Algorithms and Data Structures for Truncated Hierarchical B–splines

Gábor Kiss[1], Carlotta Giannelli[2], and Bert Jüttler[2]

[1] Doctoral Program “Computational Mathematics”
[2] Institute of Applied Geometry
Johannes Kepler University Linz, Altenberger Str. 69, 4040 Linz, Austria
e–mail: gabor.kiss@dk-compmath.jku.at, carlotta.giannelli@jku.at, bert.juettler@jku.at

Abstract. Tensor–product B–spline surfaces are commonly used as standard modeling tool in Computer Aided Geometric Design and for numerical simulation in Isogeometric Analysis. However, when considering tensor–product grids, there is no possibility of a localized mesh refinement without propagation of the refinement outside the region of interest. The recently introduced truncated hierarchical B–splines (THB–splines) [5] provide the possibility of a local and adaptive refinement procedure, while simultaneously preserving the partition of unity property. In this paper we present an effective implementation of the fundamental algorithms needed for the manipulation of THB–spline representations. By combining a quadtree data structure — which is used to represent the nested sequence of subdomains — with a suitable data structure for sparse matrices, we obtain an efficient technique for the construction and evaluation of THB–splines.

Keywords: hierarchical tensor–product B–splines; truncated basis; THB–splines; isogeometric analysis; local refinement

1 Introduction

The *de facto* standard in computer aided geometric design is the tensor–product B–spline model together with its non–uniform rational extension (NURBS). Among other fundamental properties, like minimum support, efficient refinement and degree–elevation algorithms, B–splines are nonnegative and form a partition of unity. This implies that a B–spline curve/surface is completely contained in the *convex hull* of a certain set of points, usually referred to as control net. The shape of the control net directly influences the shape of the B–spline representation, so that the designer can use it to manipulate the corresponding parametric representation in a fairly intuitive way. Unfortunately, an unavoidable drawback of the tensor–product structure is a global nature of the mesh refinement which excludes the possibility of a local refinement scheme as illustrated in Figure 1(a–c).

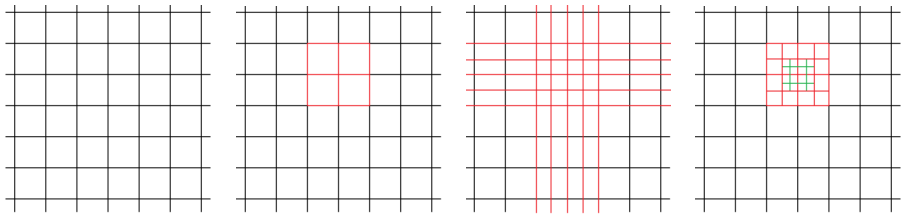


Fig. 1. Adaptive refinement of an initial tensor–product grid (a) with respect to a localized region (b) may be achieved by avoiding a propagation of the refinement due to the tensor–product structure (c) through a hierarchical approach (d).

Despite an increasing interest in the identification of adaptive spline spaces and related applications, see e.g., [7, 15, 17], local mesh refinement remains a non–trivial and computationally expensive issue. A suitable trade–off between the quality of the geometric representation (in terms of degrees of freedom needed to obtain a certain accuracy) and the complexity of the mesh refinement algorithm has necessarily to be taken into account. Different approaches have been proposed which all extend the standard tensor–product model by allowing T–junctions between axis aligned mesh segments. Among others, this led to the introduction of hierarchical B–splines (HB–splines) [4, 11, 12], T–splines [16], polynomial splines over T–meshes [2] and – more recently – truncated hierarchical B–splines (THB–splines) [5] and locally refined B–splines [3].

The idea of performing surface modeling by manipulating the parametric representation at different levels of details was originally proposed by Forsey and Bartels [4]. In order to localize the editing of detailed features, the refinement is iteratively adapted on restricted patches of the surface in terms of a sequence of *overlays* with nested knot vectors. Subsequently, Kraft [11, 12] showed that the hierarchical structure enforced on the mesh refinement procedure can be complemented by a simple and automatic identification of basis functions which naturally generalize some of the fundamental properties of tensor–product B–splines — such as nonnegativity and linear independence — to the case of HB–splines.

The multilevel approach allows to break the rigidity of a tensor–product configuration by simultaneously preserving an highly organized structure as shown in Figure 1(d). An example of hierarchical refinements over rectangular–shape regions is presented in Figure 2.

The hierarchical B–spline model found applications in data interpolation and approximation [10, 11, 13], as well as in finite element and isogeometric analysis [1, 14, 17]. Alternative spline hierarchies were also considered in the literature, see e.g., [9, 18].

Kraft's basis for HB–splines does not possess the partition of unity property without additional scaling and it possess only limited stability properties. Truncated hierarchical B–splines [5] have the potential to overcome these limitations and provide improved sparsity properties. They were introduced as a possible extension of *normalized* tensor–product B–splines to suitably handle the local refinement in adaptive surface approximation algorithms. This multilevel scheme was also generalized and further investigated in [6], where particular attention was devoted to the stability analysis of the proposed hierarchical construction.

In virtue of the multilevel nature of the hierarchical B–spline approach, the natural choice in terms of data structures is a tree–like representation where a given refinement level correspond to a certain level of depth in the tree [4]. Related and alternative solutions were also further investigated. An algorithm for scattered data interpolation and approximation by multilevel bicubic B–splines based on a hierarchy of control lattices was described in [13]. An implementation of hierarchical B–splines in terms of a tree data structure whose nodes represent the B–splines from different levels was recently presented in [1]. Another solution consists of storing in each node of the tree the data related to a knot span of a certain level, in particular the significant basis functions acting on it [14].

The goal of the present paper is to introduce an effective implementation of data structures and algorithms for the newly introduced THB–splines. To represent the subdomain hierarchy we use a *quadtree* data structure in combination with *sparse matrices.* The *quadtree* provides an efficient and dynamic data structure for representing the subdomains. It also facilitates the needed update which may be caused by an iterative refinement process. One key motivation for this choice is to reduce the memory overhead in need for storing the subdomain hierarchy as much as possible. The selection of (possibly truncated) basis functions proceeds as described in [5] by means of certain queries which use the quadtree. The result is encoded by a sequence of *sparse matrices.* The quadtree and the related sparse matrices are initially created and subsequently updated during the refinement procedure. For the hierarchical spline evaluation algorithm, however, only the access to the sparse matrices is required. This leads to a reasonable trade–off with respect to memory and time consumption during both the *construction* of THB–splines from an underlying subdomain hierarchy and their *evaluation* for given parameter values.

