A Versatile Strategy for the Implementation of Adaptive Splines

Andrea Bressan[1] and Dominik Mokriš[2]

[1] Department of Mathematics, University of Oslo, Norway
andbres@math.uio.no
[2] Institute of Applied Geometry, Johannes Kepler University Linz, Austria
dominik.mokris@jku.at

Abstract. This paper presents an implementation framework for spline spaces over T-meshes (and their d-dimensional analogs). The aim is to share code between the implementations of several spline spaces. This is achieved by reducing evaluation to a generalized Bézier extraction.
The approach was tested by implementing hierarchical B-splines, truncated hierarchical B-splines, decoupled hierarchical B-splines (a novel variation presented here), truncated B-splines for partially nested refinement and hierarchical LR-splines.

Keywords: Implementation, Bézier Extraction, THB-splines, LR-splines.

1 Introduction

A common method to represent shapes in Computer-Aided Design (CAD), Computer-Aided Engineering (CAE) and Computer-Aided Manufacturing (CAM) is to parametrize the desired geometry (or its boundary) with Non-Uniform Rational B-Splines (NURBS). B-splines have a global tensor-product structure, where each d-variate basis function is a product of d univariate basis functions. This means that changes in spatial resolution cannot be confined to a small region; they necessarily spread to a union of stripes of the domain (Fig. 1).

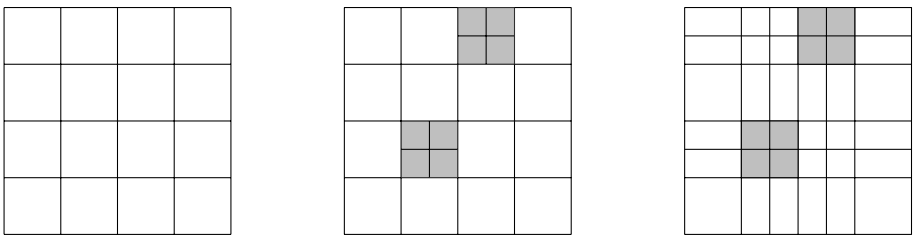


Fig. 1: Limitation of the tensor-product construction. Left: the coarse grid; Middle: the desired refinement; Right: the coarsest tensor-product grid refined on the grey area.

Different constructions that allow for local refinement were proposed during the last two decades and gained support with the introduction of IsoGeometric Analysis (IGA) [24]. Indeed, IGA pushed the use of splines in numerical simulation where local refinement is a prerequisite of adaptive methods. The following list includes the best known constructions:

- Hierarchical B-splines (HB) [16]. This is a multiscale approach: each scale is associated to a different tensor-product B-spline space. Functions from each scale are *selected* depending on the locally required resolution and together they form the hierarchical B-spline basis. There are many variations of HB, among them: the Truncated Hierarchical B-splines (THB) [17], the Truncated Decoupled Hierarchical B-splines (TDHB) [32], the Truncated B-splines for partially nested refinement (TBPN) [43] and Decoupled Hierarchical B-splines (DHB) introduced here for the first time.
- T-splines (T) [39,38]. The central notion is the T-mesh: a planar graph with lengths. A B-spline corresponds to each vertex of the graph and its knot vectors depend on the length of the neighboring edges. These B-splines generate the space. Unfortunately, they can be linearly dependent. *Analysis Suitable T-splines* (AST) avoid linear dependencies by restricting the class of allowed T-meshes [31,11]. AST spaces can be constructed in 2D [35] and also defined for 3D domains [34].
- Locally Refined splines (LR) [14]. Their definition is given in terms of *minimally supported* B-splines contained in a space of piecewise polynomials. The generators are not always linearly independent. A bivariate construction that avoids linear dependencies is the hierarchical LR-splines (HLR) [5].

Several other spaces and alternative bases exist, e.g., [36,13,8,27,6]. On the one hand, the mentioned spaces contain piecewise polynomials over box-shaped subdomains and allow for smooth functions. On the other hand, each construction was defined for a specific application and, as a consequence, described and analyzed with its own set of tools. Thus it is difficult to make a comparison involving more than a few spaces and having criteria that are not application-specific. A comparison of HB, THB and LR based on the conditioning of the mass matrix is presented in [25]. A similar approach was used in [23].

Our aim is to describe a software framework allowing to implement various spline spaces in a systematic way. The main criterion is the versatility of the code, that is, the possibility to include further spline spaces to this framework with relative ease. In this way we hope to facilitate both the comparison of different spline spaces and experimenting with alternative definitions. The proposed method is a generalization of Bézier extraction [2,37,15], which is a well-established tool in IGA. As a proof of concept, three spline spaces available in the literature and a newly designed space were implemented. The choice of the spaces has been basen on authors' personal research interests and contains only spaces with multilevel structure. Less structured spaces such as LR-splines or T-splines could be implemented as well, but they would probably require more effort due to their intrinsic complexity, particularly so when non-dyadic refinement and knot lines with multiplicities would be considered.

The framework is presented in Section 2 without any reference to specific spline spaces. Section 3 discusses the space and time complexity of the proposed approach and presents possible optimizations. Section 4 highlights the differences from Bézier extraction, while Section 5 describes how the framework can be applied to HB, THB, DHB, TBPN and HLR splines. These spaces were implemented and their implementations are used in Section 6 to show how the different spaces behave in a few selected cases.

The following notation conventions are used throughout the paper.

style	example	used for
lowercase Latin letters	a, b, ...	real numbers
bold lowercase Latin letters	$\mathbf{a}$, $\mathbf{b}$, ...	vectors of real numbers
lowercase Greek letters	α, β, ...	functions
bold lowercase Greek letters	$\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, ...	vectors of functions
uppercase Greek letters	Ω, Δ, ...	subsets of $\mathbb{R}^d$ or $\mathbb{R}^{d-1}$
uppercase Latin letters	A, B, ...	sets
bold uppercase Latin letters	$\mathbf{A}$, $\mathbf{B}$, ...	matrices and operators
calligraphic uppercase Latin letters	$\mathcal{A}$, $\mathcal{B}$, ...	function spaces

2 Implementation Method

The aim of an implementation is to evaluate the generators of a spline space at a set X of points contained in the domain $\Omega \subseteq \mathbb{R}^d$. This is sufficient for the application to interpolation problems and for the implementation of Galerkin methods based on numerical quadrature.

The spline spaces of interest are generated by piecewise polynomials on a partition of Ω into axis-aligned boxes called *elements*. Thus their restriction to an element can be expressed in terms of tensor-product Bernstein polynomials. By doing so it is possible to repurpose Finite Element Method (FEM) codebases to IGA. This approach was proposed for NURBS in [2] under the name *Bézier extraction* and later extended to other spaces [37,15].

The main idea of this paper is to replace the elements with more general subdomains and the Bernstein basis with an arbitrary local basis, possibly a different one for each subdomain. This allows the implementations to be closer to the mathematical definitions of the spline spaces, which are typically described in terms of B-splines and not of Bernstein polynomials. A detailed comparison with Bézier extraction is provided in Section 4.

2.1 Description

Let $G = \{\gamma_1, \dots, \gamma_n\}$ be the generating set of a spline space and assume that there exists a partition $T = \{\Delta_1, \dots, \Delta_s\}$ of the domain Ω and a corresponding set of local bases[3] $B = \{B_1, \dots, B_s\}$ such that the restriction of each $\gamma \in G$ to any Δ_i admits a representation in span B_i. More precisely,

$$\forall \gamma \in G,\ \forall_{i=1}^{s},\ \forall \mathbf{x} \in \Delta_i : \quad \gamma(\mathbf{x}) = \sum_{\beta \in B_i} m_{\beta,\gamma} \beta(\mathbf{x}) \ , \tag{1}$$

[3] (or, more generally, generating sets)

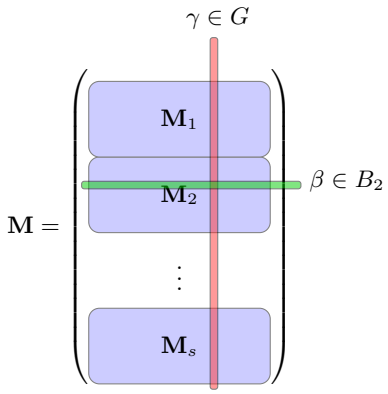


Fig. 2: Structure of the representation matrix.

and thus

$$\boldsymbol{\gamma}(\mathbf{x}) = \boldsymbol{\beta}_i(\mathbf{x})\mathbf{M}_i \ , \tag{2}$$

where $\boldsymbol{\gamma}(\mathbf{x})$ and $\boldsymbol{\beta}_i(\mathbf{x})$ are the row vectors

$$\begin{aligned} \boldsymbol{\gamma}(\mathbf{x}) &= \big(\gamma_1(\mathbf{x}), \dots, \gamma_n(\mathbf{x})\big) = \big(\gamma(\mathbf{x})\big)_{\gamma \in G} \ , \\ \boldsymbol{\beta}_i(\mathbf{x}) &= \big(\beta(\mathbf{x})\big)_{\beta \in B_i} \end{aligned} \tag{3}$$

and $\mathbf{M}_i = \big(m_{\beta,\gamma}\big)_{\beta \in B_i,\, \gamma \in G}$ is the matrix containing the coefficients from (1). The matrices $\mathbf{M}_i$ can be collected as blocks of the matrix $\mathbf{M}$ with $\sum_{i=1}^{s} \#B_i$ rows and $\#G$ columns as depicted in Fig. 2.

The generating set G is uniquely determined by T, B and $\mathbf{M}$ through (2). This suggests an implementation in which T, B and $\mathbf{M}$ are provided by the space-specific code and the evaluation of $\boldsymbol{\gamma}(\mathbf{x})$ is performed using (2). Note that different choices of T, B and $\mathbf{M}$ can describe the same G and thus there is a certain freedom to optimize for different scenarios (see Section 5).

For such implementation T requires a method `findSubdomain` that given a point $\mathbf{x} \in \Omega$ returns the index i of the corresponding subdomain $\Delta_i \ni \mathbf{x}$. The implementation of the local bases B_i should contain a method `eval` that returns a matrix containing the values of the basis functions $\beta \in B_i$ in a given set of points $X \subset \Omega$. For consistency the same interface should be implemented by the resulting spline space.

Before describing a suggested implementation of the three components T, B and $\mathbf{M}$ it is worth describing the `eval` interface in more detail. For the expected applications it is necessary to compute both function values and function derivatives at a set of points $X \subset \Omega$. As shown in (2) the values (and also the derivatives) can be transformed by a matrix multiplication. This suggest a format that allows the transformation of all the data with a single operation.

