Engineering with Computers
https://doi.org/10.1007/s00366-024-02002-1

ORIGINAL PAPER

Deal.t: an implementation of multivariate analysis suitable T-splines within the deal.ii framework

Sven Beuchler[1] · Robin Hiniborch[1] · Philipp Morgenstern[1,2]

Received: 17 July 2023 / Accepted: 7 May 2024
© The Author(s) 2024

Abstract
We present a numerical framework for solving partial differential equations within an isogeometric context using T-splines in two and three space dimensions. Within this paper, we explain the data structures used for the implementation of `deal.t` (`deal.II` with T-splines) and main differences when using `deal.t` in contrast to `deal.II`. The authors present numerical experiments with error-based refinement (2D) and a priori refinement (3D) for scalar-valued problems. A full tutorial is given in the appendix. Since the new framework is based on `deal.II`, T-splines may be applied to various different PDEs.

Keywords deal.II · T-splines · Isogeometric analysis · Finite element method · Adaptive mesh refinement

1 Introduction

This paper presents a new framework to solve partial differential equations (PDEs) using adaptive isogeometric analysis (IGA) based on T-splines, called `deal.t`. This package includes two and three-dimensional T-splines as explained in [1], and can be used to solve different PDEs of order two numerically with various (possibly mixed) boundary conditions on single patch domains.

IGA is the idea of directly using the shape functions from CAx modelling of physical domains as ansatz functions for the Finite Element Method (FEM). There are numerous software libraries available involving FEM-based solvers, e.g. `FEniCS`, see [2, 3], and `deal.II`, see [4], and libraries specifically designed for IGA-based FEM like e.g. `G+Smo`, see [5, 6], `PetIGA`, see [7]. However, the only publicly available implementations of T-splines for PDE discretizations currently known to the authors can be found in the Matlab package `igafem`, see [8], and the Python package `tIGAr`, see [9]. `tIGAr` uses a T-spline plugin for the commercial CAD software Rhinoceros 3D in order to run its calculations via T-splines. The underlying plugin has been discontinued from support and does not guarantee analysis-suitable T-splines.

IGA has been outlined in [10] where predominantly non-uniform rational B-splines (NURBS) were used as a tool to solve PDEs. Different concepts of refinement for NURBS have been introduced, i.e. h-refinement through knot insertion, p-refinement through order elevation, which are described in detail in [11], and k-refinement which simultaneously increases order and continuity.

However, local refinement techniques had already been introduced for B-splines, see e.g. *hierarchical B-splines* (HB-splines) in [12, 13], from which truncation methods of basis functions were developed in [14] (THB-splines) to further improve local refinement and properties of basis functions. T-splines have been introduced as an alternative to HB-splines in [15] for CAD as a strategy to directly refine the control mesh. They have been introduced as a mathematical tool in [16] and applied with promising results in [17], but it was eventually noticed that linear independence of basis functions is not always guaranteed, see [18]. Properties to guarantee linear independence have been introduced in [19] which introduced *analysis suitability* and *analysis suitable* (AS) T-splines. As an alternative to T-splines, *locally refined* (LR) B-splines were introduced in [20]. A first refinement algorithm for LR B-splines was introduced in [21] which

✉ Robin Hiniborch
hiniborch@ifam.uni-hannover.de

Sven Beuchler
beuchler@ifam.uni-hannover.de

Philipp Morgenstern
philipp.morgenstern@hannover-re.de

[1] Institute of Applied Mathematics, Leibniz University Hannover, Welfengarten 1, Hannover 30167, Germany

[2] Actuarial Models & Systems, Hannover Rück SE, Karl-Wiechert-Allee 50, Hannover 30625, Germany

Published online: 02 July 2024

showed promising results. However, linear independence is an issue for LR B-splines as well.

As T-splines were introduced, definitions were restricted to the two-dimensional case with degree three in both directions, and later AS T-splines of arbitrary degrees have been introduced in [22]. T-splines in arbitrary dimensions but odd polynomial degrees have been introduced in [23, 24] with proper refinement algorithms to retain analysis-suitability after refinement. Lastly, T-splines in arbitrary dimensions *and* arbitrary polynomial degrees were recently introduced in our previous work, see [1], allowing us to finally use T-splines in a framework for FEM, with arbitrary degree and local mesh refinement in three dimensions.

Recent advances in IGA can be found e.g. in [25], where a thorough comprehension of adaptive IGA is given for hierarchical splines with some applications. These results also include boundary element methods for an adaptive framework.

This paper is structured as follows. In Sect. 2, we discuss the prerequisites needed for the main parts. This includes notation and the definition of T-junctions in higher dimensions. We also briefly explain the concept of (geometric) analysis-suitability and AS T-splines with some examples. For a detailed guide through AS T-splines, we refer the reader to our previous work [1]. Section 2.2 is dedicated to the local refinement procedure explained in [23] and used in `deal.t`.

Further, we discuss in Sect. 3 possible meshes to serve as a base for implementing T-splines, with focus on the index mesh and the parametric mesh. We explain how both mesh classes are connected and how refinement on the parametric mesh affects the index mesh. Pseudocodes are given for relevant algorithms.

Section 4 explains main differences when using `deal.II` with `deal.t` compared to usual `deal.II` applications. It explains how to set up a proper triangulation, i.e. mesh, and the differences in assembly loops. `deal.t` is suited with a residual error estimator for Poisson-like problems, which is also explained in detail in Sect. 4.1. A full tutorial for readers unfamiliar with `deal.II` is given in Appendix C.

Numerical benchmark tests are discussed in Sect. 5. We consider two model problems based on a standard Poisson problem with Neumann and Dirichlet boundary conditions. The first problem is the result of the example given in Appendix C for various polynomial degrees, which is considered a pure benchmark test without singularities. The second problem is defined on the L-shape domain, given a solution with a corner singularity. A demonstration of mesh refinement in 3D is given in Fig. 2.

`deal.t` is based on the open source software `deal.II` and made available at [26], where the results of this work can be reproduced.

2 Prerequisites

In this section, we introduce the required mathematical notation. For details, see [1].

For any $x \in \mathbb{R}$ we will denote by $\lceil x \rceil$ the rounding of x to its next integer $n \in \mathbb{N}$ with $n \geq x$. Analogously, we denote by $\lfloor x \rfloor$ the rounding of x to its next integer $n \in \mathbb{N}$ with $n \leq x$.

We consider a box-shaped index domain $\widehat{\Omega} = \bigtimes_{k=1}^{d} (0, N_k)$, with $N_k \in \mathbb{N}$, for $k = 1, \ldots, d$, and an associated parametric domain $\Omega = \bigtimes_{k=1}^{d} (\xi_0^{(k)}, \xi_{N_k}^{(k)})$, with p_k-open knot vectors $\Xi^{(k)} = \{\xi_0^{(k)}, \ldots, \xi_{N_k}^{(k)}\}$, for polynomial degrees $p_k \in \mathbb{N}$. Let $\widehat{\mathcal{T}} = \mathcal{T}(\widehat{\Omega})$ be a mesh of $\widehat{\Omega}$, consisting of open axis-parallel boxes. We suppose that $\widehat{\Omega}$ is constructed via symmetric bisection from an initial tensor-product mesh with integer vertices, which is described in detail in [1, Algorithm 2.1]. Consequently, we suppose that for each vertex $\mathcal{V} = \{\mathrm{V}_1\} \times \cdots \times \{\mathrm{V}_d\}$ in the index mesh, the components are of the form $\mathrm{V}_k = a_k \cdot 2^{-b_k}$, with $a_k, b_k \in \mathbb{N}, k = 1, \ldots, d$. Therefore, all indices used below are from the set

$$\mathbb{N}^{(2)} := \{\tfrac{n}{2^b} \mid n, b \in \mathbb{N}\}. \tag{1}$$

In Sect. 3 we will see that the implementation is given on the parametric mesh $\mathcal{T} = \mathcal{T}(\Omega)$, however, relevant definitions are done on the index mesh $\widehat{\mathcal{T}}$. Throughout this paper, we will distinguish between the parametric mesh/elements $\mathrm{Q} \in \mathcal{T}$ and index mesh/elements $\widehat{\mathrm{Q}} \in \widehat{\mathcal{T}}$ with a hat in index domain notations which is dropped on the parametric domain. Where necessary, we introduce arguments to indicate which mesh we consider in this case.

For $k = 1, \ldots, d$, we denote the k-dimensional mesh entities of $\widehat{\mathcal{T}}$ by $\mathcal{H}^{(k)}(\widehat{\mathcal{T}})$, e.g. by $\mathcal{H}^{(0)}(\widehat{\mathcal{T}})$ the set of vertices, resp. nodes, by $\mathcal{H}^{(1)}(\widehat{\mathcal{T}})$ the set of one-dimensional edges without start and end point, and so on. The union of all d-dimensional element boundaries

$$\mathrm{Sk} = \bigcup_{\mathrm{Q} \in \mathcal{H}^{(d)}(\widehat{\mathcal{T}})} \partial \mathrm{Q} \tag{2}$$

is called the *skeleton* of $\widehat{\mathcal{T}}$.

For an index set $\kappa = \{\kappa_1, \ldots, \kappa_\ell\} \subset \{1, \ldots, d\}$ and a d-dimensional element $\mathrm{Q} = \mathrm{Q}_1 \times \cdots \times \mathrm{Q}_d \in \mathcal{H}^{(d)}(\widehat{\mathcal{T}})$ we denote the $(d - \ell)$-dimensional, κ-orthogonal interfaces by $\mathrm{H}^{(\kappa)}(\mathrm{Q})$, e.g.

$$\mathrm{H}^{(\kappa)}(\mathrm{Q}) := \left\{ \mathrm{E} = \bigtimes_{k=1}^{d} \mathrm{E}_k \;\middle|\; \begin{array}{l} \mathrm{E}_j \subset \partial \mathrm{Q}_j \text{ for } j \in \kappa, \\ \mathrm{E}_j = \mathrm{Q}_j \text{ for } j \notin \kappa \end{array} \right\}, \tag{3}$$

For polynomial degrees $\mathbf{p} = (p_1, \ldots, p_d) \in \mathbb{N}^d$, we split the index domain $\widehat{\Omega}$ into an *active region* $\mathrm{AR}_{\mathbf{p}}$ and a *frame region* $\mathrm{FR}_{\mathbf{p}}$, with

$$\mathrm{AR}_{\mathbf{p}} := \bigtimes_{k=1}^{d} \left[\lfloor \tfrac{p_k+1}{2} \rfloor, N_k - \lfloor \tfrac{p_k+1}{2} \rfloor \right] \tag{4}$$

$$\mathrm{FR}_{\mathbf{p}} := \widehat{\Omega} \setminus \overline{\mathrm{AR}_{\mathbf{p}}}. \tag{5}$$

We continue to explain what a T-junction in higher dimensions is.

Definition 1 (T-junctions) We call an interface $\mathtt{T} \in \mathcal{H}^{(d-2)}(\widehat{\mathcal{T}})$ with $\mathtt{T} \not\subseteq \partial\widehat{\Omega}$ a *T-junction* if it is in the boundary of a cell $\mathtt{Q} = \mathtt{Q}_1 \times \cdots \times \mathtt{Q}_d \in \mathcal{T}$ without being connected to any of its vertices, $\mathtt{T} \subset \partial\mathtt{Q}$, $\overline{\mathtt{T}} \cap \partial\mathtt{Q}_1 \times \cdots \times \partial\mathtt{Q}_d = \varnothing$. We then call $\mathtt{Q}$ the *associated cell* of $\mathtt{T}$ and write $\mathtt{Q} = \mathrm{ascell}(\mathtt{T})$. Since $\mathtt{T} = \mathtt{T}_1 \times \cdots \times \mathtt{T}_d \in \mathcal{H}^{(d-2)}(\widehat{\mathcal{T}})$, there are two unique and distinct directions $i, j \in \{1, \ldots, d\}$ such that $\mathtt{T}_i, \mathtt{T}_j$ are singletons, $\mathtt{T} \in \mathrm{H}^{(\{i,j\})}(\widehat{\mathcal{T}})$, $\mathtt{T}_i \subsetneq \mathtt{Q}_i$ and $\mathtt{T}_j \subsetneq \partial\mathtt{Q}_j$. We call i the *orthogonal direction* and j the *pointing direction* of $\mathtt{T}$, and write $\mathrm{odir}(\mathtt{T}) = i$, $\mathrm{pdir}(\mathtt{T}) = j$.

From [1] we have uniqueness of the associated cell for each T-junction.

Before we jump into definitions of anchors and index vectors, we first define a criterion for meshes that may be taken into consideration for further computations.

Definition 2 (Admissible meshes) We define the slice

$$\mathrm{S}_k(n) := \{(x_1, \ldots, x_d) \in \widehat{\mathcal{T}} \mid x_k = n\}, \tag{6}$$

for $k = 1, \ldots, d$ and $n = 0, \ldots, N_k$, and the k-th frame region

$$\mathrm{FR}_{\mathbf{p}}^{(k)} := \left\{ x \in \overline{\widehat{\Omega}} \mid x_k \in \left[0, \lfloor \tfrac{p_k+1}{2} \rfloor\right] \cup \left[N_k - \lfloor \tfrac{p_k+1}{2} \rfloor, N_k\right] \right\}. \tag{7}$$

A T-mesh $\widehat{\mathcal{T}}$ is called *admissible*, if for $k = 1, \ldots, d$, there is no T-junction $\mathtt{T}$ with $\mathrm{odir}(\mathtt{T}) = k$ or $\mathrm{pdir}(\mathtt{T}) = k$ in the k-th frame region, and

$$\mathrm{S}_k(n) \subseteq \mathrm{Sk} \quad \text{for} \quad \begin{array}{l} n \in \left[0, \lfloor \tfrac{p_k+1}{2} \rfloor\right] \cap \mathbb{N} \text{ and} \\ n \in \left[N_k - \lfloor \tfrac{p_k+1}{2} \rfloor, N_k\right] \cap \mathbb{N}. \end{array} \tag{8}$$

For the rest of this paper, the index mesh $\widehat{\mathcal{T}}$ is assumed to be admissible whenever used.

2.1 Multivariate T-splines and analysis-suitability

Anchors are a subset of mesh entities that have a one-to-one correspondence to the set of T-splines. For higher dimension and arbitrary degree $\mathbf{p} = (p_1, \ldots, p_d)$, they have been introduced in [1] as

$$\mathcal{A}_{\mathbf{p}} := \{\mathbf{A} \in \mathrm{H}^{(\kappa)}(\widehat{\mathcal{T}}) \mid \mathbf{A} \subset \mathrm{AR}_{\mathbf{p}}\}. \tag{9}$$

with $\kappa = \{\ell \in \{1, \ldots, d\} \mid p_\ell \text{ odd}\}$. We associate to each anchor and each axis direction $k = 1, \ldots, d$ a non-decreasing knot vector of $p_k + 2$ indices from $\mathbb{N}^{(2)}$. In [22, 27, 28], these index vectors (in 2D) are constructed via ray-tracing the anchor through the mesh along the k-th direction and choosing the $p_k + 2$ consecutive indices centered around the k-th component $\mathbf{A}_k$ of $\mathbf{A}$. This is generalized in [1] to arbitrary dimensions, i.e. for any mesh entity $\mathtt{E} = \mathtt{E}_1 \times \cdots \times \mathtt{E}_d$ and $k \in \{1, \ldots, d\}$, we define the projection $P_{k,n}(\mathtt{E}) = \mathtt{E}|_{\mathtt{E}_k = \{n\}}$ of $\mathtt{E}$ on the slice $\mathrm{S}_k(n)$, and the *global knot vector*

$$\mathcal{I}_k(\mathtt{E}) := \left(n \in \mathbb{N}^{(2)} \mid P_{k,n}(\mathtt{E}) \subset \mathrm{Sk}_k \right), \tag{10}$$

with entries in ascending order. The *local knot vector* $\mathrm{v}_k(\mathbf{A})$ for an anchor $\mathbf{A} = \mathbf{A}_1 \times \cdots \times \mathbf{A}_d$ is given by the $p_k + 2$ consecutive indices $\ell_0, \ldots, \ell_{p_k+1} \in \mathcal{I}_k(\mathbf{A})$, such that $\ell_k = \inf \mathbf{A}_k$ for $k = \lfloor \frac{p_k+1}{2} \rfloor$. If p_k is odd, the singleton $\mathbf{A}_k$ contains the middle entry of $\mathrm{v}_k(\mathbf{A})$, and if p_k is even, the two middle entries of $\mathrm{v}_k(\mathbf{A})$ are the boundary values of $\mathbf{A}_k$.

An example is given in Fig. 1. Depicted is the construction principle using ray-tracing to generate the corresponding local index vectors. For $\mathbf{A}^{(1)}$ on the top, we obtain $\mathrm{v}_1(\mathbf{A}^{(1)}) = (\bar{m} - 2, \bar{m} - 1, \bar{m}, \bar{m} + 1, \bar{m} + 2)$. In contrast, for $\mathbf{A}^{(2)}$, the projection $P_{1,\bar{m}+3}(\mathbf{A}^{(2)}) \not\subset \mathrm{Sk}$ onto the slice at $(\bar{m} + 3)$ is not a part of the skeleton of the mesh. The same applies to the index $\bar{m} + 1$, which yields $\mathrm{v}_1(\mathbf{A}^{(1)}) = (\bar{m} - 2, \bar{m} - 1, \bar{m}, \bar{m} + 2, \bar{m} + 4)$. Note that p_1 is odd and that $\mathbf{A}_1^{(1)} = \{\bar{m}\} = \mathbf{A}_1^{(2)}$ is the middle element of the respective local index vectors.

On the lower part of Fig. 1, we get as before $\mathrm{v}_1(\mathbf{A}^{(1)}) = (m_1 - 2, m_1 - 1, m_1, m_2, m_2 + 1, m_2 + 2)$. However, the projection of $\mathbf{A}^{(2)}$ onto the slice at m_2 is not part of the skeleton, which yields $\mathrm{v}_1(\mathbf{A}^{(2)}) = (m_1 - 2, m_1 - 1, m_1, m_2 + 1, m_2 + 2, m_2 + 3)$. Note that p_1 is even and that $\mathbf{A}_1^{(1)} = (m_1, m_2)$ is the middle section of the local index vector $\mathrm{v}_1(\mathbf{A}^{(1)})$.

Using anchors and their local index vectors, we can define a multivariate T-spline associated to an anchor.

