Consiglio Nazionale delle Ricerche
Istituto di Matematica Applicata
e Tecnologie Informatiche
"Enrico Magenes"

REPORT SERIES

R. Vázquez

A new design for the implementation of isogeometric analysis in Octave and Matlab: GeoPDEs 3.0

16-02

IMATI REPORT Series

Nr. 16-02

April 2016

Editorial Office
Istituto di Matematica Applicata e Tecnologie Informatiche "*E. Magenes*"
Consiglio Nazionale delle Ricerche
Via Ferrata, 5/a
27100 PAVIA (Italy)
Email: reports@imati.cnr.it
http://www.imati.cnr.it

IMATI Report Series Nr. 16-02 – 11th April 2016

A new design for the implementation of isogeometric analysis in Octave and Matlab: GeoPDEs 3.0

R. Vázquez

Abstract.

GeoPDEs (http://rafavzqz.github.io/geopdes) is an Octave/Matlab package for the solution of partial differential equations with isogeometric analysis, first released in 2010. In this work we present in detail the new design of the package, based on the use of Octave and Matlab classes. Compared to previous versions the new design is much clearer, and it is also more efficient in terms of memory consumption and computational time.

A new design for the implementation of isogeometric analysis in Octave and Matlab: GeoPDEs 3.0

R. Vázquez

April 11, 2016

Abstract

`GeoPDEs` (`http://rafavzqz.github.io/geopdes`) is an Octave/Matlab package for the solution of partial differential equations with isogeometric analysis, first released in 2010. In this work we present in detail the new design of the package, based on the use of Octave and Matlab classes. Compared to previous versions the new design is much clearer, and it is also more efficient in terms of memory consumption and computational time.

1 Introduction

`GeoPDEs` [11] is a software suite for the solution of partial differential equations using isogeometric analysis (IGA), a Galerkin discretization method based on splines [10]. Written in Octave [14] and fully compatible with Matlab, the package was first released in 2010 under a GNU/GPL license, with the intention to serve both as a research tool, for fast prototyping of new ideas in IGA, and as a teaching tool, to introduce isogeometric methods to other researchers and students interested in the topic.

In the last five years new IGA software has been released, with at least one more package written in Matlab, focused on computational solid mechanics [23], and several libraries written in C++ or C looking for more efficiency. In particular, the `igatools` library [24] provides a general and dimension independent implementation of IGA; the `G+Smo` library [19] is, up to the author's knowledge, the only existing open-source software for IGA which includes hierarchical splines; and `PetIGA` [8] is a C implementation of isogeometric methods based on PETSc, which has been recently extended to include the curl- and div-conforming spline discretizations [26]. A more detailed list of available software for IGA can be found in [23]. Despite the appearance of these new libraries, `GeoPDEs` remains one of the most successful available packages for IGA[1]. The easiness to learn from an Octave/Matlab[2] code, the detailed documentation (accessible with the `help` command) and the long list of existing examples have surely contributed to this success.

Since `GeoPDEs` is a fundamental tool for the author's own research, the package is under continuous development, and it has evolved during the last years to a more efficient and clearer design, sometimes inspired by the developments in other IGA libraries, and sometimes by the needs and the contributions of `GeoPDEs` users. These changes have helped to maintain the package as a tool suitable for innovative research. As a drawback, they have made the original article [11] obsolete and unsuitable to understand its current design, making necessary a new reference containing an explanation of the new design.

[1] Around 3000 downloads in the last three years.

[2] In the sequel we will mention Octave to refer indistinctly to both Octave and Matlab, unless we explicitly indicate it.

Another important motivation for the preparation of this paper is related to the future presentation of new features and methods that are being implemented within `GeoPDEs`. In particular, the author is currently working, in collaboration with E. M. Garau, on an extension of `GeoPDEs` to include adaptivity based on hierarchical splines [16]. As far as we know, this will be the first open-source software for IGA adaptivity in Octave, and we expect (or strongly hope) that it will have a great impact for the future development of IGA. Obviously, this extension is based on the current design of the code, which makes necessary to provide a valid reference to potential users to understand the design of the package.

The new version of `GeoPDEs` is not a simple modification of the first release of the package, instead, it is a complete redesign of the code. Rather than simply explaining the differences with the first version, in this work we present the complete design of `GeoPDEs` 3.0 in a clean form, with the aim of making this paper a comprehensive reference for the current release of `GeoPDEs` and to give the reader a global overview of the package. The main features of `GeoPDEs` are the following: i) dimension independent implementation, inspired by [24], in such a way that the same code is valid for curves, surfaces and volumes; ii) implementation of vector fields with div-conforming and curl-conforming discretizations, also in multipatch domains; iii) use of Octave classes instead of structures, avoiding the precomputation of fields that caused memory problems in previous versions; iv) new multipatch classes, that automatically manage the strong C^0 gluing of basis functions and the matrix assembly in multipatch geometries in a user-friendly way; v) inclusion of some properties into the classes, allowing to reduce the number of classes with respect to the original structures in previous versions.

The paper is organized as follows: in Section 2 we briefly introduce IGA in an abstract form, mainly to fix the notation, and we apply it to the solution of the Poisson problem. Based on this notation we explain in detail the design of the code in Section 3. We then present in Section 4 the implementation of discrete spaces for the approximation of vector fields, and in Section 5 we detail the new classes for the construction of isogeometric discrete spaces in geometries formed by multiple patches. Finally, in Section 6 we show a comparison of the performance of the current version of `GeoPDEs` with the one presented in [11]. The paper is complemented with several appendices with practical information for users of the software, and a webpage (`http://rafavzqz.github.io/geopdes`) to download the software.

2 The basics of Isogeometric Analysis

In this section we recall the main concepts of IGA, mainly to fix the notation, assuming that the reader has a basic knowledge of the method. The interested reader can find more details about the topic in [10, 2]. We start this section with an abstract framework for IGA, already introduced in [11], and that will serve us later to explain the design of the package. We then present the definitions of B-splines and NURBS, and show how this abstract IGA framework is applied to a simple model problem, the Poisson problem, with a NURBS discretization. For the sake of clarity, in this section we restrict ourselves to the single patch case, that is, the domain is defined as the image of the n-dimensional unit domain. A more complex frmaework for multipatch domains is explained in Section 5.

2.1 Isogeometric Analysis: an abstract framework in a single patch domain

Similar to the finite element method, the goal of IGA is the numerical approximation of the solution of partial differential equations (PDEs) with a Galerkin method[3]. The boundary value problem is written in

[3]The design of `GeoPDEs` is based on elements, and is not well suited for IGA collocation methods.

variational form, and we seek a discrete solution by solving the variational problem in a finite-dimensional space with good approximation properties.

We assume that the domain of our problem is given by a parametrization, that is, we have a *parametric domain* $\widehat{\Omega} = (0,1)^n$, and a *physical domain* $\Omega = \mathbf{F}(\widehat{\Omega}) \subset \mathbb{R}^r$, with $n \leq r$, which is defined through the *parametrization* $\mathbf{F}$. Notice that this parametrization is not required to be defined as a NURBS. The Jacobian of $\mathbf{F}$ is given by an $r \times n$ matrix, that we denote by $J_{\mathbf{F}}$.

Let V be a Hilbert space, to which the solution of the continuous problem belongs, and $V_h \subset V$ a discrete space where we look for an approximate solution. Given a bilinear form $a : V \times V \to \mathbb{R}$ and a function $f \in L^2(\Omega)$, the variational formulation of our discrete problem reads: *Find* $u_h \in V_h$ *such that*

$$a(u_h, v_h) = (f, v_h) \quad \forall v_h \in V_h, \tag{1}$$

where $(\cdot,\cdot)$ represents the L^2-inner product. In a general IGA context the *approximation space* V_h is defined as

$$V_h := \{v_h = \iota_{\mathbf{F}}(\widehat{v}_h),\, \widehat{v}_h \in \widehat{V}_h\} \tag{2}$$

where $\widehat{V}_h$ is a discrete space in the parametric domain $\widehat{\Omega}$, usually a space of splines or NURBS, and $\iota_{\mathbf{F}}$ is a suitable *push-forward* defined from the parametrization $\mathbf{F}$.

Assuming that the parametrization is regular enough, given a *basis* $\widehat{\mathcal{B}} = \{\widehat{v}_j,\, j = 1, \ldots, N_h\}$ of the finite-dimensional space $\widehat{V}_h$, with $N_h = \dim \widehat{V}_h$, we can define a basis of V_h applying the push-forward to these basis functions, that is, $\mathcal{B} = \{v_j = \iota_{\mathbf{F}}(\widehat{v}_j),\, j = 1, \ldots, N_h\}$. As a consequence, we can write our discrete solution as a linear combination of these basis functions, in the form

$$u_h = \sum_{j=1}^{N_h} \alpha_j v_j = \sum_{j=1}^{N_h} \alpha_j \iota_{\mathbf{F}}(\widehat{v}_j). \tag{3}$$

Replacing u_h in (1) with this expression, and testing against every basis function v_i, $i = 1, \ldots, N_h$, 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

$$A_{ij} = a(v_j, v_i), \quad f_i = (f, v_i),$$

respectively.

In general these terms cannot be computed exactly, and have to be approximated numerically using a suitable quadrature rule. In order to describe this rule, let us introduce a partition of the parametric domain $\widehat{\Omega}$ into N_{el} non-overlapping subregions $\widehat{\mathcal{K}}_h := \{\widehat{K}_k\}_{k=1}^{N_{el}}$, that from now we refer to as *elements*. If the parametrization $\mathbf{F}$ is not singular, the image through $\mathbf{F}$ of these elements defines a partition of Ω, denoted by $\mathcal{K}_h := \{K_k\}_{k=1}^{N_{el}}$, with $K_k = \mathbf{F}(\widehat{K}_k)$.

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

$$\widehat{\mathbf{x}}_{l,k} \in \widehat{K}_k,\ \widehat{w}_{l,k} \in \mathbb{R}, \quad l = 1, \ldots, n_k, \quad k = 1, \ldots, N_{el}.$$

After introducing a change of variable, the integral of a generic function ϕ in one element can be approximated as follows

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

where $\mathbf{x}_{l,k} = \mathbf{F}(\widehat{\mathbf{x}}_{l,k})$ are the images of the quadrature nodes in the physical domain, and $w_{l,k} = \widehat{w}_{l,k} J_{\mathbf{F}}(\widehat{\mathbf{x}}_{l,k})$, with $|J_{\mathbf{F}}|$ the measure evaluated at the quadrature points. In general this takes the form $|J_{\mathbf{F}}| = \sqrt{\det(J_{\mathbf{F}}^\top J_{\mathbf{F}})}$, and in the case $n = r$ it reduces to the determinant of the Jacobian (see [12]).

Other formulations For clarity we briefly introduce other formulations that are present in the examples contained in `GeoPDEs`, and in particular they will be necessary for the examples in Section 4.

The first formulation deals with problems written in mixed form, and in particular it will be used for the Stokes problem in Section 4.2. The mixed formulation reads: *Find $u_h \in V_h$ and $p_h \in Q_h$ such that*

$$\begin{aligned} a(u_h, v_h) + b(v_h, p_h) &= (f, v_h) \quad \forall v_h \in V_h, \\ b(u_h, q_h) &= (g, q_h) \quad \forall q_h \in Q_h. \end{aligned} \tag{4}$$

where $g \in L^2(\Omega)$ and Q_h is another discrete space, defined analogously to V_h, but possibly with a different push-forward.

The second formulation regards the solution of eigenvalue problems, and it will be used for the solution of Maxwell model problem in Section 4.3. In this case the formulation is: *Find $u_h \in V_h$ and $\lambda_h \in \mathbb{R}$ such that*

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

Finally, we remark that it is also possible to solve eigenvalue problems with mixed variational formulations. Although not present in this paper, `GeoPDEs` contains examples to solve Maxwell eigenvalue problem with the two mixed variational formulations in [3].

2.2 Splines and NURBS

We give here a brief review of B-splines and NURBS, for details we refer to [25]. B-splines of degree $p \geq 0$ are defined from the knot vector $\Xi := \{\xi_1, \xi_2, \dots, \xi_{N+p+1}\}$ using the well-known Cox-De Boor formula, with N the number of B-spline functions. We denote by $\widehat{B}_{i,p}$ the B-spline basis functions, and by $S_p(\Xi)$, or simply S_p, the space they span:

$$S_p(\Xi) \equiv S_p = \text{span}\{\widehat{B}_{i,p},\ i = 1, \dots, N\}.$$

We also introduce the vector $\{\zeta_1, \dots, \zeta_m\}$ of knots without repetitions, with corresponding multiplicity r_i, in such a way 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 $\sum_{i=1}^m r_i = N + p + 1$. We recall that the number of continuous derivatives at a knot is given by $p - r_i$, and is therefore reduced with the multiplicity. Moreover, each basis function has a local supported contained in $p+1$ knot spans, including the empty ones, and only $p+1$ basis functions do not vanish on each knot span. For simplicity, in the following we assume that the interval of definition of the splines is $[0, 1]$ and that the knot vector is open, i.e., $\zeta_1 = 0$, $\zeta_m = 1$, and $r_1 = r_m = p + 1$. None of these assumptions is a requirement in `GeoPDEs`.

The definition of multivariate B-splines in the parametric domain $[0, 1]^n$ is easily generalized by tensor-product. Given the degrees p_d, the integers N_d and the knot vectors Ξ_d, for $d = 1, \dots, n$, B-spline basis functions are defined by

$$\widehat{B}_{\mathbf{i},\mathbf{p}}(\widehat{\mathbf{x}}) := \widehat{B}_{i_1,p_1}(\widehat{x}_1) \dots \widehat{B}_{i_n,p_n}(\widehat{x}_n),$$

with the multi-index $\mathbf{i} \in \mathcal{I} = \{(i_1, \dots, i_n) : 1 \leq i_d \leq N_d\}$ and the multi-degree $\mathbf{p} = (p_1, \dots, p_n)$. The space spanned by multivariate B-splines is denoted by $S_{\mathbf{p}}(\boldsymbol{\Xi}) \equiv S_{\mathbf{p}}$, with $\boldsymbol{\Xi} = \{\Xi_1, \dots, \Xi_n\}$, but in some situations it will be more convenient to maintain a "verbose" notation $S_{p_1,\dots,p_n}(\Xi_1, \dots, \Xi_n)$, as we will see in Section 4. The dimension of the tensor-product space is $N_h = \Pi_{d=1}^n N_d$. For the implementation it is convenient to order the indices in $\mathcal{I}$ from 1 to N_h, and a common procedure is to associate each $\mathbf{i}$ to the global number given, in the 2D and 3D cases, by

$$i_1 + (i_2 - 1)N_1 \text{ for } n = 2, \quad i_1 + (i_2 - 1)N_1 + (i_3 - 1)N_1 N_2 \text{ for } n = 3. \tag{6}$$

NURBS basis functions are defined from the B-spline basis. In brief, associating to each B-spline function a weight $w_{\mathbf{i}}$, NURBS basis functions are given by

$$\widehat{N}_{\mathbf{i},\mathbf{p}} = \frac{w_{\mathbf{i}} \widehat{B}_{\mathbf{i},\mathbf{p}}}{\sum_{\mathbf{j} \in \mathcal{I}} w_{\mathbf{j}} \widehat{B}_{\mathbf{j},\mathbf{p}}},$$

and $w = \sum_{\mathbf{j} \in \mathcal{I}} w_{\mathbf{j}} \widehat{B}_{\mathbf{j},\mathbf{p}}$ is called the weight function. We denote the space spanned by the NURBS basis functions by $N_{\mathbf{p}}(\boldsymbol{\Xi}; w)$, or simply $N_{\mathbf{p}}$. We notice that several properties (dimension of the space, continuity, local support...) are the same for splines and NURBS spaces, and from the point of view of the implementation the weight only affects the computation of the basis functions and their derivatives.

A NURBS geometry is constructed by associating a control point $\mathbf{C}_{\mathbf{i}} \in \mathbb{R}^r$ to each basis function, which defines the so-called parametrization $\mathbf{F}$, given by

$$\mathbf{x} = \mathbf{F}(\widehat{\mathbf{x}}) := \sum_{\mathbf{j} \in \mathcal{I}} \widehat{N}_{\mathbf{j},\mathbf{p}}(\widehat{\mathbf{x}}) \mathbf{C}_{\mathbf{j}}. \tag{7}$$

This parametrization maps the parametric domain $\widehat{\Omega} = (0, 1)^n$ to a physical domain $\Omega \subset \mathbb{R}^r$, with $n \leq r$.

