Pigasus : Python for IsoGeometric AnalysiS and Unified Simulations.

Ahmed Ratnani

To cite this version:

Ahmed Ratnani. Pigasus : Python for IsoGeometric AnalysiS and Unified Simulations.. [Technical Report] 2012. hal-00769225

HAL Id: hal-00769225

https://hal.inria.fr/hal-00769225

Submitted on 30 Dec 2012

Πgasus : Python for IsoGeometric AnalysiS and Unified Simulations

Ahmed RATNANI [1]
[1]*C.E.A./DSM/IRFM, Cadarache, FR*
ahmed.ratnani@cea.fr

Abstract

B-splines and NURBS (Non Uniform Rational B-splines) are widely used in CAD (Computer Aider Design) models. IGA (IsoGeometric Analysis) consists of using these functions to both define the geometry and represent the unknowns that are solution of a Partial Differential Equation, using the Finite Element principle. In this paper we present a new library, namely **Πgasus** , that was developped in order to bring a common framework between the users (especially physicists) and mathematicians. We want to provide a stable and robust framework, that handles complex geometries and models as it is the case in Plasma Physics. Physicists will be able to use the recent works and results obtained by mathematicians. **Πgasus** is a 1D, 2D and 3D *Fortran* code, interfaced with *Python*. It provides a Geometry module, a FEM (Finite Element Method) computational engine and a Visualization module.

1 Introduction

The IsoGeometric Analysis has been introduced by Hughes et al [32], in order to bring a common framework between CAD (Computer Aided Design) and Numerical Simulations. During the whole process, including refinement and analysis, the geometry is maintained exact. In addition to the classical refinement strategies, *hp-refinement*, IGA introduces a new one, namelly the *k-refinement*. The latter offers the possiblity to control the regularity of the basis functions, by increasing or decreasing the multiplicities of knots. From this point of view, IGA can be seen as a generalization of standard FEMs where more regular functions are used.

In addition to handle complex and CAD geometries, it has been shown [18] that, the use of regular elements reduces the dimension of the Finite Element spaces, while keeping the desired precision. It allows to deal with higher order differential operators [25]. Moreover, it has been noticed [45], that the use of regular elements tends to give better CFL numbers.

For a comprehensive introduction on the subject, we may refeer the reader to the recent book by Cottrel et al [13]. For an algorithmic overview on B-splines we refeer to the book by DeBoor [20] or Piegl and Tiller [40]. For an extensive overview on the approximation theory, we may suggest the books [21, 50].

IGA is getting a large success inside the Engineering and Mathematical communities. The increasing number of publications shows the interest of these communities to this new approach. It has been used in electromagnetism [11, 10, 45], in incompressible fluid dynamics [9], in fluid-structure interaction [29, 6] or using a mapped Finite Volume method [26], in structural and contact mechanics [53, 33]. IGA has been used in Plasmas Physics, in MHD (Magneto-HydroDynamics) problems [46], and the Kinetic approach [14, 1, 4].

An active area of research is the study of local refinement. It seems that T-splines are getting more popular. Many studies are under investigation to understand the behavior of the discrete spaces [7, 17] or to glue 2 patchs [16], which is of a big importance for parallelization.

Another active area is how to construct a volume description of a computational domain, given by its boundary, so we can directly use all existing CAD-models. For those interested in this subject, we may refeer to the works [35, 36, 37, 12, 59, 60].

Some useful softwares can be found ([57], [19]) to accompany researchers and introduce them to this new subject. However, for real and complex applications, they present some disadvantages. They are inflexible for Numerical Simulations combined with Modeling. In the latter, the user may want to see the effect of adding a new term, or remove it, test a new time scheme, *etc* .... In order to solve partial differential equations (PDEs), one needs to copy the old written code, and then change it. Hence, we will end up with as many as codes, as the problems we have treated, even if the principal changes occure while assembling the element matrices.

All these points, motivated me to developp a new library, **Πgasus** , written in *Fortran* and interfaced with *Python*. The idea was to simplify as much as possible this interface, so we can recover the classical language of the Finite Element Method, while maintaining the advantages of a scripted-language. By sharing the library between different projects, we will keep the same *computational engine*, and so reduce redundancy. In addition, the architecture of **Πgasus** offers the user the ability to manipulate the geometry and refine it (using the *hpk-refinement* strategies), use high order and regular elements, and to manipulate the differential operators even for non-linear problems. The design of **Πgasus** gives the user some freedom to define the spaces and manipulate the grids (thus relax the isoparametric approach, by considering edge, surface, . . . elements).

Πgasus is not intend (for the moment) to be a concurrent of automated FEM softwares like *Freefem++* or *Feelpp* [41, 28].

This work was motivated by the increasing interactions between physicists and mathematicians in (computational) plasma physics, trying to answer the question : how can a mathematician study and improve physicists works and challenges, and make his developpements promptly available? It was initiated during my Phd-thesis [44] at the *INRIA*, and pursued at the *CEA*. We wanted to provide both physicists and mathematicians a common framework where they can developp their researchs, take advantage from the IGA approach, but also in the future for *production* cases.

As the reader will notice it, the user does not need to have a deep knowledge on CAD or B-splines. The important point is to know how to derive a variational (weak) formulation, and then ask **Πgasus** to assemble the different parts of the formulation. **Πgasus** is delivered with a ($2D$) simple and easy GUI interface to create and manipulate the computational domains. The user can move and manipulate the control points in order to generate the geometry. The basic distribution of **Πgasus** does not need any third package or library.

Throughout this article, we will present 2D studies even if the code treats also the 1D and 3D cases. The current paper is structured as follows. In section 2, we give a brief overview on B-splines, NURBS and IsoGeometric Analysis approach. In section 3, we present **Πgasus** on a simple example (Poisson's equation). We have made an effort to link each of **Πgasus** 's functions to the FEM language. In section 4, we give much more details on **Πgasus** : its architecture, the important concepts, utilities and features. Finally, in section 5, we a non-linear $2D$ pde as a simple application. This article is not a tutorial, but we have tried to present as mush as possible the different and interesting parts.

2 An overview on IsoGeometric Analysis

More details on the subject can be found in [21, 50, 40, 20, 13].

2.1 Basic properties of B-splines

Definition 2.1 (B-Spline) *Let $X = \{x_0, \cdots, x_p\}$ a non-decreasing sequence of $p+1$ points such that $x_0 \neq x_p$. The B-Spline is defined by the following reccurence formula:*

$$N(x; x_0, \cdots, x_p) = \frac{x - x_0}{x_{p-1} - x_0} N(x; x_0, \cdots, x_{p-1}) + \frac{x_p - x}{x_p - x_1} N(x; x_1, \cdots, x_p) \tag{2.1}$$

the initialization is given by : $N(x; x_0, x_1) = 1$, if $x_0 \neq x_1$, and 0 otherwise.

In order to construct a *B-spline* of degree p, we need $p+1$ points which are called knots in the spline terminology. Then, to create a family of *B-splines*, we will need to have a non-decreasing sequence of knots, also called knot vector.

Let $T = (t_i)_{1 \leqslant i \leqslant N+k}$ be a non-decreasing sequence of knots, with $k = p+1$. Each set of $p+1$ consecutive knots $T_j = \{t_j, \cdots, t_{j+p}\}$ will generate a *B-spline* N_j. This leads to the following definition:

Definition 2.2 (B-Spline series) *The j-th B-Spline of order k is defined by the recurrence relation:*

$$N_j^k = w_j^k N_j^{k-1} + (1 - w_{j+1}^k) N_{j+1}^{k-1}$$

where,

$$w_j^k(x) = \frac{x - t_j}{t_{j+k-1} - t_j} \qquad\qquad N_j^1(x) = \chi_{[t_j, t_{j+1}[}(x)$$

for $k \geq 1$ and $1 \leq j \leq N$.

In figures 1 and 2, we give examples of the generated B-splines families depending on the knot vectors (figure 1) and the order (figure 2).

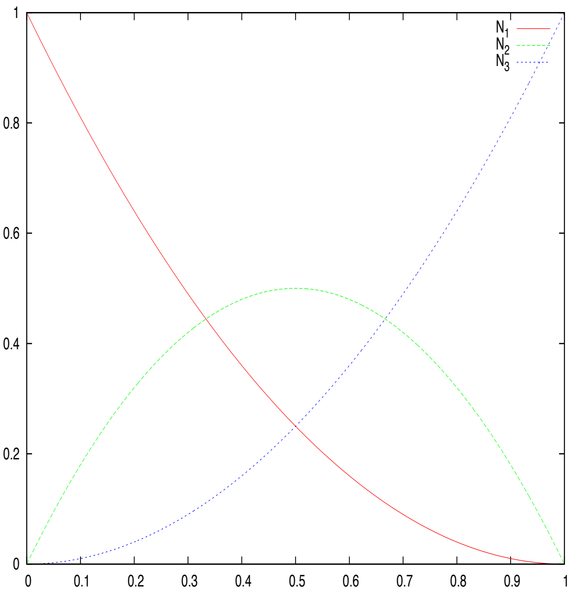

Figure 1: B-splines functions associated to the knot vector $T = \{000\ 111\}$, of order $k = 3$. These are Bernstein polynomials

We note some important properties of a B-splines basis:

- B-splines are piecewise polynomial of degree $p = k - 1$,

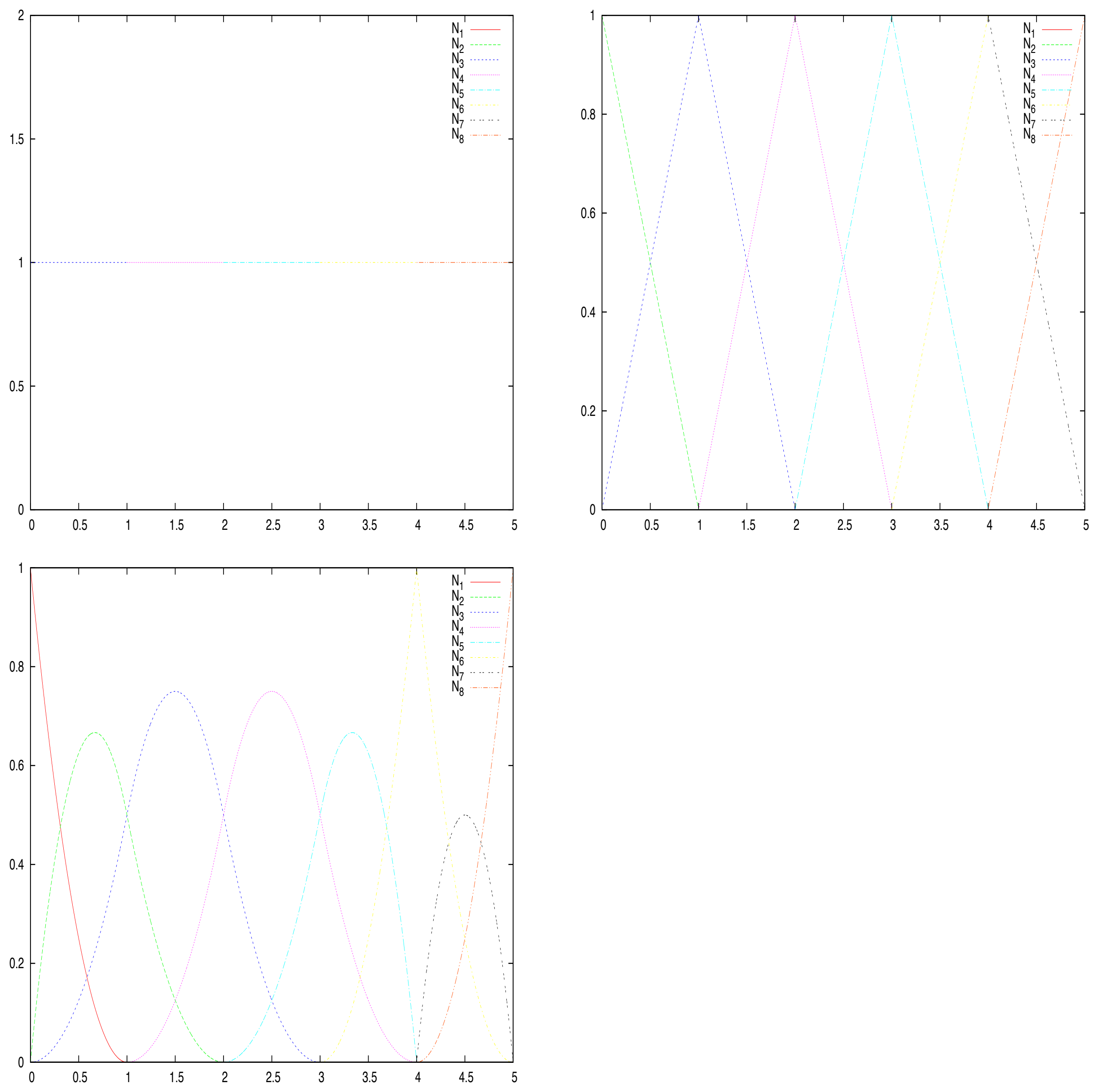

Figure 2: B-splines functions associated to the knot vector $T = \{000\ 1\ 2\ 3\ 44\ 555\}$, of order $k = 1, 2, 3$ (left to right).

