IGATOOLS: AN ISOGEOMETRIC ANALYSIS LIBRARY

M. SEBASTIAN PAULETTI*, MASSIMILIANO MARTINELLI†, NICOLA CAVALLINI‡, AND PABLO ANTOLIN§

Abstract. We present the design of an object oriented general purpose library for isogeometric analysis, where the mathematical concepts of the isogeometric method and their relationships are directly mapped into classes and their interactions. The encapsulation of mathematical concepts into interacting building blocks gives flexibility to use the library in a wide range of scientific areas and applications. We provide a precise framework for a lot of loose, available information regarding the implementation of the isogeometric method, and also discuss the similarities and differences between this and the finite element method.

We also describe how to implement this proposed design in a C++11 open source library, `igatools` (`http://www.igatools.org`). The library uses advanced object oriented and generic programming techniques to ensure re-usability, reliability and maintainability of the source code. It includes isogeometric elements of the h-div and h-curl type, and supports the development of dimension independent code (including manifolds and tensor-valued spaces).

We finally present a number code examples to illustrate the flexibility and power of the library, including surface domains, nonlinear elasticity and Navier-Stokes computations on non-trivial geometries.

Key words. isogeometric analysis, software library, open source, object oriented, generic programming, dimension independent, B-splines, NURBS, h div, h curl, CAD integration

AMS subject classifications. 65N30, 97N80, 97N40, 68N19, 65D07

1. Introduction.

Inspired by the desire to unify the fields of computer aided geometrical design (CAGD) and the finite element method (FEM), the work [30] introduced a technique for the discretization of partial differential equations dubbed *isogeometric analysis* (IGA). In this work, any numerical method fitting the isogeometric analysis framework will be referred to as an *isogeometric method* (IgM). The mainly advertised feature of an IgM has been the ability to describe exactly CAGD type geometries. This is so because the method proposes to use the same type of spaces to represent the geometry and the shape functions (mostly non uniform rational B-splines). But in addition to exact representation of CAGD geometries the use of B-spline functions allows global smoothness beyond the classical C^0 continuity of standard finite elements: this permits the design of novel numerical schemes that would be extremely difficult to obtain with standard finite elements. Isogeometric methods have been summarized in a recent book [19] and studied in [16, 17]. They have been successfully used in applications such as fluid [6, 7, 24] and structural mechanics [44, 33, 20].

After a decade of research in the area, with many successes, some issues (that

*Instituto de Matemática Aplicada del Litoral (IMAL). Consejo Nacional de Investigaciones científicas y técnicas (CONICET). Santa Fe, Argentina. Partially supported by European Research Council through the FP7 Ideas Starting Grant 205004 GeoPDEs and Agencia Nacional de Promoción Científica y Tecnológica, through grant PICT-2008-0622 (Argentina).

†Istituto di Matematica Applicata e Tecnologie Informatiche (IMATI). Consiglio Nazionale delle Ricerche (CNR). Pavia, Italy. Partially supported by FIRB "Futuro in Ricerca" RBFR08CZ0S.

‡Scuola Internazionale Superiore di Studi Avanzati. Via Bonomea 265, Trieste, Italy. Supported by the TERRIFIC project, European Communitys Seventh Framework Programme, Grant Agreement 284981 Call FP7-2011-NMP-ICT-FoF.

§Dipartimento di Ingegneria Civile ed Architettura. Università degli Studi di Pavia. Via Ferrata 3, 27100 Pavia, Italy. Supported by European Research Council through the FP7 Ideas Starting Grant 259229 ISOBIO.

require further research) and many promising ideas, it is time, at least in the research community, to have an open source modernly designed isogeometric software library.

It must be capable of dealing with the current understanding of an IgM while providing the flexibility to adapt to the future evolution of the IgM. Also important, to justify the time invested in its learning, it should follow high quality standards in design and development. Unfortunately, we did not find an isogeometric library with the features and quality standards of our expectations. This need and the scientific environment at Pavia provided the supporting framework for us to undertake the first step toward the creation of such software. Pavia enjoyed the early stages on the development of the isogeometric concept. Several scientists in mathematics and engineering contributed with their work to the theoretical and computational development of isogeometric analysis. For several years `GeoPDEs` [21] (an easy to use MATLAB toolkit) has "propelled" the dissemination of the concept in its early stages partially paving the road for our work.

The main purpose of this work is to present an object oriented design for an isogeometric software library and introduce a C++11 implementation of it, `igatools`. `igatools` is a general purpose open source library to solve partial differential equations using isogeometric type spaces. Unlike most academic software, that start as a tool to solve the specific applications of a small research group, `igatools` was conceived from the beginning to provide useful tools to aid in the approximation to a wide range of possible applications.

The design of `igatools` encapsulates the mathematical concepts of an isogeometric method into objects and map their relations into interaction between these objects. We tried hard to attain the simplest possible design that is faithful to the mathematics behind. As a guiding rule to validate the design we use the following statement: "if an awkward code is needed to perform a task consistent with the mathematics of the problem then there is a problem with the design of the library". Desirable consequences of such a design are improved maintainability, as it produces a code that reads closer to the human conception of the problem, and flexibility for extensions, as adding a new feature coming from the mathematics of the problem should face no major restrictions.

A good design is paramount for a software library to transcend the original group of developers. But it is also necessary to carefully plan and execute the structure and documentation of the code, as well as the infrastructure to support its development. In `igatools` we have considered all these principles. The library is implemented in C++11 [32, 9], its design is object oriented with extensive use of generic programming techniques that allows among other advantages to obtain code that is dimension independent while suppressing the overhead of virtual functions. The development environment follows the high quality standards of today's software engineering: we use a platform independent configuration system (`CMake`), an automatic test suite (`CTest/CDash`), a version control system (`git`), a bug tracking system, an in-source documentation system (`Doxygen`), a user discussion forum and an on-line tutorial. On top of the novel object oriented design that it implements, `igatools` provides practical useful features such as the possibility of floating-point types of different precision, integration with CAGD modelers, as well as integration with well known libraries for solving linear and nonlinear systems.

The IgM is closely related to the well known FEM, they are both Galerkin methods that provide an specific way of constructing the basis functions of the discrete approximating spaces. Some finite element spaces can be generated with the IgM

(Lagrange elements on quadrilaterals for example). In a loose sense an IgM can be interpreted as a generalized FEM where the basis functions are more regular. But this interpretation starts to diverge as a closer look is taken onto the details, for example even though the Lagrange spaces are the same, the basis functions generated using the FEM are different from the ones of the IgM. It is our view that even though a finite element code could be adapted to accept an isogeometric plug-in [13], this approach presents some shortcomings. The popular well-maintained, modernly designed and stable FE libraries are so tailored for the FEM that using them for and IgM makes some forced interpretation of a few concepts. In fact, we would need to define an isogeometric counterpart for the finite element triple or the unstructured triangulation. It is our opinion that forcing IgM concepts to fit into a FEM design, without existing a true matching between concepts, will lead to a confusing design affecting clarity, maintainability, efficiency and extensibility of the code. Still we should not forget the similarities these methods share. Our IgM design is inspired by these similarities as well as these differences to produce a software that suites an easy implementation of state of the art method and original techniques. In particular, we have absorbed many applicable concepts from an open source finite element library that seems to have survived the test of time, namely the `deal.II` library [2, 3], also for the sake of fairness we must say that most of these concepts are similarly found in many of the other excellent finite element software available.

This work is neither a manual nor a user guide for `igatools`. In fact, there are on-line documentation (more than hundreds of pages if printed) and a tutorial dedicated for these purposes. The main purpose is to present and explain a novel object oriented design for isogeometric analysis faithful to the mathematics of the IgM and introduce a C++11 implementation of it, `igatools`.

The outline for this work follows the ideal approach for designing a computer program (see [42]) in three stages: analysis, design and programming. In the analysis we need to gain a clear understanding of the problem, in our case the IgM, this is what we do in §2, where we describe the method from the viewpoint of designing a software library for it. The design is presented in §3 where the key concepts of the IgM and their relations are identified and mapped into objects. Here we also detail the main classes implemented in `igatools` (the programming stage). In §4 the software engineering tools and development model adopted are described; and some additional useful features of `igatools`, that are besides the IgM concepts, are presented in §5. Finally, in §6 we present some common applications that our library can easily accommodate including the Poisson problem, Laplace-Beltrami on surfaces, a fluid application and an elasticity problem.

2. Isogeometric spaces. In order to design an isogeometric library, it is crucial first to achieve a clear understanding of what an isogeometric method is. Since its introduction what we call isogeometric method (IgM) has evolved to incorporate new concepts and ideas. The original emphasis when IGA was introduced was on the exact representation of the geometry and the requirement of the isoparametric paradigm to be satisfied. On one hand, new reference spaces beside NURBS has been introduced (T-splines [5, 23, 15, 39], locally refined splines [22] and hierarchical splines [26, 27, 43]). On the other hand, the isoparametric constraint has been lately relaxed [18, 17, 16] to include mappings that are not necessarily in the reference space. We also support the relaxation of the exact domain geometry constraint. From the mathematical point of view, is not mandatory to have an exact representation of the domain (what is needed is an approximation of the geometry that leads

to an error of the same order of the one that is due to the approximation order of the discrete space), but from an engineering point of view it is extremely useful to have an exact representation of the geometry: indeed with an exact representation, the geometry (and the grid built upon it) can be easily modified/manipulated (for instance in optimization loops) without the needs of re-meshing procedures. For the approximation of some vector-valued problems to be stable, it is known [11, 35], that the approximating spaces must satisfy some properties. Particular examples are the Raviar-Thomas and the edge elements. A more general mathematical framework in the language of differential forms and de Rham cohomology has been studied in [1] and applied in the context of isogeometric methods in [18, 17]. Our design supports these spaces through the use of a transformation type and the concept of an associated push-forward operator.

Thus in our design, we conceive an isogeometric method to be a method that incorporates most existing flavors of isogeometric techniques such as the ones described in the previous paragraph. But moreover, based on the fact that the design strives to be conceptually faithful to the mathematics, we expect the software to be flexible enough to adapt to and encourage the creation of new methods that are unknown at the moment.

In this section we give a brief overview of the isogeometric method, B-splines and NURBS while introducing the notation used in this work. Details can be found in the standard bibliography, for example [38, 36, 19].

2.1. The isogeometric method. An IgM is a type of Galerkin method to approximate the solution of boundary value problems. To make the discussion more concrete let us consider a differential equation in a d-dimensional domain $\Omega \subset \mathbb{R}^s$, being s the dimension of the *embedding euclidean space* and $s - d$ the *codimension* of the domain. More precisely, let V be a Sobolev space (infinite dimensional) on Ω, $\mathcal{A} : V \to V'$ a differential operator and $f \in V'$ a linear functional. The variational problem is to find $u \in V$ such that $\mathcal{A}u = f$. The Galerkin procedure approximates the solution u by solving a similar problem in finite dimensional subspaces. The IgM (as well as the FEM) provides a recipe to construct these discrete spaces. More specifically, we define an IgM as one providing:

1. **A reference space**. Given a d-dimensional rectangle $\hat{\Omega} \subset \mathbb{R}^d$ that we call the *reference domain*, provides a specific way to construct a *reference space* $\hat{\mathbb{V}}(\hat{\Omega})$ of the B-spline type (see definition 2.1 on page 8).

2. **A deformation**. Provides a smooth deformation $\boldsymbol{F} : \hat{\Omega} \to \mathbb{R}^s$ (with smooth inverse). We refer to $\boldsymbol{F}$ as the *mapping* and to $\Omega = \boldsymbol{F}(\hat{\Omega})$ as the *physical domain*.

3. **A transformation type**. This is a rule that specifies how to use the deformation to transform the functions in the reference space into the ones in the physical space. Some example besides direct composition are divergence and curl preserving transformations (cf. Table 2.1).

`h_grad`	`h_curl`	`h_div`	`l_2`
$\phi = \hat{\phi} \circ \boldsymbol{F}^{-1}$	$\boldsymbol{u} = D\boldsymbol{F}^{-T}(\hat{\boldsymbol{u}} \circ \boldsymbol{F}^{-1})$	$\boldsymbol{v} = \dfrac{D\boldsymbol{F}}{\det D\boldsymbol{F}}(\hat{\boldsymbol{v}} \circ \boldsymbol{F}^{-1})$	$\varphi = \dfrac{\hat{\varphi} \circ \boldsymbol{F}^{-1}}{\det D\boldsymbol{F}}$

TABLE 2.1

Examples of different transformation types to obtain a physical discrete space $\mathbb{V}$ from a reference spline or NURBS space $\hat{\mathbb{V}}$ and a mapping $\boldsymbol{F}$. The table shows the transformation name given in the library and the formula that defines its push-forward operator.