Definition 3 (T-spline) For $p_k \in \mathbb{N}$, we denote by $T_{\mathrm{v}_k(\mathbf{A})} : \Omega \to \mathbb{R}$ the univariate B-spline function of degree p_k that is returned by the Cox-deBoor recursion with knot vector $\xi_{\mathrm{v}_k(\mathbf{A})} = (\xi^{(k)}_{\ell_0}, \ldots, \xi^{(k)}_{\ell_{p_k+1}})$, see e.g. [11]. We assume that $\xi^{(k)}_{\ell_0} < \xi^{(k)}_{\ell_{p_k+1}}$ is always fulfilled. The T-spline function associated with the anchor $\mathbf{A}$ is defined as

$$T_{\mathbf{A}}(\zeta_1, \ldots, \zeta_d) := \prod_{k=1}^{d} T_{\mathrm{v}_k(\mathbf{A})}(\zeta_k), \quad \text{for } (\zeta_1, \ldots, \zeta_d) \in \Omega, \tag{11}$$

and the corresponding T-spline space is given by $\mathcal{S}_{\mathcal{T},\mathbf{p}}(\Omega) = \mathrm{span}\{T_{\mathbf{A}} \mid \mathbf{A} \in \mathcal{A}_{\mathbf{p}}\}$. The index support of $T_{\mathbf{A}}$ will be denoted by $\mathrm{supp}_{\widehat{\Omega}}\, T_{\mathbf{A}} = \bigtimes_{k=1}^{d} \mathrm{conv}\, \mathrm{v}_k(\mathbf{A})$, where $\mathrm{conv}\, \mathrm{v}_k(\mathbf{A}) =$

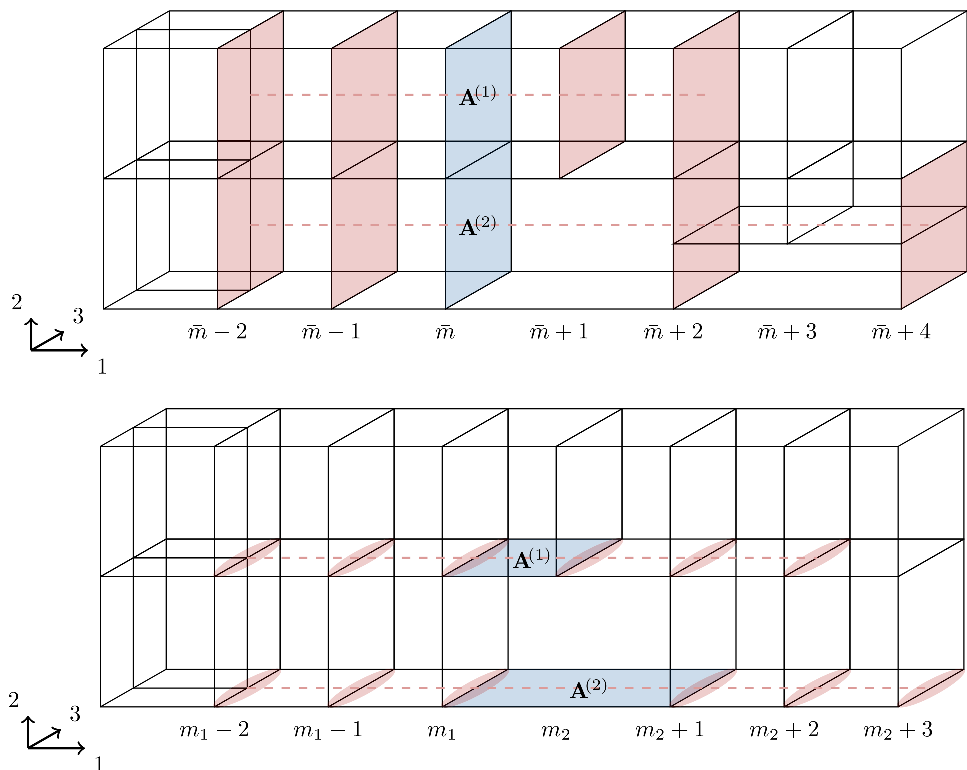


Fig. 1 Demonstration for the construction of local index vectors $\mathrm{v}_1(\cdot)$ of marked anchors for polynomial degrees $\mathbf{p}=(3,2,2)$ (top) and $\mathbf{p}=(4,1,2)$ (bottom)

$\mathrm{conv}(\ell_0,\ldots,\ell_{p_k+1}) = [\ell_0,\ell_{p_k+1}]$ is the closed interval from the first to the last entry of $\mathrm{v}_k(\mathbf{A})$.

For the concept of analysis-suitability, we use a geometric concept based on the concepts from [22, 27, 28]. Note that there are different versions of analysis-suitability, see [1] for details.

Definition 4 (Geometric analysis-suitability) Let $\mathtt{T}$ be a T-junction with $\mathtt{Q} = \mathrm{ascell}(\mathtt{T})$, $i = \mathrm{odir}(\mathtt{T})$ and $j = \mathrm{pdir}(\mathtt{T})$. We then define local knot vectors as follows.

1. For $k=j$, we define $\mathrm{v}_j(\mathtt{T}) = (\ell_0,\ldots,\ell_{p_j})$ as the vector of (p_j+1) consecutive indices from $\mathcal{I}_j(\mathtt{T})$, such that

$$\begin{aligned}&\{\ell_{p_j/2}\} = \mathtt{T}_j, && \text{if } p_j \text{ is even,}\\ &\left.\begin{aligned}\ell_{\lfloor p_j/2\rfloor} &= \inf \mathtt{Q}_j,\\ \ell_{\lceil p_j/2\rceil} &= \sup \mathtt{Q}_j,\end{aligned}\right\} && \text{if } p_j \text{ is odd.}\end{aligned} \tag{12}$$

2. For $k=i$, the local knot vector is the singleton $\mathrm{v}_i(\mathtt{T}) = \mathtt{T}_i$.
3. For $k \notin \{i,j\}$ we define $\mathrm{v}_k(\mathtt{T}) = (\ell_0,\ldots,\ell_{p_k+1+c_k})$, where $c_k = p_k \bmod 2$, as the vector of (p_k+2+c_k) consecutive indices from $\mathcal{I}_k(\mathtt{T})$, such that

$$\mathtt{T}_k = (\ell_{\lceil p_k/2\rceil}, \ell_{\lceil p_k/2\rceil+1}). \tag{13}$$

This means that the local knot vector has p_k+3 elements if p_k is odd and p_k+2 if p_k is even, and $\mathtt{T}_k$ is centred within these elements, cf.@ the definition of local knot vectors for anchors.

We then call

$$\mathrm{GTJ}_i(\mathtt{T}) := \bigtimes_{k=1}^{d} \mathrm{conv}(\mathrm{v}_k(\mathtt{T})) \tag{14}$$

the *geometric* T*-junction extension* (GTJ) of $\mathtt{T}$, and we say that it is an i-orthogonal extension in j-direction. Note, that $\mathrm{GTJ}_i(\mathtt{T}) \not\subset \mathrm{Sk}_i$.

A mesh $\mathcal{T}$ is *strongly geometrically analysis-suitable* (SGAS), if for any two T-junctions $\mathtt{T}_1, \mathtt{T}_2$ with orthogonal directions $i_1 = \mathrm{odir}(\mathtt{T}_1) \neq \mathrm{odir}(\mathtt{T}_2) = i_2$ there is

$$\mathrm{GTJ}_{i_1}(\mathtt{T}_1) \cap \mathrm{GTJ}_{i_2}(\mathtt{T}_2) = \varnothing. \tag{15}$$

We call $\mathcal{T}$ *weakly geometrically analysis-suitable* (WGAS), if (15) holds for any two T-junctions $\mathtt{T}_1, \mathtt{T}_2$ with orthogonal directions $\mathrm{odir}(\mathtt{T}_1) \neq \mathrm{odir}(\mathtt{T}_2)$ and pointing directions $\mathrm{pdir}(\mathtt{T}_1) \neq \mathrm{pdir}(\mathtt{T}_2)$.

We will omit the dependency of the orthogonal direction when it is clear from the context, e.g. write $\mathrm{GTJ}(\mathtt{T}) \equiv \mathrm{GTJ}_i(\mathtt{T})$, for $\mathrm{odir}(\mathtt{T}) = i$.

It was proven in [1] that SGAS is sufficient for linear independence of the generated spline space, which makes them suitable for application in a finite element setting.

2.2 Local refinement procedure

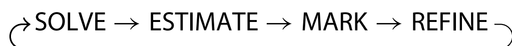
Algorithm 1 Coarse neighborhood algorithm from the `deal.t` implementation

```
get_coarse_neighborhood(𝒯, ℳ){
    𝒩 = {}
    for (Q1 : ℳ){
        𝒩 = 𝒩 ∪ {Q1}
        es = size(Q) // edge sizes from (16)
        bdry = es∘(p+3/2) // hadamard-product
            from (20)
        for (Q2 : 𝒯) {
            if (ℓ(Q2)!=ℓ(Q1)−1 || Q2∈𝒩 )
                continue;

            // Compute distance between cells
                from (19)
            dist = Dist(Q1,Q2)

        // Check if it is in the open
            environment from (21). If so, it
            also belongs to the neighborhood.
            if (dist < bdry)
                𝒩 = 𝒩 ∪ {Q2}
        }
    }
    return 𝒩;
}// get_coarse_neighborhood
```

Algorithm 2 Mesh refinement for marked cells $\mathcal{M}$ of a parametric mesh $\mathcal{T}$.

```
refine_mesh(𝒯, ℳ){
    // Find coarse neighborhood of marked cells
         from Algorithm 1
    𝒩 = get_coarse_neighborhood(𝒯, ℳ)
    ℳ = ℳ ∪ 𝒩
    n_cells_nh = 0;
    // run recursion for coarse neighborhood
    while (𝒩.size() != n_cells_nh){
        n_cells_nh = 𝒩.size()
        𝒩 = get_coarse_neighborhood(𝒯, ℳ)
    }

    for (Q : ℳ){
        𝒯 = 𝒯 \ {Q} ∪ {subdivQ}
    }
    return 𝒯 = 𝒯
} // refine_mesh
```

We now continue to explain the refinement strategy for a given T-mesh, intended to be applied in an adaptive Galerkin scheme with the steps solve, estimate, mark and refine as follows.



In Sect. 3 we will explain in detail how the parametric mesh $\mathcal{T} = \mathcal{T}(\Omega)$ and the index mesh $\widehat{\mathcal{T}}$ are connected and what we have to consider during the implementation. To this extent, we will thus only explain the refinement process on the parametric mesh $\mathcal{T}$. The definitions below strictly follow [23, 24].

As for the index mesh, we consider a box-shaped parametric mesh Ω consisting of axis-aligned open boxes that are generated via symmetric box bisections from an initial tensor-product structure. Note that a tensor-product structure is strongly geometric analysis-suitable due to the absence of T-junctions. We denote the initial tensor-product mesh by $\mathcal{T}^{(0)}$ and every consecutive mesh by $\mathcal{T}^{(n)}$, $n = 1, \ldots$

Let $\mathrm{Q} \in \mathcal{T}^{(n)}$ be some cell. We define its *level* by the index $m \leq n$, s.t. Q is an element of the m-th uniform refinement of $\mathcal{T}^{(0)}$, and write $\ell(\mathrm{Q}) = m$. Its *vector-valued size* is defined to be the component-wise length of its edges, i.e.

$$\textsc{size}(\mathrm{Q}) := \big(\sup \mathrm{Q}_i - \inf \mathrm{Q}_i\big)_{i=1}^{d}. \tag{16}$$

The *level-dependent* subdivision of a cell $\mathrm{Q} \in \mathcal{T}^{(n)}$ is defined by the $k_{\ell(\mathrm{Q})}$-orthogonal bisection of Q where $k_{\ell(\mathrm{Q})} = 1 + (\ell(\mathrm{Q}) \bmod d)$, denoted by $\textsc{subdiv}(\mathrm{Q}, k_{\ell(\mathrm{Q})})$.

For a point $z \in \Omega$, we define the *vector-valued distance* between z and Q as

$$\textsc{Dist}(\mathrm{Q}, z) := \textsc{abs}\big(\textsc{mid}(\mathrm{Q}) - z\big) \in \mathbb{R}^d \tag{17}$$

$$\text{with } \textsc{abs}(z) := \big(|z_1|, \ldots, |z_d|\big) \in \mathbb{R}^d, \tag{18}$$

and $\textsc{mid}(\mathrm{Q})$ is defined as the midpoint of Q. For two cells $\mathrm{Q}^{(1)}$ and $\mathrm{Q}^{(2)}$, we define its distance as the vector-valued distance of its cells.

$$\textsc{Dist}(\mathrm{Q}^{(1)}, \mathrm{Q}^{(2)}) := \textsc{abs}\big(\textsc{mid}(\mathrm{Q}^{(1)}) - \textsc{mid}(\mathrm{Q}^{(2)})\big). \tag{19}$$

For two vectors $x, y \in \Omega$, we denote by $x \circ y$ the component-wise product, i.e.

$$x \circ y := (x_1 y_1, \ldots, x_d y_d). \tag{20}$$

With these definitions, we define the *open environment* $U(\mathrm{Q})$ of a cell $\mathrm{Q} \in \mathcal{T}^{(n)}$ by

$$U(\mathrm{Q}) := \{z \in \mathbb{R}^d \mid \textsc{Dist}(\mathrm{Q}, z) < \textsc{size}(\mathrm{Q}) \circ (\mathbf{p} + \tfrac{3}{2})\}, \tag{21}$$

and its *coarse neighbourhood* $\mathcal{N}(\mathcal{T}^{(n)}, \mathrm{Q})$ of a cell $\mathrm{Q} \in \mathcal{T}^{(n)}$ in the mesh $\mathcal{T}^{(n)}$ by

$$\mathcal{N}(\mathcal{T}^{(n)}, \mathrm{Q}) := \left\{\mathrm{Q}' \in \mathcal{T}^{(n)} \;\middle|\; \begin{array}{l} \mathrm{Q}' \cap U(\mathrm{Q}) \neq \varnothing \\ \ell(\mathrm{Q}') = \ell(\mathrm{Q}) - 1 \end{array}\right\}. \tag{22}$$

Let $\mathcal{M} \subset \mathcal{T}^{(n)}$ be a set of elements marked for refinement. Its *closure* is then defined recursively via

1. Set $\mathcal{N}^{(0)}(\mathcal{T}^{(n)}, \mathcal{M}) := \bigcup_{\mathrm{Q} \in \mathcal{M}} \mathcal{N}(\mathcal{T}^{(n)}, \mathrm{Q})$ and $\mathcal{M}^{(0)} = \mathcal{M}$,
2. for $j = 1, 2, \ldots,$ define

   $$\mathcal{M}^{(j)} := \mathcal{N}^{(j-1)}(\mathcal{T}^{(n)}, \mathcal{M}),$$

3. break recursion if $\mathcal{M}^{(j)} = \mathcal{M}^{(j-1)}$,
4. define $\texttt{closure}(\mathcal{T}^{(n)}, \mathcal{M}) := \mathcal{M}^{(j)}$.

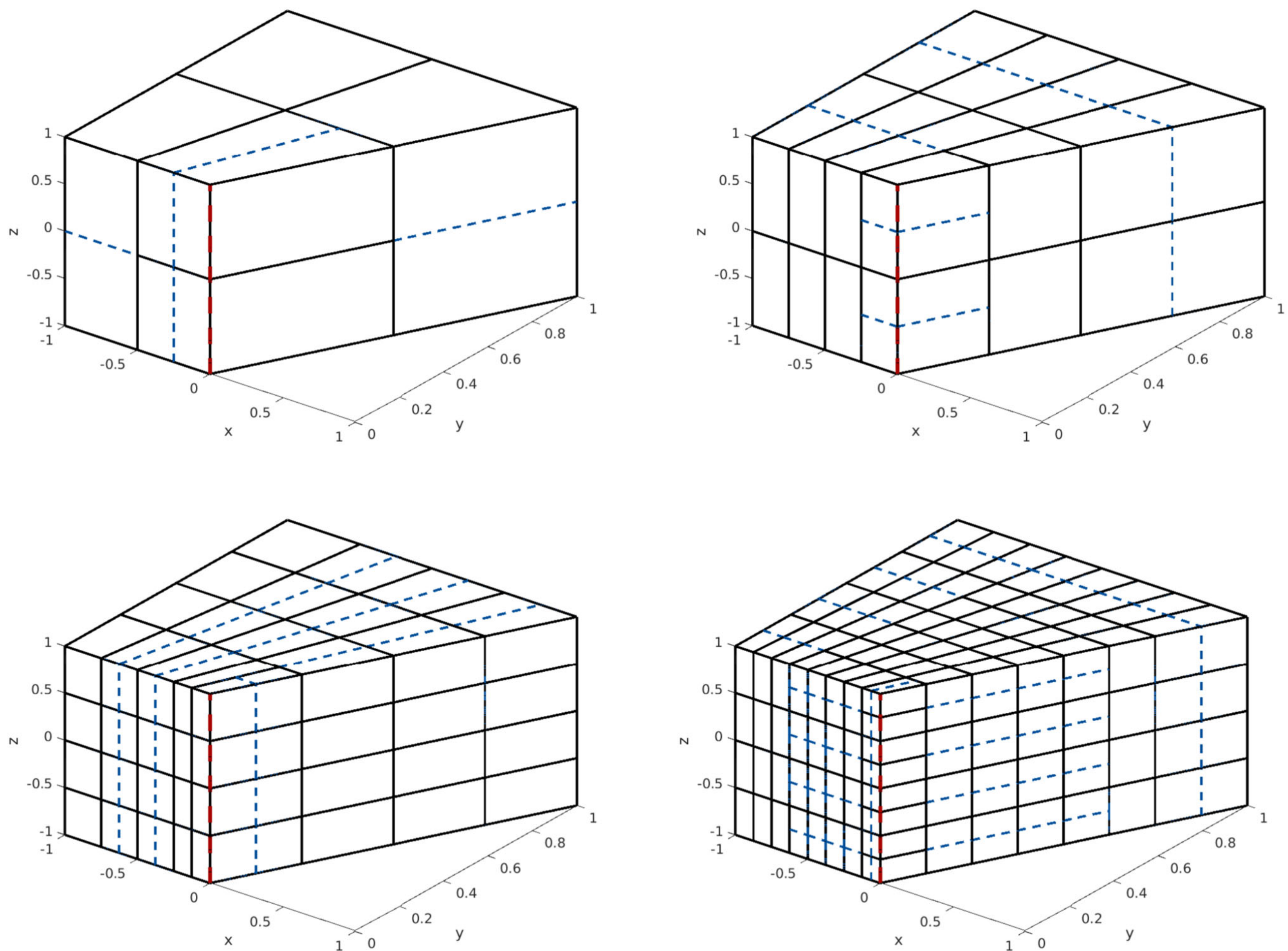


Fig. 2 Cross-sections of a 3D L-Shape domain at different refinement levels. Dashed, blue lines indicate the next level after marking cells adjacent to the thick, dashed, red line $\mathtt{L} = \{0\} \times \{0\} \times [-1, 1]$. The levels given are 3 and 4 (top left), 5 and 6 (top right), 7 and 8 (bottom left), and 9 and 10 (bottom right)

Note that this recursion will terminate in the worst case if $\mathcal{N}^{(j)}(\mathcal{T}^{(n)}, \mathcal{M}) = \mathcal{T}^{(n)}$. We then define the refined mesh $\mathcal{T}^{(n+1)}$ by

$$\mathcal{T}^{(n+1)} := \mathcal{T}^{(n)} \setminus \mathtt{closure}(\mathcal{T}^{(n)}, \mathcal{M}) \cup \bigcup_{\mathtt{Q} \in \mathtt{closure}(\mathcal{T}^{(n)}, \mathcal{M})} \mathrm{SUBDIV}(\mathtt{Q}). \tag{23}$$

From [23] we know that if $\mathcal{T}^{(n)}$ is analysis-suitable, $\mathcal{T}^{(n+1)}$ is also analysis-suitable. Note that the analysis-suitability used in [23] is abstract and thus different from the introduced geometric analysis-suitability above. However, in [1] it was shown that the geometric one yields also the abstract one. Thus, we only use geometric analysis-suitability to find Bézier elements. A C++-style pseudocode for the construction of the coarse neighbourhood is given in Algorithm 1. Note that this algorithm has to be called recursively in order to get the complete coarse neighbourhood, see Algorithm 2.

