrete basis functions and the matrices for the analysis are done separately. Moreover, all data needed for these computations is stored in independent structures, in such a way that each part of the code can be modified without affecting the others.

In this section we explain the basic data structures of `GeoPDEs` and the main functions to operate on them, making use of a very simple example and of the notation introduced in Section 3. In the next section we then discuss specializations and extensions required to adapt `GeoPDEs` to other applications and we present some more complex examples.

4.1. A very simple example

Let the computational domain $\Omega \subset \mathbb{R}^2$ be the intersection of the first quadrant of the Cartesian plane with a circular annulus of internal radius $r = 1$ and external radius $R = 2$ (see Fig. 1). This geometry can be defined as a NURBS surface whose knot vectors, control

points and weights are listed in Appendix A. Let the diffusion coefficient in (9) be $k(\mathbf{x}) = 1$, and let the source term be

$$f = \frac{(8 - 9\sqrt{x^2 + y^2})\sin(2\arctan(y/x))}{x^2 + y^2}. \tag{22}$$

Furthermore, homogeneous Dirichlet boundary conditions are imposed on the whole boundary, i.e., $\Gamma_D \equiv \partial\Omega$. In this simple case, the exact solution is given by

$$u = (x^2 + y^2 - 3\sqrt{x^2 + y^2} + 2)\sin(2\arctan(y/x)).$$

The full code for solving (9) with the data described above is displayed in Listing 1, while in the subsections below we describe the purpose of each block of lines.

```
1  geometry = geo_load ('ring_refined.mat');
2  knots = geometry.nurbs.knots;
3  [qn, qw] = msh_set_quad_nodes (knots, msh_gauss_nodes (geometry.nurbs.order));
4  msh = msh_2d_tensor_product (knots, qn, qw);
5  msh = msh_push_forward_2d (msh, geometry);
6  space  = sp_nurbs_2d_phys (geometry.nurbs, msh);
7  [x, y] = deal (squeeze (msh.geo_map(1,:,:)), squeeze (msh.geo_map(2,:,:)));
8  mat = op_gradu_gradv (space, space, msh, ones (size (x)));
9  rhs = op_f_v (space, msh, ...
       (8-9*sqrt(x.^2+y.^2)).*sin(2*atan(y./x))./(x.^2+y.^2));
10 drchlt_dofs = unique ([space.boundary(:).dofs]);
11 int_dofs = setdiff (1:space.ndof, drchlt_dofs);
12 u = zeros (space.ndof, 1);
13 u(int_dofs) = mat(int_dofs, int_dofs) \ rhs(int_dofs);
14 sp_to_vtk_2d (u, space, geometry, [20 20], 'laplace_solution.vts', 'u')
15 err = sp_l2_error (space, msh, u, ...
       @(x,y)((x.^2+y.^2)-3*sqrt(x.^2+y.^2)+2).*sin(2.*atan (y./x)))
```

Listing 1: Solving the model problem with `GeoPDEs`.

4.2. Definition of the parameterization: the geometry structure

The first step to set-up the problem to be solved is to define the geometry of the physical domain, for which we assume a parameterization as (1). In our simple example, this is done by invoking the geo_load function:

```
1  geometry = geo_load ('ring_refined.mat');
```

The function geo_load takes as input the name of a file in MATLAB binary format, which contains a structure defined by the NURBS toolbox (see Appendix A). The output is the structure **geometry**, which contains the information to compute the geometry parameterization and its derivatives. The fields of this structure are:

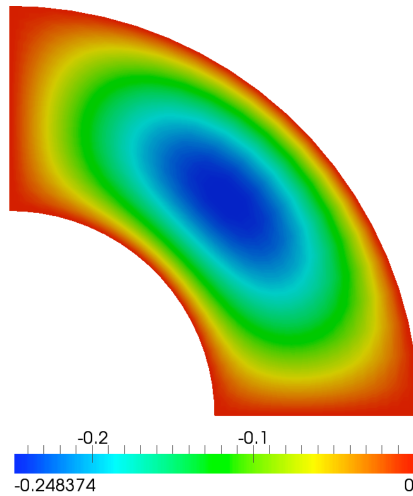


Figure 1: Solution of the model problem.

- **map**: function handle to compute the parameterization $\mathbf{F}$ at some given points in $\widehat{\Omega}$.
- **map_der**: function handle to compute the Jacobian of the parameterization $D\mathbf{F}$.

Note that the structure contains the function handles to compute $\mathbf{F}$ or $D\mathbf{F}$, not the values of the map. The structure may also contain some other fields, as a function handle to compute the second derivatives of $\mathbf{F}$, or other auxiliary fields. In our example the two fields are handles to functions of the NURBS toolbox, and the **nurbs** structure of the toolbox is saved, as well, as an auxiliary field. Different ways to invoke the function geo_load will be shown later.

4.3. Quadrature: the msh structure

The second step is to define the domain partition and to set the quadrature rule in each element, in order to compute the matrices and the right-hand side vector of the problem by numerical integration, as in (17)-(19).

The simplest possibility is to define a tensor product partition where the quadrature elements coincide with the knot-spans in the geometry. To this end we get the two knot vectors from the **geometry** structure, and compute through the function msh_gauss_nodes the quadrature nodes and weights for a standard Gaussian quadrature rule, using a number of quadrature nodes in each direction equal to the degree of the NURBS plus one:

```
2 knots = geometry.nurbs.knots;
3 [qn, qw] = msh_set_quad_nodes (knots, msh_gauss_nodes (geometry.nurbs.order));
```

The information for the quadrature rule is then stored in the structure **msh**, which is first computed in the parametric domain by tensor products, and then mapped to the physical domain using the handles of the structure **geometry**:

```
4 msh = msh_2d_tensor_product (knots, qn, qw);
5 msh = msh_push_forward_2d (msh, geometry);
```

The output is the **msh** structure, which must contain the following fields:

- **nel**: N_e, the number of elements of the partition $\mathcal{K}_h$.
- **nqn**: n_k, the number of quadrature points per element.
- **quad_nodes**: coordinates of the quadrature nodes $\widehat{\mathbf{x}}_{l,k}$ in the parametric domain $\widehat{\Omega}$.
- **quad_weights**: weights $w_{l,k}$ associated to the nodes.
- **geo_map**: $\mathbf{x}_{l,k} = \mathbf{F}(\widehat{\mathbf{x}}_{l,k})$, coordinates of the quadrature nodes in the physical domain.
- **geo_map_jac**: Jacobian matrix of the parameterization $\mathbf{F}$ evaluated at the quadrature points, i.e., $D\mathbf{F}(\widehat{\mathbf{x}}_{l,k})$.
- **jacdet**: absolute value of the Jacobian matrix determinant, evaluated at the quadrature points, i.e., $|\det(D\mathbf{F}(\widehat{\mathbf{x}}_{l,k}))|$.

The values of the last three fields are computed using the handles of the **geometry** structure. We note that these three fields are already present in the **msh** structure computed in line 4 of the example, but there they refer to the identity parameterization.

4.4. The discrete space: the space structure

The most important structure of our implementation is the one referred to as **space**. This contains the information regarding the basis functions of the discrete space V_h, and their evaluation at the quadrature nodes in order to numerically compute the integrals of the problem.

In our example we are invoking the isoparametric paradigm, which means that the space of the geometry and the discrete space of the shape functions coincide. Hence, the information for the discrete space is already contained in the **nurbs** structure of the NURBS toolbox, that we stored as a field in **geometry**. The new structure is computed using this information and the **msh** structure with the command:

```
6 space = sp_nurbs_2d_phys (geometry.nurbs, msh);
```

As in FEM, the basis functions in IGA are locally supported, thus the integrals on each element of the partition are only computed for a reduced number of basis functions. Accordingly, only the values of some basis functions are stored on each element. Analogously to what is done in FEM, a global numbering of the basis functions is introduced. Then, for each element of the quadrature partition we give its connectivity, that is, the number associated to the basis functions whose support intersects the element. Therefore, the information contained in the structure is the following:

- **ndof**: N_h, total number of degrees of freedom, which is equal to the dimension of the space V_h.
- **nsh**: N_s, number of non-vanishing basis functions in each element.
- **connectivity**: numbers associated to the basis functions that do not vanish on each element. It has size $N_s \cdot N_e$, where N_e is the number of elements.
- **shape_functions**: evaluation of the basis functions at the quadrature points, that is, the quantities $v_i(\mathbf{x}_{l,k})$ in equation (19). In our model problem, its size is $n_k \cdot N_s \cdot N_e$, where n_k is the number of quadrature points.
- **spfun**: function handle to evaluate the fields above at the points given in a **msh** structure, this is used when evaluating at points different from the quadrature points is required, e.g. for visualization.

Some optional fields may also appear, depending on the problem to be solved. For instance, in our model problem we also need the field

- **shape_function_gradients**: gradient of the basis functions evaluated at the quadrature points, that is, $\mathbf{grad}\, v_i(\mathbf{x}_{l,k})$. In our model problem it has size $d \cdot n_k \cdot N_s \cdot N_e$, where d is the dimension of the problem.

In the example, the values of the shape functions and their gradients are first computed in the parametric domain, making use of some functions of the NURBS toolbox. The values of the gradients in the physical domain are then computed by applying the push-forward, using the information contained in **msh**.

We will see later other examples where fields for the divergence or the curl are also computed. Moreover, we notice that the structure contains another field called **boundary**, that will be explained in Section 4.6.

4.5. Matrix and vector construction

Once the three basic data structures have been initialized, the next step is to assemble the stiffness matrix and the right-hand side of the linear system. GeoPDEs provides several functions that allow to compute the matrices and right-hand sides for different PDE problems, starting from the **msh** and **space** structures introduced above. These functions all have a very similar structure, which may be adapted with very little changes to handle different differential problems.

Recalling that in our problem $k(\mathbf{x}) = 1$, and the source function f is given by (22), the stiffness matrix and the right-hand side are computed with the commands

```
7 [x, y] = deal (squeeze (msh.geo_map(1,:,:)), squeeze (msh.geo_map(2,:,:)));
8 mat = op_gradu_gradv (space, space, msh, ones (size (x)));
9 rhs = op_f_v (space, msh, ...
      (8-9*sqrt(x.^2+y.^2)).*sin(2*atan(y./x))./(x.^2+y.^2));
```

where the last input arguments in lines 8 and 9 are the coefficients $k(\mathbf{x})$ and $f(\mathbf{x})$, respectively, evaluated at the quadrature points. In line 8 the **space** structure is passed twice, because the function is prepared to use different spaces for trial and test functions.

Remark 4.2. *In line 7, two rather advanced commands of the Matlab programming language appear: the function* deal *is used to compactly do multiple assignments in one single line, and the function* squeeze *to remove the singleton dimensions of multi-dimensional arrays. Although we have tried to keep the programming style of* GeoPDEs *as simple as possible, the extensive use of multi-dimensional arrays and tensor-product meshes and spaces has made it necessary to resort to other advanced constructs (e.g.,* repmat *or* **reshape***). However, the detailed description of such commands is beyond the scope of the present paper and we refer the interested reader to the Matlab or Octave manuals.*

The separation of the matrix assembly stage from that of basis function definition and evaluation is one of the main features of our code, which directly descends from its design objectives. In choosing to evaluate and store all the values of the basis functions at time of initialization of the **space** structure we have clearly favored speed in the trade-off with memory consumption. Although this choice limits the maximum size of problems that GeoPDEs can handle, the actual limit being dependent on available hardware, this bottleneck may be overcome, e.g., by changing the fields of **space** structure to be function handles rather than arrays. Anyway, since the handling of problems of such size is beyond the scope of this presentation we do not further pursue this issue here.

