SCUOLA INTERNAZIONALE SUPERIORE
DI STUDI AVANZATI

Mathematics Area

Master in High Performance Computing

Isogeometric Analysis in HPC: Object Oriented Design and Open Source Massively Parallel Implementation

Advisor:

Dott. Luca HELTAI

Co-Advisor:

Dott. Nicola CAVALLINI

Author:

Marco TEZZELE

2014-2015

Contents

Acknowledgements

This work was partially supported by the project OpenViewSHIP, “Sviluppo di un ecosistema computazionale per la progettazione idrodinamica del sistema elica-carena”, supported by Regione FVG - PAR FSC 2007-2013, Fondo per lo Sviluppo e la Coesione and by the project “TRIM Tecnologia e Ricerca Industriale per la Mobilit Marina”, CTN01-00176-163601, supported by MIUR, the italian Ministry of Instruction, University and Research. The work was also supported by NVIDIA Corporation.

Introduction

Isogeometric analysis, commonly addressed as IGA, is a technique that has become popular since [54]. In a broad sense it refers to computational mechanics applications that encapsulate computational geometry techniques or vice versa. The direct application of this concept is to use the shape functions describing the geometry as a discrete space for a Galerkin method. Numerous computational techniques benefit from this encapsulation. One of the major outcomes of IGA is a vigorous cooperation between computational mechanics and computational geometry communities. The result is a combination of techniques that was unknown to each community for decades. The number of successful applications is great and most modern computational mechanics aspects are covered. Worth mentioning are fluid dynamics applications in turbulent flows [69, 52], divergence conforming type of spaces [38] and fluid structure interaction applications [53, 19]. Structural mechanics IGA applications include shell theory [57, 15], vibration analysis [62], contact mechanics [31, 28, 32] and biomechanic applications [61]. Application of the isogeometric concept to potential flows and Stokes flows through boundary element techniques have also seen recent advances [50, 59].

Several different implementations of the IGA concept are available, both closed source (LSDYNA [47]) and opensource (GeoPDEs [30], `igatools` [21, 63], G+Smo [56] and PetIGA [23] to cite a few).

While closed source implementations usually offer a better user experience, they may be limited in development beyond built-in features, making their use not suitable for the implementation of new concepts, or experimental numerical features.

Opensource softwares, on the other hand, have the underlying goal of addressing a broader range of applications with less implementation restrictions, usually at the cost of not providing such a friendly environment, and having a somewhat steeper learning curve. A big advantage behind the use of opensource software comes from the possibility to count on a wide community of expert users and developers which makes it the ideal choice in academic communities (see, e.g., [5, 6]).

A very succesful model of opensource development is given by the `deal.II` library ([8, 9, 10, 7]) which shortens the separation between developers and users at a minimum. This is accomplished thanks to a very well documented library, an extensive test suite which ensures quality code, together with a centralised peer reviewed contribution system. We present an implementation of the IGA concept in the `deal.II` library, which exploits all those characteristics.

In Chapter 1, we review isogeometric type of shape functions, and introduce our notation. We also recall Bézier extraction technique, that is at the heart of our implementation. In Chapter 2 we explore the IGA specific data structures that are currently available in literature and we present, the way we encapsulated IGA concepts into `deal.II` data structures. We conclude the thesis with numerical applications, and software performances presented at Chapter 3.

Chapter 1

Isogeometric Analysis

Isogeometric analysis aims at incorporating the basis functions used to describe the geometry domain as basis functions for the mathematical modelling of mechanical phenomena. Since its introduction in [54], the scientific community has put a great effort in developing better and better shape functions that suite geometrical description and numerical modelling. Among others we mention the hierarchical approach [71, 58, 60], T-Splines basis functions [20, 12, 37, 68], and LR-Splines [33, 34].

In our implementation, we focus firstly on Bernstein polynomials and B-splines that, together with NURBS, are the most common choice in IGA analysis.

1.1 Bernstein polynomials and B-splines

The $n+1$ Bernstein basis polynomials of degree n are defined as

$$B_{i,n}(\xi) = \binom{n}{i} \xi^i (1-\xi)^{n-i}, \qquad i = 0, \dots, n \tag{1.1}$$

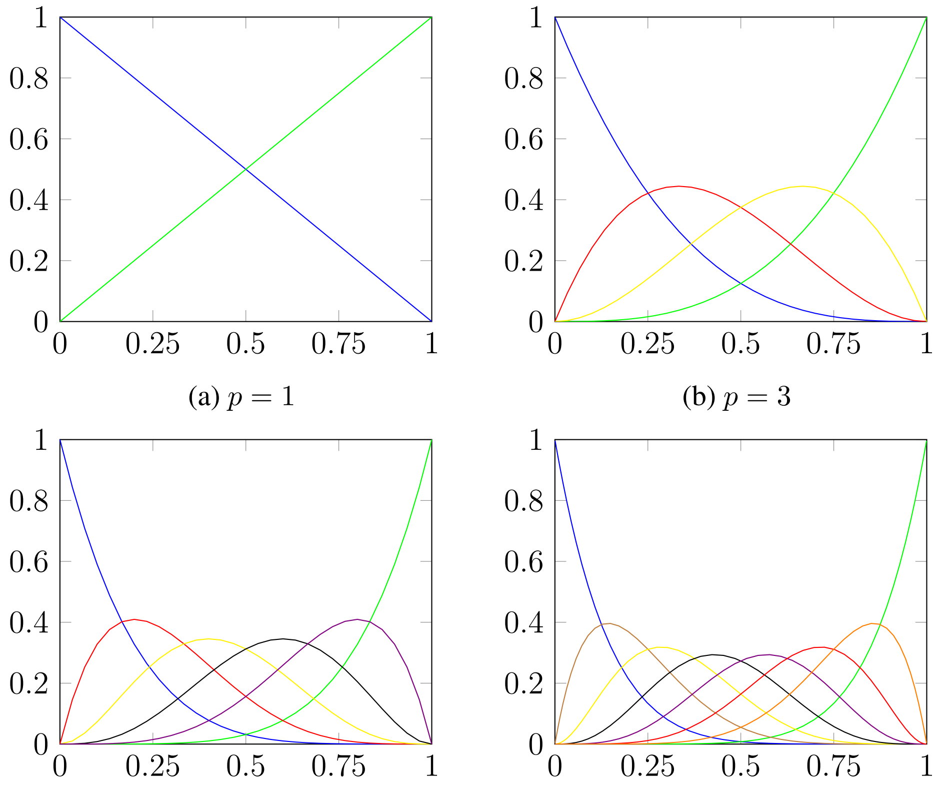
where $\xi \in [0,1]$. It is possible to write any Bernstein polynomial of degree n in terms of the power basis $\{1, \xi, \xi^2, \dots, \xi^n\}$ as follow (see for example [55])

$$B_{i,n}(\xi) = \binom{n}{i} x^i (1-\xi)^{n-i} = \sum_{j=i}^{n} (-1)^{j-i} \binom{n}{j} \binom{j}{i} \xi^j. \tag{1.2}$$

It is also possible to define the Bernstein polynomials recursively for $\xi \in [0,1]$ as

$$B_{i,n}(\xi) = (1-\xi) B_{i,n-1}(\xi) + \xi B_{i-1,n-1}(\xi). \tag{1.3}$$

In Figure 1.1 we can see them for four different degrees on the unit element of dimension one.



(a) $p = 1$

(b) $p = 3$

(c) $p = 5$

(d) $p = 7$

Figure 1.1: Bernstein basis polynomials of degree 1, 3, 5 and 7 on the unit element.

A Bézier curve is a linear combination of Bernstein polynomials and control points,

$$\mathbf{C}(\xi) = \sum_{i=1}^{n+1} \mathbf{P}_i B_{i,n}(\xi) = \mathbf{P}^T \mathbf{B}(\xi), \qquad \xi \in [0,1]. \tag{1.4}$$

Here $\mathbf{B}(\xi) = \{B_{i,n}(\xi)\}_{i=1}^{n+1}$ is the set of basis functions in ξ, and $\mathbf{P} = \{\mathbf{P}_i\}_{i=1}^{n+1}$ is the corresponding set of vector-valued control points, where $\mathbf{P}_i \in \mathbb{R}^d$, d the spatial dimension.

B-splines are defined through knot vectors: a finite nondecreasing sequence of coordinates in the parametric space, identified with a set $\Xi = \{\xi_1, \xi_2, \dots, \xi_{n+p+1}\}$. Here $\xi_i \in \mathbb{R}$, n is the number of basis functions and p is the degree of the piecewise polynomials that constitute the B-spline. It is usual to split Ξ into the vector $\{\zeta_1, \dots, \zeta_m\}$ of knots without repetitions, and the vector $\{r_1, \dots, r_m\}$ of their corresponding multiplicities, such that

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

with $\sum_{i=1}^m r_i = n + p + 1$. The maximum multiplicity allowed is $p+1$. Knot vectors without repetitions divide the parametric space into intervals called knot spans. A knot vector is said *open* if the first and the last knots are repeated $p+1$ times, i.e. $r_1 = r_m = p+1$.

We denote by $\mathbf{k} = \{k_1, \dots, k_m\}$, where $k_i := p - r_i + 1$, the vector that measures the smoothness of the spline functions at ζ_i. k_i represents the number of matching constraints associated to ζ_i. The maximum multiplicity allowed, $r_i = p+1$, gives no matching conditions ($k_i = 0$) which means a discontinuity at ζ_i. $\mathbf{k}$ collects the regularity of the basis functions at the knots.

B-spline basis functions are usually denoted as $N_{i,p}(\xi)$, where i corresponds to the i-th knot span and p corresponds with the order of the basis function. The definition of these functions is recursive in p. The degree-0 functions $N_{i,0}$ are piecewise constants which are one on the corresponding knot span and zero elsewhere. $N_{i,p}$ is a linear interpolation of $N_{i,p-1}$ and $N_{i+1,p-1}$. The latter two functions are non-zero for p knot spans, overlapping for $p-1$ knot spans. In detail, the so called Cox-de Boor recursion formula ([25, 29]) reads

$$N_{i,0}(\xi) = \begin{cases} 1 & \text{if } \xi_i \le \xi < \xi_{i+1}, \\ 0 & \text{otherwise.} \end{cases} \tag{1.5}$$

$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi), \tag{1.6}$$

where the coefficient is defined to be zero whenever the denominator is zero. In Figure 1.2 we can see an example of cubic B-spline basis functions with an open knot vector.

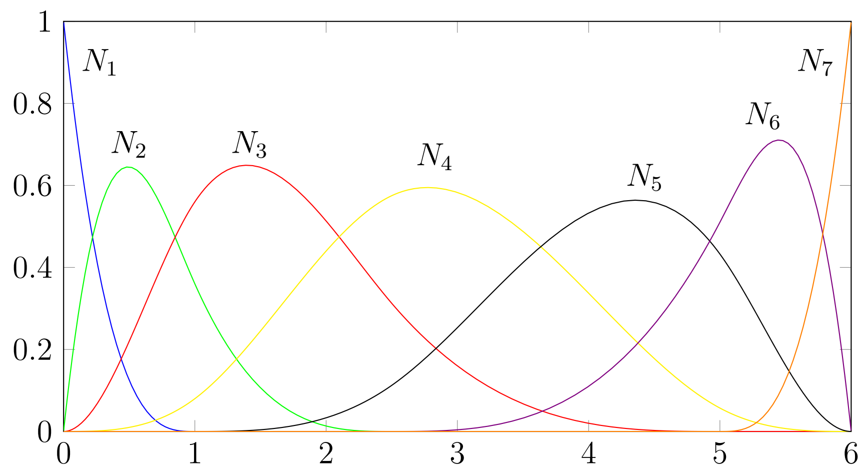


Figure 1.2: The seven cubic B-spline basis functions for the open knot vector $\Xi = \{0, 0, 0, 0, 1, 2.5, 5, 6, 6, 6, 6\}$.

A B-spline curve in $\mathbb{R}^d$ is defined as the linear combination of B-spline basis functions scaled with control points P_i. They can be loosely interpreted as the nodal coordinates in the frame of finite element analysis. We remark that control points are in general non interpolatory. Given n basis functions $N_{i,p}$ and n control points $P_i \in \mathbb{R}^d$, a piecewise-polynomial B-spline curve reads

$$C(\xi) = \sum_{i=1}^{n} N_{i,p}(\xi) P_i \,. \tag{1.7}$$

The control polygon is given by the piecewise linear interpolation of the control points. Given two knot vectors $\Xi = \{\xi_1, \xi_2, \ldots, \xi_{n+p+1}\}$ and $\mathcal{H} = \{\eta_1, \eta_2, \ldots, \eta_{m+q+1}\}$ and a control net $\{P_{i,j}\}$, with $i = 1, \ldots, n$ and $j = 1, \ldots, m$, it is possible to define a tensor product B-spline surface as follows:

$$\mathbf{F}(\xi, \eta) = \sum_{i=1}^{n} \sum_{j=1}^{m} N_{i,p}(\xi) N_{j,q}(\eta) P_{i,j}, \tag{1.8}$$

where $N_{i,p}$ and $N_{j,q}$ are B-spline basis functions of degree p and q respectively.