- Compact support; the support of N_j^k is contained in $[t_j, t_{j+k}]$,
- if $x \in\ ]t_j, t_{j+1}[$, then only the *B-splines* $\{N_{j-k+1}^k, \cdots, N_j^k\}$ are non vanishing at x,
- Positivity: $\forall j \in \{1, \cdots, N\}\ \ N_j(x) > 0,\ \ \forall x \in]t_j, t_{j+k}[$,
- Partition of unity : $\sum_{i=1}^N N_i^k(x) = 1, \forall x \in \mathbb{R}$,
- Local linear independence, and finally
- If a knot t_i has a multiplicity m_i then the B-spline is $\mathcal{C}^{(p-m_i)}$ at t_i

2.2 Multivariate tensor product splines

Let us consider d knot vectors $\mathcal{T} = \{T^1, T^2, \cdots, T^d\}$. For simplicity, we consider that those knot vectors are open, which means that k knots on each side are duplicated so that the spline is interpolating on the boundary, and of bounds 0 and 1. In the sequel we will use the notation $I = [0, 1]$. Each knot vector T^i, will generate a basis for a Schoenberg space [21], $\mathcal{S}_{k_i}(T^i, I)$. The tensor product of all those spaces is also a Schoenberg space, namely $\mathcal{S}_{\mathbf{k}}(\mathcal{T})$, where $\mathbf{k} = \{k_1, \cdots, k_d\}$. The hypercube $\mathcal{P} = I^d = [0, 1]^d$, will be referred to as a patch.
The basis for $\mathcal{S}_{\mathbf{k}}(\mathcal{T})$ is defined by a tensor product :

$$N_{\mathbf{i}}^{\mathbf{k}} := N_{i_1}^{k_1} \otimes N_{i_2}^{k_2} \otimes \cdots \otimes N_{i_d}^{k_d}$$

where, $\mathbf{i} = \{i_1, \cdots, i_d\}$.
A typical cell from $\mathcal{P}$ is a cube of the form : $Q_{\mathbf{i}} = [\xi_{i_1}, \xi_{i_1+1}] \otimes \cdots \otimes [\xi_{i_d}, \xi_{i_d+1}]$. To any cell Q, we will associate its extension $\widetilde{Q}$, which is the union of the supports of basis functions, that intersects Q.

2.3 Splines in CAD

In order to have a control on the regularity of the curve, we need to use a piecewise-polynomial form. This is why the use of *B-splines* has known a large success. Moreover, the control points (and polygone) have many geometric interpretations.
Let $(\mathbf{P}_i)_{1 \leqslant i \leqslant N} \in \mathbb{R}^d$ be a sequence of control points, forming a control polygon.

Definition 2.3 (B-Spline curve) *The B-spline curve in $\mathbb{R}^d$ associated to $T = (t_i)_{1 \leqslant i \leqslant N+k}$ and $(\mathbf{P}_i)_{1 \leqslant i \leqslant N}$ is defined by :*

$$\mathcal{C}(t) = \sum_{i=1}^{N} N_i^k(t) \boldsymbol{P}_i$$

We have the following properties for a *B-spline* curve:

- If $N = k$, then $\mathcal{C}$ is just a Bézier-curve,
- $\mathcal{C}$ is a piecewise polynomial curve,
- The curve interpolates its extremas if the associated multiplicity of the first and the last knot are maximum (*i.e.* equal to k),
- Invariance with respect to affine transformations,
- Strong convex-hull property:
  if $t_i \leq t \leq t_{i+1}$, then $\mathcal{C}(t)$ is inside the convex-hull associated to the control points $\mathbf{P}_{i-p}, \cdots, \mathbf{P}_i$,
- Local modification : moving $\mathbf{P}_i$ affects $\mathcal{C}(t)$, only in the interval $[t_i, t_{i+k}]$,
- The control polygon approaches the behavior of the curve.

Remark 2.4 *Remark that there is a kind of duality between knots and control points. We can use multiple control points : $\mathbf{P}_i = \mathbf{P}_{i+1}$, instead of multiple knots.*

Deriving a B-spline curve: We have:

$$\mathcal{C}'(t) = \sum_{i=1}^{n} N_i^{k'}(t)\mathbf{P}_i = \sum_{i=1}^{n}\left(\frac{p}{t_{i+p}-t_i}N_i^{k-1}(t)\mathbf{P}_i - \frac{p}{t_{i+1+p}-t_{i+1}}N_{i+1}^{k-1}(t)\mathbf{P}_i\right) = \sum_{i=1}^{n-1} N_i^{k-1^*}(t)\mathbf{Q}_i \tag{2.2}$$

where $\mathbf{Q}_i = p\frac{\mathbf{P}_{i+1}-\mathbf{P}_i}{t_{i+1+p}-t_{i+1}}$, and $\{N_i^{k-1^*},\ \ 1 \leq i \leq n-1\}$ are generated using the knot vector T^* which is obtained from T by reducing by one the multiplicity of the first and the last knot (in the case of open knot vector), *i.e.* by removing the first and the last knot.

Example: $T = \{000\ \frac{2}{5}\ \frac{3}{5}\ 111\},\ \ p = 2,\ \ n = 5.$

We have $\mathcal{C}(t) = \sum_{i=1}^{5} N_i^{3'}(t)\mathbf{P}_i$, then

$$\mathcal{C}'(t) = \sum_{i=1}^{4} N_i^{2^*}(t)\mathbf{Q}_i$$

where

$$\mathbf{Q}_1 = 5\{\mathbf{P}_2 - \mathbf{P}_1\},\quad \mathbf{Q}_2 = \frac{10}{3}\{\mathbf{P}_3 - \mathbf{P}_2\},$$
$$\mathbf{Q}_3 = \frac{10}{3}\{\mathbf{P}_4 - \mathbf{P}_3\},\quad \mathbf{Q}_4 = 5\{\mathbf{P}_5 - \mathbf{P}_4\}.$$

The *B-splines* $\{N_i^{2^*},\ \ 1 \leq i \leq 4\}$ are associated to the knot vector $T^* = \{00\ \frac{2}{5}\ \frac{3}{5}\ 11\}$.
In figure 3, we show an example of a B-spline curve. In this case, the B-splines generated are simply Berstein polynomials, and the obtained curve is a Bézier curve.

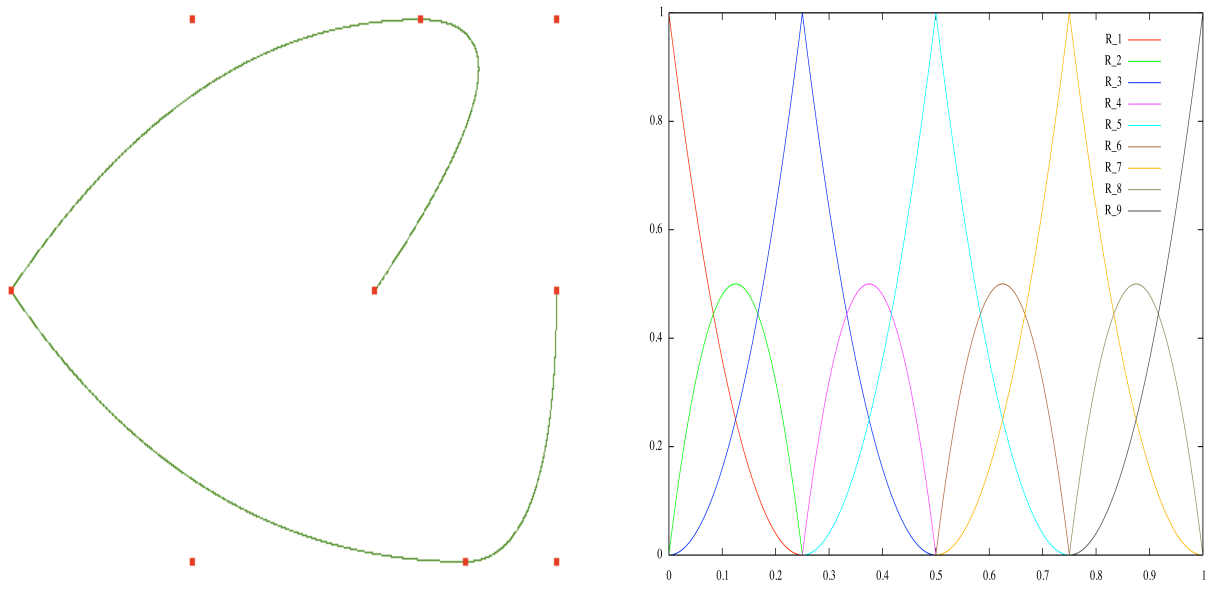

Figure 3: (left) A B-spline curve and its control points, (right) B-splines functions used to draw the curve. $N = 9, p = 2$, $T = \{000, \frac{1}{4}\frac{1}{4}, \frac{1}{2}\frac{1}{2}, \frac{3}{4}\frac{3}{4}, 111\}$

Definition 2.5 (*B-spline* surface) *The B-spline surface of order k associated to the knot vectors $\{T_\xi, T_\eta\}$ and the control points $(\boldsymbol{P}_{i,j})_{1\leqslant i\leqslant N_1, 1\leqslant j\leqslant N_2}$, is defined by*

$$\boldsymbol{M}(\xi,\eta) = \sum_{i=1}^{N_1}\sum_{j=1}^{N_2} N_{i,j}(\xi,\eta)\boldsymbol{P}_{i,j}$$

with $N_{i,j}(\xi,\eta) = N_i^{(1)}(\xi)N_j^{(2)}(\eta)$

2.4 Fundamental geometric operations

We can change the curve's parameters without changing the curve. This refers as to geometric operations. For example, we can elevate the spline's degree or insert new knots. Many algorithms are available from the CAD community, and have proved their efficiency [20, 38, 30, 42, 34].
In figure 4, we show the use of some of these geometric operations (elevation degree and knot insertion) on a quadratic B-Spline curve. We show also how one can split a domain by raising the multiplicity of knots to the spline's degree. This is an important strategy that may help us for domain decomposition and parallelization.
In figure 5, we show the use of the elevation degree and insertion knots algorithms on a $2D$ domain. As we can see, the geometry is kept unchanged.

2.5 *NURBS*

Let $\omega = (\omega_i)_{1 \leqslant i \leqslant N}$ be a sequence of non-negative reals. The *NURBS* (Non-Uniform Rational B-splines) functions are defined by a projective transformation:

Definition 2.6 (*NURBS*) *The i-th NURBS of order k, associated to the knot vector T and the weights ω, is defined by*

$$R_i^k = \frac{\omega_i N_i^k}{\sum_{j=1}^N \omega_j N_j^k}. \tag{2.3}$$

Notice that when the weights are equal to 1 the *NURBS* are *B-splines*.

Definition 2.7 (*NURBS* curve) *The NURBS curve of order k associated to the knot vector T, the control points $(\boldsymbol{P}_i)_{1 \leqslant i \leqslant N}$ and the weights ω, is defined by*

$$\boldsymbol{M}(t) = \sum_{i=1}^N R_i^k(t)\boldsymbol{P}_i.$$

2.6 Modeling conics using *NURBS*

In this section, we will show how to construct an arc of conic, using rational *B-splines*. Let us consider the following knot vector : $T = \{000\ 111\}$, the generated *B-splines* are Bernstein polynomials. According to 2.3, the general form of a rational Bézier curve of degree 2 is:

$$\mathcal{C}(t) = \frac{\omega_1 N_1^2(t)\mathbf{P}_1 + \omega_2 N_2^2(t)\mathbf{P}_2 + \omega_3 N_3^2(t)\mathbf{P}_3}{\omega_1 N_1^2(t) + \omega_2 N_2^2(t) + \omega_3 N_3^2(t)} \tag{2.4}$$

Let us consider the case $\omega_1 = \omega_3 = 1$. Because of the multiplicity of the knots 0 and 1, the curve $\mathcal{C}$ is linking the control point $\mathbf{P}_1$ to $\mathbf{P}_3$. Depending on the value of ω_2, we get different type of curves (Table 1).