To show how our approach simplifies the implementation, we report below (Listing 2) a simple OCTAVE function to compute the mass matrix in (16), using equation (19). It takes

as arguments the **space** and **msh** structures, and the coefficient ϵ already evaluated at the quadrature points. The function basically consists of a cycle over the elements, two cycles over the basis functions, and a final cycle over the quadrature points of each element.

Remark 4.3. *The function of Listing 2 slightly differs from the one actually present in the package. The latter, in fact, allows the use of different spaces for trial and test functions, and is valid for both scalar and vectorial problems. Other minor changes were done to make the function faster in* MATLAB.

```
1  function mat = op_u_v (space, msh, epsilon)
2    mat = spalloc (space.ndof,space.ndof,1);
3    for iel = 1:msh.nel
4      mat_loc = zeros (space.nsh(iel), space.nsh(iel));
5      for idof = 1:space.nsh(iel)
6        ishp = squeeze (space.shape_functions(:,idof,iel));
7        for jdof = 1:space.nsh(iel)
8          jshp = squeeze (space.shape_functions(:,jdof,iel));
9          for inode = 1:msh.nqn
10           mat_loc(idof,jdof) = mat_loc(idof,jdof) + ...
11             msh.jacdet(inode,iel) * msh.quad_weights(inode,iel) * ...
12             ishp(inode) .* jshp(inode) * epsilon(inode,iel);
13         end
14       end
15     end
16     mat(space.connectivity(:,iel), space.connectivity(:,iel)) = ...
17     mat(space.connectivity(:,iel), space.connectivity(:,iel)) + mat_loc;
18   end
19 end
```

Listing 2: MATLAB function to compute the mass matrix.

4.6. The treatment of boundary conditions: the boundary substructures

The imposition of homogeneous boundary conditions in IGA is straightforward. In fact, homogeneous Neumann conditions are automatically imposed without any change in the arrays (as in FEM). For homogeneous Dirichlet conditions, as in our example, we first need to identify the boundary degrees of freedom corresponding to functions that do not vanish on the boundary, and separate them from the internal degrees of freedom. This is done in the code with the commands

```
10 drchlt_dofs = unique ([space.boundary(:).dofs]);
11 int_dofs = setdiff (1:space.ndof, drchlt_dofs);
```

Then, the coefficients α_j in (14) for the internal degrees of freedom are computed as the solution of the linear system, whereas for the boundary ones they are set to zero:

```
12 u = zeros (space.ndof, 1);
13 u(int_dofs) = mat(int_dofs , int_dofs) \ rhs(int_dofs);
```

For the implementation of boundary conditions, both the **msh** and the **space** structures are enriched with a field called **boundary**. In order to define these fields we first divide the boundary of the parametric domain $\widehat{\Omega}$ into a certain number of sides. Then, the boundary field is defined as an array that contains, for each side, a **msh** or a **space** structure, similar to the ones previously defined: **msh.boundary** contains a partition of each boundary side in order to perform numerical integration (mostly inherited from the one defined on the whole domain), while **space.boundary** contains information about the boundary basis functions, and their values at the quadrature points given by **msh.boundary**.

There are, however, small differences with respect to the original structures we introduced before. In the **msh.boundary** structure, the field **jacdet** does not longer contain the determinant of the Jacobian. Instead, it contains the norm of the differential of the boundary parameterization $\mathbf{F}_b$, which is the term $|\mathbf{F}_b'(t_{l,k})|$ in (20) or $\left|\frac{\partial \mathbf{F}_b}{\partial s}(s_{l,k}, t_{l,k}) \times \frac{\partial \mathbf{F}_b}{\partial t}(s_{l,k}, t_{l,k})\right|$ in (21). The structure also contains the field **normal**, with the value of the unit normal exterior vector at the boundary quadrature points.

In **space.boundary** the **ndof** and **connectivity** fields refer to a local numbering of the basis functions actually supported on each boundary side. Therefore, it is also necessary to include the field **dofs**, which relates this local numbering to the global numbering in the whole domain. This field is in fact the one we have used in the example to determine which degrees of freedom had to be set to zero.

Obviously, an example with homogeneous boundary conditions is not the best suited to explain the **boundary** substructures, but its use will be made clearer in Section 5.1.4, where we solve a model problem with non-homogeneous boundary conditions.

4.7. Postprocessing: visualization and computation of the error

Once the linear system has been solved, the last step is to visualize the computed solution. As IGA is seen as a tool to improve the communication between CAD software and PDE discrete solvers, the same holds for the communication between the solvers and the visualization software. We are far from being experts on the matter, so our immediate goal is not to implement new techniques to do it. Instead, we have decided to use an already existing software (namely, `ParaView` [23]) and prepare a function to visualize our solution data with it.

The following command evaluates the solution of the problem at the points given by a

20×20 grid, uniform in the parametric domain, and saves the results in a `vtk` structured data file format, that can be visualized with `ParaView`.

```
14 sp_to_vtk_2d (u, space, geometry, [20 20], 'laplace_solution.vts', 'u')
```

The resulting plot is reported in Fig. 1. Other examples provided with the package also show how to plot the solution in OCTAVE or MATLAB using sp_eval_2d.

This handle allows to evaluate the shape functions of the discrete space, and also their gradients, at a set of given points (not necessarily being the quadrature points). The same function is used whenever the solution must be computed at a set of points. For instance, in academical cases in which we know the exact solution, the L^2–norm of the error is computed as

```
15 err = sp_l2_error (space, msh, u, ...
     @(x,y)((x.^2+y.^2)-3*sqrt(x.^2+y.^2)+2).*sin(2.*atan (y./x)))
```

where the last argument is a function handle to compute the exact solution. The function sp_l2_error also makes use of the handle **spfun** mentioned above.

5. Applying GeoPDEs to more complex problems

The sequence of steps of the previous section represents the basic structure of a script to solve a PDE problem with `GeoPDEs`. In the first part of this section we show possible modifications of this structure, to tackle more complex and interesting examples. In the second part of the section we show the main modifications that have to be done in order to solve problems in linear elasticity, fluid mechanics and electromagnetism.

5.1. Modifications of the model problem

We start by introducing minor modifications to Listing 1, in order to solve the same problem with different approaches. This will allow us to show how the code can be easily modified.

5.1.1. Introducing h-, p- and k-refinement

In the simple example of the previous section we solved the problem by loading a geometry from the NURBS toolbox, and then performed the discretization of the problem exactly in the same space in which the geometry was defined. But for real problems it is necessary to introduce a refined discrete space in order to get accurate numerical solutions. One of the advantages of IGA with respect to FEM is that the refinement can be done without affecting the geometry, and for NURBS and B-Splines it is implemented in an easy manner. In this

section we explain how the h-, p- and k-refinements are treated in GeoPDEs. In all the three cases we make use of functions contained in the NURBS toolbox (see Appendix A). We refer the reader to the "help" of these functions for more details, and to [19] for the definition of the concepts of degree elevation and knot insertion that are used in what follows.

The case of p-refinement is the easiest to explain. In this case the refinement consists on applying degree elevation, which is done by invoking the nrbdegelev function of the toolbox. For instance, in order to solve with NURBS of degree 5, the following commands should be added between lines 1 and 2 of Listing 1:

```
1 nurbs    = geometry.nurbs;
2 degelev  = max ([5 5] - (nurbs.order-1), 0);
3 nurbs    = nrbdegelev (nurbs, degelev);
4 geometry = geo_load (nurbs);
```

The purpose of the second line is to avoid executing degree elevation when the desired degree is lower than the actual one of the geometry, a useful check when automatic refinement procedures are implemented.

h-refinement is instead obtained by knot insertion, which is easily computed using the function nrbkntins of the toolbox. The important thing is to determine which knots one would like to insert. A simple and classical strategy is to add new knots uniformly, for which the function kntrefine of the toolbox can be helpful. For instance, substituting the lines 2 and 3 of the p-refinement example by:

```
1 [rknots, zeta, nknots] = kntrefine (nurbs.knots, [2 2], nurbs.order-1, [0 0]);
2 nurbs = nrbkntins (nurbs, nknots);
```

would insert two new knots in each subinterval of the original knot vectors. The last argument is used to specify the desired continuity, so in this case the new knots are added with the right multiplicity to get a discrete space of C^0 continuity.

Finally, k-refinement consists of performing first a degree elevation, and then a knot insertion with the lowest multiplicity for the new knots, in order to have higher regularity of the basis functions. The implementation is an easy combination of what is done for p- and h-refinement. In the following example the problem is be solved with NURBS of degree 3, and each interval of the original knot vector is refined by inserting one new knot without repetitions, which yields C^2 continuity at these knots:

```
1 nurbs    = geometry.nurbs;
2 degelev  = max ([3 3] - (nurbs.order-1), 0);
3 nurbs    = nrbdegelev (nurbs, degelev);
4 [rknots, zeta, nknots] = kntrefine (nurbs.knots, [1 1], nurbs.order-1, ...
5                                     nurbs.order-2);
6 nurbs = nrbkntins (nurbs, nknots);
```

```
7 geometry = geo_load (nurbs);
```

Notice that in these examples the function geo_load is invoked with a NURBS structure, rather than with a binary file. In the next section we will give more details about this function.

5.1.2. Implementation of the non-isoparametric approach

One of the main features of `GeoPDEs` is the fact that the geometry and the discrete space are treated independently. This allows to solve problems using non-isoparametric approaches, where the solution space does not coincide with the geometry space.

The computation with B-Spline spaces is easily implemented, and very similar to NURBS. In order to solve with a NURBS geometry but with a spline discretization, the example given in Listing 1 is modified by substituting line 6 with

```
space = sp_bspline_2d_phys (knots, [4 4], msh);
```

where 4 is the degree of the B-Spline space associated to the knot vectors. In fact, the knot vectors for the geometry and for the discrete space are not necessarily the same, which means that geometry refinement can be avoided. We however remark that the continuity of the B-Spline space must be related to the continuity of the geometry in order to obtain optimal convergence rates (see [10, 24]).

Since the geometry and the discrete space are unrelated, different ways of representing the geometry can be considered, as far as the geometry is given by a parameterization in the form of (1), and a **geometry** structure as that described in Section 4 can be defined. The function geo_load is prepared to create the structure for geometries defined in the following ways:

- As a structure of the NURBS toolbox, either from a file or from a variable.
- As an affine transformation, defined by a 4x4 matrix.
- As a function handle explicitly defined by the user.

In this latter case the user must provide the Matlab functions to compute the parameterization and its derivatives. Let us show this with an example. We consider the same problem as in Section 4, but with the domain defined by $\mathbf{F}(u, v) = ((u + 1)\cos(\pi v/2), (u + 1)\sin(\pi v/2))$, with $0 < u, v < 1$. Accordingly, the parameterization is defined through the following Matlab function:

```
function F = ring_polar_map (pts)
  u = pts(1,:); v = pts(2,:);
```

```
  F(1,:) = (u+1) .* cos(pi*v/2);
  F(2,:) = (u+1) .* sin(pi*v/2);
end
```

and its Jacobian matrix by:

```
function jac = ring_polar_map_der (pts)
  u = pts(1,:); v = pts(2,:);
  jac = zeros (2, 2, numel (u));
  jac(1,1,:) =  cos(pi*v/2);
  jac(2,1,:) =  sin(pi*v/2);
  jac(1,2,:) = -pi*(u+1) .* sin(pi*v/2)/2;
  jac(2,2,:) =  pi*(u+1) .* cos(pi*v/2)/2;
end
```

Then, the geometry structure may be computed using the following command:

```
geometry = geo_load ({@ring_polar_map, @ring_polar_map_der});
```

As the knot vector for the solution B-Spline space cannot in this case be taken from the geometry, the user must generate one, before calling the msh and space constructors. As an example, the following call uses the kntuniform function from the `nurbs` toolbox to generate a uniform knot vector with nine knotspans (ten breaks) for basis functions of degree two and regularity one:

```
[knots, breaks] = kntuniform ([10 10], [2 2], [1 1]);
```

Apart from the modifications above the rest of the script remains identical to that of Listing 1.