An example is given in Fig. 2, where the 3D L-shape domain is considered. Cells that are adjacent to the red line $\mathtt{L} = \{0\} \times \{0\} \times [-1, 1]$ are marked for refinement in each iteration. Given a problem on the whole 3D L-Shape domain that is symmetric along the diagonal

$$\mathtt{D} = \{(x, y, z) \in \mathbb{R}^3 \mid x = y,\ x, y \geq 0, -1 \leq z \leq 1\} \tag{24}$$

this half domain is sufficient for computations. For simplicity, we have depicted only a 3D view of a cross-section along the diagonal D. The blue dashed lines denote the next level of refinement. This example thus also demonstrates the definition of the coarse neighbourhood from Algorithm 1.

After refinement, the T-splines, resp. local knot vectors have to be updated. It was shown in [1] that under certain assumptions, local knot vectors may be inherited from *parent* anchors. The result reads as follows.

Lemma 1 *Let* $\widehat{\mathcal{T}} \in \mathrm{WGAS}$ *be the associated index mesh to the parametric mesh* $\mathcal{T}$. *Assume that each cell* $\widehat{\mathtt{Q}} \in \widehat{\mathcal{T}}$ *has active*

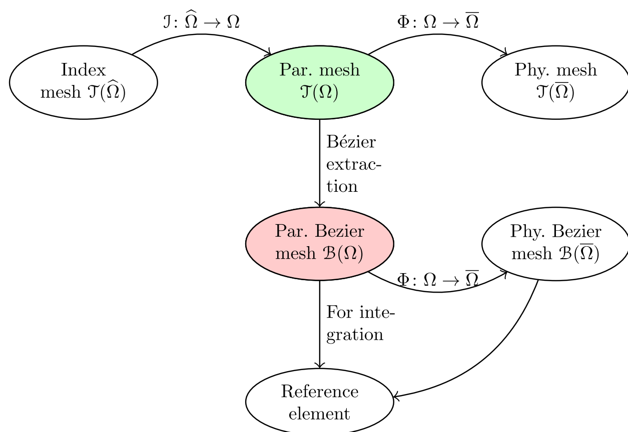


Fig. 3 An overview of possible mesh types that can be used for an implementation. This implementation is focused on the highlighted mesh types, i.e. the parametric mesh together with the parametric bezier mesh

neighbours in at least three distinct directions for $d \geq 3$ *and two distinct directions if* $d = 2$*, i.e. the neighbouring cells are in the active region. Define for an arbitrary cell* $Q \in \mathcal{T}$ *the subdivided parametric mesh*

$$\mathcal{T}' = \mathcal{T} \setminus \{Q\} \cup \text{SUBDIV}(Q, k_{\ell(Q)}) \tag{25}$$

together with its subdivided index mesh $\widehat{\mathcal{T}}' \in \text{WGAS}$*. Then for every new index anchor* $\boldsymbol{A}' \in \mathcal{A}_{\boldsymbol{p}}(\widehat{\mathcal{T}}') \setminus \mathcal{A}_{\boldsymbol{p}}(\widehat{\mathcal{T}})$ *there exists an old anchor* $\boldsymbol{A} \in \mathcal{A}_{\boldsymbol{p}}(\widehat{\mathcal{T}})$ *with* $\mathrm{v}_\ell(\boldsymbol{A}') = \mathrm{v}_\ell(\boldsymbol{A})$, $\ell \neq k_{\ell(Q)}$.

The assumption from Lemma 1 is always fulfilled if the polynomial degree is greater than one. If the polynomial degree is one in some direction k, uniform refinement steps may be performed until direction k is refined to ensure this assumption.

3 Data structures

There are several possible meshes to take into consideration for an implementation, see Fig. 3. Firstly, there is the index mesh $\mathcal{T}(\widehat{\Omega})$ where theoretical definitions are done, see Sect. 2. However, if we base computations on the index mesh alone, this will lead to integrals over empty domains, as some distinct indices can lead to knots that coincide. This is no problem per se, but it will practically cause loop iterations within the program, that essentially do nothing. This should be avoided.

Another possible solution is the mesh in the physical domain $\mathcal{T}(\overline{\Omega})$. This is indeed already done in another software project, called `G+Smo`, see [5], available at [29] with an overview of existing modules in [6], where other isogeometric concepts are already fully implemented, e.g. (hierarchical) NURBS, (hierarchical) LR-Splines, etc. However, an implementation of T-Splines in neither 2D nor 3D within the `G+Smo` library is unknown to the authors extent.

A standard finite element code is already given by the `deal.II` library, see [4], or `FEniCS`, see [30] and libraries derived thereof, e.g. [31]. We have opted to base our implementations on the `deal.II` library (version 9.3 [32]), as it features a larger group of researchers together with some important base implementations, the most important being an implementation of Bernstein polynomials which are used for Bézier extraction. We aim to keep integration as simple as possible during the implementation, which is the case if elements of a considered mesh are axis-aligned. Doing so ensures integral variables to be independent of each other, i.e. we integrate over d dimensional boxes. This is either achieved on the index domain or the parametric domain and as the index domain is already disregarded the implementation works fully on the parametric domain. However, as explained in [28, Proposition 7.6], we have to use the Bézier mesh $\mathcal{B}(\Omega)$ of the parametric domain to be able to integrate a fixed number of splines on a specific cell.

For the implementation we thus consider the associated parametric domain Ω from the underlying index domain $\widehat{\Omega}$. For the mesh $\widehat{\mathcal{T}}$, we also consider the associated parametric mesh $\mathcal{T}$, where knot repetitions are ignored; i.e. if $\widehat{\mathrm{E}} \in \mathcal{H}^{(k)}(\widehat{\mathcal{T}})$ is an arbitrary k-dimensional element of $\widehat{\mathcal{T}}$, then its associated parametric element E is an element of the parametric mesh $\mathcal{T}$, if and only if for all directions j there is

$$\xi^{(j)}_{\inf \widehat{\mathrm{E}}_j} = \xi^{(j)}_{\sup \widehat{\mathrm{E}}_j} \iff \inf \widehat{\mathrm{E}}_j = \sup \widehat{\mathrm{E}}_j. \tag{26}$$

In other words, no direction j may collapse to a single point, if the direction j was an interval in the index mesh.

Anchors are mapped *together* with its knot repetitions, i.e. if $\widehat{\mathbf{A}} \in \mathcal{A}_{\mathbf{p}}(\widehat{\Omega})$ is an anchor of the index domain, then the anchor on the parametric domain is given by

$$\mathbf{A} := \bigtimes_{j=1}^{d} \left[\xi^{(j)}_{\inf \widehat{\mathbf{A}}_j}, \xi^{(j)}_{\sup \widehat{\mathbf{A}}_j} \right]. \tag{27}$$

Consider the following example. Let $d = 2$ and $\Xi_1 = \{0, 0, 1, 1\}$, $p_1 = 1$, and $\Xi_2 = \{0, 0, 0, 1, 1, 1\}$, $p_2 = 2$. The indices are given by $I_1 = \{0, 1, 2, 3\}$ and $I_2 = \{0, 1, 2, 3, 4, 5\}$ and the set of index anchors are the horizontal lines of the index mesh constructed by $I_1 \times I_2$ within the active region. Choose $\widehat{\mathbf{A}} = \{1\} \times (2, 3)$, then its parametric anchor is given by $\mathbf{A} = \{0\} \times [0, 1]$. However, if we consider the parametric anchor from $\widehat{\mathbf{A}} = \{1\} \times (1, 2)$, we obtain $\mathbf{A} = \{0\} \times \{0\}$. This demonstrates that anchors on the parametric mesh do not follow a certain rule, as is the case for the index mesh.

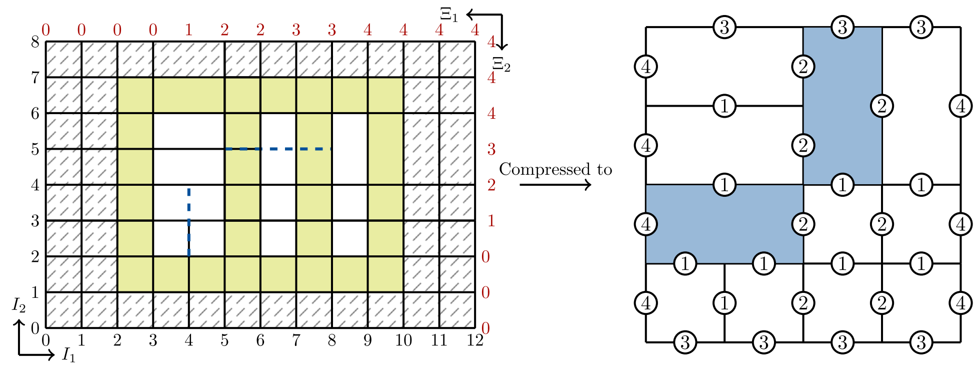


Fig. 4 The data structure on the index mesh (left) is compressed for the data structure on the parametric domain (right) to disregard empty parametric cells

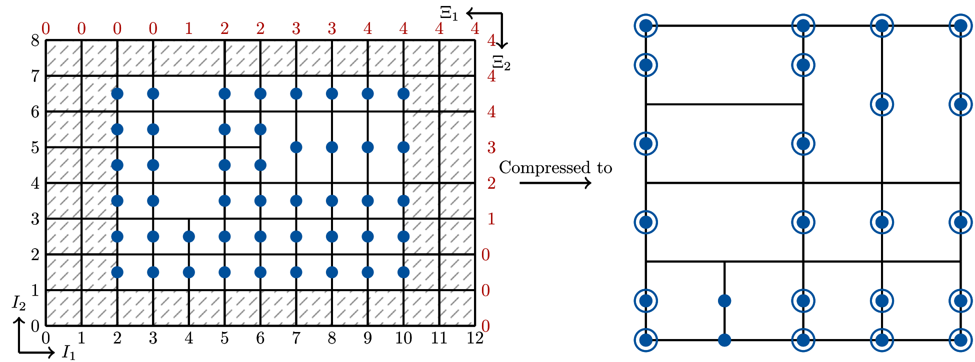


Fig. 5 Mapping behaviour of index anchors to parametric anchors. We again consider the example given in Fig. 4

Further, local index vectors $\mathrm{v}_k(\widehat{\mathbf{A}})$ for an index anchor $\widehat{\mathbf{A}}$ are linked one-to-one to its parametric anchor $\mathbf{A}$, i.e.

$$\mathrm{v}_k(\mathbf{A}) := \left(\xi_\ell^{(k)} \mid \ell \in \mathrm{v}_k(\widehat{\mathbf{A}})\right). \tag{28}$$

Local index vectors for T-junctions $\widehat{\mathtt{T}}$ are mapped in a similar way. However, if in some direction $j \neq \mathrm{pdir}(\widehat{\mathtt{T}})$, $j \neq \mathrm{odir}(\widehat{\mathtt{T}})$ the interval $\widehat{\mathtt{T}}_j$ collapses to a point, the T-junction is not considered for the parametric mesh. This case will be addressed in Sect. 3.3.

3.1 Parametric mesh

The implementation is solely on the parametric mesh $\mathcal{T}$, where knot repetitions are ignored for the construction of cells. However, they are essential for the construction of T-junction extensions and have to be available. For this, we store an array `mof` that stores the multiplicity of each face, i.e. `mof[id]` returns the multiplicity of face $\mathtt{F}_{\mathtt{id}} \in \mathcal{H}^{(d-1)}(\widehat{\mathcal{T}})$. This can later be used to define terminating conditions for the construction of T-junction extensions. Note that per deal.II standards, faces of cells are indexed consecutively for the global mesh.

From [1] we have a dual-compatible set of T-Splines, hence [28, Proposition 7.6] gives us an upper bound for the amount of T-Splines on each element. The upper bound is met exactly, if cells along T-junction extensions are subdivided accordingly. On each such element we can then compute the Bézier representation of each T-Spline, i.e. let $\mathcal{T}_{\mathtt{T}}(\Omega)$ be the mesh described above, then there exists for each cell $\mathtt{Q} \in \mathcal{T}_{\mathtt{T}}(\Omega)$ operators $\mathbf{C}_{\mathtt{Q}}$, s.t. the set of T-Splines $\mathbf{T}$ with support on $\mathtt{Q}$ are given by a linear combination of Bernstein polynomials $\mathbf{B}$ on the reference element $[0,1]^d$,

$$\mathbf{T} = \mathbf{C}_{\mathtt{Q}}^T \mathbf{B}, \tag{29}$$

see also [33, 34] for a detailed description of Bézier extraction of NURBS, resp. T-Splines.

The values of the Bézier splines on the reference elements can be computed preemptively. To use this in our programming, we need the arrays

1. `bezier_elements` that stores every (active) cell that is intersecting some T-junction extension.
2. `extraction_operators` that stores the extraction operators used in (29). Each extraction operator $\mathbf{C}_{\mathtt{Q}}$ is further represented as a matrix.
3. `IEN_array` that returns a set of indices for a specific cell $\mathtt{Q}$. The returned indices correspond to indices of the globally indexed T-splines.

The extraction operators are computed based on the algorithm described in [34], however, some minor changes had to be done. The altered code can be found as MATLAB adaption in Appendix A, Algorithm 5.

In detail, for every T-Spline $\mathbf{T}$, we use its 1D knot vectors to compute 1D extraction operator rows $\mathbf{c}_{\mathtt{Q}}^j$ from Algorithm 5 for a given cell $\mathtt{Q}$ with supp $\mathbf{T} \cap \mathtt{Q} \neq \varnothing$, and use the tensor

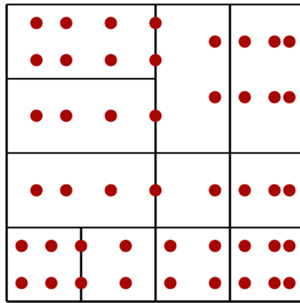


Fig. 6 Relative placement of the barycentre of each anchor given by the example from Fig. 5. Each red dot corresponds to an anchor on the index domain, resp. a T-Spline on the parametric domain

structure to generate the full extraction operator row $\mathbf{c}_{\mathtt{Q}}$ for $\mathbf{T}$, s.t.

$$\mathbf{T} = \mathbf{c}_{\mathtt{Q}}\mathbf{B}, \quad \text{with} \quad \mathbf{c}_{\mathtt{Q}} = \bigotimes_{j=1}^{d} \mathbf{c}_{\mathtt{Q}}^{j}. \tag{30}$$

To work with boundary indicators, an array `boundary_dofs` is stored that returns for a given boundary index a set of splines that have support on that boundary.

An example for the data structure used is given in Fig. 4. The example on the right displays the quantities that are stored, except for extraction operators and the `IEN_array`. In this example we consider polynomial degrees $p_1 = 3$ and $p_2 = 2$, the frame region of the index mesh on the left is highlighted by diagonal lines and consists of the outermost cells. Cells in the active region with knot repetitions are highlighted in green. Only the parametric mesh on the right, together with the multiplicities of each face, is stored. The numbers in the circles on the lines indicate the multiplicity of that line (face). These values are stored in the aforementioned `mof`-array. Further, note that the T-junction extension from $\mathtt{T} = \{6\} \times \{5\}$ yields only a single cell in the parametric mesh due to knot repetitions. Since the implementation is on the parametric mesh, this example also demonstrates that not every cell in the active region of the index mesh will be exclusively marked for refinement, albeit no theoretical restrictions forbid it. For example, it is theoretically allowed to refine $\mathtt{Q} = (2, 3) \times (3, 4)$ in any direction, but since it is practically non-existent in the parametric mesh, it will never be marked for refinement on its own.

3.2 T-Splines

A T-Spline $\mathbf{T}_{\mathbf{A}}$ is a function defined by its parametric anchor $\mathbf{A}$, resp. the corresponding knot vectors. Hence, they are internally described by a d-dimensional array of arrays `kv`, where `kv[k]` corresponds to the `k`-th index vector. Note that we do *not* have to calculate the knot vectors for each T-Spline; from Lemma 1 we know that knot vectors are inherited by bisection. We stress again the fact that the initial mesh is a tensor-product mesh of the given knots. In addition to its knot vectors we also store its associated anchor $\mathbf{A}$ as a pair of two points in the parametric mesh. They describe the bounding box for the anchor and can be used to obtain information about the global position of this T-Spline, e.g. we can easily access information whether or not the T-Spline is located at a specific face. The example from Fig. 4 is continued in Fig. 5 to demonstrate how index anchors and parametric anchors correspond to each other. In this example, index anchors are given as vertical lines of the mesh and are marked here by dots on these lines. When they are brought to the parametric domain on the right, some anchors collapse to a single point, and some anchors may coincide in their parametric representation. This is denoted by concentric circles in the parametric mesh on the right, where the number of circles denote the amount of parametric anchors on that entity. Note that we can not give a well-defined order of anchors, as some parametric anchors coincide.