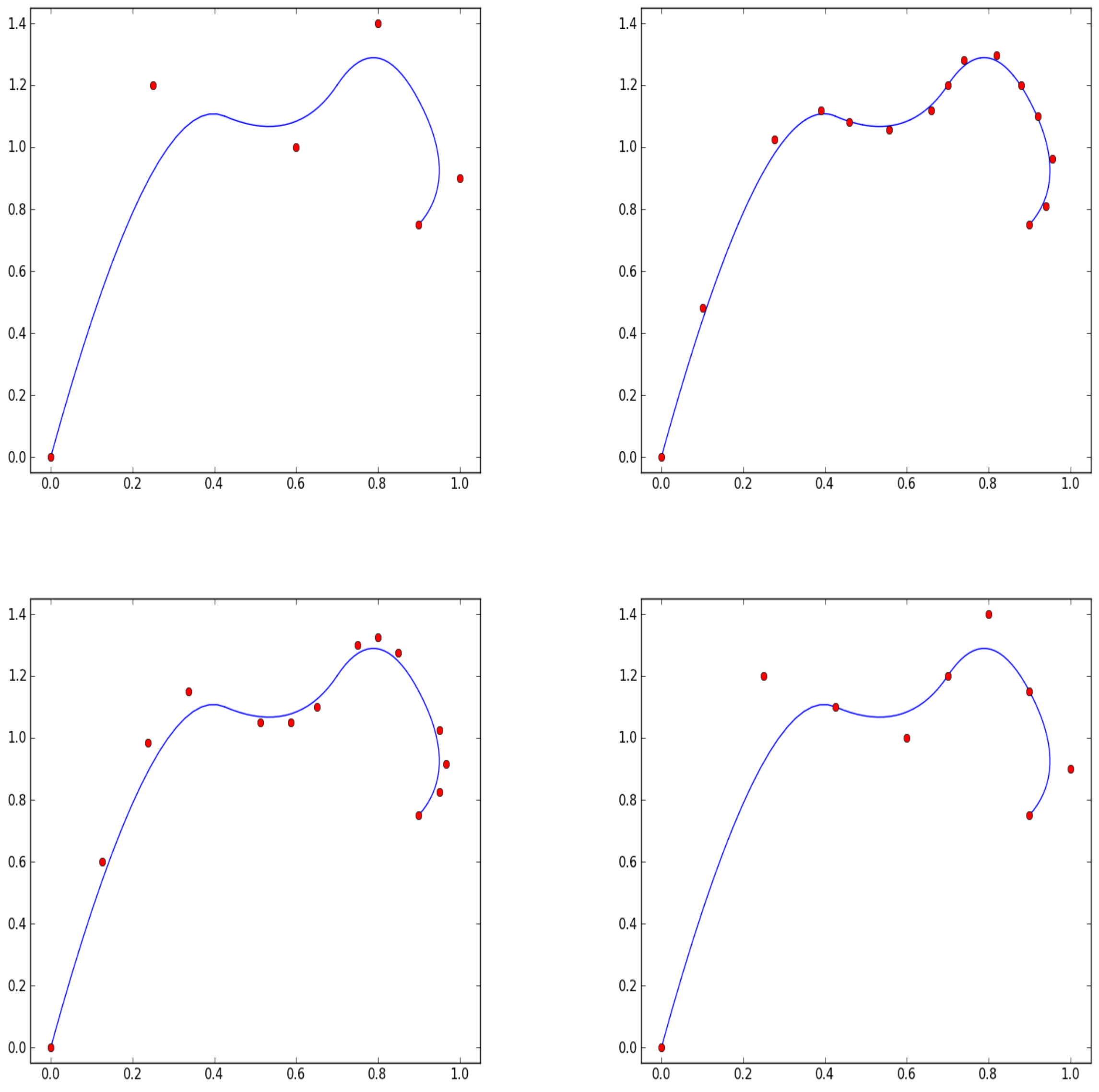

Figure 4: **First line** : (left) A quadratic B-spline curve and its control points. The knot vector is $T = \{000, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 111\}$. (right) The curve after a p-refinement, the degree was raised by 2. **Second line** : (left) The curve after a h-refinement by inserting the knots $\{0.1, 0.2, \cdots, 0.9\}$ while the degree is kept equal to 2. (right) The curve after duplicating the multiplicity of the internal knots $\{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$, this leads to a Bézier description. We can then, split the curve into 4 pieces (sub-domains), each one will corresponds to a quadratic Bézier curve.

2.7 Multivariate tensor product *NURBS*

As for splines, one can define multivariate tensor product *NURBS*. For surfaces, we have the following definition.

Definition 2.8 (*NURBS* surface) *The NURBS surface of order k associated to the knot vectors $\{T^{(1)}, T^{(2)}\}$,*

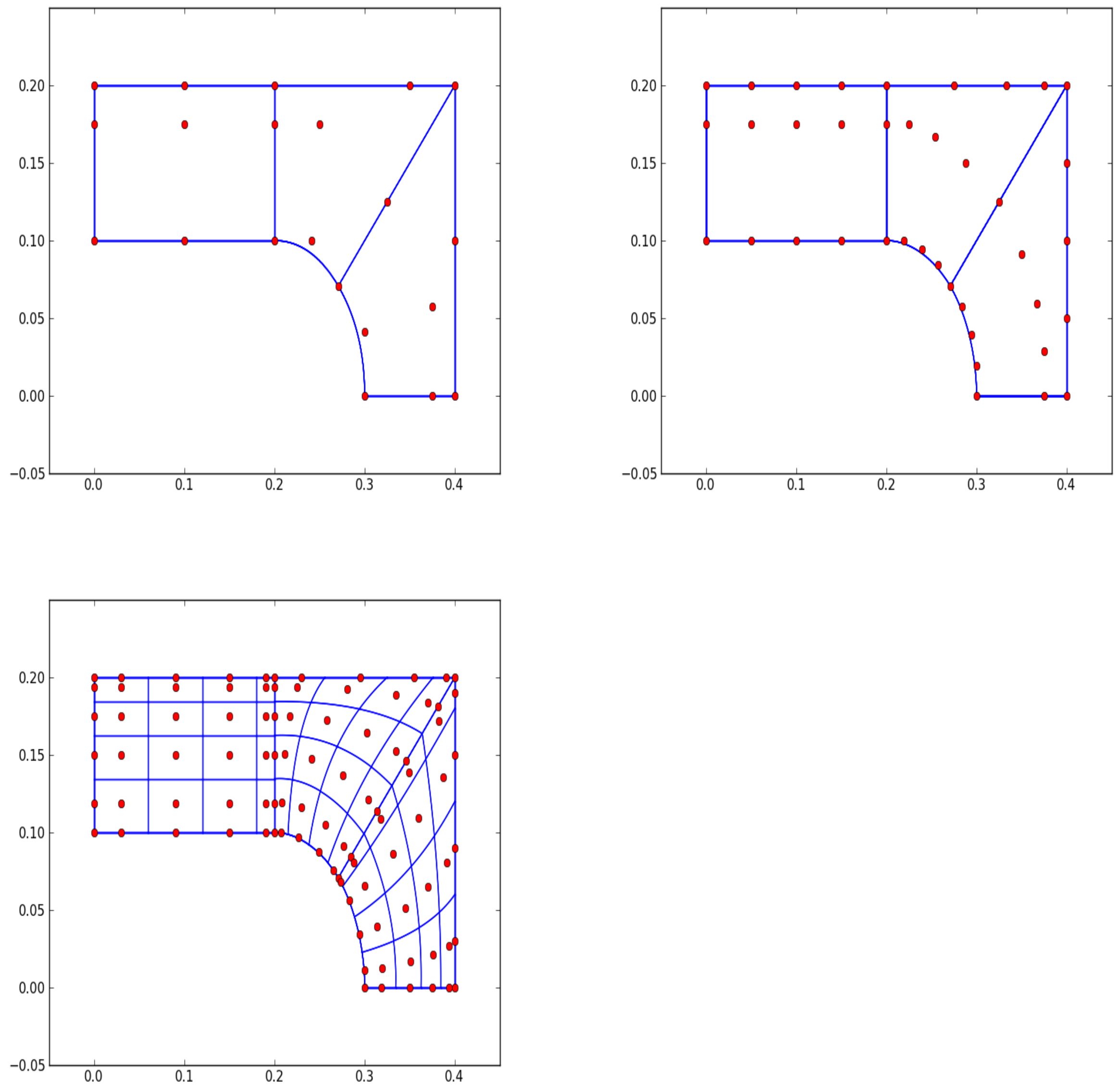

Figure 5: First line : (left) A quadratic NURBS description of a coil and its control points. The knot vector is $T = \{000, \frac{1}{3}\frac{1}{3}, \frac{2}{3}\frac{2}{3}, 111\}$. (right) The curve after a p-refinement, the degree was raised by 2. **Second line** : the curve after a h-refinement by inserting the knots $\{0.1, 0.2, \cdots, 0.9\}$ while the degree is kept equal to 2.

the control points $(\boldsymbol{P}_{i,j})_{1\leqslant i\leqslant N_1, 1\leqslant j\leqslant N_2}$ *and the weights* $(\omega_{i,j})_{1\leqslant i\leqslant N_1, 1\leqslant j\leqslant N_2}$*, is defined by*

$$\boldsymbol{M}(t^{(1)}, t^{(2)}) = \sum_{i=1}^{N_1}\sum_{j=1}^{N_2} R_{i,j}(t^{(1)}, t^{(2)})\boldsymbol{P}_{i,j}$$

with $R_{i,j}(t^{(1)}, t^{(2)}) = \frac{\omega_{i,j}N_i^1 N_j^2}{\sum_{r,s}\omega_{r,s}N_r^1 N_s^2}$

Remark 2.9 *NURBS functions inherit most of B-splines properties. Remark that in the interior of a knot span, all derivatives exist, and are rational functions with non vanishing denominator.*

	nature of the curve
$\omega_2 = 0$	line
$0 < \omega_2 < 1$	ellipse arc
$\omega_2 = 1$	parabolic arc
$\omega_2 > 1$	hyperbolic arc

Table 1: Modeling conics using *NURBS*.

We present here the definition of the perspective mapping. We construct the weighted control points $\boldsymbol{P}_i^\omega = (\omega_i x_i, \omega_i y_i, \omega_i z_i, \omega_i)$, then we define the B-spline curve in four-dimensional space as

$$\boldsymbol{M}^\omega(t) = \sum_{i=1}^{N} N_i^k(t)\boldsymbol{P}_i^\omega. \tag{2.5}$$

For fundamental geometric operations on NURBS curves, we use the latest transformation and algorithms on B-spline curves.

Remark 2.10 *NURBS functions allow us to model, exactly, much more domains than B-splines. In fact, all conics can be exactly represented with NURBS. For more details, see [34].*

2.8 IsoGeometric Analysis

The idea behind the IGA method is to use the same functions that define the physical domain, to approach the solution of a partial differential equation.
In the sequel, we consider 2 knot vectors $T_\xi = \{\xi_1, \cdots, \xi_{N_1+p_1+1}\}$ and $T_\eta = \{\eta_1, \cdots, \eta_{N_2+p_2+1}\}$. Let $W_\xi = \{\omega_1^\xi, \cdots, \omega_{N_1}^\xi\}$ and $W_\eta = \{\omega_1^\eta, \cdots, \omega_{N_2}^\eta\}$ be two weight sequences, and $(\mathbf{P}^{ij})_{1\leqslant i\leqslant N_1, 1\leqslant j\leqslant N_2}$ a sequence of control points. This defines a mapping

$$\mathbf{F}(\xi,\eta) = \sum_{1\leqslant i\leqslant N_1, 1\leqslant j\leqslant N_2} R_i^\xi(\xi) R_j^\eta(\eta)\mathbf{P}^{ij} \tag{2.6}$$

that maps the rectangular patch $[\xi_1, \xi_{N_1}] \times [\eta_1, \eta_{N_2}]$ onto the physical domain Ω. Where R^ξ and R^η are *NURBS* functions defined by knot vectors T_ξ and T_η, and weights W_ξ and W_η.
As said before, we consider only open knot vectors. Without loss of generality, we shall consider knot vectors of the form:

$$\xi_1 = \cdots = \xi_{p_1+1} = \eta_1 = \cdots = \eta_{p_2+1} = 0,$$

and

$$\xi_{N_1+1} = \cdots = \xi_{N_1+p_1+1} = \eta_{N_2+1} = \cdots = \eta_{N_2+p_2+1} = 1.$$

Let K be a cell in the physical domain. Q is the parametric associated cell and such that $K = \mathbf{F}(Q)$. Let $J_{\mathbf{F}}$ be the Jacobian of the transformation $\mathbf{F}$, that maps any parametric domain point (ξ, η) into the physical domain point (x, y) (figure 6).

2.8.1 Refinement strategies

Refining the grid can be done in 3 different ways. This is the most interesting aspects of B-splines basis,

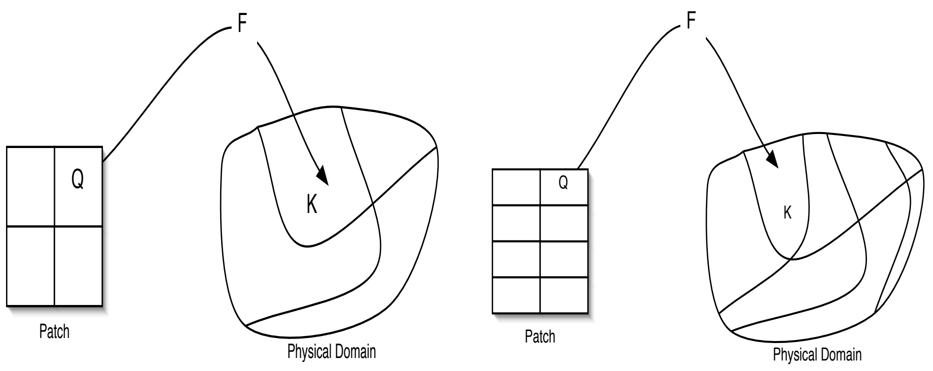

Figure 6: Mapping from the patch to the physical domain: (left) initial patch, (right) patch after h-refinement in the η direction. Here, we have $K = \mathbf{F}(Q)$

- using the patch parameter h, by inserting new knots. This is the h-refinement, it is the equivalent to mesh refinement of the classical Finite Element Method.
- using the degree p, by elevating the B-spline degree. This is the p-refinement, it is equivalent to use higher finite element order in the classical FEM.
- using the regularity of B-splines, by increasing / decreasing the multiplicity of inserted knots. This is the k-refinement. This new strategy does not have an equivalent in the classical FEM.

An active area of research is the study of local refinement. It is important to notice that the use of tensor products leads to the existence of a lot of superfluous control points, that might exist because of the presence of a cartesian grid in the parametric domain. Sederberg et al. [52] defined the notion of T-splines that allows us to reduce the number of those control points. In [22] Dörfel et al. use T-splines for local h-refinement in isogeometric analysis. It seems that T-splines are getting more popular. Many studies are under investigation to understand the behavior of the discrete spaces [7, 17] or to glue 2 patchs [16], which is of a big importance for parallelization. For more information on this subject, we refer to [48, 49, 39, 51, 8] or [58, 55] for PHT-splines. For details on implementation see [47]. In [56], authors propose the use of a hierarchical local refinement method. THB-splines are used in [24]. Dokken, Kvamsdal and their team from SINTEF are currently developing another approach for local refinement, based on LR-splines [54].

