Abstract

The main goal of this project is the creation of a C++ library that generates a B-Spline functional space. Starting from the pre-existent code GeoPDEs (a free software coded in Octave developed at CNR in Pavia), the idea is to exploit the object oriented programming for the development of a useful tool for an IGA approach.

In this paper, we will present a brief introduction to Isogeometric Analysis with a specific interest in B-spline functions. We will also examine the structure of the code and its organization and we will present a simple example of how this library can be utilized.
Contents

1 Introduction to Isogeometric Analysis 3
  1.1 The main goal of IGA ................................. 3
  1.2 Isogeometric Analysis: a general framework ........ 3
  1.3 B-Spline functions ................................. 4
  1.4 The Poisson’s problem .............................. 7

2 The library 10
  2.1 General structure .................................. 14
    2.1.1 Geometry ..................................... 15
    2.1.2 Mesh .......................................... 15
    2.1.3 Space ......................................... 16
  2.2 Concrete class implementations ................... 16

3 Development and build system 20
  3.1 Autotools ........................................... 20
    3.1.1 Autoconf ..................................... 20
    3.1.2 Automake .................................... 22
  3.2 Linking Octave ..................................... 23

4 Conclusions 25
1 Introduction to Isogeometric Analysis

In this section we will just present some of the most relevant notions. Any reader interested in deepening the subject see [3].

1.1 The main goal of IGA

Over the past years there has been a rise in interest in the so called Isogeometric Analysis; this technique aims to unify the two independently born fields of finite element analysis (FEA) and of computer-aided design (CAD).

FEA and CAD often deal with the same geometries but they represent them in completely different ways; this was the fundamental gap that the IGA approach wanted to fill. The idea is to maintain the exact description of the geometry and to use the same class of functions most commonly used in CAD (such as Bezier curves, B-Splines and NURBS) and to use them for the PDE solution space. This leads to the so called isoparametric paradigm, which means that the space of the geometry and the discrete space of the shape functions coincide.

Since NURBS spaces include, as a special case, the polynomial spaces commonly used in the Finite Element Method, IGA results in a generalization of standard FEA with more regular functions. It has been proved that this higher regularity leads not only to a better representation of CAD geometries, but also to better approximation properties.

In this paper, we will focus our attention on B-spline functions which are the simplest way to approach Isogeometric Analysis. In the last few years different kinds of functions have been considered and one of the possible improvements to this library is, therefore, the extension to other functional spaces such as NURBS, T-splines or generalized B-splines.

It is interesting to observe that, however, the generality of the hierarchy of classes developed in this project can also be applied beyond the scope of IGA, for example to FEA on distorted meshes or Free-Form deformation methods.

1.2 Isogeometric Analysis: a general framework

When solving a PDE problem, IGA and FEA share the same approach. In both cases a Galerkin procedure is used: the equations are written in their variational formulations and the solution is sought in a finite dimensional space with good approximation properties. The main difference between the two methodologies is that in FEA the basis functions and the computational geometry are defined using piecewise polynomials, while in IGA the computational geometry is defined exactly, starting from the basis functions given by CAD.

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

$$F: \hat{\Omega} \rightarrow \Omega \quad (1)$$
where \( \hat{\Omega} \) is the parametric domain, and the parameterization can be computed with the information given by CAD (we will always assume that \( F \) is smooth and with piecewise smooth inverse).

Let now \( V \) be a Hilbert space of functions defined in \( \Omega \) and let \( V' \) be its dual. Given a bilinear form \( a : V \times V \to \mathbb{R} \) and the linear functional \( l \in V' \), the variational formulation of a source problem is:

\[
\text{Find } u \in V \text{ such that } a(u, v) = \langle l, v \rangle \quad \forall v \in V
\]

(2)

where we denote by \( a(\cdot, \cdot) \) a continuous and coercive bilinear form and by \( \langle \cdot, \cdot \rangle \) the duality pairing between \( V \) and \( V' \).

Following the Galerkin approach, we now want to approximate the infinite dimensional space \( V \) with a finite dimensional space \( V_h \), and then solve the corresponding discrete source problem:

\[
\text{Find } u_h \in V_h \text{ such that } a(u_h, v_h) = \langle l, v_h \rangle \quad \forall v_h \in V_h
\]

(3)

In the standard FEA method, the space \( V_h \) is a space of piecewise polynomials, while in an IGA procedure this space is formed by, for example, B-spline functions. As done in [1] we can define the space in a general way:

\[
V_h := \{ v_h \in V : \hat{v}_h = \iota(v_h) \in \hat{V}_h \} \equiv \{ v_h \in V : v_h = \iota^{-1}(\hat{v}_h), \hat{v}_h \in \hat{V}_h \}
\]

where \( \hat{V}_h \) is a discrete space defined in the parametric domain \( \hat{\Omega} \), and \( \iota \) is a proper pull-back function defined from the parametrization \( F \).

The problem is then solved in the usual way: let \( \{ \hat{v}_j \}_{j \in J} \) be a basis for \( \hat{V}_h \), with \( J \) a proper set of indices, then, with the hypothesis on \( F \), the set \( \{ \iota^{-1}(\hat{v}_j) \}_{j \in J} \equiv \{ v_j \}_{j \in J} \) is a basis for \( V_h \). Hence the discrete solution of the problem (3) can be written as

\[
u_h = \sum_{j \in J} \alpha_j v_j = \sum_{j \in J} \alpha_j \iota^{-1}(\hat{v}_j)\]

By substituting this expression in the discrete problem (3) we obtain a linear system of equations with \( \alpha_j \) as unknowns and the entries of the matrix and the right-hand side \( a(v_i, v_j) \) and \( \langle l, v_i \rangle \) respectively.