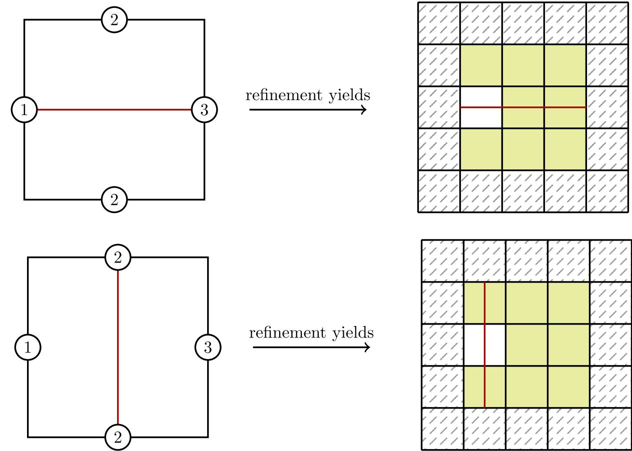


Fig. 7 Effect of subdividing a parametric cell on its corresponding index cells. On the left we have a cell $\mathtt{Q}$ marked for refinement, where each face has the multiplicity stated in the nodes on the faces. The type of refinement is indicated by the red line

Hence, we define the *barycenter* of the T-Spline associated to the parametric anchor $\mathbf{A}$ by

$$(\mathbf{P}_{\mathbf{A}})_k := \frac{1}{p_k + 2} \sum_{\xi \in \mathtt{v}_k(\mathbf{A})} \xi. \tag{31}$$

The barycenters of each T-spline can be used to define a global order. Hence, a T-Spline also stores its barycenter. The example from Figs. 4 and 5 is continued in Fig. 6 to demonstrate the placement of barycenters throughout a given 2D mesh. This information can be used to order the T-splines in the mesh, e.g. in a lexicographic way, see (57).

We have chosen to store the needed control points used for the isogeometric mapping directly within the T-Splines data structure. This way, we do not need to worry about indexing errors between T-Splines and control points. It should be mentioned that in fact the weighted control points are stored, i.e. if P is a control point with weight $\omega \in (0, 1)$, then the control point saved is $P^\omega = (\omega P, \omega)$.

3.3 T-junctions

Since the implementation works on the parametric mesh, subdividing a (parametric) cell $\mathtt{Q}$ in direction k with a face of multiplicity greater than one, will also result in a subdivision of all related index cells corresponding to that face. In detail, consider $\mathtt{Q} \in \mathcal{T}$ from its associated index cell $\widehat{\mathtt{Q}} \in \widehat{\mathcal{T}}$ with a subdivision along direction k' with a face $\mathtt{F} \subset \partial\mathtt{Q}$, $\mathtt{F} \in \mathcal{H}^{(d-1)}(\mathcal{T}) \cap \mathtt{H}^{(k)}(\mathcal{T})$, that is k-orthogonal, $k \neq k'$, and has multiplicity $m = \mathtt{mof[id]} > 1$, where $\mathtt{id}$ is the global index of the given face. Since the multiplicity is greater than one, there exists multiple cells $\widehat{\mathtt{Q}}^{(i)}$, $i = 1, \ldots, m-1$ on the index mesh $\widehat{\mathcal{T}}$ that are associated to $\mathtt{F}$, i.e. (26) reads

$$\inf \widehat{\mathtt{Q}}_k^{(i)} \neq \sup \widehat{\mathtt{Q}}_k^{(i)} \implies \xi^{(k)}_{\inf \widehat{\mathtt{Q}}_k^{(i)}} = \xi^{(k)}_{\sup \widehat{\mathtt{Q}}_k^{(i)}} = \mathtt{F}_k, \tag{32}$$

for all $i = 1, \ldots, m-1$. The subdivision of cell $\mathtt{Q}$ along direction k then also subdivides the above face $\mathtt{F}$ into two children $\mathtt{F}^{(1)}$ and $\mathtt{F}^{(2)}$ with multiplicity m. Each child then has *new* multiple cells from the index mesh $\widehat{\mathtt{Q}}^{(c,i)}$, $c = 1, 2$, $i = 1, \ldots, m-1$. Now, since $\mathtt{F}^{(1)}$ and $\mathtt{F}^{(2)}$ originate from the same face $\mathtt{F}$ there is a common interface $\mathtt{T} = \partial\mathtt{F}^{(1)} \cap \partial\mathtt{F}^{(2)}$ and in particular $\inf \mathtt{F}^{(1)}_{k'} = \sup \mathtt{F}^{(2)}_{k'}$ or vice versa. This yields common faces for the index cells $\widehat{\mathtt{Q}}^{(c,i)}$, i.e. $\inf \widehat{\mathtt{Q}}^{(1,i)}_{k'} = \sup \widehat{\mathtt{Q}}^{(2,i)}_{k'}$ or vice versa for all $i = 1, \ldots, m$. However, this is equivalent to subdividing cells $\mathtt{Q}^{(i)}$ along direction k' and obtaining the children $\mathtt{Q}^{(c,i)}$, $c = 1, 2, i = 1, \ldots, m-1$. See also Fig. 7.

The example depicted shows the effect of subdividing a 2D parametric cell $\mathtt{Q}$ (left) on its corresponding index mesh counterparts (right). As in the previous examples, for the parametric mesh the numbers in circles denote the multiplicity of the corresponding line (face). From this setup, we can infer the that there is a cell $\widehat{\mathtt{Q}}$ in the index mesh that will be mapped to the parametric mesh. Consider now its third face, i.e. the face on the top of the cell with multiplicity two. From the multiplicity we infer a knot repetition of two in the parametric domain, hence there are two indices in the index mesh. This yields a cell $\widehat{\mathtt{Q}}'$ above the corresponding index cell $\widehat{\mathtt{Q}}$ whose corresponding parametric "cell" has empty volume. From the multiplicity of face 1, i.e. the right line, we infer two consecutive index cells that collapse to a single line by the same arguments. Repeating these arguments for the remaining two faces, we obtain the corresponding index region on the right of Fig. 7. The cells found above are highlighted in green. The subdivision of $\mathtt{Q}$ yields a subdivision of its faces, and from the corresponding index cells we also get a subdivision of these cells. To stress the fact that only corresponding cells are subdivided in the index mesh, we have displayed a slightly larger region in the index mesh highlighted by diagonal lines.

Thus, the refinement of the parametric mesh ensures that every generated T-junction on the index level is mapped according to condition (26).

T-junctions are not stored internally, instead we only store the cells that intersect the T-junction extensions as mentioned in Sect. 3.1. However, to find these cells, we first have to detect T-junctions. Since the mesh is generated from a tensor product structure on the coarsest level using symmetric box subdivisions, we can find associated cells of T-junctions $\mathtt{T}$ by iterating the faces $\mathtt{F} \subset \partial\mathtt{Q}$ of a cell and check whether or not it has children. From the associated cell $\mathtt{Q}$ and a face $\mathtt{F}$ of the cell, we can then use the definition of local knot vectors for T-junctions to find all cells that intersect the T-junction extension.

Algorithm 3 Bezier cell collection algorithm in 2D for a given T-junction $\mathtt{T}$ its associated cell $\mathtt{Q} = \mathrm{ascell}(\mathtt{T})$ and the face number face_no of $\mathtt{T}$ on $\mathtt{Q}$, i.e. $\mathtt{T} \subset \mathtt{F}_{\text{face_no}} \subset \partial\mathtt{Q}$

```
bezier_cells_2D(T, Q, face_no)){                    1
    // Get the index of opposite face                2
    ofn      = opposite_face_no(                     3
                 Q, face_no);                        4
    // The pointing direction is                     5
    // defined by floor(ofn/2)                       6
    dir      = floor(ofn/2);                         7
    // Get the corresponding face                    8
    F        = face(Q, ofn);                         9
    // Define the maximum number of                  10
    // indices to use for the                        11
    // T-junction extension from the                 12
    // definition                                    13
    N        = floor((p_dir+1)/2);                   14
    count    = 0, i = 0;                             15
    𝒬 = {}                                           16
    while (count < N                                 17
           && !has_children(Q)) {                    18
        // In each iteration,                        19
        // increase count                            20
        count += mof[F];                             21
        // Add the current cell                      22
        // to the list                               23
        𝒬 = 𝒬 ∪ {Q};                                 24
        // Get the new cell                          25
        Q = neighbor(Q, ofn);                        26
        // Get the new face                          27
        F = face(Q, ofn);                            28
    }                                                29
    return 𝒬;                                        30
}                                                    31
```

How to exactly find the Bézier cells is given in Algorithm 3 and 4, for the dimensions 2 and 3 respectively. Each algorithm finds the Bézier cells $\mathcal{Q}$, i.e. the cells cut by the T-junction extension, of a T-junction $\mathtt{T}$ with its associated cell $\mathtt{Q} = \mathrm{ascell}(\mathtt{T})$. The input `face_no` refers to the corresponding index, s.t. $\mathtt{T} \subset \mathtt{F}_{\mathtt{face_no}} \subset \partial\mathtt{Q}$. Further, both algorithms use neighbouring relations between cells. This is a functionality made available by the `deal.II` library. In

general, `neighbour(Q, f)` returns the cell on the opposite side of face `f`. Note that the neighbour may be an in-active cell, i.e. it may already be refined. For the readers unfamiliar with `deal.II` internal data structures and conventions, we have briefly explained enumeration of entities and neighbours in Appendix B.

The algorithm for $d = 2$ is relatively simple, as we just have to find cells along a single direction, that is the pointing direction. We distinguish between positive and negative directions $\pm\,\mathrm{pdir}(\mathtt{T}) = \pm k$, where we traverse the neighbours in positive direction, if $\mathtt{T}_k \subset \inf \mathrm{ascell}(\mathtt{T})_k$, and in negative direction, if $\mathtt{T}_k \subset \sup \mathrm{ascell}(\mathtt{T})$. And since `face_no` yields the face of `Q` on which the T-junction lies, we simply have to search the neighbours in the opposite direction.

If the new neighbour $\mathtt{Q}' = \mathtt{neighbor}(\mathtt{Q}, \mathtt{ofn})$ in that direction is already refined, the refinement algorithm provided by [23] guarantees that this cells refinement direction is not orthogonal to the pointing direction of `T`, i.e. in 2D it is refined by a k'-orthogonal subdivision, $k' = \mathrm{odir}(\mathtt{T})$. If the next neighbour `neighbor(Q, ofn)` is again *not* refined, then the next neighbour is an associated cell of a new T-junction $\mathtt{T}'$ from which we run the algorithm again. Thus, the algorithm for `T` may stop at $\mathtt{Q}'$.

Algorithm 4 Bezier cell collection algorithm in 3D for a given T-junction $\mathtt{T}$ its associated cell $\mathtt{Q} = \mathrm{ascell}(\mathtt{T})$ and the face number face_no of $\mathtt{T}$ on $\mathtt{Q}$, i.e. $\mathtt{T} \subset F_{\mathrm{face_no}} \subset \partial\mathtt{Q}$

```
1  bezier_cells_3D(T, Q1, face_no)){
2      // Define necessary quantities as
3      // before from Algorithm 3
4      ofn      = opposite_face_no(
5                   Q1, face_no);
6      pdir     = floor(ofn/2);
7      rdir     = i ∈ {1,2,3} \ {pdir(T), odir(T)}
8      F1        = face(Q, ofn);
9      Np       = p_pdir + 1;
10     Nr       = p_rdir + 3;
11     pcount = 0, rcount = 0;
12     i = 0;
13
14     // find rmin and rmax according
15     // to (33) by traversing the
16     // grid in rdir similar
17     // to Algorithm 3
18     rmin = min v_rdir(T);
19     rmax = max v_rdir(T);
20     𝒬 = {};
21     while (pcount < floor(Np/2)
22             && !has_children(Q1)) {
23         pcount += mof[F];
24         Q2 = Q1;
25         F2 = face(Q2, rdir);
26         while (rcount < floor(Nr/2) &&
27                 F2_rdir >= rmin &&
28                 !has_children(Q2)) {
29             rcount += mof[F2];
30             𝒬 = 𝒬 ∪ {Q2};
31             Q2 = neighbor(Q2, rdir);
32             F2 = face(Q2, rdir);
33         }
34
35         Q2 = neighbor(Q1, ++rdir);
36         F2 = face(Q2, rdir);
37         while (rcount < Nr &&
38                 F2_rdir-1 <= rmax &&
39                 !has_children(Q2)) {
40             // Proceed as previous
41             // while loop
42         }
43         Q1 = neighbor(Q1, ofn);
44         F1 = face(Q1, ofn);
45     }
46     return 𝒬;
47 }
```

The 3D case is very similar, however, we have to extend the T-junction in two directions now. We denote the second direction to traverse neighbours with `rdir` in Algorithm 4. For every step in `ofn`, defined as in 2D, we traverse the neighbours in positive *and* negative direction. Since the T-junction extension is defined by the index vectors at the position of the T-junction, we have to ensure that these limits are given while traversing all neighbours. The quantities are denoted by `rmin`, resp. `rmax`, and are defined as

$$\mathtt{rmin} = \min \mathrm{v}_{\mathtt{rdir}}(\mathtt{T}) \text{ and } \mathtt{rmax} = \max \mathrm{v}_{\mathtt{rdir}}(\mathtt{T}), \tag{33}$$

for the parametric knot vectors

$$\mathrm{v}_k(\mathtt{T}) := \{\xi_\ell^{(k)} \mid \ell \in \mathrm{v}_k(\widehat{\mathtt{T}})\}, \tag{34}$$

where $\widehat{\mathtt{T}} \in \mathcal{T}(\widehat{\Omega})$ is the associated T-junction in the index domain.

4 Differences between `deal.II` and `deal.t`

In this section we will discuss main differences when using a `deal.t` based finite element solver versus a standard `deal.ii` technique. A full Tutorial for readers unfamiliar with `deal.II` is given in Appendix C.

As the |`TS_Triangulation`| class inherits from `deal.II`s |`Triangulation`|, we can use every function of the base class for the derived class. However, some functions have been overwritten to fit either the data structures explained in Sect. 3, or enforce special behaviour. For example the |`create_triangulation`| usually generates a grid from a list of vertices and a list of cell data. It has been overwritten to take the data used for the isoparametric mapping instead, i.e. a list of knot vectors, a list of polynomial degrees, and a list of weighted control points. The function header is then declared as

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::create_triangulation(
        const std::vector<
            std::vector<double>
        >& knot_vectors,
        const std::vector<
            unsigned int
        >& degree,
        const std::vector<
            Point<spacedim+1>
        >& wcps
);
```

We also provide an object |IPF_Data| that essentially stores these data types and allows us to overload the definition of |create_triangulation| with

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::create_triangulation(
        const IPF_Data<dim, spacedim>& data
);
```

These two functions are then called respectively within the constructor of TS_Triangulation. We highly recommend using the constructor based on |IPF_Data| as it allows creating the correct grid within a single line, see also the example in Appendix C.

Further, remember that the implementation differs between the parametric mesh $\mathcal{T}$ and the parametric Bézier mesh $\mathcal{B}$. The former is used for refinement and the latter for assembly routines. To switch between these types of meshes we have provided the functions

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::refine_bezier_elements();

template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::coarsen_bezier_elements();
```

that find all relevant Bézier elements using Algorithm 3 in 2D, resp. Algorithm 4 in 3D, and refines them via a symmetric, level-dependent box bisection, resp. coarsens the refined Bézier elements.

Since finite elements are defined globally and not on the reference cell, we have to provide information to applications about which T-spline, resp. DoF, is at the boundary. This information can be obtained with

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::set_boundary_dofs();

template<int dim, int spacedim>
std::map<
    types::boundary_id,
    std::vector< types::global_dof_index>
    > TS_Triangulation<dim, spacedim>
    ::get_boundary_dofs();
```

where the former function computes the object returned by the latter. We advise to directly call set_boundary_dofs after each refinement.

The rest of this chapter explains main differences when assembling vectors, matrices, (estimated) errors, etc., manipulating the mesh, and other miscellaneous functions such as output routines to print e.g. the grid.

4.1 Assembly

Similar to deal.IIs FEValues object to define finite element values on a cell, deal.T provides an object of the type TSValues that *mimics* the behaviour of the deal.II original function. Note that it does not inherit from the original. T-spline values on a cell are computed using Bézier extraction, as explained in Sect. 3.2. During construction of a TSValues object, we store tables of Bernstein-values on quadrature points on the reference element $[0, 1]^d$, $d = 2, 3$. The constructor takes as input a reference to the used triangulation, to retrieve the necessary data, i.e. control points and extraction operators, a list of integers to determine the amount of points used for Gaussian quadrature and a deal.II object UpdateFlags, see Appendix C for a short description of this object, or [4] for a detailed explanation. In summary, we get

```
TSValues(
  const TS_TriangulationBase<dim, spacedim>*
        tria,
  const unsigned int
        n_gauss_points,
  const UpdateFlags                        flags
);
```

Note that we allow different polynomial degrees in every direction, i.e. an-isotropic polynomials *can* be used for applications.

Similarly, deal.t provides an object TSFaceValues, that represents finite element values on faces. Its syntax and construction is as above.

Similar to the deal.II equivalents, these functions are suited with appropriate reinit() methods, in order to reinitialize finite element values on given cells, resp. faces.

Thus, applications with deal.t are very similar to deal.II, i.e. we get approximately the following lines

```
// Define TS_Triangulation<dim, spacedim> tria
// Define UpdateFlags flags
// somewhere
TSValues<dim, spacedim> ts_values(
    &tria,
    {/* list of gauss points */},
    flags
);
TSFaceValues<dim, spacedim> face_values(
    &tria,
    {/* list of gauss points */},
    flags
);
// ...
for (const auto& cell :
        tria.active_cell_iterators()){
    ts_values.reinit(cell);
    // assembly code for interior
    for (unsigned int f = 0;
            f < GeometryInfo<dim>::
                faces_per_cell;
            f++){
        face_values.reinit(cell, f);
        // assembly code for boundary
    }
}
```

Note, that deal.t does not provide a deal.II equivalent to the DoFHandler object. In standard deal.II applications this object does all the work of handling the information about DoFs at each vertex, line, quad, etc. This information is processed already within the TS_Triangulation implementation. An object TS_DoFHandler is already planned out for future implementations. This will enable further deal.II functionality that require an object DoFHandler.