2.8.2 Patch

Starting from the geometric description of the domain, which gives a coarse mesh, we can then, use $h/p/k$ refinements to create the computational grid. We can also use multiple patches to describe more complex domains [32, 13]. There are many ways to stick those patches together.

2.8.3 Grid generation

For this purpose, we use alternatively h and p-refinement. The minimal degree of basis functions is imposed by the domain design. When inserting knots, we can use uniformly-spaced knots or non uniformly-spaced ones. In figure 7, we give an example of such a refinement.

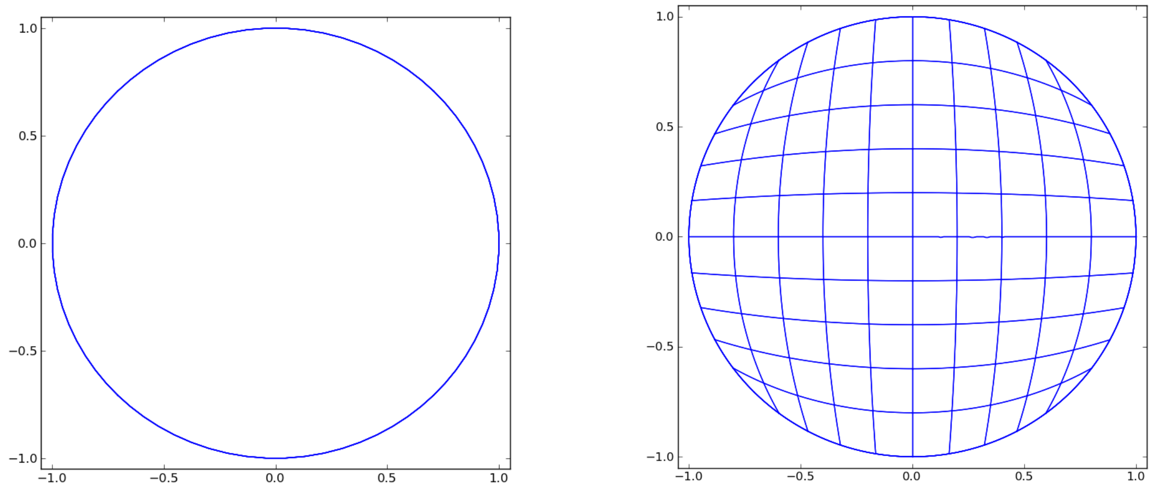

Figure 7: Grid generation: (left) The coarsest mesh, (right) Domain after h-refinement. The minimal degree of basis functions is 2 in this example.

3 Introducing Πgasus

In this section, we will consider the resolution of an elliptic partial differential equation on a square domain Ω. We will see how we can recover the classical FEM language, for the IGA approach. We also will separate the different differential operators in the equation and how we can construct them using **Πgasus** .

Let us consider the following problem:

For given functions A, f, b, find u such that:

$$\begin{cases} -\nabla \cdot (A \nabla u) + bu = f & , \Omega \\ u = 0 & , \partial\Omega \end{cases} \tag{3.7}$$

Introducing the matrix :

$$\Sigma = (\Sigma_{i,j})_{1 \leqslant i,j \leqslant n}$$

and the vectors :

$$L = (L_i)^T_{1 \leqslant i \leqslant n - n_D}$$

$$[u_h] = ([u_h]^i)^T_{1 \leqslant i \leqslant n}$$

where, we denote for $i, j \in \{1, .., n\}$

$$\Sigma_{i,j} = \int_\Omega \nabla\varphi_i \; A \; \nabla\varphi_j \; d\Omega + \int_\Omega b \; \varphi_i \; \varphi_j \; d\Omega$$

$$L_i = \int_\Omega f \; \varphi_i \; d\Omega$$

here, n denotes the dimension of the discrete space and n_D the number of Dirichlet elements.

Using a variational formulation and the Green's formulae, we know that after discretization, we have to solve the linear system:

$$\Sigma\,[u_h] = L\;.$$

The matrix Σ is nothing else but the sum of the Stiffness matrix S and Mass matrix M.

$$S = \int_\Omega \nabla\varphi_i\; A\;\nabla\varphi_j\; d\Omega$$

$$M = \int_\Omega b\;\varphi_i\;\varphi_j\; d\Omega$$

In practice, the user will need to define, discretize and assemble the two differential operators M and S. To do so, we must start by defining the discrete FE space (see subsection 3.1), with the appropriate boundary conditions and set the computational grid. Then, he needs to define two fields, one for the right hand side term, and the other for the solution of the pde (see subsection 3.2). Finally, he defines the discretized differential operators M and S (see subsection 3.3). In the following subsections, we will detail and explain each step of a typical **Πgasus** *script*. For instance, we will solve the Poisson's equation on the unit square domain with the source term :

$$f(x, y) = 8\pi^2 \sin(2\pi x)\sin(2\pi y)$$

and the analytical solution

$$u(x, y) = \sin(2\pi x)\sin(2\pi y)$$

3.1 Spaces Definition

The Galerkin Finite Element Method relies on the fact that we approach the inifinite dimensional spaces, where the unknowns may live, by a sequence of finite dimensional subspaces. With **Πgasus** we recover this approach and start by creating these finite dimensional subspaces.

```
# define the discrete space
V = space(as_file=ls_domain)
# set boundary conditions
# here we use Homogeneous Dirichlet on whole the boundary
V.dirichlet(faces=[[1,2,3,4]])
V.set_boundary_conditions()
```

The discrete space needs a geometry that may be given by in **XML** or **HDF5** file. Once the space was defined, we have to associate a grid (*quadrature points*).

```
# create a grid (using Gauss–Legendre quadrature points)
V.create_grids(type="legendre", k=lpi_glorder)
```

3.2 Fields Definition

The **Field** object can be used to define either the right hand side term or the unknown.

```
# define the right hand side
func_f = lambda x,y : [2.0 * ( (2*pi)**2 ) * sin ( 2*pi * x ) * sin ( 2*pi * y )]
F_V = field(space=V, func = func_f)

# this is the unknown
func_u = lambda x,y : [sin ( 2*pi * x ) * sin ( 2*pi * y )]
U_V = field(space=V, func = func_u)
```

3.3 Matrices Definition

Here, we show how we can construct and assembly the differential operators involved in our PDE. The Mass matrix operator is of the form Mass

$$\int_\Omega f([u], \mathbf{x})\ \varphi_b\ \varphi_{b'}\ d\Omega$$

while the Stiffness matrix is

$$\int_\Omega (A([u], \mathbf{x})\ \nabla\varphi_b)\ \cdot \nabla\varphi_{b'}\ d\Omega$$

In this example, we have $f([u], \mathbf{x}) = 1.0$ and $A([u], \mathbf{x}) = Id_{\mathbb{R}^3}$ is the identity matrix in $\mathbb{R}^3$. Using **Πgasus** , this can be defined as

```
# define the mass matrix
func_mass = lambda x,y : [1.0]
M_V = matrix(spaces=[V, V], ai_type=MASS, func=func_mass)
# define the stiffness matrix
func_stiff = lambda x,y : [1.0, 0.0, 0.0, 1.0]
S_V = matrix(spaces=[V, V], ai_type=STIFFNESS, func=func_stiff)
```

3.4 Initialization

Once we have defined all the operators and fields that we need to model our PDE, we have to initialize **Πgasus** .

```
fe.initialize()
```

3.5 Assembling Process

Using the function **assembly**, we can assemble the operators that we need. We can imagine, that for a given problem, one of the operators will remain unchanged while the others may change at each iteration. In this case, the user can specify which matrices or fields to assemble, by given it in the **matrices** or **fields** lists.

```
fe.assembly(matrices=[M_V, S_V], fields=[F_V])
```

3.6 Solving the linear system

You can either use the integrated linear solver for **Πgasus** , or export matrices to python and use your favorite solver. Next, we show how the later one can be done using **spsolve** and **SuperLU**.

```
# export matrices to scipy-csr format
Mass_V      = M_V.to_csr()
Stiffness_V = S_V.to_csr()
Sigma = Mass_V + Stiffness_V

# export the right hand side
lpr_rhs = F_V.get()

# solving using spsolve
```

```
from scipy.sparse.linalg import spsolve
lpr_u = spsolve ( Sigma, lpr_rhs )

# solving using splu
from scipy.sparse.linalg import splu
op_Sigma = splu(Sigma.tocsc()) # Factorization
lpr_u = op_Sigma.solve ( lpr_rhs )

# import the right hand side into Pigasus
U_V.set(lpr_u)
```

Remark 3.1 *The user can also manipulate the objects Fields and matrices using the operators *=, += or .dot(...)*

In (figure 8), we plot the numerical solution of Poisson's equation on the unit square.

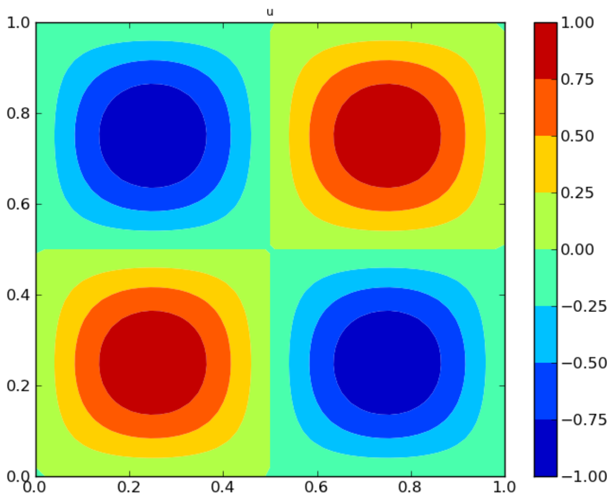

Figure 8: Poisson's equation on the unit square : plot of the numerical solution

3.7 Computing the error norm

In **Πgasus** , the norms are modeled as operators on fields. The user can compute the classical L^2, H^1, *etc*, ..., norms for a given field.
An example of declaration is:

```
N_U = norm ( field=U_V , type=NORM_L2 )
```

Here we show how to compute the L^2 norm.

```
# set the NURBS/B-splines coefficients ,
# in the case of using external solver
U_V.set(lpr_u)
# assemble the norm operator
fe.assembly(norms=[N_U])

lr_norm = N_U.get()
```

As seen, through the previous example, **Πgasus** is designed following the classical FEM language. In the next section, we shall give much more detail about the architecure of **Πgasus** and the offered notions and objects.

4 Dive into Πgasus

In this section, we will introduce some advanced utilities and give much more precision on the design of the library. We start by introducing the *Geometry* module, the architecture and then present the fundamentals concepts/objects.

4.1 Geometry Module

An important difference between CAD and the numerical simulation worlds, is the nature of a given domain. In CAD, a circle is a curve, but in order to solve the Poisson's equation (for example) on a circular domain (for example) we need a 2D description. Hence, even if there are many powerful tools for CAD, they present at least this disadvantage. It is also complicated for mathematicians and physicists to use such tools. Remember that the user wants a volume description of its domain. For more details, we refer to [35, 36, 37, 12]. For all these reasons, we prefered to developp our own module for geometry, with a simple interface for the user and a classical **XML** format. We have also added **HDF5** format to handle heavy data, as the user may want to store his domain after the refinement process.

4.1.1 Formats

There are two ways to store the geometry. For heavy data (especially in 3D), you have to use the **HDF5** format. Weights are stored here as the $d+1$ component of the control point (where d is the space dimension). In (Appendix, *source code 1*) we show the **XML**-file of the linear description of the unit square.

4.1.2 Geometry utilities

In (Appendix, *source code 2*), we show how to use a p-refinement. Now let's take the linear description of the unit square, and perform

1. a p-refinement: we will elevate the spline degree : $+1$ in the ξ-direction and $+1$ in the η-direction,
2. an h-refinement: we will insert $\{0.25, 0.5, 0.75\}$ in the ξ-direction and $\{0.3, 0.7\}$ in the η-direction,.

The detail of the script is given in (Appendix, *source code 3*). In order to do a k-refinement, you can either do a h-refinement then a p-refinement, or do directly a hp-refinement with duplicated knots (Appendix, *source code 4*). In (Appendix, *source code 5*) we show a simple example of how we can duplicate a given geometry and modify it to have a multi-patchs description.

Remark 4.1 *We have developed a 2D-CAD designer tool, in order to manipulate and generate directly, the volume description of the computational domain [43].*

4.2 Πgasus architecture

In the following subsection, we present the architecture behind **Πgasus** .
Although there is no Singleton notion in *Python*, we can define an alternative using the following classical idea.

```python
def singleton(cls):
    instances = {}
    def getinstance():
        if cls not in instances:
            instances[cls] = cls()
        return instances[cls]
    return getinstance
```

Then, we can define the **common_obj** class, which will contain all spaces, fields, matrices, . . . , declarations. Each time the user will declare a new space (for example), it will be automatically added into the **common_obj.spaces** list.

```python
@singleton
class common_obj(object):
    def __init__(self):
    # ...
    # importing the Fortran module
    # ...
        import pyfem as py
        self.pyfem = py.pyfem
    # ...

    # ...
    # defining objects lists
    # ...
        self.fields         = []
        self.matrices       = []
        self.spaces         = []
        self.mappings       = []
        self.grids          = []
        self.norms          = []
    # ...
```

Remark 4.2 *All objects (Fields, Matrices and Norms) are stored with a sub-domain (patch) id. As seen before, the user will use the fundamental geometric operations, to split the computational domain into many sub-domains while keeping the geometry exact. Then he will provide this new description to construct the Finite Element space.*

4.2.1 Spaces

The notion of **spaces** has been introduced in the previous section. Let us just give some additional remarks. The user can define a space using an *exterior* mapping (not defined by the IGA approach). For example, he can give an analytic mapping (or a new metric) or even a CAD-description using splines or nurbs.