Let $W = \{\mathbf{v}_1, \ldots, \mathbf{v}_w\}$ be the list of the multiindices corresponding to the desired derivatives. For instance, in two dimensions, the value corresponds to $(0,0)$, the first partial derivative with respect to the first direction to $(1,0)$ and the second mixed derivative to $(1,1)$ and $W = \{(0,0), (1,0), (1,1)\}$ means that all these three are computed. Then for a set of functions $F = \{\varphi_1, \ldots, \varphi_f\}$ the base format can be

$$\mathbf{E}_F(\mathbf{x}, W) = \begin{pmatrix} \partial^{\mathbf{v}_1}\varphi_1(\mathbf{x}) & \cdots & \partial^{\mathbf{v}_1}\varphi_f(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \partial^{\mathbf{v}_w}\varphi_1(\mathbf{x}) & \cdots & \partial^{\mathbf{v}_w}\varphi_f(\mathbf{x}) \end{pmatrix} .$$

The values at multiple points can be stored by collecting similar blocks. In particular, for $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_r\}$ let

$$\mathbf{B}_i(X, W) = \begin{pmatrix} \mathbf{E}_{B_i}(\mathbf{x}_1, W) \\ \vdots \\ \mathbf{E}_{B_i}(\mathbf{x}_r, W) \end{pmatrix} \text{ and } \mathbf{G}(X, W) = \begin{pmatrix} \mathbf{E}_G(\mathbf{x}_1, W) \\ \vdots \\ \mathbf{E}_G(\mathbf{x}_r, W) \end{pmatrix} .$$

Assuming the above format, a general implementation of `eval` for G is given by the following procedure.

Procedure: `eval(`X, W`)`
Input: the set of points $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_r\} \subset \Omega$
Input: the requested data $W = \{\mathbf{v}_1, \ldots, \mathbf{v}_w\}$
Output: $\mathbf{G}(X, W)$
foreach $\mathbf{x} \in X$ **do**
  $i =$ `findSubdomain(`$\mathbf{x}$`)`
  $\mathbf{E}_{B_i}(\mathbf{x}, W) = B_i$`.eval(`$\mathbf{x}, W$`)`
  $\mathbf{E}_G(\mathbf{x}, W) = \mathbf{E}_{B_i}(\mathbf{x}, W)\mathbf{M}_i$
  `/*` $\mathbf{E}_G(\mathbf{x}, W)$ `is directly written into` $\mathbf{G}(X, W)$ `*/`
end

For the common case when X is contained in one Bézier element of the space it can be useful to provide the following optimized procedure that accepts the containing subdomain as an input parameter.

Procedure: `evalSubdomain(`X, W, i`)`
Input: the set of points $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_r\} \subset \Omega$
Input: the requested data $W = \{\mathbf{v}_1, \ldots, \mathbf{v}_w\}$
Input: the index i of the subdomains containing X
Output: $\mathbf{G}(X, W)$
$\mathbf{B}_i(X, W) = B_i$`.eval(`X, W`)`
$\mathbf{G}(X, W) = \mathbf{B}_i(X, W)\mathbf{M}_i$

Now T, B and $\mathbf{M}$ will be described in more detail.

A suitable implementation of T is a binary decision tree (more precisely a binary space partition, cf. [40,41]). For the spaces of interest it is possible to assume that the subdomains Δ_i are polytopes with axis-aligned faces. Each fork

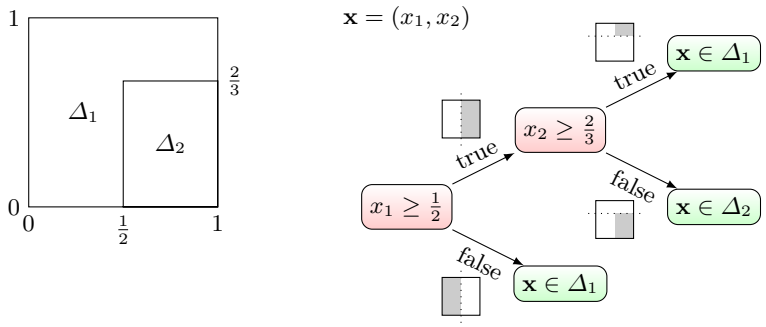


Fig. 3: A partition of Ω in Δ_1 and Δ_2 and a decision tree describing it. The darkened area next to each branch highlights the region corresponding to the branch.

in the tree corresponds to a spatial split along an axis-aligned affine hyperspace, i.e., to a comparison for a specific coordinate. Each branch corresponds to taking one of the corresponding half-spaces. Each leaf of the tree corresponds to the intersection of the taken half-spaces and Ω. Thus T can be represented by a tree storing in each leaf the index of the subdomain containing the corresponding box. Fig. 3 depicts a partition and a corresponding tree.

Binary space partitions not only provide an efficient implementation of the method `findSubdomain` but also offer useful representations of piecewise constant maps $\Omega \to \mathbb{N}$. They enable efficient computation of binary operations (see the references above for union and intersection) that can be employed both for the geometrical computation required by the construction of the different spaces and by refinement strategies.

For instance, given two trees that assign to each point a refinement level, it is easy to compute the coarsest common refinement by the pointwise-max operation. Similarly, the finest common submesh can be computed with a pointwise-min operation.

The collection of local bases $B = \{B_1, \dots, B_s\}$ is simply a list of polymorphic objects implementing the `eval` interface. This allows for arbitrary local bases and thus, for example, Bernstein polynomials as in Bézier extraction, B-splines as in all the implementations presented here, or enriched spaces such as generalized B-splines [3] with piecewise trigonometric or exponential functions.

Finally, $\mathbf{M}$ is a sparse matrix. However, the initialization of the matrix for a particular spline space usually requires most of the space-specific code.

2.2 Subspaces and Functions

Consider a subspace $\operatorname{span} G' \subset \operatorname{span} G$ generated by $G' = \{\gamma'_1, \dots, \gamma'_k\}$. If $\mathbf{N} = (n_{\gamma,\gamma'})_{\gamma \in G, \gamma' \in G'}$ is a matrix that contains in the i-th column the expansion of γ'_i in $\gamma_1, \dots, \gamma_n$, i.e.,

$$\forall \mathbf{x} \in \Omega,\ \boldsymbol{\gamma}'(\mathbf{x}) = \boldsymbol{\gamma}(\mathbf{x})\mathbf{N}\ ,$$

then G' can be implemented by $T, B, \mathbf{M}'$ with

$$\mathbf{M}' = \mathbf{M}\mathbf{N} \ .$$

As a consequence, `eval` is not only a suitable implementation of G, but also of a single function $\varphi \in \operatorname{span} G$. Indeed, this corresponds to $\mathbf{M}'$ having a single column and $\mathbf{N}$ being the column vector of the coefficients of φ. Another application of this method is enforcing homogeneous linear constraints on the space, such as boundary condition or smoothness constraints.

2.3 Multipatch Domains

The proposed framework can be extended to allow for multipatch domains. Multipatch domains are used to describe geometries Ω with nontrivial topology and for which no regular parametrization $G : \widehat{\Omega} \to \Omega$ with a box $\widehat{\Omega} \subset \mathbb{R}^d$ exists. A simple example is the unit sphere in 3D for which there exists no regular parametrization defined on a rectangle. However, it is possible to partition such a domain Ω into mutually disjoint (except at their boundaries) *patches* $\Omega_1, \dots, \Omega_w$, each with its own regular parametrization $G_p : \widehat{\Omega}_p \subset \mathbb{R}^d \to \Omega_p$. Then Ω can be thought of as the image of $G : \widehat{\Omega} \to \Omega$, where $\widehat{\Omega}$ is the disjoint union of $\widehat{\Omega}_1, \dots, \widehat{\Omega}_w$ and the points with the same image have been identified, i.e.,

$$\widehat{\Omega} = \coprod_{p=1}^{w} \widehat{\Omega}_p \,/_{\sim}$$

for a proper $\sim$. The proposed method can be extended to describe functions defined on $\widehat{\Omega}$ by simply changing `findSubdomain` to take the different patches into account. This can be achieved by an optional parameter p. In particular, `evalSubdomain` does not need modifications as long as Δ_i is contained in $\widehat{\Omega}_j$ if B_i is defined on $\widehat{\Omega}_j$.

The construction of $\mathcal{C}^k$ function spaces on multipatch domains is an active research topic in IGA [28,7,10]. This corresponds, by the isoparametric approach, to the construction of subspaces of patchwise $\mathcal{C}^k$ functions on $\widehat{\Omega}$. The relations that define the subspace depend on G and its derivatives and do not necessarily correspond to smoothness conditions on $\widehat{\Omega}$ after simple point identification.

The space of patchwise $\mathcal{C}^k$ functions can be described in the proposed framework by a block-diagonal matrix $\mathbf{M}$, where each block represents the space of $\mathcal{C}^k$ functions on each patch. As described in the previous subsection, the representation of a subspace is obtained by multiplying $\mathbf{M}$ by an appropriate $\mathbf{N}$. This strategy was used in [7], where, due to a different implementation of the patch spaces, the multiplication by $\mathbf{N}$ is done at a post-processing stage and thus incurs in an additional cost.

3 Complexity

Delegating the evaluation to a local basis and computing the linear combination incurs in an additional computational cost. Moreover, storing the coefficients of $\mathbf{M}$ can require a substantial amount of memory.

3.1 Space Complexity

The tested implementation uses a row-compressed format: only the nonzero coefficients are stored in lexicographic order of their indices. The column position of the nonzero entries is stored in a second vector. The row position is deduced by storing a pointer to the first nonzero of each row. This means that the total required memory is proportional to the sum of the number of rows plus the number of nonzero entries of $\mathbf{M}$.

The number of rows of $\mathbf{M}$ equals $\sum_{i=1}^{s} \#B_i$ and thus there is a memory cost associated to functions of the local bases even if they are not used in any Δ_i to represent G.

The number of nonzero coefficients in $\mathbf{M}$ depends on the complexity of the mesh and on the shape of the generators. The number of nonzero coefficients in the column corresponding to $\gamma \in G$ is

$$\sum_{i:\gamma|_{\Delta_i} \neq 0} \#\{\beta : m_{\beta,\gamma} \neq 0\} \ . \tag{4}$$

Thus it is minimized if γ is supported in a single Δ_i and if $\gamma = \beta$ for some $\beta \in B_i$. In contrast, the generators γ whose supports intersect many subdomains or whose shape requires many coefficients to be represented in a subdomain require more memory.

3.2 Time Complexity

The time cost of the `eval` procedure is proportional to the cardinality of X and depends on the cost of the local basis evaluation, which is unknown. Denoting $\mathfrak{C}(matrix)$ the cost of the computation of $\mathbf{B}_i(\mathbf{x}, W)\mathbf{M}_i$, it can be written as

$$\mathfrak{C}(\texttt{eval}) = \#X\Big(\mathfrak{C}(B_i.\texttt{eval}) + \mathfrak{C}(matrix) + \mathfrak{C}(\texttt{findSubdomain})\Big) \ .$$

Remembering that $w = \#W$ is the number of rows in the blocks $\mathbf{E}_{\square,W}$ it is possible to describe each term in more detail.