1.3 B-Spline functions

Curves consisting of just one polynomial or rational segment may often be inadequate (for example they require a high degree to satisfy a large number of constraints or to fit complex shapes). Some of the major strengths of B-spline and NURBS functions are:

- they are used by most of the CAD tools.
• they are highly convenient for surface modeling.
• they can exactly represent all conic sections, i.e. circles, cylinders, spheres, etc.
• they can be generated by many efficient and numerically stable algorithms.
• they can easily handle arbitrary continuity degree in single knots.

When using a FEA approach we usually deal with a mesh composed of single elements with two different representations: one in the physical space and one in the reference domain. Classical FEA basis functions are interpolatory, i.e. the degrees of freedom of the basis functions correspond to the nodal values.

B-spline basis functions, on the contrary, are not necessarily interpolatory. Moreover, we have two kinds of meshes: the control mesh and the physical one. The control points define the control mesh and the control mesh interpolates the control points. While the control mesh does not conform to the actual geometry, the physical mesh is a decomposition of it. Two notions of elements belong to this mesh: the patch and the knot span. The patch can be considered as a macro-element, or a sub-domain. Each patch has two representations, one in the physical space and one in the parent domain. In the easier cases just a single patch is enough to define even quite complicated geometries. Multi-patches can describe the most complex geometries and they can play the role of sub-domains within which element types and material models are assumed to be uniform. Each patch is then decomposed into knots spans.

A knot vector in one dimension is a non-decreasing set of coordinates in the parameter space:

\[ \Xi = [\xi_0, \xi_1, \ldots, \xi_{n+p+1}] \]

where \( \xi_i \in \mathbb{R} \) is the \( i \)-th knot, \( p \) is the polynomial degree (\( p + 1 \) is the order) and \( n \) is the number of basis functions used to build the B-spline curve. The knots divide the parameter space into elements; element boundaries in the physical space are simply the images of the knots under the B-spline mapping.

Knot vectors can be uniform if the knots are equally spaced, non-uniform otherwise. But the main advantage of the knots is that they can be repeated, and this allows us to describe complex domains and, most important, to generate important modifications to the property of the basis. In particular by changing the multiplicity of a knot we can change the level of continuity of the curve.\(^1\) A knot vector is said to be open if its first and last knots are repeated \( p + 1 \) times (i.e. the curve is interpolatory at its boundaries). In future we will always assume to be dealing with open knot vectors.

---

\(^1\)Basis functions of order \( p \) have \( p - r_i \) continuous derivatives across each knot \( \xi_i \), where \( r_i \) is the multiplicity of the \( i \)-th knot. Particular cases are obtained when:

• \( r_i = p + 1 \): the basis is interpolatory at the knot \( \xi_i \).
• \( r_i = p + 2 \): the basis becomes discontinuous at that knot; two different patches can be attached at that point.
We are now ready to define B-spline basis functions. Given the knot vector, the \( i \)-th B-spline basis function of \( p \) degree, denoted by \( N_{i,p}(u) \), is defined by the recurrence formula (Cox-de Boor):

\[
N_{i,0}(\xi) = \begin{cases} 
1 & \text{if } \xi \leq \xi \leq \xi_{i+1} \\
0 & \text{otherwise}
\end{cases}
\]

\[
N_{i,p}(\xi) = \frac{\xi - \xi_{i}}{\xi_{i+p} - \xi_{i}} N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi)
\]  

(4)

The main properties of these functions are:

- the basis constitutes a partition of unity.
- each basis function is point-wise non negative over the entire domain.
- \( N_{i,0}(u) \) is a step function, equal to zero everywhere except on the half-open interval \( u \in [u_i, u_{i+1}) \).
- for \( p > 0 \), \( N_{i,p}(u) \) is a linear combination of two \((p-1)\)-degree basis functions.
- each \( p \)-th order function has \( p-1 \) continuous derivatives across the element boundaries (i.e. across the knots if they are repeated).
- the support of the B-spline of order \( p \) is always \( p+1 \) knot spans.
- equation (4) can yield the quotient \( 0/0 \); we define that quotient to be zero.

Given B-spline basis functions, B-spline curves are built taking a linear combination of B-spline basis functions and defining a set of control points. In particular, given \( n+1 \) basis functions \( N_{i,p} \) and \( n+1 \) control points \( P_i \in \mathbb{R}^d \), \( i = 0, 1, \ldots, n \), a piecewise polynomial B-spline curve is defined by the following:

\[
C(\xi) = \sum_{i=0}^{n} N_{i,p}(\xi)P_i
\]  

(5)

The properties of these curves follow directly from the properties of the basis functions. For instance, if dealing with an open knot vector, they are interpolatory at the first and the last control point (and clearly they are interpolatory in every knot repeated \( p+1 \) times). Continuity is then controlled by the basis functions, while the control points act as weights, hence the control points can be modified without altering the curve continuity. Moreover, due to the compact support of the basis functions, moving a single control point can affect the geometry of no more than \( p+1 \) elements of the curve\(^3\).

\(^2\)The half open interval \( [u_i, u_{i+1}) \) is called the \( i \)–th knot span.

\(^3\)This particular property is obviously one of the main reason for the great usage of B-spline and NURBS in Computer Aided Design.
Another interesting property is the convex hull property: the non-negativity and partition of unity properties of the basis, combined with the compact support of the functions, leads to the fact that a B-spline curve is completely contained within the convex hull defined by its control points.