We define the tensor product B-spline basis functions as

$$T_{i,j} := N_{i,p} \otimes N_{j,q}, \qquad i = 1, \ldots, n, \quad j = 1, \ldots, m. \tag{1.9}$$

where, for notation convenience, we dropped the indices p and q from $T_{i,j}$. The corresponding tensor product B-spline space is defined as the space spanned by these basis functions, namely

$$\mathcal{S}^{p,q}_{\mathbf{k}_1,\mathbf{k}_2} \equiv \mathcal{S}^{p,q}_{\mathbf{k}_1,\mathbf{k}_2}(\mathcal{Q}_h) := \mathcal{S}^{p}_{\mathbf{k}_1} \otimes \mathcal{S}^{q}_{\mathbf{k}_2} = \text{span}\{T_{i,j}\}_{i=1,j=1}^{n,m},$$

where $\mathcal{Q}_h$ is the mesh composed by rectangles in the parametric space $\hat{\Omega}$, $\mathbf{k}_1$ and $\mathbf{k}_2$ the vectors associated to Ξ and $\mathcal{H}$ respectively. This can be generalized analogously in three dimensions as we will see in the following.

1.2 Bézier extraction operator

The isogeometric approach does not rely on the definition of a reference element, this is the major difficulty in adapting finite element oriented software to IGA applications. The Bézier extraction operation can be seen as a technique that restores the reference element concept typical of finite element softwares. Bézier extraction in fact maps B-splines and NURBS into Berstein polynomials on a reference element [24]. Here we briefly recall the technique and we will highlight its role in software development in the next section.

The principal idea behind the Bézier extraction operation is to exploit a property of B-spline curves that allows the insertion of new control points without changing the shape of the resulting curve, and to iterate such insertion by simply adding new knots at the same location of the old ones until each knot has been repeated p times.

Such operation results in a collection of basis functions which span a maximum of two knot spans, as in the finite element method. The process of knot insertion adds new degrees of freedom, and then it constraints them to be linear combination of the existing ones.

In particular, let $\Xi = \{\xi_1, \xi_2, \dots, \xi_{n+p+1}\}$ be a given knot vector and $\mathbf{P} = \{\mathbf{P}_i\}_{i=1}^{n}$ a set of control points, defining a B-spline curve. Let $\{\bar{\xi_1}, \bar{\xi_2}, \dots, \bar{\xi_j}, \dots, \bar{\xi_m}\}$ be the set of knots inserted to produce the Bézier decomposition of the B-spline. Define α_i^j, $i = 1, 2, \dots, n+j$, to be the i-th α to the j-th knot inserted. Then defining

$$\mathbf{C}^j = \begin{bmatrix} \alpha_1 & 1-\alpha_2 & 0 & \dots & & 0 \\ 0 & \alpha_2 & 1-\alpha_3 & 0 & \dots & 0 \\ \vdots & & & \ddots & & \\ 0 & \dots & & 0 & \alpha_{n+j-1} & 1-\alpha_{n+j} \end{bmatrix}$$

we can write in matrix form the sequence of control points formed from the knot insertion process (see [16]),

$$\overline{\mathbf{P}}^{j+1} = (\mathbf{C}^j)^T \overline{\mathbf{P}}^j \qquad \text{where} \qquad \overline{\mathbf{P}}^1 = \mathbf{P}.$$

Defining

$$\mathbf{C}^T = (\mathbf{C}^m)^T (\mathbf{C}^{m-1})^T \dots (\mathbf{C}^1)^T$$

the relation between the new Bézier control points and the original B-spline control points is

$$\mathbf{P}^b := \overline{\mathbf{P}}^{m+1} = \mathbf{C}^T \mathbf{P}. \tag{1.10}$$

Now we can write the B-spline curve representation $\mathbf{C}(\xi)$ in matrix form,

$$\mathbf{C}(\xi) = \sum_{i=1}^{n} N_{i,p}(\xi) \mathbf{P}_i = \mathbf{P}^T \mathbf{N}(\xi) \tag{1.11}$$

and given that the knot insertion causes no geometric nor parametric changes to the curve, we obtain

$$\mathbf{C}(\xi) = (\mathbf{P}^b)^T \mathbf{B}(\xi) = (\mathbf{C}^T \mathbf{P})^T \mathbf{B}(\xi) = \mathbf{P}^T \mathbf{C} \mathbf{B}(\xi) = \mathbf{P}^T \mathbf{N}(\xi).$$

This gives the relation between the B-spline basis functions and the Bernstein polynomials,

$$\mathbf{N}(\xi) = \mathbf{C}\mathbf{B}(\xi), \tag{1.12}$$

where **C** is the so-called Bézier extraction operator.

1.3 Localizing the extraction operator

Bézier decomposition creates a Bézier element over each knot interval. This is shown in Figure 1.3 for the knot span $[0, 1)$ of the previous example. It may be seen that the B-spline basis functions with support on a given knot span can be represented as linear combinations of the Bézier basis functions over the same interval. This way it is possible to obtain the localized element form of equation (1.12), i.e.

$$\mathbf{N}^e(\xi) = \mathbf{C}^e \mathbf{B}^e(\xi). \tag{1.13}$$

Note that $\mathbf{C}^e \in \mathbb{M}(p+1, \mathbb{R})$, this means that the global extraction operator **C** does not need to be established, only the localized extraction operators $\mathbf{C}^e$ for each element.

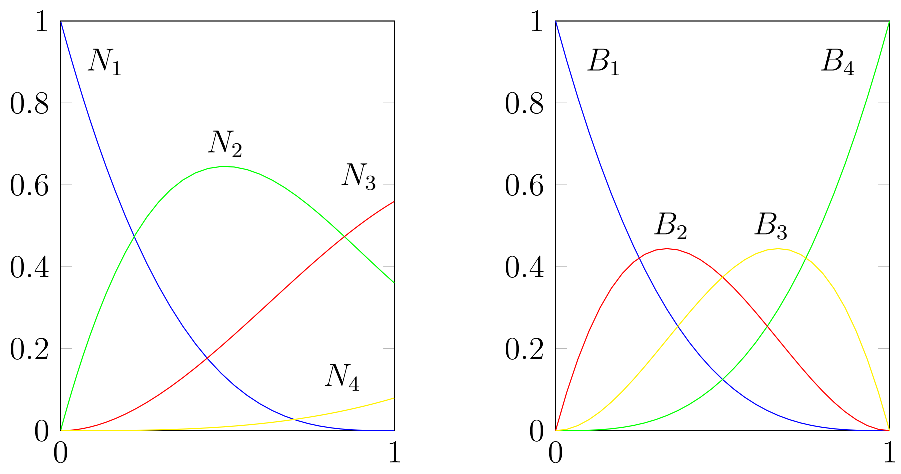


Figure 1.3: Bézier decomposition over the knot interval $[0, 1)$. N_i denotes the B-spline basis functions, while B_i indicates the corresponding Bernstein basis functions.

The localized bivariate extraction operator is defined as the tensor product of the univariate extraction operators. Let $\mathbf{C}^i_\xi, \mathbf{C}^j_\eta$ be the i-th and j-th univariate element extraction operators in the ξ, η direction. Then we have for a surface,

$$\mathbf{C}^e = \mathbf{C}^j_\eta \otimes \mathbf{C}^i_\xi = \begin{bmatrix} C^j_{\eta,11} \otimes \mathbf{C}^i_\xi & C^j_{\eta,12} \otimes \mathbf{C}^i_\xi & \dots \\ C^j_{\eta,21} \otimes \mathbf{C}^i_\xi & C^j_{\eta,22} \otimes \mathbf{C}^i_\xi & \\ \vdots & & \ddots \end{bmatrix} \tag{1.14}$$

where $\otimes$ is the Kronecker product, and e the index of the corresponding element associated to the tensor product between element i and j.

For the localized trivariate extraction operator we proceed similarly as above. Let $\mathbf{C}^i_\xi, \mathbf{C}^j_\eta, \mathbf{C}^k_\zeta$ be the i-th, j-th and k-th univariate element extraction operators in the ξ, η, ζ direction. Then we have for a solid,

$$\mathbf{C}^e = \mathbf{C}^i_\zeta \otimes \mathbf{C}^j_\eta \otimes \mathbf{C}^k_\xi. \tag{1.15}$$

Chapter 2

Data structures for IGA

Different approaches are possible in order to either integrate IGA structures in existing FEM codes or build them from scratch. We are going to illustrate some of those possibilities focusing on why we made our choice and how it is going to be effective.

2.1 Existing approaches

Existing implementations of IGA data structures are very diverse and depend heavily on the specific implementation framework. One of the earliest and most complete examples is given by GeoPDEs [30], which is a Matlab toolkit that implements basic tools to manipulate isogeometric domains and to solve partial differential equations. Its data structures are heavily dependent on the high level scripting language chosen by the authors, which is compatible with either Matlab [46] or Octave [48], and makes this tool ideal for educational and prototyping purposes but suffers sever limitations in terms of performances.

On the other side of the spectrum, we find the approach proposed in PetIGA [23], which is an extension of PETSc [4], adding NURBS discretization capabilities and the integration of forms into a framework which is specialized for linear algebra and high performance computing. The successful idea of PetIGA is to rely on PETSc for its internal data structures. This design choice inherits the pros and cons of PETSc data structures [4], and it represents a step towards high performance IGA computing.

Recent works on opensource C++ libraries also include G+SIMO [56], which is a tool that aims at bringing together mathematical tools for geometric design and numerical simulation. The goal of such a library is to serve a broad range of applications ranging from applied geometry, computational mathematics, and numerical simulation, although its development is still at an early stage, and its data structures are still under evolution.

`igatools` [63, 21] is one of the few softwares that specifically addresses IGA data structures, using modern software development techniques. `igatools` aims at building a modern IGA environment for scientific computing. All the developed data structures can be seen as grid like iterator that encapsulate specific mathematical concepts at the base of IGA, and can be summarised in the following table:

Reference Space In `igatools` the set of basis functions defined on the parametric domain is encapsulated in the object ReferenceSpace. Recalling the notation at the previous section, fixing ideas in two dimension for sake of clearness, the reference space can be addressed as $T_{i,j}$ defined at equation (1.9).

Mapping The mathematical mapping maps the reference space into the physical one. We introduced it as the deformation from $\hat{\Omega} := \Xi \otimes \mathcal{H}$ to $\Omega \subset \mathbb{R}^d$. In our notation it corresponds to the operator $\mathbf{F}$ defined in equation (1.8).

Push Forward This object combines the map and the transformation type. The transformation type is a concept borrowed from discrete differential geometry [3], defines the way functions are transformed from the reference to the physical domain. Different operators can be preserved throughout the transformation itself. Example transformations are $H(\mathbf{grad}, \Omega)$, $H(\text{div}, \Omega)$, $H(\mathbf{curl}, \Omega)$ and $L^2(\Omega)$.

Physical Space This class combines ReferenceSpace and PushForward. It contains all the information useful to recover point values of functions and derivatives on the physical domain.

Although there are notable differences between the IGA data structures presented above and those used in standard finite element libraries, we believe that it is possible to establish a one-to-one mapping between such data structures and those already available in the `deal.II` library. `deal.II` is the ideal candidate for an implementation of the IGA concept in a modern and well established scientific computing environment, which already benefits from high performance computing design and from two decades of best programming practices.

Implementing IGA oriented software from scratch has the advantage that data structures can be tailored in a very specific way to accomodate IGA requirements, as done, for example, in `igatools` [21, 63] and G+SIMO [56].

On the other hand, we were motivated in adapting the IGA requirements inside the `deal.II` library because we believe that the `deal.II` community and software are too much of a valuable development framework which comes with unquestionable benefits:

- a rigorous peer review process, ensuring high quality software;
- a systematic testing framework, running over 7.000 tests on more than 15 different system configurations after each commit to the main repository;
- a very large community user base, with more than 40 contributors worldwide, counting over 30.000 commits, at a rate of about 50 new commits per week;
- an extensive documentation, with more than 50 tutorial programs;
- direct interface with PETSc and Trilinos linear algebra, enabling massively parallel high performance computing.

2.2 deal.II data structures for IGA

`deal.II` is a library whose aim is to provide modules and basic building blocks useful to construct finite elements applications (see [10, 8] for some details). The rationale behind this choice is addressed as *General Purpose code*. In this framework, developers refrain from implementing specific applications, and concentrate their effort in identifying the major building blocks of a typical finite elements code. The major conceptual difficulty is to isolate abstract mathematical concepts and encapsulate them into corresponding data structures. The `deal.II` library benefits from a decade of development and has come to a point in which its software design is clear and very well tested. We present here a brief overview of the major classes which are ubiquitous in any finite element application:

Triangulation Triangulations are collections of their own elementary entities: **cells**, faces, vertices. Cells are defined as images of the reference hypercube $[0, 1]^{\text{dim}}$ under a suitable **mapping**. The triangulation class in `deal.II` stores the topological properties of a mesh together with some minimal information about the geometry, i.e., how the cells are connected and where their vertices are located.

Finite Element This class describes the reference finite element. It stores how many degrees of freedom are located at vertices, on lines, or in the interior of cells, and provides a way to evaluate values and derivatives of individual shape functions at *arbitrary points* on the unit cell.