The complexity of $B_i.\texttt{eval}$ depends on the specific local basis used and is clearly bounded from below by the output size $w\#B_i$. For $d \geq 2$ tensor-product B-splines can be implemented in such a way that the cost is quasi-optimal, i.e., proportional to the output size with a factor that does not depend on their degree but which depends on the dimension:

$$\mathfrak{C}(B_i.\texttt{eval}) \cong dw\#B_i \ .$$

The cost of the matrix-matrix product $\mathbf{B}_i(\mathbf{x})\mathbf{M}_i$ using standard algorithms is proportional to the product of the three dimensions of the two matrices:

$$\mathfrak{C}(matrix) \cong w\#B_i\#G \ .$$

The cost of `findSubdomain` depends on the tree structure and on the complexity of the mesh. For a balanced tree this would be proportional to $\log_2 \ell$,

where ℓ is the number of leaves in T. However, a balanced tree is not necessarily optimal, as the tree should take the usage pattern into account. For instance, if we assume a uniform sampling of the domain, then the optimal tree will have leaves of depth inversely proportional to the measure of the corresponding region. Already when avoiding unnecessary splits (without any balancing), the cost of `findSubdomain` was negligible in the profiling tests.

The total evaluation cost is thus of the following magnitude

$$\mathfrak{C}(\texttt{eval}) \cong \#X\mathfrak{C}(matrix) \cong w\#X\#B_i\#G \ . \tag{5}$$

The same result is obtained for `evalSubdomain` with the difference that $w\#X$ then means the number of rows in $\mathbf{B}_i(X)$.

Comparing this with the output size $w\#X\#G$ shows that the method is rather expensive if $\#B_i$ is big. The next sections show how this cost can be reduced.

3.3 Local Basis and Compression

If the functions in G and B_i have small supports, then the number of nonzero columns in $\mathbf{B}_i(X, W)$ and in $\mathbf{G}(X, W)$ is small compared to $\#B_i$ and $\#G$, respectively. This suggests the use of a compressed format for $\mathbf{B}_i(X, W)$ and $\mathbf{G}(X, W)$, where only the nonzero values and their positions are stored. This is standard in FEM as well as in other numerical methods and it is used in our implementation too.

Assume that $X \subset \Delta_i$. Let L be the set of functions in B_i such that the corresponding columns in $\mathbf{B}_i(X, W)$ are not zero and let A be the corresponding set of functions in G defined by

$$A = \{\gamma \in G \,:\, \exists \beta \in L \,:\, m_{\beta,\gamma} \neq 0\} \ .$$

A function φ is called *active* on X if $\varphi \in L$ or $\varphi \in A$.

Let $\mathbf{L}(X, W)$ and $\mathbf{A}(X, W)$ be the corresponding submatrices of $\mathbf{B}_i(X, W)$ and $\mathbf{G}(X, W)$,

$$\mathbf{L}(X, W) = \begin{pmatrix} \mathbf{E}_L(\mathbf{x}_1, W) \\ \vdots \\ \mathbf{E}_L(\mathbf{x}_r, W) \end{pmatrix} \ , \qquad \mathbf{A}(X, W) = \begin{pmatrix} \mathbf{E}_A(\mathbf{x}_1, W) \\ \vdots \\ \mathbf{E}_A(\mathbf{x}_r, W) \end{pmatrix} \ .$$

Then $\mathbf{B}_i(X, W)$ can be implemented with the pair $(L, \mathbf{L}(X, W))$ where the set L is implemented as a list of indices. This reduces the lower bound on $\mathfrak{C}(B_i.\texttt{eval})$ to

$$\mathfrak{C}(B_i.\texttt{eval}) \cong w\#X\#L \ .$$

Similarly, $\mathbf{G}(X, W)$ can be implemented by the pair $(A, \mathbf{A}(X, W))$. The coefficients in $\mathbf{A}$ are computed by

$$\mathbf{A}(X, W) = \mathbf{L}(X, W)\mathbf{M}_{L,A} \ ,$$

where $\mathbf{M}_{L,A}$ is the submatrix of $\mathbf{M}$ containing the $m_{\beta,\gamma}$, $\beta \in L$ and $\gamma \in A$. This reduces the cost of the linear combination to

$$\mathfrak{C}(matrix) \cong w\#X\#L\#A \ ,$$

improving (5) by the factor

$$\frac{\#L\#A}{\#B_i\#G} \ . \tag{6}$$

The described compression can be applied both to `eval` and `evalSubdomain`. In `eval` it is applied to the evaluation at single points. Then either the list containing the (per point) compressed matrices is returned or all of the matrices are merged into one matrix. The first approach is faster and simpler, the second returns a standard matrix.

The application of compression to `evalSubdomain` is straightforward but it is important to limit X to points contained in a small region so that (6) is minimized.

When using compression, the cost of evaluation is proportional to the output size $w\#X\#A$ multiplied by $\#L$. For polynomial splines and with X contained in a single element, $\#L$ depends only on the polynomial degree. As a consequence the cost of the evaluation per unit of output data does not depend on the mesh (h-refinement) but depends on the degree (p-refinement).

3.4 Tensor Factorization

The tensor-product structure allows to reduce d-variate computations to computations on univariate objects. In our case it allows to replace the computation of the linear combination of d-variate functions with d linear combinations of univariate functions. This is advantageous because the cost of the matrix-matrix product is roughly proportional to the product of the three involved dimensions. In the optimal case this optimization reduces one of the dimensions to its d-th root.

The above strategy can be applied under the weaker assumption that each $\gamma \in G$ and each $\beta \in B_i$ can be factored into products of univariate functions:

$$\gamma(\mathbf{x}) = \prod_{c=1}^{d} \gamma^{(c)}(x_c), \qquad \beta(\mathbf{x}) = \prod_{c=1}^{d} \beta^{(c)}(x_c) \ , \tag{7}$$

where $\square^{(c)}$ means the *factor* of $\square$ corresponding to the c-th coordinate. Of the implemented spaces only HB and HLR satisfy (7), thus this optimization was not implemented and the following is only a theoretical analysis.

In analogy with (7), the same notation is used to denote the factors corresponding to the coordinate directions of tensors $\square = \bigotimes_{c=1}^{d} \square^{(c)}$ and of Cartesian grids of points $\square = \bigtimes_{c=1}^{d} \square^{(c)}$. This should not be confused with the *components*

of vectors and tensors that are denoted by subscripts as in $\boldsymbol{x} = (x_1, x_2)$. In particular, the following factors are defined:

$$\begin{aligned} B_i^{(c)} &= \{\beta^{(c)} : \beta \in B_i\} \ ; \\ G^{(c)} &= \{\gamma^{(c)} : \gamma \in G\} \ ; \\ X^{(c)} &= \{x_c : \mathbf{x} = (x_1, \ldots, x_d) \in X\} \ ; \\ W^{(c)} &= \{v_c : \mathbf{v} = (v_1, \ldots, v_d) \in W\} \ . \end{aligned}$$

In the following $w^{(c)}$ denotes $\#W^{(c)}$, i.e., the number of derivatives of $\gamma^{(c)}$ that are required to compute all requested partial derivatives in W.

Necessarily $G^{(c)} \subseteq \operatorname{span} B_i^{(c)}$, which means that there exists a matrix $\mathbf{M}_i^{(c)}$ such that

$$\forall \mathbf{x} \in \Delta_i : \quad \boldsymbol{\gamma}^{(c)}(x_c) = \boldsymbol{\beta}_i^{(c)}(x_c)\mathbf{M}_i^{(c)} \ .$$

Here, analogously to (3), $\boldsymbol{\gamma}^{(c)}(x_c)$ and $\boldsymbol{\beta}_i^{(c)}(x_c)$ denote the vectors having components indexed by $G^{(c)}$ and $B_i^{(c)}$, respectively, i.e.,

$$\begin{aligned} \boldsymbol{\gamma}^{(c)}(x_c) &= \left(\gamma^{(c)}(x_c)\right)_{\gamma^{(c)} \in G^{(c)}} \ , \\ \boldsymbol{\beta}_i^{(c)}(x_c) &= \left(\beta^{(c)}(x_c)\right)_{\beta^{(c)} \in B_i^{(c)}} \ . \end{aligned}$$

Let S be the set of the multiindices that define G as a subset of $\bigotimes_{c=1}^{d} G^{(c)}$:

$$G = \left\{ \prod_{c=1}^{d} \gamma_{\mathbf{s}_c}^{(c)} : (\mathbf{s}_1, \ldots, \mathbf{s}_d) \in S,\, \gamma_{\mathbf{s}_c}^{(c)} \in G^{(c)} \right\} \subseteq \bigotimes_{c=1}^{d} G^{(c)} \ .$$

For simplicity it is assumed that $B_i = \bigotimes_{c=1}^{d} B_i^{(c)}$ and $X = \bigtimes_{c=1}^{d} X^{(c)}$, but a proper subset (similarly as for G) can be considered at the expense of a more involved notation and implementation.

The tensor-product structure propagates to the set of active functions. Here L contains the multiindices of the functions of B_i corresponding to nonzero columns of $\mathbf{B}_i$. Similarly, A contains the subset of the multiindices in S, i.e., the multiindices of functions in G that correspond to nonzero columns in $\mathbf{G}$. Analogously to the other symbols, $L^{(c)}$ and $A^{(c)}$ denote the collection of the entries relative to the c-th coordinate in L and A respectively.

The procedure `compose` assembles the matrix $\square$ out of the matrices of its factors $\square^{(c)}$ and a list of the necessary products P as in the description of S above.

Procedure: `compose`$(\square^{(1)}, \ldots, \square^{(d)}, X, W, P)$
Input: the tensor components $\square^{(c)}$
Input: the list of points X
Input: the list of derivatives W
Input: the list of required products P
Output: $\square$
foreach $\mathbf{p} = (p_1, \ldots, p_d) \in P$ **do**
  foreach $\mathbf{x} = (x_1, \ldots, x_d) \in X$ **do**
    `/* write row block of the derivatives of` $\square_{\mathbf{p}}$ `at` $\mathbf{x}$ `*/`
    foreach $\mathbf{v} = (v_1, \ldots, v_d) \in W$ **do**
      $\partial^{\mathbf{v}} \square_{\mathbf{p}}(\mathbf{x}) = \prod_{c=1}^{d} \partial^{v_c} \square^{(c)}_{\mathbf{p}_c}(x_c)$
      `/* each value is written in` $\square$`: the row corresponds to the pair` $(\mathbf{x}, \mathbf{v})$`, the column to` $\mathbf{p}$ `*/`
    end
  end
end

If P is omitted, it is assumed that $\square = \bigotimes_{c=1}^{d} \square^{(c)}$ and thus that P contains all the Cartesian multiindices.