The paper is organized as follows. In Section 2 we describe the hierarchical approach to adaptive mesh refinement together with the definition and evaluation of the THB–spline basis. Section 3 introduces the data structures and algorithms used for the representation of the subdomain hierarchy, while Section 4 explains the construction of the matrices needed during the THB–spline evaluation in more detail. Some numerical results are then presented in Section 5 to illustrate the performance of our approach. Finally, Section 6 concludes the paper.

2 THB–splines

We define an adaptive extension of the classical tensor–product B–spline construction in terms of a certain number N of hierarchical *levels* which correspond

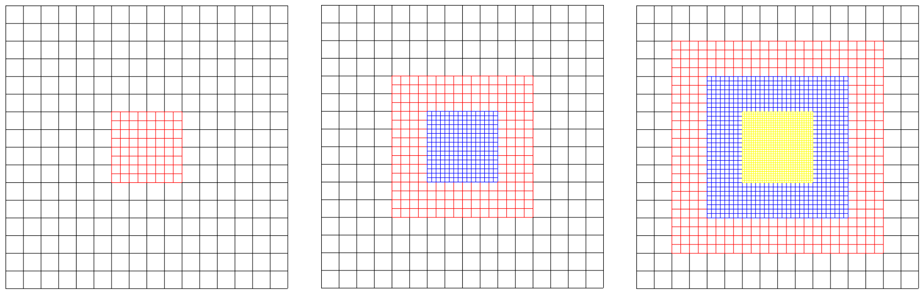


Fig. 2. An example of hierarchical refinement over rectangular–shape regions where the central area of the mesh is always refined up to the maximum level of detail: two levels (left), three levels (middle) and four levels (right).

to an increasing level of detail. At each refinement step we select a specific tensor–product grid associated with the current level. Provided that the sequence of tensor–product grids corresponds to a nested sequence of spline spaces $V^0, \ldots, V^{N-1}$ which satisfies

$$V^{\ell-1} \subset V^{\ell},$$

for $\ell = 1, \ldots, N-1$, the hierarchical framework allows to consider different types of grid refinement.

The present paper focuses on the *bivariate* tensor–product case. However, the framework can easily be adapted to the multivariate setting and even to more general spline spaces [5, 6]. Nevertheless, even if the representation model we are going to introduce may in principle be used to handle non–uniform mesh refinement and even spaces generated by degree elevation, we will consider only the dyadic uniform case throughout this paper.

More precisely, we assume that the coarsest spline space V^0 is spanned by bivariate tensor–product B–splines with respect to two bi–infinite uniform knot sequences. The finer spaces V^ℓ are obtained by iteratively applying dyadic subdivision, i.e., each cell of the original tensor-product grid is split uniformly into four cells.

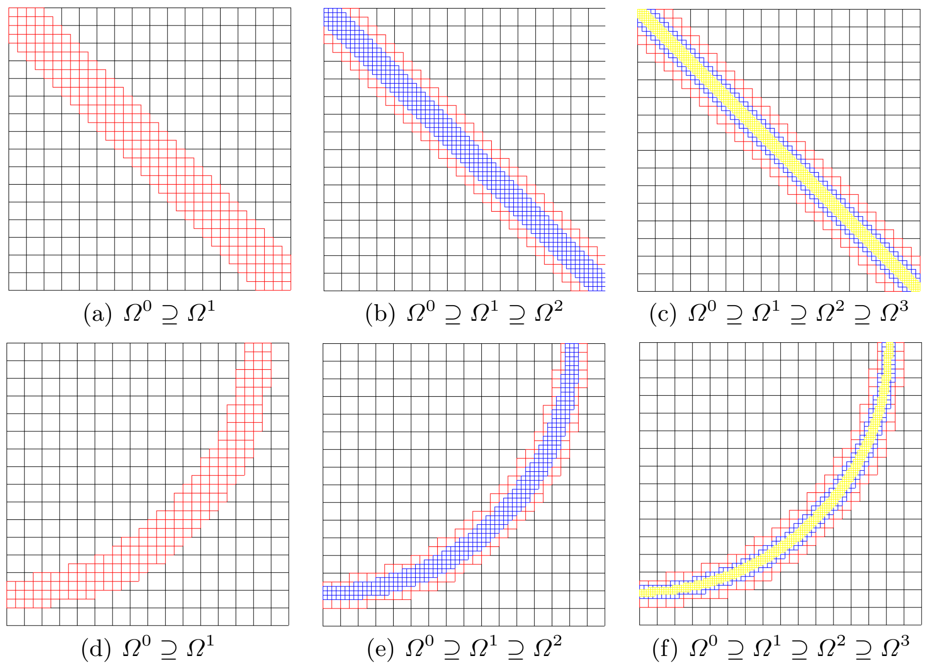
Let Ω^0 be a rectangular planar domain whose edges are aligned with the tensor-product grid of V^0, and let $\{\Omega^\ell\}_{\ell=0,\ldots,N-1}$ be a nested sequence of subdomains so that

$$\Omega^{\ell-1} \supseteq \Omega^{\ell}, \tag{1}$$

for $\ell = 1, \ldots, N-1$. Each Ω^ℓ is defined as a collection of cells with respect to the tensor–product grid of level $\ell - 1$.

Example 1. Figures 2 and 3 show three subdomain hierarchies which will be used to demonstrate the performance of our algorithms and data structures:

- *rectangular* (refinement over rectangular–shaped regions);
- *linear* (refinement along a diagonal layer);



(a) $\Omega^0 \supseteq \Omega^1$ (b) $\Omega^0 \supseteq \Omega^1 \supseteq \Omega^2$ (c) $\Omega^0 \supseteq \Omega^1 \supseteq \Omega^2 \supseteq \Omega^3$

(d) $\Omega^0 \supseteq \Omega^1$ (e) $\Omega^0 \supseteq \Omega^1 \supseteq \Omega^2$ (f) $\Omega^0 \supseteq \Omega^1 \supseteq \Omega^2 \supseteq \Omega^3$

Fig. 3. Two nested sequences of subdomains — indicated as *linear* (top) and *curvilinear* (bottom) in Example 1. They satisfy relation (1) with respect to two (left), three (middle) and four (right) hierarchical levels.