DoFHandler A DoFHandler, or *Degrees of Freedom Handler* object, combines triangulations and finite elements by distributing and identifying the degrees of freedom associated with the global vector space generated by the finite element and by the triangulation. Topological information encapsulated into triangulation couples with the information on the reference element, and DoFHandler classes allocate this space so that each vertex, line, or cell of the triangulation has the correct number of degrees of freedom. It also manages the degrees of freedom global numbering. We remark the importance of *encapsulation* noticing that DoFHandler objects are ignorant of the shape functions that correspond to the degrees of freedom it manages.

Mapping These classes map the discrete space from the unit cell to the current geometrical configuration. Calculations are not directly performed at this level, but interfaces are provided to evaluate shape functions, derivatives, quadrature points etcetera at the actual position. Mapping classes describe how to map points from unit to real space and back, and provide gradients of this derivative and Jacobian determinants.

FEValues FEValues classes offer a point-wise view of the finite element function space, and couple Mapping, FiniteElements, Triangulations and DoFHandler to provide the user with access to shape functions, gradients and mapped points at quadrature points on the actual domain.

Given the original `deal.II` design, the three major design issues that need to be addressed are:

IGA radical isoparametric paradigm IGA can be seen as a radical interpretation of isoparametric concepts already known in finite elements techniques. In particular the main goal is to use geometric basis functions (B-splines or NURBS) as basis functions for the solution space. In this context, the geometric basis functions play the same role of the `deal.II` mapping classes, providing a way to map a reference element into its geometrical shape in real space.

Reference finite element: Berstein polynomials Reference finite elements fit already the original `deal.II` design, and only require the implementation of a FiniteElement class which adheres to the `deal.II` data structures and implement the evaluation of Bernstein polynomials.

Interelement continuity Bézier extraction complements Berstein reference element and allows our design to recover IGA interelement continuity.

The development of an enhanced Mapping class that would take into account the radical IGA isoparametric concept is certainly one of the key issues in implementing IGA techniques. `deal.II` already has some classes which allow iso-parametric implementations, but these were limited to tensor products of Lagrangian based polynomials (see the documentation for the MappingQ and MappingQEulerian classes). In the isogeometric paradigm we define a finite dimensional vector space based on geometric basis functions (like Bernstein polynomials), and the geometry is fully defined in terms of linear combinations of these basis functions (see Section 2.1). In this context, the Triangulation which is defined in `deal.II` takes the role of a *reference parametric* domain, on which both the geometry and the basis functions are defined.

Such a reference space is constructed in `deal.II` by combining a DoFHandler, a Triangulation and a FiniteElement. We created a new Mapping class which exploits the three combined objects to generate an isogeometric mapping. The resulting diagram is pictured in Figure 2.1. The only requirement of such combination is the capability to interface to the generic mapping structure that was already present in `deal.II`. We named this class **MappingFEField**, owing to its capability to interface to arbitrary finite element fields. In this context, isogeometric analysis is only a special case of the isoparametric concept. This class provides an interface to a geometric finite element class (based on Bernstein polynomials) that uses the triangulation as a source of topological information for the parametric space, and transform such information into a geometric *finite element field*. The information associated with the control points is then taken care of by a DoFHandler, as is usual for all other finite element field in `deal.II`.

MappingFEField effectively decouples the topology from the geometry, by relegating all geometrical information to some components of a FiniteElement vector field. Listing 2.1 shows how all of these ingredients combine together to build the MappingFEField.

Taking advantage of the structure of the library we implemented the class FE_Bernstein based on Bernstein polynomials in the most general way. In fact it is possible to perform hp adaptive refinement for C^0 continuity between elements in two and three dimensions. This would have been extremely difficult without relying on the existing infrastructure of `deal.II`.

Bézier extraction operator is the technique that restores interelement continuity. The Bézier extraction operator depends only on the knot vector and on the degree of the B-splines as we have seen in Section 1.2. The IgaHandler class is in charge of continuity recover. It is constructed passing three arguments: a vector containing the knots without repetition, a vector with the multiplicity of each knot and finally the degree. These three elements describe uniquely the set of B-

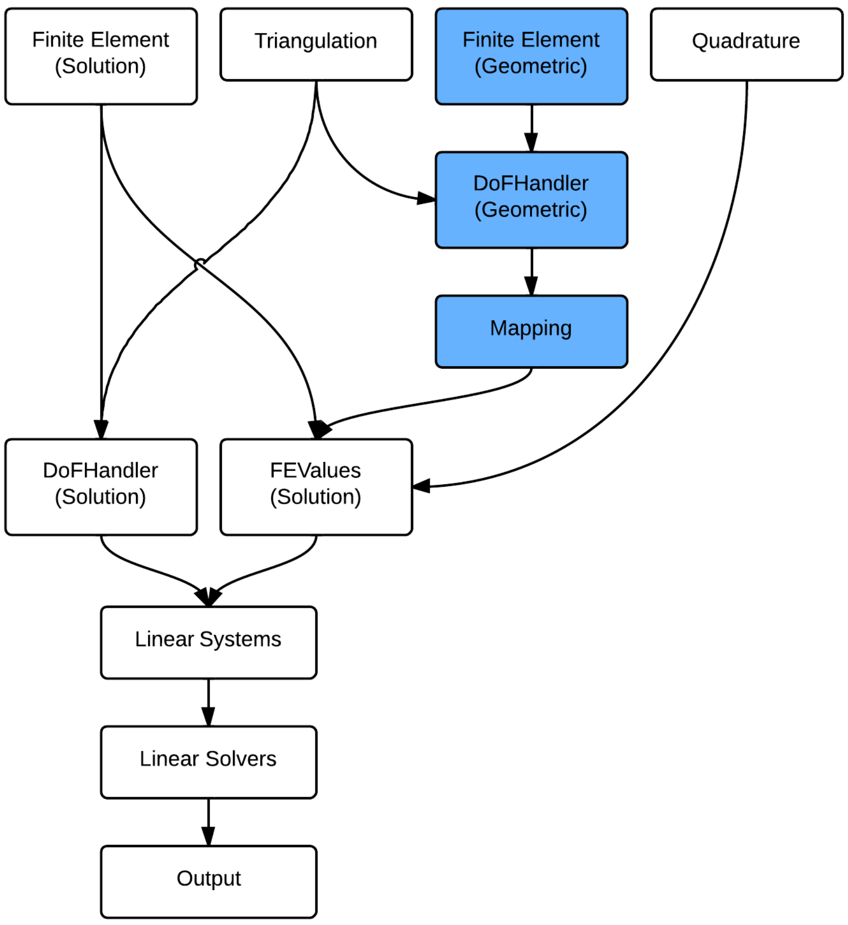


Figure 2.1: An outline of how the primary groups of classes in deal.II interact after new implementations.

```
// MappingFEField based on cubic bsplines
const FEBernstein<dim,spacedim> fe(3);
const FESystem<dim,spacedim> fesystem(fe, spacedim);
DoFHandler<dim,spacedim> dh(triangulation);
dh.distribute_dofs(fesystem);
Vector<double> control_points(dh.n_dofs());
// Fills the control points with information from
// the some iges file
read_control_points(dhq, control_points,
                    "geometry_file.iges");
MappingFEField<dim,spacedim> map(dhq, eulerq);
```

Code Listing 2.1: Usage of MappingFEField.

spline basis function that construct the operator. The class assembles the global extraction operator **C** of Equation (1.12) and all the local operators $\mathbf{C}^e$ of Equation (1.13). In 2 and 3 dimensions we perform the tensor product of the univariate extraction operators by a Kronecker product as in Equations (1.14) and (1.15). We can see an example of declaration and initialization of the IgaHandler class in Listing 2.2. Notice that all the topological information are carried by the IgaHandler class.

```
template <int dim>
class Problem
{
public:
  Problem(IgaHandler<dim,dim> &iga_handler);

private:
  IgaHandler<dim,dim>   &iga_handler;
  unsigned int          degree;
  Triangulation<dim>    &triangulation;
  FE_Bernstein<dim>     &fe;
  DoFHandler<dim>       &dof_handler;
  MappingFEField<dim>   *mappingfe;
};

template <int dim>
Problem<dim>::Problem (IgaHandler<dim,dim> &iga_handler)
    :
    iga_handler(iga_handler),
    degree(iga_handler.degree),
    triangulation(iga_handler.tria),
    fe (iga_handler.fe),
    dof_handler (iga_handler.dh),
    mappingfe(iga_handler.map_fe)
```

```
{}

int main (int argc, char *argv[])
{
  unsigned int degree = ...;
  std::vector<std::vector<double> > knots = ...;
  std::vector<std::vector<unsigned int> > mults = ...;
  IgaHandler<2,2> iga_hd2(knots, mults, degree);
  Problem<2> problem(iga_hd2);
}
```

Code Listing 2.2: Initialization of IgaHandler

The combination of these classes will be clear showing the resulting assemble code. Here we need to change the basis from Berstein polynomials to B-splines. This is carried out pre- and post-multiply the local stiffness matrix by the change of basis matrix:

$$\mathbf{B} = \mathbf{C}^{eT}\mathbf{A}\mathbf{C}^{e}, \tag{2.1}$$

A is the usual cell matrix and **B** as the new B-spline cell matrix. To transform a vector the operation is analogous: we have to simply multiply by the change of basis matrix. This is accomplished by the method distribute_local_to_global presented in Listing 2.3. We have to calculate in the usual way the cell matrix and rhs, and the class will assemble the final B-spline system matrix for us.

Applying boundary conditions is not a major difficulty, but deserves the reader's attention. Isogeometric shape functions are not interpolatory to degrees of freedom. Meaning we cannot assign Dirichlet boundary conditions evaluating the desired function at the degree of freedom location. Solution to this issue is an L^2 projection of the boundary data on the trace space of the original solution space. IgaHandler is still in charge of this operation. The implemented method is called project_boundary_values (see Listing 2.3). We would like to remark that these are the only lines that have to be changed with respect to existing user codes.

```
template <int dim>
void Problem<dim>::assemble_system ()
{
  ...
  iga_handler.distribute_local_to_global(cell_matrix,
            cell_rhs,
            cell,
            bspline_system_matrix,
            bspline_system_rhs,
            bspline_constraints);
  ...
  iga_handler.project_boundary_values(
      dirichlet_boundary_function,
        boundary_quadrature,
        bspline_constraints);
}
```

Code Listing 2.3: Usage of IgaHandler

Chapter 3

Application

The classical example we present is the solution of a Poissons problem with manufactured right hand side. We aim at validating the implemented software, and compare the performances of different approximations varying the continuity and the degree.

3.1 Test problem

Let us consider the Poisson's equation with homogenous Dirichlet boundary condition in two dimensions

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega, \end{cases} \tag{3.1}$$

where $\Omega \subset \mathbb{R}^2$ is a bounded domain with boundary $\partial\Omega$ Lipschitz continuous, $u \in C^2(\Omega) \cap C^0(\overline{\Omega})$ and $f \in C^0(\Omega)$. Under these hypotheses the problem admits existence and uniqueness of the solution (see for example [39]).

To formulate the variational equivalent of (3.1) we multiply Poisson's equation by a suitably smooth test function, integrate over Ω and then integrate by parts. Recall the definition of Sobolev spaces and their norms:

Definition 3.1.1 (Sobolev space)**.** *Given $s \in \mathbb{N}_0$ (smoothness) and $p \in [1, +\infty)$, the Sobolev spaces on an open set $\Omega \subseteq \mathbb{R}^n$ are defined as:*

$$W^{s,p}(\Omega) := \{v \in L^p(\Omega) : \forall\, |\alpha| \leq s \quad D^\alpha v \in L^p(\Omega)\}$$

with the associated norm and seminorm:

$$\|v\|_{W^{s,p}(\Omega)} = \|v\|_{s,p;\Omega} := \left(\sum_{|\alpha|\le s} \|D^\alpha v\|_{0,p;\Omega}^p \right)^{\frac{1}{p}}$$

$$|v|_{W^{s,p}(\Omega)} = |v|_{s,p;\Omega} := \left(\sum_{|\alpha|= s} \|D^\alpha v\|_{0,p;\Omega}^p \right)^{\frac{1}{p}}.$$

We will denote $H^s(\Omega) := W^{s,2}(\Omega)$ *and*

$$H_0^s(\Omega) := W_0^{s,2}(\Omega) = \{v \in W^{s,2}(\Omega) : D^\alpha v|_{\partial\Omega} = 0,\ \forall\, |\alpha| \le s-1\},$$

in the sense of trace operator.