Remark 4.3 *We have choosed to use a tensorial approach for* ***Πgasus*** *. This means that inside each element, the grid is viewed as n-arrays rather than an array of $\mathbb{R}^n$. A patch is a collection of elements. The user can stick patchs together by using the duplicate function.*

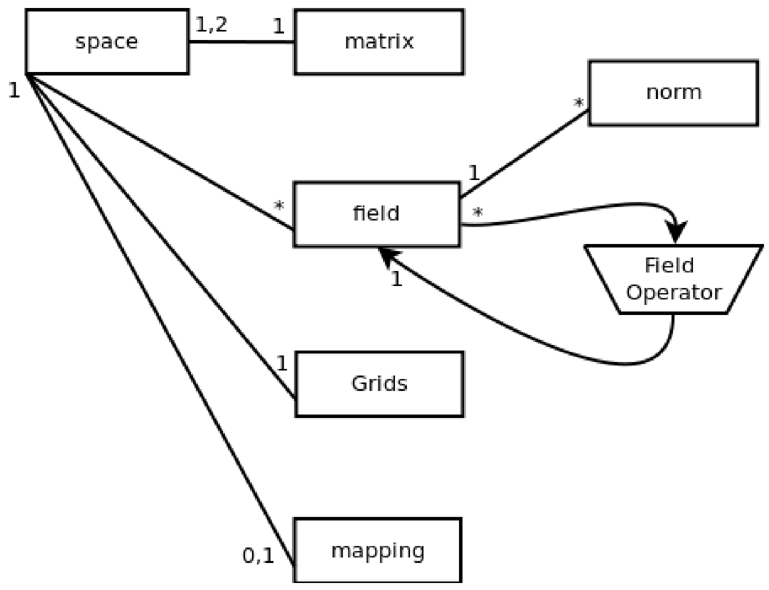

Figure 9: An UML class diagram for **Πgasus**

```
V = space(as_file=ls_domain)
V.duplicate(faces_base= list_faces_duplicated, faces= list_faces_duplicata)
V.set_boundary_conditions()
```

Remark 4.4 *The user can provide how to calculate the basis (and their derivatives) on each element. For example, he can give the values of the basis (and their derivatives) over the* 1*D-quadrature points.*

Remark 4.5 *For (non-homogeneous) boundary conditions, the user can create an adequate space using a grid with a boundary profile.*

The **dirichlet** function is used to set Dirichlet boundary condition on the appropriate domain faces. This is given as a list over patchs (see figure 10).

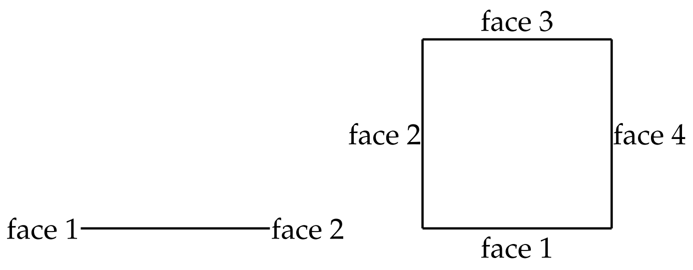

Figure 10: Faces numerotation for boundary conditions. (left) 1D case, (right) 2D case.

```
V.dirichlet(faces=[[2,4]])
```

which will set the boundaries defined by $\{\xi = 0.0, \forall \eta\}$ and $\{\xi = 1.0, \forall \eta\}$, in the 2D case, to Dirichlet condition.

In the case where we use a multi-patch description, the user can specify the Dirichlet boundary condition as shown in figure 11.

```
V.dirichlet(faces=[[1,2,3], [1,3,4]])
```

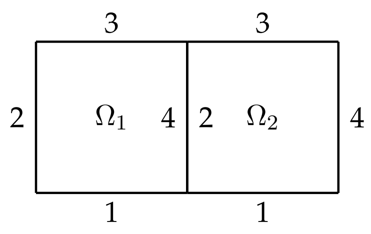

Figure 11: Faces numerotation for boundary conditions for a multi-patchs domain in the 2D case.

The **duplicate** function can be used to set periodic boundary conditions or to stick patchs together with a $\mathcal{C}^0$ condition. It tells **Πgasus** that all basis defined by "face-base" have the same ID (see [31] for details on FEM connectivities structures) as those by "face". Both "face-base" and "face" must be lists of couples [patch_id, num_face]. To set a periodic boundary condition:

```
V.duplicate(faces_base=[[0,1]], faces=[[0,3]])
```

To stick two patchs together:

```
V.duplicate(faces_base=[[0,4]], faces=[[1,2]])
```

4.2.2 Vectorial spaces

The basic definition of spaces, leads to scalar functions. In order to define a vectorial space, the user must proceed as *on paper*, he starts by defining spaces for each coordinate, on the patch and without a mapping, then he defines the whole space as a **vect_space**. In the following example, we show a typical construction, for the Maxwell's time domain problem [45].

In 2D domains, Maxwell's equations can be decoupled into two systems. The first involving the (E_x, E_y, H_z) components is called the Transverse Electric (TE) mode, and the second, involving the (H_x, H_y, E_z) components is called the Transverse Magnetic (TM) mode. As both modes can be discretized in the same manner, The TE mode reads

$$\frac{\partial \mathbf{E}}{\partial t} - \mathbf{rot}\, H = -\mathbf{J}, \tag{4.8}$$

$$\frac{\partial H}{\partial t} + \mathrm{rot}\, \mathbf{E} = 0, \tag{4.9}$$

$$\mathrm{div}\, \mathbf{E} = \rho, \tag{4.10}$$

where the components are defined by $\mathbf{E} = \begin{pmatrix} E_x \\ E_y \end{pmatrix}$, $H = H_z$. Let us define the following scalar spaces :

$$V = \mathcal{S}^{p,p}_{\alpha,\alpha}, \quad W1 = \mathcal{S}^{p,p-1}_{\alpha,\alpha-1}, \quad W2 = \mathcal{S}^{p-1,p}_{\alpha-1,\alpha}$$

and the vectorial space

$$W = W1 \times W2$$

The discrete spaces V and W are involved in the DeRham sequence [45].

```
p = 3
n = 63

ls_etiq = "_p"+str(p)  +"x"+str(p)  +"_n"+str(n)+"x"+str(n)

ls_domain_V  = "domain_p"+str(p)  +"x"+str(p)  +"_n"+str(n)+"x"+str(n)+".xml"
ls_domain_W1 = "domain_p"+str(p)  +"x"+str(p-1)+"_n"+str(n)+"x"+str(n)+".xml"
ls_domain_W2 = "domain_p"+str(p-1)+"x"+str(p)  +"_n"+str(n)+"x"+str(n)+".xml"
ls_domain_W  = "domain_W.xml"

# ***************************************
# Definition of the space V
# ***************************************
V = space(as_file=ls_domain_V)
V.set_boundary_conditions()
V.create_grids(type="legendre", k=lpi_ordregl)
# ***************************************

# ***************************************
# Definition of the space W1
# ***************************************
W1 = space(as_file=ls_domain_W1)
W1.set_boundary_conditions()
W1.create_grids(space=V)
# ***************************************

# ***************************************
# Definition of the space W2
# ***************************************
W2 = space(as_file=ls_domain_W2)
W2.set_boundary_conditions()
W2.create_grids(space=V)
# ***************************************

# ***************************************
# Definition of the space W as W1 x W2
# ***************************************
W = space_vect(spaces=[W1,W2])
W.set_boundary_conditions()
W.create_grids(space=V)
# ***************************************
```

4.2.3 Fields

As said previously, the notion of *Field* is intended to model a right hand side or an unknown. Next we list some of its attributs:

- **space** : This is the discrete space on which the field is defined.
- **func** : This is a given function for the field. It can be used either for the source term or the exact solution.
- **type** : This is the projection/interpolation type.
- **operator** : This is used to apply an operator to a field, defined previously.
- **field** : This is the operande for the operator (must be given if the type is FIELD_OPERATOR).
- **func_arguments** : This is a list of fields used for a non-linear field (*i.e.* $f([g], \mathbf{x})$).

Remark 4.6 *The user can define a field as a function of a list of fields. In the following example, we show how to model a term of the form :*

$$f(u, x, y) = u^2 + x^2 \partial_x u + (x + y)\partial_y u$$

In fact, we will need to define a new field $u2 = \mathbf{v} \cdot \nabla u$ *which is a FIELD_OPERATOR of type GRAD applied on the fields* u *with an argument function* $\mathbf{v} = [x^2, x + y]$.

```
# ...
# Defining the Field Operator
# ...
func_v = lambda x,y : [x^2, x+y]
U2 = field(space=V, operator=GRAD, field=U, func = func_v)
# ...

# ...
# this is the non-linear part
# ...
func_F = lambda list_F,x,y : [list_F[0]^2 + list_F[1]]
F = field(space=V, func = func_F, func_arguments=[U,U2])
# ...
```

4.2.4 Matrices

A matrix ise stored naturally in a compact format thanks to its profile. The latter is easily computed thanks to the spaces connectivities. In order to construct a matrix, you need to specify the following parameters:

- **space** : This is a couple of the discrete spaces on which the Differential Operator is defined.
- **ai_type** : This is the type of the Differential Operator (MASS, STIFFNESS, ADVECTION, ...)
- **func** : This is a given parameter function for the matrix.
- **matrices** : This is used to construct a composed (block) matrix.
- **transpose** : If we want to transpose the matrix during the construction.
- **func_arguments** : This is a list of fields used for a non-linear field (*i.e.* $f([u], \mathbf{x})$).
- **addto** : if we want to add the local element matrix directly to a global matrix previously defined. This will reduce the memory cost.

In table 2, we give the list of the implemented Differential Operators.

Operator	**Contribution**
Mass	$\int_\Omega f([u], \mathbf{x})\ \varphi_b\ \varphi_{b'}\ d\Omega$
Stiffness	$\int_\Omega (A([u], \mathbf{x})\ \nabla\varphi_b)\ \cdot \nabla\varphi_{b'}\ d\Omega$
Advection	$\int_\Omega \mathbf{v}([u], \mathbf{x}) \cdot\ \nabla\varphi_b\ \varphi_{b'}\ d\Omega$
Second derivatives	$\int_\Omega \mathbf{v}([u], \mathbf{x}) \cdot\ D^2\varphi_b\ d\Omega$

Table 2: List of the implemented Differential operators

The user can also create Block-matrices or import a matrix. In the following example we show how to construct a Block-Matrix. The user can choose which part to update during the assembling process.

```
M_V = matrix(matrices=[[A_1,B_1],[C_1,D_1]])
```

and then, he can change the blocks by calling

```
M_V.set(matrices=[[A_2,B_2],[C_2,D_2]])
```

Where each two matrices X_1 and X_2 ($X \in \{A, B, C, D\}$) are defined on the same spaces, and have the same profile. The user can choose which matrices to update, by calling

```
# M is a Block−matrix = [[A,B],[C,D]] defined previously by
# M = matrix(matrices=[[A,B],[C,D]])
# if for example, only A and D must be re−evaluated then the user must call

M.assembly([[True,False],[False,True]])
```

4.2.5 Grids

IIgasus offers different types of grids. The user can associate to the discrete space, a *volume*, a *surface*, an *edge* or a *boundary* grid. The *volume, surface, edge* profiles are related to the elements, while the *boundary* profile is for the whole patch (sub-domain). Here is an example of how to create a grid for a given discrete space.

```
# ****************************************
# Definition of the space V
# ****************************************
V = space(as_file=ls_domain_V)
V.set_boundary_conditions()
V.create_grids(type="legendre", k=lpi_ordregl, profile="boundary")
# ****************************************
```

By grids, we mean, by default, quadrature ones. The user can use the predefined ones, or gives his own grids, as follows:

```
# ****************************************
# Definition of the space V
# ****************************************
V = space(as_file=ls_domain_V)
V.set_boundary_conditions()
V.create_grids(list_nodes=list_x, list_weights=list_w)
# ****************************************
```

The user can also create a grid without associating it to any space. This may be interesting if he wants to take advantage of the B-splines and NURBS tranformations, in order to developp, for example, a Finite Difference code.

```
# ****************************************
# Definition of the grid G
# ****************************************
geo = cg.cad_geometry(ls_domain)
G = grids(profile="volume", api_k=lpi_ordregl, ao_geometry=geo.list[:], as_type="legendre")
```

```
# ****************************************
```

In addition, **ΠIgasus** offers the possibility to create, manipulate a *metric* object and associate it to a grid. Remember that when using a CAD-description to manipulate the metric, we never need to store the whole grid. On each element, we only store the $1D$ array, the grid will be constructed, on the fly, at each assembling step.

```
# ****************************************
# Definition of Metrics
# ****************************************

# ...
# using a CAD-description
# ...
M1 = metric(geometry=geo.list[:])
# ...

# ...
# using an analytical function
# ...
F = lambda r,t : [r * np.cos(2. * np.pi * t), r * np.sin(2. * np.pi * t)]

DF = lambda r, t : [np.cos(2. * np.pi * t) \
, - 2. * np.pi * r * np.sin(2. * np.pi * t) \
, np.sin(2. * np.pi * t) \
, 2. * np.pi * r * np.cos(2. * np.pi * t)]

M2 = metric(analytic=[F,DF])
# ...

# ...
# using a list of points and derivatives
# ...
M3 = metric(points=lpr_points)
# ...

# ****************************************
```

Then we can associate the metric to a grid, as the following.

```
# ****************************************
# Definition of the grid G
# ****************************************
geo = cg.cad_geometry(ls_domain)
G = grids(type="legendre", k=lpi_ordregl, metric=Met)
# ****************************************
```

4.2.6 Diagnostics

The user can apply different operators to a given field. Once for example, you have the Finite Element description, you can ask **ΠIgasus** to evaluate the field or its derivatives. Now let's take the example of Anisotropic Diffusion, which writes :

$$\partial_t u - \nabla \cdot (K \nabla u) = f, \quad \Omega \tag{4.11}$$

$$u = 0, \quad \partial\Omega \tag{4.12}$$

In general, u denotes the temperature (inside the plasma), and K the conductivity. In this example, we consider the evolution of a Gaussian pulse, where:

$$K = \begin{pmatrix} D\mu_{\|} \sin^2(\theta) + \mu_\perp \cos^2(\theta) & D(\mu_{\|} - \mu_\perp) \sin(\theta)\cos(\theta) \\ D(\mu_{\|} - \mu_\perp) \sin(\theta)\cos(\theta) & D\mu_{\|} \cos^2(\theta) + \mu_\perp \sin^2(\theta) \end{pmatrix} \tag{4.13}$$

We would like to visualize the perpendicular diffusion effect.

```
def func_b ( x , y ):
  lr_tetha  = get_tetha ( x , y )
  return [cos ( lr_tetha ) , sin ( lr_tetha )]

U_V.set_func(func_b)
lpr_values = U_V.eval(0, operator=GRAD_S)
```

In figure 12, we plot the numerical solution and the transverse part.