Finally, B-spline curves also possess a variation diminishing property, i.e. no plane has more intersections with the curve than it has with the control polygon. This is particularly useful when interpolating discontinuous data as it eliminates the risk of undesired oscillations.

We can now extend the concepts presented until now to B-spline surfaces. Let \( \{ P_{i,j} \}_{i=0,1,...,n; j=0,1,...,m} \) be a control points net, \( p \) and \( q \) two polynomial degrees, \( \Xi = [\xi_0, \xi_1, ..., \xi_{n+p+1}] \) and \( \Theta = [\theta_0, \theta_1, ..., \theta_{n+q+1}] \) two knot vectors, then the tensor product B-spline surface is:

\[
S(\xi, \theta) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_{i,p}(\xi)M_{j,q}(\theta)P_{i,j}
\] (6)

where \( N_{i,p} \) and \( M_{j,q} \) are univariate B-spline basis functions corresponding to the knot vectors \( \Xi \) and \( \Theta \) respectively.

In the same way we can define the tensor product B-spline solid:

\[
V(\xi, \theta, \zeta) = \sum_{i=0}^{n} \sum_{j=0}^{m} \sum_{k=0}^{l} N_{i,p}(\xi)M_{j,q}(\theta)L_{k,r}(\zeta)P_{i,j,k}
\] (7)

### 1.4 The Poisson’s problem

We are now ready to see how the general framework presented can be applied to the particular case of the Poisson’s problem, defined on a B-spline geometry and discretized with a B-spline functional space. This will be the model problem we will use to check the correctness of our code (see section 2).

Let us assume that the computational domain is constructed as a single patch and that the parametrization \( F \) is as in (1). We consider a Poisson’s problem with homogeneous Dirichlet boundary conditions:

\[
\begin{aligned}
- \text{div} \left( k(x) \text{grad} u \right) &= f \quad \text{in } \Omega \\
  u &= 0 \quad \text{on } \partial \Omega 
\end{aligned}
\] (8)

where \( \partial \Omega \) is the boundary and \( \mathbf{n} \) is the unit normal vector exterior to \( \Omega \). For simplicity we assume the source term to be \( f \in L^2(\Omega) \).

The variational formulation of the problem is:

Find \( u \in H_0^1(\Omega) \) such that

\[
\int_{\Omega} k(x) \text{grad} u \cdot \text{grad} v \, dx = \int_{\Omega} f v \, dx \quad \forall v \in H_0^1(\Omega)
\] (9)

We then have the variational formulation of the discrete problem (see (3))
Find \( u_h \in V_h(\Omega) \) such that
\[ \int_{\Omega} k(x) \mathbf{grad} u_h \cdot \mathbf{grad} v_h \, dx = \int_{\Omega} f v_h \, dx \quad \forall v_h \in V_h(\Omega) \quad (10) \]

where the discrete space \( V_h \) is defined as

\[ V_h = \{ v_h \in H^1_{0,\Gamma_\partial}(\Omega) : v_h = \tilde{v}_h \circ F^{-1}, \tilde{v}_h \in \tilde{V}_h \} \]

and \( \tilde{V}_h \) is the discrete space in the parametric domain that has to be chosen.

Let then \( \{ \tilde{v}_i \}_{i=1}^{N_h} \) be a basis for \( \tilde{V}_h \) (\( N_h = \dim(\tilde{V}_h) = \dim(V_h) \)) then, due to the assumptions made on the parametrization \( F \) we can define a basis for \( V_h \) as

\[ \{ v_i = \tilde{v}_i \circ F^{-1} \}_{i=1}^{N_h} \quad (11) \]

With these basis we can write the trial functions in (10) as

\[ u_h = \sum_{j=1}^{N_h} \alpha_j v_j = \sum_{j=1}^{N_h} \alpha_j (\tilde{v}_j \circ F^{-1}) \quad (12) \]

and their gradients as

\[ \mathbf{grad} u_h = \sum_{j=1}^{N_h} \alpha_j \mathbf{grad} v_j = \sum_{j=1}^{N_h} \alpha_j (DF)^{-T}(\mathbf{grad} \tilde{v}_j \circ F^{-1}) \quad (13) \]

where \( DF \) is the Jacobian matrix of the parametrization. It is then sufficient that the equations are verified for any test functions of the basis (11):

\[ \sum_{j=1}^{N_h} A_{ij} \alpha_j = \int_{\Omega} k(x) \sum_{j=1}^{N_h} \alpha_j \mathbf{grad} v_j \cdot \mathbf{grad} v_i \, dx = \int_{\Omega} f v_i \, dx = f_i \quad \text{for} \ i = 1, \ldots, N_h \quad (14) \]

where \( A_{ij} \) are the coefficients of the stiffness matrix and \( f_i \) are the coefficients of the right-hand side contributions from the source term. All the integrals are numerically approximated by a suitable quadrature rule. In order to describe this rule, let us introduce \( \tilde{K}_h = \{ \tilde{K}_k \}_{k=1}^{N_e} \), that is a partition of the parametric domain \( \hat{\Omega} \) into \( N_e \) non-overlapping subregions, that henceforth we refer to as elements. The assumptions on the parameterization \( F \) ensure that the physical domain \( \Omega \) can be partitioned as

\[ \Omega = \bigcup_{k=1}^{N_e} F(\hat{K}_k) \]

and the corresponding elements \( K_k = F(\hat{K}_k) \) are non-overlapping too. We denote this partition by \( \mathcal{K}_h = \{ K_h \}_{k=1}^{N_e} \).
Let now assume that a quadrature rule is defined on every element $\hat{K}_k$. Each of these rules is determined by a set of $n_k$ nodes and their corresponding weights:

$$
\{\hat{x}_{l,k}\} \subset \hat{K}_k, \quad \{\hat{w}_{l,k}\} \subset \mathbb{R}, \quad l = 1, \ldots, n_k
$$