– *curvilinear* (refinement along a curvilinear trajectory).

By starting with an initial tensor–product configuration at level 0, the tensor–product grid associated with level $\ell+1$ is obtained by subdividing any cell of the previous level into four parts. Each subdomain Ω^ℓ is then defined as a certain collection of cells with respect to the grid of level ℓ so that (1) is satisfied. Figure 2 illustrates an example of hierarchical refinement over rectangular–shape regions where the central area of the mesh is always refined up to the maximum level of detail. The other two subdomain hierarchies mentioned above are shown in Figure 3 up to four refinement levels so that $\Omega^0 \supseteq \ldots \supseteq \Omega^3$.

For each hierarchical level ℓ, with $\ell = 0, \ldots, N-1$, let $\mathcal{B}^\ell$ be the normalized B–spline basis of the spline space V^ℓ with respect to a certain degree (d, d) defined on corresponding nested knot sequences. We say that

$$\beta \in \mathcal{B}^\ell \text{ is active } \Leftrightarrow \operatorname{supp}^0 \beta \subseteq \Omega^\ell \wedge \operatorname{supp}^0 \beta \not\subseteq \Omega^{\ell+1},$$

where $\operatorname{supp}^0 \beta = \operatorname{supp} \beta \cap \Omega^0$ is a slightly modified support definition which makes local refinements possible also along the boundaries of Ω^0. A B–spline

$\beta \in \mathcal{B}^\ell$ is then *active* if it is completely contained in Ω^ℓ but not in $\Omega^{\ell+1}$, and *passive* otherwise.

We may assume the initial domain Ω^0 to be an axis-aligned box[3]. By denoting with k the number of knot spans of level 0 along the edges of Ω^0, which is assumed to be the same for both directions, we define a *characteristic matrix* X^ℓ of size $s^\ell \times s^\ell$, with $s^\ell = (2^\ell k + d)$, for $\ell = 0, \ldots, N-1$. These matrices collect the information about active/passive B–splines level by level, namely

$$\chi^\ell_{i,j} = \begin{cases} 1 & \text{if } \beta^\ell_{i,j} \text{ is active,} \\ 0 & \text{otherwise,} \end{cases}$$

where $\beta^\ell_{i,j}$ is a B–spline of level ℓ. The indices i, j are chosen such that exactly the B-splines $\beta^\ell_{i,j}$ with $i, j = 1, \ldots, s^\ell$ act on Ω^0.

Definition 1 ([11, 12], extended in [17]). *The hierarchical B–spline (HB–spline) basis $\mathcal{H}$ is defined as the set of all* active *B–splines defined over the tensor–product grid of each level,*

$$\mathcal{H} = \bigcup_{\ell=0,\ldots,N-1} \{\beta_{i,j} \in \mathcal{B}^\ell : \chi^\ell_{i,j} = 1\}.$$

Truncated hierarchical B–splines [5, 6] form a different basis for the same multilevel B–spline space. The key idea behind this alternative hierarchical construction is to properly exploit the *refinable* nature of the B–spline basis which allows to express a B–spline of level ℓ in terms of $(d+2)^2$ functions which belong to level $\ell + 1$ and of certain binomial coefficients scaled by a factor 2^{-d} with respect to any dimension. By using this subdivision rule, any function $\tau \in V^\ell$ can be represented according to a two–scale relation with respect to the basis $\mathcal{B}^{\ell+1}$ of $V^{\ell+1}$, namely

$$\tau = \sum_{\beta \in \mathcal{B}^{\ell+1}} c^{\ell+1}_\beta(\tau)\, \beta,$$

with certain coefficients $c^{\ell+1}_\beta(\tau) \in \mathbb{R}$. The *truncation* of $\tau \in V^\ell$ with respect to $\mathcal{B}^{\ell+1}$ and $\Omega^{\ell+1}$ is the function $\operatorname{trunc}^{\ell+1}\tau \in V^{\ell+1}$ defined as:

$$\operatorname{trunc}^{\ell+1}\tau = \sum_{\beta \in \mathcal{B}^{\ell+1}, \operatorname{supp}\beta \not\subseteq \Omega^{\ell+1}} c^{\ell+1}_\beta(\tau)\, \beta.$$

The overall truncation of a hierarchical B–spline $\beta \in \mathcal{B}^\ell \cap \mathcal{H}$ is defined by recursively applying the truncation with respect to the different levels,

$$\operatorname{trunc}\beta = \operatorname{trunc}^{N-1}(\operatorname{trunc}^{N-2} \ldots (\operatorname{trunc}^{\ell+1}\beta)).$$

By recursively discarding the contribution of active B–splines of subsequent levels from coarser B–splines, we obtain the definition of the truncated basis.

[3] Different shapes are easily identified at subsequent levels as shown in Figure 3.

Definition 2 ([5]). *The truncated hierarchical B-spline (THB–spline) basis $\mathcal{T}$ is defined by*

$$\mathcal{T} = \{\operatorname{trunc} \beta_{i,j}^{\ell} : \chi_{i,j}^{\ell} = 1, \ell = 0, \ldots, N-2\} \cup \{\beta_{i,j}^{N-1} : \chi_{i,j}^{N-1} = 1\}.$$

Truncated hierarchical B–spline are linearly independent, non-negative, form a partition of unity and preserve the nested nature of the spline spaces [5]. Moreover, the construction of THB–splines is strongly stable with respect to the supremum norm provided that the knot vectors satisfy certain reasonable assumptions — see [6] for more details.

In addition to the characteristic matrices $\{X^{\ell}\}_{\ell=0}^{N-1}$, we consider another sequence of matrices $\{C^{\ell}\}_{\ell=0}^{N-1}$ of the same size and with the same sparsity pattern, i.e. $\chi_{i,j}^{\ell} = 0$ implies $c_{i,j}^{\ell} = 0$. These matrices store the coefficients associated to the (active) basis functions in the representation of a spline function with respect to the truncated basis. The following simple algorithm performs the evaluation of a hierarchical spline function which is represented in terms of THB–splines.