Finally, we note that the knot vectors Ξ_d define a Cartesian grid in the parametric domain. Mapping the elements of this grid to the physical domain through the parametrization $\mathbf{F}$, we obtain a structured grid in Ω. This grid, which is sometimes called the Bézier mesh, plays in IGA a similar role to the mesh in finite elements.

In the following, we refer to the basis functions indicating the global index (6) and remove the degree from the subindex. That is, B-splines and NURBS basis functions will be denoted by $\widehat{B}_i$ and $\widehat{N}_i$, respectively, with $1 \leq i \leq N_h$.

2.3 A simple model problem: Poisson's problem

The abstract framework of Section 2.1 can be applied to many different problems, such as the Stokes and Maxwell problems that we already mentioned, but also to high order PDEs such as the bilaplacian, that is also present in the `GeoPDEs` examples. However, for explaining the design of `GeoPDEs` it is convenient to use a simple model problem. We now apply the abstract framework to the particular case of Poisson's problem in a domain defined with NURBS, and discretized using an isoparametric approach, that is, using the same discrete space for the geometry and the approximation of the solution.

Let us assume that the n-dimensional computational domain $\Omega \subset \mathbb{R}^r$ is constructed with a NURBS parametrization as in (7), and defined from the NURBS space $N_{\mathbf{p}}(\boldsymbol{\Xi}; w)$. We consider a Poisson's problem in Ω with homogeneous Dirichlet boundary conditions on $\partial\Omega$, for which the equations of the problem read

$$\left\{ \begin{array}{rcll} -\operatorname{div}(\epsilon(\mathbf{x}) \, \mathbf{grad}\, u) & = & f & \text{in } \Omega, \\ u & = & 0 & \text{on } \partial\Omega, \end{array} \right. \tag{8}$$

where $f \in L^2(\Omega)$, and for manifolds **grad** and div represent the surface gradient $\mathbf{grad}_\Gamma$ and its adjoint div_Γ, respectively. The variational formulation of the discretized problem reads: *Find* $u_h \in V_{0,h}$ *such that*

$$\int_\Omega \epsilon(\mathbf{x})\, \mathbf{grad}\, u_h \cdot \mathbf{grad}\, v_h \, \mathrm{d}\mathbf{x} = \int_\Omega f\, v_h \, \mathrm{d}\mathbf{x} \quad \forall v_h \in V_{0,h}. \tag{9}$$

To define the discrete space $V_{0,h}$, we first select in the parametric domain the space $\widehat{V}_h = N_{\mathbf{p}}(\boldsymbol{\Xi}; w)$. Then, the discrete space in the physical domain is defined applying the isoparametric concept, that is,

$$V_h = \{v_h := \widehat{v}_h \circ \mathbf{F}^{-1},\, \widehat{v}_h \in \widehat{V}_h\}, \tag{10}$$

where $\mathbf{F}$ is the NURBS parametrization. Thus, in our abstract framework we are considering the push-forward $\iota_{\mathbf{F}}(\widehat{v}_h) = \widehat{v}_h \circ \mathbf{F}^{-1}$. A basis for V_h is constructed applying this push-forward to the NURBS basis functions, that is, $N_i = \widehat{N}_i \circ \mathbf{F}^{-1}$. Finally, the subspace $V_{0,h} \subset V_h$ is the subspace of functions that vanish on the boundary $\partial\Omega$.

Remark 2.1 *It is possible to use different kinds of functions for the parametrization and the space in the parametric domain, following a non-isoparametric approach. For example, in the definition of* V_h *in* (10) *we can take* $\mathbf{F}$ *defined from the NURBS space* $N_{\mathbf{p}}(\boldsymbol{\Xi}; w)$*, and* $\widehat{V}_h$ *equal to the spline space* $S_{\mathbf{p}}(\Xi)$*.*

Expressing the solution of our problem as a linear combination of the basis functions, as in (3), and testing against each basis function, we can find the discrete solution to our problem as the solution of the linear system

$$\mathbf{A}\boldsymbol{\alpha} = \mathbf{f},$$

where the entries of the stiffness matrix and the right-hand side are, respectively

$$A_{ij} = \int_\Omega \epsilon(\mathbf{x})\, \mathbf{grad}\, N_j \cdot \mathbf{grad}\, N_i \, \mathrm{d}\mathbf{x}, \qquad f_i = \int_\Omega f\, N_i \, \mathrm{d}\mathbf{x}.$$

These integrals are numerically approximated using a suitable quadrature rule, as we have already seen in Section 2.1

$$\begin{aligned} A_{ij} &\simeq \sum_{k=1}^{N_{el}} \sum_{l=1}^{n_k} w_{l,k}\ \epsilon(\mathbf{x}_{l,k})\ \ \mathbf{grad}\, N_j(\mathbf{x}_{l,k}) \cdot \mathbf{grad}\, N_i(\mathbf{x}_{l,k})\ , \\ f_i &\simeq \sum_{k=1}^{N_{el}} \sum_{l=1}^{n_k} w_{l,k}\, f(\mathbf{x}_{l,k})\, N_i(\mathbf{x}_{l,k}). \end{aligned}$$

With the push-forward defined above, and a simple application of the chain rule, the gradient of the functions in the physical domain can be computed in the following way

$$\mathbf{grad}\, N_i = (J_{\mathbf{F}}^+)^\top (\widehat{\mathbf{grad}}\, \widehat{N}_i \circ \mathbf{F}^{-1}),$$

where $\widehat{\mathbf{grad}}$ is the gradient in parametric coordinates, and $J_{\mathbf{F}}^+$ is the Moore-Penrose pseudoinverse, $J_{\mathbf{F}}^+ = (J_{\mathbf{F}}^\top J_{\mathbf{F}})^{-1} J_{\mathbf{F}}^\top$. Notice that when $n = r$ it coincides with the standard inverse.

Finally, it is worth noting that, since the mesh in the parametric domain is a Cartesian grid, the simplest way to define the quadrature rule is by tensorization of one-dimensional quadrature formulae. Indeed, for the kth element in the dth parametric direction, let us define a one-dimensional quadrature

rule of n_k^d points, with quadrature nodes $\widehat{x}_{l,k}^d$ and weights $\widehat{w}_{l,k}^d$, for $1 \leq l \leq n_k^d$. Let us also denote by N_{el}^d the number of elements in the dth direction, in such a way that the total number of elements is $N_{el} = \prod_{d=1}^n N_{el}^d$. Defining a numbering of the elements (and the quadrature points) analogous to (6), the quadrature rule for an element of the Cartesian grid is given by the points and weights

$$\widehat{\mathbf{x}}_{l,k} = (\widehat{x}_{l_1,k_1}^1, \dots, \widehat{x}_{l_n,k_n}^n), \quad \widehat{w}_{l,k} = \prod_{d=1}^n \widehat{w}_{l_d,k_d}, \qquad \text{for } l = 1, \dots, \prod_{d=1}^n n_{k_d}^d.$$

3 The design of GeoPDEs 3.0

In this section we explain in detail the design of the new version of GeoPDEs, which relies on the abstract framework presented in the section above. Since the same framework was also used for the development of version 1, there are many similarities with respect to that version. However, the new design of the package includes several significant changes, that made the code more efficient and versatile. In the first part of this section we explain those changes with the help of a sample code for the solution of a problem already presented in [11]. Then, we enter in more detail into each part of this sample code, putting the focus on the new features of the package.

All the examples in the following sections are taken from the GeoPDEs source code, sometimes with minor modifications to make them more readable. We will mention in the footnotes in which files the code can be found. The source code is available in the git repository `https://github.com/rafavzqz/geopdes`.

3.1 Changes with respect to previous versions: the simple example revisited

We start with a simple example to present the package. Let the computational domain Ω be a quarter of a ring of internal radius $R_i = 1$, and external radius $R_e = 2$, defined with a quadratic NURBS parametrization. In problem (8) we take $\epsilon(\mathbf{x}) = 1$, and the right-hand side

$$f = \frac{(8 - 9\sqrt{x^2+y^2})\sin(2\arctan(y/x))}{x^2+y^2}, \tag{11}$$

which is chosen to obtain the exact solution

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

The full code for solving the variational problem (9) with the data described above is displayed in Listing 1[4].

The design of GeoPDEs consists of two main classes: a *mesh* class, that defines the quadrature rule (more precisely, the points where we want to evaluate the functions), and a *space* class, with the information to compute the basis functions of the discrete space V_h. These two classes are complemented with a *geometry* structure, with the information to evaluate the parametrization $\mathbf{F}$ and its derivatives. The geometry structure, and the objects of the mesh and space classes are constructed in lines 1–5 of Listing 1. The remaining lines of the code perform the general operations to solve the discrete problem: matrix assembly, enforcing of boundary conditions, solution of the linear system, and post-processing.

A quick comparison of Listing 1 with its previous version from [11], that we reproduce in Listing 2, reveals the first main change in the software: all the references to the dimension of the problem, that were

[4] Available in the file `ex_article_15lines.m` of the repository.

```
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_cartesian (knots, qn, qw, geometry);
5  space = sp_nurbs (geometry.nurbs, msh);
6  mat = op_gradu_gradv_tp (space, space, msh);
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:numel(space.boundary)
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*ones(msh.ndim,1), '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 1: Solving the model problem with GeoPDEs 3.0

present in the function names, have disappeared (lines 4, 5 and 15 in Listing 1). This is a consequence of the dimension independent implementation, which makes the same code valid for curves, surfaces and volumes. The dimension now only appears in the definition of the right-hand side function and the exact solution, but these are functions that have to be defined by the user, and are problem dependent. Moreover, this dimension independent implementation is very helpful when dealing with boundary conditions, since the restriction of the space (or the mesh) to the boundary can be defined exactly in the same way as the space in the whole domain, as we will see in Section 3.7.

The second main change is the use of Octave classes for the definition of the quadrature rule and the discrete space. In fact, **msh**, which was defined in [11] as a structure with all the required information for the quadrature rule (see also Appendix C), is now an object of the **msh_cartesian** class. Analogously, **space** was defined as a structure in the previous versions, but it is now an object of the **sp_scalar** class, computed with the constructor **sp_nurbs**. Although this second change is less evident when looking at the sample code, it has required a major redesign of the software, and has very important implications in terms of clarity and efficiency. First of all, in these classes we are only storing one-dimensional information, which has strongly reduced memory consumption with respect to previous versions, as we will see in Section 6. Moreover, we make a more efficient use of the tensor product structure of the spaces, in particular during matrix assembly, which has reduced the computational time.

It is also important to mention that the classes in GeoPDEs were first used in version 2 of the package[5]. In version 3 we have introduced a smarter design of these classes, in particular regarding those that define the discrete spaces. With this new design, the push-forward in (2) becomes a property of the class, in such a way that discrete spaces with different push-forwards can be defined as objects of the same class.

[5] A brief explanation of version 2 was given as an appendix in the technical report distributed with the package.

```
geometry = geo_load ('ring_refined.mat');
knots = geometry.nurbs.knots;
[qn, qw] = msh_set_quad_nodes (knots, msh_gauss_nodes (geometry.nurbs.order));
msh = msh_2d_tensor_product (knots, qn, qw);
msh = msh_push_forward_2d (msh, geometry);
space  = sp_nurbs_2d_phys (geometry.nurbs, msh);
[x, y] = deal (squeeze (msh.geo_map(1,:,:)), squeeze (msh.geo_map(2,:,:)));
mat = op_gradu_gradv (space, space, msh, ones (size (x)));
rhs = op_f_v (space, msh, (8-9*sqrt(x.^2+y.^2)).*sin(2*atan(y./x))./(x.^2+y.^2));
drchlt_dofs = unique ([space.boundary(:).dofs]);
int_dofs = setdiff (1:space.ndof, drchlt_dofs);
u = zeros (space.ndof, 1);
u(int_dofs) = mat(int_dofs, int_dofs) \ rhs(int_dofs);
sp_to_vtk_2d (u, space, geometry, [20 20], 'laplace_solution.vts', 'u')
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 2: Solving the model problem with `GeoPDEs` 1.1

Combined with the dimension independent implementation, this change has allowed us to reduce the number of classes for discrete spaces from the original twelve to only two, one for scalar spaces and another one for vector-valued spaces.

In the following we describe in more detail the sample code of Listing 1. To make this document self-contained, we also give an explanation for those lines that do not present any change with respect to previous versions, but our focus will be on the parts that have changed with the new design: the quadrature rule (Section 3.3), the discrete space (Section 3.4), the matrix assembly (Section 3.5) and the boundary conditions (Section 3.7).

3.2 The parametrization: *geometry* structure

The first step is to define the geometry of our domain. This is done in the first line of Listing 1 using the function **geo_load**, which reads a NURBS geometry created from the NURBS toolbox (see [27] and Appendix D). The output is a **geometry** structure, which contains the information to evaluate the parametrization $\mathbf{F}$, in particular the following fields:

- **map**: a function handle to compute the parametrization $\mathbf{F}$.
- **map_der**: a function handle to compute the Jacobian of the parametrization $\mathbf{F}$.
- **map_der2**: a function handle to compute the second derivatives of $\mathbf{F}$ (optional).

For NURBS geometries, the function handles to evaluate the parametrization use the functions present in the NURBS toolbox, and the **nurbs** structure of that toolbox is also stored as a field. Notice that the **geometry** structure contains the function handles to evaluate the parametrization $\mathbf{F}$, not its evaluation at any point.

Name	Type	Size	Description	Notation
ndim	Scalar	1×1	Dimension of the parametric domain	n
rdim	Scalar	1×1	Dimension of the physical space in which the domain is embedded	r
nel	Scalar	1×1	Number of elements of the mesh	N_{el}
nel_dir	Array	1×ndim	Number of elements in each parametric direction	N_{el}^d
nqn	Scalar	1×1	Number of quadrature points per element	n_k
nqn_dir	Array	1×ndim	Number of quadrature points per element, in each parametric direction	n_k^d
qn	Cell array	1×ndim (nqn_dir×nel_dir)	Coordinates of the quadrature nodes in each parametric direction	$\widehat{x}_{l_d,k_d}^d$
qw	Cell array	1×ndim (nqn_dir×nel_dir)	Quadrature weights in each parametric direction	$\widehat{w}_{l_d,k_d}^d$
breaks	Cell array	1×ndim (nel_dir+1)	Breakpoints that determine the elements, in each parametric direction	—
boundary	**msh_cartesian** array	1×(2*ndim)	A mesh object for each boundary side (see Section 3.7)	—

Table 1: The properties of the **msh_cartesian** class. For cell arrays we give the size of the cell array and, between parentheses, the size of each entry in the cell array.

3.3 The quadrature rule: the *msh_cartesian* class

The second step is the definition of the rule for numerical quadrature, which is done in lines from 2 to 4 of the example in Listing 1. Here we use a Gaussian quadrature rule in the elements of the Cartesian partition determined by the knots of the geometry. We start collecting the knot vectors of our NURBS geometry in line 2, then in line 3 we set the quadrature nodes and weights for the quadrature rule in each parametric direction, selecting a Gauss-Legendre quadrature rule with the number of points per element equal to the order (degree plus one) of our geometry. As in standard finite elements, the number of quadrature points is chosen as the minimum to obtain exact integration (in the parametric domain) of the product of two polynomials of degree p.

Then, in line 4, we call the constructor of the **msh_cartesian** class,

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

which gives as output an object of this class which contains the quadrature rules in each parametric direction, and all the necessary information to compute the tensorized quadrature rule, and to evaluate the parametrization at the quadrature points. The properties/fields of the class are listed in Table 1, together with the size of the variables, a short description of the corresponding mathematical objects, and the notation used in the paper to denote them.

Notice that only univariate information is stored in the properties of the class. For instance, for

the quadrature nodes we do not store the complete list of nodes $\{\mathbf{x}_{l,k}\}$, but only the coordinates along each direction in the parametric domain, that is, $\widehat{x}^d_{l_d,k_d}$. Since one-dimensional information occupies very little space in memory, this has reduced the memory consumption with respect to version 1, where all multivariate information was computed. In turn, it forces us to recompute the multivariate information and to evaluate the parametrization $\mathbf{F}$ at the quadrature points whenever we have to perform some computation in the physical domain.

The **msh_cartesian** class contains several methods/functions that perform the evaluation of the parametrization at the quadrature points for different subsets of the partition. These subsets are always a collection of elements of the mesh:

- **msh_precompute**: compute the quadrature rule, the parametrization $\mathbf{F}$ and its derivatives, for all quadrature points and all elements in the grid.
- **msh_evaluate_col**: compute the same quantities in one "column" of the mesh, that is, fixing the element number in the first parametric direction. Hence, this computes the information for a total of N_{el}/N^1_{el} elements.
- **msh_evaluate_element_list**: compute the same quantities in a given list of elements, without exploiting the Cartesian structure of the grid.