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

$$
A_{ij} \cong \sum_{k=1}^{N_e} \sum_{l=1}^{n_k} k(\hat{x}_{l,k}) w_{l,k} \text{grad} v_j(\hat{x}_{l,k}) \cdot \text{grad} v_i(\hat{x}_{l,k}) |\det(D\mathbf{F}(\hat{x}_{l,k}))| \quad (15)
$$

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

$$
f_i \cong \sum_{k=1}^{N_e} \sum_{l=1}^{n_k} f(\hat{x}_{l,k}) w_{l,k} v_i(\hat{x}_{l,k}) |\det(D\mathbf{F}(\hat{x}_{l,k}))| \quad (16)
$$
2 The library

The starting point for the coding of this library is, as mentioned above, the pre-existent Octave package GeoPDEs. The main purpose of the library is to create the functional space, not to solve PDE problems: the actual solver should link our code and use it to create the matrices. As a consequence of this, we decided to use the structure required in LifeV (as it was in the summer of 2012); in this way it is possible to use this library to generate the functional space and then do the assemblage of the matrices in LifeV. The simple interface, however, should guarantee the possibility to use this library with any PDE solver. For instance it should be possible to replicate the interface in GeoPDEs in our library and use it to address our classes; this way one could easily include our code in Octave itself.

![Figure 1: The Steps of test-first development (TFD).](image)

For the development of the code we have decided to follow a very useful approach called Test Driven Development. TDD is a programming technique and its primary goal is to help the developer write clean code that works correctly. The basic idea is to add a test, run the test suite and ensure whether the test does or does not fail. But instead of writing functional code first and then the testing code as an afterthought, the test code is written before the functional code. Furthermore, you do so in very small steps: one test and a small bit of corresponding functional code at a time. A programmer taking a TDD approach refuses to write a new function until there is first a test that fails because that function isn’t present. Once the test is in place they then do the work required to assure that the test suite now passes (the new code may
break several existing tests as well as the new one). The steps are illustrated in Figure 1.

TDD completely turns traditional development around. When a new feature is to be implemented, the first problem to deal with is whether the existing design is the best design possible that enables to implement that functionality. If so, one can proceed via a Test First Development approach. If not, one can refactor it locally to change the portion of the design affected by the new feature. As a result, the quality of the design will always be improved, making it easier to work with in the future.

With traditional testing a successful test finds one or more defects. It is the same with TDD; when a test fails you have made progress because you now know that you need to resolve the problem. More importantly, you have a clear measure of success when the test no longer fails. TDD increases your confidence that the system actually works and therefore you can proceed.

Another useful aftermath of this approach is that the test suite provide a thorough working specification of the functional code. When trying to understand a class or operation the sample code in the tests can be as useful as the written documentation (maybe even more useful for some programmers).

As a consequence of this we will first discuss testlaplacian.cpp which is the main file of the test suite that can be found in the test/ folder. Passing this test was the goal of this stage of development.

The objective of this test is to compute the stiffness matrix and the right-hand side term of the simple Poisson’s problem presented in Section 1.4.

```cpp
1 //load knots and degrees
2 std::vector<Real> knots(14);
7 std::vector<UInt> degree(2);
8 degree[0] = 2; degree[1] = 2;
9
10 //create quadrature rule
11 std::vector<Real> qn(3), qw(3);
12 webbur::legendre_compute(3, &(*qn.begin())), &(*qw.begin()));
13
14 //load geometry
15 iga::geometry_rep geo = new iga::geo_bspline_2d (DATADIR "ring_refined.txt");
16 iga::geometry geo_ring (geo);
17
18 //create 1d meshes
19 iga::mesh_rep mshldu = new iga::mesh_1d (knots, qn, qw);
20 iga::mesh_rep mshldv = new iga::mesh_1d (knots, qn, qw);
21 iga::mesh msh_u (mshldu), msh_v (mshldv);
22
23 //create 2d mesh
24 iga::mesh_rep msh2d = new iga::mesh_tensor_product_2d (geo_ring, msh_u, msh_v);
25 iga::mesh msh (msh2d);
26
27 //create 1d spaces
28 iga::sp_bspline_1d sp_u (msh_u, knots, degree[0]);
29 iga::sp_bspline_1d sp_v (msh_v, knots, degree[1]);
30
31 //create 2d space
32 iga::sp_bspline_2d sp (geo_ring, sp_u, sp_v);
```
First of all we load the knot vectors and we create the quadrature rule using \texttt{sandia\_rules} library\(^4\). Since the degree of the B-spline in each direction is two, we want the rule to have three (or more) quadrature points, this way the integrals will be computed exactly.

We then call the constructors of our geometry, mesh and space. The geometry is loaded starting from the file \texttt{ring\_refined.txt}. This file contains the information to build a B-spline geometry that represents a quarter of a ring in the first quadrant (see Figure 2). The flag \texttt{DATADIR} is set in the makefile and will automatically address the correct data folder.