Let us consider a test function $v \in H_0^1(\Omega)$, $f \in L^2(\Omega)$ and suppose $u \in H_0^1(\Omega)$. In addition, let ν denote the outward unit normal vector to $\partial\Omega$, which is by assumption in $L^\infty(\partial\Omega)$, and we set

$$\frac{\partial u}{\partial \nu} = \nu \cdot \nabla u.$$

Application of Green formula for the Laplacian results in

$$-\int_\Omega \Delta u\, v\, d\Omega = \int_\Omega \nabla u \cdot \nabla v\, d\Omega - \int_{\partial\Omega} \frac{\partial u}{\partial \nu} v\, d\gamma = \int_\Omega \nabla u \cdot \nabla v\, d\Omega,$$

where the last equivalence follows from $v \in H_0^1(\Omega)$, that is $v|_{\partial\Omega} = 0$ in the sense of trace operator. Taking this into account, we get the following weak formulation for problem (3.1):

find $u \in H_0^1(\Omega)$ such that

$$\int_\Omega \nabla u \cdot \nabla v\, d\Omega = \int_\Omega f v\, d\Omega \qquad \forall v \in H_0^1(\Omega). \tag{3.2}$$

Furthermore we have the following regularity result (see [18] and [65]):

Theorem 3.1.1. *Let Ω be a bounded domain, $k \in \mathbb{N}_0$ such that $\partial\Omega \in C^{k+2}$ and $f \in H^k(\Omega)$. Then the solution u of the problem* (3.1) *is in $H^{k+2}(\Omega)$.*

We want to choose the right hand side in such a way the problem admits the following exact solution

$$u(x, y) = \sin(a\pi x)\sin(b\pi y), \tag{3.3}$$

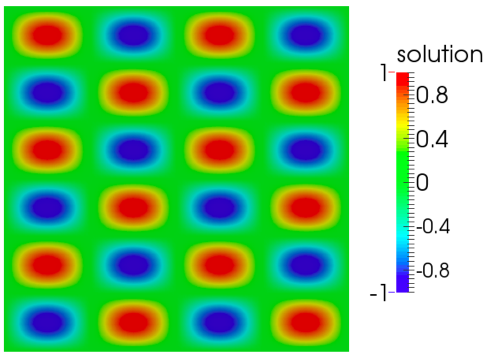


Figure 3.1: Plot of the approximated solution of the problem (3.1).

hence it reads

$$f(x,y) = (a^2 + b^2)\pi^2 \sin(a\pi x)\sin(b\pi y) = (a^2 + b^2)\pi^2 u(x,y), \tag{3.4}$$

where $a = 2$, $b = 3$ and $\Omega = [-1,1] \times [-1,1]$. This particular choice of a and b is due to avoid symmetry and possible superconvergence effects. The approximated solution for the problem (3.1) with these data is illustrated in Figure 3.1.

3.2 Serial version

Our application is composed of three main methods: setup of the problem, assembly of the system matrix and right hand side, and finally solution of the linear system.

In the `setup_system()` function we allocate the memory for the system matrix, the right and side, the constraints matrix for the boundary conditions and above all we compute the sparsity pattern of the system matrix. That is the set of all the entries of the matrix that are different from zero.

FEM applications typically result in the solution of linear systems characterised by sparse matrices. The easiest way to identify all the non-zero entries is to cycle over all the cells and extract the degrees of freedom associated to them. As we are going to explain there are other possibilities less intense from a computational point of view.

The next step is to compute the entries of the system matrix and right hand side that form the linear system from which we compute the solution. This is done in the `assemble_system()` method. This is the central function of each finite element program.

The general approach to assemble matrices and vectors is to loop over all cells, and on each cell to compute the contribution of that cell to the global matrix and right hand side by quadrature.

Finally we want to solve the the discretized equation by solving the linear system. This is done in the `solve()` method.

In Table 3.1 we profiled the serial version of code, to identify the most computationally intensive operations. In particular we observe that the assembly method is where we spend most of the time. This is a consequence of the fact that we are cycling over each cell of the triangulation. The good news is that this operation is embarrassing parallel because each process will take care of its own set of cells. The bad news is that the solve function is not embarrassingly parallel. As a solver we use the preconditioned conjugate gradient, with the hypre implementation of BoomerAMG provided by PETSc library as a preconditioner. It is an algebraic multigrid solver.

Table 3.1: Total wallclock time elapsed for the Laplace problem for degree 5, continuity 4, 409600 cells and 416025 dofs.

Section	**No. calls**	**Wall time**	**% of time**
Setup	1	75.36 s	5.28 %
Assembly	1	1899 s	91.10 %
Solve	1	110 s	3.62 %
Total		2084.36 s	100 %

It is interesting to extract from Table 3.1 the time spent to actually perform IGA related calculations. We want to underline the bottlenecks of the IgaHandler class and to show the differences from a classical FEM implementation.

Let us introduce the methods introduced with IGA implementation:

Compute local extractors The computation of the local Bézier extraction operators exploits the tensor product of the univariate extraction operators by a Kronecker product as in Equations (1.14). We only have to pay attention to the numbering of the dofs used internally by the library, but with a simple permutation everything goes smoothly.

Make sparsity pattern The global system matrix is a sparse matrix, therefore we need to create its sparsity pattern. That is the set of all the possible non-zero entries of the matrix. The method **make sparsity pattern** is the one that takes care of this action.

We assume that a certain finite element basis function is non-zero on a cell only if its degree of freedom is associated with the interior, a face, an edge or a vertex of this cell. As a result, the matrix entry between two basis functions can be non-zero only if they correspond to degrees of freedom of at least one common cell. Therefore, make sparsity pattern just loops over all cells and enters all couplings local to that cell. Listing 3.1 shows how the core of the function works for usual FEM codes.

```
for (typename DoFHandler<dim>::active_cell_iterator
       cell = dh.begin_active(); cell!=dh.end(); ++cell)
    for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
        for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
             sparsity_pattern.add(cell->dof_indices[i],
                                  cell->dof_indices[j]);
```

Code Listing 3.1: Core of the make sparsity pattern method.

We adapted this method in order to work with the IGA structures we created. To do so we access the **iga objects** vector in order to find the B-spline degrees of freedom associated the each cell. In Listing 3.2 we can see the serial version of the method.

```
for (typename DoFHandler<dim>::active_cell_iterator
       cell = dh.begin_active(); cell!=dh.end(); ++cell)
    for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
        for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
             sparsity_pattern.add(
                       iga_objects[cell].dof_indices[i],
                       iga_objects[cell].dof_indices[j]);
```

Code Listing 3.2: Core of the make sparsity pattern method in the IgaHandler class.

Transform cell matrix This method refers to the change of basis operation performed on the local stiffness matrix before adding it to the global system matrix. It consists in pre- and post-multiply the local stiffness matrix by the local Bézier extraction operator associated to that cell (see Equation (2.1)).

Transform cell rhs To transform a vector the operation is analogous to the one described above: we have to simply multiply the local rhs by the change of basis matrix.

We can see in Table 3.2 that the most expensive operation is the matrix matrix multiplications of the local cell matrices for the local Bézier extraction operator (see Equation (2.1)). In fact we have to pre- and post- multiply small but dense matrices $\in \mathbb{M}^{(p+1)\times(p+1)}$, where p is the degree of the B-splines employed.

Table 3.2: Total wallclock time elapsed for the IgaHandler methods for the Laplace problem above.

Section	**No. calls**	**Wall time**	**% of time**
Compute local extractors	2	0.0004771 s	0.00 %
Make sparsity pattern	1	73.78 s	3.54 %
Transform cell matrix	4.096e+05	904.6 s	43.40 %
Transform cell rhs	4.096e+05	0.4902 s	0.02 %
Total IgaHandler		978.87 s	46.96 %
Total Laplace		2084.36 s	

The second most expensive operation is the construction of the sparsity pattern of the global system matrix. This seems negligible with respect the change of basis operation but as we will see it affects the behaviour of the parallel solver implementation.

3.3 Parallel version

Having in mind the behaviour of the IGA data structures implemented we are going to show in detail the parallelization of each method. The main idea is to decompose the domain and exploit how the mesh and the local Bézier extraction operators are generated. This affect in particular the assemble part of the code since it operates on the single cell resulting in an embarrassing parallel execution.

We also notice that the construction of the sparsity pattern affects the solve part of the code, so we put much effort in optimizing that method. In computer science and mathematics, the term *domain decomposition* gets two different, in some cases complementary, meanings. In computer science it refers to the splitting of a computational domain and process each subdomain on a different processor. In mathematics it refers to a technique where the domain of a problem is divided in subdomains and appropriate interface conditions have to be set up. This second strategy is commonly used in complex multiphysyical problems. In the latter we are going to refer to the first one.

3.3.1 Domain decomposition

Since we want to exploit the tensor product performed by the library, it is natural to decompose the domain row-wise. This because we want the degrees of freedom to be contiguous in memory in order to use PETSc. In Figure 3.2 we show an example of such a decomposition in the case of 4 processes.

Each cell of a triangulation has associated with it a property called the **subdomain id**. While in principle this property can be used in any way application programs deem useful (it is simply an integer associated with each cell that can indicate whatever you want), at least for programs that run in parallel it usually denotes the processor a cell is associated with.

For programs that are parallelized based on MPI but where each processor stores the entire triangulation (as in this case), subdomain ids are assigned to cells by partitioning a mesh, and each MPI process then only works on those cells it "owns", i.e. that belong to a subdomain that the processor is associated with (traditionally, this is the case for the subdomain id whose numerical value coincides with the rank of the MPI process within the MPI communicator).

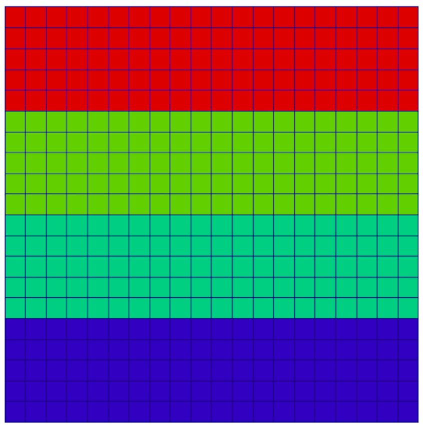


Figure 3.2: Example of domain decomposition for 4 processes.

So setting the subdomain id for all the cells means we are decomposing the domain. This operation is trivial if we pay attention to the reminder of the division of the rows of cells by the number of MPI processes. In Listing 3.3 we can see how we cycle over the cells and assign the subdomain id by computing the position of the cell with respect the patch.

This domain decomposition allows us to perform the parallelization for all the cycles over the cells of the mesh by checking whether the cell is "owned" or not (see Listing 3.4).

```
for (typename DoFHandler<dim,spacedim>::active_cell_iterator
      cell = dh.begin_active(); cell != dh.end(); ++cell, ++el)
   {
     cell_coordinates = compute_cell_coordinates(cell->index(),
                               subdivisions);
     cell->set_subdomain_id(
                   std::min(cell_coordinates[1]/rows_per_proc,
                   n_mpi_processes-1));
   }
```

Code Listing 3.3: Decomposing the domain by setting the subdomain id.

```
for (typename DoFHandler<dim>::active_cell_iterator
        cell = dh.begin_active(); cell!=dh.end(); ++cell)
    if (cell->subdomain_id() == this_mpi_process)
        ...
```

Code Listing 3.4: Parallelization of all the cycles over the cells.

3.3.2 IgaHandler

Sparsity pattern creation

To parallelize the **make sparsity pattern** method we tried different solutions. We are going to show them in the chronological order in which they were implemented, in order to clearly show the improvements. From now on we will refer to an *index set* as the abstract data type that can store indices (dofs) without any particular order, and no repeated values. Moreover with relevant dofs of a MPI process we intend all the dofs associated to the cell owned by the process plus the ones associated to the B-splines that have support on cells owned by two different processes. In other words we can say that relevant dofs are the dofs owned by this processor plus some "ghost" ones, which are important to this processor, but belong to another processor (and for which we exploit library utilities for communications).

Previously we assigned a subdomain id to each cell. Exploiting this, the sparsity pattern is built only on cells that have a subdomain id equal to the rank of the MPI process (see Listing 3.4). This is useful in parallel contexts where the matrix and sparsity pattern may be distributed and not every MPI process needs to build the entire sparsity pattern. In this case, it is sufficient that every process only builds that part of the sparsity pattern that corresponds to the subdomain id for which it is responsible. Notice that only for those cells the local Bézier extraction operators have been computed.

In this way we do not considering the ghost cells. We have to add the dofs associated to the B-splines that have support on cells owned by two different processes. The easiest way to do this is to identify the so called ghost dofs by subtracting the dofs owned by each process and the relevant ones. Then add for all the relevant dofs che corresponding entry in sparsity pattern corresponding to the ghost dofs (see Listing 3.5). We will refer to this implementation as the version 1.

```
IndexSet ghost_dofs = this_cpu_set_relevant_bspline;
ghost_dofs.subtract_set(this_cpu_set_owned_bspline);
for (auto i : this_cpu_set_relevant_bspline)
  for (auto j : ghost_dofs)
    dsp.add(i,j);
```

Code Listing 3.5: Adding the ghost dofs to the sparsity pattern - version 1.

This strategy is easy to implement, but inefficient, mainly because we are cycling over index sets. Therefore we are adding a lot of entries not needed. This affect the behavior of the solver. When we distort the structure of the system matrix the algebraic multigrid solver simply does not scale at all, resulting in a useless parallel implementation.