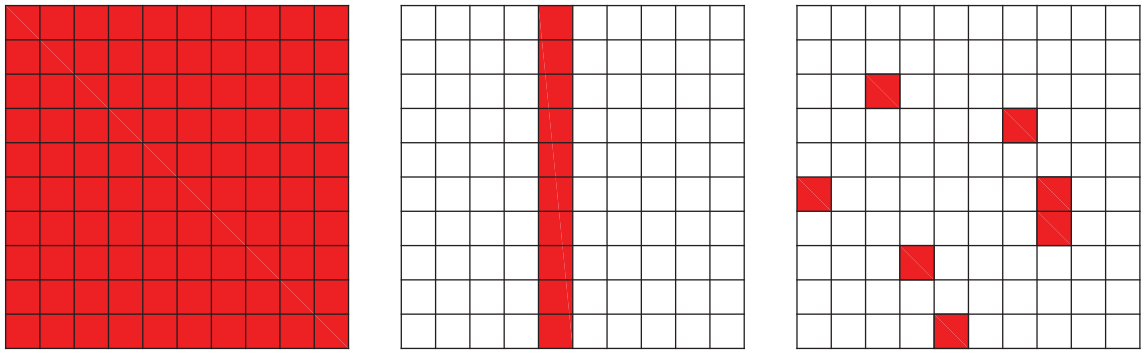
In Figure 1 we give some examples of the elements for which we compute the quantities when calling these functions, for a square domain. The first function is helpful for didactic purposes, as it computes the information for all the elements in the domain, but it is not used in practice due to its high memory consumption, specially in three-dimensional cases. The second computes the information for a full "column" of elements, exploiting the Cartesian structure of the grid. We make use of this for matrix assembly, as we will see in Section 3.5. The last one computes the information for a given list of elements. It can be used to perform computations element by element, and it is also used in our implementation of hierarchical splines to perform computations only on active elements [16].

All these functions give as output a structure with all the relevant information about the parametrization and the quadrature rule in the physical domain, that is, the evaluation of $\mathbf{F}$ and its derivatives at the quadrature points $\widehat{\mathbf{x}}_{l,k}$. This structure, which should not be confused with the object of the **msh_cartesian** class, is basically the same already used in version 1 of `GeoPDEs`, and for the reader convenience we recall its main fields in Appendix C. Moreover, a complete list of the methods in the **msh_cartesian** class is given in Table 9 of Appendix A.

3.4 The discrete space: the *sp_scalar* class

After the construction of the quadrature rule, the next step is to define the discrete space V_h. In our example this is defined as in (10), that is, we take in the parametric domain the same NURBS space used to define the parametrization $\mathbf{F}$, and to obtain a basis in the physical domain the NURBS basis functions are mapped using the same $\mathbf{F}$. The **nurbs** field in the **geometry** structure already contains the necessary information to define the space in the parametric domain: the knot vector, the degree and the weights. The next line of code from Listing 1 constructs the object **space** with all the data to compute the basis functions of the discrete space V_h at the quadrature points of the mesh defined in **msh**.

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



(a) **msh_precompute** (b) **msh_evaluate_col** (c) **msh_evaluate_element_list**

Figure 1: Examples of the elements for which the information is computed with the methods of the **msh_cartesian** class.

Apart from the constructor **sp_nurbs**, `GeoPDEs` includes a constructor **sp_bspline** for tensor-product spline spaces mapped to a NURBS geometry, as explained in Remark 2.1. Actually, due to the similarities of splines and NURBS, a single class **sp_scalar** is used for both, and the two constructors return an object of this class. The differences on the computations of the basis functions (and their derivatives), due to the presence of the weight function in the denominator of NURBS functions, are internally managed by the methods of the class.

As we have done for the mesh, we present in Table 2 the fields/properties of the **sp_scalar** class, along with their sizes, a short description and their notation in the paper. We remark the presence of the property **nsh_max**, which gives the maximum number of non-vanishing functions per element. In most cases this number is equal to $\Pi_{d=1}^{n}(p_d+1)$, for instance when the mesh coincides with the Bézier mesh determined by the knot vectors, but it is also possible to define different grids in which this number is higher, for instance meshes formed by macroelements as in [18].

Similarly to the mesh object, inside the space object only univariate information in the parametric domain is stored, reducing the memory consumption with respect to version 1. In particular, the field **sp_univ** contains the evaluation, at the quadrature points along each parametric direction, of the non-vanishing basis functions of the univariate spline spaces. Notice that, due to the presence of the weight, the multivariate NURBS space is not the tensor product of univariate NURBS spaces. Thus, also for NURBS we have to store univariate spline spaces, plus the coefficients $w_{\mathbf{i}}$ that define the weight function.

To compute the multivariate information in a region of the physical domain, the **sp_scalar** class contains several methods which correspond to analogous methods of the **msh_cartesian** class, and are usually called after them. These methods are the first three listed in Table 10 of Appendix A, and their output is a structure that contains, for each element, the list of non-vanishing basis functions and the evaluation of these functions (and their derivatives) at the quadrature points of the element, among other information. Similar to what we have seen for the mesh, this structure is essentially the same already used in version 1 of

Name	Type	Dimensions	Description	Notation
space_type	String		Either *spline* or *NURBS*	
knots	Cell array	1×ndim	Knot vector in each parametric direction	$\boldsymbol{\Xi}(\Xi_d)$
degree	Matrix	1×ndim	Degree in each parametric direction	$\mathbf{p}(p_d)$
weights	NDArray	ndof_dir	Weight associated to each basis function, only for NURBS spaces	$w_{\mathbf{i}}$
ndof	scalar	1×1	Total number of degrees of freedom	N_h
ndof_dir	Matrix	1×ndim	Number of degrees of freedom in each parametric direction	N_d
nsh_max	Scalar	1×1	Maximum number of non-vanishing functions per element	—
nsh_dir	Matrix	1×ndim	Maximum number of non-vanishing functions per element in each parametric direction	—
sp_univ	Space struct (see Table 14)	1×ndim	Univariate spline spaces in each parametric direction, with basis functions evaluated at points given by **msh.qn**	$S_{p_d}(\Xi_d)$
transform	String		Either *grad-preserving* or *integral-preserving* (see Section 4)	$\iota_{\mathbf{F}}$
constructor	Function handle		Function handle to generate the same discrete space in a different Cartesian grid	—
boundary	**sp_scalar** array	1×(2*ndim)	A space object for each boundary side. Empty for *integral-preserving* transformation (see Section 3.7)	—
Properties for boundary spaces				
dofs	Array	1×ndof	Global numbering of the functions on each boundary. Empty for *integral-preserving* transformation.	—

Table 2: The properties of the **sp_scalar** class.

`GeoPDEs`, but restricted to the selected elements. For the reader convenience, we also recall in Appendix C the fields contained in this structure.

It is important to note that these three methods operate in the same way: first the list of non-vanishing basis functions and their values (and derivatives) are computed in the parametric domain, generating a structure as the one in version 1, with values in the parametric domain; then we apply the push-forward to map to the physical domain, obtaining the structure that is given as the output. `GeoPDEs` contains a function to apply this push-forward, which takes as the input the structure in the parametric domain, and it is valid for the three methods.

Remark 3.1 *An interesting property of the space class is the* **constructor**, *which allows to create a new space object for the same space V_h, but to evaluate the basis functions at the points of a different mesh. This can be useful, for instance, to compare the solution obtained with coarse and fine spaces, evaluating the solution of the coarse space in the fine mesh. In* `GeoPDEs` *it is used to evaluate the computed solution*

in a set of given points, with the function **sp_eval**. *Inside this function we create an auxiliary mesh object with the information of the points where the solution has to be evaluated.*

3.5 Matrix and vector assembly

After the construction of the mesh and space objects, we can assemble the matrix and the right-hand side. Recalling that in our problem $\epsilon(\mathbf{x}) = 1$, and the source function f is given by (11), the assembly is done in lines 6 and 7 of our sample code, namely

```
mat = op_gradu_gradv_tp (space, space, msh);
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 line for the assembly of the stiffness matrix, and the second one for the right-hand side. The last argument in the second function is a *function handle* to compute, at any given point, $f(\mathbf{x})$. By default the diffusion coefficient $\epsilon(\mathbf{x})$ is taken equal to one, but it is also possible give a non-constant value passing a function handle as the last input argument in the first function. We remark that the function to assemble the matrix requires two space objects, one for trial functions and the other for test functions.

The assembly in IGA is usually done as in finite elements[6]: loop over the elements of the mesh, and for each element compute a local elementary matrix, which is then assembled into the global one using the connectivity, that is, an array to connect the local numbering of the non vanishing functions on the element to their global numbering (6). This procedure is the one used in `GeoPDEs`, but to avoid expensive computations inside the element loop, which are not recommended in interpreted languages such as Octave and Matlab, we exploit the tensor product structure of the spaces to precompute some quantities outside the loop. Indeed, the `_tp` ending in the function name stands for *tensor product*, and this function is in fact a method of the **sp_scalar** class (see Table 12 in Appendix B), which only works for tensor product spaces with a quadrature rule defined over a Cartesian grid.

Let us look in more detail at the implementation of the matrix assembly in `GeoPDEs`, which is shown in Listing 3. We start with a loop over the elements in the first parametric direction, and inside the loop we use the functions **msh_evaluate_col** and **sp_evaluate_col** to compute the parametrization and the discrete basis functions in the elements of one "column" of the mesh, that is, fixing the element in the first parametric direction. With the output of these two functions we call the function **op_gradu_gradv** (without the `_tp` ending), which assembles the elementary matrices corresponding to the elements in the "column". The procedure is graphically represented in Figure 2: at each step we compute all the required quantities for the elements in red, and the corresponding elementary matrices add information to the global matrix in the entries marked with a red cross. Notice that there is always some overlap between the contributions computed in previous steps (in blue) and the contributions of the current step (in red), due to functions with support both on red and blue elements. The higher the continuity, the bigger this overlap becomes.

It is worth noting that, as already mentioned in the previous subsections, the output of the functions **msh_evaluate_col** and **sp_evaluate_col** are essentially the same structures used in version 1 of `GeoPDEs`, limited to a certain list of elements, in this case the elements in one "column". Therefore, the assembly of the matrix restricted to one column can be done calling exactly the same function that was already used in version 1 of `GeoPDEs`.

[6] An alternative assembly strategy, based on quadrature points instead of elements, was proposed in [20].

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

Listing 3: Assembly of the stiffness matrix for a tensor product space.

Remark 3.2 *In version 1 of* `GeoPDEs` *all the quantities were precomputed for the elements of the whole mesh, which made the code much clearer for didactic purposes at the cost of great memory consumption. An alternative would be to perform the computations element by element, at the cost of increasing the computational time in Octave and Matlab. The implementation by "columns" gives priority to efficiency over clarity, but we believe that the way to perform the assembly is still simple enough to maintain* `GeoPDEs` *as a valid didactic software.*

Remark 3.3 *The user looking for more efficiency can make several changes to the assembly. First, the assembly of the matrix and the right-hand side should be done simultaneously, taking advantage of the computations in one "column" to assemble both. Second, if the test and trial spaces are the same, the basis functions can be computed only once. Moreover, in this case one could also take advantage of the symmetry of the matrix to compute only one triangular part of the matrix.*

Remark 3.4 *Most of the functions devoted to matrix assembly are given as* `oct`*-files, with the source code written in C++. This improves the efficiency of the code for matrix assembly in Octave (see Section 6). All these functions have a corresponding* `m`*-file version, that can be used both in Octave and Matlab.*

3.6 Solution of the linear system and post-processing

The remaining lines in the sample code of Listing 1 are the ones devoted to the solution of the linear system and post-processing. First, in lines 8 to 12 we identify the degrees of freedom on the boundary, that have to be set to zero, and the internal degrees of freedom. In line 13 we initialize our solution, which in practice sets the homogeneous boundary conditions, and then in line 14 we solve the linear system only for internal degrees of freedom. More details about how to impose boundary conditions will be given in Section 3.7.

After solving the linear system, the computed degrees of freedom determine our discrete solution, as in (3). Once we have computed a solution, the last step is to perform some post-processing to visualize the numerical results and to evaluate them. The last two lines of code are indeed for post-processing: in line 15 we export the solution to a VTK structured data file format, which can be read with `ParaView` [1]. Line 16 serves to evaluate the error in the L^2-norm, provided that the exact solution is known. The

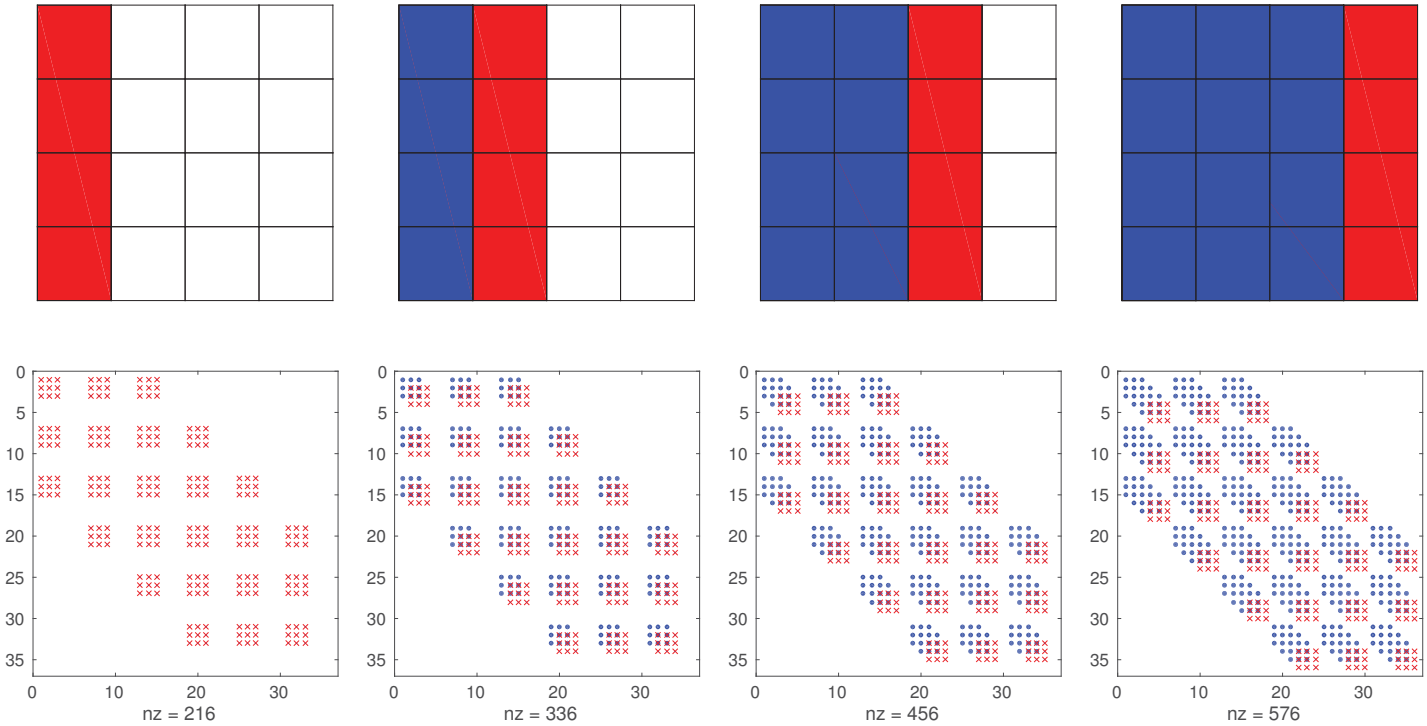


Figure 2: Graphical representation of the matrix assembly in a 4×4 mesh for biquadratic splines, with the number of nonzero entries in the matrix. At each step, the "column" of red elements is assembled, updating the corresponding entries of the matrix, represented with red crosses.

functions used in these two lines are in fact methods of the **sp_scalar** class, which contains other useful methods for the evaluation of the solution. We refer to Table 10 of Appendix A for the complete list.

3.7 Imposition of boundary conditions: the boundary objects

The dimension independent design of the new version of `GeoPDEs` leads to a more elegant implementation of boundary conditions, and in general to an easier computation of boundary features. We explain in this section how the boundary entities can be computed.

3.7.1 Boundary objects

The restriction of the parametrization $\mathbf{F}$ to a boundary side can be seen as a map $\mathbf{F}_\Gamma$ from the $(n-1)$-dimensional parametric domain $\widehat{\Gamma}$, to the boundary side $\Gamma \subset \mathbb{R}^r$ in the physical domain, with $n-1 < r$. An example is given in Figure 3 for $n = r = 3$. Moreover, the Cartesian grid in the parametric domain automatically defines a Cartesian grid on $\widehat{\Gamma}$, which by the map $\mathbf{F}_\Gamma$ defines a structured grid on Γ.

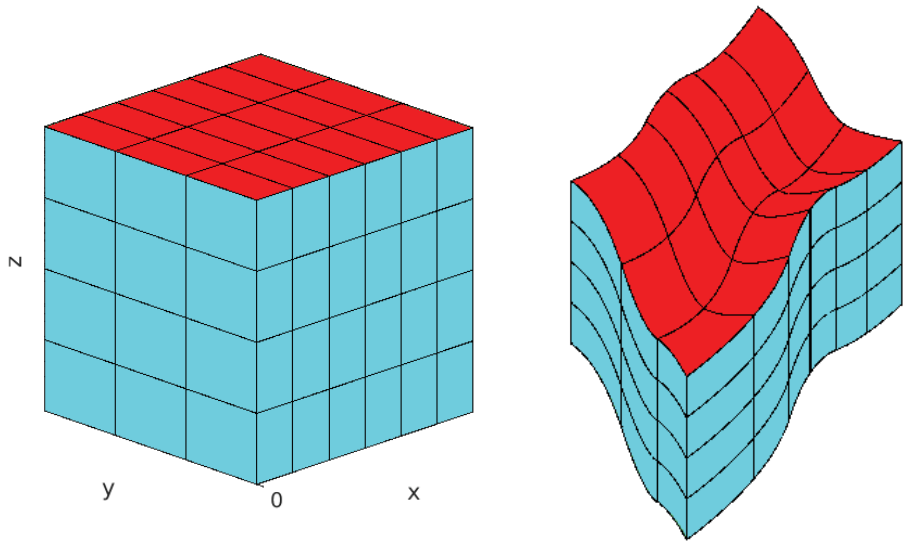


Figure 3: Correspondence between a boundary side in the parametric and the physical domains.

Taking advantage of the dimension independent implementation we can define the structured grid on the boundary, and the corresponding quadrature rule, as an object of the **msh_cartesian** class of Section 3.3. In order to do so, the constructor has to be called with the univariate information (breakpoints, quadrature points and weights) relative to the parametric directions that are relevant to each boundary side. For instance, in the example of Figure 3 only the first and second parametric directions have to be considered for the selected side. In fact, the construction of the boundary objects is already included in `GeoPDEs`, in such a way that the class constructor automatically generates a field called **boundary**, which is an array of objects of the same class, one for each boundary side.

In a similar way, in the case of using an open knot vector, the restriction of the basis functions to the boundary (that is, their trace) gives us a discrete space defined on the boundary. This boundary space is defined in the parametric domain $\widehat{\Gamma}$ by considering the knot vectors in the relevant parametric directions, and then mapped to the physical domain through $\mathbf{F}_\Gamma$. For instance, in the example of Figure 3, since the relevant parametric directions are the first and the second the space is defined with the knot vectors Ξ_1 and Ξ_2. Assuming that the dimension of the trivariate space is $N_h = N_1 N_2 N_3$, the dimension of this bivariate boundary space is $N_1 N_2$.

Analogously to the mesh constructor, the constructor of the **sp_scalar** class automatically generates a **boundary** field, which contains an array of objects of the same space class, one for each boundary side. The boundary space uses a local numbering for the basis functions (from 1 to $N_1 N_2$ in our example), and it is necessary to relate this numbering of the boundary functions with their number in the volumetric domain. This relation is provided in the property **dofs** of the class, which is only used for boundary spaces.

Remark 3.5 *It is important to note that the boundary objects constructed in this way only contain the*

$(n-1)$-dimensional information of the boundary sides. This means that quantities requiring information from the interior of the volumetric domain, such as the normal derivative, cannot be computed using only the boundary structures. In these cases the computation can be done in the volumetric domain fixing one of the parametric coordinates to zero or one (depending on the side). We will see an example in Section 4.2.2.

3.7.2 Imposition of non-homogeneous boundary conditions.

Let us now consider a variation of the model problem (8) with mixed and non-homogeneous boundary conditions. We split the boundary of our NURBS domain in two disjoint parts, $\partial\Omega = \Gamma_N \cup \Gamma_D$, with $\Gamma_N \cap \Gamma_D = \emptyset$, and $\Gamma_D \neq \emptyset$. The equations of the problem are now

$$\begin{cases} -\operatorname{div}(\epsilon(\mathbf{x})\,\mathbf{grad}\,u) = f & \text{in } \Omega, \\ \epsilon(\mathbf{x})\dfrac{\partial u}{\partial \mathbf{n}} = g_N & \text{on } \Gamma_N, \\ u = g_D & \text{on } \Gamma_D, \end{cases}$$

where $\mathbf{n}$ is the unit normal vector exterior to Ω, $f \in L^2(\Omega)$ and $g_N \in L^2(\Gamma_N)$. The variational formulation reads: *Find $u_h = \widetilde{u}_h + u_{0,h}$, with $\widetilde{u}_h|_{\Gamma_D} = g_D$, and $u_{0,h} \in V_{0,h}$ such that*

$$\int_\Omega \epsilon(\mathbf{x})\,\mathbf{grad}\,(\widetilde{u}_h + u_{0,h}) \cdot \mathbf{grad}\, v_h \,\mathrm{d}\mathbf{x} = \int_\Omega f\, v_h \,\mathrm{d}\mathbf{x} + \int_{\Gamma_N} g_N\, v_h \,\mathrm{d}\Gamma \quad \forall v_h \in V_{0,h},$$

where $V_{0,h} \subset V_h$ is the subspace of functions that vanish on Γ_D. As before, the problem is solved expressing the solution as a linear combination of the basis functions, and testing against each basis function.

The imposition of Neumann conditions requires to compute, for functions that do not vanish on the boundary, the integral

$$g_i = \int_{\Gamma_N} g_N N_i \mathrm{d}\Gamma,$$

which is approximated numerically with a quadrature rule on the boundary, and adds a contribution to the right-hand side. Using the boundary entities already present in `GeoPDEs`, the Neumann condition is imposed in the following way[7]:

```
1 for side = nmnn_sides
2  dofs = space.boundary(side).dofs;
3  rhs(dofs) = rhs(dofs) + op_f_v_tp(space.boundary(side),msh.boundary(side),g);
4 end
```

Notice that the function to compute the boundary integral is exactly the same already used to compute the right-hand side in the example of the previous subsections. Using the boundary entities, the integral is only computed for the functions that do not vanish on the boundary, and the field **dofs** is used in order to assemble the computed vector in the correct position.

The imposition of Dirichlet boundary conditions requires the computation of a lifting $\widetilde{u}_h$ such that $\widetilde{u}_h|_{\Gamma_D} = g_D$, which in practice consists on setting the value of the degrees of freedom associated to functions that do not vanish on Γ_D, and then the contribution of $\widetilde{u}_h$ is moved to the right-hand side.

[7] The code for Neumann and Dirichlet conditions is available in the file `ex_article_section_514.m`.

In general, the boundary condition cannot be imposed exactly, and it must be approximated by some numerical method. For convenience, the class **sp_scalar** contains a method to impose the values of the Dirichlet boundary condition through an L^2 projection (see Table 10 in Appendix A), that we explain now.

To set the degrees of freedom that do not vanish on Γ_D with the L^2 projection, it is necessary to solve the problem: Find $\widetilde{u}_h \in V_{h,\Gamma_D}$ such that

$$\int_{\Gamma_D} \widetilde{u}_h v_h \, \mathrm{d}\Gamma = \int_{\Gamma_D} g_D v_h \, \mathrm{d}\Gamma \quad \forall v_h \in V_{h,\Gamma_D},$$

where $V_{h,\Gamma_D} \subset V_h$ is the subspace of functions that do not vanish on the boundary Γ_D. Thus, one has to compute the mass matrix on the Dirichlet sides, and a right-hand side vector for the right term in the equation. This is implemented in `GeoPDEs` with the following lines of code:

```
drchlt_dofs = [];
for side = drchlt_sides
 side_dofs = space.boundary(side).dofs;
 drchlt_dofs = union (drchlt_dofs,side_dofs);
 M(side_dofs,side_dofs) = M(side_dofs,side_dofs) + op_u_v_tp ...
     (space.boundary(side),space.boundary(side),msh.boundary(side),f_one);
 rhs_drchlt(side_dofs) = rhs_drchlt(side_dofs) + ...
     op_f_v_tp(space.boundary(side),msh.boundary(side),g);
end
u(drchlt_dofs) = M(drchlt_dofs,drchlt_dofs) \ rhs_drchlt(drchlt_dofs);
```

Notice that, as we have seen for Neumann conditions, the functions to compute the matrix and the right-hand side are exactly the same used for the volumetric terms, the only difference is that now the input arguments are boundary objects.

After the degrees of freedom for boundary functions have been imposed, their contribution is moved to the right-hand side, and the linear system is solved:

```
int_dofs = setdiff (1:space.ndof, drchlt_dofs);
rhs(int_dofs) = rhs(int_dofs) - mat(int_dofs, drchlt_dofs) * u(drchlt_dofs);
u(int_dofs) = mat(int_dofs, int_dofs) \ rhs(int_dofs);
```

We have focused on the imposition of Dirichlet boundary conditions using the L^2 projection. Other methods to impose strongly the boundary conditions, such as quasi-interpolation or collocation, can be easily implemented with the help of the **boundary** objects.

4 Isogeometric spaces for vector fields

In the model problem of the previous section the solution is a scalar field, and the discrete space was constructed as an object of the **sp_scalar** class. There exist many other problems for which the solution is a vector-valued field, such as the displacement in solid mechanics or the velocity in fluid mechanics. For this kind of problems one can consider an approximating space consisting of vector-valued functions. In `GeoPDEs` these spaces are constructed as objects of a different class, which is called **sp_vector**. As we

will see, one important feature of this class is that it allows to compute in a simple manner the discrete differential forms of [6].

We explain in this section the construction and the main properties and methods of the **sp_vector** class. Then, to show how the class is used in practice, we apply it to solve two simple model problems: Stokes problem, solved with two different discretization schemes, and Maxwell eigenvalue problem.

4.1 The class *sp_vector*

4.1.1 Definition of vector-valued discrete spaces.

We recall that, in the abstract setting, the discrete space in the physical domain is defined as in (2): first one sets the basis functions for the discrete space in the parametric domain, and then these are mapped to the physical domain with a suitable push-forward, which depends on the parametrization $\mathbf{F}$. For the definition of vector-valued spaces in IGA, we first define in the parametric domain a NURBS (or spline) space for each component, taking the basis functions of the vector-valued space null for all components except one. For instance, in the case $\widehat{\Omega} \subset \mathbb{R}^2$ we can take the NURBS space $N_{\mathbf{p}}(\boldsymbol{\Xi};\omega)$ of dimension N for each component, and the vector-valued space of dimension $N_h = 2N$ in the parametric domain is spanned by the basis functions

$$\left\{ \begin{bmatrix} \widehat{N}_i \\ 0 \end{bmatrix} \right\}_{i=1}^{N} \bigcup \left\{ \begin{bmatrix} 0 \\ \widehat{N}_i \end{bmatrix} \right\}_{i=1}^{N}.$$

The simplest and most common way to map the discrete space to the physical domain is to apply the same push-forward in (10) componentwise. We denote this mapping as the *grad-preserving* transformation. This leads to a construction that is purely a Cartesian product of scalar spaces. However, there are also other discretization spaces, that appear in the context of isogeometric differential forms [6], that are inherently vector-valued. For these spaces it is necessary to map the functions to the physical domain with a suitable transformation, either a *curl-preserving* transformation, also known as the contravariant transformation, or a *div-preserving* transformation, also known as Piola or covariant transformation, see [22, Section 3.9]. Thus, the possible transformations are:

$$\begin{aligned} &\mathbf{v}_h = \widehat{\mathbf{v}}_h \circ \mathbf{F}^{-1}, && \text{(grad-preserving transformation)}, \\ &\mathbf{v}_h = (J_{\mathbf{F}}^{+})^{\top}(\widehat{\mathbf{v}}_h \circ \mathbf{F}^{-1}), && \text{(curl-preserving transformation)}, \\ &\mathbf{v}_h = \frac{1}{|J_{\mathbf{F}}|} J_{\mathbf{F}}(\widehat{\mathbf{v}}_h \circ \mathbf{F}^{-1}), && \text{(div-preserving transformation)}, \end{aligned} \tag{12}$$

where $J_{\mathbf{F}}^{+} = (J_{\mathbf{F}}^{\top} J_{\mathbf{F}})^{-1} J_{\mathbf{F}}^{\top}$ is the Moore-Penrose pseudoinverse and $|J_{\mathbf{F}}|$ is the measure, already defined in Sections 2.3 and 2.1, respectively. It is clear from the definition that the grad-preserving transformation acts on each component separately, whereas the curl-preserving and the div-preserving transformations act on all the components of the vector together. In the case $n < r$, these two transformations give as a result a vector in the tangent space of Ω. Notice that in this case, and expressing the vector in terms of Cartesian components, the number of components in the parametric domain and in the physical domain will differ.

Remark 4.1 *It is important to remark that, for scalar-valued spaces, two different transformations are also allowed: the* grad-preserving *transformation, that we already used in the previous section, and the* integral-preserving *transformation, that takes the form:*

$$V_h = \{ v_h = \frac{1}{|J_{\mathbf{F}}|}(\widehat{v}_h \circ \mathbf{F}^{-1}), \widehat{v}_h \in \widehat{V}_h \}.$$

4.1.2 The class *sp_vector*: properties and methods

Vector-valued spaces are defined in `GeoPDEs` as objects of the class **sp_vector**. The construction of these spaces requires first to build the scalar spaces for each component in the parametric domain, which is done as in Section 3.4, and then to set the transformation to the physical domain. An example of how to construct a vector-valued space is shown in the following lines of code[8]:

```
space_scalar = sp_nurbs (geometry.nurbs, msh);
for idim = 1:msh.rdim
  scalar_spaces{idim} = space_scalar;
end
space = sp_vector (scalar_spaces, msh, 'grad-preserving');
```

Assuming that the **geometry** and the **msh** have been built as in the example of Section 3, we start constructing an auxiliary scalar-valued NURBS space, using the same constructor of Section 3, and store a copy of it for each component. Then, we call the constructor of the class **sp_vector**, which takes as input arguments the scalar spaces, the mesh object and the transformation. As we will see in the examples below, one can change the scalar spaces for each component and the transformation to define different discretization schemes.

The output of the constructor is an object of the **sp_vector** class, which has been designed analogously to the space class we have seen in Section 3.4 for the scalar-valued case. For instance, most of the properties of this new class, which are listed in Table 3, are similar to those of Table 2 and adapted to the vector-valued case. In particular, only univariate information is stored in memory, in the **scalar_spaces** field, and the multivariate information is computed on the fly whenever needed.

The similarities of the two classes are even more evident when comparing their methods: all the methods in Table 10 of Appendix A for scalar-valued spaces have their counterpart in the **sp_vector** class, in such a way that the method is invoked in the same manner independently of the class. For instance, the evaluation of the basis functions in one "column", or the computation of the error in the L^2 norm, is done in the same way for scalar-valued or vector-valued spaces. Moreover, the evaluation of the basis functions is also performed in an analogous way for both classes: for each element we evaluate the non-vanishing basis functions in the parametric domain, and then map them to the the physical domain applying the correct transformation. The application of the chosen transformation is automatically managed by the class.

Finally, we remark that there are also some methods that are specific to the **sp_vector** class, and are listed in Table 11.

4.1.3 Boundary spaces

We have seen in Section 3.7 that the restriction of a scalar space to the boundary is described in `GeoPDEs` as a space of the same class, but defined on a parametric domain of lower dimension. To be more precise, the basis functions of the boundary space are obtained by applying the trace operator to the basis functions of the volumetric space. Something similar can also be done for vector-valued spaces, applying the natural trace operators induced by each transformation. Indeed, the trace operators associated to the grad-preserving, the curl-preserving and the div-preserving transformations are, respectively [17,

[8] The code can be found in `solve_linear_elasticity.m`.

Field name	Type	Dimensions	Description
transform	String		One between *grad-preserving*, *curl-preserving* and *div-preserving*
ncomp	Scalar	1×1	Number of components in the physical domain, usually equal to rdim
ncomp_param	Scalar	1×1	Number of components in the parametric domain. Equal to rdim for *grad-preserving*, equal to ndim for *curl-preserving* and *div-preserving*
ndof	Scalar	1×1	Number of degrees of freedom (basis functions)
ndof_dir	Matrix	ncomp_param×ndim	Number of degrees of freedom in each parametric direction, for each component
nsh_max	Scalar	1×1	Maximum number of non-vanishing functions per element
scalar_spaces	Cell array	1×ncomp_param	Spaces for each component in the parametric domain
comp_dofs	Cell array	1×ncomp_param	Numbering of the degrees of freedom corresponding to each component
constructor	Function handle		A function handle to generate the same discrete space in a different Cartesian grid
boundary	**sp_vector** or **sp_scalar** array	1×(2*ndim)	A space object for each boundary, with the information for the traces (see Section 4.1.3).
Properties for boundary spaces			
dofs	Array	1×ndof	Global numbering of the functions on each boundary.

Table 3: The properties of the **sp_vector** class.

Section 2.2],

$$\begin{aligned} &\gamma\mathbf{v} = \mathbf{v}, \\ &\gamma_{\mathbf{t}}\mathbf{v} = \mathbf{n} \times (\mathbf{v} \times \mathbf{n}), \\ &\gamma_{\mathbf{n}}\mathbf{v} = \mathbf{v} \cdot \mathbf{n}. \end{aligned} \tag{13}$$

For the grad-preserving transformation the trace operator is the standard one, and the construction of the boundary space is analogous to the one we have seen in Section 3.7, with the difference that now it is an object of the **sp_vector** class.

For the curl-preserving transformation the trace operator only takes into account the tangential components of the vector. Since the curl-preserving transformation preserves the tangential components, the basis functions such that the tangential trace does not vanish can be easily identified in the parametric domain. The **boundary** field is then defined as a vector-valued space using a curl-preserving transformation, considering only the tangential components in the parametric domain. That is, the boundary space is defined as an object of the same class and with the same transformation.

For the div-preserving transformation the trace operator returns the normal component of the vector, but in the form of a *scalar*. Since the div-preserving operator preserves the normal components, only the functions with non-vanishing normal component in the parametric domain are taken into account. In this case, since the trace operator returns a scalar, the boundary space must be built as an object of the **sp_scalar** class, and the correct transformation to apply is the integral-preserving one. If necessary, one can always multiply the scalar-valued functions of this space by the unitary normal vector to obtain a vector in the normal direction. However, we note that this does not give the restriction of the vector function to the boundary, since the tangential components of the vector are lost.

Finally, we remark that the field **dofs**, that for scalar valued spaces provides the relation between the function numbering on the boundary and in the volumetric domain, plays the same role for vector-valued spaces, with the difference that for curl-conforming (respectively, div-conforming) spaces only the functions with non-vanishing tangential (resp. normal) components are taken into account. This has changed with respect to version 1 of `GeoPDEs`, where for the div-preserving transformation all basis functions with a non-vanishing component on the boundary were computed, in order to implement the boundary conditions for Stokes problems as in [5].

Remark 4.2 *In the case of scalar spaces defined with the integral-preserving transformation, as in Remark 4.1, there is no trace operator, and the boundary field is left empty.*

4.2 Application to the Stokes problem

We now apply the construction of vector-valued spaces to the discretization of simple model problems. As the first problem we have chosen to solve Stokes equations with Dirichlet homogeneous boundary conditions:

$$\begin{cases} -\operatorname{div}(\mu\,\mathbf{grad}\,\mathbf{u}) + \mathbf{grad}\,p = \mathbf{f} & \text{in } \Omega, \\ \operatorname{div}\mathbf{u} = 0 & \text{in } \Omega, \\ \mathbf{u} = \mathbf{0} & \text{on } \partial\Omega, \end{cases}$$

where $\mathbf{u}$ is the fluid velocity, p is the hydrostatic pressure, and μ is the viscosity. We present two different kind of discretizations: the first one is a generalization of Taylor-Hood elements, based on [4], whereas the second is based on div-conforming spline spaces [5, 15], which implies the exact pointwise satisfaction of the incompressibility condition. For this second case the boundary conditions are imposed weakly, which prevents us to use the same variational formulation for both cases.

4.2.1 Taylor-Hood

We start with the generalized version of Taylor-Hood elements. In this case we consider the variational problem: *Find* $\mathbf{u}_h \in V_{0,h}^{\mathrm{TH}}$ *and* $p_h \in Q_h^{\mathrm{TH}}/\mathbb{R}$ *such that*

$$\int_\Omega \mu\,\mathbf{grad}\,\mathbf{u}_h : \mathbf{grad}\,\mathbf{v}_h\,\mathrm{d}\mathbf{x} - \int_\Omega p_h\ \operatorname{div}\mathbf{v}_h\,\mathrm{d}\mathbf{x} = \int_\Omega \mathbf{f}\cdot\mathbf{v}_h\,\mathrm{d}\mathbf{x} \quad \forall \mathbf{v}_h \in V_{0,h}^{\mathrm{TH}},$$
$$\int_\Omega q_h\ \operatorname{div}\mathbf{u}_h\,\mathrm{d}\mathbf{x} = 0 \quad \forall q_h \in Q_h^{\mathrm{TH}}/\mathbb{R}.$$

Let us assume again that the NURBS parametrization $\mathbf{F}$ is constructed from the NURBS space $N_{\mathbf{p}}(\boldsymbol{\Xi}; w)$. We choose the discrete spaces as follows: in the parametric domain, to approximate the pressure we choose the spline space $S_{\mathbf{p}}(\boldsymbol{\Xi})$, with the same degree and knot vector of the NURBS space used to define the geometry; for the velocity space we apply degree elevation once, and define the space $S_{\mathbf{p}+\mathbf{1}}(\widetilde{\boldsymbol{\Xi}})$, where

each $\widetilde{\Xi}_d$ has the same knots as Ξ_d but the multiplicity has been increased by one, that is, it has the same continuity as the pressure space. This spline space is used for each component of the velocity, for which we define the discrete space $\widehat{V}_h^{\mathrm{TH}} = (S_{\mathbf{p}+1}(\widetilde{\boldsymbol{\Xi}}))^r$. Then, the spaces in the physical domain are defined applying the grad-preserving transformation:

$$V_h^{\mathrm{TH}} := \{\mathbf{v}_h = \widehat{\mathbf{v}}_h \circ \mathbf{F}^{-1},\, \widehat{\mathbf{v}}_h \in \widehat{V}_h^{\mathrm{TH}}\}, \quad Q_h^{\mathrm{TH}} := \{q_h = \widehat{q}_h \circ \mathbf{F}^{-1}, \widehat{q}_h \in \widehat{Q}_h^{\mathrm{TH}}\}.$$

The subspace $V_{0,h}^{\mathrm{TH}} \subset V_h^{\mathrm{TH}}$ is the space of discrete functions that vanish on the boundary of Ω. Notice that, as explained in remark 2.1, we are considering spline spaces mapped to the physical domain with a NURBS parametrization, but it is also possible to use NURBS spaces of the same degree and continuity, instead.

For the implementation, the loading of the NURBS geometry, with its degree and knot vector, and the construction of the Cartesian mesh object are done as in Section 3. Assuming that these have been already defined, we show in Listing 4 the construction of the discrete spaces for Stokes problem[9], that is very similar to the one in Section 4.1.2. The space for the pressure is a scalar space with the same degree and knot vector of the NURBS geometry, and is constructed in line 1 as an object of the **sp_scalar** class already seen in Section 3.4, with the difference that the space type is *spline* instead of *NURBS*. For the construction of the velocity space, we apply degree elevation to obtain the new knot vector $\widetilde{\boldsymbol{\Xi}}$ in line 2, and take a copy of the scalar space $S_{\mathbf{p}+\mathbf{1}}(\widetilde{\boldsymbol{\Xi}})$ in line 3 for each component of the velocity. The vector-valued velocity space is generated in line 7 as an object of the **sp_vector** class.

```
1 space_p = sp_bspline (knots, degree, msh);
2 nurbs = nrbdegelev (nurbs, ones(msh.ndim, 1));
3 space_scalar = sp_bspline (nurbs.knots, degree+1, msh);
4 for idim = 1:msh.rdim
5   scalar_spaces{idim} = space_scalar;
6 end
7 space_v = sp_vector (scalar_spaces, msh, 'grad-preserving');
```

Listing 4: Construction of the pressure and velocity spaces for generalized Taylor-Hood.

After the construction of the discrete spaces we can assemble the matrices of the problem. The variational formulation of Stokes problem is a mixed formulation as in (4), therefore we compute the two submatrices of the saddle point problem

```
A = op_gradu_gradv_tp (space_v, space_v, msh, viscosity);
B = op_div_v_q_tp (space_v, space_p, msh);
```

where the last argument in the first line is a function handle to compute the viscosity μ. Notice that, although the basis functions are now vector-valued, the computation of the $\mathbf{A}$ matrix is done exactly as in the model problem of Section 3.

The remaining part of the example (imposition of boundary conditions, solution of the linear system and postprocessing) is analogous to the sample code of Section 3, and we do not detail it here.

[9] The Stokes code is available in the file `solve_stokes.m`. See also `sp_bspline_fluid.m` for the construction of the spaces.

4.2.2 Raviart-Thomas

As a second example we apply the divergence-preserving splines for the same Stokes problem, which leads to a solution where the incompressibility constraint is satisfied exactly. We have seen above that in this case the trace operator only takes into account the normal component of the vector, and therefore only this component can be imposed in a strong form. We follow the approach proposed in [15] to impose the tangential boundary conditions in weak form by means of Nitsche's method, which changes the variational formulation with respect to the previous subsection. The variational formulation in this case is: *Find* $\mathbf{u}_h \in V_{0,h}^{\mathrm{RT}}$ *and* $p_h \in Q_h^{\mathrm{RT}}/\mathbb{R}$ *such that*

$$\begin{aligned}\int_\Omega \mu\, \mathbf{grad}\, \mathbf{u}_h : \mathbf{grad}\, \mathbf{v}_h \,\mathrm{d}\mathbf{x} - \int_{\partial\Omega} (\mu(\mathbf{grad}\, \mathbf{u}_h)\mathbf{n}\cdot\mathbf{v}_h - \mu(\mathbf{grad}\, \mathbf{v}_h)\mathbf{n}\cdot\mathbf{u}_h)\, \mathrm{d}\Gamma + \\ \int_{\partial\Omega} \mu(C_{pen}/h_e)(\mathbf{u}_h\cdot\mathbf{v}_h)\,\mathrm{d}\Gamma - \int_\Omega p_h\ \mathrm{div}\ \mathbf{v}_h\,\mathrm{d}\mathbf{x} = \int_\Omega \mathbf{f}\cdot\mathbf{v}_h\,\mathrm{d}\mathbf{x} \qquad \forall \mathbf{v}_h \in V_{0,h}^{\mathrm{RT}}, \\ \int_\Omega q_h\ \mathrm{div}\ \mathbf{u}_h\,\mathrm{d}\mathbf{x} = 0 \qquad \forall q_h \in Q_h^{\mathrm{RT}}/\mathbb{R},\end{aligned} \tag{14}$$

where C_{pen} is a certain penalty constant, and h_e is the size of the element in the normal direction.

Remark 4.3 *For non-homogeneous boundary conditions, the right-hand side is modified with the terms*

$$-\int_{\partial\Omega} \mu(\mathbf{grad}\, \mathbf{v}_h)\mathbf{n}\cdot\mathbf{u}_b\,\mathrm{d}\Gamma + \int_{\partial\Omega} \mu(C_{pen}/h_e)(\mathbf{v}_h\cdot\mathbf{u}_b)\,\mathrm{d}\Gamma,$$

where $\mathbf{u}_b$ *is the velocity imposed on the boundary.*

Assuming as before that the parametrization is a NURBS constructed from the space $N_{\mathbf{p}}(\boldsymbol{\Xi}; w)$, we define for $d = 1, \ldots, n$ the knot vector $\Xi'_d = \{\xi_{d,2}, \ldots \xi_{2,N_d+p_d}\}$, that is, we are removing the first and last knots. We now use the "verbose" notation, and define in the parametric domain the *spline* spaces, for the two-dimensional case

$$\begin{aligned}\widehat{V}_h^{\mathrm{RT}} &:= S_{p_1,p_2-1}(\Xi_1, \Xi'_2) \times S_{p_1-1,p_2}(\Xi'_1, \Xi_2), \\ \widehat{Q}_h^{\mathrm{RT}} &:= S_{p_1-1,p_2-1}(\Xi'_1, \Xi'_2).\end{aligned} \tag{15}$$

and for the three-dimensional case

$$\begin{aligned}\widehat{V}_h^{\mathrm{RT}} &:= S_{p_1,p_2-1,p_3-1}(\Xi_1, \Xi'_2, \Xi'_3) \times S_{p_1-1,p_2,p_3-1}(\Xi'_1, \Xi_2, \Xi'_3) \times S_{p_1-1,p_2-1,p_3}(\Xi'_1, \Xi'_2, \Xi_3), \\ \widehat{Q}_h^{\mathrm{RT}} &:= S_{p_1-1,p_2-1,p_3-1}(\Xi'_1, \Xi'_2, \Xi'_3),\end{aligned} \tag{16}$$

Notice that, since the multiplicity of the internal knots in Ξ'_d is the same as in Ξ_d, the normal component in the velocity space has the same continuity as the parametrization $\mathbf{F}$, while the continuity of the tangential components is reduced by one. The continuity of the pressure space is also reduced by one with respect to the parametrization. The spaces in the physical domain are then defined applying the divergence-preserving transformation in (12) to the velocity space, and the integral-preserving transformation to the pressure space, namely:

$$V_h^{\mathrm{RT}} = \{\mathbf{v}_h = \frac{1}{|J_{\mathbf{F}}|} J_{\mathbf{F}}(\widehat{\mathbf{v}}_h \circ \mathbf{F}^{-1}), \widehat{\mathbf{v}}_h \in \widehat{V}_h^{\mathrm{RT}}\},\ Q_h^{\mathrm{RT}} = \{q_h = \frac{1}{|J_{\mathbf{F}}|}(\widehat{q}_h \circ \mathbf{F}^{-1}), \widehat{q}_h \in \widehat{Q}_h^{\mathrm{RT}}\}. \tag{17}$$

The space $V_{0,h}^{\mathrm{RT}}$ is defined as the subspace of functions of V_h^{RT} such that the normal component is null on the boundary. As we have seen in Section 4.1.3, the functions with non-null normal component on the boundary are those identified in the **boundary** field.

Remark 4.4 *If the geometry is only C^0, the normal component of the velocity is continuous, but the tangential components are not, which implies that the discrete velocity space is not contained in $(H^1(\Omega))^r$, and the gradient is not correctly defined in Ω. In this case, it was proposed in [15] to modify the variational formulation by introducing a penalty term along these discontinuity lines, in the spirit of discontinuous Galerkin methods. This procedure has been already implemented in* `GeoPDEs` *for multipatch geometries.*

The definition of these discrete spaces in `GeoPDEs` is shown in Listing 5[10]. In the first line we generate the knot vectors and the degrees for each component of the velocity space, with the `GeoPDEs` function **knt_derham**. This function automatically generates the knot vectors (and degrees) for the spaces in (15)-(16), and in general for any of the spline spaces of the De Rham sequence introduced in [6]. Knowing the knot vectors, we construct a scalar space for each component in the parametric domain, and then the vector-valued space of the class **sp_vector**, setting a div-preserving transformation. The construction of the pressure space is done in an analogous way: in line 6 we compute the knot vector using again the function **knt_derham**, and in line 7 we compute a space of the class **sp_scalar**, using the contructor **sp_bspline**, and giving as the last input argument the integral-preserving transformation, which is the one used for the space Q_h^{RT} in (17). After the construction of the spaces, the assembly of the matrices is

```
1 [knots_v, degree_v] = knt_derham (knots, degree, 'Hdiv');
2 for idim = 1:msh.ndim
3   scalar_spaces{idim} = sp_bspline (knots_v{idim}, degree_v{idim}, msh);
4 end
5 space_v = sp_vector (scalar_spaces, msh, 'div-preserving');
6 [knots_p, degree_p] = knt_derham (knots, degree, 'L2');
7 space_p = sp_bspline (knots_p, degree_p, msh, 'integral-preserving');
```

Listing 5: Construction of the velocity and pressure spaces for generalized Raviart-Thomas

identical to the previous case, and it is done exactly with the same lines of code.

For the imposition of Dirichlet boundary conditions, as said above, the normal component is imposed in a strong way. The class **sp_vector** contains a method (see Table 11) to impose the normal component of the vector through an L^2-projection, which is very similar to what we have seen in Section 3.7.2.

The imposition of the tangential component, instead, is done in weak form using Nitsche's method, which adds the boundary terms in (14). These boundary integrals are approximated with a quadrature rule defined on the boundary, for which we can use the boundary entities of Section 3.7. However, the shape functions and their derivatives must be computed using information from the interior of the domain, for which the boundary entities are not sufficient. In this case we define an auxiliary **msh** object in the volumetric domain, which contains the same quadrature points of the boundary rule. This is accomplished setting the parametric coordinate that is constant on the boundary either to zero or one, depending on the boundary side. Then, the basis functions can be computed using this auxiliary **msh** object. In `GeoPDEs` this is done, for a fixed side number iside, with the following lines of code:

```
msh_side = msh_eval_boundary_side (msh, iside);
msh_side_int = msh_boundary_side_from_interior (msh, iside);
sp_side = space.constructor (msh_side_int);
```

[10]The code for the construction of the spaces is available in the file `sp_bspline_fluid.m`.