![Figure 2: The B-spline geometry created from \texttt{ring\_refined.txt}](image)

After the creation of the geometry a series of two one-dimensional meshes and spaces has to be created and these can be used, exploiting the tensor product nature, to create the actual functional space. We have now created the space and we are ready to use its methods to compute the elements of the right hand side and of the stiffness matrix. We can do this by calling the respective functions \texttt{op\_f\_v} and \texttt{op\_gradu\_gradv} which can be found in the file \texttt{operators.h}. We can observe that these functions use Octave SparseMatrix class: in Section 3 we will explain how it is possible to include Octave code in a C++ applications and how to correctly compile them.

The results are then written in two output files and they are validated by comparison with the analogous files generated with GeoPDEs.

\(^4\)In the \texttt{sandia\_rules/} folder, \texttt{libsandia\_rules.a} can be found, a C library which generates a variety of quadrature rules of Gaussian type of various orders and types in a fast and efficient way. For more information about this library see [4]. In Section 3 we will show how the presence of this library is automatically checked when launching \texttt{./configure}. 

```cpp
//compute right hand side term
std::vector<Real> rhs;
iga::op_f_v (f, &sp, &msh, rhs);

//compute stiffness matrix
Real mu = 1;
SparseMatrix mat;
iga::op_gradu_gradv (&sp, &sp, &msh, mu, mat);
```
This example shows how simple it is to set up a PDE problem with this scheme.

<table>
<thead>
<tr>
<th>Test file</th>
<th>Functions/classes tested</th>
<th>Validation method</th>
</tr>
</thead>
<tbody>
<tr>
<td>testindexing.cpp</td>
<td>ind_to_sub, sub_to_ind</td>
<td>Tests if ind_to_sub is the inverse of ind_to_sub and if ordering is column-major</td>
</tr>
<tr>
<td>testfindspan.cpp</td>
<td>findspan</td>
<td>testfindspan.tst</td>
</tr>
<tr>
<td>testbasisfun.cpp</td>
<td>one_basis_fun, der_one_basis_fun</td>
<td>testbasisfun.tst</td>
</tr>
<tr>
<td>testgeoidentity.cpp</td>
<td>geo_identity</td>
<td>Tests if map and jacobian returns the identity map.</td>
</tr>
<tr>
<td>testgeobspine.cpp</td>
<td>geo_bspine_2d, geo_bspine_3d</td>
<td>testgeobspine2d.tst, testgeobspine3d.tst</td>
</tr>
<tr>
<td>testmesh.cpp</td>
<td>mesh_tensor_product_2d, mesh_tensor_product_3d</td>
<td>testmesh2d.tst, testmesh3d.tst</td>
</tr>
<tr>
<td>testsp1d.cpp</td>
<td>sp_bspine_1d</td>
<td>Tests if eval returns 1 and eval_der returns 0 in each quadrature node.</td>
</tr>
<tr>
<td>testsp2d.cpp</td>
<td>sp_bspine_2d</td>
<td>Tests if eval returns 1 and eval_der returns 0 in each quadrature node.</td>
</tr>
<tr>
<td>testsp3d.cpp</td>
<td>sp_bspine_3d</td>
<td>Tests if eval returns 1 and eval_der returns 0 in each quadrature node.</td>
</tr>
<tr>
<td>testexport2d.cpp</td>
<td>sp_bspine_2d:export_to_vtk</td>
<td>testexport2d.vtu</td>
</tr>
<tr>
<td>testexport3d.cpp</td>
<td>sp_bspine_3d:export_to_vtk</td>
<td>testexport3d.vtu</td>
</tr>
</tbody>
</table>

In addition to testlaplacian.cpp a series of test can be found in the test/ folder to check the correctness of the code from the first and most basic functions and
classes to the more complex ones; all the methods of the classes are tested and
the results are compared to the correct ones by a series of assertions.

By launching *make check* all the tests are compiled and launched, and one
can easily see if everything goes well (i.e. every assertion is passed and all the
tests reach the end). This procedure allowed us to readily detect any errors and
to debug the code in an easier and faster way.

## 2.1 General structure

We can see the general structure of the code in the file *iga.h*. All the classes
and the functions of the library are included in the namespace *iga*. We have 5
abstract classes which inherit one from another:

- **space**: this class is responsible for the actual creation of the basis func-
tions.
- **mesh**: this class handles the mesh information such as the quadrature
nodes and the update of the current element.
- **geometry**: this class is responsible for all the geometrical information.
  Since the mesh and the space are built on a reference domain we will
  use the methods of this class to map the points on the physical domain,
  to compute the Jacobian matrix of the map and its determinant and to
  compute the inverse of the Jacobian map.

![Diagram of the general abstract classes]

Figure 4: Structure of the general abstract classes

To simplify the code we have introduced two classes: *geometry_rep* and
*mesh_rep*. These classes are responsible for the actual implementation, while
*geometry* and *mesh* acts as handler. *geometry*, which inherits from *geometry_rep*
(see Figure 4), contains a *geometry_rep* pointer; it then uses this pointer to over-
load the methods inherited from *geometry_rep* with its corresponding methods.
For example here is the implementation of *geometry::nbCoor()*:
virtual const UInt4
nbCoor () const { return geo -> nbCoor (); }

Where geo is the geometry_rep pointer. This way all the subsequent derivate
classes will not have to overload the geometry_rep methods, but they will over-
load them automatically. mesh and mesh_rep operates in an analogous way.

In the next few paragraphs we will briefly present the methods of these
classes and their concrete implementations. For a more detailed description of
all the methods and of the parameters they require as input see the Doxygen
documentation in the doc/ folder.