5.1.3. Introducing other modifications: a different quadrature rule

As already noted in Remark 3.1, although in many cases they coincide, the partition of the domain $\widehat{\Omega}$ used for quadrature, $\widehat{\mathcal{K}}_h$, and the Cartesian partition $\mathcal{Q}_h$ induced by the knot vectors of the solution function space V_h are, in general, different. One simple yet notable example where selecting the elements of $\widehat{\mathcal{K}}_h$ so that they do not coincide with those of $\mathcal{Q}_h$ is the case where the quadrature rules on macro–elements introduced in [22] are used to reduce the total number of quadrature nodes by exploiting the higher smoothness of the B-Spline basis functions. In this particular case, each element of $\widehat{\mathcal{K}}_h$ is the union of a few neighboring cells in $\mathcal{Q}_h$. Implementing this approach is particularly simple in `GeoPDEs` where the definition of the domain partition is independent of the definition of the function space. To clarify this with an example, we solve the same problem as in Section 5.1.2 using a quadrature rule that integrates exactly functions of $(S_0^4)^2$ on 3 neighboring knot spans (this rule is given in Table 9 of [22] and is reported in Table 1 below for convenience). The nodes and weights of the quadrature rule need to be stored in the rows of a matrix as:

nodes	weights
$5.168367524056075 \times 10^{-2}$	$1.254676875668223 \times 10^{-1}$
$2.149829914261059 \times 10^{-1}$	$1.708286087294738 \times 10^{-1}$
$3.547033685486441 \times 10^{-1}$	$1.218323586744639 \times 10^{-1}$
$5.000000000000000 \times 10^{-1}$	$1.637426900584793 \times 10^{-1}$
$6.452966314513557 \times 10^{-1}$	$1.218323586744638 \times 10^{-1}$
$7.850170085738940 \times 10^{-1}$	$1.708286087294738 \times 10^{-1}$
$9.483163247594394 \times 10^{-1}$	$1.254676875668223 \times 10^{-1}$

Table 1: Quadrature rule to integrate exactly functions of S_0^4 on 3 neighboring knot spans

```
1 rule     = [nodes; weights];
```

Using the same knot vector as in Section 5.1.2 above, we can define the breaks for the partition $\widehat{\mathcal{K}}_h$ simply eliminating the redundant ones:

```
2 breaks{1} ([2:3:end-2,3:3:end-1]) = [];
3 breaks{2} ([2:3:end-2,3:3:end-1]) = [];
```

We can then proceed to assemble the msh structure with the following line

```
4 [qn, qw]  = msh_set_quad_nodes (breaks, {rule, rule}, [0 1]);
```

where the last input argument specifies that the quadrature rule is defined on the interval $[0, 1]$. Again, apart from the modifications above, the rest of the script remains identical to that of Listing 1.

5.1.4. Implementation of non-homogeneous boundary conditions

The previous examples had the only purpose of explaining the code and the construction of the different structures on which it is based, but we only considered a problem with homogeneous boundary conditions. We now address how to implement non-homogeneous boundary conditions of both Neumann and Dirichlet type using the **boundary** substructures of Section 4.6.

Let us consider the same domain Ω of Section 4, with the boundary parts $\Gamma_N = \{(x, y) : 0 \leq x \leq 1,\, y = 0\}$ and $\Gamma_D = \partial\Omega \setminus \Gamma_N$. Let us assume that the coefficient $k(\mathbf{x})$ is constant

and equal to one. Then, the problem

$$\begin{cases} -\Delta u &= 0 & \text{in } \Omega, \\ \dfrac{\partial u}{\partial \mathbf{n}} &= g = -e^x \cos(y) & \text{on } \Gamma_N, \\ u &= h = e^x \sin(y) & \text{on } \Gamma_D, \end{cases} \tag{23}$$

has exact solution $u = e^x \sin(y)$. Since the source term is equal to zero, the first modification to introduce in Listing 1 is to substitute line 9 by:

```
rhs = zeros (space.ndof, 1);
```

The exact solution has to be changed as well, so the line 15 now reads:

```
err = sp_l2_error (space, msh, u, @(x,y) exp(x) .* sin(y))
```

In order to apply the boundary conditions it is necessary to determine which kind of condition has to be applied on each side of the domain. This is done through the arrays nmnn_sides and drchlt_sides, that must be supplied by the user. Then we recall that the structure **msh.boundary** contains all the information for a quadrature rule defined on the boundary, and that **space.boundary** contains the information about the boundary functions and their values at the quadrature points. Using these structures, the computation of the Neumann condition is included after line 9 of Listing 1 as follows:

```
1 for iside = nmnn_sides
2  x = squeeze (msh.boundary(iside).geo_map(1,:,:));
3  y = squeeze (msh.boundary(iside).geo_map(2,:,:));
4  gval = -exp(x) .* cos(y);
5  rhs_side = op_f_v (space.boundary(iside), msh.boundary(iside), gval);
6  rhs(space.boundary(iside).dofs) = rhs(space.boundary(iside).dofs) + rhs_side;
7 end
```

That is, for each boundary with a Neumann condition, we first evaluate the function g at the quadrature points of the boundary partition. Then, the boundary term appearing in (10) is computed using the function op_f_v, which is the same as for the source term, but is invoked here with the fields defined on the boundary. Finally, the assembling of the global right-hand side is done by using the field **dofs**, already explained in Section 4.6.

The implementation of the Dirichlet boundary condition in IGA is not trivial, and it is still a matter of research (see [7, 25]). For our examples we have imposed the condition by means of an L^2–projection of the boundary data. The method is neither local nor efficient, but it provides a good example to show how the structures of the code can be used. The following piece of code should substitute lines from 10 to 12 in Listing 1:

```
1 drchlt_dofs = unique ([space.boundary(drchlt_sides).dofs]);
2 int_dofs = setdiff (1:space.ndof, drchlt_dofs);
```

```
3  M_drchlt = spalloc (space.ndof, space.ndof, space.ndof);
4  rhs_drchlt = zeros (space.ndof, 1);
5  for iside = drchlt_sides
6   sp_bnd = space.boundary(iside);
7   msh_bnd = msh.boundary(iside);
8   x = squeeze (msh_bnd.geo_map(1,:,:));
9   y = squeeze (msh_bnd.geo_map(2,:,:));
10  hval = exp(x) .* sin(y);
11  M_side = op_u_v (sp_bnd, sp_bnd, msh_bnd, ones(size(x)));
12  M_drchlt(sp_bnd.dofs, sp_bnd.dofs) = M_drchlt(sp_bnd.dofs, sp_bnd.dofs) +
        M_side;
13  rhs_side = op_f_v (sp_bnd, msh_bnd, hval);
14  rhs_drchlt(sp_bnd.dofs) = rhs_drchlt(sp_bnd.dofs) + rhs_side;
15 end
16 u = zeros (space.ndof, 1);
17 u(drchlt_dofs) = M_drchlt(drchlt_dofs, drchlt_dofs) \ rhs_drchlt(drchlt_dofs);
18 rhs(int_dofs) = rhs(int_dofs) - mat(int_dofs, drchlt_dofs) * u(drchlt_dofs);
```

The first four lines identify the degrees of freedom on the Dirichlet boundary, and provide the needed initializations. Lines 6 and 7 are inserted only for the sake of clarity. Then, for each Dirichlet boundary we compute the value of the function h at the quadrature points, by using the information of the **boundary** fields. From line 11 to 14 the matrix and the right-hand side are assembled to compute the L^2–projection, which is done in line 17. Finally, the right-hand side of the problem is corrected on line 18.

5.2. Linear elasticity

The present section is devoted to discussing how `GeoPDEs` can be utilized to solve structural mechanics problems, which represent one of the first and, up to now, most prominent applications of IGA,

As in the sections above, we choose a reference problem and go through the steps to solve it with `GeoPDEs`. In particular, let us consider the case of a linear, elastic, isotropic body filling the region $\Omega \subset \mathbb{R}^d$, $d = 2, 3$, with a part Γ_D of its boundary being kept fixed while a distributed load $\mathbf{g}$ is applied on the remaining part of the boundary Γ_N. The displacement $\mathbf{u}$ of the elastic body is given by the solution of the problem:

$$\textit{Find } \mathbf{u} \in V = \left(H^1_{0,\Gamma_D}(\Omega)\right)^d \textit{ such that}$$

$$\int_\Omega (2\mu\,\epsilon(\mathbf{u}) : \epsilon(\mathbf{v}) + \lambda\,\mathrm{div}(\mathbf{u})\,\mathrm{div}(\mathbf{v})) = \int_\Omega \mathbf{f}\cdot\mathbf{v} + \int_{\Gamma_N} \mathbf{g}\cdot\mathbf{v} \quad \forall \mathbf{v} \in V, \tag{24}$$

where λ and μ are the Lamé parameters of the material and $\epsilon(\mathbf{u})$ is the strain tensor, which is the symmetric part of the displacement gradients. For $d = 2$, this represents a problem in the plane strain regime.

The main peculiarity of (24) in comparison to the model problem considered up to now is that the space V, and its finite dimensional approximant V_h, are now spaces of

vector-valued functions. Constructing a vector-valued function space in GeoPDEs is quite straightforward. For example, in the case $d = 3$ one first computes, in the same manner we have seen in Section 4, the **msh** structure and the space structures **spx**, **spy** and **spz**, to which the components of the displacement belong. The vector-valued space structure is then computed as

```
sp = sp_scalar_to_vector_3d (spx, spy, spz, msh, 'divergence', true);
```

The main difference of the new structure **sp** with the one we have seen in Section 4.4, is that the fields **sp.shape_functions** and **sp.shape_function_gradients** have size $d \cdot n_k \cdot N_s \cdot N_e$, and $d \cdot d \cdot n_k \cdot N_s \cdot N_e$, respectively. Each basis function in the new space structure will have only one non-zero component, and the indices of the basis functions for which the i-th component is non-zero are listed in the array **sp.comp_dofs**{i}, $i = 1, \ldots, d$, while the number d of components of the vector-valued space is stored in the field **sp.ncomp**.

We also remark that, in the command above, the option 'divergence' is set to true in the call to sp_scalar_to_vector_3d, which means that the divergences of each basis function are to be precomputed and stored in **sp**.shape_function_divs in order to save time when assembling the stiffness matrix. Such assembly is done via the function call

```
mat = op_su_ev (sp, sp, msh, lambda, mu);
```

where lambda, mu are the Lamé parameters evaluated at the quadrature points. Note that the call to the operator assembly function is the same regardless of whether $d = 2$ or $d = 3$.

Apart from the modifications described above, everything in the problem solution script follows the same approach as shown for previous examples. In particular, postprocessing functions are implemented in such a way that they operate both on scalar and vector-valued spaces of functions. To show the behavior of GeoPDEs for the solution of elasticity problems, we present below both a 2D and a 3D numerical example.