```
sp_side = sp_precompute (sp_side, msh_side_int, 'value', true, 'gradient', true)
```

We note that, since the **msh** and **space** are built from different geometrical objects (one from the boundary, the other from the whole domain), we cannot use the operators inside the space class to assemble the matrices and vectors. Hence, it is necessary to convert them into structures as in version 1 of `GeoPDEs`, see Appendix C, using the methods **msh_eval_boundary_side** and **sp_precompute**. Once these structures are computed, the assembly of the boundary terms is similar to what we have seen previously for other terms:

```
B = op_gradv_n_u (sp_side, sp_side, msh_side, viscosity);
coeff = Cpen * viscosity ./ msh_side.charlen;
C = op_u_v (sp_side, sp_side, msh_side, coeff);
```

where charlen is the characteristic length of the element in the normal direction.

Remark 4.5 *The class* ***sp_vector*** *contains the method* ***sp_weak_drchlt_bc*** *for the imposition of weak boundary conditions for Stokes problems with the div-conforming transformation, which has been implemented by A. Côrtes for his work [9].*

Remark 4.6 *Besides the generalized Taylor-Hood and Raviart-Thomas splines, in* `GeoPDEs` *we have also implemented a generalization of the Nédélec elements of the second family, as proposed in [5], and also the so-called sub-grid method, in which the velocity space is defined from a refined knot vector with respect to the pressure space [4].*

4.3 Application to Maxwell eigenvalue problem

As the second model problem for vector fields we consider the Maxwell eigenvalue problem, on which we will show some of the advantages of the dimension independent implementation. Let $\Omega \subset \mathbb{R}^r$ be an n-dimensional domain, with $2 \leq n \leq r \leq 3$, and let us introduce the same notation $\mathbf{curl}$ for the standard curl operator for $n = r = 3$, the scalar curl operator for $n = r = 2$, $\mathbf{curl}\,\mathbf{u} := \partial u_2/\partial x_1 - \partial u_1/\partial x_2$, and $\mathbf{curl} = \mathrm{curl}_\Gamma$ the scalar tangential curl operator for $n = 2$, $r = 3$, which is defined as the adjoint of the tangential curl operator $\mathbf{curl}_\Gamma\,\mathbf{u} = (\mathbf{grad}_\Gamma\,\mathbf{u}) \times \mathbf{n}$.

With this notation, the variational formulation of the discrete eigenvalue problem can be written as in (5), and it reads: *Find* $\mathbf{E}_h \in V_{0,h}$ *and* $\lambda_h \in \mathbb{R}$ *such that*

$$\int_\Omega \mathbf{curl}\,\mathbf{E}_h \cdot \mathbf{curl}\,\mathbf{v}_h = \lambda_h \int_\Omega \mathbf{E}_h \cdot \mathbf{v}_h \quad \forall \mathbf{v}_h \in V_{0,h},$$

with $V_{0,h} \subset V_h$ the subspace of functions such that their tangential components vanish on $\partial\Omega$. Following [6], and using the same notation for the knot vectors as in (15)-(16), we define the discrete spaces in the parametric domain

$$\begin{aligned} \widehat{V}_h &= S_{p_1-1,p_2}(\Xi_1', \Xi_2) \times S_{p_1,p_2-1}(\Xi_1, \Xi_2'), && n = 2, \\ \widehat{V}_h &= S_{p_1-1,p_2,p_3}(\Xi_1', \Xi_2, \Xi_3) \times S_{p_1,p_2-1,p_3}(\Xi_1, \Xi_2', \Xi_3) \times S_{p_1,p_2,p_3-1}(\Xi_1, \Xi_2, \Xi_3'), && n = 3. \end{aligned}$$

The spaces in the physical domain are defined using the curl-preserving transformation in (12), namely

$$V_h = \{\mathbf{v}_h = (J_{\mathbf{F}}^+)^\top (\widehat{\mathbf{v}}_h \circ \mathbf{F}^{-1}), \widehat{\mathbf{v}}_h \in \widehat{V}_h\}.$$

As already explained in Section 4.1.3, the curl-preserving transformation also preserves the tangential component of the vector, and the basis functions with non-vanishing tangential component to define the space $V_{0,h}$ can be easily identified in the parametric domain.

For the solution of the problem in GeoPDEs, the **geometry** and the **msh** object with the quadrature rule are generated as in the previous examples. The discrete space is created similarly to the example for Raviart-Thomas spaces, first using the function **knt_derham** to generate the knot vector, then defining the spaces for each component in the parametric domain, and finally defining the space in the physical domain through a curl-preserving transformation, as shown in the following lines of code[11]:

```
[knots_hcurl, degree_hcurl] = knt_derham (knots, degree, 'Hcurl');
for idim = 1:msh.ndim
 scalar_spaces{idim} = sp_bspline (knots_hcurl{idim},degree_hcurl{idim},msh);
end
space = sp_vector (scalar_spaces, msh, 'curl-preserving');
```

The matrices of the discrete problem are computed in a similar fashion to what we have seen in previous examples, using the corresponding operators:

```
stiff_mat = op_curlu_curlv_tp (space, space, msh);
mass_mat  = op_u_v_tp (space, space, msh);
```

Once the matrices are computed, homogeneous tangential boundary conditions are imposed making use of the **dofs** field of the boundary space. Since this field only contains the functions with non-vanishing tangential component, this is identical to what we have seen in lines 8–12 of Listing 1. The eigenvalue problem is then solved using the Octave built-in function **eigs** for sparse matrices. It is important to note that the same code can be used independently of the dimensions n and r. In Figure 4 we show an example of an eigenvector computed for a test surface in $\mathbb{R}^3$, with the surface colored by the vector magnitude.

5 Multipatch geometries

In all the examples we have seen so far the physical domain is constructed with a single patch, that is, it is the image of the parametric domain $\widehat{\Omega} = (0,1)^n$ through a single parametrization. This is clearly not enough to define a complex computational domain, as required in most practical problems. In GeoPDEs we have implemented multipatch domains imposing strongly C^0 continuity in the same spirit of [21], assuming that the patches match conformingly, and enforcing interface continuity between patches in a strong. Although the solution of problems in multipatch domains was already present in version 1 of GeoPDEs, we have incorporated in the current version new multipatch classes, that manage all the data of these domains in an automatic way, and that make the solution of the problem much simpler and clearer for the user.

We also remark that weak enforcement of continuity between patches is also possible, through mortar methods or Lagrange multipliers, for instance. In GeoPDEs we have implemented the imposition of tangential continuity for div-conforming spaces with a penalty method, as already mentioned in Remark 4.4. The implementation of different methods and for other problems and spaces, although not included yet in GeoPDEs, can be obtained using the GeoPDEs classes and structures in a similar way.

[11] The code for Maxwell eigenvalue problem is available in the file `solve_maxwell_eig.m`.

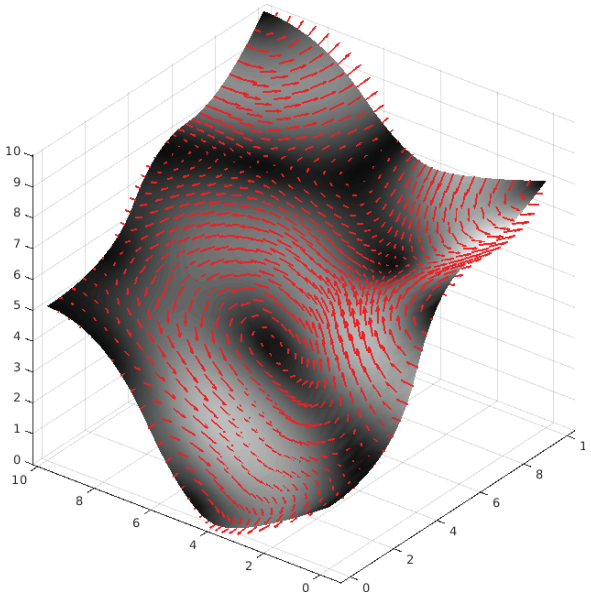


Figure 4: Eighth eigenvector for Maxwell eigenvalue problem in a 3D surface

5.1 The multipatch setting

Before presenting the new classes, we recall the most important ideas of the discretization in multipatch geometries, first for spaces of scalar fields, and then for spaces of vector fields.

5.1.1 Scalar valued spaces

We assume that the physical domain Ω is formed by the union of N_{ptc} patches, in such a way that $\overline{\Omega} = \bigcup_{i=1}^{N_{ptc}} \overline{\Omega}_i$, with $\Omega_i \cap \Omega_j = \emptyset$ for $i \neq j$. Each patch is defined as the image of the same parametric domain through different parametrizations, that is, $\Omega_i = \mathbf{F}^{(i)}(\widehat{\Omega})$. We will always require that the patches are fully matching, in the sense that at the common interface there is a one-to-one correspondence of boundary functions, i.e., there are no hanging nodes or T-junctions. More rigorously, let $V_h^{(i)}$ the discrete space in the patch Ω_i, and let $\mathcal{B}^{(i)}$ be the corresponding basis. Then, following [21], if $\Gamma_{ij} = \partial\Omega_i \cap \partial\Omega_j \neq \emptyset$ we require that

I) Γ_{ij} is either a vertex, or the image of a full edge, or the image of a full face of $\widehat{\Omega}$, for both $\mathbf{F}^{(i)}$ and $\mathbf{F}^{(j)}$;

II) For any basis function $\beta^{(i)} \in \mathcal{B}^{(i)}$ there exists a function $\beta^{(j)} \in \mathcal{B}^{(j)}$ such that $\beta^{(i)}|_{\Gamma_{ij}} = \beta^{(j)}|_{\Gamma_{ij}}$.

Recalling that boundary entities can be defined as in Section 3.7, for splines and NURBS geometries these conditions imply that the control points defining the interface coincide for the two patches (although they are not necessarily on the interface), and the knot vectors relevant to the interface are the same,

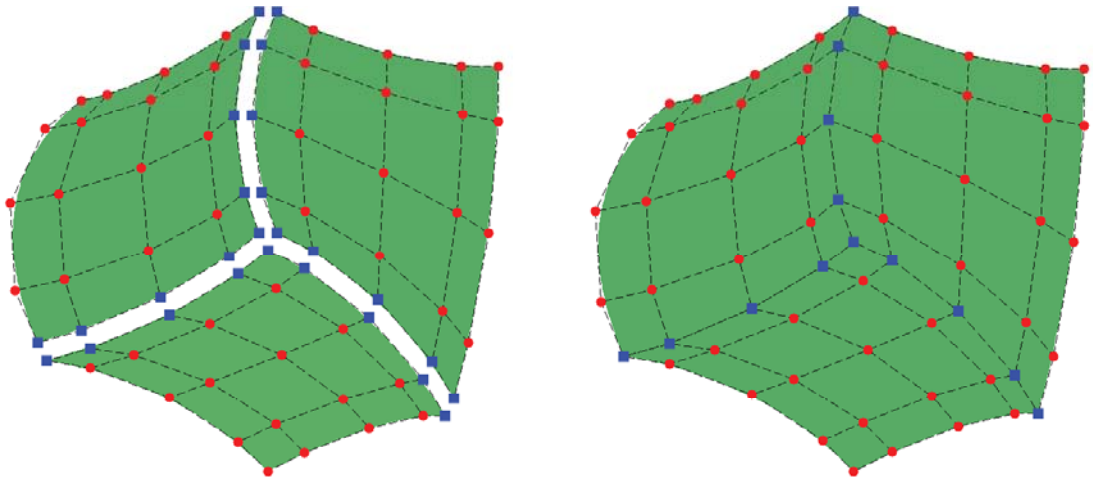
including knot repetitions, up to an affine relation $\mathbf{G}$. Moreover, this affine mapping also relates the boundary parametrizations, that is, $\mathbf{F}^{(j)}_{\Gamma_{ij}} = \mathbf{F}^{(i)}_{\Gamma_{ij}} \circ \mathbf{G}$.

We define the discrete space in the multipatch domain as the space of continuous functions such that their restriction restricted to any patch belongs to the same spaces $V_h^{(i)}$ we have used until now, that is,

$$V_h = \{v \in C^0(\Omega) : v|_{\Omega_i} \in V_h^{(i)}, \quad i = 1, \dots, N_{ptc}\}.$$

The continuity condition ensures that the discrete space is a subspace of $H^1(\Omega)$.

In order to impose the continuity at the interfaces we have to define suitable basis functions. A basis is easily built setting each couple of matching functions in condition II) above as the same basis function in the multipatch domain. We refer to [2, Section 3.2] for the details. In practice, this requires to define a new indexing (or numbering) of the basis functions, setting the same index for the matching functions, and to relate the local indices on each patch to the global index on the multipatch domain. An example is shown in Figure 5, where the square control points are associated to basis functions that coincide on the interfaces. The discrete spaces $V_h^{(i)}$ on each patch are generated by 25 basis functions, whereas the multipatch space generated after merging is generated by 61 basis functions. It is then necessary to relate the local indexing inside each patch (25 indices) to the global indexing (61 indices). We notice that this procedure is analogous to setting the connectivity in the finite element method, where one uses a local numbering for each element, and a global one for the whole discrete space. A more detailed example is given in Example 5.1 below.



(a) Control mesh of the three separate patches

(b) Control mesh of the multipatch domain

Figure 5: Generation of a multipatch domain with conforming meshes. The blue square control points are associated to basis functions that coincide on the interface.

5.1.2 Vector valued spaces

The condition II) above states that basis functions from different patches must coincide when restricted to the interface, or more precisely, that their traces must coincide. The situation for vector valued spaces

is similar, but there are two additional things that must be taken into account.

First, the matching condition on the interface is not required for all the components of the basis functions, but only for those given by the trace operators (13). That is, for the grad-conforming spaces all the components have have to be glued; 2for the curl-conforming spaces only the tangential components are taken into account, while the normal component is left free; for div-conforming spaces it is the normal component to be taken into account, and the tangential components are left free.

Second, since the functions are vector-valued, the basis functions coming from each patch may coincide on the interface *except for the sign.* Similar to what is done for edge (and face) finite elements, to enforce continuity it is then necessary to give an orientation to these basis functions, that is, to multiply them either by one or minus one. An example for curl-conforming spaces is shown in Figure 6: the basis functions on the interface coming from each patch, indicated by the purple arrows, have different orientations before merging; the orientation of the functions of the upper patch has to be corrected in order to obtain tangential continuity. The orientation can be set using the information on how the two patches match, from which we can infer the affine transformation $\mathbf{G}$, although this is never computed in practice. For div-conforming spaces the situation is similar, as it can happen, for instance, that the two functions with non-vanishing normal component point on different directions. Assuming that every patch has the same orientation[12], the information to set the orientation of the basis functions comes from the position of the interface on each patch. It is also important to remark that, for curl-conforming spaces, since the basis functions are associated to the edges, it may be necessary to glue together functions coming from more than two patches (for instance, the functions on the reentrant edge of the thick L-shaped domain), and the same orientation has to be given to all of them. This situation never happens for div-conforming spaces, since the basis functions are associated to the faces.

Finally, we remark that the conditions for the curl-conforming spaces (respectively, div-conforming) ensure tangential (resp. normal) continuity of the basis functions between patches, which implies that the discrete spaces are contained in $\mathbf{H}(\mathrm{curl};\Omega)$ (resp. $\mathbf{H}(\mathrm{div};\Omega)$). We refer to [7] for a more detailed discussion on vector-valued spaces for multipatch domains.

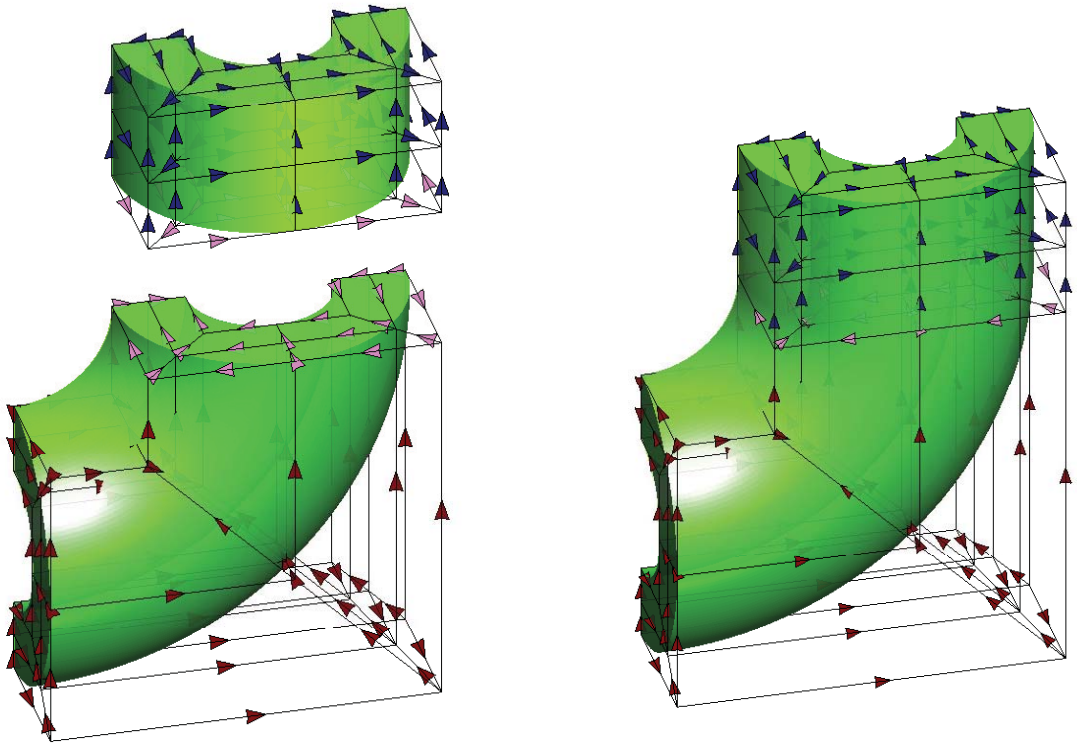
5.2 Classes for multipatch objects: *msh_multipatch* and *sp_multipatch*

For the implementation of multipatch domains in `GeoPDEs` we have created two new classes, one for the mesh and one for the space, with all the information needed to generate and evaluate the basis functions in the multipatch domain. The construction of objects of these classes only requires the corresponding objects (mesh and space) for each patch, and some additional information about the interfaces and the boundary of the domain, that we now detail.

We store in a structure called **boundaries** the information about which sides of each patch are in fact on the boundary of the domain Ω. In most situations, these are the sides that do not belong to any interface. The structure **interfaces** details, for each interface, the involved patches and their local sides, and their relative orientation to understand how they are connected (see [13] for details). The structure **boundary_interfaces** contains similar information, but for boundary entities (more details are given in the subsection below). The interested reader can find more details about these structures in the help of the function **nrbmultipatch** of the NURBS toolbox, that can be used to automatically generate these structures.

Once we have the structures mentioned above, and assuming that we have constructed a mesh and

[12]In the case $n = r$, this means that the determinant of the Jacobian of $\mathbf{F}$ is always positive (or always negative). For orientable manifolds, the orientation of each paetch should be the same as that of the manifold.



(a) Representation of the basis functions of the two patches.

(b) Representation of the basis functions of the multi-patch domain after merging.

Figure 6: Implementing continuity for curl-conforming spaces on a two-patch domain. The orientation of the basis functions associated to the interface edges (purple arrows) is chosen as that of the lower patch. The orientation for basis functions not on the interface (blue and red arrows) remains unchanged after merging.

a space (either scalar or vector valued) object for each patch, stored in the cell arrays **local_meshes** and **local_spaces**, we can construct the multipatch objects with the two following commands[13]:

```
msh = msh_multipatch (local_meshes, boundaries);
space = sp_multipatch (local_spaces, msh, interfaces, boundary_interfaces);
```

The first line of code generates an object of the class **msh_multipatch**, with the information to evaluate the quadrature rule in the unstructured mesh of the multipatch domain. Among the properties of the class, which are listed in Table 4, we have the number of patches, the total number of elements, and the meshes associated to each patch, which are objects of the class **msh_cartesian**. Notice that, since we store the meshes for each patch, some of the information may be redundant.

[13] Available in the file `solve_laplace_mp.m`. See also the `multipatch` folder of the repository.

Field name	Type	Dimensions	Description
npatch	Scalar	1×1	Number of patches of the domain
ndim	Scalar	1×1	Dimension of the parametric domain
rdim	Scalar	1×1	Dimension of the physical space in which the domain is embedded
nel	Scalar	1×1	Total number of elements
nel_per_patch	Scalar	1×npatch	Number of elements on each patch
msh_patch	Cell array of **msh_cartesian**	1×npatch	A Cartesian grid for each patch
boundary	**msh_multipatch** object	1×1	A mesh object for the whole boundary
Properties only for boundary objects			
boundaries	Struct array		Sides that belong to the boundary of the domain, grouped by boundary conditions
patch_numbers	Array	1×npatch	For each boundary patch, the volumetric patch to which it belongs
side_numbers	Array	1×npatch	For each boundary patch, the side number occupied in the volumetric patch

Table 4: The properties of the **msh_multipatch** class

The second line constructs an object of the class **sp_multipatch**, with all the necessary information to compute the discrete basis functions in the multipatch domain. The properties of this class are listed in Table 5. Analogously to the mesh, we store the total number of basis functions and the space objects for each patch. In addition, the class contains the property **gnum**, a cell array that relates the local numbering of the basis functions in the patch to their global numbering, see Example 5.1 below for more details. Moreover, the property **dofs_ornt**, which is only used for vector-valued spaces, relates the orientation of the basis function on each patch (before merging) with the global orientation (after merging), as we have seen in Figure 6.

The advantage of these two classes is that they can be used in the same way as the mesh and space classes defined in Section 3. In fact, the methods of Tables 9-11 marked with an asterisk have their counterpart in the multipatch class. All these methods work in a similar way: we perform a loop in the number of patches, and inside the loop we compute the required local information on the patch, usually with the method for the single patch, and add this information to the global one for the multipatch domain. This same procedure is also used for matrix assembly. For instance, a simple way to compute the stiffness matrix in a multipatch domain is the following[14]:

```
for iptc = 1:space.npatch
  local_msh = msh.msh_patch{iptc}
  local_space = space.sp_patch{iptc};
  A_loc = op_gradu_gradv_tp (local_space, local_space, local_msh);
  A(gnum{iptc},gnum{iptc}) = A(gnum{iptc},gnum{iptc}) + A_loc;
```

[14]See the file `op_gradu_gradv_mp.m` inside the **sp_multipatch** class folder.

Field name	Type	Dimensions	Description
npatch	Scalar	1×1	Number of patches of the domain
ncomp	Scalar	1×1	Number of components in the physical domain, usually equal to rdim
ndof	Scalar	1×1	Total number of degrees of freedom in the multipatch domain, after patch gluing
ndof_per_patch	Array	1×npatch	Number of degrees of freedom that do not vanish on each patch
sp_patch	Cell array	1×npatch	Spaces on each patch, either **sp_scalar** or **sp_vector**
transform	String		One between *grad-preserving*, *curl-preserving*, *div-preserving* and *integral-preserving*
interfaces	Struct array		For each interface, the two involved patches, the corresponding sides on the patch, and information on how they coincide
gnum	Cell array	1×npatch	Global numbering of the degrees of freedom after gluing the interfaces
dofs_ornt	Cell array	1×npatch	Orientation of the basis functions on the patch with respect to the global orientation. For *curl-preserving* and *div-preserving* spaces
constructor	Function handle		A function handle to generate the same discrete space in a different multipatch grid
boundary	**sp_multipatch** object	1×1	A space object for the boundary, with the information for the traces (see Section 5.2.1).
Properties only for boundary spaces			
dofs	Array	1×ndof	Global numbering of the functions on each boundary.
boundary_orientation	Array	1×ndof	Only for boundary spaces with *curl-preserving* transformation, it identifies the orientation of the basis functions on the boundary space with respect to the volumetric functions.

Table 5: The properties of the **sp_multipatch** class