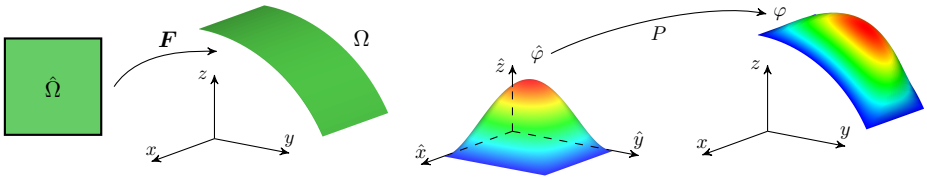


FIG. 2.1. *The isogeometric method delivers a recipe to construct a discrete space by providing a reference space of the B-Spline type, a deformation* $\boldsymbol{F}$ *and a transformation type. The transformation type indicates how to use the deformation to define the push-forward operator* P*. The physical space* $\mathbb{V}(\Omega)$ *is obtained as the image of the reference space* $\hat{\mathbb{V}}(\hat{\Omega})$ *through* P*.*

The mapping together with the *transformation type* define a *push-forward* operator[1] P and this operator acts on the reference space to give the *physical space* $\mathbb{V}(\Omega)$ (see Figure 2.1) as $\mathbb{V}(\Omega) = \{\phi = P(\hat{\phi}) \text{ with } \hat{\phi} \in \hat{\mathbb{V}}(\hat{\Omega})\}$. We say that $\mathbb{V}(\Omega)$ is an *isogeometric space* and a function $u \in \mathbb{V}(\Omega)$, is generally referred to as a *field*.

2.2. B-splines and NURBS.

2.2.1. Univariate B-spline. Given a non negative number p, a *spline* of degree p on the interval $[a, b]$ is a real valued piecewise polynomial function of degree at most p on each subinterval of $[a, b]$ determined by the partition $a = \zeta_1 < \cdots < \zeta_m = b$. The ζ_i's are called the *knots* and they form the *knot vector* $\boldsymbol{\zeta} = (\zeta_1, \ldots, \zeta_m)$. At each knot, the spline function is allowed to have a regularity that ranges from discontinuous (C^{-1}) to C^{p-1}, this is usually indicated using the so called *regularity vector* $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_m)$ where $\alpha_i \in \mathbb{Z}$ and $-1 \leq \alpha_i \leq p-1$. It is sometimes convenient to encode both knot and regularity vectors into a single vector of repeated knots $\boldsymbol{\xi} = (\underbrace{\zeta_1, \ldots, \zeta_1}_{r_1 \text{ times}}, \underbrace{\zeta_2, \ldots, \zeta_2}_{r_2 \text{ times}}, \ldots, \underbrace{\zeta_m, \ldots, \zeta_m}_{r_m \text{ times}})$, where $r_i = p - \alpha_i$, and $\boldsymbol{r} = (r_1, \ldots, r_m)$ is called the *multiplicity vector*. When the multiplicity of the first and last knots is $p+1$ we call $\boldsymbol{\xi}$ an *open knot vector*.

We define $S^p_{\boldsymbol{\xi}}$ to be the space of spline functions of degree p subordinated to the knot vector with repetition $\boldsymbol{\xi}$. It is a well known result that the dimensionality of this space is $n = \sum_{i=1}^{m} r_i - p - 1$. The classical Cox-de Boor recursive algorithm [36] allows to construct a basis for $S^p_{\boldsymbol{\xi}}$, known as the B-spline basis. We denote these basis functions by B_i (see Figure 2.2). Some important properties of B-spline basis functions are:

(i) *Non-negativity*: each $B_i(x) \geq 0$ for all $x \in [a, b]$

(ii) *Small support*: each B_i is non-zero only in $[\xi_i, \xi_{i+p+1})$ and on each interval $[\zeta_i, \zeta_{i+1})$ (with $i = 1, \ldots, m-1$), only $p+1$ basis functions have non-zero values

(iii) *Partition of unity*: the set of basis functions $\{B_1, \ldots, B_n\}$ satisfies that $\sum_{i=1}^{n} B_i(x) = 1$, for all $x \in [a, b]$ if $\boldsymbol{\xi}$ is an open knot vector.

A convenient method to compute values and derivatives of univariate B-spline functions can be achieved using the so called *Bézier extraction* operator technique. This method was introduced by Borden et al. [13], as a local adaptation of the global Bézier decomposition algorithm [36, A5.6]. More precisely, as a B-spline function of degree p restricted to the k-th interval $I_k = [\zeta_k, \zeta_{k+1})$ is a polynomial of degree p, we

[1] In the context of differential geometry P would be called the pullback (as there is no distinction between reference and physical), for our setting it is more natural and clear to refer to P as the push-forward of reference functions to physical functions.

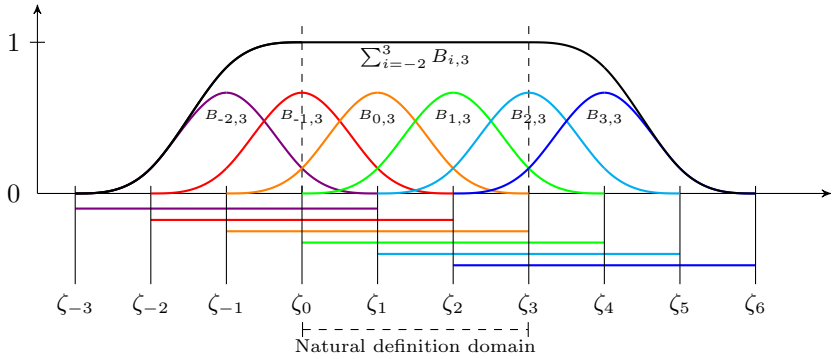


FIG. 2.2. *One dimensional B-spline basis functions of degree 3 for a maximum regularity knot vector. As it can be seen, all the functions are non-negative; they span over four knot intervals (small support); on each interval only four function have non-zero values; and they form a partition of unity on the interval* $[\zeta_0, \zeta_3]$.

can write this polynomial as a linear combination of the $p+1$ Bernstein polynomial basis functions $b_{j,p}$ of degree p as

$$B_i(x)\Big|_{[\zeta_k,\zeta_{k+1})} = B_{k,i}(x) = \sum_{j=1}^{p+1} c_{ij}^{(k)} b_{j,p}\Big(\frac{x-\zeta_k}{\zeta_{k+1}-\zeta_k}\Big), \qquad i = 1,\dots,p+1, \tag{2.1}$$

with $b_{j,p}(t) = \binom{p}{j-1} t^{j-1}(1-t)^{p-j+1}$ for $0 \le t \le 1$ and $j = 1,\dots,p+1$. The coefficients $c_{ij}^{(k)}$ define the k-th interval matrix $C^{(k)} = (c_{ij}^{(k)})$, referred to as the k-th *Bézier extraction operator* [2]. A simple consequence of equation (2.1) is that the m-th derivative of a B-spline function can be written as linear combination of the m-th derivatives of the Bernstein polynomials with the same coefficients $c_{ij}^{(k)}$ as the function. More precisely,

$$B_{k,i}^{(m)}(x) = \sum_{j=1}^{p+1} c_{ij}^{(k)} \frac{d^m b_{j,p}}{d\zeta^m}\Big(\frac{x-\zeta_k}{\zeta_{k+1}-\zeta_k}\Big) = \frac{1}{(\zeta_{k+1}-\zeta_k)^m} \sum_{j=1}^{p+1} c_{ij}^{(k)} b_{j,p}^{(m)}(t) \tag{2.2}$$

with $t = \frac{x-\zeta_k}{\zeta_{k+1}-\zeta_k}$. This property leads to an efficient method to compute values and derivatives of univariate B-splines. More precisely, assume we have a spline space with N intervals and we want to compute the values and derivatives of the B-spline functions using the same quadrature scheme on each interval. According to equation (2.2) we need to compute the matrices $C^{(k)}$ (N times) and $b_{j,p}^{(m)}$ at the unit quadrature points (one time, independent of N). With this information pre-computed we can compute all the required values and derivatives of the B-spline functions over all the intervals. Thus making this evaluation method efficient and convenient, in the context of an IgM.

2.2.2. Multivariate B-splines. The univariate spline spaces can be used to generate multidimensional spline spaces through tensor product multiplication. More precisely, given a positive integer d we consider the spline spaces $S_{\boldsymbol{\xi}_i}^{p_i}([a_i, b_i])$ for $i =$

[2] The $c_{ij}^{(k)}$ coefficients are computed using the recursion formula described by the algorithm [36, A5.6].

$1, \ldots, d$ and define the multivariate spline space $S^{p_1,\ldots,p_d}_{\boldsymbol{\xi}_1,\ldots,\boldsymbol{\xi}_d}(\hat{\Omega}) = S^{p_1}_{\boldsymbol{\xi}_1}([a_1, b_1]) \otimes \ldots \otimes S^{p_d}_{\boldsymbol{\xi}_d}([a_d, b_d])$, where $\hat{\Omega}$ is the hyper-rectangle $[a_1, b_1] \times \cdots \times [a_d, b_d]$. In this case the multivariate B-Spline basis functions are

$$B_{i_1,\ldots,i_d}(x_1, \ldots, x_d) = B^1_{i_1}(x_1) \ldots B^d_{i_d}(x_d) \tag{2.3}$$

with $B^i_j \in S^{p_i}_{\boldsymbol{\xi}_i}$ for $j = 1, \ldots, n_i$ being the B-spline basis of the univariate spaces. The dimension of the space is $n = \prod_{i=1}^{d} n_i$.

To simplify the notation in the multidimensional setting we define $\boldsymbol{p} = (p_1, \ldots, p_d)$ to be the vector of degrees, $\overrightarrow{\boldsymbol{\xi}} = (\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_d)$ the vector of knot vectors with repetition and $\overrightarrow{\boldsymbol{\zeta}} = (\boldsymbol{\zeta}_1, \ldots, \boldsymbol{\zeta}_d)$ the vector of knot vectors without repetition. Then we can write $S^{\boldsymbol{p}}_{\overrightarrow{\boldsymbol{\xi}}}(\hat{\Omega})$ instead of $S^{p_1,\ldots,p_d}_{\boldsymbol{\xi}_1,\ldots,\boldsymbol{\xi}_d}(\hat{\Omega})$. Similarly, we consider a vector index like $\boldsymbol{i} = (i_1, \ldots, i_d)$ for tensor product structures such as basis functions or elements, or $\boldsymbol{m} = (m_1, \ldots, m_d)$ as a derivative multi-index. Thus we can write $D^{\boldsymbol{m}} B_{\boldsymbol{i}}(\boldsymbol{x})$ as a short notation for $\frac{\partial^m B_{i_1,\ldots,i_d}}{\partial x_1^{m_1} \ldots \partial x_d^{m_d}}(x_1, \ldots, x_d)$, with $m = |\boldsymbol{m}|$. Associated to $\overrightarrow{\boldsymbol{\zeta}}$ we define the *grid* (or *mesh*, or *cartesian grid*) $\hat{\mathcal{Q}} = \{\hat{Q}_{i_1,\ldots,i_d} = [\zeta^1_{i_1}, \zeta^1_{i_1+1}] \times \cdots \times [\zeta^d_{i_d}, \zeta^d_{i_d+1}]$ with $1 \le i_j \le m_j$ and $1 \le j \le d\}$, and its members $\hat{Q}_{\boldsymbol{i}}$ are called the *elements*.

One way `igatools` exploits the computational benefits of tensor products and B-splines is stated in the following remark:

REMARK 1 (Uniform computation of any order derivatives). *The tensor product structure of a multivariate B-spline together with the Bezier extraction technique provide the foundation to an algorithm that translates to a single piece of code capable of computing any order of derivatives for multivariate B-spline functions.*

To see this consider the multi-index $\boldsymbol{m}$ and differentiate (2.3) from where it follows that $D^{\boldsymbol{m}} B_{\boldsymbol{i}}(\boldsymbol{x}) = \prod_{j=1}^{d} (B^j_{i_j})^{(m_j)}(x_j)$. This formula can be implemented with a single code for any size of m (whether 1 or 50). If the univariate derivatives $(B^j_{i_j})^{(m_j)}$, required by the formula, can be easily precomputed this provides a uniform and efficient code that can compute multivariate B-splines derivatives of any order. In fact this is the case happening under the Bezier extraction operation described in the previous section. This level of computational simplicity and efficiency is proper of B-splines and cannot be achieved with NURBS functions.