For a given domain dimension d the cost of the procedure `compose` is proportional to the size of its output with factor d,

$$\mathfrak{C}(\texttt{compose}) = dw\#P\#X \ .$$

Only an application of this optimization to `evalSubdomain` is presented, but it can also be applied to `eval`. By using `compose`, the evaluation of the local basis can be split in two steps: evaluation of the components of the local basis and their composition. The original `evalSubdomain` can be equivalently rewritten as:

Procedure: `evalSubdomain(`X`,`W`,`i`)`
Input: the set of points $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_r\} \subset \Omega$
Input: the requested data $W = \{\mathbf{v}_1, \ldots, \mathbf{v}_w\}$
Input: the index i of the subdomains containing X
Output: $\mathbf{G}(X, W)$
for $c = 1, \ldots, d$ **do**
  $\mathbf{B}^{(c)} = B_i^{(c)}$`.eval(`$X^{(c)}, W^{(c)}$`)` `/* local evaluation */`
end
$\mathbf{B}_i(X) =$ `compose(`$\mathbf{B}^{(1)}, \ldots, \mathbf{B}^{(d)}, X, W$`)` `/* composition */`
$\mathbf{G}(X, W) = \mathbf{B}_i(X, W)\mathbf{M}_i$ `/* linear combination */`

Table 1: Comparison of the cost for the standard and optimized evaluation for spaces with tensor-product structure.

	standard	optimized
linear combination	$w\#X\#A\#L$	$\sum_{c=1}^{d} w^{(c)}\#X^{(c)}\#A^{(c)}\#L^{(c)}$
composition	$dw\#X\#L$	$dw\#X\#A$

As described, the computational cost of the above is determined by the computation of the matrix-matrix product. This can be reduced by leaving `compose` as the last operation and computing d products of smaller matrices as follows:

Procedure: `evalSubdomain(`X,W,i`)`
Input: the set of points $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_r\} \subset \Omega$
Input: the requested data $W = \{\mathbf{v}_1, \ldots, \mathbf{v}_w\}$
Input: the index i of the subdomains containing X
Output: $\mathbf{G}(X, W)$
for $c = 1, \ldots, d$ **do**
  $\mathbf{B}^{(c)} = B_i^{(c)}$`.eval(`$X^{(c)}, W^{(c)}$`)` `/* local evaluation */`
  $\mathbf{G}^{(c)} = \mathbf{B}^{(c)}\mathbf{M}_i^{(c)}$ `/* linear combination */`
end
$A = S \cap \bigtimes_{c=1}^{d} A^{(c)}$ `/* actives */`
$\mathbf{G}(X, W) =$ `compose(`$\mathbf{G}^{(1)}, \ldots, \mathbf{G}^{(d)}, X, W, A$`)` `/* composition */`

The cost estimates of each step for the two different versions are reported in Table 1.

If there exists m such that for $c = 1, \ldots, d$

$$\#L^{(c)}\#A^{(c)} \leq m\#A \ , \tag{8}$$

then the evaluation cost of the optimized version is proportional to the output size $w\#X\#A$. This result holds independently of the mesh size (h-refinement) and the polynomial degree (p-refinement). Note however that the output size depends on the number of active functions $\#A$ and on the size of the requested data, thus on the polynomial degree, on the set of points X and on w.

The assumption in (8) holds in situations that are of interest for the applications. In particular, it holds for splines of degree p and for points X contained in a single element if $\#A^{(c)} \leq m(p+1)^{d-1}$ for some m independent of the degree. This is the case for m-admissible HB meshes [9], where $\#A^{(c)} \leq m(p+1)$, and for the HLR basis described in [5], for which $\#A^{(c)} \leq 2(p+1)$.

4 Comparison with Bézier extraction

Bézier extraction was proposed for NURBS [2] and extended to T-splines [37] and THB [22]. It is an implementation technique aimed at reusing standard FEM codebases in IGA by representing the basis functions as linear combinations of Bernstein polynomials on each element. The *extraction operator*, i.e., the linear transformation of the basis, is stored in a combination of a per element matrix and a global index of the per element active functions called IEN.

The proposed framework reduces to Bézier extraction when the following choices are made:

T is the partition of the domain into elements;
B_i is the Bernstein basis remapped to the element Δ_i;
$\mathbf{M}_i$ contains the expansion of γ_j in the Bernstein basis in column j.

Thus all the spaces that can be implemented with Bézier extraction – such as T-splines or LR-splines – can be also incorporated to the proposed framework.

Even if the underlying concepts are the same, the implementation differs as the two approaches are optimized for different scenarios. The main differences are collected in Table 2 and their consequences are described below.

Table 2: Qualitative comparison of the two frameworks.

	Bézier extraction	proposed framework
mesh description	list	tree
expansion in local basis	per element	global matrix
local basis	Bernstein polynomials	any

4.1 Mesh Description

Bézier extraction is based on per element data structures. This reflects the original aim: IGA with per element quadrature integration. For such application it is both simple and efficient. The drawback is that it does not provide a feature-rich description of regions that can be used in the implementation of different spaces. Every space must implement its own strategy both for identifying the element containing a point and for the description of the mesh used in its construction.

The tree-based description of the subdomains in the proposed framework provides an efficient tool to describe "arbitrary" regions contained in the domain, to compute intersections, unions and for testing whether a region contains a point (see `findSubdomain` in Section 2). This common code is shared by all the implemented spaces, thus decreasing the per-space code requirements.

Iteration on the elements in this framework is done by nested iteration: the outer iteration is on the tree leaves and the inner on the elements provided by the local basis and contained in the region corresponding to the current leaf. The nested loop is a part of the shared code.

4.2 Expansion in Local Basis

Both approaches store the expansion of the generators with respect to a local basis. In Bézier extraction the linear operator is represented by submatrices (the *extraction operators*) together with their indices (the IEN), whereas in the proposed framework it is represented by a sparse matrix.

From this point of view Bézier extraction can be seen as a specialized matrix format. However, avoiding the specialized format makes the implementation of subspaces and multipatch straightforward, for they correspond to matrix multiplication as has been described in Subsections 2.2 and 2.3 and the code is already provided and optimized by the linear algebra library. It is true that products involving submatrices of a sparse matrix are less efficient then products involving full matrices, but by allowing different generators the total memory requirement can be lowered as it is discussed in the next subsection.

4.3 Local Basis

While Bézier extraction was only described for spaces of piecewise tensor-product polynomials with Bernstein polynomials as local generators, there is no practical issue to extend it to other local bases such as enriched splines spaces as those from [3]. This means that both techniques are roughly equally applicable. Nevertheless, there are two advantages of the proposed framework compared to Bézier extraction.

The first is that most of the spaces of interest are defined in terms of collections of B-splines functions and not Bernstein polynomials. Thus the proposed code stays closer to the definition of the space and it is easier to write.

The second is that by using B-splines as local generators the coefficients are shared between many elements and thus the memory requirement is decreased. Applying (4) shows that the amount of coefficients stored per a generator $\gamma \in G$ in Bézier extraction equals

$$\mathbf{e}_\gamma \dim \mathbb{T} \ ,$$

where $\mathbf{e}_\gamma$ is the number of elements contained in $\operatorname{support} \gamma$ and $\mathbb{T}$ is the space of tensor-product polynomials. This is an upper bound for the amount of coefficients stored using B-splines as local generators. Increasing the number of coefficients does not only increase the memory requirements, but it also increases the cost of space initialization.

In favor of Bézier extraction stands the fact that in IGA applications it is not necessary to evaluate the Bernstein polynomials on the quadrature nodes: it is enough to scale the derivatives computed on a reference element according to the current element.

Summarizing, the proposed framework permits testing of various definitions with a reduced development time. Bézier extraction optimizes the matrix assembling in IGA applications for a specific space.

5 Implemented spaces

The proposed strategy was tested by implementing HB, THB, TBPN, DHB and HLR splines spaces in the *G+Smo* object oriented library [26]. The spaces were coded as templated C++03 classes. They all derive from a common base class that is the realization of the described approach.

The interested reader can compare with other implementations that are either available or described in the literature. (T)HB are implemented in the *G+Smo* open-source library [19]. The code, as of 2014, is described in [29]. Another implementation of (T)HB tailored for IGA research is described in [42]. The source code of bivariate LR-splines is available as a part of the goTools library [20], but no technical description is available.

The first subsection describes the shared code. The following subsections specialize to various spaces. The last subsection reports the size of the implementation measured in lines of code.

5.1 Shared code

The shared code contains the implementation of the ideas described in Section 2 as well as common utilities such as input-output, tensor-product B-splines and debugging functions.

Part of the required code was already present in the *G+Smo* library, in particular, vectors, matrices, sparse matrices (all of them based on the Eigen library [21]) and tensor-product B-splines. Some parts were coded anew such as a specialized version of Boehm's knot-insertion algorithm, the binary tree, functions for transforming between flat-indices and multiindices of multivariate B-splines and others to export data in the *ParaView* format [1].

The implementation of T uses the binary tree as described in Section 2 and includes an interface for performing arbitrary unary and binary operations, possibly by restricting the operation to a box contained in the domain. This mechanism is used in the construction of the spaces: for instance, the Kraft selection mechanism of (T)HB corresponds to finding the minimum of the indices of the subdomains intersecting the support of a function, the decoupling procedure requires methods to compute intersections and unions of polytopes with axis-aligned faces. The implementation of T automatically removes unnecessary branches at construction by collapsing equal subtrees.

A component of *G+Smo* that was developed for smooth multipatch spaces [7] was reused for $\mathbf{M}$. At its core it is a sparse matrix with additional methods for computing A from L and extracting $\mathbf{M}_{L,A}$, see Subsection 3.3. It also contains conversion functions to and from other data types related to the implementation of multipatch geometries and boundary conditions in *G+Smo*.

The base class of all the implemented spaces contains the reference to T, $\mathbf{M}$ and B, the evaluation procedure, constructors allowing for multipatch domains and utilities to obtain functions and subspaces as described in Subsection 2.2.

5.2 (Truncated) Hierarchical B-Splines

The hierarchical basis is defined from a sequence $P_1, \ldots, P_s$ of tensor-product B-spline bases and a corresponding sequence $\Omega = \Omega_1 \supseteq \cdots \supseteq \Omega_s = \emptyset$ of closed domains. For simplicity it is assumed here that Ω is a box in $\mathbb{R}^d$ and that the bases have clamped knots on its boundary. It is required that the tensor-product spaces form a *hierarchy*, i.e.:

$$i < j \Rightarrow \operatorname{span} P_i \subset \operatorname{span} P_j \ . \tag{9}$$

The *hierarchical basis* (HB-splines) is defined by Kraft's *selection* criteria [30]:

$$H = \bigcup_{i=1}^{s} \{\psi \in P_i \ : \ \operatorname{support} \psi \subseteq \Omega_i \text{ and } \operatorname{support} \psi \cap (\Omega_i \setminus \Omega_{i+1}) \neq \emptyset\} \ . \tag{10}$$

The *truncated hierarchical basis* (THB-splines) described in [17] is defined by recursive truncation

$$H' = \{\mathbf{T}_s \cdots \mathbf{T}_{i+1} \psi \ : \ \psi \in H \cap P_i\} \ .$$

The truncation operator $\mathbf{T}_i : \operatorname{span} P_i \to \operatorname{span} P_i$ is defined by

$$\mathbf{T}_i(\varphi) = \sum_{\psi \in P_i \,:\, \psi|_{\Omega \setminus \Omega_i} \neq 0} c_{\varphi,\psi} \psi \ ,$$

where the coefficients $c_{\varphi,\psi}$ are taken from the expansion of φ in P_i:

$$\varphi = \sum_{\psi \in P_i} c_{\varphi,\psi} \psi \ .$$

The matrix representation of $\mathbf{T}_i$ with respect to the basis P_i is thus diagonal with entries

$$t_{\psi,\psi} = \begin{cases} 1 & \text{if } \psi|_{\Omega \setminus \Omega_i} \neq 0 \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

The truncation procedure improves the locality of the resulting basis, guarantees that H' forms a convex partition of unity and preserves the same coefficients as P_i for polynomial expansion [18]. The drawback is that it breaks the tensor-product structure, i.e., the functions $\psi' \in H'$ are not tensor-product B-splines. Thus the optimization described in Section 3.4 cannot be applied for H'.