```
end
```

For each patch we compute a local stiffness matrix, **A_loc**, with the dimension of the local space defined on that patch, using the same function that we have seen in previous examples. The local matrix is then assembled into the global one using the information of the global numbering stored in **gnum**. Actually, this is analogous to the assembly of the matrix in the finite element method, with the patches playing

the role of elements, and **gnum** being the connectivity.

We notice that the class **sp_multipatch** already contains several methods, listed in Table 12, to assemble different matrices and vectors. To improve the efficiency of the code, the implementation used in these methods is different from the one seen above, although the same idea is always applied.

5.2.1 Boundary entities

Similar to what we have seen in Section 3.7, where each boundary side of Ω was defined as an $(n-1)$-dimensional domain, also the boundary of a multipatch domain can be defined as an $(n-1)$-dimensional multipatch domain, formed by several curves or surfaces, which are not necessarily connected. For instance, the boundary of the domain in Figure 5 is a one-dimensional domain formed by six curves. Therefore, we can also restrict the elements and the basis functions to the boundary, to obtain a mesh and a discrete space that can be described as in the multipatch setting.

Making use of the dimension independent implementation, and analogously to the definition of boundary entities inside the mesh and space classes in Section 3.7, the multipatch classes also include a property called **boundary**, which defines an object of the same class, either **msh_multipatch** or **sp_multipatch**, with all the information to compute the quadrature points and to evaluate the basis functions on the boundary.

Besides the usual information for multipatch domains, for multipatch boundary entities we also require the information to relate them with the volumetric multipatch domain. In particular, in the class **msh_multipatch** we have included two properties, **patch_numbers** and **side_numbers**, that indicate, for each patch on the boundary, its relative position in the volumetric domain. Moreover, in the class **sp_multipatch**, the property **dofs** plays the same role already seen in Section 3.7, relating the numbering on the boundary (global in the multipatch sense, but local to the boundary) with the global numbering in the multipatch domain. For curl-conforming spaces, it is also necessary to define a similar property for the orientation, since for some basis functions the orientation chosen in the boundary space may not coincide with the one of the volumetric space. We also remark that, for div-conforming spaces, the boundary field representing the normal trace is mapped with the integral-preserving transformation, and it is discontinuous between patches.

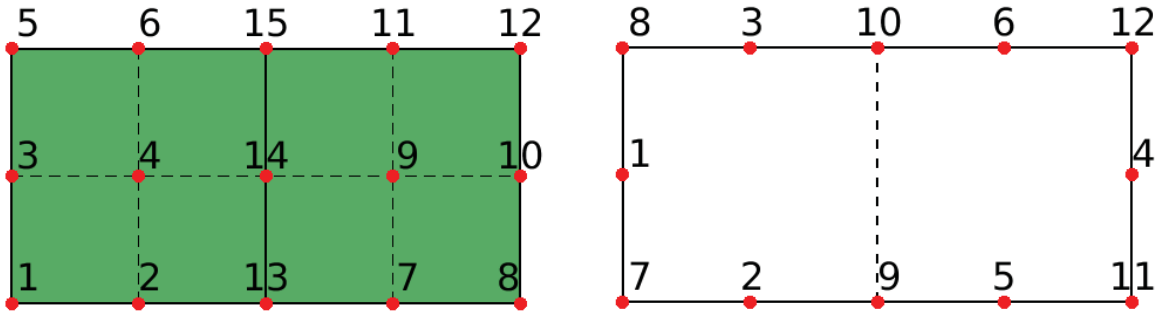
We now show, in a very simple geometry formed by two patches, an example of how the numbering works.

Remark 5.1 *In* `GeoPDEs` *the boundary sides are numbered in the parametric domain, following the ordering* $\widehat{x}=0$, $\widehat{x}=1$, $\widehat{y}=0$, $\widehat{y}=1\ldots$

Example 5.1 *The example is the one shown in Figure 7, with* $\Omega_1=(0,1)\times(0,1)$ *and* $\Omega_2=(1,2)\times(0,1)$. *The interface is given by the second side of* Ω_1 *and the first side of* Ω_2, *and coinciding functions on this interface are given the same global number. According to Figure 7(a), the numbering stored in the property* ***gnum*** *is [1 2 13 3 4 14 5 6 15] for the first patch, and [13 7 8 14 9 10 15 11 12] for the second patch.*

Regarding the boundary entities, the boundary is formed by six patches: the first, third and fourth sides of Ω_1, *and the second, third and fourth sides of* Ω_2. *This information is contained in* ***patch_numbers*** *([1 1 1 2 2 2]) and* ***side_numbers*** *([1 3 4 2 3 4]). Moreover, there are twelve basis functions on the boundary space. Following the ordering of the patches we have just seen, the numbering stored in* ***gnum*** *for the six boundary patches (see Figure 7(b)) is [7 1 8], [7 2 9], [8 3 10], [11 4 12], [9 5 11] and [10 6 12], respectively. The relation of the boundary numbering with the "volumetric" one, stored in the property* ***dofs****, is [3 2 6 10 7 11 1 5 13 15 8 12].*

The source code for this example is available in the file `ex_articlev3_example51.m` *of the* `GeoPDEs` *repository.*



(a) Numbering of functions in a multipatch geometry

(b) Numbering of boundary functions

Figure 7: Example of the numbering of basis functions for a simple multipatch geometry

6 Comparison of the two versions in terms of performance

In this last section we show a comparison of the performance of versions 3.0 and 1.1 of `GeoPDEs`, in terms of memory consumption and computational time. To carry out this comparison we have chosen a simple numerical test: the domain is the unit cube, discretized with splines of degree $p = 1, \dots, 4$ and regularity C^{p-1}. Starting from a coarse mesh of eight elements (two elements in each parametric direction), we run the tests for a family of uniformly refined meshes, until having 64 elements in each parametric direction. The number of quadrature points per element is chosen to be equal to $(p+1)^3$. Since the linear system solver is identical for both versions, the tests only cover the cost of generating the `GeoPDEs` objects or structures, and the assembly of the stiffness matrix and the right-hand side, that is, the analogous to lines 1-7 in Listing 1.

To run the tests we used a laptop Dell Precision M4800, with four Intel Core i5-4200M @2.50 GHz processors and 15.6 GiB of RAM memory, running Ubuntu version 14.04 LTS. The tests were run in version 4.0 of Octave, with version 1.3.11 of the NURBS package installed.

In Table 6 we compare the maximum memory usage, computed with the built-in function **getrusage**. As already explained in Section 3.5, in version 1 of `GeoPDEs` all the fields required to assemble the matrix were precomputed, causing a high memory consumption. In this new version only univariate information is stored in memory, and the assembly of the matrix is performed by "columns", which has strongly reduced the memory usage, allowing to solve problems in much finer meshes. The results also show that, in both versions, the required memory highly depends on the degree. This is expected, since the number of quadrature points and non-vanishing functions per element, and the number of non-zero entries in the matrix, increase with the degree.

As already mentioned in Remark 3.4, the matrix assembly can be performed either with compiled `oct`-files, for which the source code is written in C++, or with `m`-files. We show in Tables 7 and 8 the computational time for the same tests when using `oct`-files and `m`-files, respectively. All the numerical tests were run five times, and the mean computational time is shown.

Regarding the results for `oct`-files, since matrix assembly is the most consuming part, and its implementation for `oct`-files has not changed, the computational times are similar for both versions. Actually, in general the computational time is lower for version 3, despite the fact that the computations for each

	Version 1.1				Version 3.0			
Mesh size	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 1$	$p = 2$	$p = 3$	$p = 4$
1/2	48.78	53.21	51.84	61.94	50.33	53.01	56.66	67.81
1/4	49.20	57.60	79.24	158.7	50.48	55.64	70.82	123.5
1/8	54.63	93.15	281.2	857.3	51.75	65.06	126.7	336.0
1/16	82.99	389.5	1768.9	6497.3	59.05	114.7	373.6	1154.6
1/32	327.0	2586.7	13764	OoM	87.89	319.0	1003.9	3170.6
1/64	2135.7	OoM	OoM	OoM	314.5	1274.2	4174.6	13469

Table 6: Maximum memory usage in MBs. (OoM: Out of Memory).

	Version 1.1				Version 3.0			
Mesh size	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 1$	$p = 2$	$p = 3$	$p = 4$
1/2	0.1630	0.2189	0.3248	0.5112	0.2113	0.2262	0.2477	0.3664
1/4	0.2936	0.5132	0.9423	1.918	0.3692	0.4064	0.5989	1.552
1/8	0.7974	1.726	3.967	10.90	0.7077	1.015	2.556	10.03
1/16	2.858	7.330	20.83	71.65	1.656	3.886	16.09	76.69
1/32	11.61	34.72	125.3	——	5.863	22.61	123.8	614.2
1/64	51.32	——	——	——	33.29	174.6	991.9	4867

Table 7: Computational time (in seconds) using `oct`-files

"column" have to be done twice, one for the matrix and one for the right-hand side vector. This is a consequence of several small changes in different parts of the code, that have increased its efficiency.

Concerning the results for `m`-files in Table 8, they show a huge time reduction for the new version, specially for splines of high degree. Moreover, the computational time is not far from the one obtained with the compiled files in Table 7. This is due to a new implementation (in the `m`-files) of matrix and vector assembly, already introduced in version 2 of `GeoPDEs`, which makes a more efficient use of vectorization.

Acknowledgments

I would like to thank Carlo de Falco and Alessandro Reali for the fruitful collaboration that led to the first version of `GeoPDEs`, and specially Carlo de Falco for many comments and suggestions during the development of this new version. I would also like to thank Pablo Antolín, Ericka Brivadis, Annalisa Buffa, Jacopo Corno, Eduardo M. Garau, Massimiliano Martinelli and Lorenzo Tamellini, for beta-testing and/or for their careful reading and valuable comments about this work. `GeoPDEs` has been enriched by the contributions of several developers, the complete list of contributors is available in the `GeoPDEs` webpage[15]. Regarding this particular work, I am grateful to Andrea Bressan and Adriano Côrtes for important contributions to Stokes problem.

This work was partially supported by the European Research Council through the FP7 ERC Consolidator Grant no. 616563 HIGEOM, by European Union's Horizon 2020 research and innovation pro-

[15] `http://rafavzqz.github.io/geopdes`

	Version 1.1				Version 3.0			
Mesh size	$p = 1$	$p = 2$	$p = 3$	$p = 4$	$p = 1$	$p = 2$	$p = 3$	$p = 4$
1/2	0.1859	0.4778	1.766	5.888	0.2160	0.2375	0.2845	0.4334
1/4	0.5147	2.606	12.37	44.87	0.4259	0.5612	0.9584	2.129
1/8	2.556	18.48	96.58	353.8	1.190	2.269	5.371	14.92
1/16	16.73	141.1	748.5	2889	5.597	14.13	38.85	115.9
1/32	121.4	1107	6033	——	37.17	105.6	305.0	926.1
1/64	926.7	——	——	——	279.3	831.1	2446	7389

Table 8: Computational time (in seconds) using `m`-files

gramme through the grant no. 680448 CAxMan, and by the Italian Ministry of Education, University and Research through the project PRIN-2012HBLYE4. This support is gratefully acknowledged.

A Methods of the classes

In this appendix we include a set of tables (Tables 9 to 11) with the methods of the different classes present in `GeoPDEs`.

Method name	Output	Description
msh_precompute	msh struct (see Table 13)	Compute the parametrization at the quadrature nodes of all elements
msh_evaluate_col	msh struct	Compute the parametrization at the quadrature nodes of a "column" of elements
msh_evaluate_element_list*	msh struct	Compute the parametrization at the quadrature nodes of a given list of elements
msh_eval_boundary_side	msh struct	Compute the restriction of the parametrization to a given boundary side, for all the elements of that side
msh_boundary_side_from_interior	msh struct	Compute the parametrization for all the elements of a given boundary side, taking into account information from the interior (useful for normal derivatives)
msh_refine*	**msh_cartesian** object	Generate a uniformly refined mesh from the given one

Table 9: The methods of the **msh_cartesian** class. Methods marked with an asterisk (*) have an analogous version in **msh_multipatch**.

Method name	Output	Description
Methods to compute the basis functions		
sp_precompute	space struct (see Table 14)	Compute the shape functions at the quadrature points of all elements
sp_evaluate_col	space struct	Compute the shape functions at the quadrature points of a "column" of elements
sp_evaluate_element_list*	space struct	Compute the shape functions at the quadrature points of a given list of elements
sp_eval_boundary_side	space struct	Compute the restriction of the space to the selected boundary side, and for all the elements of that side
Methods for post-processing		
sp_l2_error*	Scalar	Evaluate the error of the computed solution in L^2 norm
sp_h1_error*	Scalar	Evaluate the error of the computed solution in H^1 norm and seminorm
sp_eval	NDArray or cell array (see the function help)	Evaluate the computed solution in a Cartesian grid of points
sp_to_vtk*	External file	Export a VTK file to be read in `ParaView`, using the structured data file format
Basic connectivity operations		
sp_get_basis_functions*	Array	Compute the functions that do not vanish in a given list of elements
sp_get_cells*	Array	Compute the elements on which a list of functions is supported
sp_get_neighbors*	Array	Compute the functions such that their (open) support intersects the one of the input function(s)
Other methods		
sp_drchlt_l2_proj*	Two arrays	Compute the degrees of freedom to impose the Dirichlet boundary condition using the L^2 projection
sp_refine*	**sp_scalar** or **sp_vector** object	Refine the discrete space. Usually called after **msh_refine**

Table 10: The methods common to the **sp_scalar** and **sp_vector** classes. Methods marked with an asterisk (*) have an analogous version in **sp_multipatch**.

Method name	Output	Description
Methods for post-processing		
sp_hcurl_error*	Scalar	Evaluate the error of the computed solution in $H($ **curl** $)$ norm
Boundary conditions		
sp_drchlt_l2_proj_udotn*	Two arrays	Compute the degrees of freedom to impose the Dirichlet boundary conditions for the normal component, to be used with the *div-preserving* transformation
sp_weak_drchlt_bc*	One matrix, one vector	Compute the matrix and vector to impose Dirichlet boundary conditions in weak sense, for Stokes problem with *div-preserving* transformation

Table 11: Methods of the **sp_vector** class. Methods marked with an asterisk (*) have an analogous version in **sp_multipatch**.

B Available operators for matrix and vector assembly

We give in Table 12 the available operators for matrix and vector assembly, grouped by their presence in different classes.

Methods present in both **sp_scalar** and **sp_vector**
op_f_v_tp, op_gradu_gradv_tp, op_u_v_tp
Methods only present in the class **sp_scalar**
op_gradgradu_gradgradv_tp, op_laplaceu_laplacev_tp, op_mat_stab_SUPG_tp, op_rhs_stab_SUPG_tp, op_vel_dot_gradu_v_tp
Methods only present in the class **sp_vector**
op_curlu_curlv_tp, op_curlu_v_tp, op_div_v_q_tp, op_su_ev_tp, op_v_gradp_tp
Methods present in the class **sp_multipatch**
op_f_v_mp, op_gradu_gradv_mp, op_u_v_mp, op_curlu_curlv_mp, op_div_v_q_mp, op_su_ev_mp, op_v_gradp_mp

Table 12: Methods for matrix and vector assembly, grouped by classes

C Fields in the mesh and space structures

In order to make this document self-contained, we include two tables with a short description of the fields in the *mesh* and *space* structures (Tables 13 and 14, respectively), as they were introduced in the first version of `GeoPDEs`. As we have seen in Sections 3.3 and 3.4, these structures are also used in the new version when it is necessary to compute some information in a certain region of the domain.

Field name	Type	Dimensions	Description
nel	scalar	1×1	Number of elements *in the structure*
quad_nodes	NDArray	ndim×nqn×nel	Coordinates of the quadrature nodes in the parametric domain
quad_weights	Matrix	nqn×nel	Weights associated to the quadrature nodes
geo_map	NDArray	rdim×nqn×nel	Physical coordinates of the quadrature nodes
geo_map_jac	NDArray	rdim×ndim×nqn×nel	Jacobian matrix of the parametrization, evaluated at the quadrature nodes
geo_map_der2	NDArray	rdim×ndim×ndim×nqn×nel	Second order derivatives of the parametrization, evaluated at the quadrature nodes
jacdet	Matrix	nqn×nel	Element measure evaluated at the quadrature nodes. Determinant of the Jacobian for $n = r$
element_size	Array	1×nel	Estimated size of the elements, computed using the given quadrature rule
normal	NDArray	rdim×nqn×nel	Exterior normal evaluated at quadrature nodes, for boundary entities and for surfaces in $\mathbb{R}^3$

Table 13: The **msh** data structure.

Field name	Type	Dimensions	Description
ndof	scalar	1×1	Total number of degrees of freedom of the space
nsh	Array	1×nel	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
Other optional fields, their availability depends on the space class and the chosen transformation			
shape_function_gradients	NDArray	(ncomp×)rdim×nqn×nsh_max×nel	Gradient of the basis functions
shape_function_hessians	NDArray	(ncomp×)rdim×rdim×nqn×nsh_max×nel	Hessian of the basis functions
shape_function_laplacians	NDArray	(ncomp×)nqn×nsh_max×nel	Laplacian of the basis functions
shape_function_curls	NDArray	nqn×nsh_max×nel (for $n = 2$) ncomp×nqn×nsh_max×nel (for $n = 3$)	Curl of the basis functions
shape_function_divs	NDArray	nqn×nsh_max×nel	Divergence of the basis functions

Table 14: The **space** data structure.

D 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, and now a package of the Octave-forge project [27]. The original toolbox was mainly developed for the construction of NURBS curves and surfaces, based on the algorithms of [25]. In order to deal with three-dimensional problems in IGA, we have extended some of the basic algorithms to trivariate NURBS. We give now a short explanation of the main features of the toolbox, and refer the reader to [27] 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**: control points in homogeneous (projective) coordinates, sometimes also called weighted control points. These are 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. 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 [25, 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. We list here the most important ones:

- **nrbline**: construct a straight line as a linear NURBS.
- **nrbcirc**: construct a circular arc as a quadratic NURBS.
- **nrb4surf**: construct a NURBS bilinear surface.
- **nrbrevolve**: construct a NURBS surface, or volume, by revolution of a NURBS curve or surface, respectively.
- **nrbextrude**: construct a NURBS surface, or volume, by extrusion of a NURBS curve or surface, respectively, along a constant vector.
- **nrbruled**: construct a ruled surface between two NURBS curves.
- **nrbcoons**: construct a Coons patch from four given NURBS curves.
- **nrbtform**: apply an affine transformation to a NURBS geometry.
- **nrbextract**: construct NURBS curves or surfaces, extracting the boundary information from a surface or volume, respectively.
- **nrbkntins**: perform knot insertion.

- **nrbdegelev**: perform degree elevation.
- **nrbkntplot**: plot the NURBS geometry, with the image of the knot vector (the Bézier mesh).
- **nrbctrlplot**: plot the NURBS geometry with its control net.
- **nrb2iges**: write a NURBS curve or surface to an IGES file.
- **nrbexport**: export a NURBS geometry to a text file with the format used in `GeoPDEs`.
- **nrbmultipatch**: construct the information for gluing several NURBS patches.

References

[1] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application*, Kitware, 2015.

[2] L. Beirão da Veiga, A. Buffa, G. Sangalli, and R. Vázquez, *Mathematical analysis of variational isogeometric methods*, Acta Numerica, 23 (2014), pp. 157–287.

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

[4] A. Bressan and G. Sangalli, *Isogeometric discretizations of the Stokes problem: stability analysis by the macroelement technique*, IMA J. Numer. Anal., 33 (2013), pp. 629–651.

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

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

[7] A. Buffa, G. Sangalli, and R. Vázquez, *Isogeometric methods for computational electromagnetics: B-spline and T-spline discretizations*, J. Comput. Phys., 257, Part B (2014), pp. 1291 – 1320.

[8] N. O. Collier, L. Dalcin, and V. M. Calo, *PetIGA: High-performance isogeometric analysis*, 2013.

[9] A. Côrtes, A. Coutinho, L. Dalcin, and V. Calo, *Performance evaluation of block-diagonal preconditioners for the divergence-conforming B-spline discretization of the stokes system*, Journal of Computational Science, 11 (2015), pp. 123 – 136.

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

[11] C. de Falco, A. Reali, and R. Vázquez, *GeoPDEs: a research tool for Isogeometric Analysis of PDEs*, Adv. Engrg. Softw., 42 (2011), pp. 1020–1034.

[12] L. Dedè and A. Quarteroni, *Isogeometric analysis for second order partial differential equations on surfaces*, Comput. Methods Appl. Mech. Engrg., 284 (2015), pp. 807 – 834.

[13] T. Dokken, E. Quak, and 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), Genova, Italy, 2010, IMATI-CNR, pp. 11–15.

[14] J. W. Eaton, D. Bateman, and S. Hauberg, *GNU Octave Manual Version 3*, Network Theory Ltd., 2008.

[15] J. A. Evans and T. J. R. Hughes, *Isogeometric divergence-conforming B-splines for the Darcy-Stokes-Brinkman equations.*, Math. Models Methods Appl. Sci., 23 (2013), pp. 671–741.

[16] E. M. Garau and R. Vázquez, *Algorithms for adaptive isogeometric methods based on hierarchical splines, with an implementation in GeoPDEs.* In preparation.

[17] R. Hiptmair, *Finite elements in computational electromagnetism*, Acta Numer., 11 (2002), pp. 237–339.

[18] T. J. R. Hughes, A. Reali, and G. Sangalli, *Efficient quadrature for NURBS-based isogeometric analysis*, Comput. Methods Appl. Mech. Engrg., 199 (2010), pp. 301 – 313.

[19] B. Juettler, U. Langer, A. Mantzaflaris, S. Moore, and W. Zulehner, *Geometry + simulation modules: Implementing isogeometric analysis*, Proc. Appl. Math. Mech., 14 (2014), pp. 961–962. Special Issue: 85th Annual Meeting of the Int. Assoc. of Appl. Math. and Mech. (GAMM), Erlangen 2014.

[20] A. Karatarakis, P. Karakitsios, and M. Papadrakakis, *GPU accelerated computation of the isogeometric analysis stiffness matrix*, Comput. Methods Appl. Mech. Engrg., 269 (2014), pp. 334–355.

[21] S. K. Kleiss, C. Pechstein, B. Jüttler, and S. Tomar, *IETI-Isogeometric Tearing and Interconnecting*, Comput. Methods Appl. Mech. Engrg., 247–248 (2012), pp. 201 – 215.

[22] P. Monk, *Finite element methods for Maxwell's equations*, Oxford University Press, Oxford, 2003.

[23] V. P. Nguyen, C. Anitescu, S. P. Bordas, and T. Rabczuk, *Isogeometric analysis: An overview and computer implementation aspects*, Math. Comput. Simulation, 117 (2015), pp. 89 – 116.

[24] M. S. Pauletti, M. Martinelli, N. Cavallini, and P. Antolin, *Igatools: An isogeometric analysis library*, SIAM J. Sci. Comput., 37 (2015), pp. C465–C496.

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

[26] A. F. Sarmiento, A. M. A. Cortes, D. A. Garcia, L. Dalcin, N. Collier, and V. M. Calo, *PetIGA-MF: a multi-field high-performance toolbox for structure-preserving B-splines spaces*, 2016.

[27] M. Spink, D. Claxton, C. de Falco, and R. Vázquez, *The NURBS toolbox*. `http://octave.sourceforge.net/nurbs/index.html`.

Recent titles from the IMATI-REPORT Series:

16-01: *Optimal strategies for a time-dependent harvesting problem*, G.M. Coclite, M. Garavello, L.V. Spinolo, 2016.

16-02: *A new design for the implementation of isogeometric analysis in Octave and Matlab: GeoPDEs 3.0, R. Vázquez, 2016.*

Istituto di Matematica Applicata e Tecnologie Informatiche ``*Enrico Magenes",* CNR
v. Ferrata 5/a, 27100, Pavia, Italy
Genova Section: Via dei Marini, 6, 16149 Genova, Italy • **Milano Section:** Via E. Bassini, 15, 20133 Milano, Italy

http://www.imati.cnr.it/