5.2.1. Plane strain example

We consider a cross-section of a thick cylinder subjected to a constant pressure on its interior and to a prescribed, radially directed, displacement on the exterior. Exploiting the symmetry of the problem we can simulate only one quarter of the structure by imposing that the displacement is radially directed at the artificial boundaries, we can therefore use for the simulation the same computational domain geometry as for the example, with an internal radius $R_i = 1$ and an external one $R_o = 2$. The exact solution of this problem can be expressed in polar coordinates as:

$$u_r = \frac{PR_i^2}{E(R_o^2 - R_i^2)} \left((1-\nu)r + \frac{(1+\nu)R_o^2}{r} \right).$$

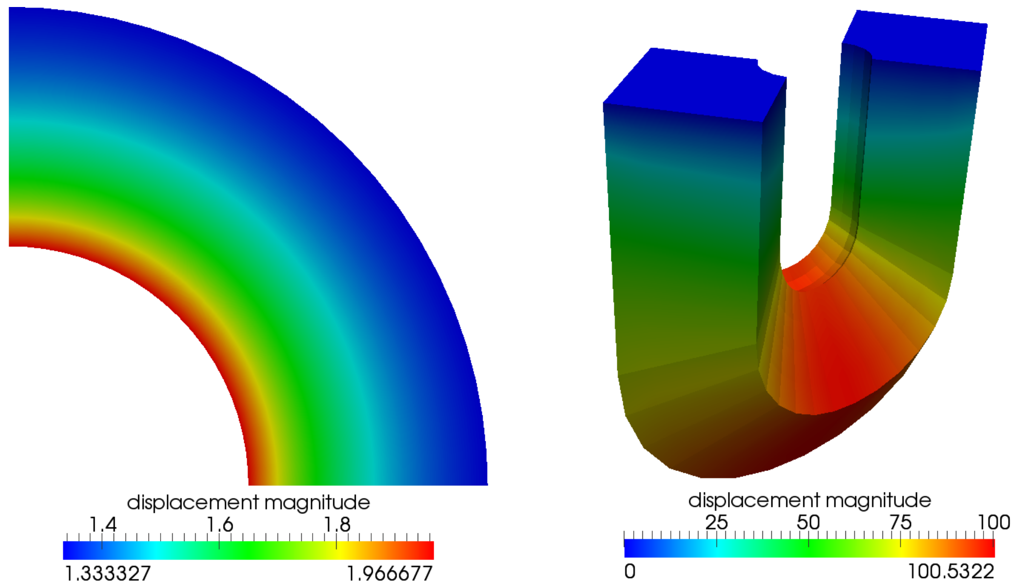


Figure 2: Solution of linear elasticity examples

The displacement magnitude for a value of the pressure of $P = 1$ and for a material with Young modulus $E = 1$ and Poisson ratio $\nu = 0$ (corresponding to $\lambda = 0$ and $\mu = 1/2$), computed in a space of NURBS of degree 3 and regularity 2, are shown in Fig. 2(a).

5.2.2. 3D Linear elasticity example

As an example of 3D structural analysis we consider the "horseshoe"-shaped solid presented in Section 4.3 of [1]. We set the material properties to $E = 1$ and $\nu = .3$, we prescribe a null displacement on the flat faces and homogeneous Neumann boundary conditions on the remaining faces, while the body force $\mathbf{f}$ is directed in the negative z direction and has a constant modulus of 1. The contour plot of the displacement magnitude for a geometry represented by degree 3 NURBS functions is depicted in Fig. 2(b).

5.3. Stokes equations

While in structural analysis the isoparametric paradigm is of great importance, in other applications, where it is not as fundamental, this requirement may be relaxed in order to make the most of other features of IGA. In incompressible fluid dynamics, for instance, the isoparametric paradigm can be traded off to obtain a discretization method in which the incompressibility constraint is satisfied exactly [11]. In this section we show how to use `GeoPDEs` to solve the 2D Stokes problem with the approximation methods introduced in [11]. The mixed variational formulation of the Stokes equation describing the flow of a viscous

incompressible fluid of constant viscosity μ reads:

$$\begin{aligned}
&\textit{Find } \mathbf{u} \in (H^1_0(\Omega))^2 \textit{ and } p \in L^2(\Omega)/\mathbb{R} \textit{ s.t.} \\
&\int_\Omega \mu \boldsymbol{\nabla} \mathbf{u} : \boldsymbol{\nabla} \mathbf{v} - \int_\Omega p \text{ div } \mathbf{v} = \int_\Omega \mathbf{f} \cdot \mathbf{v} \qquad \forall \mathbf{v} \in (H^1_0(\Omega))^2 \\
&\int_\Omega q \text{ div } \mathbf{u} = 0 \qquad \forall q \in L^2(\Omega)/\mathbb{R}.
\end{aligned} \tag{25}$$

Three compatible pairs of discretization spaces are considered in [11] in which the discrete counterpart of (25) can be set. Recalling the notation of Section 2.2, and assuming for simplicity that the regularity α is the same at all knots, we first define a B-Spline space for the pressure in the parametric domain as $\widehat{Q}_h \equiv \widehat{Q}_h(p, \alpha) = S^{p,p}_{\alpha,\alpha}$, which will be the same for the three pairs. The discrete spaces for the velocity are then defined, in the parametric domain, as

$$\widehat{V}_h^{\text{TH}} = S^{p+1,p+1}_{\alpha,\alpha} \times S^{p+1,p+1}_{\alpha,\alpha}; \quad \widehat{V}_h^{\text{RT}} = S^{p+1,p}_{\alpha+1,\alpha} \times S^{p,p+1}_{\alpha,\alpha+1}; \quad \widehat{V}_h^{\text{NDL}} = S^{p+1,p+1}_{\alpha+1,\alpha} \times S^{p+1,p+1}_{\alpha,\alpha+1}.$$

Here TH, RT and NDL stand for Taylor-Hood, Raviart-Thomas and Nédélec (of the second kind) respectively, as these B-Spline spaces are consistent generalizations of the well-known finite element spaces known by these names. As explained in [11], the way to map the spaces to the physical domain is different in each case. In particular the TH spaces can be mapped to the physical domain Ω by the same "component-wise" mapping used in the previous section, i.e.

$$V_h^{\text{TH}} = \{\mathbf{v} : \mathbf{v} \circ \mathbf{F} \in \widehat{V}_h^{\text{TH}}\}, \quad Q_h = \{q : q \circ \mathbf{F} \in \widehat{Q}_h\}. \tag{26}$$

The same mapping for the pressure space is used in the two other pairs. On the other hand, to get stable discretizations, the velocity spaces for RT and NDL are transformed via a Piola-type mapping, i.e.

$$V_h^{\text{RT}} = \left\{ \frac{D\mathbf{F}}{\det(D\mathbf{F})} \mathbf{v} : \mathbf{v} \circ \mathbf{F} \in \widehat{V}_h^{\text{RT}} \right\}, \quad V_h^{\text{NDL}} = \left\{ \frac{D\mathbf{F}}{\det(D\mathbf{F})} \mathbf{v} : \mathbf{v} \circ \mathbf{F} \in \widehat{V}_h^{\text{NDL}} \right\}. \tag{27}$$

Furthermore, when no-slip boundary conditions are applied to the whole boundary of Ω, the discretization based on RT spaces suffers a loss of accuracy in the pressure approximation. One of the possible remedies proposed in [11] is to substitute the pressure space Q_h by a smaller space $\widetilde{Q}_h \subset Q_h$, which is a space of T-Spline functions (see [26]) obtained by eliminating one degree-of-freedom per corner, as shown in Fig. 5.3.

`GeoPDEs` provides functions to build the function space structures corresponding to the TH, NDL and RT space pairs. The way these functions are invoked is shown in the following three lines of code:

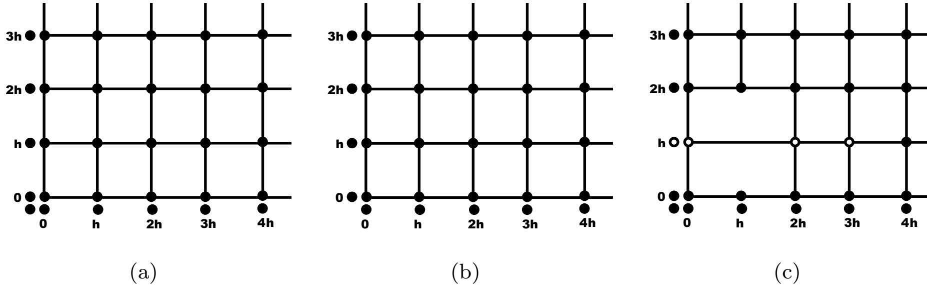


Figure 3: Removal of redundant degrees of freedom of $\widehat{Q}_h^{\mathrm{RT}}$

```
[spv, spp]    = sp_bspline_th_2d_phys  (knotsp, degree, msh);
[spv, spp]    = sp_bspline_ndl_2d_phys (knotsp, degree, msh);
[spv, spp, P] = sp_bspline_rt_2d_phys  (knotsp, degree, msh);
```

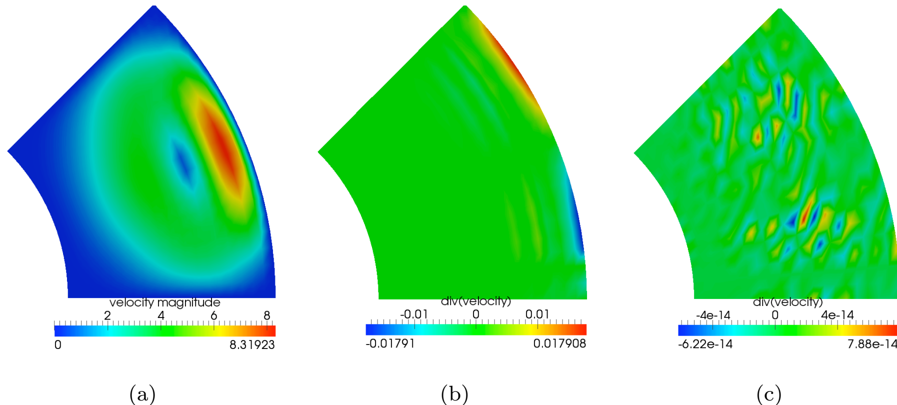
In each of the calls above, knotsp and **degree** refer to the discrete space for the pressure. At the interior of these functions, the pressure space structure **spp** is computed as in Section 4.4. Then, the knot vectors and degrees for each component of the velocity are automatically computed from those of the pressure, and the **spv** structure is computed as in Section 5.2. Finally, for NDL and RT spaces the Piola-mapping is also applied.

Of particular interest is the additional output P given by the RT space constructor, which is a matrix that operates the change of basis from the B-Spline space Q_h to the T-Spline space $\widetilde{Q}_h$. The purpose for computing P is that it allows to work with functions in $\widetilde{Q}_h$ without the need of implementing a constructor for general T-Spline spaces.

As an example we consider a problem set on one eighth of an annulus, described through a NURBS parameterization, with no-slip boundary conditions on the whole domain boundary. The exact solution and right-hand side can be found in the file test_stokes_annulus of the package. In Fig. 4(a) we show the velocity computed with RT elements for $p = 3$ and $\alpha = 1$, while in Fig. 4(b) and Fig. 4(c) we compare the divergence of the velocity field obtained by the TH discretization scheme with that of a RT scheme with the same degree and regularity for the pressure component of the solution.

5.4. Maxwell equations

The main interest for using IGA in electromagnetism, apart from the exact description of geometry, is the higher regularity of the solution with respect to finite elements. For edge finite elements, spreadly used in computational electromagnetims, the normal component of the computed solution is discontinuous. This property is useful to simulate problems



(a) (b) (c)

Figure 4: Velocity magnitude for RT (a), and divergence for TH (b) and RT (c).

with several materials, since the normal component of the physical solution is also discontinuous, but within each material higher regularity may be desirable. The B-Spline-based discretization introduced in [9, 10], besides maintaining exact geometry, provides more regular solutions than edge elements. We now show how these methods are implemented in `GeoPDEs`, explaining the main modifications that have to be done in order to solve Maxwell's eigenvalue problem. The explanation of the discretization scheme is extensively described in the aforementioned papers, along with several numerical examples.