For error estimations we have suited the TS_Triangulation with an internal error estimator for scalar Poisson-like prob-

lems, i.e. find $u\colon \mathbb{R}^d \to \mathbb{R}$, s.t.

$$\begin{cases} -\nabla\cdot(\sigma\nabla u) = f, & \text{in } \overline{\Omega} \\ \frac{\partial u}{\partial n} = g_i^N, & \text{on } \Gamma_i^N \subset \partial\overline{\Omega}, \\ u = g_j^D, & \text{on } \Gamma_j^D \subset \partial\overline{\Omega}, \end{cases} \tag{35}$$

where $\sigma\colon \mathbb{R}^d \to \mathbb{R}$, $f\colon \mathbb{R}^d \to \mathbb{R}$, $g_i^N\colon \mathbb{R}^d \to \mathbb{R}$, $i = 1,\dots,n$, $g_j^D\colon \mathbb{R}^d \to \mathbb{R}$, $j = 1,\dots,m$, $n, m \in \mathbb{N}$ are given. The error estimator used is given by a standard residual error estimator, i.e. $\eta_{\mathcal{T}}\colon \mathcal{T} \to \mathbb{R}$ given the Galerkin solution u_h is defined by

$$\eta_{\mathcal{T}}(\mathtt{Q}) := \Bigg(h_{\mathtt{Q}}^2 \|\nabla\cdot(\sigma\nabla u_h) + f\|_{\mathtt{Q}}^2 + \sum_{\substack{\mathtt{E}\subset\partial\mathtt{Q} \\ \mathtt{E}\in\mathcal{H}^{(d-1)}(\mathcal{T})}} h_{\mathtt{E}} \|R_{\mathtt{E}}(u_h)\|_E^2 \Bigg)^{\frac{1}{2}}, \tag{36}$$

where $h_{\mathtt{Q}}$ is the diameter of $\mathtt{Q}$ given by

$$h_{\mathtt{Q}} = \max_{i=1,\dots,d} (\sup \mathtt{Q}_i - \inf \mathtt{Q}_i) \tag{37}$$

$\mathtt{E}$ is a face of $\mathtt{Q}$, $h_{\mathtt{E}}$ is the diameter of $\mathtt{E}$, i.e.

$$h_{\mathtt{E}} = \max_{i\neq k} (\sup \mathtt{E}_i - \inf \mathtt{E}_i), \tag{38}$$

where k is the index s.t. $\mathtt{E}$ is a singleton. The *edge residual* R_E is given by

$$R_{\mathtt{E}}(u_h) := \begin{cases} \frac{1}{2}[\![\frac{\partial u_h}{\partial n}]\!]_{\mathtt{E}}, & \text{if } \mathtt{E} \in \overline{\Omega}\setminus\partial\overline{\Omega}, \\ g_i - \frac{\partial u_h}{\partial n} & \text{if } \mathtt{E} \in \Gamma_i^N, \\ 0 & \text{if } \mathtt{E} \in \Gamma_j^D, \end{cases} \tag{39}$$

where n is a normal to the face $\mathtt{E} = \mathtt{Q} \cap \mathtt{Q}'$ and $[\![\bullet]\!]_{\mathtt{E}} = \bullet|_{\mathtt{Q}} - \bullet|_{\mathtt{Q}'}$ denotes the jump along the face $\mathtt{E}$.

The general syntax of the implemented error estimator is given by

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::poisson_residual_error_estimate(
    const std::vector< unsigned int >&
        n_gauss_points,
    const Function<spacedim>*
        rhs_fcn,
    const Function<spacedim>*
        sigma,
    const std::map<types::boundary_id,
            Function<spacedim>* >&
        neumann_bc,
    const Vector<double>&
        solution,
    std::map< cell_iterator,
                double >&
        residuals
);
```

Corresponding overloads are given for $\sigma = 1$ and/or $n = 0$, i.e. no Neumann boundary conditions, by dropping the respective arguments.

Note that if there is at least C^1-continuity along some face $\mathtt{E}$, the jump along this face is zero. The implemented error estimator uses the `mof`-array, described in Sect. 3, to detect C^0-faces and only computes jumps at those faces. A face $\mathtt{E} \in \mathtt{H}^{(k)}(\mathcal{T}) \subset \mathcal{H}^{(d-1)}(\mathcal{T})$ has C^0-continuity if $\mathtt{mof[F]} = p_f$. This follows from the definition of B-splines.

4.2 Manipulation of the mesh

We have provided different functionality to modify the mesh. We have overwritten the base class function `execute_coarsening_and_refinement()` to perform the refinements of the mesh *and* execute knot insertion from the subdivided cells. For more information on knot insertion see e.g. [11, 35]. It also computes the coarse neighbourhood of marked cells, as explained in Sect. 2.2. Note that despite the name, we have not yet implemented a proper coarsening algorithm for T-splines, since there is no coarsening algorithm given for T-splines to the authors extent.

We have also overwritten the base class function `refine_global(int t)` that refines the mesh globally `t`-times using a simple for loop of declaring the appropriate refinement flags on a cell and then executing the refinement.

For local refinement, the user has to provide a list of cells to be marked for refinement. Internally, upon initialization of a |`TS_Triangulation`|, we store an offset |`off`| for refinement that ensures that the first refinement is along the longest edge. E.g. if the mesh is given by the cell $\mathtt{Q} = [0,1]\times[0,2]\times[0,\frac{1}{2}]$ then the cell is longest in the second dimension and the first refinement will be carried out by a 2-orthogonal subdivision to obtain $\mathtt{Q}_1 = [0,1]\times[0,1]\times[0,\frac{1}{2}]$ and $\mathtt{Q}_2 = [0,1]\times[1,2]\times[0,\frac{1}{2}]$. Using this offset, the level-dependent subdivision of a cell $\mathtt{Q} \in \mathcal{T}$ becomes the $k_{\ell(\mathtt{Q})}$-orthogonal bisection of $\mathtt{Q}$ where $k_{\ell(\mathtt{Q})} = 1 + \big((\ell(\mathtt{Q}) + \mathtt{off}) \bmod d\big)$.

The user may handle this offset on his own and set the refinement flags manually, however, we also provide a function that considers this. To let the class handle the refinement flags, we use

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::set_refine_flags(
        const std::vector<
            TriaActiveIterator<
                ::CellAccessor<dim, dim>
            >
        > &mark
);
```

where we provide a list of cells to be flagged for refinement.

Another important part of isogeometric grid manipulation is degree elevation, see again [11, 35] for details. Degree elevation in this implementation is only allowed at the coarsest level and can be obtained with

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::degree_elevate(
        const unsigned int d,
        const unsigned int t
);

template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::degree_elevate_global(
        const unsigned int t
);
```

where the former elevates the degree of all T-splines by `t` in direction `d` and the latter elevates the degree in every direction by `t`.

4.3 Miscellaneous functions

It may be of interest to print different results of the grid to some files. One possibility from `deal.II` is to use a `GridOut` object to print the grid, e.g. to a `.svg` file. However, this will only print the underlying parametric mesh and not the mapped parametric mesh, which is of more significance for results.

We have provided functions to print quadrature points in the physical domain, grid lines, vertices, and the T-spline functions to `".dat"` files.

To print the isoparametric mapping and or the respective T-splines used, we simply use

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::printIPF(
        const std::string& out_name,
        const unsigned int precision = 4,
        bool print_splines = true,
        bool print_IPF = false
);
```

where we provide a `std::string` as output file, and a `precision` that determines how many decimals are being saved. A suffix `"_IPF_2d.dat"` or `"_IPF_3d.dat"` is added to the name for the isoparametric mapping, depending on the *space dimension*. The suffix `"_splines.dat"` is added to the name of the T-spline file. Note that corresponding folders and sub-folders have to exist. Note further that this function assumes that we are on the parametric Bézier mesh and have already calculated the Bernstein tables with `setup_tables`.

To obtain the grid lines in the physical domain, we use

```
template<int dim, int spacedim>
void TS_Triangulation<dim, spacedim>
    ::print_IPF_wireframe(
        const std::string& out_name,
        const unsigned int precision = 4,
        const unsigned int n_intervals = 11
);
```

where a suffix `"_parametric_grid.dat"` is added to the declared output name. The new input `n_intervals` determines the amount of sub-intervals an existing edge is partitioned for the output. Choose a low number if the physical mesh consists of straight lines only, see e.g. Figure 2 and choose a higher number if the mesh has curved parts, see Fig. 8.

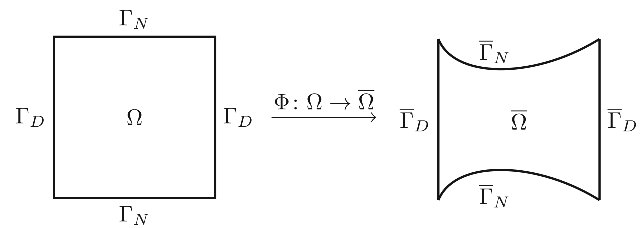


Fig. 8 Depiction of the physical domain used for demonstration along its parametric counterpart which is actually implemented

To obtain the grid lines in the parametric domain, we use `print_grid(...)` with the same arguments as before. A suffix `"_grid.dat"` is added to the declared output name.

To obtain the corresponding grid lines of the Bézier mesh, rerun the above functions on the Bézier mesh, preferably with a different name to ensure the previous outputs are not overwritten. Alternatively, for the parametric Bézier mesh, there is a function `print_bezier_grid()`.

4.4 Limitations

As mentioned before, one big limitation is the restriction of scalar-valued problems.

Note that the algorithm provided by [23] allows q-graded refinement, i.e. the k-orthogonal subdivision of a cell `Q` into q equal parts. However, by `deal.II` limitations this is not possible since multi-level hanging nodes are not implemented.

Further, note that the refinement we carry out within our implementation is anisotropic. There is currently no support for a parallel triangulation when using anisotropic refinements. This, however, does not restrict us from using parallel computations as assembly loops may be divided along processors or threads. It just means that the subdivision of a `TS_Triangulation` may not be carried out on multiple processors or threads.

The way we designed the `TSValues` and `TSFaceValues` objects uses *a lot* of space to store the necessary T-spline value tables. This can be reduced by further adapting `deal.II` behaviour, i.e. by reinitializing the tables of a `TSValues` object on each cell instead of assigning each cell a new object of that type.

5 Experiments

In this section we consider a few examples to use `deal.t`. Each subsection gives the corresponding problem to be solved, explains the data used for the geometric mapping, and gives an example after some refinement steps. Where

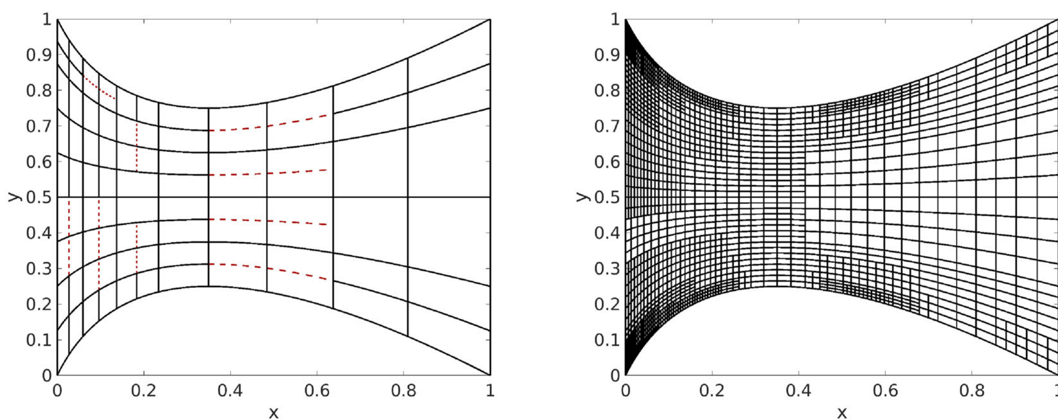


Fig. 9 Physical mesh from Sect. 5.1 after different refinement levels. On the left is the physical mesh with red dashed lines indicating Bézier extensions. On the right is the physical Bézier mesh after 15 refinement levels

necessary for Neumann boundary conditions, the geometric mapping is explicitly stated. Lastly, results are given for various polynomial degrees.

Note that every considered problem uses isotropic polynomial degrees, i.e. $p_x = p_y (= p_z)$. However, it is technically possible to use anisotropic polynomials. Further, the geometric mapping is given at a base level, and higher polynomial degrees are obtained by order elevation, see [11], and where necessary global refinements are employed to obtain a finer initial mesh.

Further, the resulting linear systems $K_h u_h = F_h$ are solved using CG-iterations and a diagonal preconditioner. On level $l > 0$ a relative accuracy of $e_{l-1} \cdot 10^{-4}$ for the iterative solver was chosen, and for level $l = 0$, the relative accuracy was defined as 10^{-4}.

5.1 Poisson's equation on a squished domain

We consider

$$\begin{cases} -\Delta u = f(x, y), & \text{in } \overline{\Omega} \\ \frac{\partial u}{\partial n}(x, y) = g(x, y), & \text{on } \Gamma_N, \\ u(x, y) = 0, & \text{on } \Gamma_D, \end{cases} \tag{40}$$

where the right-hand-side function f and the boundary function g are defined using the exact solution

$$u(x, y) = \frac{1}{5\pi^2} \sin(2\pi x) \cos(\pi y). \tag{41}$$

The physical domain is defined using the data $\Xi_x = \{0, 0, 0, 1, 1, 1\} = \Xi_y$, $p = 2$, and controls

$$\mathbf{P}^w = \begin{bmatrix} 0.0 & 0.2 & 1.0 & 0.0 & 0.2 & 1.0 & 0.0 & 0.2 & 1.0 \\ 0.0 & 0.5 & 0.0 & 0.5 & 0.5 & 0.5 & 1.0 & 0.5 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}. \tag{42}$$

The resulting domain is depicted in Fig. 8 together with its boundaries Γ_N and Γ_D. This data yields the isoparametric mapping

$$\Phi(x, y) = \begin{bmatrix} 0.4x + 0.6x^2 \\ (x - x^2)(1 - 2y) + y \end{bmatrix} \tag{43}$$

from which we can define the normal vectors to the boundary $\overline{\Gamma}_N$ as orthogonal vectors to the tangents $\partial_x \Phi(x, 0)$ and $\partial_x \Phi(x, 1)$, since

$$\Gamma_N = \bigcup_{i=0,1} \{\Phi(x, i) \mid x \in [0, 1]\}. \tag{44}$$

For error estimation, we use the standard residual error estimator $\eta(\mathrm{Q})$ on a cell Q given in Sect. 4.1. We use a quantile marking strategy by refining the

$$N_r = \lceil \alpha \# \mathcal{T} \rceil \tag{45}$$

cells with the highest errors, where $\alpha = 0.0075$. The resulting meshes are depicted in Fig. 9. On the left we see the physical mesh for polynomial degree $p = 3$ after 8 refinement levels. Red dashed lines indicate Bézier extensions and hence the Bézier mesh. On the right, we see the same domain after 15 refinement levels.

We have computed the numerical solution for T-spline degrees $p = 3, \dots, 7$ and given the H^1-errors to the exact solution over the degrees of freedom in Fig. 10. Additionally, the H^1-error using a standard finite element method with $p = 1$ is given. We have performed refinements, until we reached an error smaller than 10^{-8}. Note, that in FEM, each new cell yields $(p+1)^d$ new basis functions. Using IGA-FEM with T-splines on the other hand, a bisection of a cell yields $(p+1)^{d-1}$ additional basis functions, assuming the same polynomial degree in every direction. Thus, for a full refinement of a cell, i.e. the cell is bisected once in each

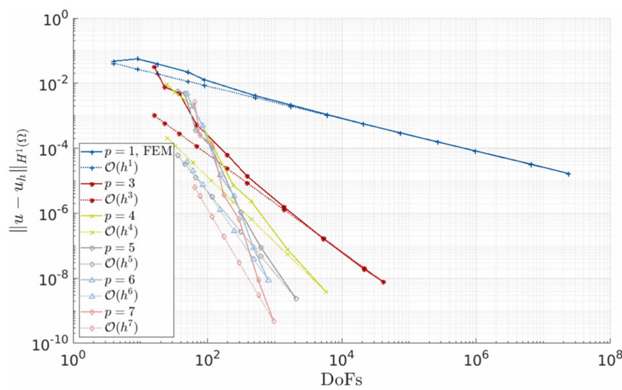


Fig. 10 H^1-Errors of the problem from Sect. 5.1 with various polynomial degrees. The reference lines are given in matching colours and markers as dashed lines

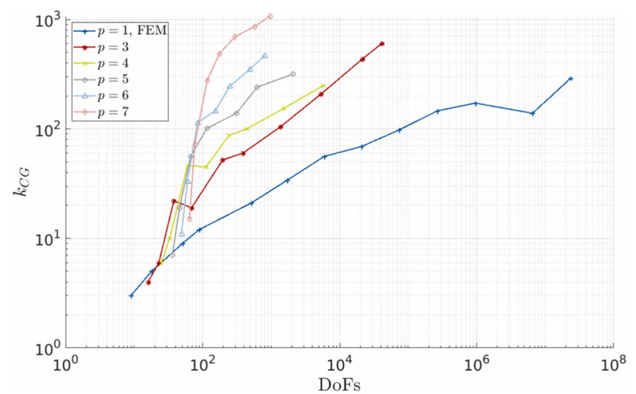


Fig. 11 CG-iteration numbers over degrees of freedom for the Problem considered in Sect. 5.1

direction, we obtain ($d > 1$)

$$(p+1)^{d-1}\sum_{i=0}^{d-1} 2^i = (p+1)^{d-1}(2^d - 1) \tag{46}$$

additional basis functions. Since for fixed $d > 1$, the term $2^d - 1$ is constant in p, the number of new basis functions added to the system grows by one order slower.

Note that we consider this example as a benchmark for our implementation. It does not involve any sort of singularities. It is just a short demonstration of `deal.t` to show its usage as explained in Appendix C and demonstrates how to set Neumann boundary conditions for a manufactured problem on curved domains.

Asymptotically, we obtain optimal convergence rates, as shown in Fig. 10. Note that the resulting linear systems are relatively small compared to systems derived from standard FEM. In fact, the considered example goes as high as around 10^5 DoFs for the $p = 3$ case. The other polynomial degrees are below 10^4 DoFs. As mentioned in the introduction, for a fair comparison we opted to use a CG-solver with a diagonal preconditioner to solve the linear systems. However, considering the size of the problems, it is more reasonable to use a direct method for T-splines.

This also becomes clear, when considering iteration numbers for the iterative method in Fig. 11. The reason lies in the nature of IGA-FEM, since the basis functions span multiple cells of a triangulation, especially for higher degrees. This results in mass- and stiffness-matrices with low sparsity compared to standard FEM.