2.2.3. Non uniform rational B-splines (NURBS). NURBS functions are quotients of spline functions, thus a subspace of the rational functions. A NURBS space is constructed from a spline space $S^{\boldsymbol{p}}_{\overrightarrow{\boldsymbol{\xi}}}(\hat{\Omega})$ and a given strictly positive weight function $\omega \in S^{\boldsymbol{p}}_{\overrightarrow{\boldsymbol{\xi}}}(\hat{\Omega})$ as follows

$$N^{\boldsymbol{p}}_{\overrightarrow{\boldsymbol{\xi}}}(\hat{\Omega}) = N^{p_1,\ldots,p_d}_{\boldsymbol{\xi}_1,\ldots,\boldsymbol{\xi}_d}(\hat{\Omega}) = \{\frac{\phi}{\omega} : \phi \in S^{\boldsymbol{p}}_{\overrightarrow{\boldsymbol{\xi}}}(\hat{\Omega})\}.$$

NURBS are a fundamental tool in computer graphics. In the original isogeometric method the physical domain (cf. §2.1) is always a NURBS deformation. Without disregarding its ubiquitous use in the CAGD community (for their exact representation of conics) and the historical role in the spread of IGA into a broader community, it is important to remark that the approximation properties of a NURBS space $N^{\boldsymbol{p}}_{\overrightarrow{\boldsymbol{\xi}}}(\hat{\Omega})$ is not better than the one of its associated spline space $S^{\boldsymbol{p}}_{\overrightarrow{\boldsymbol{\xi}}}(\hat{\Omega})$. With the addition that spline spaces enjoy important theoretical and computational properties that NURBS spaces do not (for example, a spline space is closed under the differentiation operation, while a NURBS space is not).

2.2.4. Tensor-valued multivariate B-splines. Most interesting problems in applications involve vector or tensor valued quantities or even systems of them. A tensor-valued spline space is one where each component function belongs to a scalar-valued multivariate spline space as described in §2.2.2. Given the integers $s > 0$ and $k \geq 0$ let $\mathcal{T}^k(\mathbb{R}^s)$ be the space of rank k tensors over the vector space $\mathbb{R}^s$. Given s^k scalar-valued spline spaces $S_{\boldsymbol{i}} = S^{\boldsymbol{p_i}}_{\boldsymbol{\xi_i}}(\hat{\Omega})$ where $\boldsymbol{i} = (i_1, \ldots, i_k)$ is a tensor index with $1 \leq i_j \leq s$, we define the tensor valued spline space $\boldsymbol{S}^{s,k}$ to be the set of all tensor valued functions $\boldsymbol{\phi} : \hat{\Omega} \to \mathcal{T}^k(\mathbb{R}^s)$ such that $\phi_{i_1,\ldots,i_k} \in S_{i_1,\ldots,i_k}(\hat{\Omega})$. In particular we get scalar, vector and matrix valued spaces if k equals 0, 1 or 2 respectively. We can similarly define a tensor valued NURBS space.

Definition 2.1 (B-spline type spaces). *We use the term* B-spline type space *to refer to either a spline or NURBS space. And in general to refer to any space that can be generated by some operation involving B-spline basis functions.*

2.3. Multipatch. So far we have considered a physical domain Ω that is the smooth deformation of a hyper-rectangle $\hat{\Omega}$. For some geometries this is enough but in some cases, even for simple geometries, such as the one in Figure 2.3, this is not possible. The natural solution used in CAGD is to construct the physical domain as a

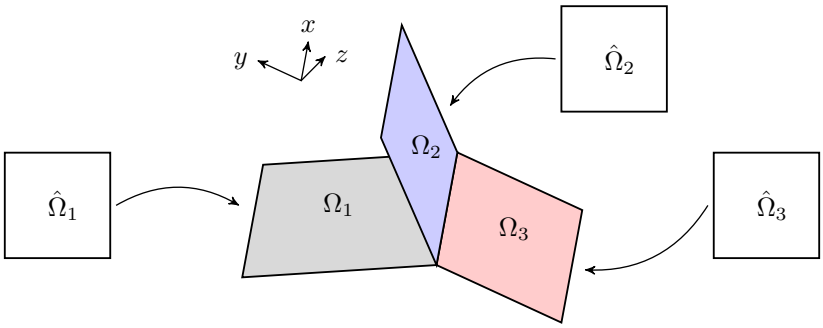


Fig. 2.3. *Simple geometry that requires more than a single patch to be represented.*

gluing of the box-like domains (the *patches*). Proceeding in a similar way one would glue together the physical spaces over each patch to obtain a physical space in the physical geometry. Even though the spaces within each patch can be very smooth, the gluing of these spaces beyond C^0 is an area of active research, beyond the scope of this paper. At the moment, the library provides C^0 gluing of patches (conforming and non-conforming) and we are experimenting on constrained based techniques to impose a C^1 gluing.

3. IgM software design. The main theoretical contribution of this work is a novel (the first one as far as we know) object oriented software design for the IgM. The main practical contribution is a C++11 implementation of this design in the open source library `igatools`. Figure 3.1 shows a top level blueprint of this design that will be described in this section. Starting by a global overview in §3.1 we continue to emphasize two important considerations that contributed to the design: a discussion on the similarities and differences with the FEM and a generic programming model for dimension and range independent code. Then we give a detailed description of the main objects, generally accompanied with snippet of code to illustrate its use finishing with a simple code to assemble a local operator in Listing 13 that integrates some of the main classes.

3.1. Design overview. Following §2.1, an IgM is a special case of a Galerkin method where the basis functions in the reference domain are of B-spline type (NURBS, T-spline, etc) and the basis functions in the physical domain are their push-forwards (see Table 2.1). The regularity of the mapping must be compatible with the one of the reference basis and the transformation type. In this conception of an isogeometric method the main mathematical concepts are the reference domain $\hat{\Omega}$, the reference space $\hat{\mathbb{V}}(\hat{\Omega})$, the physical domain Ω, the mapping $\boldsymbol{F} : \hat{\Omega} \to \Omega$, the transformation type that together with the mapping gives the push-forward operator P and the physical space $\mathbb{V}(\Omega)$. These concepts and their relations are depicted on the left column of Figure 3.1. In the physical space we are interested in handling quantities such as the differential operators $\mathcal{A} : \mathbb{V}(\Omega) \to \mathbb{V}(\Omega)'$, source operators $f \in \mathbb{V}(\Omega)'$, and fields $u \in \mathbb{V}(\Omega)$ (see lower part of Figure 3.1). We need access to these quantities both on the global level (where the system is solved and solution plotted) as well as on the local level (where the operators are assembled). Under the object oriented paradigm, a good software design for an IgM library is one that remains faithful to the mathematics of isogeometric analysis. The right column of Figure 3.1 sketches the design of `igatools` emphasizing the relation between the main concepts in an IgM (left column) in synchronization with the classes used to represent these concepts in the library.

3.2. Similarities and differences with the FEM. For the creation of our design it was important to identify the similarities and differences between the IgM and the FEM. One approach that has been proposed ([10, 13]) to implement isogeometric codes is to add an isogeometric plugin on existing FEM software packages. This idea only exploits the similarities, and could make sense for specific engineering applications where decades of FEM coding has been invested. Another approach, which is the one we follow, is to take advantages not only of the similarities but also of the differences. Thus, we use the successfully proven finite element design ideas that also apply to an IgM but we do not include FE design ideas that are not relevant to the IgM. Instead we create new concepts to produce a design naturally suitable for an IgM. The similarities come from both being Galerkin methods with basis functions of small support which makes convenient to perform the assembling by adding local contributions. Also the handling of the global system is similar but this is not even the ground of the FEM itself but of a linear algebra system. As far as the differences, we should start by saying that there are concepts in each of the methods that do not make sense for the other. For example in an IgM we have no master element, or finite element triple, or degrees of freedom associated with geometric objects of the triangulation, but we have concepts such as a reference space, a push-forward and a physical space, which on the ground of implementation have not made a lot of sense for a FEM code design. Having a design with native support for these concepts makes the whole approach to the IgM more natural and comprehensible, not only for a user but even more for new programmers that want to develop in the isogeometric concept without having to produce a forced translation to finite elements. Figure 3.2 pictorially compares the different concepts used in a FEM and an IgM to generate the physical space.

Another difference between the FEM and IgM is on how to deal with boundary conditions. In general, the values of the degrees of freedom associated with the boundary of the domain must be imposed. In the FEM, a degree of freedom is associated with a geometrical object (say a point) on which it is interpolatory. This means that an evaluation of the boundary function at the point in the domain gives the value to

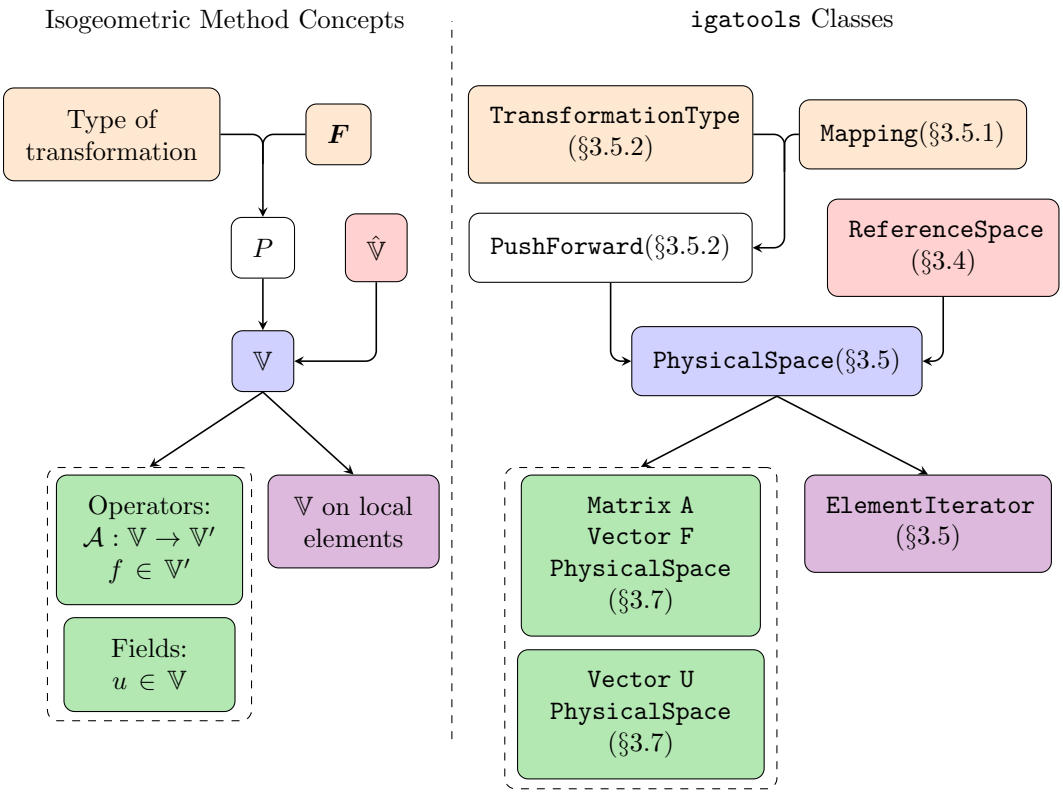


FIG. 3.1. *The conceptual diagram sketches the IgM object oriented design advocated in this work and implemented in* `igatools`*, on the left column there are the mathematical concepts found in isogeometric analysis and on the right column the corresponding realization of such concepts through the main classes in* `igatools`*. Both location and color are used to emphasis the synchronization of concepts-classes and their interaction. Under the object oriented paradigm, a good software design for an IgM library is one that remains faithful to the mathematics of isogeometric analysis. The classes include a reference to the paper section where they are described.*

impose as the corresponding boundary constraint. In the IgM the degrees of freedom are not associated with geometric points in the domain, so just evaluating at points is not an option. One way to impose the constraint is to perform a projection on the trace space. In section 6.1 we show how this is implemented in an example.

3.3. Dimension independent code. Besides the implementation of a new design, a key feature of `igatools` is the capability of writing dimension and range independent code. All classes are designed in such a way that the space dimension, co-dimension and the tensorial range are selected as template parameters. The use of templates is very convenient for scientific computing, they allow to have a single code that is resolved at compile time generating optimized code as if it was written for each instance of the dimension and with no run-time checks that would affect performance. This approach allows us to write an application code that is independent of the dimension of the domain and the range of the functions in the space (provided that the mathematical problem can so be formulated). The range, in principle is restricted to tensor spaces $\mathcal{T}^k(\mathbb{R}^s)$. For instance, let's say that a user writes the code for a 2D problem that can be mathematically formulated in any dimension (such as Poisson's

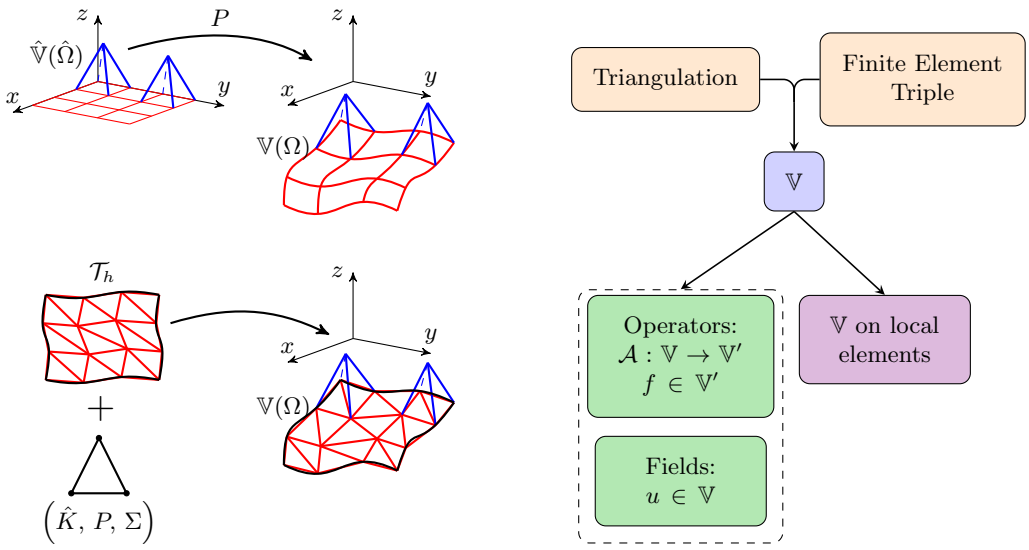


FIG. 3.2. *Both the FEM and the IgM are particular cases of the Galerkin method. They provide a specific way of constructing a sequence of discrete approximating spaces. Both share the properties of having basis function with small support, but the way to generate the spaces is essentially different as illustrated in the picture. In the FEM, the approximating space $\mathbb{V}(\Omega)$ is constructed from the finite element triple $(\hat{K}, P, \Sigma)$ and the triangulation $\mathcal{T}_h$. In the IgM the physical space is generated from a global reference space and a push-forward operator. On the right we present the FEM conceptual diagram for comparison with the IgM of figure 3.1.*

equation). If this code is written with some minimal care, then the same code will run for the problem with the physical domain being 1D, 3D or a 2D manifold embedded in $\mathbb{R}^4$ and for the solution being scalar-, vector- or matrix-valued (cf. §6.1). Table 3.1 shows an example of how the template technique for dimension independent code appears to the user in the case of the class `BSplineSpace` (§3.4). The dimension independent approach can be discovered in any piece of code shown in this work.