2.1.1 Geometry

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbCoor</td>
<td>Returns the number of coordinates.</td>
</tr>
<tr>
<td>map</td>
<td>Maps a point in the parametric space to the physical space.</td>
</tr>
<tr>
<td>jacobian</td>
<td>Computes the jacobian of the geometric map.</td>
</tr>
<tr>
<td>jac_det</td>
<td>Computes the determinant of the jacobian of the geometric map at the desired point.</td>
</tr>
<tr>
<td>jacobian_inv</td>
<td>Computes the inverse of jacobian of the geometric map.</td>
</tr>
</tbody>
</table>

All the methods of the class are listed in the table above. We will see some of
the details of the possible implementation of these methods in Section 2.2; here
we would like to bring your attention to the jac_det and jacobian_inv methods.
These two methods are general for every kind of geometry map: basically they
just have to compute the determinant of a matrix and its inverse. Then, since
it is general, the implementation can be found directly in the abstract class
geometry_rep.

In the file linear_algebra.h a series of inline functions are implemented to
help handle this kind of calculations (this results in a much cleaner coding of
the methods jac_det and jacobian_inv).

2.1.2 Mesh

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbQuad</td>
<td>Returns the number of quadrature nodes in the current element.</td>
</tr>
<tr>
<td>update</td>
<td>Updates the number of the current element.</td>
</tr>
<tr>
<td>quadPointCoor</td>
<td>Returns the coordinates of the required quadrature point .</td>
</tr>
<tr>
<td>weight</td>
<td>Returns the weight of the required quadrature point .</td>
</tr>
</tbody>
</table>

As can be seen in the table, the interface of this class is quite simple. As a
matter of fact we want the methods to be the most general since, although in
this paper we are always dealing with tensor product meshes, this class should
be able to handle more complex ones, with different kind of elements (clearly if

\footnote{In configure.ac the presence of Eigen is checked. In a future improvement of the library, if configure finds Eigen headers, it should be possible to use it to do this kind of calculations.}
one wants to maintain an IGA approach this could lead to the choice of more complex functional spaces).

2.1.3 Space

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbDof</td>
<td>Returns the number of degrees of freedom in the current element.</td>
</tr>
<tr>
<td>numTotalDof</td>
<td>Returns the total number of degrees of freedom in the space.</td>
</tr>
<tr>
<td>localToGlobalMap</td>
<td>Returns the global id of the desired degree of freedom.</td>
</tr>
<tr>
<td>phi</td>
<td>Returns the value of the desired basis function at the quadrature nodes.</td>
</tr>
<tr>
<td>dPhi</td>
<td>Returns the value of the derivative of the desired basis function at the quadrature nodes.</td>
</tr>
<tr>
<td>eval</td>
<td>Returns the value of the space at the desired point.</td>
</tr>
<tr>
<td>eval_der</td>
<td>Returns the value of the derivative of the space at the desired point.</td>
</tr>
</tbody>
</table>

In this class we have two methods, \( \phi \) and \( d\phi \), to handle the evaluation of the basis functions at the quadrature nodes, and two methods, \( \text{eval} \) and \( \text{eval}_\text{der} \), to evaluate the space itself at the desired points (not necessarily the quadrature nodes). Although, for now, this library just deals with scalar spaces, the structure of the methods in this class is general and they are already written to guarantee that they can handle vectorial ones.

2.2 Concrete class implementations

As shown in Figure 5 all the geometries implemented in the library derive from the abstract class \( \text{geometry}_	ext{rep} \). The classes we have created are:

- **geo_identity**: this is obviously the simplest geometry map. The constructor takes the number of dimensions desired as input.
- **geo_affine**: the constructor of this class needs the transformation matrix and the translation vector.
- **geo_spherical**: this class is built from three intervals: one for the radius \([r_{\text{min}}, r_{\text{max}}]\) and the other two for the angles \([\theta_{\text{min}}, \theta_{\text{max}}], [\varphi_{\text{min}}, \varphi_{\text{max}}]\).
- **geo_bspline_2d**
- **geo_bspline_3d**
The implementation of the methods of the first three classes is negligible.

We would like to focus our attention on the geo.bspline_2d class (the 3d one is analogous) since evaluation of B-spline curves and surfaces is a little more complex. The main constructor of the class takes a text file as input (examples of the structure of these files can be found in the test/data_files/ folder). The number of dimensions and the control points are stored and a B-spline space geo_sp is built starting from the knot vector and the degree required in each direction. It is important to note that geo_sp is just the space that contains all the basis functions of the geometry and these can be different to the ones we are going to create for our functional space. The geometry map and the space are two separate entities (we can in fact build a B-spline space on a non B-spline geometry), nevertheless, if we are dealing with a CAD generated geometry, the IGA approach suggests that we build a functional space with the same kind of basis functions and the same knot vectors and degrees (or higher).

It is also possible to create a B-spline geometry from a B-spline space: in this case the input space is used to directly build geo_sp and the user just has to define the set of control points with which the basis functions will be weighted.

The map method exploits the evaluation method of the B-spline space (see paragraph 2.1.3) transforming the point from the reference domain to the physical domain. The jacobian method operates in an analogous way calculating the derivative information.

The computation of determinants and inverse of the matrices is derived from the geometry_rep class.

We have created three classes to handle the mesh depending on whether it is one, two or three dimensional:

- mesh_1d
- mesh_tensor_product_2d
- mesh_tensor_product_3d

Each of these classes derives from mesh_rep (see Figure 5). All the meshes are created in the reference domain and the geometry map will then be used to transform the nodes and the weights on the physical domain.