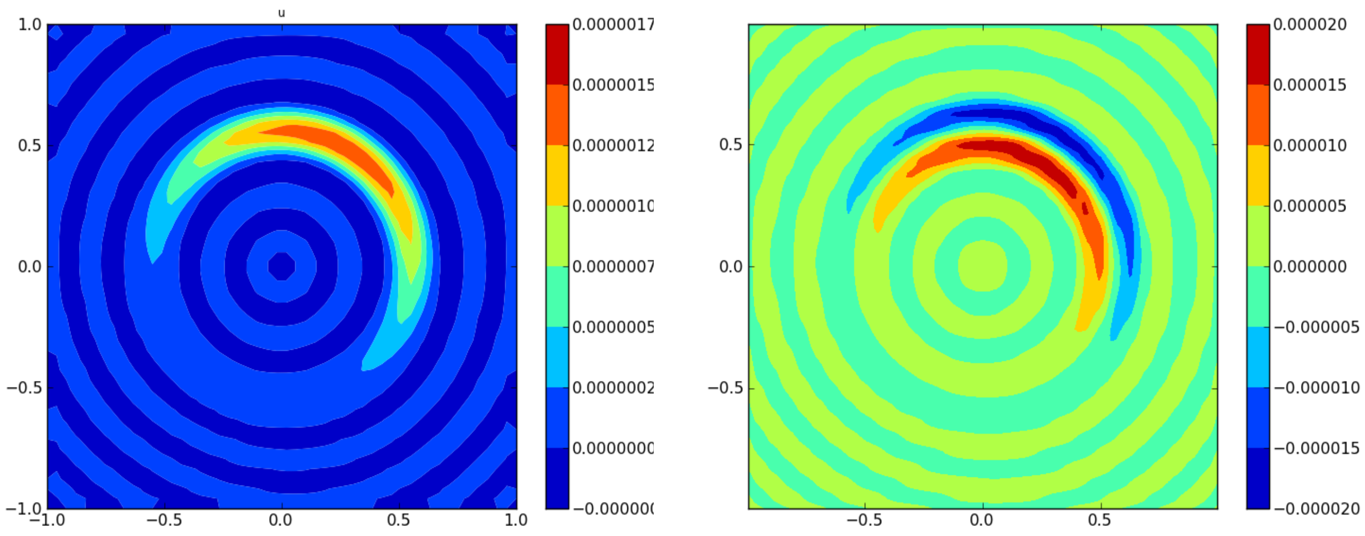

Figure 12: Anisotropic Diffusion on the $[-1,1] \times [-1,1]$ square : plot of the solution (right) and the normal (transverse) part (left)

4.3 Visualization using Pylab

Let us start by recalling a very important result for splines curves.

The following result shows how we can approach the values at some knots using the *B-splines* coefficients. Let us define the average knots as, $\bar{t}_j = \frac{t_{j+1}+\cdots+t_{j+k-1}}{k-1}$, for each $j \in \{1, \cdots, N+1\}$.

Theorem 4.7 *With the above notations, we have, for each $S \in \mathcal{S}_k(T, I) \cap \mathcal{C}^1$,*

$$|S(\bar{t}_j) - [S]^j| \leqslant C(k)h^2 \|D^2 S\|_{[t_{j+1}, t_{j+k-1}]} \tag{4.14}$$

where $h := \max_{i\in\Lambda}(t_{i+1} - t_i)$.
If we define $VS = \sum_{i\in\Lambda} S(\bar{t}_i)N_i$, then we have

$$\|S - VS\| \lesssim h^2, \quad \forall S \in \mathcal{S}_k(T, I) \cap \mathcal{C}^m, \;\; m \geqslant 2 \tag{4.15}$$

As we can notice, the results of this form are important; we do not need to evalute the spline surface for visualization. We will only need to associate at each average knot, the correspondant coefficient. In the next example, we show how to do a *fast-plot* of a given field:

```
# Default call
F_V.fast_plot()
```

The function **fast_plot** has the following arguments:

ai_patch_id the current patch,

useControlPoints is *True* if the *B-slines* coefficients will be associated to the control points as a first approximation,

savedPoints used if **useControlPoints** is *False*. If it is *True*, the user must provide **list_P**.

list_P a list of the associated points to *B-slines* coefficients.

The user still has the possibility to create his own grid and then evaluate the field.

4.4 Parallelization

Πgasus was designed in order to be parallelized, even if the current version is sequential. In the future it will be possible to use generic frameworks as *Murge* [27], *PetSc* [5], *Hips* [23] and have a direct link with some linear solvers (*Pastix [27], Mumps [2, 3]*).

5 Application to a non-linear equation

In this section, we show as an application of IGA, the resolution of a non-linear partial differential equation. We will present two methods: Picard and Newton's algorithms.
In the sequel, we shall consider the following problem:
Find u such that:

$$\begin{cases} -\nabla \cdot (A \nabla u) + Bu = F(\mathbf{x}, u) & , \Omega \\ u = 0 & , \partial\Omega \end{cases} \tag{5.16}$$

Let $\mathcal{V}_h$ be the discrete space, such that $\mathcal{V}_h = \mathbf{span}\{\varphi_b, \ \ b \in \{1, \cdots, n\}\}$, then the variational formulation of (5.16) is :

$$\int_\Omega (A\nabla u) \cdot \nabla \varphi_b + \int_\Omega B u \varphi_b = \int_\Omega F(\mathbf{x}, u)\varphi_b, \quad \forall b \in \{1, \cdots, n\} \tag{5.17}$$

thus, by expanding u_h over $\mathcal{V}_h$, using $u_h = \sum_{b \in \{1, \cdots, n\}} [u]^b \varphi_b$, we get :

$$\sum_{b \in \{1, \cdots, n\}} [u]^b \{ \int_\Omega (A \nabla \varphi_b) \cdot \nabla \varphi_{b'} + \int_\Omega B \varphi_b \varphi_{b'} \} = \int_\Omega F(\mathbf{x}, u_h) \varphi_{b'}, \quad \forall b' \in \{1, \cdots, n\} \tag{5.18}$$

this leads to solve the problem :

$$\mathcal{S}[u] = \mathcal{F}([u]) \tag{5.19}$$

where,

$$\mathcal{S}_{b,b'} = \int_\Omega (A\nabla\varphi_b) \cdot \nabla \varphi_{b'} + \int_\Omega B \varphi_b \varphi_{b'}, \quad \forall b, b' \in \{1, \cdots, n\} \tag{5.20}$$

$$\mathcal{F}([u])_{b'} = \int_\Omega F(\mathbf{x}, u_h) \varphi_{b'}, \quad \forall b' \in \{1, \cdots, n\} \tag{5.21}$$

5.1 Picard's algorithm

To solve iteratively 5.19, let us start with the Picard algorithm, which is the simplest one but also the less accurate.

1. X^0 is given,
2. knowing X^n, we solve :

$$\mathcal{S}X^{n+1} = \mathcal{F}(X^n) \tag{5.22}$$

5.2 Newton's algorithm

Let us define the function :

$$g(X) = \mathcal{S}X - \mathcal{F}(X) \tag{5.23}$$

thus $[u]$ is a zero of the function g. To solve 5.19, we use Newton's method. As $J_{g(X)} = \mathcal{S} - J_{\mathcal{F}(X)}$, the Newton's method is:

- X^0 is given,
- knowing X^n, we solve :

$$J_{g(X^n)}(X^{n+1} - X^n) = -g(X^n) \tag{5.24}$$

The algorithm is the following:

1. we compute the mass matrix associated to the function : $\partial_u\mathcal{F}$, *i.e* :

$$M^n_{b,b'} = \int_\Omega \partial_u F(\mathbf{x}, \sum_{b\in\{1,\cdots,n\}} X^n_b \varphi_b)\varphi_b\varphi_{b'} \tag{5.25}$$

2. compute the term $\mathcal{F}(X^n)$:

$$[\mathcal{F}(X^n)]_{b'} = \int_\Omega F(\mathbf{x}, \sum_{b\in\{1,\cdots,n\}} X^n_b \varphi_b)\varphi_{b'} \tag{5.26}$$

3. compute $g(X^n)$:

$$g(X^n) = \mathcal{S}X^n - \mathcal{F}(X^n) \tag{5.27}$$

4. compute $J_{g(X^n)}$:

$$J_{g(X^n)} = \mathcal{S} - J_{\mathcal{F}(X^n)} = \mathcal{S} - M^n \tag{5.28}$$

5. solve $J_{g(X^n)}(X^{n+1} - X^n) = -g(X^n)$, and then find X^{n+1}

5.3 Numerical results : Example from combustion theory

In this section, we shall solve the equation :

$$-\Delta u = -a\mathrm{e}^{\beta u} \tag{5.29}$$

This example occurs in combustion theory, but also models the electrostatic potential in a charged body.
The general form of solutions is :

$$u(x, y) = \frac{1}{\beta} \ln \frac{8C}{a\beta} - \frac{2}{\beta} \ln |(x + A)^2 + (y + B)^2 - C| \tag{5.30}$$

for more solutions, we refer to [15].
In order to have the function u, vanishing at the boundary, we shall take the following values of parameters:

$$C = -\frac{1}{2}, \quad A = B = 0, \quad a\beta = -4$$

which gives,

$$u(x, y) = -\frac{2}{\beta} \ln |x^2 + y^2 + \frac{1}{2}| \tag{5.31}$$

One can easily check that u verifies:

$$-\Delta u = \frac{4}{\beta}\mathrm{e}^{\beta u} \tag{5.32}$$

In the following test, we took $\beta = -1$.
To have homogeneous Dirichlet boundary condition, the domain will be a circle of radius $\frac{\sqrt{2}}{2}$, centered at 0.

In figure 13, we plot the numerical solution. In figure 14, we plot the evolution of the error with respect to the number of iterations for Picard and Newton algorithms. As expected, we recover the good behavior of the Newton's algorithm.

In the Appendix, we give the detailed codes for Picard's (*source code 6*) and Newton's methods (*source code 7*).

6 Conclusions

In this article, we present a new library **Πgasus** developed to use the new paradigm of the IsoGeometric Analysis. **Πgasus** was designed in order to give a *comfortable* framework for both mathemacians and users. The parallelization of the library is under developpement. The library will also offer local refinements and multiple entry-points for linear solvers. Another active aera is to provide fast solvers and preconditionners, for a specific class of domains and problems, especially in plasmas physics problems and electromagnetism.

For the moment, the library is mainly used at the *C.E.A. Cadarache*, and shortly at the *IPP Garching* and *FOM Institute for Plasma Physics*. People interested by the library, can contact the author. In the future, it will be available for downloads, subjected to a a specific licence.

Acknowledgement : We warmly thank G. Latu, V. Grandgirard, G. Dif-Pradalier (CEA-Cadarache), B. Nkonga (University of Nice) and E. Sonnendrücker (University of Strasbourg and IPP-Garching), for useful discussions and advices. This project was founded by the ANYMOS-ANR.