$\hat{\mathbb{V}}(\hat{\Omega})$ is a spline space and $\hat{\phi} \in \hat{\mathbb{V}}(\hat{\Omega})$	`igatools` realization of $\hat{\mathbb{V}}(\hat{\Omega})$
$\hat{\phi} : \mathbb{R}^2 \to \mathbb{R}$	`BSplineSpace<2,1,1> v_hat(...);`
$\hat{\boldsymbol{\phi}} : \mathbb{R}^3 \to \mathbb{R}^3$	`BSplineSpace<3,3,1> v_hat(...);`
$\hat{\boldsymbol{\phi}} : \mathbb{R}^2 \to \mathbb{R}^{3\times 3}$	`BSplineSpace<2,3,2> v_hat(...);`
$\hat{\boldsymbol{\phi}} : \mathbb{R}^d \to \mathcal{T}^k(\mathbb{R}^s)$	`BSplineSpace<d,s,k> v_hat(...);`

TABLE 3.1

The library uses template parameters to attain dimension and range independent code, the table shows an example on how the code looks when constructing an object of the `BSplineSpace` type. The left column shows the domain and range of a spline function for a given spline space, and the right column the how `igatools` uses template parameters to define a spline space for that domain and range combination. The last row shows the generic format of this class. The second and third arguments can be omitted in which case the library assumes some default values.

3.4. Reference space (`BSplineSpace` and `NURBSSpace`). Currently the library provides two classes or types for the reference space $\hat{\mathbb{V}}(\hat{\Omega})$. These are `BSplineSpace` and `NURBSSpace`, realizations of the spaces $S^{\boldsymbol{p}}_{\vec{\boldsymbol{\xi}}}(\hat{\Omega})$ and $N^{\boldsymbol{p}}_{\vec{\boldsymbol{\xi}}}(\hat{\Omega})$ described in §2.2 (in-

cluding their tensor-valued counterparts of §2.2.4). These classes can have different degrees and multiplicities in each domain direction and different scalar spaces for each range component. The last type of spaces are necessary for example in the context of fluid [25, 17]. We will use the generic class `ReferenceSpace` to refer to either `BSplineSpace` or `NURBSSpace`. In order to construct a reference space it is necessary to provide the reference domain $\hat{\Omega}$, the vector of knot vectors $\overrightarrow{\boldsymbol{\xi}} = (\boldsymbol{\xi}_1, \dots \boldsymbol{\xi}_d)$, the degrees $\boldsymbol{p} = (p_1, \dots p_d)$ and in the case of a NURBS space the weight function ω. In `igatools`, the class `CartesianGrid` provides a realization for the vector of knot vectors without repetition as well as for the reference domain (a hyper-rectangle). Listing 1 shows example of creating `CartesianGrid`s.

```
1  // 2-dim unit grid with 4 uniformly distributed knots in each direction
2  auto grid_2d = CartesianGrid<2>::create(4);
3
4  // 3-dim unit grid with 2, 4 and 3 knots in each direction
5  auto grid_3d = CartesianGrid<3>::create({2, 4, 3});
```

Listing 1

Creating `CartesianGrid`s in `igatools`, some details are explained in the Listing as comments. The code also shows the use of the create idiom implemented in `igatools` for simple handling of smart pointers (cf. §4.3).

The vector with repetition $\overrightarrow{\boldsymbol{\xi}}$ is obtained from a `CartesianGrid` and a corresponding knot multiplicities (`Multiplicity`). The NURBS space weight function is passed through its coefficients. In Listing 2 we show some examples of constructing reference spaces.

```
1  // Scalar B-spline space over [0,1]^2 of degree 2 with maximum regularity
2  auto space1 = BSplineSpace<2>::create(grid_2d, 2);
3
4  // Idem to space1 but providing the multiplicities
5  auto space2 = BSplineSpace<2>::create(grid_2d, 2, {{3,1,1,3},{3,2,1,3}});
6
7  // Vector-valued B-spline space over [0,1]^2 of
8  // degree 2 and 1 in each component and maximum regularity
9  auto space3 = BSplineSpace<2, 2>::create(grid_2d, {2,1});
10
11 // Vector-valued B-spline space over [0,1]^3 of degree 2
12 auto space4 = BSplineSpace<3, 3>::create(grid_3d, 2);
13
14 // Scalar NURBS space of degree 2 with given weight coefficients
15 auto space5 = NURBSSpace<2>::create(grid_2d, 2, weight);
```

Listing 2

Creating eference spaces. Examples of constructing B-spline type spaces in `igatools`, some details are explained inside as comments. The code is referring to the grids created in Listing 1.

3.5. Physical space (`PhysicalSpace`). The physical space $\mathbb{V}(\Omega)$ in `igatools` is realized by the class `PhysicalSpace`, this class needs to be provided with a `RefereceSpace` and a `PushForward`. The latter contains the geometry and transformation type.

3.5.1. Geometry (`Mapping`). The geometry or physical domain is described by the deformation $\boldsymbol{F}$ of the reference domain, in `igatools` the deformation is realized by the `Mapping` class. It is basically a function from $\mathbb{R}^d$ to $\mathbb{R}^s$ with the added feature of being element aware in the sense of §3.6, so that its values can be handled with the

grid-like iterators for uniformity and efficiency. Listing 3 shows examples of creating mappings. In particular, we remark that the encapsulation of the geometry in the

```
// Creating a Mapping object from the provided library of maps
auto map = SphericalMap::create(grid);

// Creating a Mapping object from a space and a vector of control points
auto bs_map = IgMapping::create(bspace, control_points);

// Importing a Mapping object from a file
IgReader<dim> reader("pipe.xml");
auto nurbs_map = reader.get_map();
```

LISTING 3

Code examples showing how to create some geometries (`Mappings`). The library allows the use of different types of deformations, including analytic functions and isogeometric mapping.

`Mapping`s class allows to cleanly integrate a geometric modeler (or a CAD system) by implementing a `Mapping` specialization (see §5.2).

3.5.2. Push-forward operator (`PushForward`). The other ingredient besides a reference space required to construct a physical space is a push-forward operator. In `igatools` it is realized by the `PushForward` class, which is constructed from a `Mapping` and a `Transformation`. Currently, `igatools` provides support for `h_grad`, `h_div`, `h_curl` and `l2` transformation types (see Table 2.1). Listing 4 shows an example for creating a push-forward operator and a physical space.

```
using PushFw = PushForward<Transformation::h_grad, dim>;
using Space  = PhysicalSpace<RefSpace, PushFw>;

auto push_fw = PushFw::create(map);
auto space   = Space::create(ref_space, push_fw);
```

LISTING 4

Creating a physical space. The physical space is created from a reference space and a push-forward operator. The push-forward operator is defined from the map using the `h_grad` transformation type. The variables `ref_space` and `map` are assumed to have been defined as shown in Listings 2 and 3 respectively. Also notice the use of the `using` keyword to simplify long type names (see §4.3).

3.6. Element level access (`ElementIterator`). A general purpose library like `igatools` cannot (and should not try to) guess what equation or problem the user will want to solve. Instead it should provide a flexible and intuitive way to access the quantities that seem to be common to all typical problems. Most operators require access to basis functions (values and derivatives of any order) and to geometric quantities (values and derivatives of the mapping, determinants and curvatures). Given the small support (in number of elements) of the basis functions, the assembling of the global quantities is conveniently performed by adding the local contributions. The `ElementIterator` is the mechanism that `igatools` provides to compute and access these values on the *element level*.

For example the `ReferenceSpace` is essentially a container of basis functions (with small support), but it can also be seen as a container of elements which contains the basis functions. The realization of this local access to basis values on the element is done with an *element iterator* in the spirit of the standard template library (STL) [41].Simply speaking, given an object collection (the container), an iterator may be

thought of as a type of pointer that has two primary operations: referencing one particular element in the container (called element access), and modifying itself so it points to the next element (called element traversal). The primary purpose of an iterator is to allow a user to process every element of a container while isolating the user from the internal structure of the container. This allows the container to store elements in any manner it wishes while allowing the user to treat it as if it were a simple sequence or list. An iterator class is designed in tight coordination with the corresponding container class.

In `igatools` we see the classes `CartesianGrid`, `BSplineSpace`, `NURBSSpace`, `Mapping`, `PushForward` and `PhysicalSpace` as containers of elements that we collectively call *grid-like* containers. The containers provide the methods for creating the iterators and thus a consistent way to iterate on the different data structures. In this way the code is more readable, reusable, and less sensitive to changes in the data structure. Moreover the container/iterator model for the grid-like containers fits perfectly in the dimension independent paradigm advocated by our design. Listing 5 shows a typical use of element iterators in `igatools` comparing the cases of a `CartesianGrid`, `ReferenceSpace` and `PhysicalSpace`. In particular notice the consistency in the treatment despite the different data stored in each class.

```
1  for(auto elem : *grid)
2  {
3    cout << elem.vertex(0);
4  }
5
6  for(auto elem : *ref_space)
7  {
8    int n_basis = elem.get_num_basis();
9    for (int i = 0; i < n_basis; ++i)
10   {
11     // do something with the reference basis
12   }
13   auto loc_dofs = elem.get_local_to_global();
14 }
15
16 for(auto elem : *phys_space)
17 {
18   int n_basis = elem.get_num_basis();
19   for (int i = 0; i < n_basis; ++i)
20   {
21     // do something with the physical basis
22   }
23 }
```

LISTING 5

Uses of element iterators on `igatools` containers. We iterate on the elements of a `CartesianGrid`, a `ReferenceSpace` and a `PhysicalSpace`. Notice the consistent treatment that iterators allow to access different data. We assume that `grid`, `ref_space` and `phys_space` were defined as shown in listings 1, 2 and 4 respectively.

3.6.1. Cache mechanism. To gain computational efficiency it is important to pre-compute and store quantities that are going to be used several times. In general, to gain in computational efficiency the code becomes more complex and difficult to understand and debug. A cache is a (smart) piece of memory that depending on the desired quantities to be evaluated it precomputes and stores commonly used computations to increase efficiency. In `igatools` this cache mechanism is implemented and

handled by the `ElementIterator`, which allows an organized and uniform design for the caches. In order to work with the cache mechanism, the clean interface of the range based `for` loop shown in Listing 5 needs to be slightly modified. In Listing 6 we show the actual code for the assembling of the stiffness matrix (using the efficient cache mechanism) for a Poisson's problem.

```
1  void PoissonProblem::assemble_matrix()
2  {
3    ValueFlags flag = ValueFlags::gradient | ValueFlags::w_measure;
4
5    auto elem = space->begin();
6    auto end  = space->end();
7    elem->init_cache(flag, quad);
8    for(; elem != end; ++elem)
9    {
10     elem->fill_cache();
11     loc_matrix = 0;
12     auto w_meas =  elem->get_w_measures();
13     for (int i = 0; i < n_basis; ++i)
14     {
15       auto grd_phi_i = elem->get_basis_gradients(i);
16       for (int j = 0; j < n_basis; ++j)
17       {
18         auto grd_phi_j = elem->get_basis_gradients(j);
19         for (int qp = 0; qp < n_qp; ++qp)
20         {
21           loc_matrix(i, j) +=
22              scalar_product(grd_phi_i[qp], grd_phi_j[qp]) * w_meas[qp];
23         }
24       }
25     }
26     auto loc_dofs = elem->get_local_to_global();
27     matrix->add(loc_dofs, loc_dofs, loc_matrix);
28   }
29 }
```

LISTING 6

Code for the assembling of the stiffness matrix for the Poisson problem (cf. §6.1. In order to use the efficient cache mechanism the range based `for` loops of Listing 5 needs to be modified, by adding the lines 5, 6, 7 and 10. Notice that `init_cache()` is execute one time and `fill_cache()` for each element.