Algorithm `EVAL(seqmat X, seqmat C, int D, int LMAX, float U,V)`

```
\\ seqmat X is the sequence of characteristic matrices, i.e., X[L] is the char-
    acteristic matrix of level L
\\ seqmat C is the sequence of coefficient matrices associated to the spline
    function f, i.e., C[L] is the coefficient matrix of level L
\\ int D is the degree in both directions
\\ int LMAX is the maximum refinement level N − 1
\\ float U,V are evaluation parameters

Identify the (D+1)×(D+1) sub–matrix M of C[0] which contains the coeffi-
    cients of those B–splines of level 0 that are non–zero at (U,V)
for L = 1 to LMAX do {
    Generate the matrix S by applying one step of B–spline subdivision to M
    Identify the (D+1)×(D+1) sub–matrix T of S which contains the coeffi-
        cients of those B–splines of level L that are non–zero at (U,V)
    for each pair of indices i,j in T do {
        if X[L](i,j) == 1 then T(i,j) = C[L](i,j) }
    M = T }
return the value f obtained by applying de Boor's algorithm to M
```

In this algorithm, the sub-matrices `M,S,` and `T` at a certain level are always accessed by global indices, i.e., indices with respect to the entire array of all tensor–product splines of that level. The following proposition clarifies the connection between the evaluation algorithm and the truncated hierarchical B–spline basis.

Theorem 1. *The value $f(u, v)$ computed by the algorithm is the value of a function represented in the THB–spline basis.*

This can be proved by applying the algorithm to Kronecker–type coefficient data (where exactly one coefficient is nonzero and equals 1).

The *cost* of the THB–spline evaluation algorithm `EVAL` is equal to $N-1$ times the application of the B–spline subdivision rule plus the cost due to the standard de Boor's algorithm. Consequently, it grows linearly with the number of levels and quadratically with the degree of the splines. It could be further reduced

– by starting the `for` loop at the minimum level of functions which are active at the given point (u, v), and
– by stopping it at the maximum level of functions which are active at this point.

With this modification, the computational costs grows linearly with the number of levels which are active at the given point. This number can be controlled by choosing a suitable refinement strategy.

The following sections discuss data structures and algorithms for manipulating and storing the subdomain hierarchy and for representing the characteristic matrices and coefficient matrices.

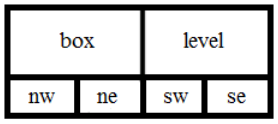
3 Representing and manipulating the subdomain hierarchy

The domain Ω^0 is now assumed to be a box consisting of $2^n \times 2^n$ cells of the coarsest tensor-product grid, where n is a non-negative integer, i.e., $k = 2^n$. This assumption is made in order to facilitate the use of a quadtree data structure. Moreover, in order to simplify the implementation, the edges of the coarsest tensor–product grid should have the length 2^{N-2}, where N is the number of levels. Under this assumption, all coordinates of bounding boxes in the algorithms presented below are integers.

3.1 The subdomain hierarchy quadtree

We represent the *entire* subdomain hierarchy by a *single* quadtree. Each node of the quadtree takes the form

```
struct qnode{
  aabb  box;
  int   level;
  *node nw;
  *node ne;
  *node sw;
  *node se; };
```



where the axis–aligned bounding box `aabb box` is characterized by coordinates of its upper left and lower right corner, `level` defines the highest level in which the box is completely contained and `nw`, `ne`, `sw`, `se` are pointers to the four children of the node. These children represent the northwestern, northeastern, southwestern and southeastern part of the box after the dyadic subdivision. All pointers to these children are set to null until the node is created during the insertion process, which is described by the `INSERTBOX` algorithm below.

Let $\Omega^\ell = \bigcup_i b_i^\ell$, where each b_i^ℓ is a collection of cells forming a rectangular box. During the creation of the quadtree which represents the subdomain hierarchy, for each level ℓ, we insert all boxes b_i^ℓ which define Ω^ℓ. The following recursive algorithm performs the insertion of a box b_i^ℓ into the quadtree:

Algorithm `INSERTBOX(box B, qnode Q, int L)`

```
\\ box B is the box which will be inserted
\\ qnode Q is the current node of the quadtree
\\ int L is the level
if B == Q.box then {
    Q.level = L
    visit all nodes in the subtree with root Q; if the level of a node is less
        than L then increase it to L }
else {
    for child in {Q.nw, Q.ne, Q.sw, Q.se} do {
        if child != null then {
            If B∩Q.box $\neq \emptyset$ then INSERTBOX(B∩Q.box, child, L) }
        else {
            create the box childbox of child
            if B∩childbox $\neq \emptyset$ then
              create the node child
              set child.box to childbox, child.level to Q.level and the
                  four children to null
              INSERTBOX(B∩childbox, child, L) } } }
```

After each box insertion we perform a cleaning step, visiting all sub–trees and deleting those where all nodes have the same level. This reduces the depth of the tree to a minimal value and optimizes the performance of all algorithms.

Example 2. To explain the `INSERTBOX` algorithm, we consider the subdomain hierarchy composed of three levels ($N = 3$), two of which (level 0 and 1) are initially present. This is shown in Figure 4, together with the related quadtree representation. The domain Ω^0 has $k = 16$ edges of length $2^{N-2} = 2$. The box $b = [16, 8] \times [24, 12]$ will be inserted at level 2 into the hierarchy. The cells with respect to the tensor–product grid of level 1 covered by b are depicted in red in Figure 4.

The execution of the algorithm is illustrated in Figure 5. At each step, we highlight the current node `Q` and the corresponding box in the subdomain hierarchy (Figure 5, right and left column, respectively). The insertion starts by considering the root of the tree, where the box b is compared with the axis–aligned bounding box stored in the root. Since these two boxes are not the same, the level of the root remains unchanged.