As a last note on this subsection, let us explain the procedure for solving this problem using standard FEM. In this case, we have to define a mapping to get the new vertices of the triangulation after refinement, in order to reduce the discretization error at the smooth boundary Γ_N. This mapping is know as Φ. A new vertex after (isotropic) subdivision is

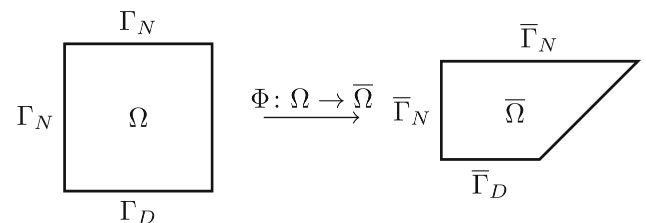


Fig. 12 Depiction of the physical domain used for the L-Shape domain along with its parametric counterpart

then given as

$$P^{new} = \Phi^{-1}\left(\tfrac{1}{2}\Phi(P_0) + \tfrac{1}{2}\Phi(P_1)\right), \tag{47}$$

where P_0, and P_1 are two neighbored vertices of a common cell. This includes knowledge of the inverse, Φ^{-1}, which in this case is given by

$$\Phi_1^{-1}(\hat{x},\hat{y}) = \frac{(\sqrt{15\hat{x}+1} - 1)}{3} \tag{48}$$

$$\Phi_2^{-1}(\hat{x},\hat{y}) = \frac{9\hat{y} - 5(\sqrt{15\hat{x}+1} - 3\hat{x} - 1)}{30\hat{x} - 10\sqrt{15\hat{x}+1} + 19}. \tag{49}$$

There are also other methods to get an *approximation* of the boundary vertices, e.g. by assigning certain types of manifolds to it. However, the above method guarantees exact boundary vertices.

5.2 L-Shape domain

We consider again problem (40) on the L-Shape domain $\Omega_L = [-1,1]^2 \setminus (-1,0)^2$. For this setting, we have $f(x,y) = 0$ and the boundary function g is defined from the exact solution

$$u(r,\varphi) = r^{\frac{2}{3}} \sin\left(\tfrac{2\varphi+\pi}{3}\right), \tag{50}$$

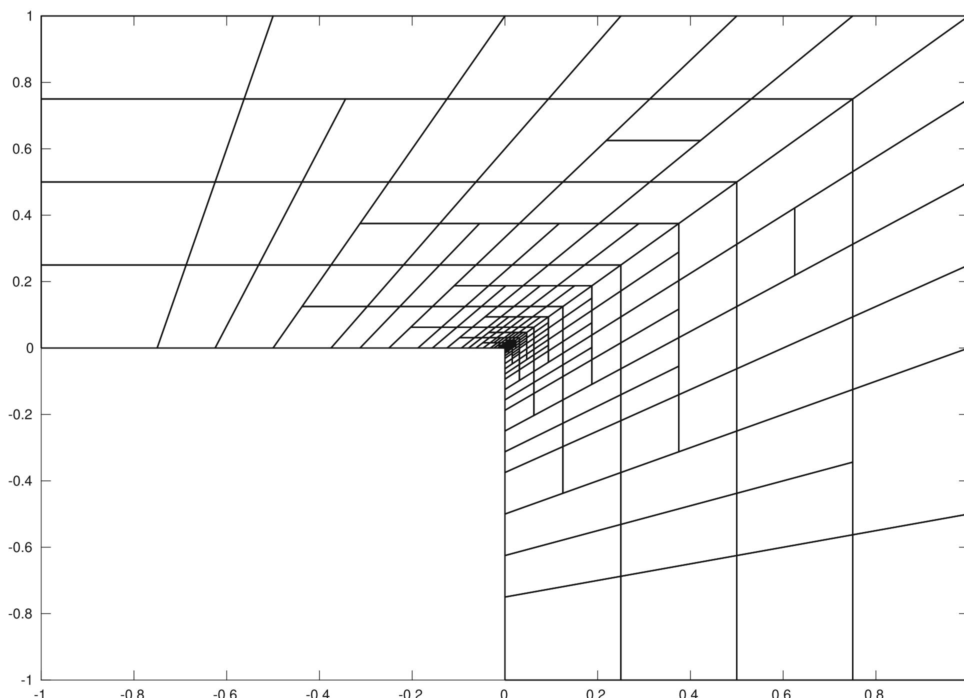


Fig. 13 Employing symmetry on the domain from Fig. 12, we obtain the complete domain

given in polar coordinates. The solution is symmetric along the diagonal $y = x$, hence we only give a parametrization of half the L-Shape domain by knot vectors $\Xi_x = \{0, 0, 1, 1\} = \Xi_y$, degrees $p = 1$, and control points

$$\mathbf{P}^w = \begin{bmatrix} -1.0 & +0.0 & -1.0 & +1.0 \\ +0.0 & +0.0 & +1.0 & +1.0 \end{bmatrix}. \tag{51}$$

The resulting mapping is depicted in Fig. 12. Note that we did not set boundary conditions along the line $y = x$. An intermediate mesh at level 21 and polynomial degree $p = 2$ is given in Fig. 13, where symmetry is already employed. Computations began after three initial global refinement steps, yielding eight elements as initial mesh. In general, the initial mesh for computations for any considered polynomial degree p was globally refined $N_p = \lfloor \frac{3p}{2} \rfloor$ times to obtain 2^{N_p} elements before local refinements and error estimations took part.

The marking strategy used for refinement is again the same as in Sect. 5.1. The results for polynomial degrees $p = 3, \ldots, 7$ are given in Fig. 14. Note that in contrast to Sect. 5.1, we refined until we reached 41 (half-)levels, yielding 20 full refinement steps.

Note that the reference lines correspond to $\mathcal{O}(h^{p+1})$, thus the results exceed optimal convergence. Note also that we beat the optimal convergence rates by up to three magnitudes, e.g. the result for $p = 6$ behaves asymptotically as $\mathcal{O}(h^9)$. This is most likely due to some symmetry effects of the considered PDE.

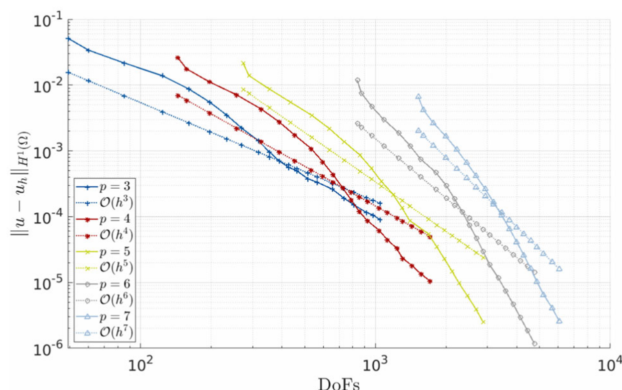


Fig. 14 H^1 errors of the problem from Sect. 5.2 with various polynomial degrees. The reference lines are given in matching colours and markers as dashed lines

For comparison we have solved this problem with standard Q_p elements for the same polynomial degrees on the whole domain. The results are depicted in Fig. 15. There, we have not run a proper number of global refinements, which can be seen in the plots as well, by a sudden drop of the error. The super-convergent effects mentioned before can be seen here as well. This may be subject to Gaussian quadrature rules used in this example. For simplicity's sake, we have omitted the reference lines in this plot.

To get an idea about the condition number of the matrix, we take a look at the quotient

$$\frac{k_i^p / k_{i+1}^p}{N_i^p / N_{i+1}^p}, \tag{52}$$

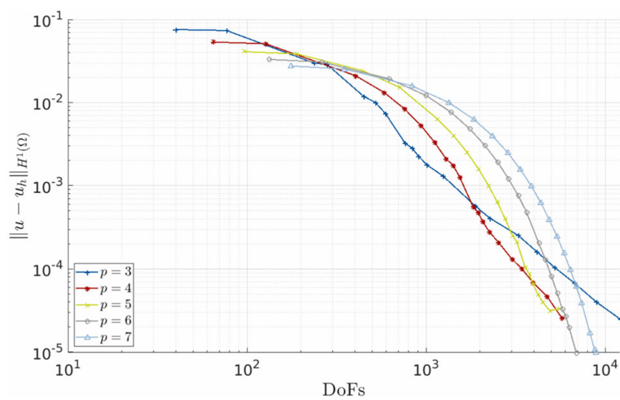


Fig. 15 H^1 errors of the problem from Sect. 5.2 with various polynomial degrees using standard Q_p elements. Reference lines are omitted for simplicity

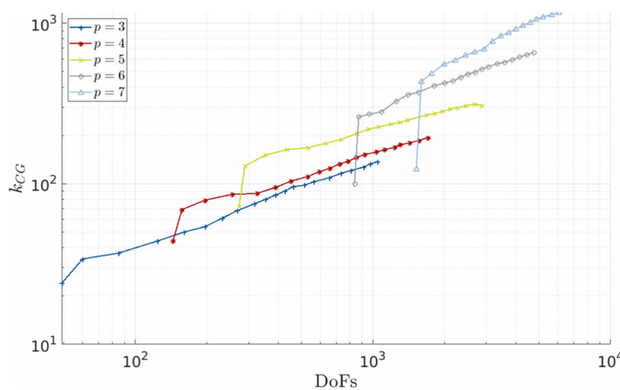


Fig. 16 CG iterations for the problem described in Sect. 5.2 using T-splines

where k_i^p and N_i^p are the iteration numbers, resp. number of DoFs of the CG-iteration for the basis functions of degree p at level $i > 0$. Take e.g. $p = 4$, and $i = 4$. From Fig. 16 we infer

$$\frac{k_i^p / k_{i+1}^p}{N_i^p / N_{i+1}^p} = \frac{87/95}{325/389} \approx 1.0961. \tag{53}$$

In fact, we can see that

$$\frac{k_i^p / k_{i+1}^p}{N_i^p / N_{i+1}^p} \sim 1, \tag{54}$$

for all $p = 3, \ldots, 7$, and i. This yields

$$\kappa(K_{p,h}) \sim \mathcal{O}(N_p^2), \tag{55}$$

for the condition number $\kappa(K_{p,h})$ of the stiffness matrix from polynomial degree p.

We finish this subsection with a comparison of DoFs between the two approaches. Note that the data from Fig. 15 was computed on a full parametrization of the L-shape, with three initial quads. In Table 1 we see a direct comparison of the DoFs needed to obtain an error smaller than $5 \cdot 10^{-4}$. Note that T-splines need less DoFs for each polynomial degree.

Table 1 Number of DoFs needed to obtain an H^1 error smaller then $5 \cdot 10^{-4}$

p	3	4	5	6	7
IGA-DoFs	515	659	1057	1803	2681
FEM-DoFs	2278	1950	2762	3645	4870

6 Outlook

We have successfully implemented and demonstrated a framework to use isogeometric analysis within `deal.II`. Standard error estimators are given in the implementation for different Poisson-like problems, see (35). During applications, the main difficulty for the user is the correct application of Neumann boundary data for the error estimator. In the presented examples, it was necessary to use the explicit definition of the geometric mapping to define the normals on the boundary. This will be automated in future extensions.

We only considered scalar elliptic PDEs in this work, however, in the next step the focus will also shift to vector valued problems, e.g. for linear elasticity. Once this is done, `deal.II` allows us to extend the results to almost arbitrary PDEs of order two.

Further, note that the considered problems are defined on single patch domains. We will also focus on an implementation of multiple patched domains to apply T-splines with `deal.t` in a real-world setting with vector valued problems. The source-code is available at [26] with instructions to reconstruct the given examples.

A Bézier extraction algorithm of TSplines

The code stated in Algorithm 5 is a direct application of the algorithm provided by [34] with two corrections. The function below computes the extraction operator rows used for a single B-Spline over each knot-interval and inserts additionally some interior knots provided by `knot_in`. The function `compute_extended_kv()` returns the extended knot vector of a given *local* knot vector by adding the first and last knots until multiplicity `p` is reached, e.g. `compute_extended_kv([0 0 1 2])` returns `Ubar = [0 0 0 1 2 2 2]`, `nf = 1` for one insertion at the front, and `ne = 2` for two insertions at the end.

Originally, the special case for the multiplicity of a knot (mult == p) has not been handled, see Algorithm 65 and fol-

lowing, as well as properly initializing the new extraction operator row, see Algorithm 36. The code can also be found online at [36].

Algorithm 5 Bezier extraction of B-Splines with knot insertions

```
function [C, nf] = bezier_extraction( xi, knot_in)

% Ensure the knot vector is non-
% decreasing:
xi = sort(xi);
p = length(xi) - 2;
m = length(knot_in);
[Ubar, nf, ne] = compute_extended_kv(xi);
a = p+1;
b = a+1;
C = zeros(1, p+1);
C(1, nf+1) = 1;
mbar = ne + 2 + nf + m;
si = 1;
nb = 1;
while (b < mbar)
    % Initialize extraction operator
    C(end+1, :) = zeros(1, p+1);
    add = 0;
    % Check if additional knot needs
    % to be inserted:
    if (si <= m && Ubar(b) >= knot_in(si))
        mult = 0;
        add = 1;

        Ubar = [Ubar(1:b-1) knot_in(si) Ubar(b:end)];
        si = si+1;
    else
        i = b;
        while (b < mbar && Ubar(b+1) == Ubar(b))
            b = b + 1;
        end
        mult = b - i + 1;
    end % if ( si )
    % Initialize extraction row
    C(end, nf + 2 - nb + (si-1) - mult - add) = 1;
    if (mult < p)
        numer = Ubar(b) - Ubar(a);
        alphas = zeros(1, p - mult);
        for j = p:-1:mult+1
            alphas(j - mult) = numer / (Ubar(a + j + add) - Ubar(a));
        end % for ( j )
        r = p - mult;
        %Update Matrix coefficients:
        for j = 1 : r
            save = r - j + 1;
            s = mult + j;
            for k = p+1 : -1 :s+1
                alpha = alphas(k - s);
                C(nb, k) = alpha * C(nb, k) + (1 - alpha)*C(nb, k-1);
            end % for ( k )

            if b < mbar
                % Update overlapping
                % coefficients
                C(nb + 1, save) = C(nb, p + 1);
            end % if ( b )
        end % for ( j )
        % Update coefficients for
        % next iteration
        if (b < mbar)
            a = b;
            b = b + 1;
        end % if ( b )
    else
        % Special case: mult = p
        % => There are only two
        & elements given by the
        % right-most, resp. left-most,
        % bezier splines
        C(nb + 1, 1) = 1;
        a = b;
        b = b+1;
    end
    % Advance to next element
    nb = nb + 1;
end
```

B Important deal.II data structures

To fully understand parts of Algorithm 3 and 4 it is important to learn deal.II internal indexing of quads, lines, and vertices of a cell. For a fixed cell $\mathtt{Q}$ the vertices $\mathtt{V}_i$ are numbered in lexicographic ordering, i.e.

$$\mathtt{V}_i < \mathtt{V}_j :\Longleftrightarrow \tag{56}$$

$$\begin{cases} \mathtt{V}_{i,d} < \mathtt{V}_{j,d}, & \text{if } \mathtt{V}_{i,d} \neq \mathtt{V}_{j,d}, \text{ else} \\ \mathtt{V}_{i,d-1} < \mathtt{V}_{j,d}, & \text{if } \mathtt{V}_{i,d-1} \neq \mathtt{V}_{j,d-1}, \text{ else} \\ \vdots \\ \mathtt{V}_{i,1} < \mathtt{V}_{j,1}. \end{cases} \tag{57}$$

Its faces $\mathtt{F}_i$ are ordered according to its outward-pointing normals, i.e.

$$\mathtt{F}_i < \mathtt{F}_j :\Longleftrightarrow n_{\mathtt{Q}}(\mathtt{F}_i) < n_{\mathtt{Q}}(\mathtt{F}_j), \tag{58}$$

where $n_{\mathtt{Q}}(\mathtt{F})$ is the outward pointing normal of face $\mathtt{F}$ from cell $\mathtt{Q}$, e.g. for a 2D reference cell we get the normal vectors $-e_1, +e_1, -e_2, +e_2$ (in that order).

This defines the ordering of vertices, lines in 2D, and quads in 3D. The order of lines in 3D is as follows. From a cell $\mathtt{Q}$, we first enumerate the lines of the face $\mathtt{F} = \mathtt{Q}_1 \times \mathtt{Q}_2 \times \{\inf \mathtt{Q}_3\}$ using 2D line ordering, resp. (58). Then we enumerate the lines of face $\mathtt{F} = \mathtt{Q}_1 \times \mathtt{Q}_2 \times \{\sup \mathtt{Q}_3\}$ using 2D line ordering, and finally the lines in z-direction are enumerated in lexicographic ordering.

A visual explanation is given in Fig. 17.

Further, Algorithm 3 and 4 use neighbor relations between cells. A neighbor $\mathtt{neighbor}(\mathtt{Q}, \mathtt{face_no})$ of a cell $\mathtt{Q}$ at face $\mathtt{F}_{\mathtt{face_no}}$ is defined to have at maximum the level of refinement as the cell itself, i.e.

$$\mathtt{level}(\mathtt{Q}) \geq \mathtt{level}(\mathtt{neighbor}(\mathtt{Q}, \mathtt{face_no})). \tag{59}$$

Let $\mathtt{Q}_1$ and $\mathtt{Q}_2$ be two adjacent cells, with distinct refinement levels. If $\mathtt{Q}_1$ is coarser than $\mathtt{Q}_2$ and we ask for its neighbor on the other side of the common face $\mathtt{F} = \partial\mathtt{Q}_1 \cap \partial\mathtt{Q}_2$, then $\mathtt{parent}(\mathtt{Q}_2)$ with the same refinement level of $\mathtt{Q}_1$ is returned; if the parent of $\mathtt{Q}_2$ has a higher refinement, the parent of the parent is returned, and so on. On the other hand, if we ask for the neighbor of $\mathtt{Q}_2$ at face $\mathtt{F}$, simply $\mathtt{Q}_1$ is returned,

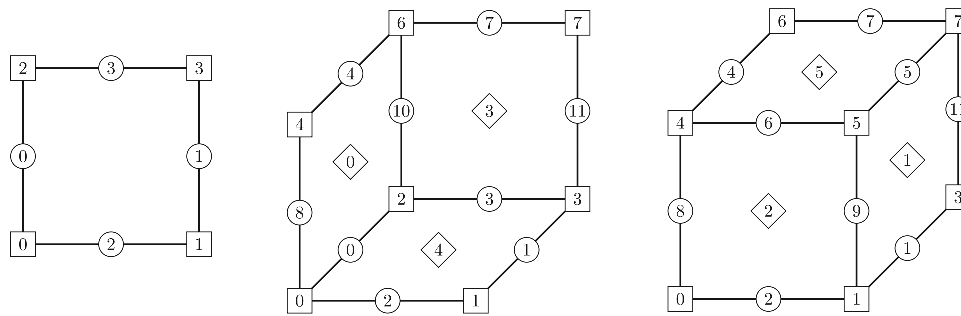


Fig. 17 `deal.II` internal enumeration of quads, lines, and vertices of a 2D cell (left) and a 3d cell (middle and right). Numbers in rectangles correspond to the index of the vertex at that position, numbers in circles correspond to the index of the lines at that position, and numbers in diamonds correspond to the index of the quad at that position. The numbering for the 3D cell has been split into two parts, which show the cell interior (middle) and the cell exterior (right)

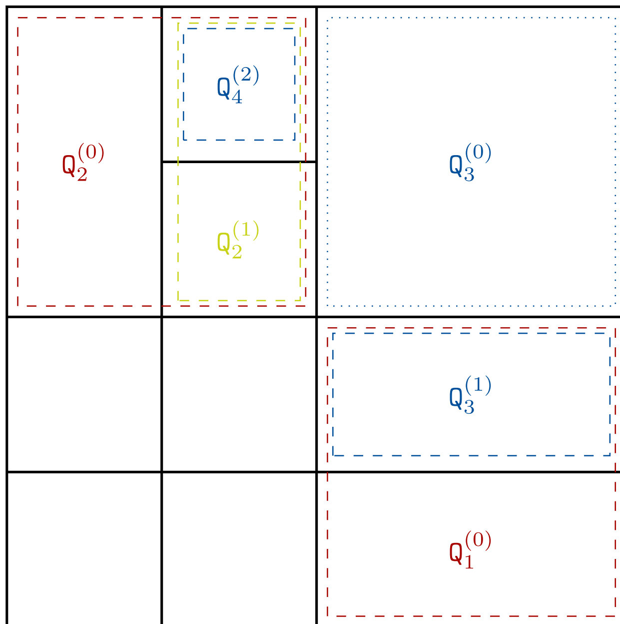


Fig. 18 `deal.II` neighbour relations of cells in 2D. All relevant cells are marked, $Q_i^{(\ell)}$ corresponds to the i-th cell on level ℓ. Explained are the neighbour relations of the adjacent cells $Q_4^{(2)}$ and $Q_3^{(0)}$, and $Q_3^{(1)}$ and $Q_3^{(0)}$, respectively

in which case neighbor and cell have different refinement levels.