3.7. Global quantities: fields, operators and the linear algebra. We have discussed the local handling of basis functions to assemble the local contribution of operators. The global quantities, such as fields and global operators, are stored in containers of the linear algebra system (e.g. global vectors and matrices). They are assembled from the local element contributions through the bookkeeping mechanism kept by the space. For each element, the non-zero basis functions have a local index for which space knows the corresponding global index. The query for this information is obtained through the element iterator through the `get_local_to_global()` function Listing 6, line 26). Almost any of the many excellent linear algebra systems could be easily plugged into the proposed design. From a bias preference of the original developers (and for the sake of selecting one), in `igatools` we have used the linear (and non-linear) algebra provided by some packages of the `Trilinos` library [29]. `Trilinos`, among other things, provides state of the art storage for distributed vectors and matrices as well as a complete collection of linear and non linear solvers. In order

to have a consistent style with the other pieces of the library, `igatools` provides simplified wrappers to basic `Trilinos` vectors, matrices and linear solvers. At the same time we do not restrict the more advanced and powerful use that can be obtained by dealing directly with the `Trilinos` object.

4. `igatools` implementation standards. The effort in `igatools` has been directed to develop a high quality implementation of the isogeometric software design presented in §3. To partially justify the statement, in this section we briefly describe some of the software engineering tools and development model we have adopted.

4.1. Development infrastructure. The development of `igatools` is supported by today's standard sofware engineering tools for that purpose. We use the distributed version control system and source code management `git`[3], and the bug tracking system `Trac`[4]. For managing the build process, we use `CMake`[5], allowing the software compilation process using a simple platform and compiler independent configuration files. Most importantly, we strive for high standards for documentation and user support. The documentation can be divided in three level: this paper, that describes the design concepts and how they work together; the tutorial examples provided with the source to give a hand-on introduction to use the library; and a reference manual generated by processing the in-source documentation with `Doxygen`[6]. For community support we have a user group[7] for discussion between the `igatools` developers and users and a development group[8] where the `igatools` developers discuss about new features, library design, implementation details, discovered bugs and possible remedies. We also maintain a wiki page [9] working as the webpage to the world of the library.

4.2. Development model (testing and debugging). `igatools` has hundreds of unit tests (small programs) that are run automatically using `CDash/CTest`[10] to verify that new changes are not breaking working features.

We use and advocate the use of the test driven development model [8]. This basically means that we write a unit test for the feature we want to implement or fix, the test initially fails, then we write code until the test passes, from that moment on the test is executed automatically. Now we can refactor the code (clean, remove any duplication, rename variables and methods, optimize, etc.) being confident that the new code is not damaging any existing functionality. In addition to the unit tests, `igatools` has automatic integration and validation tests (more cpu-time expensive than unit tests) in order to verify that the different modules interact correctly.

The library makes use *defensive programming* techniques to detect and easily find bugs at runtime, through the exception handling mechanism. We adopt two levels of checks. One, more expensive in terms of running time, but only active when `igatools` is compiled in *Debug* mode. We perform this kind of checks wherever there may be a chance of error, e.g. out-of-bound index for accessing to vectors elements, uninitialized objects, invalid object states, mismatching dimensions, etc. The second

[3] `http://www.git-scm.com`
[4] `http://trac.edgewall.org`
[5] `http://www.cmake.org`
[6] `http://www.doxygen.org`
[7] `https://groups.google.com/forum/#!forum/igatools-users`
[8] `https://groups.google.com/forum/#!forum/igatools-development`
[9] `http://code.google.com/p/igatools/w/list`
[10] `http://www.cdash.org`

level of checks is always active (both in *Debug* and in *Release* mode) and it is used for checking anomalies that may be introduced by the input data.

The typical workflow for a user of `igatools` would be to first write his code and test it with the library in *Debug* mode on a small-size problem. When this is working as expected and (virtually) bug-free, link the code with `igatools` in *Release* mode on a real-size problem.

4.3. Programming language. The main language used in the development of `igatools` is `C++11`[32]. It includes many additions to the previous standard `C++03`[31], mostly aimed to produce a code that is cleaner, safer, more efficient and easier to maintain. In particular, `igatools` makes heavy use of some of the language features somehow defining certain aspects of its programming style, as for example:

(i) **Smart pointers and create facility**. Objects in `igatools` can have big sizes and can be shared by different other objects. This sharing mechanism is handled through the use of smart pointers. They provide automatic memory management ensuring that the resource they control is automatically destroyed when its last (or only) owner is destroyed. Moreover, the `igatools` classes intended to be used through smart pointers provide a create facility for this purpose. For each constructor of the class there is a static function (called `create`) with the same arguments of the constructor that returns an instance of the class wrapped by a shared pointer. A code example showing the implementation and use of the technique is shown in Listing 7.

```
1  class MyClass
2  {
3  public:
4  // MyClass constructor using arg1 of type T1 and arg2 of type T2
5    MyClass(T1 arg1, T2 arg2);
6
7  // The static function create() uses the constructor
8  // MyClass(T1 arg1, T2 arg2) to build a MyClass
9  // instance wrapped by a std::shared_ptr
10   static std::shared_ptr<MyClass> create(T1 arg1, T2 arg2)
11   {
12      return std::make_shared<MyClass>( MyClass(arg1, arg2) );
13   }
14 };
15
16 void foo(T1 a, T2 b)
17 {
18   // Create an instance of MyClass wrapped by a std::shared_ptr
19   std::shared_ptr<MyClass> my_obj = MyClass::create(a, b);
20 }
```

LISTING 7

Create facility. For each class that is intended to be used through a smart pointer, `igatools` provides a create facility. The static function `create` returns an instance of the class wrapped by a `std:shared_ptr`. The function `foo` shows how the pointer is created.

(ii) **Template types (using and auto).** `C++11` provides better management of templates than `C++03` and `igatools` extensively uses *templates* in order to attain efficiency and the dimension independent paradigm. One drawback of using templates is the fact that the template arguments may become inhumanly difficult to express. We solve this problem with the `C++11 using` alias facility, that allows to create human readable aliases for the necessary types. Related is the automatic type

inference facility `auto` that deduces the type of an explicit initialization. An appropriate combination of `using`, `create` and `auto` results in a much more simple and human comprehensible syntax, as shown in Listing 8, creating a certain style to code with `igatools`.

```
void brute_force_syntax()
{
  std::shared_ptr<CartesianGrid<dim>> grid =
    std::make_shared<CartesianGrid<dim>>(CartesianGrid<dim>(4));
  std::shared_ptr<Mapping<dim>> map =
    std::make_shared<BallMapping<dim>>(BallMapping<dim>(grid));
  std::shared_ptr<BSplineSpace<dim,1,1>> ref_space =
    std::make_shared<BSplineSpace<dim,1,1>>(BSplineSpace<dim,1,1>(3,grid));
  std::shared_ptr<PushForward<Transformation::h_grad,dim>> push_fw =
    std::make_shared<PushForward<Transformation::h_grad,dim>>(
      PushForward<Transformation::h_grad,dim>(map));
  std::shared_ptr<PhysicalSpace<BSplineSpace<dim,1,1>,
    PushForward<Transformation::h_grad,dim>>> space =
      std::make_shared<PhysicalSpace<BSplineSpace<dim,1,1>,
        PushForward<Transformation::h_grad,dim>>>(
          PhysicalSpace<BSplineSpace<dim,1,1>,
            PushForward<Transformation::h_grad,dim>>(ref_space,push_fw));
}

void human_syntax()
{
  using Grid     = CartesianGrid<dim>;
  using Map      = Mapping<dim>;
  using RefSpace = BSplineSpace<dim, 1, 1>;
  using PushFw   = PushForward<Transformation::h_grad, dim>;
  using Space    = PhysicalSpace<RefSpace, PushFw>;

  auto grid      = Grid::create(4);
  auto map       = BallMapping<dim>::create(grid);
  auto ref_space = RefSpace::create(3,grid);
  auto push_fw   = PushFw::create(map);
  auto space     = Space::create(ref_space, push_fw);
}
```

LISTING 8

Library facilities `using`, `auto` and `create`. The two functions perform the same operations, but the second one uses the combination of the `using`, `auto` and `create` facilities advocated by the `igatools` programming style, resulting in a more human readable syntax than the brute force counterpart.

5. `igatools` practical features. Besides implementing the novel object oriented design presented in this work `igatools` provides features that make it useful and attractive for practical applications.

5.1. Input/output facilities. It is important for the library to use NURBS geometries generated by other software. At the moment there is no scientific computing dedicated format for isogeometric type of spaces. The developers of `igatools` and GeoPDEs in a joint effort defined an Extensible Markup Language (XML) format to describe such geometries. XML is an extremely flexible format that is supported by wide number of libraries and applications. The `igatools` input XML specification can be found in in the library wiki page [11]. Thus `igatools`, through it `IgReader`

[11] `https://code.google.com/p/igatools/wiki/InputSpecification_2_0`

class (see Listing 3, lines 8 and 9) supports importing geometries in this data format. Utility scripts to convert geometries generated with the `MATLAB` NURBS toolbox[12] to the XML format are distributed with the library sources.

Similarly important for the library is to provide a useful mechanism to analyze and visualize the computational results. `igatools` generates output for this purpose through its `Writer` class (see Listing 15). So far the writer provides output in `vtk` format [37] which can be visually processed (among other things) with Paraview [40].

5.2. Interaction with CAGD software. One of the most interesting features of isogeometric analysis is the integration between design and analysis, provided in a natural mathematical framework. In the context of `igatools` this can be cleanly achieved by implementing a `Mapping` specialization (see §3.5.1) that internally use all geometric features of a modern CAD system, provides a clear interface to the user without him worrying about the implementation details. Now, after a decade from the introduction of IGA there is no dedicated CAD software that directly generates a 3D analysis suitable geometry, this being one of the limiting issues in the use of the IgM in the industrial community. One reason is that most CAD software when dealing with a 3D geometries, only model the boundary surfaces of the represented domain. An exception is the geometric modeler `IRIT` [13] developed by G. Elber at Technion, that natively handles trivariate volumes. We have tested the integration of `igatools` with this particular geometric modeler, by implementing a `Mapping` specialization (`MappingIRIT`) that internally uses the `IRIT`'s routines and data structures for the evaluation of the geometric quantities required by `Mapping`. This `MappingIRIT` class is used for the geometry in the non-linear elasticity example presented in §6.4.

6. Examples. In this section we illustrate the library flexibility to naturally handle and adapt to different situations. We do this by showing its use to solve typical application problems[14] We start with a rather detailed description for the implementation of a simple Poisson equation. This is the de facto prototype example in any Galerkin numerical method that we also employ to describe the basic building blocks typical to the use of the library and to which we refer to in the subsequent examples. We continue with a tiny adaptation to the Poisson problem code that solves the Laplace-Beltrami problem on a manifold. Then we illustrate the handling of vector-valued and many spaces problems by discussing the implementation of the Stokes equation. Finally, we present a non-linear elasticity problem showing the flexibility of the library to cleanly interact with other software packages.

Notice that in the snippets of code listed in the section some non-relevant pieces are omitted (or more precisely replaced by a comment of the form "`//...`") to help with the understanding of the main pieces. Also as we move on in the examples, less details are provided as we assume they can be transferred from the previous examples.

6.1. Poisson Equation. The Poisson problem consists in finding $u : \Omega \subset \mathbb{R}^d \to \mathbb{R}$ such that

$$-\Delta u = f \text{ in } \Omega, \quad u = g \text{ on } \partial\Omega. \tag{6.1}$$

It can be rewritten in weak form as: find $u = u_0 + u_g$, with $u_0 \in H^1_0(\Omega)$ and u_g the lift of g, such that $\mathcal{A}u(v) = F(v)$ for all $v \in H^1_0(\Omega)$, where $\mathcal{A}u(v) = \int_\Omega \nabla u \cdot \nabla v$

[12] `http://www.aria.uklinux.net/nurbs.php3`
[13] `http://www.cs.technion.ac.il/~irit/`
[14] An archive file with the `igatools` version used in these examples is in the supplementary material

and $F(v) = \int_\Omega fv$. The IgM for this problem (cf. §2.1) requires: the definition of the isogeometric space $\mathbb{V}(\Omega)$; the assembling of the stiffness matrix $\boldsymbol{K}$ and right hand side $\boldsymbol{F}$ by adding the element contributions; and the solving of the linear system $\boldsymbol{KU} = \boldsymbol{F}$.