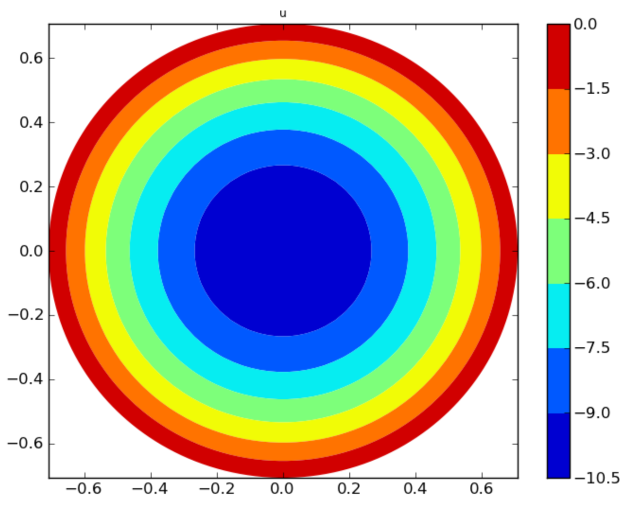

Figure 13: Non-elliptic equation using Picard's algorithm : plot of the numerical solution

Appendix

Geometry module

```
<xml>
<patch>
<param_domain>
<n>2,2</n>
<p>1,1</p>
</param_domain>
<knots>
 0.0, 0.0, 1.0, 1.0
</knots>
<knots>
 0.0, 0.0, 1.0, 1.0
</knots>
<points>
 0.0, 0.0, 1.0;
 1.0, 0.0, 1.0;
 0.0, 1.0, 1.0;
 1.0, 1.0, 1.0
</points>
</patch>
</xml>
```

Source Code 1: XML description of the unit square

```
import cad_geometry as cg

ls_file = "domain.xml"
geo = cg.cad_geometry(ls_file)
geo.refine(patch_id=0,api_p=[2,3])
geo.save("out.xml")
```

Source Code 2: p-refinement example: Starting from the latter description, we use the following script to elevate the spline degree : $+2$ in the ξ-direction and $+3$ in the η-direction.

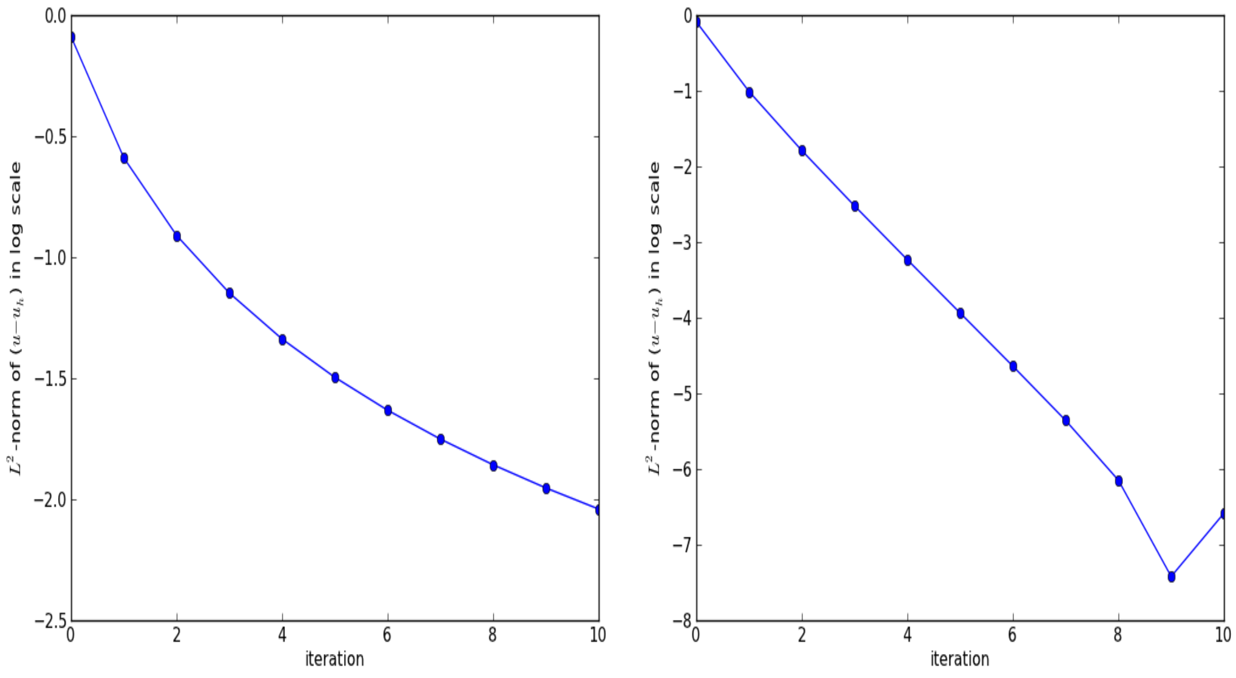

Figure 14: Non-elliptic equation: evolution of the error during the (left) Picard's and (right) Newton's algorithms.

```
import cad_geometry as cg
import numpy as np

ls_file = "domain.xml"
geo = cg.cad_geometry(ls_file)

N = 5
lpr_ksi = np.linspace(0.0, 1.0, N)[1:-1]
lpr_eta = np.array([0.3,0.7])
geo.refine(patch_id=0,api_p=[1,1], apr_t=[lpr_ksi, lpr_eta])

geo.save("out.xml")
```

Source Code 3: hp-refinement example

```
import cad_geometry as cg
import numpy as np

ls_file = "domain.xml"

geo = cg.cad_geometry(ls_file)

lpi_P = [1,1]
lpi_N = [14,14]

# ...
li_d = 0
li_m = int(lpi_N[li_d]/2)
a = np.linspace(0.0,0.5,li_m+2)[1:-1]
b = np.linspace(0.5,1.0,li_m+2)[1:-1]
lpr_x = np.zeros(lpi_N[li_d], dtype=np.double)
lpr_x [:li_m] = a ; lpr_x [li_m:] = b
```

```python

lpr_ksi = lpr_x
# ...

# ...
li_d = 1
li_m = int(lpi_N[li_d]/2)
a = np.linspace(0.0,0.5,li_m+2)[1:-1]
b = np.linspace(0.5,1.0,li_m+2)[1:-1]
lpr_x = np.zeros(lpi_N[li_d], dtype=np.double)
lpr_x [:li_m] = a ; lpr_x [li_m:] = b

lpr_eta = lpr_x
# ...

geo.refine(patch_id=0,api_p=lpi_P, apr_t=[lpr_ksi, lpr_eta])

lpr_ksi = np.array([0.5, 0.5])
lpr_eta = np.array([0.5, 0.5])
geo.refine(patch_id=0, apr_t=[lpr_ksi, lpr_eta])

ls_domain = "domain_p"+str(lpi_P[0]+1)+"x"+str(lpi_P[1]+1)+"_n"+str(lpi_N[0]+1)+"x"+str(lpi_N↩
    [1]+1)+".xml"
geo.save(ls_domain)
```

Source Code 4: hpk-refinement example

```python
import cad_geometry as cg

ls_file = "domain.xml"
geo = cg.cad_geometry(ls_file)

# we duplicate the patch
geo.list *= 2

import numpy as np
N = [63,63]
P = [1,1]

lpr_ksi = np.linspace(0.0, 1.0, N[0])[1:-1]
lpr_eta = np.linspace(0.0, 1.0, N[1])[1:-1]

# refine the 1st patch
geo.refine(patch_id=0,api_p=P, apr_t=[lpr_ksi, lpr_eta])

# refine the 2nd patch
geo.refine(patch_id=1,api_p=P, apr_t=[lpr_ksi, lpr_eta])

# translation through ex = [1,0]
v = [1.0,0.0]
geo.list[1].translate(v)

ls_domain  = "domain_p"+str(P[0]+1)  +"x"+str(P[1]+1)  +"_n"+str(N[0])+"x"+str(N[1])+"_mp.xml"
geo.save(ls_domain)
```

Source Code 5: multi-patchs example

Scripts for the nonlinear examples

```python
#! /usr/bin/python

from test_params import *
from pigasus.fem.constants import *
from pigasus.fem.field import *
from pigasus.fem.norm import *
from pigasus.fem.grids import *
from pigasus.fem.matrix import *
from pigasus.fem.space import *

import pigasus.fem.fem as fem
fe = fem.fem(stdoutput=True,ai_detail=0)

# ****************************************
# Definition of the space V
# ****************************************
V = space(as_file=ls_domain)
V.dirichlet(faces=[[1,2,3,4]])
V.set_boundary_conditions()
V.create_grids(type="legendre", k=lpi_ordregl)
# ****************************************

# ****************************************
# Definition of fields
# ****************************************
from numpy import log, exp

func_u0 = lambda x,y : [- 2.0 * log ( x**2 + y**2 + 0.5 )]
U0_V = field(space=V, func = func_u0)

func_u = lambda x,y : [- 2.0 * log ( x**2 + y**2 + 0.5 )]
U_V = field(space=V, func = func_u)

# ... this is the non-linear part
def func_F (list_F, x, y):
    return [ 4.0 * exp ( list_F[0] )]
F_V = field(space=V, func = func_F, func_arguments=[U_V])
# ****************************************

# ****************************************
# Definition of norms
# ****************************************
N_U = norm(field=U_V, type=NORM_L2)

# ****************************************
# Definition of matrices
# ****************************************
func_mass = lambda x,y : [1.0]
M_V = matrix(spaces=[V, V], ai_type=MASS, func=func_mass)
func_stiff = lambda x,y : [1.0, 0.0, 0.0, 1.0]
S_V = matrix(spaces=[V, V], ai_type=STIFFNESS, func=func_stiff)
# ****************************************

fe.initialize()

fe.assembly(matrices=[M_V, S_V], fields=[U0_V, U_V, F_V])

Mass_V      = M_V.to_csr()
Stiffness_V = S_V.to_csr()

from scipy.sparse.linalg import spsolve

# initialization
#lpr_u = spsolve ( Mass_V, U0_V.get() )
lpr_u = np.zeros(U_V.size)
U_V.set(lpr_u); np.savetxt("runs/u_0.txt", lpr_u )

list_norm = []

fe.assembly(norms=[N_U])
list_norm.append(N_U.get())

# ****************************************
```

```python
i = 0
miniter  = i
minerror = error
list_minerror = []
lpr_u_old = lpr_u
# ****************************************
while ( ( error > tol ) and ( i < niter ) ):

    print("iteration = "+str(i))

    i = i + 1

    F_V.reset()

    fe.assembly(fields=[F_V])
    lpr_source = F_V.get()
#        print "F(u) = ", lpr_source
    lpr_u = spsolve ( Stiffness_V, lpr_source )

    #computing the error between X^(n+1) and X^n : X^(n+1) - X^n
    lpr_delta = lpr_u - lpr_u_old
    lpr_u_old = lpr_u
    U_V.set(lpr_u)

    error = max (abs(lpr_delta))
    if ( minerror > error ) :
            minerror = error
            miniter  = i
    list_minerror.append(minerror)

    if ( np.mod ( i , nfreq ) == 0 ):
            numdiag = i / nfreq

            np.savetxt("runs/u_"+str(numdiag)+".txt", lpr_u )

        fe.assembly(norms=[N_U])
        list_norm.append(N_U.get())

np.savetxt('runs/l2norm'+ls_etiq+'.txt', np.array([list_norm]))
np.savetxt("runs/minerror.txt", np.array(list_minerror))
print("Solving done using Picard's method")
print("after "+str(i)+" iterations, with a tolerance of "+str(tol))
#print "list_minerror = ", list_minerror
#print "list_norm = ", list_norm
```

Source Code 6: Non-elliptic equation using Picard's algorithm

```python
#! /usr/bin/python

from test_params import *
from scipy.io import mmread, mmwrite
from pigasus.fem.constants import *
from pigasus.fem.field import *
from pigasus.fem.norm import *
from pigasus.fem.grids import *
from pigasus.fem.matrix import *
from pigasus.fem.space import *

import pigasus.fem.fem as fem
fe = fem.fem(stdoutput=True,ai_detail=0)

# ****************************************
# Definition of the space V
# ****************************************
V = space(as_file=ls_domain)
V.dirichlet(faces=[[1,2,3,4]])
V.set_boundary_conditions()
```

```
V.create_grids(type="legendre", k=lpi_ordregl)
# ****************************************

# ****************************************
# Definition of fields
# ****************************************
from numpy import log, exp

func_u0 = lambda x,y : [- 2.0 * log ( x**2 + y**2 + 0.5 )]
U0_V = field(space=V, func = func_u0)

func_u = lambda x,y : [- 2.0 * log ( x**2 + y**2 + 0.5 )]
U_V = field(space=V, func = func_u)

# ... this is the non-linear part
def func_F (list_F, x, y):
    return [ 4.0 * exp ( list_F[0] )]
F_V = field(space=V, func = func_F, func_arguments=[U_V])

def func_dF (list_F, x, y):
    return [ 4.0 * exp ( list_F[0] )]
dF_V = field(space=V, func = func_dF, func_arguments=[U_V])
# ****************************************

# ****************************************
# Definition of norms
# ****************************************
N_U = norm(field=U_V, type=NORM_L2)
# ****************************************

# ****************************************
# Definition of matrices
# ****************************************
func_mass = lambda x,y : [1.0]
Ma_V = matrix(spaces=[V, V], ai_type=MASS, func=func_mass)

Mn_V = matrix(spaces=[V, V], ai_type=MASS, func=func_dF, func_arguments=[U_V])

func_stiff = lambda x,y : [1.0, 0.0, 0.0, 1.0]
S_V = matrix(spaces=[V, V], ai_type=STIFFNESS, func=func_stiff)
# ****************************************

fe.initialize()

fe.assembly(matrices=[Ma_V,S_V], fields=[U0_V, U_V])

Mass_V = Ma_V.to_csr()
Stiffness_V = S_V.to_csr()

from scipy.sparse.linalg import spsolve

# initialization
#lpr_u = spsolve ( Mass_V, U0_V.get() )
lpr_u = np.zeros(U_V.size)
#print "size : ", U_V.size
U_V.set(lpr_u); np.savetxt("runs/u_0.txt", lpr_u )

list_norm = []

fe.assembly(norms=[N_U])
list_norm.append(N_U.get())

# ****************************************
i = 0
miniter  = i
minerror = error
list_minerror = []
# ****************************************
while ( ( error > tol ) and ( i < niter ) ):

    print("iteration = "+str(i))

    i = i + 1

    F_V.reset()
    dF_V.reset()
    fe.assembly(fields=[F_V, dF_V])
    fe.assembly(matrices=[Mn_V])

    # get the assemblied terms
    M_n = Mn_V.to_csr()
    lpr_source = F_V.get()

    #compute g(X^n)
    lpr_rhs = - Stiffness_V.dot ( lpr_u ) + lpr_source

    #compute J_n
    J_n = Stiffness_V - M_n
    lpr_delta = spsolve ( J_n, lpr_rhs )

    #compute X^(n+1) = X^n + delta
    lpr_u = lpr_u + lpr_delta

    U_V.set(lpr_u)

    error = max (abs(lpr_delta))
    if ( minerror > error ) :
            minerror = error
            miniter  = i
    list_minerror.append(minerror)

    if ( np.mod ( i , nfreq ) == 0 ):
            numdiag = i / nfreq
            np.savetxt("runs/u_"+str(numdiag)+".txt", lpr_u )

        fe.assembly(norms=[N_U])
        list_norm.append(N_U.get())

np.savetxt('runs/l2norm'+ls_etiq+'.txt', np.array([list_norm]))
np.savetxt("runs/minerror.txt", np.array(list_minerror))
print("Solving done using Newton's method")
print("after "+str(i)+" iterations , with a tolerance of "+str(tol))
print "final error is ", str(minerror)
```

Source Code 7: Non-elliptic equation using Newton's algorithm

References

[1] Abiteboul, J., Latu, G., Grandgirard, V., Ratnani, A., Sonnendrücker, E., and Strugarek, A. Solving the vlasov equation in complex geometries. *ESAIM: Proc.*, 32:103–117, 2011.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[4] Back, A., Crestetto, A., Ratnani, A., and Sonnendrücker, E. An axisymmetric pic code based on isogeometric analysis. *ESAIM: Proc.*, 32:118–133, 2011.