One solution is to avoid cycling over a set exploiting the fact that the dofs are contiguous in memory. We can calculate the first and the last dof of the two index sets identifying the two intervals of dofs we have to add (see Listing 3.6). We will refer to this implementation as the version 2. Since we do not access the sets anymore we have a significant gain in performances. The problem is that we did not reduce the number of entries, resulting is a still suboptimal implementation.

```
for (unsigned int i=lowest_relevant; i<=greatest_relevant; ++i)
{
  for (unsigned int j=lowest_relevant; j<lowest_owned; ++j)
    dsp.add(i,j);
  for (unsigned int j=greatest_owned+1; j<=greatest_relevant;
   ++j)
    dsp.add(i,j);
}
```

Code Listing 3.6: Adding the ghost dofs to the sparsity pattern - version 2.

On one hand, we want to be able to add only the entries needed in an optimal way and on the other hand to do so exploiting the contiguity in memory. By using this strategy, we exploit all different levels of cache memory, avoiding cache misses, which results in a great improvement of performances. We have to rethink the entire method, especially the core (see Listing 3.2). If we assume the worst-case scenario, that is, when the continuity is equal to the degree minus one, given a dof index we can couple it with the $p+1$ dof indices in its neighborhood, where p is the degree of the B-plines. When we use the maximum continuity, the basis functions have the largest support, in particular they spans $p+1$ intervals. Given this, we are able to construct the sparsity pattern cycling only over degrees of freedom and not over cells, which are complex data structures, stored in a non-contiguous array in memory, making their access rather expensive. In Listing 3.7 we can see version 3, the last one, of the method.

```
unsigned int dpe_bspline = std::sqrt(n_bspline);

for (unsigned int i=lowest_relevant; i<=greatest_relevant; ++i)
  for (unsigned int k=0; k<=degree+1; ++k)
    for (int j=-int(degree); j<=int(degree); ++j)
    {
      unsigned int col_id = std::min( std::max(int(i) + j,
    int(lowest_relevant)) + k * dpe_bspline,
    greatest_relevant);
      dsp.add(i, col_id);
      dsp.add(col_id, i);
    }
```

Code Listing 3.7: Adding the ghost dofs to the sparsity pattern - version 3.

The gain in performance is quite good. As we can see in Figure 3.3 we spend almost 99% of time less with version 3 with respect to version 1 of the method. Moreover for version 1 and 2 the solver does not scale at all, while as we will see with version 3 we have a scaling up to 10 with 32 processes. Having a sparsity pattern too dense make the parallel algebraic multigrid solver quite ineffective.

Local change of basis

The two methods referred to **transform cell matrix** and **transform cell rhs** are parallelized expoliting the domain decomposition. Each MPI process takes care of its own cells. While the matrix matrix multiplication and the matrix vector multiplication are performed by the optimized version of Intel® Math Kernel Library (Intel® MKL) used internally by the deal.II library. It is the fastest and most used math library for Intel and compatible processors according to Evans Data Software Developer surveys 2011-2013. We will see in Section 3.3.3 how this affects the scalability with respect to the degree employed.

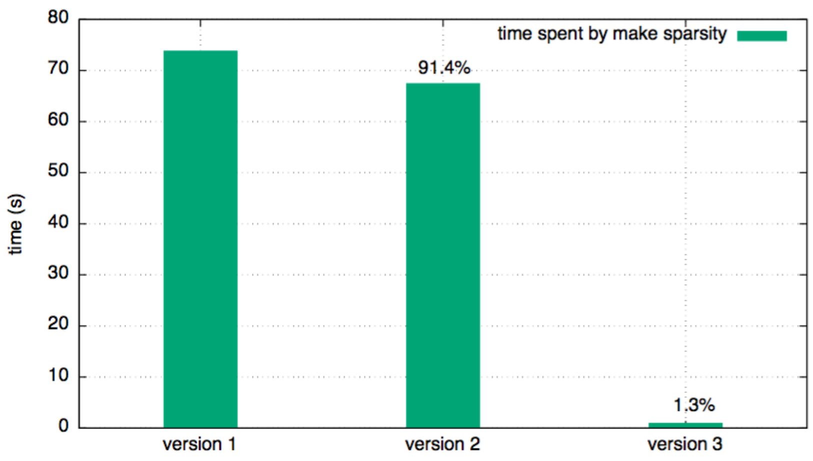


Figure 3.3: Comparison between different versions of make sparsity pattern. Degree equals to 5 and C^4 continuity between elements for a mesh composed by 640×640 cells.

Setup of the index sets

In the **setup index sets** method we simply construct all the index sets of the dofs and the ghost dofs associated to the cells owned by each MPI process. It is not an heavy operation from the computational side. It also saves the first and the last dof for each index set in order to avoid to cycle over sets. This permits to save lot of time in the other methods.

3.3.3 Strong scalability

We perform a strong scalability test for different degrees, and for each degree we test two different continuities. In particular we show results for degree p equals to 4 and 5. The continuity is set to 0 and $p - 1$. We use the new Ulysses cluster facility located at ICTP headquarter. In particular we use 2 nodes per time with 20 cpus each and 20 GB RAM each.

We observe that up to degree 4 the scaling is quite satisfactory (see Figures 3.4 and 3.5). For the Laplace problem methods we notice that the solver does not scale as we would. The reason is that the preconditioner we use with Boomer-AMG is initialised using the system matrix itself, and this has been shown to be suboptimal for high degrees of polynomials. We expect better result using a different preconditioner, for instance an AMG preconditioners based on a system matrix constructed with B-splines of order 1 over the control points. This will be the subject of a future investigation.

The change of basis operation (identified with *cell matrix* in Figures 3.5 and

3.7) does not scale well for 16 and 32 processes, this could be due to the fact that all the core assigned to the job are near each other and they cannot exploit the shared levels of cache. Also the slices of mesh and the matrices are quite small so the overhead due to the communication is significant.

We notice a not satisfactory behaviour of the setup method in the Laplace problem class for C^3 continuity between the elements (see Figure 3.6), this is also do to the way the core are assigned. In fact one process is clearly slower than the others and the plots are done with that specific time.

From degree $p = 5$ we start to superscale performing the matrix matrix multiplication for the change of basis. This is due to the fact that the deal.II library uses the optimized version of the Intel® MKL as we have seen.

The degree 5 is the first one that shows the superscaling (see Figures 3.8, 3.9, 3.10 and 3.11). We think it is because with lower degrees the local cell matrices are not big enough to exploit the optimization. Since the local matrices are computed in the assemble method of the Laplace problem, also that method superscales.

We underline that despite the assemble method superscales the time spent solving the system dominates by one or two order of magnitude the time spent assembling the system, depending on the degree and the continuity. So we assist to a shift in the percentage of time spent on all the methods.

3.3.4 Weak scalability

We perform weak scalability test for the same combinations of degrees and continuity as above. We double the number of dofs and accordingly we double the number of processes.

For degree equals to 4 (Figures 3.12, 3.13, 3.14 and 3.15) the results are quite unpredictable. We see that the Intel® MKL does not perform well when the matrices are small. Moreover we have the proximity of the cpus problem. If two cpus share the same L1 and L2 cache the performance are affected.

We experience fluctuating behaviour especially for degree equals to 5 (Figures 3.16, 3.17, 3.18 and 3.19). Despite that we have almost parallel lines so we can affirm the weak scalability is good.

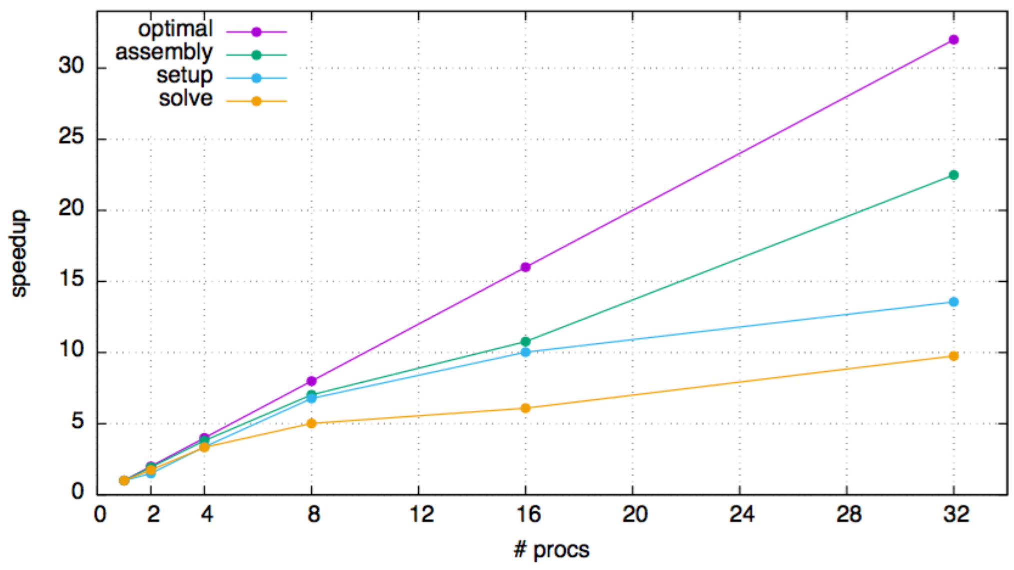


Figure 3.4: Strong scalability for Laplace problem with IgaHandler. Degree $p = 4$, C^0 continuity, 320×320 cells.

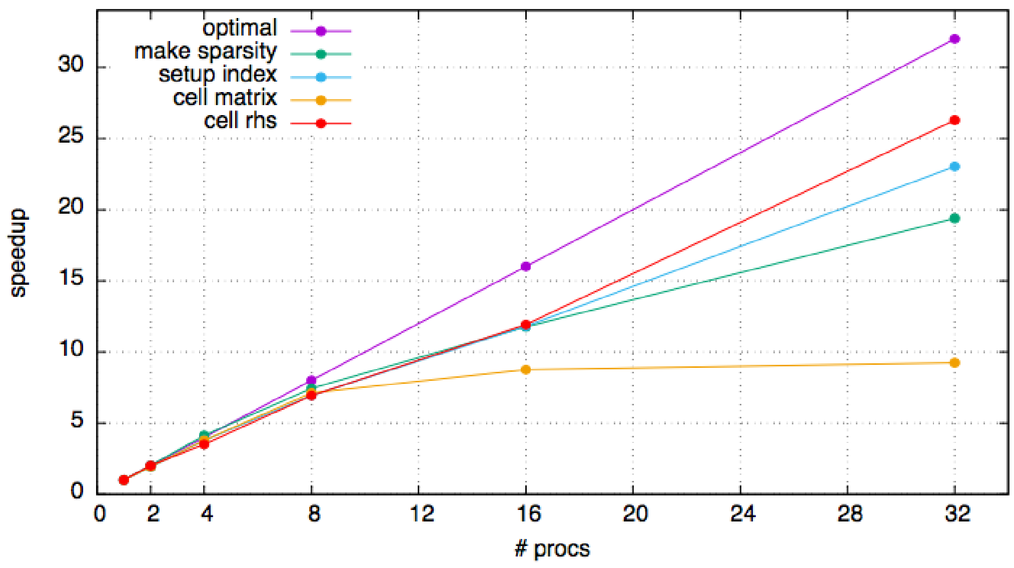


Figure 3.5: Strong scalability for IgaHandler class. Degree $p = 4$, C^0 continuity, 320×320 cells.

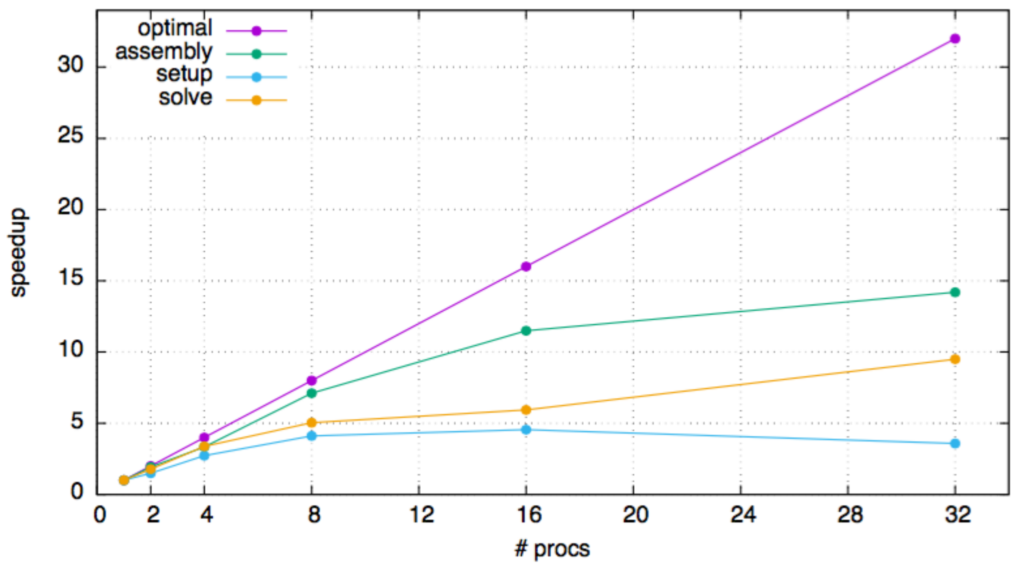


Figure 3.6: Strong scalability for Laplace problem with IgaHandler. Degree $p = 4$, C^3 continuity, 960×960 cells.