Note that the composition of the truncation operators differs from the truncation by the finest level: in general if $\psi \in P_i$ then for any $k \geq i$

$$\mathbf{T}_k \cdots \mathbf{T}_{i+1} \psi|_{\Omega_k} \neq \mathbf{T}_k \psi|_{\Omega_k} \ . \tag{11}$$

The equality in (11) holds if the mesh is sufficiently graded.

Two implementations are described. Both assume that the bases $P_1, \ldots, P_s$ have the same degree (i.e., only h-refinement is allowed) and that the subdomains Ω_i are unions of elements of $\operatorname{span} P_i$. The first implementation is closer to the definition and has actually been coded. The second is described in order to show that memory requirements can be lowered with more complex code.

Implementation 1 The simplest implementation defines T by:

$$\Delta_i = \Omega_i \setminus \Omega_{i+1}, \quad i = 1, \dots, s \ ,$$

and B by $B_i = P_i, i = 1, \dots, s$.

Most of the construction of $\mathbf{M}$ is common for HB and THB. In the coded implementation truncation is controlled by a construction option in order to decrease code duplication. The procedure `constructTHB` describes the constructor.

Procedure: `constructTHB`$(\Omega_1, \dots, \Omega_s, P_1, \dots, P_s,$ `t`$)$
  Input: Subdomains $\Omega_1, \dots, \Omega_s$
  Input: Bases $P_1, \dots, P_s$
  Input: Option `t`: switch between HB and THB
  for $\ell = s$ **to** 1 **do**
    $L_\ell = \{\psi \in P_\ell \ : \ \psi|_{\Omega_\ell} \neq 0\}$
    foreach $\tau \in L_\ell$ **do**
      $\ell_m =$ `minLevelIntersecting`$(\operatorname{support} \tau)$
      if $\ell_m == \ell$ **then** `/* `$\tau \in H$ `due to (10) */`
        $\gamma =$ `addGenerator`$()$
        $m_{\tau,\gamma} = 1$ `/* New column of` $\mathbf{M}$ `with exactly one 1. */`
        $\mathbf{d} = (d_\psi)_{\psi \in P_\ell} = (0, \dots, 0, d_\tau = 1, 0, \dots, 0)$
        for $j = \ell + 1$ **to** ℓ_m **do**
          $\mathbf{d} =$ `refine`$(\mathbf{d}, j)$ `/* Now` $\mathbf{d} = (d_\psi)_{\psi \in P_j}$`. */`
          if `t` **then** `/* THB */`
            foreach $\psi \in P_j$ **do**
              if $\psi \in L_j \setminus H$ **then** `/* Save the coeff. */`
                $m_{\psi,\tau} = d_\psi$
              else `/* Truncate the coeff. */`
                $d_\psi = 0$
              end
            end
            `/* Now` $\sum_{\psi \in P_j} d_\psi \psi|_{\Omega_j} = \mathbf{T}_j \dots \mathbf{T}_i \tau|_{\Omega_j} = \gamma|_{\Omega_j}$. `*/`
          else `/* HB */`
            foreach $\psi \in L_j$ **do** `/* Save the coeff. */`
              $m_{\psi,\gamma} = d_\psi$
            end
          end
        end
      end
    end
  end

The lists L_ℓ are constructed by traversing the leaves of T and using the implementation of tensor-product B-splines. The procedure `minLevelIntersecting(box)` returns the minimum of $\{i : \Delta_i \cap$ `box` $\neq \emptyset\}$ and it is provided by the shared code. The procedure `refine`$(\mathbf{d}, j)$ uses Boehm's algorithm to compute the expansion

of the following function σ in terms of level j,

$$\sum_{\psi \in P_{j-1}} d_\psi \psi = \sigma = \sum_{\varphi \in P_j} c_{\varphi,\sigma} \varphi \ ,$$

and returns $(c_{\varphi,\sigma})_{\varphi \in P_j}$.

The levels are iterated from the finest to the coarsest. In this way the difference $L_j \setminus H$ can be computed because $H \cap P_j$ is already known. Consequently the full construction of the space can be performed in one loop over the levels. A different solution (used for instance in *G+Smo*) is to delay the computation of the expansion after determining the selected functions from all levels.

The fact that only the coefficients $m_{\psi,\tau}$ with $\psi \in L_j$ are saved is a memory optimization, the same code runs with P_j in place of L_j except for the test for $\psi \in L_j \setminus H$ that would be modified accordingly.

This strategy was tested against the reference implementation in *G+Smo*. The comparison showed both faster evaluation and smaller memory consumption for selected 2D examples.

Implementation 2 The choices above are the simplest, but they can cause a very high memory consumption. According to Subsection 3.1 the memory usage depends on the total number of rows in $\mathbf{M}$. For dyadic refinement of the P_i the number of rows grows as $2^{d(s+1)}$, where d is the domain dimension and s is the number of levels. Since each row requires a memory pointer, this means that an empty $\mathbf{M}$ for a 3D example with 10 levels exceeds 10 gigabytes in size.

The problem can be solved with slightly more complex code. The main idea is to remove the rows of $\mathbf{M}$ containing only zeros, that is, to define B_i and Δ_i so that $\psi|_{\Delta_i} = 0$ does not happen for any $\beta \in B_i$.

Denoting $\widetilde{T}_i$ the set of leaves of the binary partition tree representing the domains $\widetilde{\Delta}_i = \Omega_i \setminus \Omega_{i+1}$, define

$$T = \bigcup_{i=1}^{s} \widetilde{T}_i \ .$$

For each $\Delta_k \in T$ there is exactly one j such that $\Delta_k \in \widetilde{T}_j$; define

$$B_k = \{\beta \in P_j \ : \ \beta|_{\Delta_k} \neq 0\} \ .$$

Since Δ_k is a box (a Cartesian product of intervals), B_k is a tensor-product basis. Note that typically $\#T > s$ but all B_k are quite small.

The construction of $\mathbf{M}$ is done as in the previous implementation except for the necessary shifts of indices. This solution is not coded for (T)HB, but the required machinery was implemented for DHB.

5.3 Truncated B-Splines for Partially Nested Refinement

This is a generalization of (T)HB-splines that was proposed in [43]. It allows for independent refinement in different parts of the domain (see Fig. 4) and can help for multipatch geometries as shown in Example 3.

Fig. 4: Left: TBPN-splines allow to refine the subdomains Λ_a and Λ_b independently. Right: THB-splines require nested knot vectors for any pair of subdomains.

The requirement (9) is dropped and the sequence of nested domains is replaced by a partition of Ω into *patches* $\Lambda_1, \dots, \Lambda_s$. Note that here the word "patch" has a different meaning from the context of multipatch domains, cf. Subsection 2.3. The construction requires the following compatibility condition: if Λ_i and Λ_j share a $(d-1)$-dimensional interface $\Gamma_{i,j} = \partial\Lambda_i \cap \partial\Lambda_j$, then

$$\operatorname{span} P_i \subset \operatorname{span} P_j \quad \text{or} \quad \operatorname{span} P_j \supset \operatorname{span} P_i \ .$$

This means that $\{\operatorname{span} P_i \ : \ i = 1, \dots, s\}$ is not totally ordered anymore, only *partially ordered*. In particular, if the boundaries are disjoint or their intersection is not $(d-1)$-dimensional, the spaces $\operatorname{span} P_i$ and $\operatorname{span} P_j$ do not have to be comparable for inclusion. Note that the construction requires "sufficient separation" of the patches associated to two incomparable spaces. See [43] for details.

Basis functions are again a subset of $\bigcup_{i=1}^{s} P_i$ and are selected using a modification of Kraft's procedure based on *slave functions*. A function $\psi \in P_i$ is called a *slave* if it is active on an $(n-1)$-dimensional interface $\Gamma_{i,j}$ with $\operatorname{span} P_j \subset \operatorname{span} P_i$. The set of slaves of level i can be written as

$$S_i = \{\psi \in P_i \ : \ \exists j : \psi|_{\Gamma_{i,j}} \neq 0, \operatorname{span} P_j \subset \operatorname{span} P_i, \dim \Gamma_{i,j} = d-1\} \ .$$

The above can be explained as follow. Slave functions are the generators in P_i whose coefficient is uniquely determined by the restriction of the function and its derivatives (up to the smoothness) on the interfaces $\Gamma_{i,j}$ with $\operatorname{span} P_j \subset \operatorname{span} P_i$. This means that their coefficients are determined by the coefficients of the functions of coarser bases together by the smoothness on the interfaces.

The selected functions are defined by

$$M = \bigcup_{i=1}^{s} M_i \ ,$$

where M_i contains the *master functions* of level i, i.e., the functions of P_i that are active on Λ_i and that are not slaves:

$$M_i = \{\psi \in P_i : \psi|_{\Lambda_i} \neq 0, \ \psi \notin S_i\} \ . \tag{12}$$

Truncation is defined in the same way as in the case of THB-splines. The resulting basis is called *truncated B-splines for partially nested refinement* (TBPN). The set M forms a non-negative partition of unity, it is a basis and, similarly to THB, it preserves the coefficients of polynomial representation. Moreover, if (9) holds, then TBPN reduces to THB with the same bases and appropriate subdomains. See [43] for details.

Implementation Only the truncated version of the construction was implemented. The partition T can be defined as

$$\Delta_i = \Lambda_i, \quad i = 1, \dots, s$$

and B by

$$B_i = P_i, \quad i = 1, \dots, s \ .$$

The matrix $\mathbf{M}$ is built iteratively while discovering the functions selected by the modified Kraft procedure. First the bases $P_1, \dots, P_s$ are analyzed and the nesting relations are stored in a matrix $\mathbf{Z}$. Then, as for (T)HB, the lists L_i of the functions in P_i that are active on Δ_i are computed.