We focus here on the 3D Maxwell eigenproblem. Let $\Omega \subset \mathbb{R}^3$ be defined as in (1) and let its boundary be split into two disjoint parts, $\partial\Omega = \Gamma_D \cup \Gamma_N$. The problem with mixed boundary conditions reads:

$$\begin{cases} \mathbf{curl}\,(\mu^{-1}\,\mathbf{curl}\,\mathbf{E}) = \lambda\epsilon\mathbf{E} & \text{in } \Omega, \\ \mathbf{E}\times\mathbf{n} = \mathbf{0} & \text{on } \Gamma_D, \\ \mu^{-1}\,\mathbf{curl}\,\mathbf{E}\times\mathbf{n} = \mathbf{0} & \text{on } \Gamma_N. \end{cases}$$

The equivalent weak problem is to find an electric field $\mathbf{E} \in \mathbf{H}_{0,\Gamma_D}(\mathbf{curl};\Omega)$ such that

$$\int_\Omega \mu^{-1}\,\mathbf{curl}\,\mathbf{E}\cdot\mathbf{curl}\,\mathbf{v} = \lambda \int_\Omega \epsilon\mathbf{E}\cdot\mathbf{v} \quad \forall \mathbf{v} \in \mathbf{H}_{0,\Gamma_D}(\mathbf{curl};\Omega)\,,$$

with $\mathbf{H}_{0,\Gamma_D}(\mathbf{curl};\Omega)$ the space of square integrable vectorial functions in Ω such that their curl is also square integrable, and their tangential components are null on the boundary Γ_D.

Using the notation of Section 2.2, and assuming again that the multiplicity is the same for all internal knots, the discrete space in the parametric domain $\widehat{\Omega}$ is taken equal to $\widehat{V}_h := S^{p_1-1,p_2,p_3}_{\alpha_1-1,\alpha_2,\alpha_3} \times S^{p_1,p_2-1,p_3}_{\alpha_1,\alpha_2-1,\alpha_3} \times S^{p_1,p_2,p_3-1}_{\alpha_1,\alpha_2,\alpha_3-1}$. This belongs to a sequence of discrete spaces that, along with their continuous counterparts, form a commuting De Rham diagram (see

[9]). The discrete space in the physical domain is then defined by using a curl conserving transformation [27, Section 3.9], in the form

$$V_h = \left\{ \mathbf{u} : (D\mathbf{F})^T (\mathbf{u} \circ \mathbf{F}) \in \widehat{V}_h \right\}.$$

For the implementation, the **geometry** and **msh** structures are computed exactly as in all the previous problems. The only differences appear in the structure **space**. This is constructed with the commands:

```
[knots1, knots2, knots3, degree1, degree2, degree3] = knt_derham (knots, degree);
space = sp_bspline_curl_transform_3d (knots1, knots2, knots3, degree1, ...
    degree2, degree3, msh);
```

where the first line automatically constructs the knot vectors and degrees for every component of the product space V_h, and the second one constructs the new **space** structure. Internally, this function first builds the structure in the parametric domain in the same fashion we explained in Section 5.2, and then applies the curl-conserving transform, similarly to what done in Section 5.3 for the Piola transform.

The resulting structure is roughly the same we have seen for the other vectorial problems, the main difference being that it includes the field shape_function_curls, which has size equal to $3 \cdot n_k \cdot N_s \cdot N_e$. The second important difference is that, since the boundary conditions just involve the tangential components of the solution, only the shape functions for the tangential components are stored in the **boundary** field.

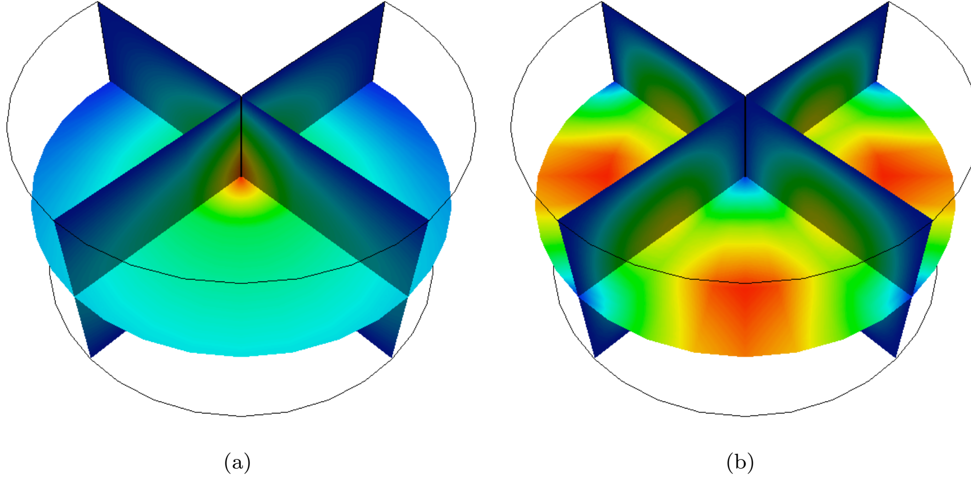
After the construction of the structure, the matrices are assembled in the same manner we have seen for previous problems, using the commands

```
stiff_mat = op_curlu_curlv_3d (space, space, msh, 1./mu);
mass_mat  = op_u_v (space, space, msh, epsilon);
```

where mu and epsilon are the values of the physical parameters evaluated at the quadrature points. The eigenvalue problem is then solved by using the **eig** command from Matlab or Octave.

The postprocessing part is also analogous to what we have explained before. In Fig. 5 we show the magnitude of the second and fifth eigenfunctions in a three quarters of the cylinder, with $\partial\Omega = \Gamma_D$. In this case the geometry is exactly constructed with only three elements.

Finally, we would like to remark that the code includes examples for different formulations of the eigenvalue problem (see [28]), and for the source problem with non-homogeneous boundary conditions.



(a) (b)

Figure 5: Magnitude of the second and fifth eigenfunctions in three quarters of the cylinder.

Remark 5.1. *The 2D and the 3D cases are a little bit different, since in the former we have two different differential operators:* **curl** *and* curl *. The operator* **curl** *acts on scalar variables and returns vector-fields, whereas the operator* curl *acts on vector-fields and returns scalar variables. Something similar occurs with the tangential boundary conditions. For this reason, some functions have different versions for the 2D and the 3D cases.*

6. The treatment of NURBS multipatch conforming geometries

We have included in the code a small package to deal with multipatch geometries. For this we have always assumed that the patches are compatible, in the sense that the meshes and control points must coincide on the interface, even after refinement. More sophisticated approaches, allowing for different refinements on each patch, can be found in [12, Ch. 3].

Let us assume that the domain Ω is formed by the union of n_p disjoint subdomains, or patches, in the form $\overline{\Omega} = \cup_{l=1}^{n_p} \overline{\Omega}_l$, with $\Omega_l \cap \Omega_{l'} = \emptyset, \forall l \neq l'$. Each patch is defined, analogously to (8), as $\mathbf{F}_l : \widehat{\Omega} \longrightarrow \Omega_l$, with $\widehat{\Omega}$ being always the unit square or cube. We require that the patches match conformingly, in the sense that, if Ω_i and Ω_j are two patches with a common interface $\Gamma_{ij} = \overline{\Omega}_i \cap \overline{\Omega}_j \neq \emptyset$, and $\mathcal{Q}_h^i, \mathcal{Q}_h^j$ are their respective meshes, then they must coincide on the interface, i.e., $\mathcal{Q}_h^i|_{\Gamma_{ij}} = \mathcal{Q}_h^j|_{\Gamma_{ij}}$, with the same multiplicity for the corresponding knots.

Since the meshes are the same, the basis functions are also the same for both patches. Hence, it is sufficient to define a connectivity array, which identifies the matching basis functions on each patch with one single function in the global domain (see [12]). In order

to define this connectivity, we have made use of the data format for multipatch geometries proposed in [29].

Let us start with the 2D case. For each interface we must know: the number of the two patches, the numbers of the coinciding boundary surfaces on each patch, and a flag telling if the parametric direction on the first patch coincides with the parametric direction on the second. The first numbers let us know, for each patch, which degrees of freedom do not vanish on the interface (stored in **boundary.dofs**). The flag tells us whether the degrees of freedom match automatically, or if those of the second patch must be reordered (using the **fliplr** command).

In the 3D case, instead, the information we need for each interface is the following: the number of the two patches, and of their matching boundaries, a flag telling if the first parametric coordinate on the first surface coincides with the first parametric coordinate of the second surface, and two flags telling if each parametric direction coincides. Using the first numbers, we recover the degrees of freedom for each patch from **boundary.dofs**, and rewrite them in the form of two matrices. The first flag tells if the second matrix should be transposed, whereas the two other flags let us know if the degrees of freedom in this matrix should be reordered, one with the **fliplr** command and the other with the **flipud** command.

Remark 6.1. *Dealing with multipatch geometries for vector field approximations is a little bit more complicated. In particular, when applying the curl-conserving or Piola transforms, it is necessary to define an orientation for the tangential and the normal components, respectively. However, this is not much different from what is done for edge and face finite elements.*

7. Conclusion

In this paper, we have presented the design philosophy and main features of `GeoPDEs`, a suite of free software tools for applications on Isogeometric Analysis. A first goal of `GeoPDEs` is to constitute an entry point for researchers interested in the practical issues related to the implementation of an IGA code, while another important aim is to provide a rapid prototyping and testing tool for the development of novel IGA algorithms.

With these main objectives in mind, we have tried to explain all basic features and capabilities of `GeoPDEs` on a simple model problem (i.e., Poisson), showing also how to use the code as a starting point to develop new IGA methods to be applied to different engineering fields. For instance, applications to elasticity, Stokes and Maxwell problems have been discussed and the solution of some numerical examples has been shown. Moreover,

among the different topics related to the use and possible extension of `GeoPDEs` covered in this paper, we have considered in particular some delicate issues such as $h-$, $p-$, or $k-$refinement strategies, boundary condition imposition, implementation of non-isoparametric methods, use of different quadrature strategies, treatment of multipatch geometries.

As a conclusion, we believe that the present work, along with the release of the `GeoPDEs` suite, may complement the literature on IGA (see, e.g., [12] and references therein), constituting on one hand an important tool for people meeting IGA for the first time and on the other hand the basis for the rapid development of new IGA ideas and applications.

Acknowledgments

The authors were partially supported by the European Research Council through the FP7 Ideas Starting Grant 205004: *GeoPDEs – Innovative compatible discretization techniques for Partial Differential Equations.* This support is gratefully acknowledged. The authors also wish to thank M. Bercovier, A. Buffa, T.J.R. Hughes and G. Sangalli, for many fruitful discussions during the preparation of this work.

A. The NURBS toolbox

For the description of NURBS geometric entities, and also for the computation of the shape functions, we use the NURBS toolbox, originally implemented in Matlab by Mark Spink [30]. The original toolbox was mainly developed for the construction of NURBS curves and surfaces, based on the algorithms of [19]. In order to deal with three-dimensional problems in IGA, we have extended some of the basic algorithms to trivariate NURBS. The new version can be found in the svn repository of the Octave sourceforge project, and it is still compatible with Matlab. We give now a short explanation of the main features of the toolbox, and refer the reader to [30] for more detailed documentation.

The geometric entities in the NURBS toolbox are described by a structure, that contains all the necessary information. The main fields of this structure are listed below, and explained using the same notation as in Section 2.2:

- **order**: a vector with the order in each direction. In the splines literature, the B-Splines of degree p are said to have order $k = p + 1$. Thus, it stores the values $p_d + 1$.
- **knots**: knot vectors Ξ_d, stored as a cell array.
- **number**: n_d, number of basis functions in each direction.