Below we describe a possible approach for the implementation of this problem with `igatools`. The full running code for this problem is part of the tutorial distributed with the library. To be concrete, we choose: the physical domain Ω to be a d-dimensional spherical sector for $d = 1, 2$ and 3; a homogeneous boundary condition; and a constant source of 5. The code would look practically the same if we decide to use a CAD geometry for the domain, a non-homogeneous boundary function and/or a non-constant source term.

In order to undertake this problem, a class for it (see Listing 9) is defined, its public interface provides two members, the constructor `PoissonProblem` and the `run()` function. Within the private members one can identify the declaration of objects for the space and the linear algebra. We can also see the `using` keyword to create short

```
template<int dim>
class PoissonProblem
{
public:
  PoissonProblem(TensorSize<dim> &n_knots, int deg);
  void run();

private:
  void assemble();
  void solve();
  void output();

private:
  using RefSpace = BSplineSpace<dim>;
  using PushFw   = PushForward<Transformation::h_grad, dim>;
  using Space    = PhysicalSpace<RefSpace, PushFw>;

  shared_ptr<CartesianGrid<dim>>  grid;
  shared_ptr<RefSpace>            ref_space;
  shared_ptr<Mapping<dim>>        map;
  shared_ptr<Space>               space;

  // ... quadratures defined

  ConstantFunction<dim> f;
  ConstantFunction<dim> g;

  shared_ptr<Matrix> matrix;
  shared_ptr<Vector> rhs;
  shared_ptr<Vector> solution;
};
```

LISTING 9

Class to approximate the solution of a Poisson problem. The public interface provides the constructor that prepares the object and the function `run()` that computes and plots the solution. In the private members we can find objects to handle the space and the linear algebra.

alias for the usually long templated types and the use of smart pointers (`shared_ptr`), both common techniques within the programming style of `igatools` (cf. §4.3). For the actual solving of the problem we define objects of `PoissonProblem` type in the program's `main()` (see Listing 10). Here we can see how the dimension independent template technique (cf. §3.3) is put to use. We are solving the Poisson problem in 1, 2

```
1  int main()
2  {
3    // ... n_knots and deg defined
4    PoissonProblem<1> poisson_1d(deg, {n_knots});
5    poisson_1d.run();
6
7    PoissonProblem<2> poisson_2d(deg, {n_knots, n_knots});
8    poisson_2d.run();
9
10   PoissonProblem<3> poisson_3d(deg, {n_knots, n_knots, n_knots});
11   poisson_3d.run();
12   // ...
13 }
```

LISTING 10

Main routine to solve the Poisson problem for different domain dimensions. For each dimension 1, 2 *and* 3 *we construct the problem and then assemble, solve and output the result by calling the function* `run()`.

and 3 dimensions with the same templated code. As a matter of fact, with the same code we could solve this problem in 4 dimensions if we wish to. Each object definition (lines 4, 7 and 10) calls the constructor that prepares the object to be ready to use and then we call the object member function `run()` that computes the solution. The

```
1  void PoissonProblem<dim>::run()
2  {
3    assemble();
4    solve();
5    output();
6  }
```

LISTING 11

The `run` *function is the public interface that solves the Poisson problem by assembling and solving the discrete linear system and saving the solution in graphical format.*

function `run()` (see Listing 11) is just a high level interface that calls the private members `assemble()`, `solve()` and `output()` that will do the actual computational work and are explained later on. After executing this program, graphical output files (in `vtk` format) for the different dimensions with the problem solution are saved. Figure 6.1 shows these files.

Now we explain the implementation of the different member functions of the `PoissonProblem` class. The constructor (see Listing 12) is in charge of making the object ready to be used. So it creates a physical space `space` by first creating a `grid`, a reference space `ref_space`, a mapping `map` and a push forward.

The `assemble()` routine assembles the global stiffness matrix `matrix` and global right hand side `rhs` by adding the element contributions. In `igatools` we use the element iterator (§3.6) for this purpose. We have already seen in Listing 6 the assemble of the global matrix, here in Listing 13 we include the assembling of the right hand side and the treatment of the boundary conditions. The later is treated with a projection, as described in section 3.2, involving two steps: first we L^2 project the boundary function g onto the trace space (line 27) and then we enforce these values as constraints in the linear system (line 28). Recall that a projection is necessary as splines are not interpolatory.

As discussed in §3.7, `igatools` relies on external algebra packages for solving the

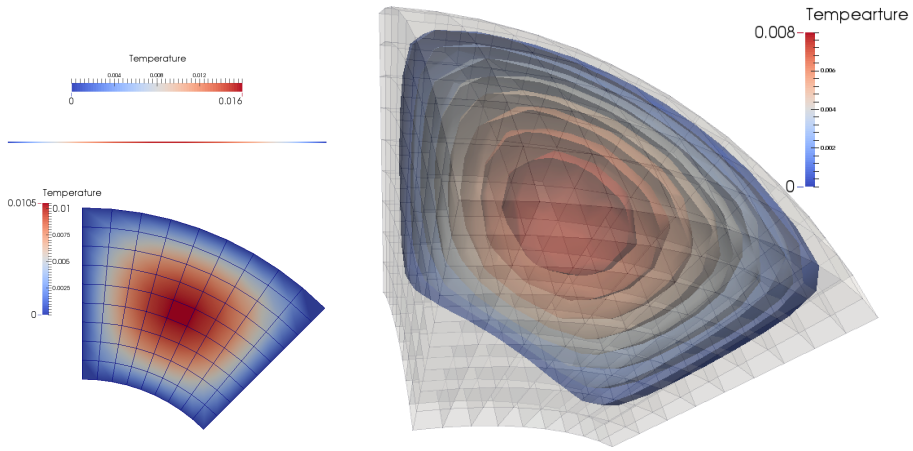


FIG. 6.1. *Plot of the solutions to the Poisson problem with homogeneous boundary conditions and constant source term on a spherical sector in* 1, 2 *and* 3 *dimensions. Notice that the three plots were generate with the same code, by only changing the dimension template argument.*

```
1  PoissonProblem(int deg, TensorSize<dim> &n_knots)
2    :
3    // ... initilize elem_quad and face_quad
4    f({5.}), g({0.})
5  {
6    // ... define box, a bounding box for the grid
7
8    grid      = CartesianGrid<dim>::create(box, n_knots);
9    ref_space = RefSpace::create(deg,grid);
10   map       = BallMapping<dim>::create(grid);
11   space     = Space::create(ref_space, PushFw::create(map));
12
13  // ... initialize matrix, rhs and solution
14 }
```

LISTING 12

Poisson's problem constructor. It construct the physical space, for which it needs to construct a reference space and a push-forward which in turn requires the construction of a map and a cartesian grid. It also initializes the quadrature schemes, the linear algebra and the source and boundary functions.

global system. The function `solve()` in Listing 14 defines a `Solver` object which is a wrapper class provided by `igatools` for basic use of `Trilinos` solvers. The solution of the system is returned in `solution`. The lines in this listing look simple enough to be self-explanatory.

To manage the graphical output (§5.1), `igatools` provides a `Writer` class that can be used to handle geometries and fields that are to be saved in a graphical output format (for now a `vtk` file). In Listing 15 a `Writer` object associated with the geometry of `map` is defined (line 4), then the solution field is added to it (line 5), and finally `writer` is asked to save the geometry and the solution field to disk (line 6).

6.2. A surface example. The code of §6.1 can be trivially modified to solve a partial differential equation on a surface. `igatools` can naturally handle manifold domains such as surfaces or curves.

Let Γ be a d-dimensional manifold, and consider the Laplace-Beltrami equation:

```cpp
void PoissonProblem<dim>::assemble()
{
  // ... same code as listing 6 we are only adding
  // ... the pieces for the right hand side

  for (; elem != elem_end; ++elem)
  {
    loc_rhs = 0;
    auto points  = elem->get_points();
    auto phi     = elem->get_basis_values();
    auto f_values = f.evaluate(points);

    for (int i = 0; i < n_basis; ++i)
    {
      // ... matrix assembling (Listing 6)

      auto phi_i = phi.get_function(i);
      for (int qp = 0; qp < n_qp; ++qp)
      loc_rhs(i) +=
        scalar_product(phi_i[qp], f_values[qp]) * w_meas[qp];
    }
    // ...
    rhs->add(loc_dofs, loc_rhs);
  }

  // Apply dirichlet boundary condition
  project_boundary_values(g, space, face_quad, dirichlet_id, dof_values);
  apply_boundary_values(dof_values, *matrix, *rhs, *solution);
}
```

Listing 13

Assembling of the global system for Poisson problem. The matrix assembling was already shown in Listing 6, here we only show the additional code required to assemble the right hand side and take care of the dirichlet boundary constraints.

```cpp
void PoissonProblem<dim>::solve()
{
  Solver solver(SolverType::CG);
  solver.solve(*matrix, *rhs, *solution);
}
```

Listing 14

Solver for the Poisson problem. We use the simple wrapper `igatools` provides to the `Trilinos` solvers.

```cpp
void PoissonProblem<dim>::output()
{
  // ... define filename
  Writer<dim> writer(map);
  writer.add_field(*space, *solution, "Temperature");
  writer.save(filename);
}
```

Listing 15

Saving the solution of Poisson problem in a graphical output.

find $u : \Gamma \to \mathbb{R}$ such that

$$-\Delta_\Gamma u = f \text{ in } \Gamma, \quad u = g \text{ on } \partial\Gamma, \tag{6.2}$$

which can be rewritten in weak form as find $u = u_0 + u_g$, with $u_0 \in H^1_0(\Gamma)$ and u_g the lift of g, such that $\mathcal{A}u(v) = F(v)$ for all $v \in H^1_0(\Gamma)$, where $\mathcal{A}u(v) = \int_\Gamma \nabla_\Gamma u \cdot \nabla_\Gamma v$ and $F(v) = \int_\Gamma fv$. Except for the fact that the differential operators are surface operators, one can see the similarity of this problem with equation (6.1). The library understands manifolds, thus when the physical space has codimension different from 0, `igatools` understand that gradients refers to surface gradients. In this way, minor modifications to the Poisson problem code of §6.1 allow to solve the Laplace-Beltrami problem.

More precisely, without loss of generality, let assume we want to solve equation (6.2) on a d-dimensional spherical piece of codimension 1. It only requires minor changes on a few lines in the problem class, first the definitions of `codim` and `space`

```
static const int codim = 1;
static const int spacedim = dim + codim;
```

and then some small changes in lines 15, 20, 25 and 26 of Listing 9 by the following four lines in the given order

```
using PushFw = PushForward<Transformation::h_grad, dim, codim>;
shared_ptr<Mapping<dim, codim>> map;
ConstantFunction<spacedim> f;
ConstantFunction<spacedim> g;
```

The only change required in the constructor is to replace line 10 of Listing 12 by

```
map = SphereMapping<dim>::create(grid);
```

In Figure 6.2 we plot the isogeometic approximate solution to the Laplace-Beltrami problem over a spherical piece given by a spherical coordinate map.

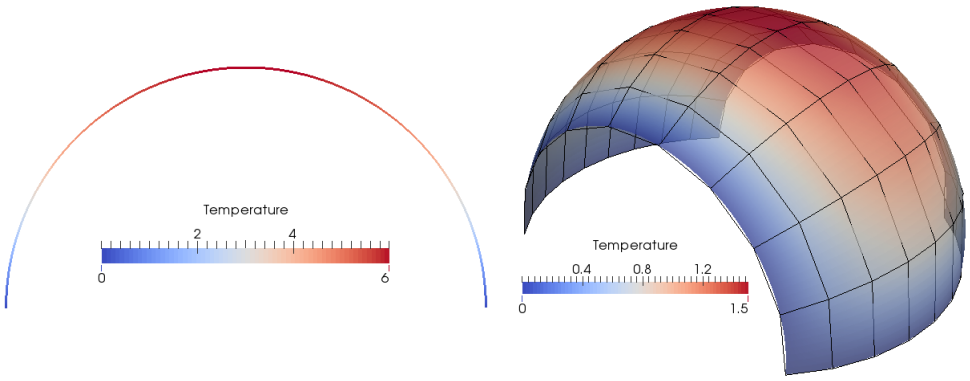


FIG. 6.2. *Plot of the solution to the Laplace-Beltrami problem with homogenous boundary condition and constant source term on a piece of a d-sphere, for* $d = 1, 2$.