1d meshes are created from a geometry, a breakpoints vector, the quadrature rule and the quadrature weights (which can be easily obtained with the sandia_rules library). In the constructor it is possible to generate the breakpoints directly from the knot vector that we will use for the space, since the repeated knots are eliminated. We then transform the quadrature rule and the quadrature weights on the elements and we create two vectors: one with all the nodes and one with all the weights.

The implementation of the methods of this class is quite simple.

2d meshes operate in a similar way. Since we are dealing with a tensor product approach the constructor receives two 1d meshes, one for each direction. As a consequence of this in the update method not only the current element index
has to be updated, but also the current element index of each one-dimensional mesh. When dealing with these kinds of multi-dimensional structures one of the possible difficulties was to correctly index them. In the file indexing.h two functions can be found which are very useful to address the elements stored:\footnote{Note that the indexing starts from zero and the priority of the subscripts follows the order with which they are passed to the function.}

- **sub_to_ind** takes as input two unsigned integer vectors, one with the subscripts and one with the total number of subscripts in each dimension. The return value is the index desired.

- **ind_to_sub** operates in the opposite way: from the index to the subscripts.

Using these two functions it is possible to handle any dimension since they exploit the built-in methods of the vector class to read or generate the subscripts. The 3d class is a direct consequence of the two-dimensional one.

Finally, these are the classes that create the B-spline functional spaces:

- **sp_bspline_1d**
- **sp_bspline_2d**
- **sp_bspline_3d**

Let’s start by analyzing the one-dimensional class **sp_bspline_1d**. The main constructor takes a *mesh*, the knot vector and the degree desired as input. We create two vectors, *fun* and *fun_der*, in which we pre-calculate the values of all the non-zero basis function on each node. All the internal functions used to evaluate B-spline basis functions are listed in the *basis bspline.h* file. The algorithms used can be found in [2].

To make it easier to access the values stored in *fun* and *fun_der*, the *dimensions* vector is created: the first value is the actual number of non-zero basis function on each element, the second one is the number of quadrature points on each element and the last one is the total number of elements; this vector will be used every time we need to access a particular value using the *sub_to_ind* function.

We have seen that *mesh_tensor_product_2d* contained two one-dimensional meshes which, since we are dealing with a tensor product policy, handle the mesh information along each direction. In the same way when the sp_bspline_2d constructor is called, two one-dimensional spaces are built: sp_bspline_1d’s methods will then be used to implement the 2d ones. An example, *localToGlobalMap*, can be seen below:

```c++
1 static std::vector<Uint> el_subs (2), loc_subs (2), glob_subs (2);
2 3 //! Return the global id of the idof-th degree of freedom in the iel-th element.
4 const ID
5 sp_bspline_2d:localToGlobalMap (const Uint& iel, const Uint& idof) const
```
In addition to the methods mentioned above for the one-dimensional class, we have the `export_to_vtk` method. This takes a vector of weights and the desired number of evaluation points in each direction as input and it generates a `.vts` file with the evaluation of the space. The results can then be visualized in any VTK viewer software.

3d spaces implementation directly follows from the two-dimensional ones remembering that we are dealing with tensor product spaces.
3 Development and build system

3.1 Autotools

Autotools is a collection of three tools:

- Autoconf
- Automake
- Libtool

Libtool is used to create shared, platform-independent libraries. However, for the creation of our library Autoconf and Automake have been primarily used, so we will focus our attention on them.

3.1.1 Autoconf

Autoconf is a powerful tool that creates shell scripts to automatically configure software source code packages. These configuration scripts can adapt the packages to many kinds of UNIX-like systems and, since they are independent, their users do not need to have Autoconf.

The configuration scripts produced by Autoconf require no manual user intervention when launched; they do not normally even need an argument specifying the system type. Instead, they individually test for the presence of each feature that the relative software package might need.

For each software package, Autoconf creates a configuration script from a template file that lists the system features that the package needs or can use. Once the shell code to recognize and respond to a system feature has been written, Autoconf allows it to be shared by many software packages that can use (or need) that feature. If it later turns out that the shell code needs adjustment for some reason, it needs to be changed in only one place; all of the configuration scripts can be regenerated automatically to take advantage of the updated code.

The configuration scripts that Autoconf produces are by convention called configure. When launched, configure creates several files, replacing configuration parameters in them with appropriate values.

To create a configure script with Autoconf, one needs to write an Autoconf input file, configure.ac, that contains invocations of the Autoconf macros that test the system features the package needs or can use (one can also use Autoconf template macros to produce custom checks).

We can analyze the configure.ac file created to build this library:

```plaintext
# Process this file with autoconf to produce a configure script.

AC_INIT([libiga], [0.0.1], [])
AM_INIT_AUTOMAKE
AC_CONFIG_SRCDIR([src/geo_spherical.cpp])
AC_CONFIG_HEADERS([src/config.h])
```
The first lines initialize Autoconf and Automake: `AC_PREREQ` indicate the version of autoconf that has been used, `AC_INIT` takes in as parameters the name of the package, its version number, and a contact address for bug-reports about the package (this address is output at the end of `./configure --help`, for instance). `AM_INIT_AUTOMAKE` can receive a series of options that should be applied to every `makefile.am`.

`AC_CONFIG_SRCDIR` receives some unique file that is in the package’s source directory as input; `configure` will check for this file’s existence to make sure that the directory that it is told contains the source code in fact does. The `AC_CONFIG_HEADERS` invocation causes the configure script to create a `config.h` file gathering #defines defined by other macros in `configure.ac`. Finally `AC_PROG_CXX` and `AC_PROG_CC` determine the C++ and C compiler to use, while `AC_RANLIB` check the presence of `ranlib`.