- **coefs**: weighted control points, stored in an array of size $(4, n_1)$ for a curve, $(4, n_1, n_2)$ for a surface, and $(4, n_1, n_2, n_3)$ for a volume. That is, the first three rows contain the coordinates of the control points $\mathbf{C_j}$ multiplied by the weight $w_\mathbf{j}$, and the fourth row contains the weight $w_\mathbf{j}$ (see [19, Section 4.2]). The weights are always stored in the fourth coordinate, even for two-dimensional geometries, and for B-Splines they are equal to one.

The toolbox contains several functions performing the basic operations with NURBS. The function nrbmak is very useful to create simple NURBS geometries. It is invoked passing the control points and the knots as arguments, and it returns a NURBS structure as we have just seen. For instance, the lines of code

```
1 coefs (1:3, 1, 1) = [1; 0; 0];                      coefs(4, 1, 1) = 1;
2 coefs (1:3, 1, 2) = [sqrt(2)/2; sqrt(2)/2; 0];      coefs(4, 1, 2) = sqrt(2)/2;
3 coefs (1:3, 1, 3) = [0; 1; 0];                      coefs(4, 1, 3) = 1;
4 coefs (1:3, 2, 1) = [2; 0; 0];                      coefs(4, 2, 1) = 1;
5 coefs (1:3, 2, 2) = [sqrt(2); sqrt(2); 0];          coefs(4, 2, 2) = sqrt(2)/2;
6 coefs (1:3, 2, 3) = [0; 2; 0];                      coefs(4, 2, 3) = 1;
7 knots = {[0 0 1 1], [0 0 0 1 1 1]};
8 nurbs = nrbmak (coefs, knots);
```

would create the geometry of Fig. 1. The assignment of the weight values has been separated for the sake of clarity.

In Section 5.1.1 we have used the functions nrbkntins and nrbdegelev, which perform knot insertion and degree elevation, respectively. The first one is invoked by passing a NURBS structure and a cell-array containing the knots that will be inserted in each direction. The arguments for the second one are a NURBS structure, and an array telling how much the degree will be raised in each direction. The output of both functions is a NURBS structure. The following lines of code

```
 9 nurbs = nrbdegelev (nurbs, [1 0]);
10 new_knots = linspace (0, 1, 10);
11 nurbs = nrbkntins (nurbs, {new_knots(2:end-1) new_knots(2:end-1)});
```

would raise the degree of the surface by 1 in the first parametric direction, and then new knots would be inserted uniformly in both directions. The result is exactly the NURBS geometry used in the example of Section 4, and contained in the file 'ring_refined.mat'.

B. Summary of data structures

In this section we summarize for convenience the fields that compose the main data structures of GeoPDEs. These can be found in Tables 2, 3 and 4 for the **geometry**, **msh** and **space** structures, respectively.

Field name	Type	Dimensions	Description
map	function handle	1×1	a function to evaluate the parameterization
map_der	function handle	1×1	a function to evaluate the first derivatives of the parameterization
Optional fields			
map_der2	function handle	1×1	a function to evaluate the second order derivatives of the parameterization
nurbs	struct	1×1	If the geometry is a NURBS, a structure in the NURBS toolbox format

Table 2: The **geometry** data structure

C. A new version of the code, `GeoPDEs` 2.0.0

In September 2011, `GeoPDEs` 2.0.0 was released. The main goal of this version was to improve the efficiency of the code, both in terms of computational time and memory consumption, for which it was necessary to change important parts of the implementation. We present here a brief description of the changes introduced in this version, with the help of the examples given in Sections 4 and 5 to show how they can be adapted to `GeoPDEs` 2.0.0. All these examples are also included in the packages to download.

C.1. The simple example revisited

Let us begin with the simple example given in Section 4.1, that is solved in the new version with the code given in Listing 3. All the important differences are in lines from 4 to 11, so we will focus on them.

The main change with respect to previous versions is that the **msh** and **space** structures have been transformed into classes. However, the commands to access the information on these classes is the same that was used for the structures, so a normal user should not find any problem about this. What is new is that, in order to save memory, the biggest fields of the old structures are not precomputed at every quadrature point, but instead they are computed whenever they are needed, as we will see below.

Remark C.1. *In fact there are two different classes for the mesh:* msh_2d *and* msh_3d*, and four different classes for the scalar spaces, namely* sp_nurbs_2d*,* sp_nurbs_3d*,* sp_bspline_2d *and* sp_bspline_3d*. We will often refer to them in a general way as the* **msh** *and* **space** *classes.*

Field name	Type	Dimensions	Description
nel	scalar	1×1	number of elements of the partition
nqn	scalar	1×1	number of quadrature nodes per element
quad_nodes	NDArray	nd×nqn×nel	coordinates of the quadrature nodes in parametric space, nd being the number of space dimensions (2 or 3)
quad_weights	Matrix	nqn×nel	weights associated to the quadrature nodes
geo_map	NDArray	nd×nqn×nel	physical coordinates of the quadrature nodes
geo_map_jac	NDArray	nd×nd×nqn×nel	Jacobian matrix of the map evaluated at the quadrature nodes
jacdet	Matrix	nqn×nel	determinant of the Jacobian evaluated at the quadrature nodes
boundary	struct-array	1×(2 · nd)	an (nd - 1)-dimensional **msh** structure for each side of the boundary
Optional fields			
qn	cell-array	1×nd	if the mesh is tensor-product, quadrature nodes along each direction
breaks	cell-array	1×nd	if the mesh is tensor-product, breaks along each direction
geo_map_der2	NDArray	nd×nd×nd×nqn×nel	second order derivatives of the map evaluated at the quadrature nodes
normal	NDArray	nd×nqn×nel	if the mesh is a boundary mesh, exterior normal evaluated at quadrature nodes

Table 3: The **msh** data structure (version 1.1.0 or earlier)

Field name	Type	Dimensions	Description
ndof	scalar	1×1	total number of degrees of freedom
ncomp	scalar	1×1	number of components of the vector field
nsh_max	scalar	1×1	maximum number of shape functions per element
nsh	Matrix	1×nel	actual number of non-vanishing shape functions on each element
connectivity	Matrix	nsh_max×nel	indices of non-vanishing shape functions on each element
shape_functions	NDArray	ncomp×nqn× nsh_max×nel	basis functions evaluated at each quadrature node in each element
shape_function_gradients	NDArray	ncomp×nd×nqn× nsh_max×nel	basis function gradients evaluated at each quadrature node in each element
spfun	function handle	1×1	function to evaluate an element of the discrete function space, given the Fourier coefficients and a set of points in the parametric space
boundary	struct array	1×(2 · nd)	struct array representing the space of traces of basis functions on each edge
Optional fields			
ndof_dir	Matrix	1×nd	if the space is tensor product, degrees of freedom along each direction
comp_dofs	cell-array	1×ncomp	indices of shape functions for each component of the vector field
dofs	Matrix	1×ndof	if the space is a boundary space, global indices of shape functions

Table 4: The **space** data structure (version 1.1.0 or earlier)

```
1 geometry = geo_load ('ring_refined.mat');
2 knots = geometry.nurbs.knots;
3 [qn, qw] = msh_set_quad_nodes (knots, msh_gauss_nodes (geometry.nurbs.order));
4 msh = msh_2d (knots, qn, qw, geometry);
5 space  = sp_nurbs_2d (geometry.nurbs, msh);
6 mat = op_gradu_gradv_tp (space, space, msh, @(x,y) ones (size (x)));
7 rhs = op_f_v_tp (space, msh, ...
     @(x,y)(8-9*sqrt(x.^2+y.^2)).*sin(2*atan(y./x))./(x.^2+y.^2));
8 drchlt_dofs = [];
9 for iside = 1:4
10   drchlt_dofs = union (drchlt_dofs, space.boundary(iside).dofs);
11 end
12 int_dofs = setdiff (1:space.ndof, drchlt_dofs);
13 u = zeros (space.ndof, 1);
14 u(int_dofs) = mat(int_dofs, int_dofs) \ rhs(int_dofs);
15 sp_to_vtk (u, space, geometry, [20 20], 'laplace_solution.vts', 'u')
16 err = sp_l2_error (space, msh, u, ...
     @(x,y)((x.^2+y.^2)-3*sqrt(x.^2+y.^2)+2).*sin(2.*atan (y./x)))
```

Listing 3: Solving the model problem with GeoPDEs.

The constructor of the **msh** class works in one single call, instead of computing the mesh in the parametric domain and then passing to the physical domain. The constructor is called in the following way:

```
msh = msh_2d (knots, qn, qw, geometry);
```

The output is an object of the class msh_2d, that contains the fields (in fact, they should be called properties) listed in Table 5. Notice that most of these are fields that already existed for the **msh** structure in previous versions. They can be accessed in the same way as the fields of any structure (for instance, **msh.nel** gives the number of elements).

Then, the constructor of the **space** is called in the line:

```
space  = sp_nurbs_2d (geometry.nurbs, msh);
```

We have removed any reference to the physical domain in the name of the function. The output is an object of the class sp_nurbs_2d, that contains the fields (properties) listed in Table 7. Again, most of these were already contained in the old **space** structure, but the biggest arrays, such as **shape_function_gradients**, have been removed, and will be only computed when necessary.

We will give some more details about the **msh** and **space** classes later, as soon as we need them. For now let us concentrate on the example. The next two lines are for the computation of the matrix and the right-hand side:

```
mat = op_gradu_gradv_tp (space, space, msh, @(x,y) ones (size (x)));
```

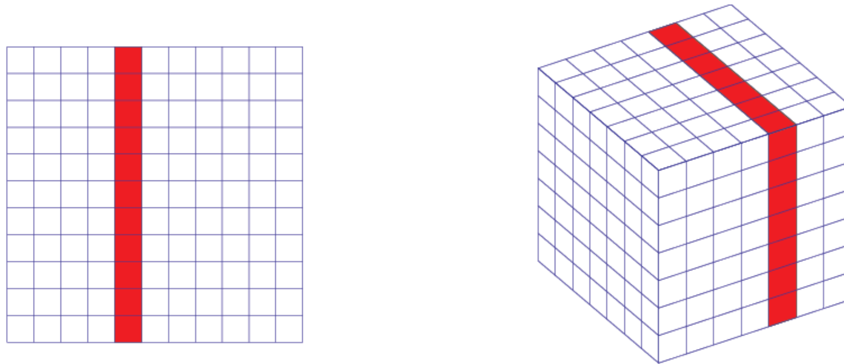


Figure 6: Elements considered at each step of the loop in the tensor product operators

```
rhs = op_f_v_tp (space, msh, ...
    @(x,y)(8-9*sqrt(x.^2+y.^2)).*sin(2*atan(y./x))./(x.^2+y.^2));
```

The first important change here is that the last argument is not the coefficient evaluated at the quadrature points, but a function handle to compute it. The coefficients will be evaluated inside each function. The second difference is in the names of the functions of the operators, that now end with _tp, that stands for *tensor product*. This is because the new version of the operators makes a better use of the tensor product structure of the spaces. As an example, we show below the code of the operator op_gradu_gradv_tp, to compute the stiffness matrix.

```
1  function A = op_gradu_gradv_tp (space1, space2, msh, coeff)
2    A = spalloc (space2.ndof, space1.ndof, 3*space1.ndof);
3    ndim = numel (msh.qn);
4    for iel = 1:msh.nel_dir(1)
5      msh_col = msh_evaluate_col (msh, iel);
6      sp1_col = sp_evaluate_col (space1,msh_col,'value',false,'gradient',true);
7      sp2_col = sp_evaluate_col (space2,msh_col,'value',false,'gradient',true);
8      for idim = 1:ndim
9        x{idim} = reshape (msh_col.geo_map(idim,:,:),msh_col.nqn,msh_col.nel);
10     end
11     A = A + op_gradu_gradv (sp1_col,sp2_col,msh_col,coeff(x{:}));
12   end
13 end
```

The first two lines are for memory allocation and to compute the dimension of the problem, since the function must work both for 2D and 3D problems. But the important thing is the loop and what happens inside it. The idea is very simple: at every step of the loop we fix the element in the first parametric direction, and then compute everything in one "column" of elements, obtained by changing the other directions (see Figure 6).