For each function $\psi \in L_i$, $i = s, \dots, 1$, the modified Kraft conditions (12) are tested. The test requires the computation of the intersections $\Gamma_{i,j} \cap \operatorname{support} \psi$ that is achieved by computing $\operatorname{support} \psi \cap \Lambda_i$ and then decomposing its boundary into segments. If $\dim(\Gamma_{i,j} \cap \operatorname{support} \psi) = d - 1$ for some j with $\operatorname{span} P_j \subset \operatorname{span} P_i$ then ψ is a slave and it is saved in the list S_i. Otherwise the conditions (12) are satisfied and a new column is added to $\mathbf{M}$. The coefficients $m_{\beta,\gamma}$ are computed using a recursive algorithm. For all j such that $\operatorname{span} P_i \subset \operatorname{span} P_j$ and $\dim(\Gamma_{i,j} \cap \operatorname{support} \psi) = d-1$ the expansion of ψ with respect of P_j is computed by knot insertion. Then for each functions in S_j with a nonzero coefficient the procedure is repeated, giving the coefficients of slaves of finer levels. It is possible that the same $\beta \in B_k$ appears during different recursions while computing the representation of the same generator γ. In this case the sum of the coefficients computed from functions of the same coarsest level must be saved in $\mathbf{M}$.

The implementation described has the same problem as the first implementation of (T)HB: unreasonable memory consumption for the 3D case. This can be solved by using the same strategy described for (T)HB.

5.4 Decoupled Hierarchical B-Splines

Contrarily to tensor-product B-splines, (T)HB do not always span the full space of piecewise polynomials on their mesh [33]. This observation was the starting point of the development of TDHB [32]. There *decoupling* is used in conjunction with truncation in order to enlarge the space and span the full piecewise polynomial space for a broader class of meshes. A modification of TDHB called decoupled hierarchical B-splines was coded and it is presented here for the first time. The novelty is that truncation is abandoned in favor of recursive decoupling. By doing so the spanned space can be further enlarged as showed in Example 1.

First, decoupling will be introduced in a slightly more general version compared to [32]. Let $\varphi \in \operatorname{span} P$ be a function, let $c_{\varphi,\psi}$ be the coefficients of its expansion with respect to P,

$$\varphi = \sum_{\psi \in P} c_{\varphi,\psi} \psi \ ,$$

and let $\Theta \subseteq \Omega$ be a domain. The *decoupling graph* $R(\varphi, P, \Theta)$ is the graph whose vertices are

$$R_V(\varphi, P, \Theta) = \{\psi \in P \ : \ c_{\varphi,\psi} \neq 0\} \tag{13}$$

and the edges are

$$R_E(\varphi, P, \Theta) = \{(\psi, \psi') \ : \ \operatorname{support} \psi \cap \operatorname{support} \psi' \cap \overline{\Theta} \neq \emptyset\} \ . \tag{14}$$

The *decoupling operator* $\mathbf{D}_{P,\Theta}$ associates to function $\varphi \in \operatorname{span} P$ one or more *decoupled functions* in $\operatorname{span} P$:

$$\mathbf{D}_{P,\Theta} : \varphi \mapsto \left\{ \sum_{\psi \in K} c_{\varphi,\psi} \psi \ : \ K \text{ is a connected component of } R(\varphi, P, \Theta) \right\} \ .$$

Let $P_1 \subset \cdots \subset P_s$ and $\Omega_1 \supseteq \cdots \supseteq \Omega_s \supset \Omega_{s+1} = \emptyset$ be as in (T)HB. Denote $D_s = P_s$ and

$$D_i = \bigcup_{\psi \in P_i} \mathbf{D}_{D_{i+1}, \Omega \setminus \Omega_{i+1}}(\psi) \ . \tag{15}$$

Then using Kraft's method the *decoupled hierarchical basis* (DHB-splines) D is defined as

$$D = \bigcup_{i=1}^{s} \{\varphi \in D_i \ : \ \operatorname{support} \varphi \subseteq \Omega_i \text{ and } \operatorname{support} \varphi \cap (\Omega_i \setminus \Omega_{i+i}) \neq \emptyset\} \ .$$

Given $P_1, \ldots, P_s$ and $\Omega_1, \ldots, \Omega_s$, all of the HB, THB, TDHB and DHB bases are defined. Denoting Z the TDHB basis from [32], the following inclusions hold

$$\operatorname{span} H = \operatorname{span} H' \subseteq \operatorname{span} Z \subseteq \operatorname{span} D \ .$$

Recall that H, H' and D are the HB, THB and DHB bases, respectively, cf. Subsection 5.2. See also Example 1.

Implementation For DHB it is not possible to identify the local bases B_i with the defining bases P_i. This is because a function $\psi \in P_i$ can be decoupled in multiple functions that must be distinguished.

The definitions of T and B are the same as in Implementation 2 in Subsection 5.2. The construction of $\mathbf{M}$ follows the definition of the space. First, each D_i is constructed: for each function in D_i its expansion with respect of D_{i+1} and its originating function in P_i are stored. Then the Kraft selection mechanism is employed and for each selected function a column is inserted in $\mathbf{M}$. Computing the coefficient $m_{\beta,\gamma}$ for $\gamma \in D_i$ and $\beta \in P_j$ $(j > i)$ is performed by first composing the precomputed change of bases from D_i to D_j and then storing the obtained coefficients according to the subdomain and the originating function.

5.5 Hierarchical Locally Refined Splines

HLR-splines are a special case of LR-splines. Let K denote a partition of Ω into boxes and let μ denote a function that assigns each interface between two boxes a nonnegative integer. To the triplet (K, μ, p) corresponds the spline space $\mathcal{S}$ of the piecewise polynomials of degree p in each variable on K and such that their smoothness across each interface Γ between two boxes is greater than or equal to $p - \mu(\Gamma)$.

A B-spline β is *nested* in a B-spline β' relatively to $\mathcal{S}$, written as $\beta \prec \beta'$, if there exists a sequence of B-splines $\beta = \beta_1, \ldots, \beta_n = \beta'$ such that each $\beta_i \in \mathcal{S}$ and such that each β_{i+1} is obtained from β_i by knot insertion.

LR-splines are the set of minimal elements for the ordering $\prec$ that are comparable with at least one Bernstein polynomial on Ω. They can be linearly dependent and do not necessarily span the entire space of piecewise polynomials satisfying the smoothness conditions [14,5]. However, many properties of the generators are linked together, in particular local linear independence and partition of unity property are equivalent [4].

HLR-splines are a class of LR-splines enjoying local linear independence and thus also the partition of unity property. This is achieved by mimicking the HB approach in constructing K. Take a sequence of tensor-product B-spline spaces $\mathcal{V}_1 \subset \ldots \subset \mathcal{V}_s$ with $\mathcal{V}_i = \operatorname{span} P_i$ and a corresponding sequence of subdomains $\Omega_1 \supseteq \ldots \supseteq \Omega_s \supset \Omega_{s+1} = \emptyset$. Then define

$$K = \bigcup_{i=1}^{s} \{\Theta \text{ element of the partition corr. to } \mathcal{V}_i \,:\, \Theta \subseteq \Omega_i \setminus \Omega_{i+1}\} \tag{16}$$

and μ that describes the smoothness of the space $\mathcal{V}_i$ on $\Omega_i \setminus \Omega_{i+1}$. Then assuming that each $\mathcal{V}_i$ is obtained from $\mathcal{V}_{i-1}$ by refining a single tensor-component of $\mathcal{V}_{i-1}$ (i.e., h-refinement in a single direction denoted by u_i) and that the borders of the subdomains Ω_i are sufficiently separated, the generators form a partition of unity and they are locally linearly independent [5].

Implementation A simple choice is to define T by:

$$\Delta_i = \Omega_i \setminus \Omega_{i+1}, \quad i = 1, \ldots, s$$

and B by

$$B_i = P_i, \quad i = 1, \ldots, s \ .$$

The generators in HLR-splines are either function from P_i for some i or are obtained from a function of P_i by inserting a sequence of knots in u_i-th component of its knot vector. Both types of functions must be active on Ω_i. To find these functions, P_i is projected to a $(d-1)$-variate spaces $\bar{P}_i$ by ignoring its u_i-th tensor factor. For each function $\bar{\psi} \in \bar{P}_i$ the description of the subdomains T is restricted to support $\bar{\psi} \times \mathbb{R} \cap \Omega$, where $\mathbb{R}$ is in direction u_i. This information is used to build a sequence of knot-vectors for the u_i-th direction that describes all the functions in P_i or refinement of functions of P_i that are in LR and have the same knot vector as $\bar{\psi}$ in the directions different from u_i. For each such function a column is added to $\mathbf{M}$ and filled with the appropriate coefficients.

5.6 Implementation Size

This subsection reports the amount of code that implements the spaces described above. While this is a debatable metric, it is the standard approximation of the coding effort and it highlights the amount of code shared between the different spaces. The data reported is the result of the *cloc* tool [12] run on appropriate subsets of the files.

The code is in C++03 standard and contains verbose template parts that can be avoided. Only the (T)HB space is a complete implementation with initialization, refinement (with possible coefficient update) and serialization to the *G+Smo* xml format. For the other spaces only the initialization is provided.

The line count for the different components can be read in Tables 3 and 4. As shown, most of the code implements the shared functionality and the space-specific code amounts to roughly 500 lines per space. The numbers compare favourably with the size of the reference implementation of HB and THB in *G+Smo* that together amount to roughly 6000 lines of code.

Table 3: Lines of the shared code

	files	blank	comment	code
utils	5	198	96	1008
domain code (T)	11	352	302	1998
matrix code ($\mathbf{M}$)	4	133	248	483
base-class (`eval`)	2	105	59	443
Total	22	788	705	3932

Table 4: Lines of code of specific spaces.

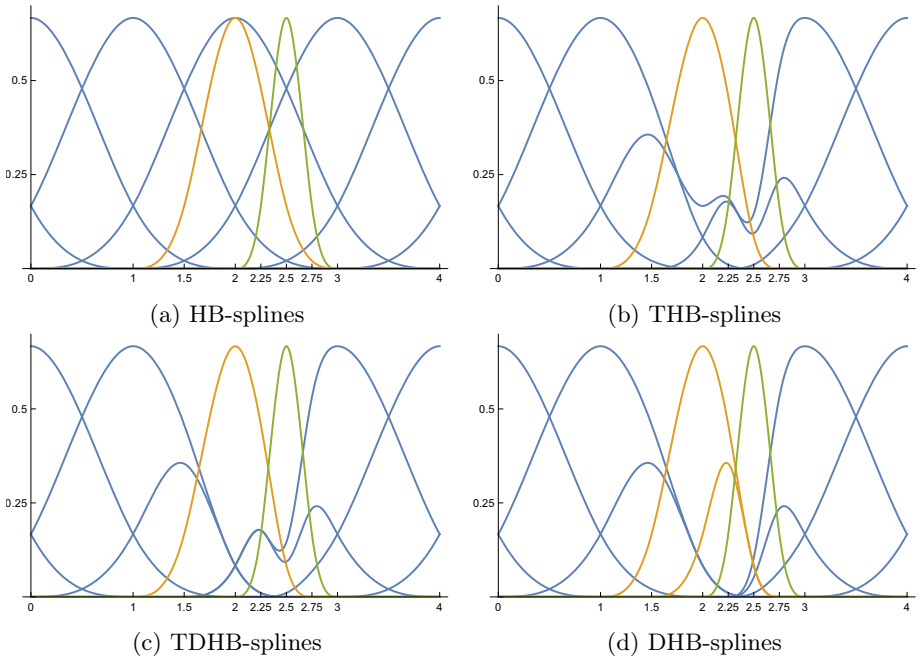
	files	blank	comment	code
(T)HB	1	88	73	421
TBPN	1	80	22	396
DHB	1	92	31	548
HLR	1	127	33	550
Total	4	387	159	1915

6 Examples

This section contains selected examples that can be useful to grasp the similarities and the differences between the implemented spline spaces. The basis functions have been plotted with *ParaView* [1] using the data produced with the implementations described in the previous section. The only exception is

Example 1 that has been created in Mathematica, because it involves TDHB [32], which we did not implement.