6.3. Navier-Stokes Equations. An incompressible viscous fluid flowing in a laminar regime is modeled by the Navier-Stokes equations

$$\rho\,(\partial_t \mathbf{u} + (\nabla \mathbf{u})\mathbf{u}) - \mu \Delta \mathbf{u} + \nabla p = \mathbf{f} \qquad \text{and} \qquad \operatorname{div} \mathbf{u} = 0 \qquad \text{in } \Omega \times (0, T),$$

supplied with appropriate initial and boundary conditions. Here $\mathbf{u}$ is the fluid velocity field and p its pressure. A distinctive attribute of these equations is the combination of hyperbolic and parabolic terms [28]. The key to the application of the Galerkin method to the weak form of these equations lays into its parabolic term with the

incompressibility constraint leading to a saddle point problem. In particular, the spaces for velocity and pressure have to satisfy the well-known inf-sup condition [11]. We assume that $\mathbb{V} \subset H^1(\Omega)^d$ and $\mathbb{Q} \subset L^2(\Omega)$ is a pair of spaces for the velocity and pressure that satisfy this condition, exactly what subspaces are taken depends on the type of boundary conditions. The discrete problem is obtained by considering discrete spaces $\mathbb{V}_h \subset \mathbb{V}$ and $\mathbb{Q}_h \subset \mathbb{Q}$ for the velocity and pressure respectively, and can be written in matrix form as:

$$\begin{pmatrix} \boldsymbol{A} & \boldsymbol{B}^t \\ \boldsymbol{B} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{U} \\ \boldsymbol{P} \end{pmatrix} = \begin{pmatrix} \boldsymbol{F} \\ \boldsymbol{0} \end{pmatrix}, \tag{6.3}$$

where the submatrix $\boldsymbol{A}$ corresponds to the terms involving the fluid velocity and $\boldsymbol{B}$ corresponds to the mixed term. For the discretization of this problem we use two spaces: one scalar-valued (for the pressure) and the other vector-valued (for the velocity). Most importantly, they should form a stable pair (i.e. satisfy a discrete inf-sup condition).

Below we illustrate the use of `igatools` to treat the two coupled spaces by listing the relevant pieces of code for a Stokes type of problem. Listing 16 shows part of the `StokesProblem` class declaration. In particular, notice the scalar-valued space type `PreSpace` being similar to the space of section 6.1 while for the `VelSpace` type the addition of the second template parameter renders it into a vector-valued space.

```
1  class StokesProblem
2  {
3  public:
4    StokesProblem(int deg, int n_knots, int reg);
5    void run();
6
7  private:
8    void assemble_Bt();
9    // ...
10 private:
11   using PreSpace = BSplineSpace<dim>;
12   using VelSpace = BSplineSpace<dim, dim>;
13
14   shared_ptr<PreSpace> pre_space;
15   shared_ptr<VelSpace> vel_space;
16   // ...
17   shared_ptr<Matrix> Bt;
18 };
```

LISTING 16

Sketch of a class to solve the Stokes problem. Observe the declaration of two spaces one vector-valued for the velocity and one scalar-valued for the pressure.

For the purpose of this example we consider the Taylor-Hood type of spaces adapted to the isogeometric setting [4, 14]. In which the pressure and velocity spaces share the same global regularity and the velocity degree is one more that of the pressure. These spaces are constructed in the `StokesProblem` constructor (see Listing 17). Here, given the degree for the pressure space and the global regularity as arguments, the constructor builds the multiplicity vectors `pre_mult` and `vel_mult` and the degrees `pre_deg` and `vel_deg` for both spaces, which are created in lines 17 and 18. Recall that to have a global regularity r in a spline space of degree d the multiplicity of the interior knots must be $d-r$ and the end knots to be interpolatory require a multiplicity $d+1$.

```
StokesProblem(int deg_p, int n_knots, int reg)
{
  // ... define pre_mult, vel_mult, pre_deg and vel_deg
  int deg_v = deg_p + 1;
  vector<int> mult_p(n_knots, deg_p - reg);
  vector<int> mult_v(n_knots, deg_v - reg);
  mult_p[0] = mult_p[n_knots-1] = deg_p+1;
  mult_v[0] = mult_v[n_knots-1] = deg_v+1;

  pre_mult.fill(mult_p);
  vel_mult.fill(mult_v);

  pre_deg.fill(deg_p);
  vel_deg.fill(deg_v);

  auto grid = CartesianGrid<dim>::create(n_knots);
  pre_space = PreSpace::create(pre_deg, grid, pre_mult);
  vel_space = VelSpace::create(vel_deg, grid, vel_mult);

  Bt = Matrix::create(*vel_space, *pre_space);
}
```

LISTING 17

Constructor for Stokes class. The scalar and vector-valued spaces for the isogeometric version of the Taylor-Hood elements are created. The pressure and velocity spaces share the same global regularity and the velocity degree is one more that of the pressure.

An interaction between the velocity and pressure spaces is required, for example, in the assembling of the mixed term $\boldsymbol{B}^t$. This involves the simultaneous access to basis functions from both spaces over the same element and quadrature points. Listing 18 illustrates how this is attained in `igatools` using two element iterators. We use the iterators `pre_el` and `vel_el`, one for the pressure and one for the velocity space in a common `for` loop, with a simultaneous increment. This works because both spaces are defined on the same `CartesianGrid`, so the increment operator ensures that both iterators traverse the elements in synchronization.

We use the above code to solve the time dependent Navier-Stokes equations on the domain shown in Figure 6.3. This geometry is inspired in vascular valves leaflets but there is no claim to perform a simulation that has a physical significance. Our main motivation is using our software in a non-trivial geometry. Figure 6.4 shows some of the streamlines and presures at different time intervals. The time integration is performed using a first order operator splitting technique [28]. The geometry size

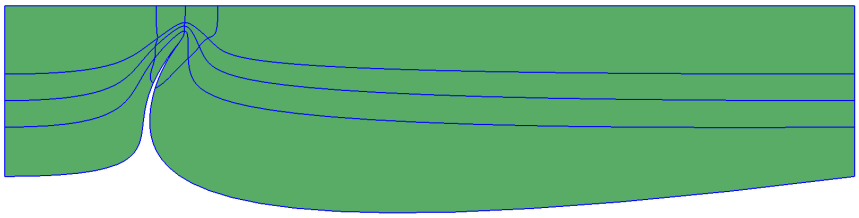


FIG. 6.3. *Domain given by a quintic spline deformation of the unit square inspired in a vascular leaflet.*

is a 1 by 5 box. The leaflet tip is modeled with three control points separated by a

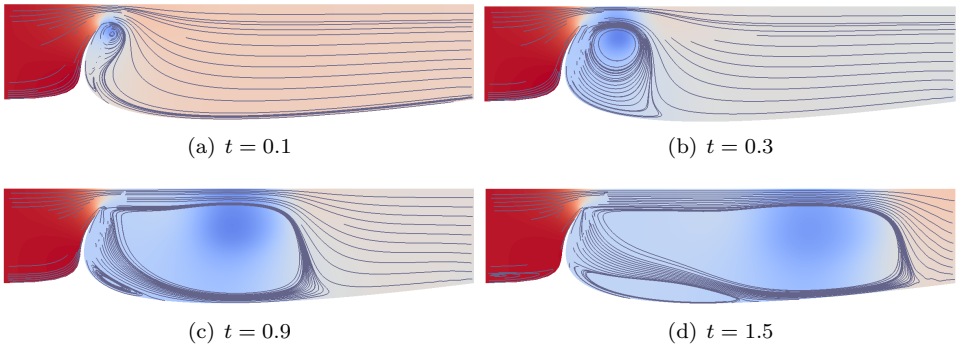
```
void StokesProblem<dim>::assemble_Bt()
{
  // ... define  vel_n_basis, pre_n_basis, vel_loc_dofs, pre_loc_dofs...
  DenseMatrix loc_mat(vel_n_basis, pre_n_basis);
  auto vel_el = vel_space->begin();
  auto pre_el = pre_space->begin();
  auto end_el = vel_space->end();

  ValueFlags vel_flag = ValueFlags::divergence|ValueFlags::w_measure;
  ValueFlags pre_flag = ValueFlags::value;
  vel_el->init_cache(vel_flag, elem_quad);
  pre_el->init_cache(pre_flag, elem_quad);

  for (; vel_el != end_el; ++vel_el, ++pre_el)
  {
    loc_mat = 0.;
    vel_el->fill_cache();
    pre_el->fill_cache();

    auto q      = pre_el->get_basis_values();
    auto div_v  = vel_el->get_basis_divergences();
    auto w_meas = vel_el->get_w_measures();

    for (int i = 0; i < vel_n_basis; ++i)
    {
      auto div_i = div_v.get_function(i);
      for (int j = 0; j < pre_n_basis; ++j)
      {
        auto q_j = q.get_function(j);
        for (int qp = 0; qp < n_qp; ++qp)
        loc_mat(i, j) -= scalar_product(div_i[qp], q_j[qp]) * w_meas[qp];
      }
    }
    vel_loc_dofs = vel_el->get_local_to_global();
    pre_loc_dofs = pre_el->get_local_to_global();
    Bt->add(vel_loc_dofs, pre_loc_dofs, loc_mat);
  }
}
```

LISTING 18

Assembling B^t involves simultaneous access to the basis functions of both the pressure and velocity spaces. `igatools` approach is the use of two element iterators that are incremented simultaneously.

distance of 10^{-3}. The geometry mapping as well as the Taylor-Hood spaces have a C^4 global regularity, with degree 5 and 6 for the pressure and velocity respectively. Figure 6.4 shows the resulting pressure and streamlines at different time frames. Boundary conditions are no-slip at the bottom side, unitary horizontal velocity at the inlet, symmetry on top, and no stress at the outlet. The main vortex originates, as expected, from the leaflet tip. As time evolves, a secondary vortex originates from the bottom of the leaflet.

6.4. Non-linear elasticity. The library design provides a flexible access to the isogeometric spaces. This flexibility and encapsulation permit easy integration not only with different algebra systems but also with other scientific packages for specific purposes. This section presents an example in which we have integrated `igatools` with `FEBio` [34], a non-linear finite element software for biomechanics.



(a) $t = 0.1$ (b) $t = 0.3$

(c) $t = 0.9$ (d) $t = 1.5$

FIG. 6.4. *A time dependent Navier-Stokes simulation for the rigid leaflet geometry of Figure 6.3. For illustrative reasons we decided to reproduce the shape of a vascular leaflet without any particular modelling motivation, we were interested in the vortex shedding by this geometry. In particular we notice the main vortex started at the tip of the leaflet, and a second one downstream at the bottom of the leaflet. The colormap represents the pressure.*

The behavior of an elastic body is characterized by the equations

$$\text{div}\,\boldsymbol{\sigma} + \boldsymbol{f} = \mathbf{0} \text{ in } \Omega; \quad \boldsymbol{\sigma}\boldsymbol{n} = \boldsymbol{t} \text{ on } \Gamma_\sigma; \quad \boldsymbol{u} = \bar{\boldsymbol{u}} \text{ on } \Gamma_u, \tag{6.4}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor; $\boldsymbol{f}$ and $\boldsymbol{u}$ are the body force and displacement vectors, respectively; $\boldsymbol{n}$ is the outter surface normal; and $\boldsymbol{t}$ is the surface traction vector. For this example we assume a non-linear constitutive relation between $\boldsymbol{\sigma}$ and $\boldsymbol{u}$ that leads to a non-linear problem.

The variational form of the equation (6.4) can be written as

$$\delta W(\varphi, \delta\boldsymbol{v}) = \int_\Omega \boldsymbol{\sigma} : \nabla\delta\boldsymbol{v}\mathrm{d}\Omega - \int_\Omega \boldsymbol{f}\cdot\delta\boldsymbol{v}\mathrm{d}\Omega - \int_{\Gamma_\sigma} \boldsymbol{t}\cdot\delta\boldsymbol{v}\mathrm{d}\Gamma = 0\,,$$

being φ the map such that the current (deformed) configuration is expressed as $\Omega = \varphi(\hat{\Omega})$. At the discrete level this equation can be expressed in a vector form as $\delta W(\varphi, \delta\boldsymbol{v}) = \delta\mathbf{v}^{\mathrm{T}}\mathbf{R} = 0$, where $\mathbf{R}$ is the residual vector. Due to the fact that $\mathbf{R}$ depends non-linearly on the unknowns vector $\mathbf{u}$ it is needed to perform a linearization of this equation. The matrix form of the linearization is $\mathbf{K}(\varphi_k)\Delta\mathbf{u}_k = -\mathbf{R}(\varphi_k)$, being $\mathbf{K}$ the stiffness matrix (further details can be found in [12]). They are referred to as the deformed configuration of the body $\Omega_k = \varphi_k(\hat{\Omega})$. An iterative procedure in k is applied for achieving an equilibrium configuration φ_e such as $\mathbf{R}(\varphi_e) \approx \mathbf{0}$, for a given tolerance.