The **msh** class contains a function (to be precise, it should be called a method) that

computes all the information related to the quadrature rule in one column. The function is invoked by giving as arguments an **msh** object and the number of the element in the first parametric direction, as follows:

```
msh_col = msh_evaluate_col (msh, iel);
```

The output is a **msh** structure, as the one described in Section 4.3, but only the values for the selected column are computed.

Analogously, the **space** class contains a function (method) that computes the fields of the **space** structure for one column of the mesh. An example of its usage is the following

```
sp1_col = sp_evaluate_col (space1,msh_col,'value',false,'gradient',true);
```

The first two input arguments are a **space** object, and a **msh** structure for just one column, as the one computed before. The other arguments are to decide which fields of the structure should be computed. For instance, in our example the output argument is a **space** structure with the values for one column of the mesh, but since we are computing the stiffness matrix, we ask the method to compute the field **shape_function_gradients**, but not the field **shape_functions**, that is not necessary.

After that, the command

```
x{idim} = reshape (msh_col.geo_map(idim,:,:),msh_col.nqn,msh_col.nel);
```

computes the quadrature points, to evaluate the coefficients. Finally, the contribution to the stiffness matrix by the column of elements is given in the line

```
A = A + op_gradu_gradv (sp1_col,sp2_col,msh_col,coeff(x{:}));
```

Notice that this function is the same as the one described in Section 4.5, and it is invoked in the same way[2]: the first arguments are two **space** structures and a **msh** structure, with the difference that now they only contain the fields for one column of the mesh. And as in the previous version, the last argument is the coefficient evaluated at the quadrature points, but for just one column of the mesh.

Let us now continue with the example of Listing 3. The lines from 8 to 11 are to identify the degrees of freedom on the boundary.

```
drchlt_dofs = [];
for iside = 1:4
  drchlt_dofs = union (drchlt_dofs, space.boundary(iside).dofs);
end
```

[2]The functions without the _tp termination compute the same thing that in version 1.1.0, but in a more efficient way than before.

This is not nice, since we are doing in four lines the same thing that was done in one single line in the previous version. The problem is that a structure that is a field of a class, is not accessed in the same way as a structure that is a field of another structure[3]. This will force to access the boundary fields always within a loop, but as the number of boundaries is small, we think that this will not affect the efficiency of the code. We will give more details about boundary conditions in the next section.

Finally, in the lines 15 and 16 the difference is that now the functions for postprocessing receive as input arguments the **space** and **msh** objects, instead of the old structures. These functions also take advantage of the tensor product structure of the spaces, in the same way we have seen for the operators. Another difference is that the dimension of the space has been removed from the name of the sp_to_vtk function, since now the same function works for 2D and 3D domains.

C.2. Implementation of non-homogeneous boundary conditions

In the next sections we explain how the examples of Section 5 can be adapted to `GeoPDEs` 2.0.0. We will skip some of the examples, since the modifications are analogous to the ones we have just seen, and start with the implementation of non-homogeneous boundary conditions, i.e., the example in Section 5.1.4.

In previous versions of the code, the **msh** and **space** structures contained a field called **boundary**, with all the information about the quadrature rule and the shape functions on the boundary. In the new version, in order to save memory, the corresponding **boundary** field only contains part of this information (see Tables 5 and 7 for details). The new classes also include the methods to compute the rest of the information on each side of the boundary. These methods are called **msh_eval_boundary_side** and **sp_eval_boundary_side**, respectively. The following lines of code, that are used to impose the Neumann boundary condition of the problem, show how these functions can be used:

```
for iside = nmnn_sides
  msh_side = msh_eval_boundary_side (msh, iside);
  sp_side  = sp_eval_boundary_side (space, msh_side);
  x = squeeze (msh_side.geo_map(1,:,:));
  y = squeeze (msh_side.geo_map(2,:,:));
  gval = - exp(x) .* cos(y);
  rhs_loc = op_f_v (sp_side, msh_side, gval);
  rhs(sp_side.dofs) = rhs(sp_side.dofs) + rhs_loc;
end
```

[3]This can be fixed by changing the method subsref in each new class.

The function **msh_eval_boundary_side** takes as input arguments the **msh** object and the number of the boundary side for which we want to do the computations. The output is a structure with all the necessary fields to apply the quadrature rule on that boundary, as in the **boundary** field of the old version.

The function **sp_eval_boundary_side**, takes as input arguments the **space** object and, instead of the side number, the **boundary** structure just computed for the **msh**. It gives as the output a structure equivalent to the **boundary** field of the old **space** structure.

Since the computed **boundary** structures contain the same fields than in previous versions, in the other lines of code we only have to replace msh.**boundary**(iside) and sp.**boundary**(iside) by msh_side and sp_side, respectively. Notice that in this case the operator is the one of the previous version (not the tensor product one).

Also for the computation of Dirichlet boundary conditions it is necessary to construct the structures for each boundary. Thus, lines 6 and 7 in the example of Section 5.1.4, must be replaced by the following two lines:

```
msh_bnd = msh_eval_boundary_side (msh, iside);
sp_bnd  = sp_eval_boundary_side (space, msh_bnd);
```

The outputs are two boundary structures, as in the previous version of `GeoPDEs`, . For this reason, the other lines of code do not have to be changed.

C.3. Implementation of vectorial basis functions

The changes that we have seen in the previous sections have been also implemented for vector-valued spaces. This means that also in the vectorial case, the old **space** structure has been transformed into a class, and the fields are not stored in memory anymore, but instead they are computed when needed. We will see how the **space** classes are used in the examples of linear elasticity, Stokes equations and Maxwell equations.

C.3.1. Linear elasticity

Let us consider the same example of linear elasticity already described in Section 5.2. The first difference with respect to older versions of the code is that the **msh** structure, and the scalar spaces **spx**, **spy** and **spz** are not structures anymores. Instead, they are objects computed in the same manner that we have seen for the scalar example. Once the scalar space objects have been defined, the vectorial space object is constructed with the command

```
sp = sp_vector_3d (spx, spy, spz, msh);
```

The output is a space object of the class sp_vector_3d, that only contains some of the fields of the old structure. The list of the fields of the vectorial classes can be seen in Table 8.

After constructing the space, the stiffness matrix is assembled with the command

```
mat = op_su_ev_tp (sp, sp, msh, lambda, mu);
```

As we have seen before, the differences are in the name of the operator (_tp), and in the last two arguments lambda and mu, that are now the function handles to compute the values of the coefficients at any given set of points.

We also notice that in the construction of the space it was not necessary to set any option to compute the divergence, since this will not be stored as a field of the **space**. Instead, the divergence is computed inside the operator function, in a similar way to what we have seen for the gradient in the example of Section C.1.

C.3.2. Stokes equations

Apart from the generalizations of Taylor-Hood (TH), Raviart-Thomas (RT) and Nédélec (NDL) elements, the code now includes a new discretization scheme, called subgrid element (SG), that allows for higher regularity than Taylor-Hood. However, since the implementation of this method in `GeoPDEs` is very similar to Taylor-Hood elements, we will not explain it in this paper, and refer to [31] for details.

We have created one single function that is used to compute any of the four discretizations. The function is invoked as follows

```
[spv, spp, P] = sp_bspline_fluid_2d (element_name, ...
                geometry.nurbs.knots, nsub, degree, regularity, msh);
```

where the input arguments are the type of element ('TH', 'SG', 'RT' or 'NDL'), the knot vector of the starting geometry, the number of subdivisions for refinement, the degree and regularity of the pressure space, and a **msh** object, already computed like in the previous examples of this appendix.

The first task of the function is the same for any element type, and it consists on refining the knot vector and computing the scalar **space** object for the discrete pressure space, that is returned as the output argument **spp**. Since it is a scalar space, its construction is analogous to what we have seen in previous sections of this appendix.

The second task is the construction of the vectorial **space** object for the velocity, that is, the output argument **spv**. For this it is necessary to compute the knot vector and the scalar **space** object for each component of the velocity. Since each component is a scalar, they are computed as in previous sections. Then, the vectorial **space** object is constructed considering the right transformation: a component-wise mapping for TH and SG elements, and a Piola mapping for RT and NDL elements. In the first case, the **space** object belongs to

the class sp_vector_2d, and it is constructed as in the linear elasticity example. In the second case, the **space** object belongs to the class sp_vector_2d_piola_transform, and it is constructed with the command

```
spv = sp_vector_2d_piola_transform (sp1, sp2, msh);
```

where **sp1** and **sp2** are the scalar spaces for each component. The fields of the vector **space** classes are those listed in Table 8, independently of the mapping we choose. The difference between the two vectorial classes is that, when computing the shape functions and their derivatives, for instance with **sp_evaluate_col**, they will apply the mapping corresponding to their class.

Finally, the last output argument is the matrix P already mentioned in Section 5.3, that is only used for RT elements.

C.3.3. Maxwell equations

To adapt the example of Section 5.4, the computation of the knot vectors and the degrees for the discrete spaces is not changed. The difference is that now the vectorial space is not constructed in one single call, but it is necessary to first compute the scalar spaces, exactly as we have seen in the first sections of this appendix.

```
sp1 = sp_bspline_3d (knots_u1, degree1, msh);
sp2 = sp_bspline_3d (knots_u2, degree2, msh);
sp3 = sp_bspline_3d (knots_u3, degree3, msh);
```

Then, the vectorial space is constructed as an object of the class sp_vector_3d_curl_transform, with the following command

```
space = sp_vector_3d_curl_transform (sp1, sp2, sp3, msh);
```

Comparing this class with the other vectorial **space** classes, the main difference is that on each boundary, the field **comp_dofs** only gives the degrees of freedom for the tangential functions. Analogously, the function **sp_eval_boundary_side** will only compute the information for the tangential boundary functions.

For the assembly of the matrix, the commands we have to use are the following

```
invmu = @(x, y, z) 1./mu(x, y, z);
stiff_mat = op_curlu_curlv_tp (space, space, msh, invmu);
```

Again, the first input arguments are two **space** objects and one **msh** object, and the last input argument is the function handle to compute the coefficients. The dimension of the problem has been removed from the name of the operator, since the function automatically checks whether the problem is 2D or 3D.

C.4. Adapting your own code to GeoPDEs *2.0.0*

In the previous examples we have explained all the important changes that have to be done to adapt the examples to his new version. They can be summarized in the following list

1. The **msh** and **space** structures have been transformed into classes, and their construction is different.
2. The largest fields (as **shape_function_gradients**) are not stored in memory anymore, but computed when needed with the function **sp_evaluate_col**.
3. Also the information for the boundary is not stored in memory, but computed with the function **sp_eval_boundary_side**.
4. For matrix and vector assembly, the operators include the termination _tp, for tensor product.
5. The last argument in the operators is not the coefficient evaluated at the quadrature points, but instead it is a function handle to evaluate it.
6. Some functions, and in particular for postprocessing, have lost any reference to the dimension in their name.

Finally, it is possible that in your code you were using some of the fields that were precomputed in previous versions, and you may prefer not to change that part. In this case, you can use the functions **msh_precompute** and **sp_precompute**, that allow you to precompute any of the fields of the old structures (except **boundary**). For instance, the line of code

```
space = sp_precompute (space, msh, 'connectivity', true);
```

would give you as the output an object of the **space** class, for which the **connectivity** field is also precomputed.