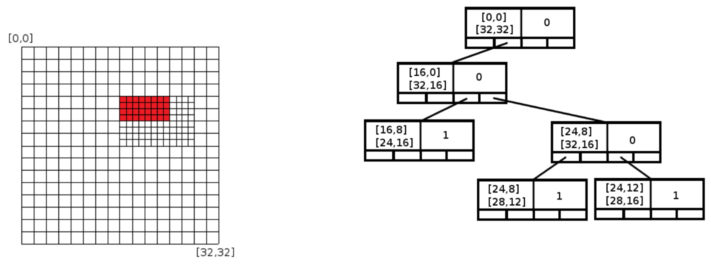


Fig. 4. Initial subdomain structure (left) and corresponding quadtree (right) which stores the boxes related to level 0 and 1 in the hierarchy. The box $b = [16, 8] \times [24, 12]$ (red) has to be inserted into the quadtree at level 2.

Subsequently, we have to identify which boxes between the ones stored in the four children of the root overlap with b, see Figure 5(a). In this case b is completely contained in the box represented by the `ne` child of the root. The recursive call of `INSERTBOX` is therefore applied to this child only. The situation in Figure 5(b) is similar to the previous case. After the split, the algorithm is recursively applied to the `sw` child.

In the third step shown in Figure 5(c) instead, the box b overlaps with the boxes related to two children (`nw` and `ne`) of the current node. Then, b is also subdivided and the recursion is called on both children.

Figure 5(d) shows the last step executed by the insertion of the box b. Two new nodes are created and inserted into the quadtree. Since these nodes coincide with the two parts of b, we set their level to 2. Clearly, the box to be inserted does not necessarily become a single node of the quadtree but it may be stored into several nodes.

3.2 Queries

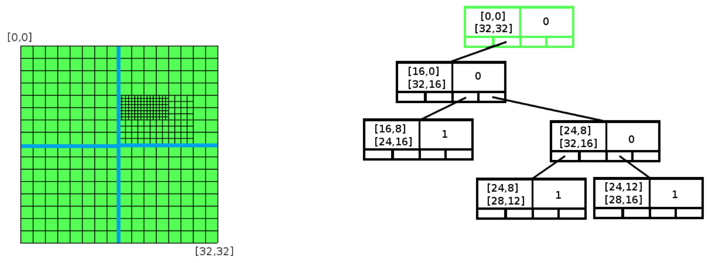
In order to create the characteristic matrices introduced in Section 2, we define three query functions on the quadtree. These queries allow to understand if all basis functions β of a certain hierarchical level whose support is contained in a given box b are active or passive.

Given a box b defined as a collection of cells with respect to the tensor–product grid of level ℓ, the first query (`QY1`), returns true if

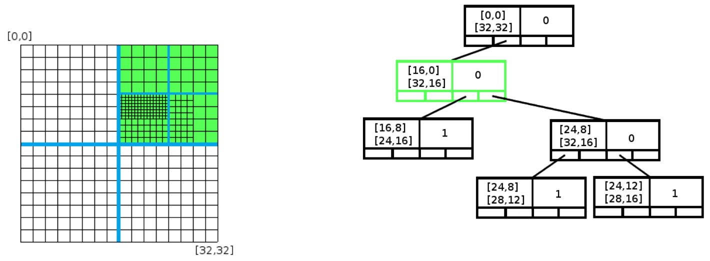
$$b \subseteq \Omega^{\ell} \quad \wedge \quad b \cap \Omega^{i} = \emptyset, \quad i > \ell. \tag{2}$$

Thus, if `QY1` returns true, then all the basis functions of level ℓ whose support is completely contained in the box b are *active*, i.e., they are present in the hierarchical spline basis.

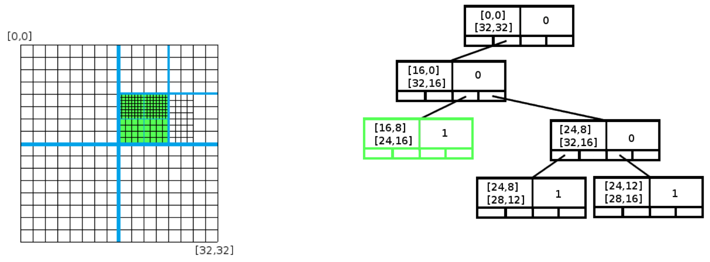
If the second query `QY2` returns true then all the basis functions of level ℓ whose support is contained in the box b are *passive*, i.e., they are not present in



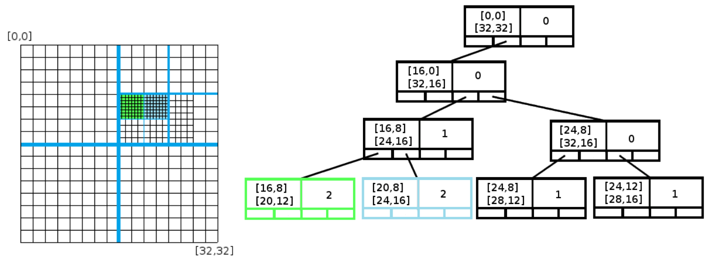
(a) first split (left) and quadtree (right)



(b) second split (left) and quadtree (right)

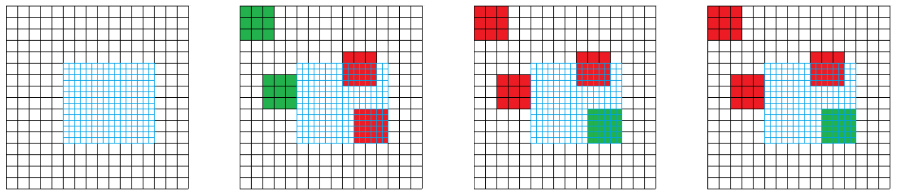


(c) third split (left) and quadtree (right)



(d) two new boxes (left) are inserted into the quadtree (right)

Fig. 5. Different steps performed by the `INSERTBOX` function to insert the box $b = [16, 8] \times [24, 12]$ into the subdomain hierarchy of Figure 4. The subsequent splits are shown on the subdomain hierarchy (blue lines on the left) with respect to the visit of the quadtree (right).



(a) level 0 and 1 (b) `QY1` for level 0 (c) `QY2` for level 0 (d) results of `QY3`

Fig. 6. Results of the three queries functions with respect to a subdomain hierarchy (a) with two levels. In case of `QY1` (b) and `QY2` (c), the green/red boxes correspond to a positive/negative answer. `QY3` (d) returns 1 for the green boxes and 0 for the red ones.

the hierarchical spline basis. This is characterized by the following condition:

$$b \cap \Omega^\ell = \emptyset \quad \vee \quad b \subseteq \Omega^\ell, \quad \text{for some } i > \ell. \tag{3}$$

The third query `QY3` returns the highest level ℓ with the property that Ω^ℓ contains the box b.