A detailed example is given in Fig. 18. Consider first the two (active) adjacent cells $Q_3^{(0)}$ and $Q_4^{(2)}$. Obviously, for the first neighbor of $Q_4^{(2)}$ we get $Q_3^{(0)}$, i.e.

$$\texttt{neighbor}(Q_4^{(2)}, 1) = Q_3^{(0)}. \tag{60}$$

However, on the other side of face 0 from $Q_3^{(0)}$ are two distinct cells, and it is arbitrary to return either as zeroth neighbor. Instead of returning either of the children of $Q_2^{(1)}$, the parent with refinement level 0 is returned, that is the parent $Q_2^{(0)}$ of the parent $Q_2^{(1)}$ of the cell $Q_4^{(2)}$, and hence we get

$$\texttt{neighbor}(Q_3^{(0)}, 0) = Q_2^{(1)}. \tag{61}$$

Similar arguments for $Q_3^{(1)}$ and $Q_3^{(0)}$ lead to

$$\texttt{neighbor}(Q_3^{(0)}, 2) = Q_3^{(1)}. \tag{62}$$

C A tutorial for `deal.t`

We demonstrate the usage with the example from Sect. 5.1.

Since all `deal.II` and `deal.t` functions and classes are in a namespace `dealii`, resp. `dealt` we firstly import the respective namespaces into our program. with the following lines

```
using namespace dealii;
using namespace dealt;
```

We define a new class for each new application. Further classes are introduced to resemble abstract mathematical functions. This incorporates `deal.II`s `Function<dim, spacedim>` object that resembles an abstract base class from which every (mathematical) function should be derived from. Since this object is an abstract class, every derived class needs to define its own functions to calculate values, derivatives, etc.

The following subsections give the source code for the application to the above problem with sufficient explanations between relevant lines. Each source code is numbered within its own block, i.e. a function may be split in two or more parts, but the whole function is numbered consecutively line-by-line.

C.1 Writing a `C++` class for each problem

The data for the isogeometric mapping is stored within an additional object from the `deal.t` namespace, called |`IPF_Data<int dim, int spacedim>`|. It essentially stores only the provided (global) knot vectors, the corresponding degrees,

as well as the control points. For this purpose, the parametric dimension `dim` is two, as well as the physical dimension `spacedim`. Further, it is important to store the (parametric) mesh used for applications. This is the main part of `deal.t` and is called `|TS_Triangulation<int dim, int spacedim>|`

Depending on what the user wishes to accomplish, there is a variety of additional variables the user may choose to store. For this demonstration we intend to calculate

1. the stiffness matrix $\underline{K}_h$,
2. the right-hand-side $\underline{F}_h$
3. the solution vector $\underline{u}_h$, s.t. $\underline{K}_h \underline{u}_h = \underline{F}_h$,
4. the exact L_2-error $\|u - u_h\|_{L_2(\overline{\Omega})}$,
5. the exact H^1-error $\|u - u_h\|_{H^1(\overline{\Omega})}$, and
6. the residual error $\|\Delta u + f\|_{L_2(\mathsf{Q})}$ for each cell Q of the triangulation.

Since the size of the stiffness matrix is usually large, it is defined as a `dealii::SparseMatrix<double>`, which has to be initialized using a sparsity pattern.

The implementation is encapsuled in a separate namespace to avoid ambiguity with similar names along other namespaces. This is done by

```
namespace Poisson_Neumann {
```

The right-hand-side function f is stored in a separate class and derived from `deal.II` `|Function<dim, spacedim>|` as mentioned before, i.e.

```
class Poisson_RHS : public Function<2> {
public:
  Poisson_RHS() : Function<2>(1) {}
  ~Poisson_RHS() = default;

  virtual double value(
    const Point<2>&    p,
    const unsigned int component = 0
  ) const override;
};
```

To compute the exact L_2- and H^1-errors, we define a class for the exact solution which is again derived from `deal.II` `|Function<dim, spacedim>|`, but also needs to compute the gradient. Hence, its header declaration is extended by an override of the base gradient function.

```
class Poisson_SOL : public Function<2> {
public:
  Poisson_SOL() : Function<2>(1) {}
  ~Poisson_SOL() = default;

  virtual double value(
    const Point<2>&     p,
    const unsigned int  component = 0
  ) const override;

  virtual Tensor<1, 2> gradient(
    const Point<2>&     p,
    const unsigned int  component = 0
  ) const override;
};
```

We further need another object to describe values along the Neumann boundary Γ_N. This is done by yet another `deal.II` inherited function as above similar to `Poisson_RHS`, called `Poisson_NC` that will resemble the boundary values. The header declarations are the same as in `Poisson_RHS` with exchanged names; thus we will not display the code here. We will also skip the implementations of the values and gradients for these classes in this subsection.

For the main class, we get from the explanations above the following member variables.

```
class Poisson_Benchmark
{
private:
  unsigned int ref;
  unsigned int order;

  IPF_Data<2, 2>  data;

  TS_Triangulation<2, 2>  tria;

  SparsityPattern     sparsity_pattern;
  SparseMatrix<double>    system_matrix;

  Vector<double>  solution;
  Vector<double>  system_rhs;

  Vector<double>  l2_error;
  Vector<double>  h1_error;
  Vector<double>  mesh_size;
  Vector<double>  dofs_per_level;

  std::map<CellId, double>  cell_errors;

  Poisson_RHS     rhs_fcn;
  Poisson_SOL     sol_fcn;
  Poisson_NC      neumann_bc;
```

Here, `ref` determines the amount of refinement steps and `order` determines for which (global) degree we want to compute the solution. The remaining variables `mesh_size` and `dofs_per_level` store the smallest cell diameter, resp. the DoFs at level `l`. For the diameter we define

$$h_n = \min_{\mathsf{Q} \in \mathcal{T}^{(n)}} h_{\mathsf{Q}}, \tag{63}$$

where h_{Q} is the diameter of a cell Q defined in (37).

Next, we need to determine the methods, resp. functions, we want to employ. Firstly, we need a public constructor for the class that initializes every necessary variable. It should take the variables `ref`, and `order` and store them for use in the program. We declare the constructor as

```
  // Public functions:
public:
  Poisson_Benchmark(int ref, int order = 0);
```

To prevent mistakes on the user side, only a single function is allowed to be called outside the class. That function calls other functions from this class in the right order. It does not need any input arguments, as it is supposed to simply start the program. It is declared as

```
  void run();
```

Every other function is declared **`private`**. For a smooth initialization of the triangulation, we figured it is the best technique to write a function that returns the correct `IPF_Data<2, 2>`, i.e.

```
57      IPF_Data<2, 2> get_IPF_data();
```

The source for this function is omitted in this paper.

There are a few steps that have to be done after every refinement:

1. The system to be solved has to be setup to the correct dimensions,
2. The system has to be assembled,
3. The system has to be solved after assembly,
4. After the system is solved, we have to compute the L_2- and H^1-errors,
5. important quantities shall be printed to files
6. additionally, we may be interested in the point-wise error for each cell,
7. The cells have to be marked and refined for further computations.

These tasks are split into their respective subroutines,

```
58      void setup_system();
59      void assemble_system();
60      void solve();
61      void compute_h1_error();
62      void output_system();
63      void print_error();
64      void estimate_and_mark();
65    }; // End of Poisson_Benchmark class
66  } // End of namespace declaration
```

C.2 Creating the grid

Most of the variables introduced in subsection C.1 can be initialized before the actual source code is given. This follows the syntax `.A(int a): var(a)` and reads in the source code of the main class as

```
1  Poisson_Neumann::Poisson_Benchmark::
       Poisson_Benchmark(int ref, int order) :
2          ref(ref),
3          order(order),
4          data(this->get_IPF_data()),
5          tria(data),
6          rhs_fcn(),
7          sol_fcn(),
8          neumann_bc()
9  {
```

Note that Algorithm 5 initializes the `TS_Triangulation`.

Firstly, we increase the initial polynomial degree according to the user input across every direction using

```
10    tria.degree_elevate_global(order);
```

There is also a function `degree_elevate(int d, int times)` that increases the polynomial degree only in a specified direction. Note that the implementation makes no restrictions on the polynomial degrees, i.e. anisotropic polynomials are a possibility.

Next, we will define the boundary indicators using `deal.II` internal iterators over faces. From the base class `Triangulation<dim, spacedim>` we can use inherited functions for our derived class `TS_Triangulation<dim, spacedim>`. Figure 8 shows which parametric boundary is mapped to which physical boundary. From this, and the definition of Ξ_x and Ξ_y we get that the parametric Neumann boundary is given at faces whose second coordinate is either zero or one. Thus, the source code for the boundary indicators is

```
11    for (auto& face : tria.active_face_iterators
          ()){
12      const Point<2>& c = face->center();
13      if (face->at_boundary() &&
14            (c(1) == 0 || c(1) == 1 ) )
15        face->set_boundary_id(1);
16    }
```

Where an iterator can be thought of as a pointer to some object resembling an active face. It is dereferenced using the arrow notation `->`. Note that initially every boundary id of each face is given by zero.

In standard finite element code, boundary DoFs are recognized during assembly, as we define the same finite element functions for each cell. Using T-Splines this is not possible, and in fact we have to preemptively determine every spline that has non-zero values on the boundary. This is done in the following line

```
17    tria.set_boundary_dofs();
```

where the boundary DoFs are sorted by boundary ids, i.e. internally within `tria` we save a container `boundary_dofs` that encodes a boundary id to a list of boundary DoF indices.

Since we wish to perform computations on this mesh, we have to tell the program that it should switch from the parametric mesh to the parametric Bézier mesh with a call to

```
18    tria.refine_bezier_elements();
```

At this point, this function essentially just sets a `bool` which enables functionality we will use for matrix assembly. However, later on this will perform one additional refinement step by subdividing active cells along T-junction extensions. Since the program is now on the parametric Bézier mesh, it can compute extraction operators used for Bézier extraction. This information is computed using

```
19    tria.compute_extraction_operators();
```

After this, we can initialize the other quantities of interest to the right size as follows

```
20    l2_error.reinit(ref + 1);
21    h1_error.reinit(ref + 1);
22
23    mesh_size.reinit(ref + 1);
24    dofs_per_level.reinit(ref + 1);
25  } // end of constructor
```

C.3 System setup

The routine

```
1  void Poisson_Neumann::Poisson_Benchmark::
       setup_system(){
```

should initialize the right-hand-side vector F_h, the solution vector u_h, and the stiffness matrix K_h to the correct size. Remember that to this extent the finite element space is $\{T_i \circ \Phi^{-1}\}_i$ for each T-Spline T_i and that the support of each T-Spline spans multiple parametric cells. The number of active splines of the mesh thus sets the dimensions of the system. Since F_h and u_h are simply vectors, they are initialized with

```
system_rhs.reinit(tria.n_active_splines());
solution.reinit(tria.n_active_splines());
```

The stiffness matrix has only non-zero entries (i, j), whenever $\operatorname{supp} T_i \cap \operatorname{supp} T_j \neq \emptyset$. This information is cell-wise stored within `IEN_array` and is returned by

```
const std::map<
  TriaIterator<::CellAccessor<2,2>>,
  std::vector<unsigned int>
  >& IEN_array =
      tria.get_IEN_array();
```

For a given `cell` it returns which global DoF has support on it. There is also an overload of this function that takes a `cell` as input, to return the vector of local DoFs on the specified cell. This will be used in the assembly routine.

Next, the sparsity pattern of the sparse stiffness matrix has to be defined. From `deal.II`, a `SparsityPattern` object can be reinitialized using a call to `reinit(int n, int m, int nnz)`, where `n` corresponds to the rows of the sparse matrix, `m` corresponds to the columns of the sparse matrix and `nnz` correspond to the maximum number of non-zero support on each row. Unfortunately, we are lacking a proper estimate on this quantity, hence we set this number to the maximum, i.e.

```
sparsity_pattern.reinit(
    tria.n_active_splines(),
    tria.n_active_splines(),
    tria.n_active_splines() );
```

We then have to add information about which entries are actually non-zero. This is done using `add(int i, int j)` that allows to add entries at position (`i`, `j`). The complete information over all non-zero supports is given by a few nested for-loops

```
for (const auto& [_, arr] : IEN_array)
  for (unsigned int i : arr)
    for (unsigned int j : arr)
      sparsity_pattern.add(i, j);
```

Before we use this pattern to initialize the system matrix K_h, it has to free superfluous space by compressing the data as much as possible. Only then can it be used to define the system matrix

```
  sparsity_pattern.compress();
  system_matrix.reinit(sparsity_pattern);
} // end of setup_system
```

C.4 Matrix assembly and solving

The main part of the implementation is the matrix assembly in

```
void Poisson_Neumann::Poisson_Benchmark::
    assemble_system(){
```

The implementation of T-Splines is based on Bézier extraction. We can then define tables for the Bernstein polynomials on the reference cell $Q^{(ref)} = [0, 1]^d$ a-priori and use these tables to define the values of a specific T-spline on a cell. Values on the reference cell are computed using Gaussian quadrature rules. Before we start the computations, we thus have to generate objects that represents the finite element values on cells, resp. faces. This is done using the `TSValues` and `TSFaceValues` objects. They mimic the behavior of their `deal.II` variants as explained in 4, and are initialized in the application as

```
  std::vector< unsigned int > degrees = tria.
      get_degree();
 TSValues<2> ts_values(
      {degrees[0] + 1,
       degrees[1] + 1},
      update_values |
      update_gradients |
      update_quadrature_points |
      update_JxW_values);
 TSFaceValues<2> face_values({}
      {degrees[0] + 1,
       degrees[1] + 1},
      update_values |
      update_gradients |
      update_quadrature_points |
      update_normal_vectors |
      update_JxW_values);
```

Since it is costly, to write into the whole stiffness matrix K_h, we define a smaller matrix that stores the integral values for a specific cell together with a smaller right-hand-side vector.

```
  const unsigned int dofs_per_cell = TSValues::
      n_dofs_per_cell();
  FullMatrix<double> cell_matrix(dofs_per_cell,
       dofs_per_cell);
  Vector<double>     cell_rhs(dofs_per_cell);
```

Afterwards we can assemble the stiffness matrix cell-by-cell using a loop over cell iterators of the triangulation

```
  for (const auto& cell : tria.
      active_cell_iterators()){
```

At the beginning of each iteration, we have to get the corresponding values of T-splines on this `cell` with

```
    ts_values.reinit(cell);
```

and reset the small cell matrix and rhs with

```
    cell_matrix = 0;
    cell_rhs    = 0;
```

Then we start the quadrature sum with

```
    for (const unsigned int q :
        ts_values.quadrature_point_indices()){
```

We initially store

```
      double dx = ts_values.JxW(q);
```

as it is used in two distinct places. To compute the cell-matrix at the given quadrature point `q`, we have to calculate

$$\sum_q \nabla T_i(q) \cdot \nabla T_j(q) \mathrm{d}x \tag{64}$$

and add that to position i and j of K_h. This will be done later, instead we store it in the cell matrix using local DoF indices.

```
      for (const unsigned int i :
            ts_values.dof_indices())
        for(const unsigned int j :
            ts_values.dof_indices())
          cell_matrix(i,j) +=
            ts_values.shape_grad(i, q)      //
              grad T_i(q)
            * ts_values.shape_grad(j, q)    //
              grad T_j(q)
            * dx;
```

Further, the values to the local right-hand-side have to be computed using

$$\sum_q T_i(q) f(\Phi(q)) \mathrm{d}x, \tag{65}$$

which is done in a similar loop

```
      const Point<2>& mapped_q =
        ts_values.quadrature_point(q); // Phi(q
          )
      for (const unsigned int i :
            ts_values.dof_indices())
        cell_rhs(i) +=
            ts_values.shape_value(i, q) // T_i(
              q)
            * rhs_fcn.value(mapped_q)    // f(
              Phi(q))
            * dx ;
    } // for ( q )
```

and ends the quadrature loop. To get the Neumann values, we have to check, whether or not the cell is on the boundary and then check if a face of that cell is at the Neumann boundary. For this purpose, we have set the boundary indicators during the creation of the mesh. The code-block for the Neumann values then becomes similar to the code-block for a cell, where we calculate

$$\sum_q g(q) T_i(q) \mathrm{d}x \tag{66}$$

with

$$g(q) = \nabla f(q) \cdot n(q), \tag{67}$$

where $n(q)$ is the normal vector at quadrature point q. This gives

```
    if (cell -> at_boundary()){
      for (unsigned int f = 0;
            f < GeometryInfo<2>::faces_per_cell
              ;
            f++){
        if (cell
            ->face(f)
            ->at_boundary()
            && cell
            ->face(f)
            ->boundary_id() == 1){
          face_values = reinit(cell,f);
          for (const unsigned int q :
        face_values.quadrature_point_indices())
            {
            const long double g =
                sol_fcn.gradient(
                  face_values.quadrature_point(
                    q)) *
                  face_values.normal_vector(q);
            for (const unsigned int i :
                  face_values.dof_indices())
              cell_rhs(i) +=
                g * face_values.shape_value(i,
                    q)
                  * face_values.JxW(q);
          }
        } // if ( neumann_bc )
      } // for ( f )
    } // if (cell -> at_boundary())
```