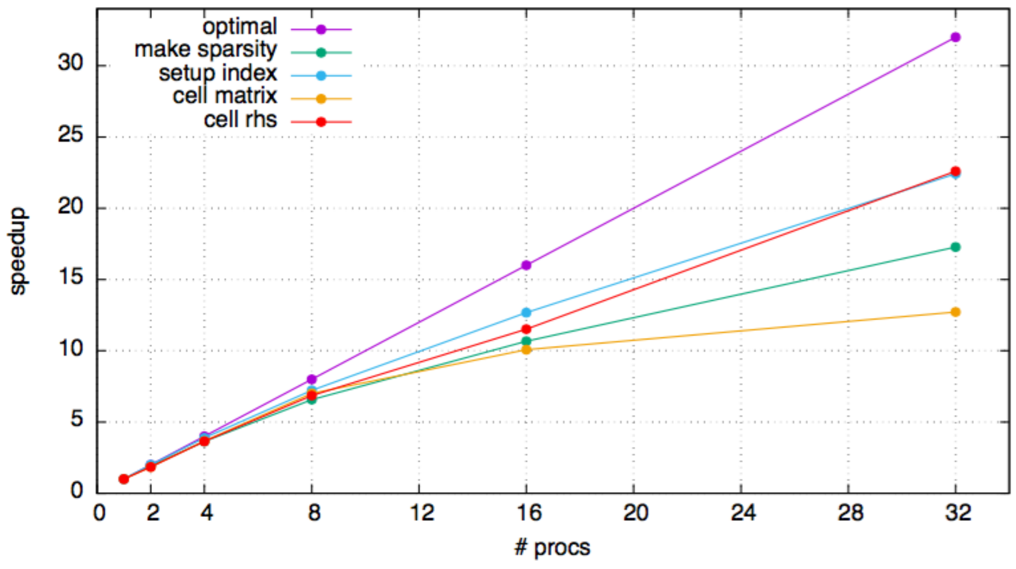


Figure 3.7: Strong scalability for IgaHandler class. Degree $p = 4$, C^3 continuity, 960×960 cells.

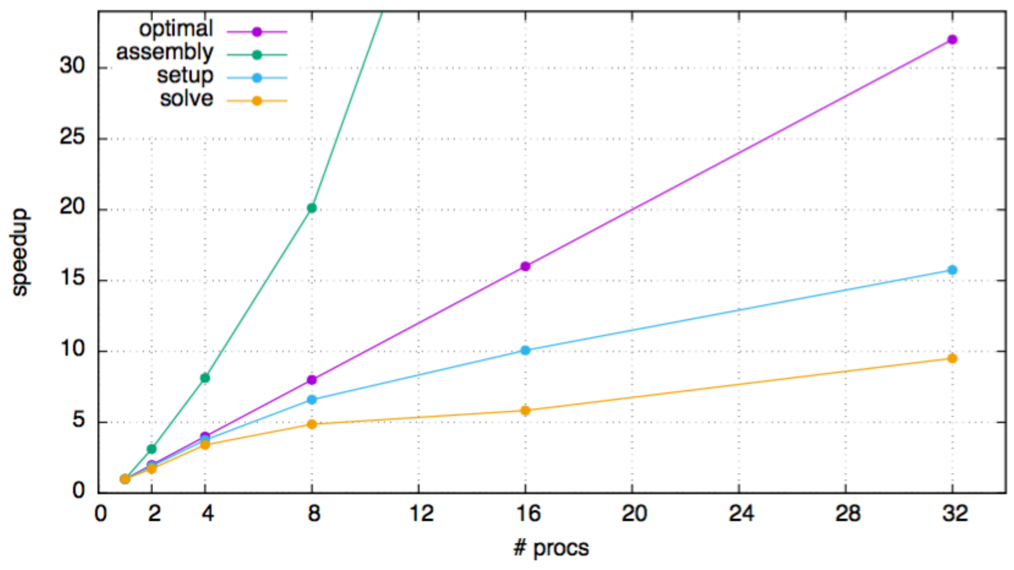


Figure 3.8: Strong scalability for Laplace problem with IgaHandler. Degree $p = 5$, C^0 continuity, 320×320 cells.

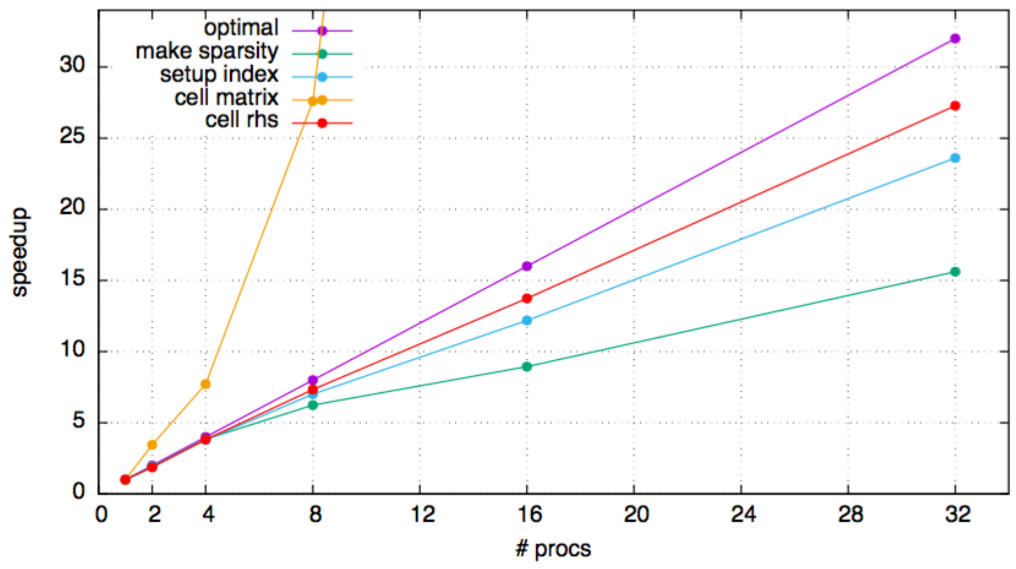


Figure 3.9: Strong scalability for IgaHandler class. Degree $p = 5$, C^0 continuity, 320×320 cells.

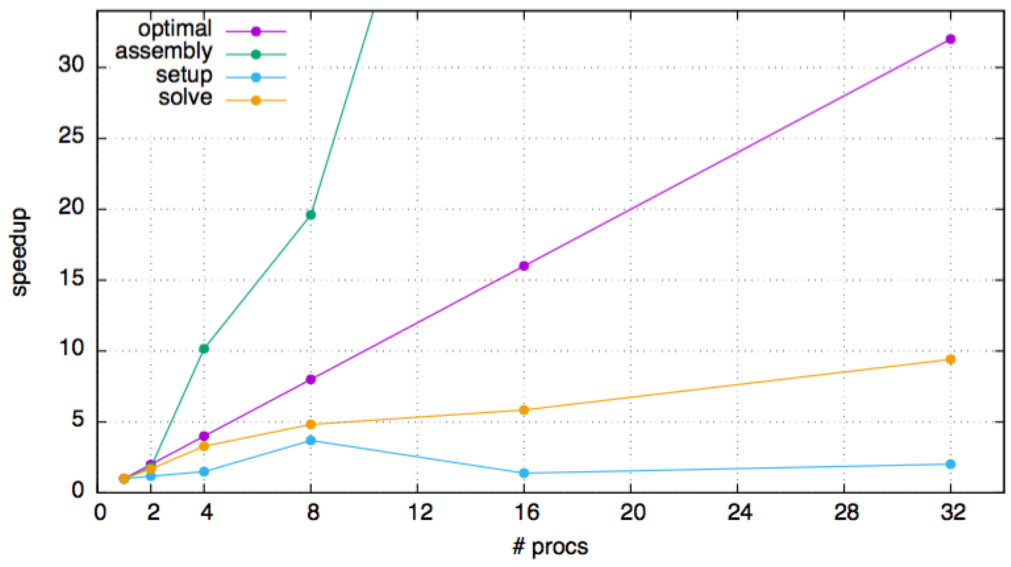


Figure 3.10: Strong scalability for Laplace problem with IgaHandler. Degree $p = 5$, C^4 continuity, 640×640 cells.

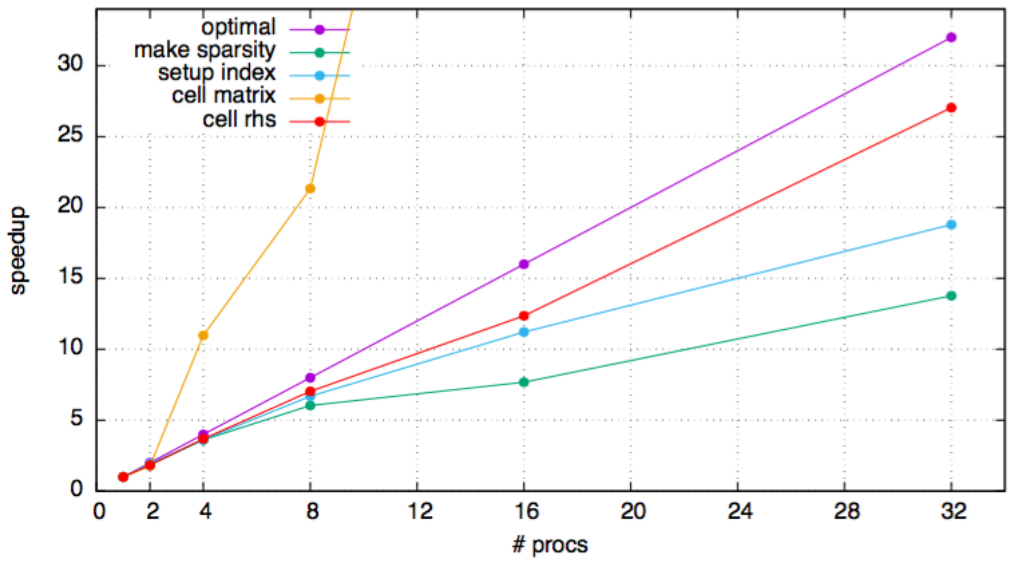


Figure 3.11: Strong scalability for IgaHandler class. Degree $p = 5$, C^4 continuity, 640×640 cells.

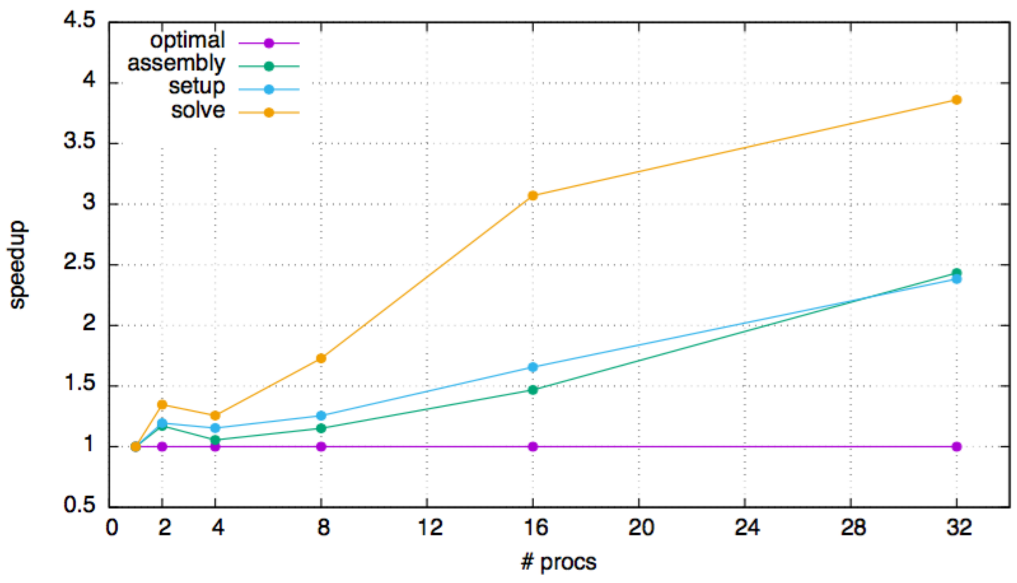


Figure 3.12: Weak scalability for Laplace problem with IgaHandler. Degree $p = 4$, C^0 continuity.