`configure` will then have to find out if `sandia_rules` is correctly installed somewhere in the computer and if the header files of Eigen and LifeV are present too. The library needs `sandia_rules` to generate the quadrature rules, therefore, if it is not found, an error message is returned and the configuration is stopped; if Eigen or LifeV are missing, on the other hand, the library can still operate and `configure` will just print a warning to inform the user of the situation.
Finally we check the presence of `mkoctfile` (and indirectly the presence of Octave itself) and we save the values of the flags in two output variables, `OCTAVE_COMPILE_FLAGS` and `OCTAVE_LINK_FLAGS`, which we will use to compile the files that use Octave’s classes (see Section 3.2). If `mkoctfile` is not found `testlaplacian` will not be compiled.

```
AC_CONFIG_FILES([Makefile
data/Makefile
src/Makefile
test/Makefile])

AC_OUTPUT
```

In the last lines the `AC_CONFIG_FILES` macro declares the list of files that configure should create from their `.in` templates. Automake also scans this list to find the `Makefile.am` files it must process. The `AC_OUTPUT` line is a closing command that actually produces the part of the script in charge of creating the files registered with `AC_CONFIG_HEADERS` and `AC_CONFIG_FILES`.

### 3.1.2 Automake

Automake is a tool for automatically generating `Makefile.ins` from files called `Makefile.am`. A `Makefile.am` has the same syntax as an ordinary `Makefile`. When Automake processes a `Makefile.am` it copies the entire file into the output `Makefile.in` (that will be later turned into `Makefile` by `configure`) but will react to certain variable definitions by generating some build rules and other variables. Often `Makefile.ams` contain only a list of variable definitions, for example variables that end with _PROGRAMS are special variables that list programs that the resulting `Makefile` should build. This kind of suffix is called a primary. Automake recognizes other primaries such as _HEADERS, _SOURCES, _SCRIPTS, _DATA, etc. corresponding to different types of files.

We can understand the neatness of this approach by looking, for example, at the `Makefile.am` in the `src/` folder:

```
libiga_adir = $(libdir)
libiga_a_CPPFLAGS = -Wall -style=cast -Werror
libiga_a_LIBRARYS = libiga.a
```
3.2 Linking Octave

The math libraries that Octave itself uses, can be utilized in standalone applications. These applications then have access, for example, to the array and matrix classes as well as to all the Octave algorithms. Including Octave code in a C++ application is very simple since it is sufficient to add the line:

```
#include <octave/oct.h>
```

To correctly compile files including Octave libraries it is necessary to use a series of flags that specify which dynamic libraries are needed and where to find them in the computer. An easy way to do this is calling `mkoctfile`: this function automatically compiles source code written in C, C++, or Fortran using the right options for our environment and the compiled code can be called within Octave or can be used as a standalone executable.

In `Makefile.am` in the `test/` folder we can see how the variables defined in `configure.ac` are used: if `mkoctfile` is present in the environment then `testlaplacian` is added to the compilation along with the operators functions. The flags loaded from `mkoctfile` are included in the compilation options for these files using the primaries `_LDFLAGS`, `LDADD`, `CPPFLAGS`.

```
AM_CPPFLAGS = -I$(top_builddir)/src -I$(top_srcdir)/src -I$(top_srcdir)/test -DDATADIR="$(top_srcdir)/data/" AUTOMAKE_OPTIONS = nostdinc
LDADD = $(top_builddir)/src/libiga.a

check_PROGRAMS = testindexing testgeoidentity testgeospherical testfindspan testbasisfun
testmesh testsp1d testsp2d testsp3d testgeobspline testexport2d testexport3d
testlaplacian

testlaplacian_SOURCES = operators.h testlaplacian.cpp operators.cpp
testlaplacian_LDFLAGS = @OCTAVE_LINKFLAGS@
testlaplacian_LDADD = @OCTAVE_LINK_ADD@$LDADD)
testlaplacian_CPPFLAGS = @OCTAVE_COMPILE_FLAGS@$AM_CPPFLAGS
endif
check_PROGRAMS += testlaplacian
testlaplacian_SOURCES = operators.h testlaplacian.cpp operators.cpp
testlaplacian_LDFLAGS = @OCTAVE_LINKFLAGS@
testlaplacian_LDADD = @OCTAVE_LINK_ADD@$LDADD)
testlaplacian_CPPFLAGS = @OCTAVE_COMPILE_FLAGS@$AM_CPPFLAGS
endif
```

Then we just have the list of source files needed to compile the other tests and we can see, again, how Automake simplifies the writing of a makefile.
4 Conclusions

In this paper we have presented how this library was created and how it is possible to effectively use it to create a functional space within the general framework of Isogeometric Analysis. The tests allowed us to check the correct functioning of the library.

The code can be improved in different directions. First of all the functionalities of the B-spline space could be enhanced allowing the treatment of vectorial spaces. Secondly, although most of the geometries can be represented with a single patch, the possibility to deal with multi-patch geometries could be added. Finally the extension to other functional spaces should be considered: the implementation of NURBS spaces, given the B-spline ones, is straightforward, but other spaces could be considered, such as generalized B-splines or T-splines which are already used in CAD design.

In the near future this library will be applied, using LifeV as solver, to an electromagnetic problem, in collaboration with Technische Universität (TU) Darmstadt. In particular, we are interested in solving an eigenmode problem in a cavity and then study the sensitivity of the cavity eigenmode with respect to the geometric parameters which is of great importance when studying microphoning and Lorentz detuning of accelerators’ cavities.
References