All the three queries can easily be implemented with the help of the quadtree structure described before. In particular, the structure of queries `QY1` and `QY2` is similar. We visit the quadtree until we find a leaf node or a node where the result of the query changes from to true to false. At that point, we can conclude the visit and return false. On the other hand, query `QY3` requires a complete visit of the quadtree.

Example 3. Figure 6(b–d) shows the results of the three queries with respect to the subdomain hierarchy composed of two levels (level 0 and 1) shown on Figure 6(a) for four sampled boxes of level 0. Figures 6(b) and (c) display the results of `QY1` and `QY2` for $\ell = 0$, respectively. The boxes in green correspond to a positive answer to the query, the red boxes to a negative one. Finally, Figure 6(d) shows the results for `QY3`. The green boxes correspond to answer 1 and the red ones to answer 0.

4 Characteristic matrices

The characteristic matrices identify the tensor–product basis functions which are present in the hierarchical basis.

4.1 Creating characteristic matrices

By using the quadtree structure defined in Section 3, we can generate the characteristic matrices introduced in Section 2 to represent and evaluate THB–splines. For the creation of these matrices we considered two different approaches:

- the *one–by–one approach* where we determine the entries of the characteristic matrices one by one by applying `QY3` to each single basis function;

– the *all–at–once approach* where we try to set as many values as possible in one single step. This requires a more sophisticated algorithm.

During the creation of the characteristic matrices by the all–at–once approach, we try to set many entries of the matrices at the same time. In order to do this, the query functions are initially called for boxes which cover the initial domain Ω^0. The SETMAT algorithm below creates the characteristic matrices for all subdomains in the subdomain hierarchy.

Algorithm SETMAT(qnode Q, seqmat X)

```
\\ qnode Q is the root of the quadtree which stores the subdomain hierarchy
\\ seqmat X is the sequence of characteristic matrices, i.e. X[L] is the characteristic matrix of level L
for all levels L do {
    Create the index set I for all functions of level L acting on $\Omega^0$. I is an
        axis-aligned box in index space.
    SETBOX(B,X[L]) }
```

SETMAT calls the algorithm SETBOX. When the answer active/passive cannot be given for the current call, the considered box is split into 4 disjoint axis–aligned bounding boxes and SETBOX function is recursively applied to them.

Algorithm SETBOX(aabbis I, mat XL)

```
\\ aabbis I is an axis-aligned box in index space
\\ mat XL is a characteristic matrix of level L
The level L is a global variable
Create the axis-aligned bounding box B covering all cells of level L which
    belong to the supports of functions with indices in I
if QY1(B, L) then {
    for all indices (i,j) in I do XL[i,j]=1 }
elseif QY2(B, L) then {
    for all indices (i,j) in I do XL[i,j]=0 }
elseif I is a single pair (i,j) then {
    k =  QY3(B, L)
    if k == L then XL[i,j]=1
    else XL[i,j]=0 }
else {
    split I into 4 disjoint axis-aligned bounding boxes I1-I4 by subdividing
        each edge (approximately) into halves in index space.
    Apply SETBOX to I1-I4 and XL }
```

Example 4. Figure 7 shows a subdomain hierarchy with two levels, consisting of a square Ω^0 and a subdomain Ω^1 in the southeastern corner, which is shown in blue. The four pictures (a–e) visualize several index sets I (shown by circles) and the associated boxes B (grey) which are considered by SETBOX when creating X^0 for biquadratic splines.

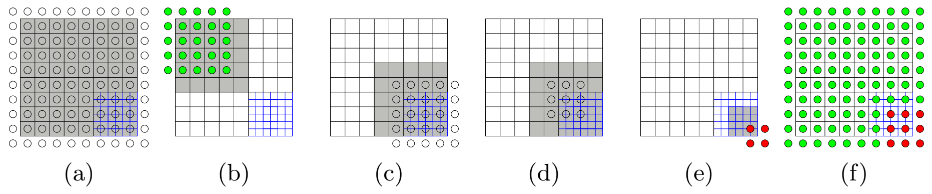


Fig. 7. A subdomain hierarchy with two levels and some of the boxes `I` in index space (shown as circles) along with the associated bounding boxes `B` in parameter space (grey) considered by `SETBOX` when creating the characteristic matrix X^0 for this subdomain hierarchy (a–e). Active (green) and passive (red) functions of level 0 (f).

Initially, `SETBOX` considers the entire set of basis functions (a) and concludes that it has to be subdivided. The northwestern subset is shown in (b). Query `QY1` returns 1, therefore the functions are all active; no subdivision is needed. The northeastern and southwestern subsets (not shown) are dealt with similarly. The southeastern subset (c), however, has to be subdivided. Considering its northwestern subset (d) does not lead to a conclusion again, needing another subdivision. The functions in this index set have to be analyzed one-by-one (not shown). The northeastern and southwestern subsets (not shown) are dealt with similarly. For the southeastern subset (e), however, query `QY2` returns 1, therefore the functions are all passive. Finally, the procedures arrives at the correct classification of basis functions of level 0 (f).

As Example 5 shows the all–at–once approach is not necessarily faster compared to the one–by–one mentioned at the beginning of this section. However, the approach becomes considerably faster with an increasing number of levels. This is demonstrated by the next example.

Example 5. Figure 8 compares the all–at–once setting with the one–by–one method. The number of queries called by the one-by-one approach is the same for the three hierarchical refinements in Figure 9 since it depends solely on the number of basis functions. This approach is faster for small numbers of basis functions, which typically correspond to a small numbers of hierarchical levels. On the other hand, the all–at–once approach becomes faster for higher numbers of basis functions in all the three considered cases since the number of queries grows sub–linearly with respect to the number of basis functions.

4.2 Using sparse data structures

The representation of THB–splines in terms of characteristic matrices allows a fast look–up during the evaluation process and a simple update of the values when the underlying subdomain hierarchy changes. The drawback of this representation is the rather large memory consumption, which can exceed the available physical memory even for relatively small meshes and low numbers of levels. Indeed, it grows exponentially with the number of levels.