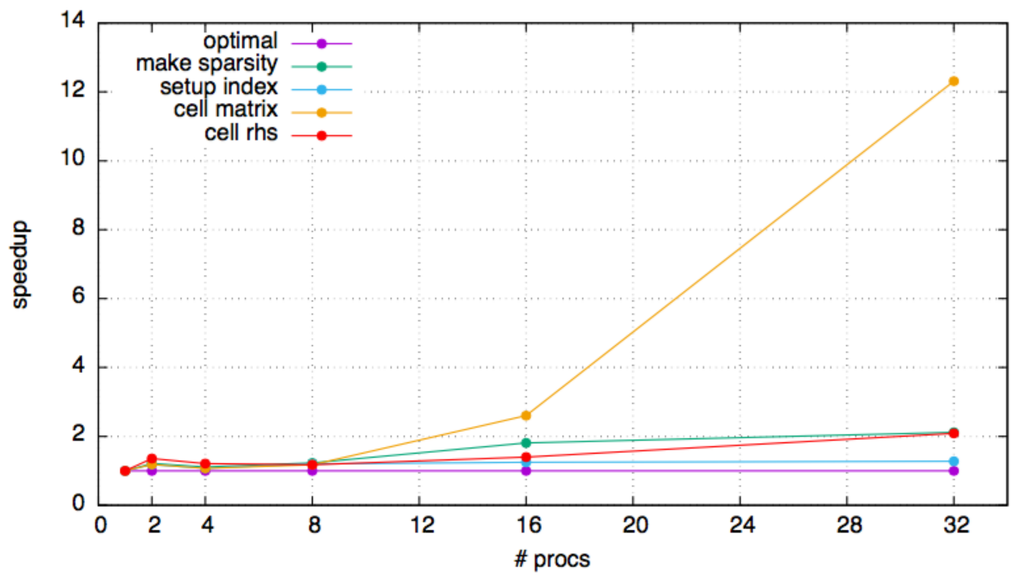


Figure 3.13: Weak scalability for IgaHandler class. Degree $p = 4$, C^0 continuity.

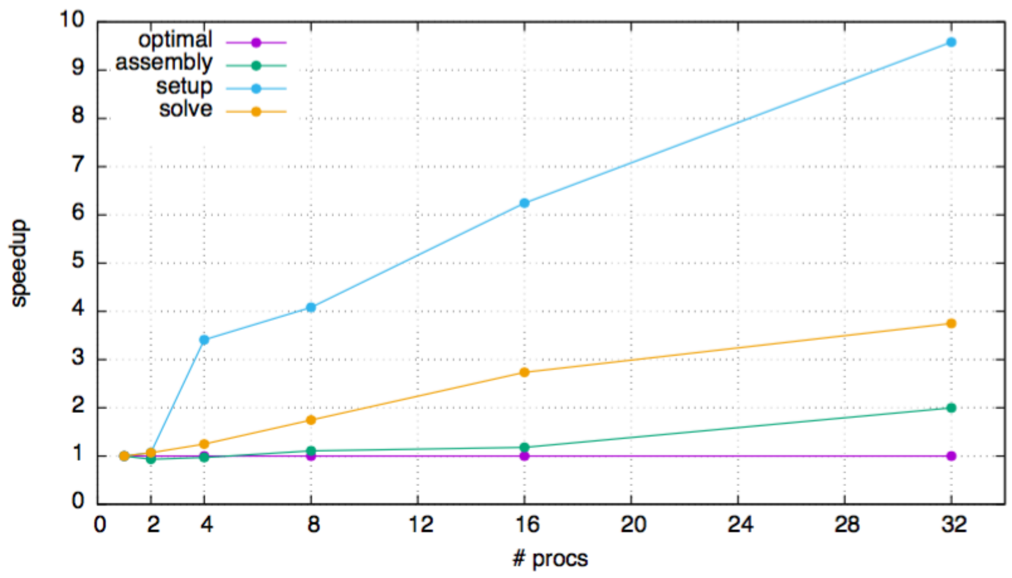


Figure 3.14: Weak scalability for Laplace problem with IgaHandler. Degree $p = 4$, C^3 continuity.

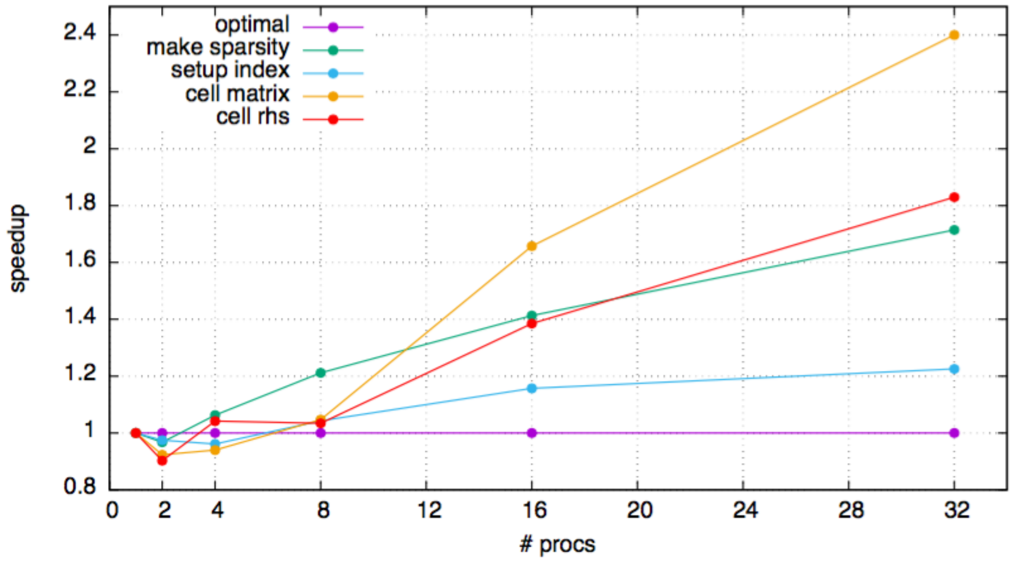


Figure 3.15: Weak scalability for IgaHandler class. Degree $p = 4$, C^3 continuity.

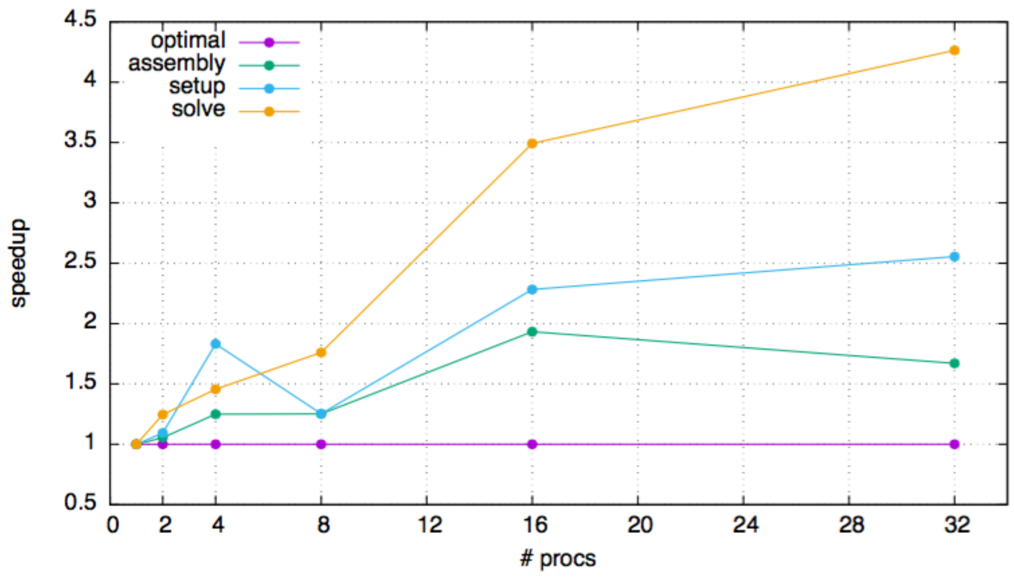


Figure 3.16: Weak scalability for Laplace problem with IgaHandler. Degree $p = 5$, C^0 continuity.

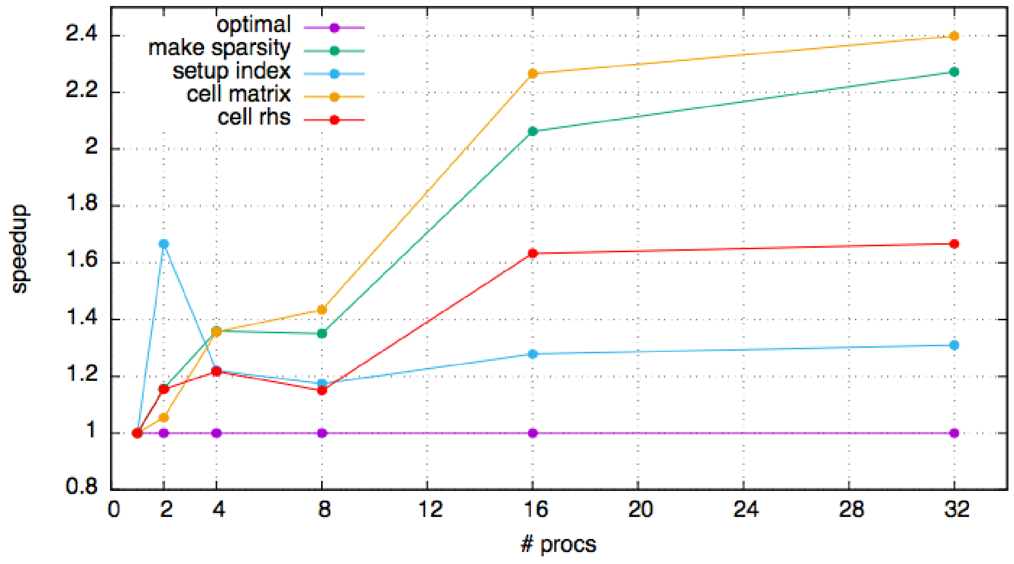


Figure 3.17: Weak scalability for IgaHandler class. Degree $p = 5$, C^0 continuity.

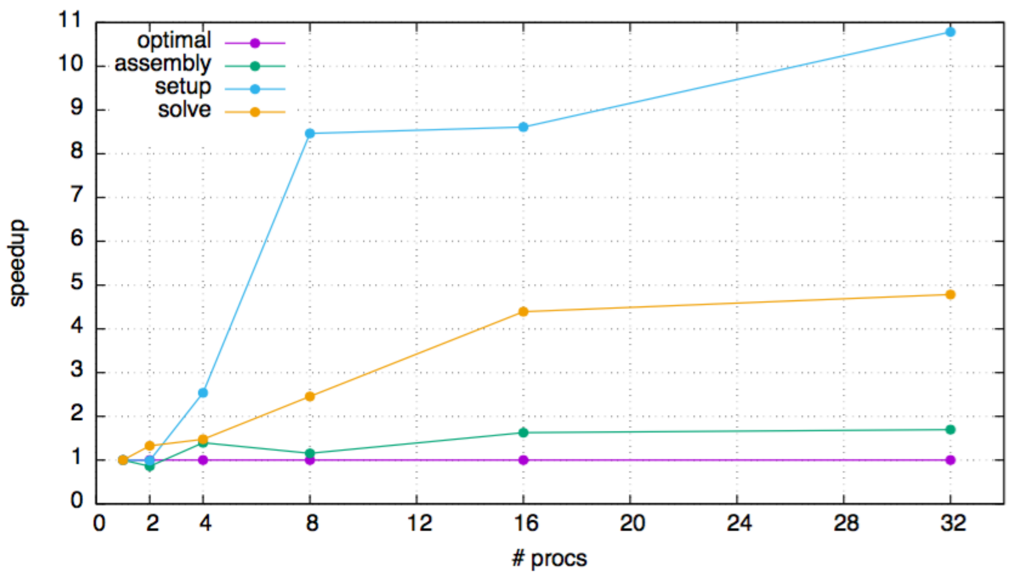


Figure 3.18: Weak scalability for Laplace problem with IgaHandler. Degree $p = 5$, C^4 continuity.

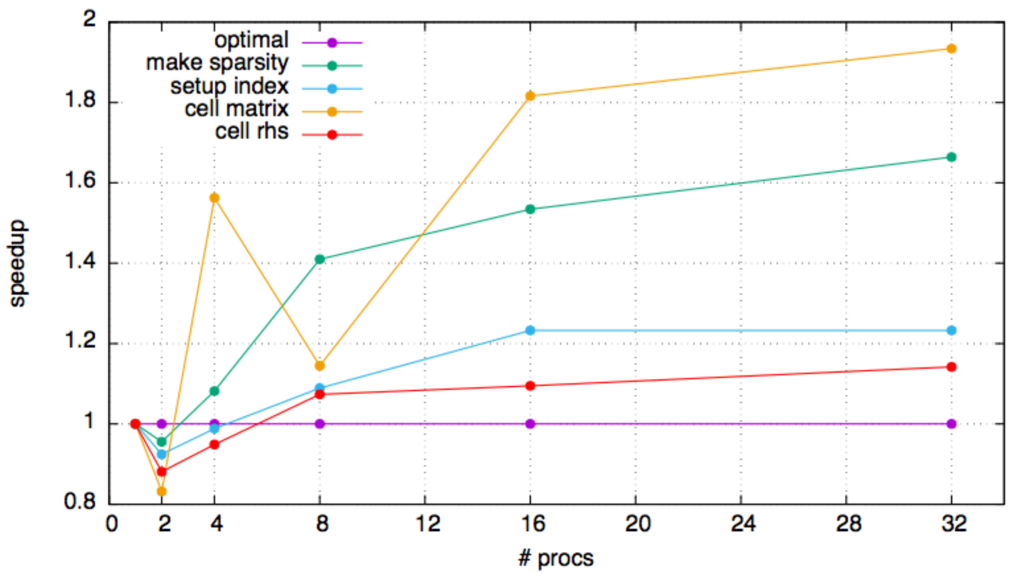


Figure 3.19: Weak scalability for IgaHandler class. Degree $p = 5$, C^4 continuity.