`igatools` has been integrated with `FEBio` in a minimally invasive way: only the input/output and the computation of the element right-hand side vector and stiffness matrix have been modified, meanwhile linear algebra, degrees of freedom, material level computations, etc., are still managed by `FEBio`. The `IGAElasticDomain` class (Listing 19) materializes this integration, being derived from the `FEBio`'s class `FEBioElasticDomain` it adds the isogeometric components such as the physical space. As it can be seen, a `NURBSpace` is used for the displacement. The final integration occurs in the functions for assembling the global stiffness matrix $\mathbf{K}$ and residual vector $\mathbf{R}$ by iterating over the elements and computing local contributions. For example, Listing 20 shows the assembling of $\mathbf{K}$ by computing the local matrices $\mathbf{K}^e$ given by

$$\mathbf{K}^e_{i,j} = \int_{\Omega_e} \nabla B_i : \boldsymbol{\mathcal{C}} : \nabla B_j \mathrm{d}\Omega + \int_{\Omega_e} \nabla B_i \cdot \boldsymbol{\sigma}\nabla B_j \mathrm{d}\Omega\,, \tag{6.5}$$

```
1  class IGAElasticDomain : public FEBioElasticDomain
2  {
3  public:
4    // ... public functions and members
5    using Space = PhysicalSpace<RefSpace, PForward>;
6
7  private:
8    // ... private functions and members
9    Space space;
10   void stiffness_matrix(FESolidSolver* psolver);
11   void residual(FESolidSolver* psolver);
12 };
```

LISTING 19

Class for the solution of the non-linear elasticity problem by means of the IgM integrated into `FEBio`*. The spaces and push forward types are shown, as well as the public functions for assembling the residual vector and the stiffness matrix.*

to be then added, in line 29, into a global `FEBio` matrix.

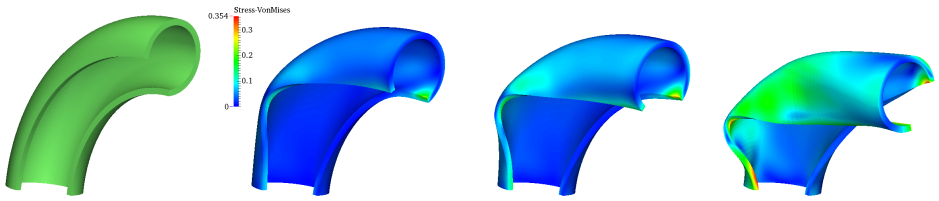
```
1  void
2  IGAElasticDomain::stiffness_matrix(FESolidSolver* psolver)
3  {
4    // ... define loc_mat, elem, elem_end, init elem values with quadrature
5
6    for(; elem!= elem_end; ++elem)
7    {
8      // Elasticity and Cauchy stress tensors evaluated by FEBio
9      auto sigma = this->get_stress_tensor(elem);
10     auto c = this->get_elasticity_tensor(elem);
11
12     loc_mat = 0;
13     auto w_meas = elem->get_w_measures();
14     for (int i = 0; i < n_basis; ++i)
15     {
16       auto Dphi_i = elem->get_basis_gradients(i);
17       for (int j = 0; j < n_basis; ++j)
18       {
19         auto Dphi_j = elem->get_basis_gradients(j);
20         for (int qp = 0; qp < n_points; ++qp)
21         loc_mat += // Compute formula (6.5) using
22              // Dphi_i[qp], Dphi_j[qp], c[qp], sigma[qp], w_meas[qp]
23       }
24     }
25     auto loc_to_global      = this->get_local_to_global(elem);
26     auto nodal_connectivity = this->get_nodal_connectivity(elem);
27
28     // Add element contribution into global FEBio stiffness matrix.
29     psolver->AssembleStiffness(nodal_connectivity,
30                                loc_to_global, loc_mat);
31   }
32 }
```

LISTING 20

Implementation of the stiffness matrix for the nonlinear elasticity case. The local element matrices are assembled into the `FEBio` *expected structure.*

Finally, Figure 6.5 shows a simulation for the torsion of a curved tube with an open section. The geometry is given by a three dimensional single NURBS patch of

degree two and its material modeled by a Neo-Hookean formulation.



(a) The geometry map. (b) $t = 0.3$ (c) $t = 0.5$ (d) $t = 1.0$

FIG. 6.5. *A non-linear elasticity simulation for an open tube is shown. The map is a second order deformation of a square 6.5(a). In this simulation a rotation around the axis of the tube is prescribed in one of the ends of tube whereas the other is fixed. Finite deformations are considered for an hyperelastic material formulation. Colormap represents von Mises stress.*

7. Conclusions. In this work we have laid down the basis for the design of a software library for the isogeometric method. It required the formulation of an abstract definition of this method highlighting the relations between the mathematical concepts and the design blocks. We have programmed this design in C++11 as the open source software library `igatools`, where special effort has been directed towards a well documented, clean, and manageable source code. We have illustrated the flexibility of the library by showing several examples, including not only Poisson problem, but also nonlinear elasticity and the Navier-Stokes equations. These examples also allow us to see how the basic classes are put together mimicking the mathematical concepts of the isogeometric method. A careful judgment has been made in the use of virtual functions and templates to attain a human friendly interface while maintaining computational efficiency. The library is a continuously evolving project. At the time of this writing, support for hierarchical spline spaces as well as multi-patch spaces with regularity beyond C^0 are under development. We also plan to include tutorial examples using parallel assembling and solving (both MPI and TBB based). We believe that this work, along with the release of `igatools`, provides a useful contribution for solving problems from applications using the isogeometric method. It also provides a building platform for further development and testing of new ideas from the research community of isogeometric methods.

Acknowledgements. We would like to thank A. Buffa and G. Sangalli for feedback and suggestions and to the beta testers of `igatools` for their patience and feedback, in particular to A. Bressan, E. Brivadis, and R. Vázquez. We also thank the reviewers for their thoughtful comments and suggestions.

REFERENCES

[1] D. N. ARNOLD, R. S. FALK, AND R. WINTHER, *Finite element exterior calculus: from Hodge theory to numerical stability*, Bull. Amer. Math. Soc. (N.S.), 47 (2010), pp. 281–354.

[2] W. BANGERTH, R. HARTMANN, AND G. KANSCHAT, *deal.II – a general purpose object oriented finite element library*, ACM Trans. Math. Softw., 33 (2007), pp. 24/1–24/27.

[3] W. BANGERTH, T. HEISTER, AND G. KANSCHAT, `deal.II` *Differential Equations Analysis Library, Technical Reference.* `http://www.dealii.org`.

[4] Y. BAZILEVS, L. BEIRÃO DA VEIGA, J. A. COTTRELL, T. J. R. HUGHES, AND G. SANGALLI, *Isogeometric analysis: Approximation, stability and error estimates for h-refined meshes*, Math. Mod. Meth. Appl. S., 16 (2006), pp. 1031–1090.

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

[6] Y. Bazilevs, C. Michler, V. M. Calo, and T. J. R. Hughes, *Isogeometric variational multiscale modeling of wall-bounded turbulent flows with weakly enforced boundary conditions on unstretched meshes*, Comput. Methods Appl. Mech. Engrg., 199 (2010), pp. 780–790.

[7] Y. Bazilevs, C. Michler, V. M. Calo, and T. J. R. Hughes, *Isogeometric variational multiscale modeling of wall-bounded turbulent flows with weakly enforced boundary conditions on unstretched meshes*, Comput. Methods Appl. Mech. Engrg., 199 (2010), pp. 780–790.

[8] K. Beck, *Test Driven Development: By Example*, Addison-Wesley Longman, 2002.

[9] P. Becker, *Working draft, standard for programming language C++*, Tech. Report N3242=11-0012, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Feb. 2011.

[10] D. J. Benson, Y. Bazilevs, E. De Luycker, M.-C. Hsu, M. Scott, T. J. R. Hughes, and T. Belytschko, *A generalized finite element formulation for arbitrary basis functions: From isogeometric analysis to XFEM*, Int. J. Numer. Meth. Eng., 83 (2010), pp. 765–785.

[11] D. Boffi, F. Brezzi, and M. Fortin, *Mixed Finite Element Methods and Applications*, Springer Series in Computational Mathematics, Springer London, Limited, 2013.

[12] J. Bonet and R. D. Wood, *Nonlinear Continuum Mechanics for Finite Element Analysis*, Cambridge University Press, New York, NY, USA, 2 ed., 2008.

[13] M. J. Borden, M. A. Scott, J. A. Evans, and T. J. R. Hughes, *Isogeometric finite element data structures based on Bézier extraction of NURBS*, Int. J. Numer. Meth. Eng., 87 (2011), pp. 15–47.

[14] A. Bressan and G. Sangalli, *Isogeometric discretizations of the Stokes problem: stability analysis by the macroelement technique*, IMA J. Numer. Anal., (2012).

[15] A. Buffa, D. Cho, and G. Sangalli, *Linear independence of the T-spline blending functions associated with some particular T-meshes*, Comput. Methods Appl. Mech. Engrg., 199 (2010), pp. 1437–1445.

[16] A. Buffa, C. de Falco, and G. Sangalli, *IsoGeometric Analysis: stable elements for the 2D Stokes equation*, Int. J. Numer. Meth. Fl., 65 (2011), pp. 1407–1422.

[17] 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.

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

[19] J. A. Cottrell, T. J. R. Hughes, and Y. Bazilevs, *Isogeometric Analysis: Toward Integration of CAD and FEA*, Wiley, 2009.

[20] J. A. Cottrell, A. Reali, Y. Bazilevs, and T. J. R. Hughes, *Isogeometric analysis of structural vibrations*, Comput. Methods Appl. Mech. Engrg., 195 (2006), pp. 5257–5296.

[21] C. de Falco, A. Reali, and R. Vázquez, *GeoPDEs: A research tool for isogeometric analysis of PDEs*, Adv. Eng. Softw., 42 (2011), pp. 1020 – 1034.

[22] T. Dokken, T. Lyche, and K. F. Pettersen, *Locally refinable splines over box-partitions*, tech. report, SINTEF, February 2012.

[23] M. Dörfel, B. Jüttler, and B. Simeon, *Adaptive isogeometric analysis by local h-refinement with T-splines*, Comput. Methods Appl. Mech. Engrg., 199 (2009), pp. 264–275.

[24] J. A. Evans and T. J. R. Hughes, *Isogeometric divergence-conforming B-spline for the steady Navier–Stokes equations*, Math. Mod. Meth. Appl. S., 23 (2013), pp. 1421–1478.

[25] ———, *Isogeometric divergence-conforming B-splines for the unsteady Navier–Stokes equations*, J. Comput. Phys., 241 (2013), pp. 141–167.

[26] D. R. Forsey and R. H. Bartels, *Hierarchical B-spline refinement*, in SIGGRAPH '88 Proceedings of the 15th annual conference on Computer graphics and interactive techniques, 1988, pp. 205–212.

[27] C. Giannelli, B. Jüttler, and H. Speleers, *THB–splines: The truncated basis for hierarchical splines*, Compute. Aided Geometric D., 29 (2012), pp. 485–498.

[28] R. Glowinski, P. G. Ciarlet, and J. L. Lions, *Numerical Methods for Fluids*, vol. 3 of Handbook of numerical analysis, Elsevier, 2002.

[29] M. Heroux, R. Bartlett, V. E. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, *An Overview of Trilinos*, Tech. Report SAND2003-2927, Sandia National Laboratories, 2003.

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

[31] *ISO/IEC 14882:2003 – Programming languages – C++*, 2003.
[32] *ISO/IEC 14882:2011 – Information technology – programming languages – C++*, 2011.
[33] J. Kiendl, Y. Bazilevs, M.-C. Hsu, R. Wüchner, and K.-U. Bletzinger, *Kirchhoff-Love shell structures comprised of multiple patches*, Comput. Methods Appl. Mech. Engrg., 199 (2010), pp. 2403–2416.
[34] S. A. Maas, B. J. Ellis, G. A. Ateshian, and J. A. Weiss, *FEBio: Finite elements for biomechanics*, J. Biomed. Eng., 134 (2012).
[35] P. Monk, *Finite element methods for Maxwell's equations*, Numerical Mathematics and Scientific Computation, Oxford University Press, New York, 2003.
[36] L. A. Piegl and W. Tiller, *The NURBs Book*, Monographs in Visual Communication Series, Springer-Verlag GmbH, 1997.
[37] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object-oriented Approach to 3D Graphics*, Kitware, 2006.
[38] L. L. Schumaker, *Spline functions: basic theory*, Cambridge Mathematical Library, Cambridge University Press, Cambridge, third ed., 2007.
[39] M. A. Scott, X. Li, T. W. Sederberg, and T. J. R. Hughes, *Local refinement of analysis-suitable T-splines*, Comput. Methods Appl. Mech. Engrg., 213–216 (2012), pp. 206 – 222.
[40] A. H. Squillacote, *The ParaView Guide*, Kitware, 2007.
[41] A. Stepanov and M. Lee, *The standard template library*, tech. report, WG21/N0482, ISO Programming Language C++ Project, 1994.
[42] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 2000.
[43] A.-V. Vuong, C. Giannelli, B. Jüttler, and B. Simeon, *A hierarchical approach to adaptive local refinement in isogeometric analysis*, Comput. Methods Appl. Mech. Engrg., 200 (2011 Dec), pp. 3554–3567.
[44] O. Weeger, U. Wever, and B. Simeon, *Isogeometric analysis of nonlinear Euler Bernoulli beam vibrations*, Nonlinear Dynam., 72 (2013), pp. 813–835.