Example 1. Compare the cubic univariate HB, THB, TDHB and DHB with dyadic refinement, the level 0 knot vector $(\dots, -1, 0, 1, \dots)$ and $\Omega_0 = [0, 4]$, $\Omega_1 = [1, 3]$ and $\Omega_2 = [2, 3]$. Figure 5 shows all the basis functions of each spline space and highlights the level of the B-spline from which they where derived by truncation or decoupling. Table 5 lists the number of generators according to the level of the original B-spline. Note that DHB is the only space that spans the entire space of $\mathcal{C}^2$ piecewise cubic polynomials on the mesh.



(a) HB-splines (b) THB-splines

(c) TDHB-splines (d) DHB-splines

Fig. 5: Comparison of various hierarchical bases. Generators originating from functions of level 0 in blue, from level 1 in orange and from level 2 in green.

Example 2. Consider bivariate hierarchical splines of bi-degree $(4, 4)$ on a mesh shown in Fig. 6. The function with the knot lines indicated in red is selected in the hierarchical basis (Fig. 7 left), truncated in the truncated hierarchical basis (Fig. 7 right) and decoupled into four different functions (that are selected) in the decoupled hierarchical basis (Fig. 8).

Note that due to properties of DHB the sum of the four functions in Fig. 8 equals the truncated function in Fig. 7 right.

Table 5: Number of generators by the level of the originating function for Example 1.

	HB	THB	TDHB	DHB
level 0	7	7	8	8
level 1	1	1	1	2
level 2	1	1	1	1
total	9	9	10	11

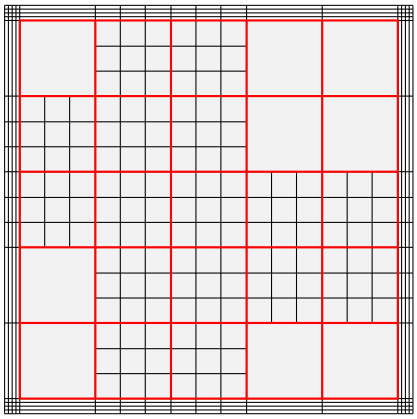


Fig. 6: Hierarchical mesh and a support of a function from Example 2.

Example 3. The design process often involves several patches. To achieve continuity between the patches without losing accuracy, it is necessary that the restrictions of the two spaces are compatible on the interface. That means that one space has to be a subspace of the other.

Sometimes a new patch must be introduced to bridge between two given patches that should not be modified. Thus the restriction of the space of the bridge patch to each boundary must be a superspace of the restrictions of the other space. If the two given patches have different knot vectors, THB-splines would lead to significant refinement. On the other hand, the TBPN space can achieve interface compatibility without adding unnecessary degrees of freedom.

The bicubic THB basis on the mesh depicted in Fig. 9 has 72 degrees of freedom, whereas the TBPN basis on the same mesh has only 60.

Example 4. Cubic HB, THB, DHB and HLR are compared on a mesh shown in Fig. 10. For each of these spaces all the basis functions are plotted in Fig. 11. Note that the number of basis function in the middle of the patch is higher for HLR and DHB. In particular, HB and THB basis have 49 elements each; HLR and DHB have 53 and are complete, as the meshes fulfill the assumptions from [5] and [32].

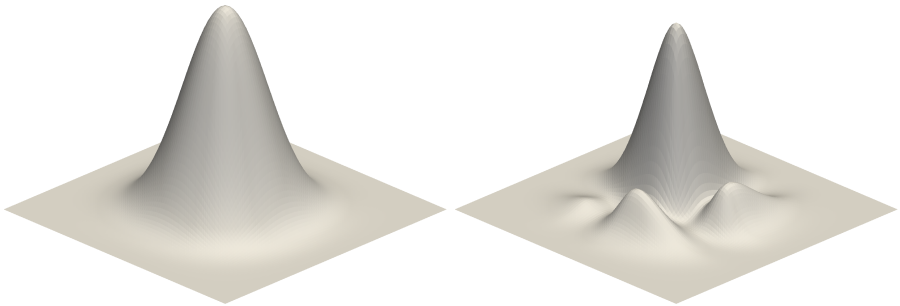


Fig. 7: Function with the support in Fig. 6 as selected into the hierarchical basis (left) and truncated in the truncated basis (right).

7 Conclusions

The effectiveness of the proposed implementation framework is demonstrated by the implementation of various spline spaces that share the same evaluation code. The space-specific code is reduced to the initialization of the required data structures as demonstrated by the implementations of HB, THB, TBPN, DHB and HLR. Moreover, the proposed approach grants the following advantages:

1. code reduction both by sharing evaluation between different spaces and between spaces and functions;
2. arbitrary local bases that, in principle, open the way to experimentation with hierarchical constructions based on generalized splines [3], or to the use of ad-hoc functions near a priori known singularities;
3. transparent handling of multipatch domains.

Acknowledgments

The authors have been supported by the Austrian Science Fund (FWF, NFN S117 “Geometry + Simulation”) and by the Seventh Framework Programme of the EU (project EXAMPLE, GA No. 324340). This support is gratefully acknowledged. The authors would also like to thank Dr. Rafael Vázquez for commenting on an earlier version of this paper and to the reviewers for their valuable suggestions.

References

1. Ayachit, U.: The paraview guide: a parallel visualization application (2015)
2. Borden, M.J., Scott, M.A., Evans, J.A., Hughes, T.J.R.: Isogeometric finite element data structures based on Bézier extraction of NURBS. Internat. J. Numer. Methods Engrg. 87(1-5), 15–47 (2011)

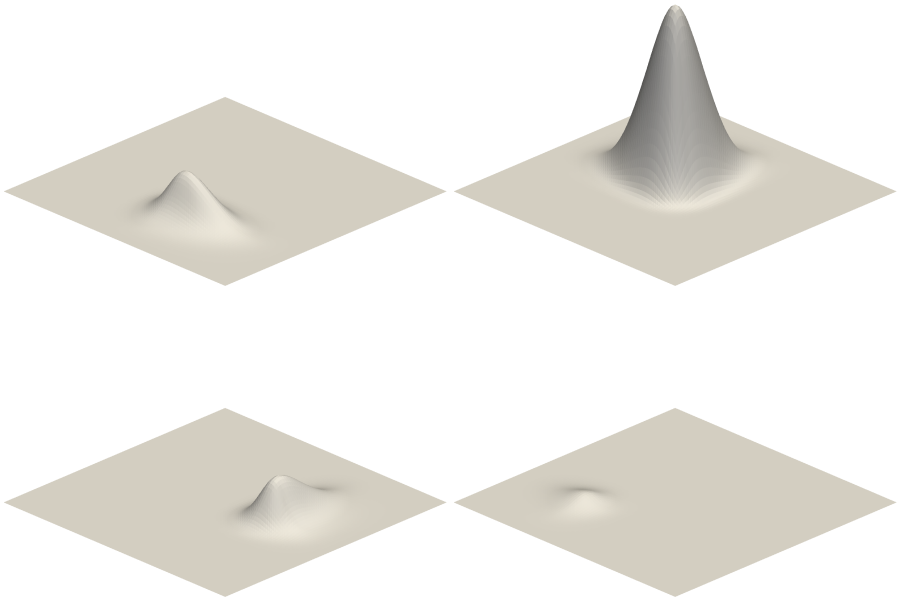


Fig. 8: Decoupled functions stemming from the B-spline with the support indicated in Fig. 6.

3. Bracco, C., Lyche, T., Manni, C., Roman, F., Speleers, H.: Generalized spline spaces over T-meshes: dimension formula and locally refined generalized B-splines. Appl. Math. Comput. 272(part 1), 187–198 (2016)
4. Bressan, A.: Some properties of LR-splines. Comput. Aided. Geom. Design 30(8), 778–794 (2013)
5. Bressan, A., Jüttler, B.: A hierarchical construction of LR meshes in 2D. Comput. Aided. Geom. Design 37, 9–24 (2015)
6. Brovka, M., López, J., Escobar, J., Montenegro, R., Cascón, J.: A simple strategy for defining polynomial spline spaces over hierarchical T-meshes. Comput.-Aided Des. 72, 140–156 (2016)
7. Buchegger, F., Jüttler, B., Mantzaflaris, A.: Adaptively refined multi-patch B-splines with enhanced smoothness. Appl. Math. Comput. 272(part 1), 159–172 (2016)
8. Buffa, A., Garau, E.M.: Refinable spaces and local approximation estimates for hierarchical splines. IMA J. Numer. Anal. drw035 (2016)
9. Buffa, A., Giannelli, C.: Adaptive isogeometric methods with hierarchical splines: error estimator and convergence. Math. Models Methods Appl. Sci. 26(1), 1–25 (2016)
10. Collin, A., Sangalli, G., Takacs, T.: Analysis-suitable G^1 multi-patch parametrizations for C^1 isogeometric spaces. Computer Aided Geometric Design 47, 93–113 (2016)
11. Da Veiga, L.B., Buffa, A., Sangalli, G., Vázquez, R.: Analysis-suitable T-splines of arbitrary degree: definition, linear independence and approximation properties. Math. Models Methods Appl. Sci. 23(11), 1979–2003 (2013)

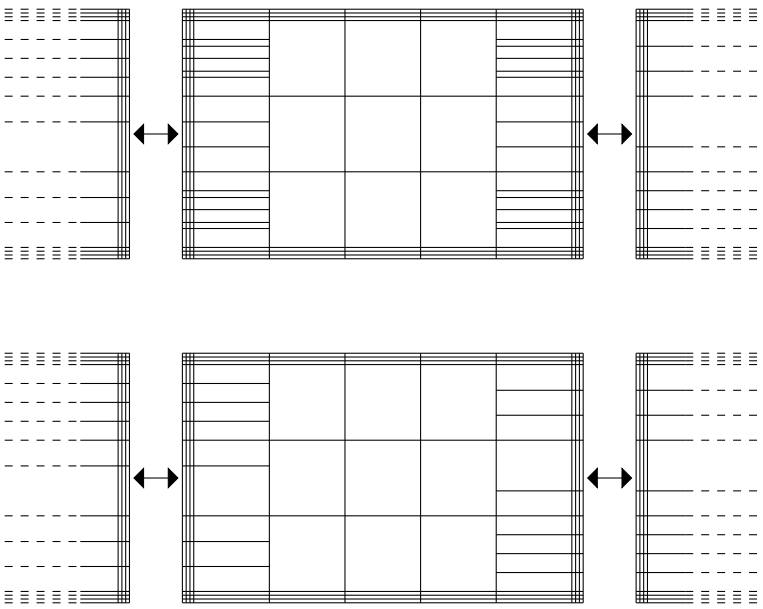


Fig. 9: Meshes from Example 3. Top: THB mesh; bottom: TBPN mesh.