Conclusions

In this thesis we establish a one-to-one mapping between classical finite elements data structures and IGA data structures. The `deal.II` library proved to be an ideal framework to host this implementation. The comparison with existing data structures highlighted that three major ingredients are required in this process, respectively: a radical isoparametric concept, coupled with Bernstein polynomials as reference elements and a technique to restore higher interelement continuity.

We explored and presented our solution for the serial version. Then we explained in details how we parallelized it.

Our implementation allows to integrate IGA data structures in an easy way also in a parallel context. Moreover we take advantage of the existing `deal.II` infrastructure. In this way we benefit from already implemented massively parallel technologies. The IGA specific methods introduced exploit the Intel® MKL in a very effective way with a simple domain decomposition.

Bibliography

[1] B. S. Anmol Goyal. On penalty-free formulations for multipatch isogeometric kirchhoff-love shells. submitted.

[2] P. F. Antonietti, L. B. da Veiga, and M. Verani. A mimetic discretization of elliptic obstacle problems. Technical Report 14, MOX, 2010.

[3] D. N. Arnold, R. S. Falk, and R. Winther. Finite element exterior calculus, homological techniques, and applications. *Acta numerica*, 15:1–155, 2006.

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[5] W. Bangerth and T. Heister. What makes computational open source software libraries successful? *Computational Science & Discovery*, 6:015010/1–18, 2013.

[6] W. Bangerth and T. Heister. Quo vadis, scientific software? Editorial, SIAM News, January 2014.

[7] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, and B. Turcksin. The `deal.II` library, version 8.3. *preprint*, 2015.

[8] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. Young. The dealii library, version 8.2. *Archive of Numerical Software*, 3(100), 2015.

[9] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. D. Young. The `deal.ii` library, version 8.1. *arXiv preprint* `http://arxiv.org/abs/1312.2266v4`, 2013.

[10] W. Bangerth, T. Heister, and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*. `http://www.dealii.org`.

[11] Y. Bazilevs, L. Beirao da Veiga, J. Cottrell, T. Hughes, and G. Sangalli. Isogeometric analysis: approximation, stability and error estimates for h-refined meshes. *Mathematical Models and Methods in Applied Sciences*, 16(07):1031–1090, 2006.

[12] 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(5-8):229–263, 2010.

[13] L. Beirão Da Veiga, A. Buffa, J. Rivas, and G. Sangalli. Some estimates for h-p-k-refinement in Isogeometric Analysis. *Numerische Mathematik*, 118(2):271–305, 2011.

[14] L. Beirão Da Veiga, D. Cho, L. F. Pavarino, and S. Scacchi. Bddc preconditioners for isogeometric analysis. *Mathematical Models and Methods in Applied Sciences*, 23(06):1099–1142, 2013.

[15] L. Beirão Da Veiga, T. Hughes, J. Kiendl, C. Lovadina, J. Niiranen, A. Reali, and H. Speleers. A locking-free model for reissner–mindlin plates: Analysis and isogeometric implementation via nurbs and triangular nurps. *Mathematical Models and Methods in Applied Sciences*, 25(08):1519–1551, 2015.

[16] 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. (August 2010):15–47, 2011.

[17] S. C. Brenner and R. Scott. *The mathematical theory of finite element methods*. Springer, 2008.

[18] H. Brezis. *Functional analysis, Sobolev spaces and partial differential equations*. Springer, 2011.

[19] J. Bueno, C. Bona-Casas, Y. Bazilevs, and H. Gomez. Interaction of complex fluids and solids: theory, algorithms and application to phase-change-driven implosion. *Computational Mechanics*, pages 1–14, 2014.

[20] 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(23–24):1437–1445, 2010.

[21] N. Cavallini, O. Weeger, M. S. Pauletti, M. Martinelli, and P. Antolín. Effective integration of sophisticated operators in isogeometric analysis with igatools. In *Isogeometric Analysis and Applications, IGAA 2014*, pages 1–8. Springer, 2015.

[22] P. G. Ciarlet. *The finite element method for elliptic problems*. Access Online via Elsevier, 1978.

[23] N. Collier, L. Dalcin, and V. M. Calo. Petiga: high-performance isogeometric analysis. *arXiv preprint arXiv:1305.4452*, 2013.

[24] J. A. Cottrell, T. J. Hughes, and Y. Bazilevs. *Isogeometric Analysis: Toward Integration of CAD and FEA*. John Wiley & Sons, 2009.

[25] M. G. Cox. The numerical evaluation of B-splines. *IMA Journal of Applied Mathematics*, 10(2):134–149, 1972.

[26] B. Da Veiga, L. F. Pavarino, S. Scacchi, O. B. Widlund, and S. Zampini. Isogeometric bddc preconditioners with deluxe scaling. *SIAM Journal on Scientific Computing*, 36(3):A1118–A1139, 2014.

[27] A.-V. V. Daniela Fußeder, Bernd Simeon. Fundamental aspects of shape optimization in the context of isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, 286:313–331, 2015.

[28] L. De, J. Lorenzis, T. Evans, and A. R. Hughes. Isogeometric collocation: Neumann boundary conditions and contact, ices report 14-06. *The University of Texas at Austin*, 2014.

[29] C. De Boor. On calculating with B-splines. *Journal of Approximation Theory*, 6(1):50–62, 1972.

[30] C. De Falco, A. Reali, and R. Vázquez. Geopdes: a research tool for isogeometric analysis of pdes. *Advances in Engineering Software*, 42(12):1020–1034, 2011.

[31] L. De Lorenzis, P. Wriggers, and T. J. Hughes. Isogeometric contact: a review. *GAMM-Mitteilungen*, 37(1):85–123, 2014.

[32] L. De Lorenzis, P. Wriggers, and G. Zavarise. A mortar formulation for 3d large deformation contact using nurbs-based isogeometric analysis and the augmented lagrangian method. *Computational Mechanics*, 49(1):1–20, 2012.

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

[34] T. Dokken, T. Lyche, and K. F. Pettersen. Polynomial splines over locally refined box-partitions. *Computer Aided Geometric Design*, 30(3):331–356, 2013.

[35] M. Donatelli, C. Garoni, C. Manni, S. Serra-Capizzano, and H. Speleers. Robust and optimal multi-iterative techniques for iga collocation linear systems. *Computer Methods in Applied Mechanics and Engineering*, 284(0):1120 – 1146, 2015. Isogeometric Analysis Special Issue.

[36] M. Donatelli, C. Garoni, C. Manni, S. Serra-Capizzano, and H. Speleers. Robust and optimal multi-iterative techniques for iga galerkin linear systems. *Computer Methods in Applied Mechanics and Engineering*, 284(0):230 – 264, 2015. Isogeometric Analysis Special Issue.

[37] M. R. Dörfel, B. Jüttler, and B. Simeon. Adaptive isogeometric analysis by local h-refinement with t-splines. *Computer methods in applied mechanics and engineering*, 199(5):264–275, 2010.

[38] 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(08):1421–1478, 2013.

[39] L. C. Evans. *Partial differential equations*. American Mathematics Society, 1998.

[40] G. E. Farin. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann, 2002.

[41] J. Frohne and W. Bangerth. Step 41. `http://www.dealii.org/developer/doxygen/deal.II/step_41.html`, 2012.

[42] J. Frohne, W. Bangerth, and T. Heister. *Efficient Numerical Methods for the Large Scale, Parallel Solution of Elastoplastic Contact Problems*. Univ., 2014.

[43] K. Gahalaut, S. Tomar, and J. Kraus. Algebraic multilevel preconditioning in isogeometric analysis: Construction and numerical studies. *Computer Methods in Applied Mechanics and Engineering*, 266(0):40 – 56, 2013.

[44] C. Giannelli, B. Jüttler, S. Kleiss, A. Mantzaflaris, B. Simeon, and J. Špeh. Thb-splines: an effective mathematical technology for adaptive refinement in geometric design and isogeometric analysis. submitted.

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

[46] M. U. Guide. The mathworks. *Inc., Natick, MA*, 5:333, 1998.

[47] J. O. Hallquist et al. Ls-dyna theory manual. *Livermore software Technology corporation*, 3, 2006.

[48] J. S. Hansen. *GNU Octave: Beginner's Guide: Become a Proficient Octave User by Learning this High-level Scientific Numerical Tool from the Ground Up*. Packt Publishing Ltd, 2011.

[49] R. Hartmann. Higher order Boundary approximation. `http://www.dealii.org/8.0.0/reports/mapping_q/index.html`, 2001.

[50] L. Heltai, M. Arroyo, and A. DeSimone. Nonsingular isogeometric boundary element method for stokes flows in 3d. *Computer Methods in Applied Mechanics and Engineering*, 268:514–539, 2014.

[51] M. Hintermüller, K. Ito, and K. Kunisch. The primal-dual active set strategy as a semismooth newton method. *SIAM Journal on Optimization*, 13(3):865–888, 2002.

[52] M.-C. Hsu, Y. Bazilevs, V. M. Calo, T. E. Tezduyar, and T. J. Hughes. Improving stability of stabilized and multiscale formulations in flow simulations at small time steps. *Computer Methods in Applied Mechanics and Engineering*, 199(13):828–840, 2010.

[53] M.-C. Hsu, D. Kamensky, F. Xu, J. Kiendl, C. Wang, M. C. Wu, J. Mineroff, A. Reali, Y. Bazilevs, and M. S. Sacks. Dynamic and fluid–structure interaction simulations of bioprosthetic heart valves using parametric design with t-splines and fung-type material models. *Computational Mechanics*, pages 1–15, 2015.

[54] T. J. Hughes, J. A. Cottrell, and Y. Bazilevs. Isogeometric Analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. *Computer methods in applied mechanics and engineering*, 194(39):4135–4195, 2005.

[55] K. I. Joy. Bernstein polynomials. *On-line Geometric Modelling Notes*, 2000.

[56] B. Jüttler, U. Langer, A. Mantzaflaris, S. E. Moore, and W. Zulehner. Geometry+ simulation modules: Implementing isogeometric analysis. *PAMM*, 14(1):961–962, 2014.

[57] J. Kiendl, M.-C. Hsu, M. C. Wu, and A. Reali. Isogeometric kirchhoff–love shell formulations for general hyperelastic materials. *Computer Methods in Applied Mechanics and Engineering*, 291:280–303, 2015.

[58] G. Kiss, C. Giannelli, U. Zore, B. Jüttler, D. Großmann, and J. Barner. Adaptive cad model (re-) construction with thb-splines. *Graphical models*, 76(5):273–288, 2014.

[59] A. Manzoni, F. Salmoiraghi, and L. Heltai. Reduced basis isogeometric methods (rb-iga) for the real-time simulation of potential flows about parametrized naca airfoils. *Computer Methods in Applied Mechanics and Engineering*, 284:1147–1180, 2015.

[60] D. Mokriš, B. Jüttler, and C. Giannelli. On the completeness of hierarchical tensor-product b-splines. *Journal of Computational and Applied Mathematics*, 271:53–70, 2014.

[61] S. Morganti, F. Auricchio, D. Benson, F. Gambarin, S. Hartmann, T. Hughes, and A. Reali. Patient-specific isogeometric structural analysis of aortic valve closure. *Computer Methods in Applied Mechanics and Engineering*, 284:508–520, 2015.

[62] B. S. Oliver Weeger, Utz Wever. Nonlinear frequency response analysis of structural vibrations. *Computational Mechanics*, 54(6):1477–1495, 2014.

[63] M. S. Pauletti, M. Martinelli, N. Cavallini, and P. Antolin. Igatools: An isogeometric analysis library. *I.M.A.T.I.-C.N.R.*, pages 1–27, 2014.

[64] L. Piegl and W. Tiller. The NURBS Book. Monograph in Visual Communication. *Springer-Verlag*, 1997.

[65] S. Salsa and G. Verzini. *Equazioni a derivate parziali*. Springer, 2004.

[66] D. Schillinger, J. A. Evans, A. Reali, M. A. Scott, and T. J. Hughes. Isogeometric collocation: Cost comparison with galerkin methods and extension to adaptive hierarchical {NURBS} discretizations. *Computer Methods in Applied Mechanics and Engineering*, 267(0):170 – 232, 2013.

[67] L. Schumaker. *Spline Functions: Basic Theory*. Cambridge University Press, Cambridge, 2007.

[68] M. Scott, X. Li, T. Sederberg, and T. Hughes. Local refinement of analysis-suitable t-splines. *Computer Methods in Applied Mechanics and Engineering*, 213:206–222, 2012.

[69] K. Takizawa, B. Henicke, T. E. Tezduyar, M.-C. Hsu, and Y. Bazilevs. Stabilized space–time computation of wind-turbine rotor aerodynamics. *Computational Mechanics*, 48(3):333–344, 2011.

[70] R. Tremolieres, J.-L. Lions, and R. Glowinski. *Numerical analysis of variational inequalities*. Elsevier, 2011.

[71] A. V. Vuong, C. Giannelli, B. Jüttler, and B. Simeon. A hierarchical approach to adaptive local refinement in isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, 200(49-52):3554–3567, 2011.