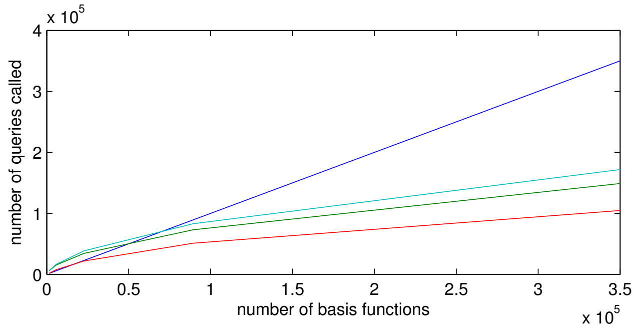


Fig. 8. The plot visualizes the number of queries needed to create the characteristic matrices for the three examples in Figure 2, 3 and 9. Compared to the all–at–once approach (cyan: linear, green: curvilinear, red: square-shaped refinement), the one–by–one approach (blue: same for all examples) is faster for small numbers of levels and basis functions, but it becomes slower for higher ones.

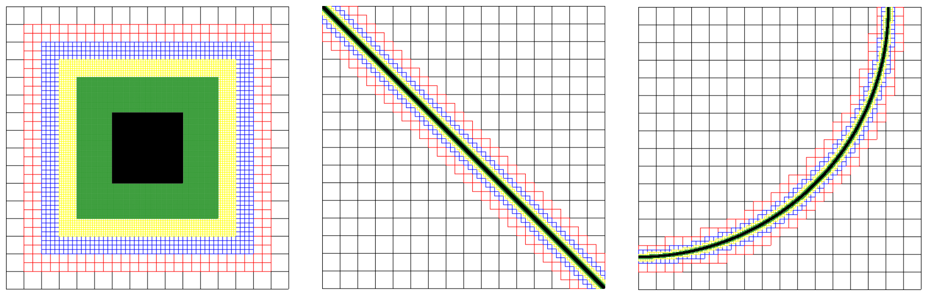


Fig. 9. The three subdomain hierarchies considered in Example 6: rectangular (left), linear (middle) and curvilinear (right) refinement, all with six levels.

This problem can be solved by using a suitable sparse matrix data structure. For our experiments, we chose the *compressed sparse column* (CSC) data structure. The nonzero elements (read first by column) are stored in a one–dimensional array. A second array stores the row indices corresponding to these values and a third one collects the indices into the first two arrays of the leading entry in each column [8].

As detailed in the next section, the CSC structure significantly reduces the memory consumption of our approach (see Example 6). In addition, the price paid for reducing the memory requirements is only a small increase of the computational time (see Examples 7 and 8).

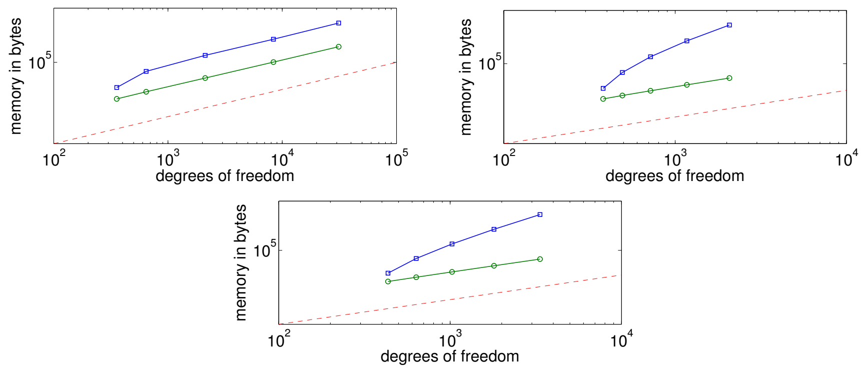


Fig. 10. Memory needed to represent the characteristic matrices without (blue) and with (green) the use of sparse data structures for different numbers of degrees of freedom related to the square (top left), the circle (top right) and the line refinement (bottom) refinement. The dashed red line has slope 1 and indicates linear growth.

5 Examples

We implemented the proposed algorithms and data structures in `C++`. For the manipulation of the characteristic matrices we used the sparse MATLAB representation in terms of the *compressed sparse column* approach mentioned at the end of the previous section.

Example 6. We compare the memory consumption of full characteristic matrices with the memory consumption of the matrices represented in the CSC structure for the three subdomain hierarchies in Figure 9 (rectangular, linear, and curvilinear).

The experimental results in Figure 10 show that the memory needed by the sparse matrix data structure is considerably smaller then the one related to the full matrix representation. Moreover, the memory consumption grows only *linearly* with the numbers of degrees of freedom (instead of exponentially with the number of levels). This is the optimal result that one can expect, since a coefficient for each active basis function needs to be stored anyway.

We observe a difference between the results related to the rectangular–shaped refinement with respect to the linear and curvilinear case. The reason is the different nature of the refinement procedure. In the linear and curvilinear case, the refined area is reduced at each new level and the coarser levels do not change (see Figure 3). In the rectangular case, the refined area of the highest level is constant and the size of lower level subdomains increases (see Figure 2). Thus, using the sparse data structure does not decrease the *order* of memory consumption in this case, since the number of degrees of freedom grows exponentially with the number of levels.

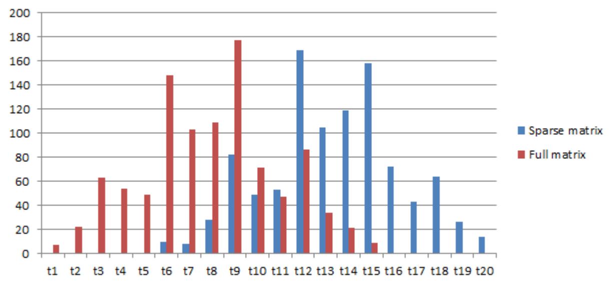


Fig. 11. The labels t1,. . . ,t20 on the horizontal axis represent uniform time intervals between minimal (0.153 ms) and maximal (0.195 ms) time needed by the evaluation algorithm. The vertical axis indicates the number of points whose evaluation time falls into these intervals.

The next example analyzes the influence of using the sparse data structures to the time needed to evaluate the multilevel spline functions using the algorithm `EVAL`.