D. Summary of classes in GeoPDEs 2.0.0

Here we summarize for convenience the fields (properties) and functions (methods) that compose the **msh** and **space** classes in GeoPDEs 2.0.0. The fields and functions for the **msh** class are listed in Tables 5 and 6, respectively. The fields for **space** scalar and vectorial spaces are summarized in Tables 7 and 8, respectively. Finally, the functions for the **space** class (both scalar and vectorial) are given in Table 9.

References

[1] T. J. R. Hughes, J. A. Cottrell, Y. Bazilevs, Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, Comput. Methods Appl. Mech. Engrg. 194 (39-41) (2005) 4135–4195.

[2] T. J. R. Hughes, A. Reali, G. Sangalli, Duality and unified analysis of discrete approximations in structural dynamics and wave propagation: comparison of p-method finite elements with k-method NURBS, Comput. Methods Appl. Mech. Engrg. 197 (49-50) (2008) 4104–4124.

[3] F. Auricchio, L. Beirão da Veiga, A. Buffa, C. Lovadina, A. Reali, G. Sangalli, A fully "locking-free" isogeometric approach for plane linear elasticity problems: a stream function formulation, Comput. Methods Appl. Mech. Engrg. 197 (1-4) (2007) 160–172.

[4] H. Gómez, T. J. R. Hughes, X. Nogueira, V. M. Calo, Isogeometric analysis of the isothermal Navier-Stokes-Korteweg equations, Comput. Methods Appl. Mech. Engrg. 199 (25-28) (2010) 1828 – 1840.

[5] Y. Bazilevs, V. M. Calo, J. A. Cottrell, J. A. Evans, T. J. R. Hughes, S. Lipton, M. A. Scott, T. W. Sederberg, Isogeometric analysis using T-splines, Comput. Methods Appl. Mech. Engrg. 199 (5-8) (2010) 229 – 263.

[6] M. R. Dörfel, B. Jüttler, B. Simeon, Adaptive isogeometric analysis by local h-refinement with T-splines, Comput. Methods Appl. Mech. Engrg. 199 (5-8) (2010) 264 – 275.

[7] P. Costantini, C. Manni, F. Pelosi, M. L. Sampoli, Quasi-interpolation in isogeometric analysis based on generalized B-splines, Comput. Aided Geom. Design. 27 (8) (2010) 656 – 668.

[8] C. Manni, F. Pelosi, M. L. Sampoli, Generalized B-splines as a tool in isogeometric analysis, Comput. Methods Appl. Mech. Engrg. 200 (5-8) (2011) 867–881.

[9] A. Buffa, J. Rivas, G. Sangalli, R. Vázquez, Isogeometric discrete differential forms in three dimensions, SIAM J. Numer. Anal. 49 (2) (2011) 818–844.

[10] A. Buffa, G. Sangalli, R. Vázquez, Isogeometric analysis in electromagnetics: B-splines approximation, Comput. Methods Appl. Mech. Engrg. 199 (17-20) (2010) 1143 – 1152.

[11] A. Buffa, C. de Falco, G. Sangalli, Isogeometric Analysis: stable elements for the 2D Stokes equation, Internat. J. Numer. Methods Fluids. 65 (11-12) (2011) 1407–1422.

[12] J. A. Cottrell, T. J. R. Hughes, Y. Bazilevs, Isogeometric Analysis: toward integration of CAD and FEA, John Wiley & Sons, 2009.

[13] D. J. Benson, Y. Bazilevs, E. De Luycker, M.-C. Hsu, M. Scott, T. J. R. Hughes, T. Belytschko, A generalized finite element formulation for arbitrary basis functions: from isogeometric analysis to XFEM, Internat. J. Numer. Methods Engrg. 83 (6) (2010) 765–785.

[14] M. J. Borden, M. A. Scott, J. A. Evans, T. J. R. Hughes, Isogeometric finite element data structures based on Bézier extraction of NURBS, Internat. J. Numer. Methods Engrg.`doi:10.1002/nme.2968`.

[15] C. de Falco, A. Reali, R. Vázquez, Geopdes web page, `http://geopdes.sourceforge.net` (October 2010).

[16] J. W. Eaton, GNU Octave Manual, Network Theory Limited, 2002.

[17] C. de Boor, A practical guide to splines, revised Edition, Vol. 27 of Applied Mathematical Sciences, Springer-Verlag, New York, 2001.

[18] L. L. Schumaker, Spline functions: basic theory, 3rd Edition, Cambridge Mathematical Library, Cambridge University Press, Cambridge, 2007.

[19] L. Piegl, W. Tiller, The Nurbs Book, Springer-Verlag, New York, 1997.

[20] C. V. Verhoosel, M. Scott, R. de Borst, T. J. R. Hughes, An isogeometric approach to cohesive zone

modeling, Internat. J. Numer. Methods Engrg.http://dx.doi.org/10.1002/nme.3061 doi:10.1002/nme.3061.

[21] J. Cottrell, T. Hughes, A. Reali, Studies of refinement and continuity in isogeometric structural analysis, Comput. Methods Appl. Mech. Engrg. 196 (2007) 4160–4183.

[22] T. Hughes, A. Reali, G. Sangalli, Efficient quadrature for NURBS-based isogeometric analysis, Comput. Methods Appl. Mech. Engrg. 199 (5-8) (2010) 301 – 313.

[23] ParaView, http://www.paraview.org/.

[24] Y. Bazilevs, L. Beirão da Veiga, J. A. Cottrell, T. J. R. Hughes, G. Sangalli, Isogeometric analysis: approximation, stability and error estimates for h-refined meshes, Math. Models Methods Appl. Sci. 16 (7) (2006) 1031–1090.

[25] D. Wang, J. Xuan, An improved NURBS-based isogeometric analysis with enhanced treatment of essential boundary conditions, Comput. Methods Appl. Mech. Engrg. 199 (37-40) (2010) 2425 – 2436.

[26] T. Sederberg, J. Zheng, A. Bakenov, A. Nasri, T-splines and T-NURCCSs, ACM Transactions on Graphics 22 (3) (2003) 477–484.

[27] P. Monk, Finite Element Methods for Maxwell's Equations, Oxford University Press, Oxford, 2003.

[28] D. Boffi, Approximation of eigenvalues in mixed form, discrete compactness property, and application to hp mixed finite elements, Comput. Methods Appl. Mech. Engrg. 196 (37-40) (2007) 3672–3681.

[29] T. Dokken, E. Quak, V. Skytt, Requirements from Isogeometric Analysis for changes in product design ontologies, in: Proceedings of the FOCUS K3D Conference on Semantic 3D Media and Content (INRIA Sophia Antipolis - Méditerranée, 2010), IMATI-CNR, Genova, Italy, 2010, pp. 11–15.

[30] M. Spink, NURBS toolbox, http://www.aria.uklinux.net/nurbs.php3, http://octave.sourceforge.net/nurbs/index.html.

[31] A. Bressan, G. Sangalli, Isogeometric discretizations of the Stokes problem: stability analysis by the macroelement technique, Tech. rep. (2011).

Field name	Type	Dimensions	Description
nel	scalar	1×1	number of elements of the partition
nel_dir	scalar	1×1	number of elements in each parametric direction
nel_col	scalar	1×1	number of elements in one column of the mesh, fixing the element in the first parametric direction
nqn	scalar	1×1	number of quadrature nodes per element
nqn_dir	scalar	1×1	number of quadrature nodes per element in each parametric direction
breaks	cell-array	1×nd	breaks along each parametric direction
qn	cell-array	1×nd	quadrature nodes along each parametric direction
qw	cell-array	1×nd	quadrature weights along each parametric direction
der2	logical	1×1	an option to say whether the second derivative has to be computed. By default it is set to false
map	function handle	1×1	a copy of the **map** handle of the **geometry** structure
map_der	function handle	1×1	a copy of the **map_der** handle of the **geometry** structure
Optional fields			
map_der2	function handle	1×1	a copy of the **map_der2** handle of the **geometry** structure
boundary	struct-array	1×(2 · nd)	an (nd-1)-dimensional **msh** structure for each side of the boundary, with the fields: **side_number**, **nel**, **qn**, **breaks** (2D and 3D), and **nel_dir**, **nqn_dir**, **qn**, **qw** (only 3D)

Table 5: Properties of the **msh** class (version 2.0.0 or later)

Function name	Description
msh_precompute	computes any of the fields of the **msh** structure, given in Table 3, or the whole **msh** structure (except the **boundary** field).
msh_eval_boundary_side	computes the parameterization at the quadrature points of one side of the domain.
msh_evaluate_col	computes the parameterization (and its derivatives) at the quadrature points in one column of the mesh, i.e., fixing the element in the first parametric direction.

Table 6: Methods of the **msh** class (version 2.0.0 or later)

Field name	Type	Dimensions	Description
knots	cell-array	1×nd	knot vectors for the univariate spaces in each parametric direction
degree	Matrix	1×nd	space degree for the univariate spaces in each parametric direction
spu, **spv**, (**spw**)	struct	1×1	univariate **space** structure in each direction
ndof	scalar	1×1	total number of degrees of freedom
ndof_dir	Matrix	1×nd	number degrees of freedom of the univariate spaces
nsh_max	scalar	1×1	maximum number of shape functions per element
nsh_dir	Matrix	1×nel	maximum nuber of shape functions for the univariate spaces
ncomp	scalar	1×1	number of components of the vector field (1 in the scalar case)
constructor	function handle	1×1	function handle to construct the same discrete space with a different **msh** object, useful for postprocessing
Optional fields			
weights	NDArray	ndof_dir(1)×ndof_dir(2) (×ndof_dir(3))	for NURBS space, the weights associated to the control points
boundary	struct array	1×(2 · nd)	an (nd-1)-dimensional **space** structure for each side of the boundary, with the fields: **ndof**, **dofs**, **nsh_max**, **ncomp** (2D and 3D cases) and **ndof_dir**, **nsh_dir** (only 3D case).

Table 7: Properties of the **space** class, scalar case (version 2.0.0 or later)

Field name	Type	Dimensions	Description
sp1, **sp2**, (**sp3**)	space object	1×1	scalar **space** object for each component in the parametric domain
ndof	scalar	1×1	total number of degrees of freedom
ndof_dir	Matrix	nd×nd	number degrees of freedom of the univariate spaces correspoding to each component
comp_dofs	cell-array	1×nd	indices of the degrees of freedom correspoding to each component in the parametric domain
nsh_max	scalar	1×1	maximum number of shape functions per element
ncomp	scalar	1×1	number of components of the vector field (1 in the scalar case)
constructor	function handle	1×1	function handle to construct the same discrete space with a different **msh** object, useful for postprocessing
Optional fields			
boundary	struct array	1×(2 · nd)	an (nd-1)-dimensional **space** structure for each side of the boundary, with the fields: **ndof**, **dofs**, **nsh_max**, **ncomp**, **comp_dofs** (2D and 3D cases) and **ndof_dir**, **nsh_dir** (only 3D case).

Table 8: Properties of the **space** class, vector case (version 2.0.0 or later)

Function name	Description
sp_precompute	computes any of the fields of the **space** structure, given in Table 4, or the whole **space** structure (except the **boundary** field).
sp_eval_boundary_side	evaluate the basis functions on one side of the boundary.
sp_evaluate_col	compute the basis functions (and derivatives) in one column of the mesh (that is, fixing the element in the first parametric direction).
sp_evaluate_col_param	compute the basis functions (and derivatives) in one column of the mesh, before applying the map to the physical domain (**scalar spaces only**).

Table 9: Methods of the **space** class (version 2.0.0 or later)