This almost ends the cell-loop. We still have to add the computed values to the global system. For this, we need information about the global indices of the used local indices. The standard in `deal.II` is to use a function `copy_local_to_global()` that does the trick automatically together with applying Dirichlet boundary conditions, however, it assumes background information we cannot provide at this point. Hence, we have to do it manually. The local cell information is copied to the global matrix using the `IEN_array` as before, with

```
   std::vector< unsigned int >
       local_dof_indices =
            tria.get_IEN_array(cell);

    system_matrix.add(local_dof_indices,
        cell_matrix);
    system_rhs.add(local_dof_indices, cell_rhs)
        ;
  } // for ( cell )
```

To end the matrix assembly, we have to apply Dirichlet conditions. In this case, we have homogeneous boundary conditions, which simplifies the process. For Dirichlet boundary DoFs we simply set the corresponding column and row to 0, add a 1 to the diagonal and set the right-hand-side to 1:

```
  const auto& boundary_dofs =
          tria.get_boundary_dofs();

  unsigned int n_global_dofs = tria.
      n_active_splines();
  for (const auto& dof : boundary_dofs.at(0)){
    for (unsigned int j = 0; j < n_global_dofs;
          j++){
      system_matrix.set(dof, j, 0);
      system_matrix.set(j, dof, 0);
    }

    system_rhs(dof) = 0;
    system_matrix.set(dof, dof, 1);
  }
} // end assemble_system()
```

To solve the system, we use a standard `deal.II` solver. We have to define an object `SolverControl` that defines the maximum number of iterations and a tolerance and give it to the constructor of a solver. For this application, we chose

a CG-solver. The `solve()` function of the solver takes the stiffness matrix $\underline{K}_h$, the solution vector $\underline{u}_h$, the right-hand-side $\underline{F}_h$, and a precondition matrix which is chosen to be the diagonal of $\underline{K}_h$, i.e. a Jacobi-type preconditioner. The complete routine is

```
void Poisson_Neumann::Poisson_Benchmark::solve
    (){
  SolverControl solver_control(
        750 * tria.n_active_splines(),
        1e-6
    );
  SolverCG<Vector<double>> solver(
        solver_control
    );
  PreconditionJacobi<SparseMatrix<double>>
      preconditioner;
  preconditioner.initialize(system_matrix);
  solver.solve(
        system_matrix,
        solution,
        system_rhs,
        preconditioner
    );
} // end solve()
```

C.5 Error estimation and marking

The residual error estimation from

```
void Poisson_Neumann::Poisson_Benchmark::
    estimate_and_mark(){
```

defines a list of – possibly in-active – cells to be marked for refinement

```
    std::vector<
        TriaIterator<
            ::CellAccessor<2, 2>
        >
    > mark;
```

Note that matrix assembly, error calculations and error estimations are performed on the parametric Bézier mesh $\mathcal{B}(\Omega)$, while the mesh refinement itself *has to be performed* on the parametric mesh $\mathcal{T}(\Omega)$. This stems from the definition of anchors and their knot vectors on the parametric mesh. Finite element values, however, are given on the parametric Bézier mesh. If the error estimator detects a parametric Bézier cell to be refined, instead we save its parent in the temporary list `mark`. A special case occurs if there is only a single active cell. In that case, we simply skip error estimation and save the single cell

```
  if (tria.n_active_cells() == 1) {
    mark.push_back(tria.begin_active());
  } else {
```

Otherwise, we need to access the local residuals for each cell. The core implementation of `deal.t` offers some functions to calculate the residual errors of Poisson-like problems, see Sect. 4.1.

For this example, we have $\sigma \equiv 1$ with in-homogeneous Neumann boundary conditions. The list of local residuals is simply defined by

```
    std::map<
        TriaIterator<
            ::CellAccessor<2, 2>
        > , double>
    local_residuals;
```

that will later return the residual of a cell iterator `cell` by simply calling `local_residuals[cell]`. Further, the Neumann data is stored in a similar list, that encodes a boundary id to a given function. This enables the error computations of problems with multiple Neumann functions g_i^N at different Neumann boundaries Γ_i^N. For our purpose we thus get

```
    std::map<
        types::boundary_id,
        Function<2>* >
    neumann_data = {{1, &neumann_bc}};
```

Note that we use a pointer to the different boundary conditions. We then call the corresponding error estimator with

```
    const std::vector<unsigned int>& degrees =
        tria.get_degree();
    tria.poisson_residual_error_estimate(
        {degrees[0]*degrees[0] + 1,
         degrees[1]*degrees[1] + 1},
        &rhs_fcn,
        neumann_data,
        solution,
        local_residuals
    );
```

The marking strategy is explained in detail in Sect. 5.1. For this, we define a list of all cells for which we have a local residual error, i.e. every active cell, then sort the list in descending order, such that the local residual of cell `i` is greater than the local residual of cell `j` if `i<j`. We get

```
    std::vector<
        TriaIterator<
            ::CellAccessor<2, 2>
        >
    > cell_list;
    for (const auto& [c, _] :
            local_residuals)
      cell_list.push_back(c);

    std::sort(cell_list.begin(),
            cell_list.end(),
        [local_residuals](
            const auto& c1,
            const auto& c2){
      return local_residuals.at(c1)
            > local_residuals.at(c2);
    });
```

Next, we need a list that tells us which cell of the mesh was refined to obtain the parametric Bézier mesh. These elements are stored separately within a `TS_Triangulation` and can be obtained by `get_bezier_elements()`. It returns a `std::vector` that stores (in-active) parent cells of refined cells during `refine_bezier_elements()`. The total amount of active parametric cells (without refined Bézier cells) is given by the number of active cells subtracting the number of Bézier cells. Before we push a cell into the container of marked elements, we have to check if its parent has been refined during the transition from parametric mesh to the parametric Bézier mesh.

```
    const auto& bezier =
            tria.get_bezier_elements();
    const double percentile = 0.075;
    const unsigned int n_cells =
        cell_list.size() - bezier.size();
    const unsigned int n_mark  =
        std::ceil(n_cells * percentile);

    for (unsigned int i = 0;
             i < n_cells &&
             mark.size() < n_mark;
             i++){
      if (cell_list[i] -> level() == 0)
        mark.push_back(cell_list[i]);
      else if (
        std::find(bezier.begin(), bezier.end(),
             cell_list[i] -> parent()) !=
        bezier.end()
      ){
        mark.push_back(cell_list[i] -> parent()
            );
      } else {
        mark.push_back(cell_list[i]);
      }
    }
  } // if ( special case )
```

The next few calls are essential when working with our code. We can not simply refine the marked cells, since there may be parent elements in this list. Further, we already stressed the fact that refinement is only possible on the parametric mesh. Hence, we have to coarsen the previously refined Bézier cells with a call to

```
  tria.coarsen_bezier_elements();
```

This additionally sets a boolean within `TS_Triangulation` that allows further refinements. The user may now decide whether to set the refinement flags manually, or let the `TS_Triangulation` set the refine flags of the cells. We encourage the user to use the line

```
  tria.set_refine_flags(mark);
```

As there are some internal processes at the setup of a `TS_Triangulation` which the user has to consider when setting refinement flags manually, see section 4.

Now, that cells are marked for refinement, we can perform refinement. Note that the program will usually refine not only the marked cells, but also computes its *coarse neighbourhood* from [24] to assure analysis suitability, and hence linear independence, of T-Splines, see Sect. 2.2. The refinement process is then essentially carried out using

```
  tria.execute_coarsening_and_refinement();
```

where the algorithms described in Sect. 2.2 are used to determine the next level. Note that, despite the name, coarsening of cells is not yet implemented—the name for this function had to be inherited from the base class.

Next, we first use the parametric mesh, to define the list of boundary DoFs, as in the initialization of this mesh

```
  tria.set_boundary_dofs();
```

then, we have to switch back to the parametric Bézier mesh using the opposite function of `coarsen_bezier_elements()`

```
  tria.refine_bezier_elements();
```

that essentially performs another refinement step of the mesh. Finally, we have to recompute the extraction operators for the new T-Splines with

```
  tria.compute_extraction_operators();
} // end estimate_and_mark()
```

This ends the `estimate_and_mark()` routine.

The most important function

```
void Poisson_Neumann::Poisson_Benchmark::run(){
```

will then call these functions in the correct order, i.e.

```
  while (tria.n_levels() - 1 <
            ref + 1){
    this -> setup_system();
    this -> assemble_system();
    this -> solve();
    this -> output_system();
    this -> print_error();
    this -> compute_h1_error();
    this -> estimate_and_mark();
  }
  // Print commands for
  // h1-, l2-error, and
  // other quantities
  ...
} // end run()
```

where the function `print_error()` prints the point-wise error to some files, `output_system()` prints the stiffness matrix, the right-hand-side, the solution, the mapped (Bézier) mesh wireframe, and the mapped quadrature points on the physical domain, i.e. the mapping Φ, to distinct files, and `compute_h1_error()` computes the H^1 and L_2 error of the solution to another file. We do not use `deal.II` functions to output the mesh with a `GridOut` object, as it again imposes conditions on the system matrix we cannot provide at this point, e.g. the dimension of the matrix being $n_B \cdot n_C$, where n_B is the number of Bernstein polynomials on the reference cell and n_C is the number of active cells.

The errors computed in `print_error()` and `compute_h1_error()` are computed using a similar technique as for the matrix assembly in `assemble_system()` and are hence not given here. Further, the output commands from `output_system` are rather tedious and not considered important, and hence also not given here.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Financial or non-financial interests The first author acknowledges the Deutsche Forschungsgemeinschaft (DFG) under Germany Excellence Strategy within the Cluster of Excellence PhoenixD (EXC 2122, Project ID 390833453). The authors have no relevant financial or non-financial interests to disclose.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as

long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Hiniborch R, Morgenstern P (2023) Multivariate analysis-suitable t-splines of arbitrary degree. Comput Methods Appl Math. https://doi.org/10.1515/cmam-2022-0071
2. Logg A, Mardal K-A, Wells GN (2012) Automated Solution of Differential Equations by the Finite Element Method, vol 84, 1st edn. Lecture Notes in Computational Science and Engineering. Springer, Heidelberg
3. Kirby RC, Logg A (2006) A compiler for variational forms. ACM Trans Math Softw 32(3):417–444. https://doi.org/10.1145/1163641.1163644
4. Arndt D, Bangerth W, Davydov D, Heister T, Heltai L, Kronbichler M, Maier M, Pelteret J-P, Turcksin B, Wells D (2021) The deal.II finite element library: design, features, and insights. Comput Math Appl 81:407–422. https://doi.org/10.1016/j.camwa.2020.02.022
5. Jüttler B, Langer U, Mantzaflaris A, Moore S, Zulehner W (2014) Geometry + simulation modules: Implementing isogeometric analysis. Proc. Appl. Math. Mech. 14(1):961–962 Special Issue: 85th GAMM, Erlangen 2014
6. Mantzaflaris A (2020) An overview of geometry plus simulation modules. In: Mathematical Aspects of Computer and Information Sciences, pp. 453–456. Springer, Cham. https://doi.org/10.1007/978-3-030-43120-4_35
7. Dalcin L, Collier N, Vignal P, Côrtes AMA, Calo VM (2016) Petiga: A framework for high-performance isogeometric analysis. Comput Methods Appl Mech Eng 308:151–181. https://doi.org/10.1016/j.cma.2016.05.011
8. Nguyen VP, Anitescu C, Bordas SPA, Rabczuk T (2015) Isogeometric analysis: An overview and computer implementation aspects. Math Comput Simul 117:89–116. https://doi.org/10.1016/j.matcom.2015.05.008
9. Kamensky D, Bazilevs Y (2019) tigar: Automating isogeometric analysis with fenics. Comput Methods Appl Mech Eng 344:477–498. https://doi.org/10.1016/j.cma.2018.10.002
10. Hughes TJR, Cottrell JA, Bazilevs Y (2005) Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. Comput Methods Appl Mech Eng 194(39):4135–4195. https://doi.org/10.1016/j.cma.2004.10.008
11. Piegl L, Tiller W (1997) The NURBS Book, 2nd edn. Monographs in visual communication. Springer, Heidelberg. https://doi.org/10.1007/978-3-642-97385-7
12. Forsey DR, Bartels RH (1988) Hierarchical b-spline refinement. SIGGRAPH. Comput Graph 22(4):205–212. https://doi.org/10.1145/378456.378512
13. Forsey DR, Bartels RH (1995) Surface fitting with hierarchical splines. ACM Trans Graph 14(2):134–161. https://doi.org/10.1145/221659.221665
14. Giannelli C, Jüttler B, Speleers H (2012) THB-splines: The truncated basis for hierarchical splines. Computer Aided Geometric Design. 29(7):485–498. https://doi.org/10.1016/j.cagd.2012.03.025 Geometric Modeling and Processing 2012
15. Sederberg TW, Zheng J, Bakenov A, Nasri A (2003) T-splines and T-NURCCs. ACM Trans Graph 22(3):477–484. https://doi.org/10.1145/882262.882295
16. Bazilevs Y, Calo VM, Cottrell JA, Evans JA, Hughes TJR, Lipton S, Scott MA, Sederberg TW (2010) Isogeometric analysis using t-splines. Computer Methods in Applied Mechanics and Engineering 199(5):229–263. https://doi.org/10.1016/j.cma.2009.02.036 Computational Geometry and Analysis
17. Dörfel MR, Jüttler B, Simeon B (2010) Adaptive isogeometric analysis by local h-refinement with T-splines. Computer Methods in Applied Mechanics and Engineering. 199(5):264–275. https://doi.org/10.1016/j.cma.2008.07.012 Computational Geometry and Analysis
18. Buffa A, Cho D, Sangalli G (2010) Linear independence of the T-spline blending functions associated with some particular T-meshes. Comput Methods Appl Mech Eng 199(23):1437–1445. https://doi.org/10.1016/j.cma.2009.12.004
19. Analysis-suitable T-splines are dual-compatible (2012) Beirão da Veiga, L., Buffa, A., Cho, D., Sangalli, G. Computer Methods in Applied Mechanics and Engineering 249–252:42–51. https://doi.org/10.1016/j.cma.2012.02.025 Higher Order Finite Element and Isogeometric Methods
20. Dokken T, Lyche T, Pettersen KF (2013) Polynomial splines over locally refined box-partitions. Comput Aided Geometric Design 30(3):331–356. https://doi.org/10.1016/j.cagd.2012.12.005
21. Johannessen KA, Kvamsdal T, Dokken T (2014) Isogeometric analysis using lr b-splines. Comput Methods Appl Mech Eng 269:471–514. https://doi.org/10.1016/j.cma.2013.09.014
22. Beirão da Veiga L, Buffa A, Sangalli G, Vázquez R (2013) Analysis Suitable T-splines of arbitrary degree: Definition, linear independance and approximation properties. Mathematical Models and Methods in Applied Sciences 23(11), 1979–2003. https://doi.org/10.1142/S0218202513500231
23. Morgenstern P (2016) Globally structured three-dimensional analysis-suitable T-splines: Definition, linear independence and m-graded local refinement. SIAM J Numer Anal 54(4):2163–2186. https://doi.org/10.1137/15M102229X
24. Morgenstern P (June 2017) Mesh refinement strategies for the adaptive isogeometric method. PhD thesis, Friedrich-Wilhelm-University Bonn. https://hdl.handle.net/20.500.11811/7237
25. Buffa A, Gantner G, Giannelli C, Praetorius D, Vázquez R (2022) Mathematical foundations of adaptive isogeometric analysis. Arch Comput Methods Eng 29(7):4479–4555. https://doi.org/10.1007/s11831-022-09752-5
26. Hiniborch R (2024) deal.t: An implementation of TSplines within deal.II. Zenodo. https://doi.org/10.5281/zenodo.7994627
27. Li X, Zheng J, Sederberg TW, Hughes TJR, Scott MA (2012) On linear independence of T-spline blending functions. Computer Aided Geometric Design 29(1), 63–76. https://doi.org/10.1016/j.cagd.2011.08.005. Geometric Constraints and Reasoning
28. da Veiga LB, Buffa A, Sangalli G, Vázquez R (2014) Mathematical analysis of variational isogeometric methods. Acta Numerica 23:157–287. https://doi.org/10.1017/S096249291400004X
29. Mantzaflaris A (2021) ..., others (see website): G+Smo (Geometry plus Simulation modules) v21.12. http://github.com/gismo
30. Scroggs MW, Dokken JS, Richardson CN, Wells GN (2022) Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. ACM Transactions on Mathematical Software. https://doi.org/10.1145/3524456. To appear
31. Scroggs MW, Baratta IA, Richardson CN, Wells GN (2022) Basix: a runtime finite element basis evaluation library. Journal of Open Source Software 7(73), 3982. https://doi.org/10.21105/joss.03982
32. Arndt D, Bangerth W, Blais B, Fehling M, Gassmöller R, Heister T, Heltai L, Köcher U, Kronbichler M, Maier M, Munch P, Pelteret J-P, Proell S, Simon K, Turcksin B, Wells D, Zhang J (2021) The

deal.II library, version 9.3. Journal of Numerical Mathematics 29(3), 171–186. https://doi.org/10.1515/jnma-2021-0081

33. Borden MJ, Scott MA, Evans JA, Hughes TJR (2011) Isogeometric finite element data structures based on bézier extraction of nurbs. Int J Numer Methods Eng 87(1–5):15–47. https://doi.org/10.1002/nme.2968
34. Scott MA, Borden MJ, Verhoosel CV, Sederberg TW, Hughes TJR (2011) Isogeometric finite element data structures based on bézier extraction of T-splines. Int J Numer Methods Eng 88(2):126–156. https://doi.org/10.1002/nme.3167
35. Cottrell JA, Hughes TJR, Bazilevs Y (2009) Isogeometric Analysis. John Wiley & Sons, Ltd, West Sussex. https://doi.org/10.1002/9780470749081
36. Hiniborch R (2023) Correction on Bezier extraction of TSplines. Zenodo. https://doi.org/10.5281/zenodo.7994448

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.