12. Danial, A.: cloc: Count Lines of Code (2006–2017), `https://github.com/AlDanial/cloc`
13. Deng, J., Chen, F., Li, X., Hu, C., Tong, W., Yang, Z., Feng, Y.: Polynomial splines over hierarchical T-meshes. Graph. Models 70(4), 76–86 (2008)
14. Dokken, T., Lyche, T., Pettersen, K.F.: Polynomial splines over locally refined box-partitions. Comput. Aided. Geom. Design 30(3), 331–356 (2013)
15. Evans, E.J., Scott, M.A., Li, X., Thomas, D.C.: Hierarchical T-splines: analysis-suitability, Bézier extraction, and application as an adaptive basis for isogeometric analysis. Comput. Methods Appl. Mech. Engrg. 284, 1–20 (2015)
16. Forsey, D.R., Bartels, R.H.: Hierarchical B-spline refinement. SIGGRAPH Comput. Graph. 22(4), 205–212 (Jun 1988)
17. Giannelli, C., Jüttler, B., Speleers, H.: THB-splines: the truncated basis for hierarchical splines. Comput. Aided. Geom. Design 29(7), 485–498 (2012)
18. Giannelli, C., Jüttler, B., Speleers, H.: Strongly stable bases for adaptively refined multilevel spline spaces. Adv. Comput. Math. 40(2), 459–490 (2014)
19. Geometry + simulation modules (G+Smo): Open source C++ library for isogeometric analysis (2016), `http://www.gs.jku.at/gismo`
20. GoTools: Collection of C++ libraries connected to geometry (2016), `https://github.com/SINTEF-Geometry/GoTools`
21. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010), `http://eigen.tuxfamily.org`
22. Hennig, P., Müller, S., Kästner, M.: Bézier extraction and adaptive refinement of truncated hierarchical NURBS. Comput. Methods Appl. Mech. Engrg. 305, 316–339 (2016)
23. Hennig, P., Kästner, M., Morgenstern, P., Peterseim, D.: Adaptive mesh refinement strategies in isogeometric analysis—a computational comparison. Comput. Methods Appl. Mech. Engrg. 316, 424–448 (2016)

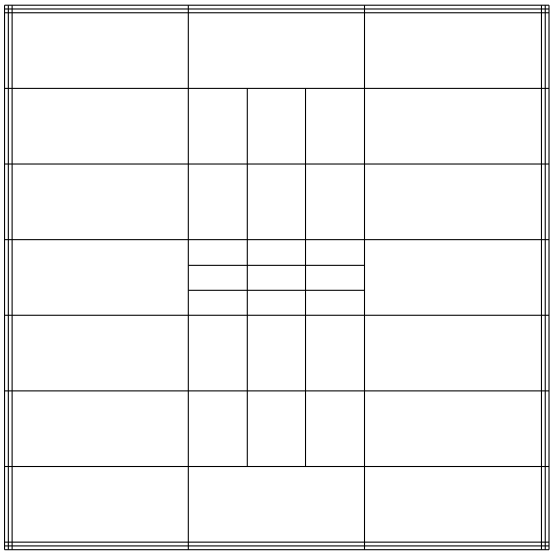
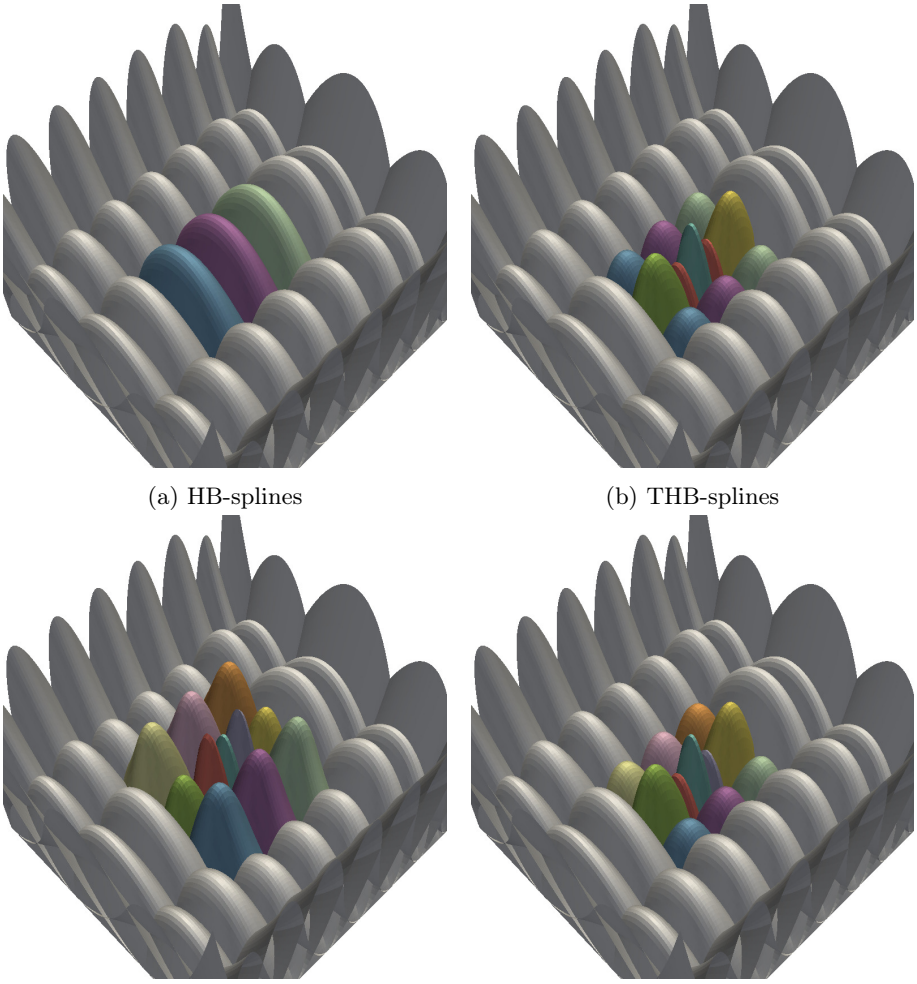


Fig. 10: Mesh from Example 4.

24. Hughes, T.J.R., Cottrell, J.A., Bazilevs, Y.: Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. Comput. Methods Appl. Mech. Engrg. 194(39-41), 4135–4195 (2005)
25. Johannessen, K.A., Remonato, F., Kvamsdal, T.: On the similarities and differences between classical hierarchical, truncated hierarchical and LR B-splines. Comput. Methods Appl. Mech. Engrg. 291, 64–101 (2015)
26. Jüttler, B., Langer, U., Mantzaflaris, A., Moore, S.E., Zulehner, W.: Geometry + simulation modules: Implementing isogeometric analysis. PAMM 14(1), 961–962 (2014)
27. Kang, H., Xu, J., Chen, F., Deng, J.: A new basis for PHT-splines. Graph. Models 82, 149–159 (2015)
28. Kapl, M., Vitrih, V., Jüttler, B., Birner, K.: Isogeometric analysis with geometrically continuous functions on two-patch geometries. Comput. Math. Appl. 70(7), 1518–1538 (2015)
29. Kiss, G., Giannelli, C., Jüttler, B.: Algorithms and data structures for truncated hierarchical B-splines. In: Floater, M., Lyche, T., Mazure, M.L., Mørken, K., Schumaker, L.L. (eds.) Mathematical methods for curves and surfaces. MMCS 2012, Lecture Notes in Computer Science, vol. 8177, pp. 304–323. Springer, Heidelberg (2014)
30. Kraft, R.: Adaptive and linearly independent multilevel B–splines. In: Le Méhauté, A., Rabut, C., Schumaker, L.L. (eds.) Surface Fitting and Multiresolution Methods. pp. 209–218. Vanderbilt University Press, Nashville (1997)
31. Li, X., Zheng, J., Sederberg, T.W., Hughes, T.J.R., Scott, M.A.: On linear independence of T-spline blending functions. Comput. Aided. Geom. Design 29(1), 63–76 (2012)
32. Mokriš, D., Jüttler, B.: TDHB-splines: the truncated decoupled basis of hierarchical tensor-product splines. Comput. Aided. Geom. Design 31(7-8), 531–544 (2014)



(a) HB-splines (b) THB-splines

(c) HLR-splines (d) DHB-splines

Fig. 11: Details of the bases from Example 4. Only the basis functions that differ have been marked in colour. Note that not all the HB basis functions are visible: three are hidden in the central area.

33. Mokriš, D., Jüttler, B., Giannelli, C.: On the completeness of hierarchical tensor-product B-splines. J. Comput. Appl. Math. 271, 53–70 (2014)
34. Morgenstern, P.: Globally structured three-dimensional analysis-suitable T-splines: Definition, linear independence and m-graded local refinement. SIAM J. Numer. Anal. 54(4), 2163–2186 (2016)
35. Morgenstern, P., Peterseim, D.: Analysis-suitable adaptive T-mesh refinement with linear complexity. Comput. Aided. Geom. Design 34, 50–66 (2015)
36. Rabut, C.: Locally tensor product functions. Numer. Algorithms 39(1-3), 329–348 (2005)

37. Scott, M.A., Borden, M.J., Verhoosel, C.V., Sederberg, T.W., Hughes, T.J.R.: Isogeometric finite element data structures based on Bézier extraction of T-splines. Internat. J. Numer. Methods Engrg. 88(2), 126–156 (2011)
38. Sederberg, T.W., Cardon, D.L., Finnigan, G.T., North, N.S., Zheng, J., Lyche, T.: T-spline simplification and local refinement. ACM Trans. Graph. 23(3), 276–283 (Aug 2004)
39. Sederberg, T.W., Zheng, J., Bakenov, A., Nasri, A.: T-splines and T-NURCCs. ACM Trans. Graph. 22(3), 477–484 (Jul 2003)
40. Thibault, W.C., Naylor, B.F.: Set operations on polyhedra using binary space partitioning trees. In: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques. pp. 153–162. SIGGRAPH '87, ACM, New York, NY, USA (1987)
41. Toth, C.D., O'Rourke, J., Goodman, J.E.: Handbook of discrete and computational geometry. CRC press (2004)
42. Vázquez, R., Garau, E.: Algorithms for the implementation of adaptive isogeometric methods using hierarchical splines. Tech. Rep. 16-08, IMATI-CNR, Pavia (July 2016)
43. Zore, U.: Constructions and Properties of Adaptively Refined Multilevel Spline Spaces. Dissertation, Johannes Kepler University Linz (2016), `http://epub.jku.at/obvulihs/download/pdf/1273941`