Example 7. Figure 11 visualizes the distribution of the computation times needed to evaluate the multilevel spline function at 1000 points with (blue bars in the plot) and without (red bars in the plot) the use of sparse data structures for the linear refinement shown in Figure 9. Two facts can be observed:

– the evaluation time does not depend significantly on the location of the point with respect to the subdomain hierarchy;
– using the sparse data structure increases the evaluation time only by a very small amount.

Note that the evaluation times in this example vary between 0.153 and 0.195 milliseconds.

Finally we analyze the relation between evaluation time and the number of levels in the hierarchy.

Example 8. We consider the curvilinear refinement shown on the right of Figure 9. Figure 12 compares the evaluation times for 10,000 parameters obtained by using either the full or the sparse matrix representation. We may note that the computational time grows *linearly* with the increasing level of refinement for both representations, with a small overhead caused by using the sparse data structure. The values do not include the time necessary for creating the corresponding data structures, only the evaluation algorithm `EVAL` is considered.

6 Conclusion

We proposed an efficient implementation of data structures and related algorithms for the evaluation and manipulation of truncated hierarchical B–splines.

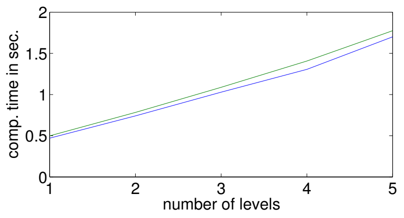


Fig. 12. Computational time needed to evaluate the multilevel spline function at 10,000 points for curvilinear refinement with various numbers of levels with (green) and without (blue) using the sparse data structure.

Several examples show the advantageous behavior of the data structures and algorithms in terms of memory overheads and computational costs. Indeed, the memory consumption grows only linearly with the number of degrees of freedom, but there is no significant increase of the time needed to evaluate the multilevel spline function.

The generalization of the proposed algorithms to handle the non–uniform case and multiple knots can be addressed by considering the subdomain hierarchy in index space rather than in the physical one. Interesting subjects for future research include the extension to multivariate splines and the identification of the refinement algorithm for THB–splines.

Acknowledgments. Gábor Kiss is supported by the Austrian Science Fund (FWF) through the Doctoral Program in Computational Mathematics (W1214, DK3). Carlotta Giannelli is a Marie Curie Postdoctoral Fellow within the 7th European Community Framework Programme under grant agreement n°272089 (PARADISE). Bert Jüttler has received support by the Austrian Science Fund (FWF) through the National Research Network Geometry + Simulation (S117).

References

1. Bornemann, P. B., and Cirak, F.: A subdivision–based implementation of the hierarchical B–spline finite element method, Comput. Methods Appl. Mech. Engrg., to appear (2012)
2. Deng, J., Chen, F. and Feng, Y.: Dimensions of spline spaces over T–meshes, J. Comput. Appl. Math. 194, 267–283 (2006)
3. Dokken, T., Lyche, T. and Pettersen, K. F.: Locally refinable splines over box–partitions, Tech. Rep. A22403, SINTEF (2012)
4. Forsey, D. R., and Bartels, R. H.: Hierarchical B–spline refinement, Comput. Graphics 22, 205–212 (1988)
5. Giannelli, C., Jüttler, B., and Speleers, H.: THB–splines: the truncated basis for hierarchical splines. Comput. Aided Geom. Design 29, 485–498 (2012)
6. Giannelli, C., Jüttler, B., and Speleers, H.: Strongly stable bases for adaptively refined multilevel spline spaces. Preprint (2012)

7. Giannelli, C., and Jüttler, B.: Bases and dimensions of bivariate hierarchical tensor–product splines, J. Comput. Appl. Math. 239,162–178 (2013)
8. Gilbert, J. R., Moler, C., and Schreiber, R.: Sparse matrices in MATLAB: design and implementation. SIAM J. Matrix Anal. Appl. 13, 333–356 (1992)
9. Gonzalez-Ochoa, C., and Peters, J., Localized–hierarchy surface splines (LeSS), In Proceedings of the 1999 symposium on Interactive 3D graphics, ACM, New York, NY, USA, 7–15 (1999)
10. Greiner G., and Hormann K.: Interpolating and approximating scattered 3D–Data with hierarchical tensor Product B–splines, In Méhauté, A. L., Rabut, C., Schumaker, L. L. (Eds.), Surface Fitting and Multiresolution Methods. In Innovations in Applied Mathematics. Vanderbilt University Press, Nashville, TN, 163–172 (1997)
11. Kraft, R.: Adaptive and linearly independent multilevel B–splines, in: Le Méhauté, A. and Rabut, C. and Schumaker, L. L. (Eds.), Surface Fitting and Multiresolution Methods, Vanderbilt University Press, Nashville, 209–218 (1997).
12. Kraft, R.: Adaptive und linear unabhängige Multilevel B–Splines und ihre Anwendungen, PhD Thesis, Universität Stuttgart (1998).
13. Lee, S., Wolberg, G., and Shin, S. Y.: Scattered data interpolation with multilevel B–splines, IEEE Trans. on Visualization and Computer Graphics 3, 228–244 (1997)
14. Schillinger, D., Dedè, L., Scott, M. A., Evans, J. A., Borden, M. J., Rank, E., and Hughes, T.J.R., An isogeometric design–through–analysis methodology based on adaptive hierarchical refinement of NURBS, immersed boundary methods, and T–spline CAD surfaces, Comput. Methods Appl. Mech. Engrg. 249–252, 116–150 (2012)
15. Schumaker, L. L. and Wang, L., Approximation power of polynomial splines on T–meshes, Comput. Aided Geom. Design 29, 599–612 (2012)
16. Sederberg, T. W., Zheng, J., Bakenov, A. and Nasri, A.: T–splines and T–NURCCS, ACM Trans. Graphics 22, 477–484 (2003)
17. Vuong, A.-V., Giannelli, C., Jüttler, B., and Simeon, B., A hierarchical approach to adaptive local refinement in isogeometric analysis, Comput. Methods Appl. Mech. Engrg. 200, 3554-3567 (2011)
18. Yvart A., and Hahmann S., Hierarchical triangular splines, ACM Trans. Graphics 24, 1374–1391 (2005)