[5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry 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.

[6] Y. Bazilevs, M.-C. Hsu, and M.A. Scott. Isogeometric fluidøstructure interaction analysis with emphasis on non-matching discretizations, and with application to wind turbines. *Computer Methods in Applied Mechanics and Engineering*, (0):–, 2012.

[7] Dmitry Berdinsky, Min jae Oh, Tae wan Kim, and Bernard Mourrain. On the problem of instability in the dimension of a spline space over a t-mesh. *Computers Graphics*, 36(5):507 – 513, 2012. Shape Modeling International (SMI) Conference 2012.

[8] A. Buffa, D. Cho, and M. Kumar. Characterization of t-splines with reduced continuity order on t-meshes. *Computer Methods in Applied Mechanics and Engineering*, 201-204(0):112 – 126, 2012.

[9] A. Buffa, C. de Falco, and G. Sangalli. Isogeometric analysis: Stable elements for the 2d stokes equation. *International Journal for Numerical Methods in Fluids*, 65(11-12):1407–1422, 2011.

[10] A. Buffa, J. Rivas, G. Sangalli, and R. Vázquez. Isogeometric discrete differential forms in three dimensions. *SIAM J. Numerical Analysis*, 49(2):818–844, 2011.

[11] A. Buffa, G. Sangalli, and R. V·zquez. Isogeometric analysis in electromagnetics: B-splines approximation. *Computer Methods in Applied Mechanics and Engineering*, 199(17ø20):1143 – 1152, 2010.

[12] E. Cohen, T. Martin, R.M. Kirby, T. Lyche, and R.F. Riesenfeld. Analysis-aware modeling: Understanding quality considerations in modeling for isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, 199(5-8):334 – 356, 2010. Computational Geometry and Analysis.

[13] J.A Cottrell, T. Hughes, and Y. Bazilevs. *Isogeometric Analysis, toward Integration of CAD and FEA*. John Wiley & Sons, Ltd, first edition, 2009.

[14] N. Crouseilles, A. Ratnani, and E. Sonnendrücker. An isogeometric analysis approach for the study of the gyrokinetic quasi-neutrality equation. *Journal of Computational Physics*, 231:373–393, 2012.

[15] Polyanin A. D. and Zaitsev V. F. *Handbook of Nonlinear Partial Differential Equations*. Chapman, Hall CRC, 2004. BocaRaton.

[16] L. Beirào da Veiga, A. Buffa, D. Cho, and G. Sangalli. Isogeometric analysis using t-splines on two-patch geometries. *Computer Methods in Applied Mechanics and Engineering*, 200(21-22):1787 – 1803, 2011.

[17] L. Beirào da Veiga, A. Buffa, D. Cho, and G. Sangalli. Analysis-suitable t-splines are dual-compatible. *Computer Methods in Applied Mechanics and Engineering*, (0):–, 2012.

[18] L. Beirao daVeiga, A. Buffa, J. Rivas, and G. Sangalli. Some estimates for h-p-k refinement in isogeometric analysis. *Numerische Matematik*, 118:271 – 305, 2011.

[19] C. de Falco, A. Reali, and R. V·zquez. Geopdes: A research tool for isogeometric analysis of pdes. *Advances in Engineering Software*, 42(12):1020 – 1034, 2011.

[20] C. DeBoor. *A practical guide to splines*. Springer-Verlag, New York, applied mathematical sciences 27 edition, 2001.

[21] R.A. DeVore and G.G. Lorentz. *Constructive Approximation*. Springer-Verlag, Berlin, Heidelberg, 1993.

[22] Michael R. Dörfel, Bert Jüttler, and Bernd Simeon. Adaptive isogeometric analysis by local h-refinement with t-splines. *Computer Methods in Applied Mechanics and Engineering*, 199(5-8):264 – 275, 2010. Computational Geometry and Analysis.

[23] Jérémie Gaidamour and Pascal Hénon. HIPS : a parallel hybrid direct/iterative solver based on a Schur complement approach. In *PMAA 08*, Neuchâtel, Suisse, 2008-06.

[24] Carlotta Giannelli, Bert Juttler, and Hendrik Speleers. Thb-splines: The truncated basis for hierarchical splines. *Computer Aided Geometric Design*, 29(7):485 – 498, 2012. Geometric Modeling and Processing 2012.

[25] Hector Gomez, Thomas J.R. Hughes, Xes˙s Nogueira, and Victor M. Calo. Isogeometric analysis of the isothermal navierøstokesøkorteweg equations. *Computer Methods in Applied Mechanics and Engineering*, 199(25ø28):1828 – 1840, 2010.

[26] Ch. Heinrich, B. Simeon, and St. Boschert. A finite volume method on nurbs geometries and its application in isogeometric fluidøstructure interaction. *Mathematics and Computers in Simulation*, 82(9):1645 – 1666, 2012.

[27] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.

[28] C. PrudhH́omme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, and G. Pena. Feel++: A computational framework for galerkin methods and advanced numerical methods. 2012. Submitted.

[29] Ming-Chen Hsu and Yuri Bazilevs. Blood vessel tissue prestress modeling for vascular fluidøstructure interaction simulation. *Finite Elements in Analysis and Design*, 47(6):593 – 599, 2011. The Twenty-Second Annual Robert J. Melosh Competition.

[30] Qi-Xing Huang, Shi-Min Hu, and Ralph R. Martin. Fast degree elevation and knot insertion for b-spline curves. *Computer Aided Geometric Design*, 22(2):183 – 197, 2005.

[31] T. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Publications Inc., 2003.

[32] T.J.R. 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(39-41):4135 – 4195, 2005.

[33] Alexander Konyukhov and Karl Schweizerhof. Geometrically exact theory for contact interactions of 1d manifolds. algorithmic implementation with various finite element models. *Computer Methods in Applied Mechanics and Engineering*, 205ø208(0):130 – 138, 2012. Special Issue on Advances in Computational Methods in Contact Mechanics dedicated to the memory of Professor J.A.C. Martins.

[34] W. Tiller L. Piegl. *The NURBS Book*. Springer-Verlag, Berlin, Heidelberg, 1995. second ed.

[35] T. Martin, E. Cohen, and R.M. Kirby. Volumetric parameterization and trivariate b-spline fitting using harmonic functions. *Computer Aided Geometric Design*, 26(6):648 – 664, 2009. Solid and Physical Modeling 2008, ACM Symposium on Solid and Physical Modeling and Applications.

[36] Tobias Martin and Elaine Cohen. Volumetric parameterization of complex objects by respecting multiple materials. *Computers Graphics*, 34(3):187 – 197, 2010. Shape Modelling International (SMI) Conference 2010.

[37] Tobias Martin, Elaine Cohen, and Robert M. Kirby. Mixed-element volume completion from nurbs surfaces. *Computers Graphics*, 36(5):548 – 554, 2012. Shape Modeling International (SMI) Conference 2012.

[38] Goldman R. N. and Lyche T. *Knot Insertion and Deletion Algorithms for B-Spline Curves and Surfaces*. SIAM, Philadelphia, USA, 1993.

[39] N. Nguyen-Thanh, H. Nguyen-Xuan, S.P.A. Bordas, and T. Rabczuk. Isogeometric analysis using polynomial splines over hierarchical t-meshes for two-dimensional elastic solids. *Computer Methods in Applied Mechanics and Engineering*, 200(21-22):1892 – 1908, 2011.

[40] L.A. Piegl and W. Tiller. *The NURBS book*. Springer Verlag, 1997.

[41] O. Pironneau, F. Hecht, and A. Le Hyaric. Freefem++. http://www.freefem.org/ff++/ftp/freefem++doc.pdf.

[42] Hartmut Prautzsch and Bruce Piper. A fast algorithm to raise the degree of spline curves. *Comput. Aided Geom. Des.*, 8:253–265, October 1991.

[43] A. Ratnani. Caid : A cad tool for isogeometric computational domains. INRIA report: In preparation.

[44] A. Ratnani. Isogeometric analysis in plasma physics and electromagnetism. 2011. Phd thesis, INRIA, Université de Strasbourg. URL : http://tel.archives-ouvertes.fr/tel-00628060/en/.

[45] A. Ratnani and E. Sonnendrücker. An arbitrary high-order spline finite element solver for the time domain maxwell equations. *Journal of Scientific Computing*, pages 1–20, 2011. 10.1007/s10915-011-9500-8.

[46] A. Ratnani and E. Sonnendrücker. Isogeometric analysis in reduced magnetohydrodynamics. *Computational Science & Discovery*, 5(1):014007, 2012.

[47] Daniel Rypl and Boek Patz·k. Object oriented implementation of the t-spline based isogeometric analysis. *Advances in Engineering Software*, 50(0):137 – 149, 2012. CIVIL-COMP.

[48] Dominik Schillinger, Luca Dedè, Michael A. Scott, John A. Evans, Michael J. Borden, Ernst Rank, and Thomas J.R. Hughes. An isogeometric design-through-analysis methodology based on adaptive hierarchical refinement of nurbs, immersed boundary methods, and t-spline cad surfaces. *Computer Methods in Applied Mechanics and Engineering*, 2012.

[49] Dominik Schillinger and Ernst Rank. An unfitted hp-adaptive finite element method based on hierarchical b-splines for interface problems of complex geometry. *Computer Methods in Applied Mechanics and Engineering*, 200(47-48):3358 – 3380, 2011.

[50] L. L. Schumaker. *Spline Functions: Basic Theory*. Wiley (New York), 1981.

[51] M.A. Scott, X. Li, T.W. Sederberg, and T.J.R. Hughes. Local refinement of analysis-suitable t-splines. *Computer Methods in Applied Mechanics and Engineering*, 213-216(0):206 – 222, 2012.

[52] T.W. Sederberg, D.L. Cardon, J. Zheng, and T. Lyche. T-spline simplification and local refinement. *ACM Trans, Graphics*, 23:276–283, 2004.

[53] S. Shojaee, E. Izadpanah, N. Valizadeh, and J. Kiendl. Free vibration analysis of thin plates by using a nurbs-based isogeometric approach. *Finite Elements in Analysis and Design*, 61(0):23 – 34, 2012.

[54] Dokken T. Workshop on: "Non-Standard Numerical Methods for PDE's", Pavia, Italy, jun 29 - jul 02.

[55] Li Tian, Falai Chen, and Qiang Du. Adaptive finite element methods for elliptic equations over hierarchical t-meshes. *Journal of Computational and Applied Mathematics*, 236(5):878 – 891, 2011. The 7th International Conference on Scientific Computing and Applications, June 13-16, 2010, Dalian, China.

[56] A.-V. Vuong, C. Giannelli, B. Juttler, 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.

[57] A.-V. Vuong, Ch. Heinrich, and B. Simeon. Isogat: A 2d tutorial matlab code for isogeometric analysis. *Computer Aided Geometric Design*, 27(8):644 – 655, 2010. Advances in Applied Geometry.

[58] Ping Wang, Jinlan Xu, Jiansong Deng, and Falai Chen. Adaptive isogeometric analysis using rational pht-splines. *Computer-Aided Design*, 43(11):1438 – 1448, 2011. Solid and Physical Modeling 2011.

[59] Gang Xu, Bernard Mourrain, Régis Duvigneau, and André Galligo. Optimal analysis-aware parameterization of computational domain in 3d isogeometric analysis. *Computer-Aided Design*, 2011.

[60] Gang Xu, Bernard Mourrain, Régis Duvigneau, and André Galligo. Parameterization of computational domain in isogeometric analysis: Methods and comparison. *Computer Methods in Applied Mechanics and Engineering*, 200(23-24):2021 – 2031, 2011.