Computers & Graphics 129 (2025) 104237

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on SIBGRAPI 2023 Tutorials

A fast high-dimensional continuation hypercubes algorithm ☆

Lucas Martinelli Reia [a,*], Marcio Gameiro [b], Tomás Bueno Moraes Ribeiro [a], Antonio Castelo [a]

[a] Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, Avenida Trabalhador São-carlense 400, São Carlos, 13566-590, São Paulo, Brazil

[b] Department of Mathematics, Rutgers University, 110 Frelinghuysen Road, Piscataway, 08854-8019, NJ, USA

ARTICLE INFO

Keywords:
Manifold
approximation/polygonization/tracing
Isomanifold
Triangulation
High-Dimensional Marching Cubes
Continuation method

ABSTRACT

This paper introduces the Fast Continuation Hypercubes (FCH) algorithm, a method for generating piecewise linear approximations of implicitly defined manifolds of arbitrary dimension. By integrating and mixing key aspects of existing approaches, the FCH algorithm offers significant improvements in both speed and memory efficiency. It traverses the domain by generating and processing only the necessary cells, which reduces the computational cost associated with high-dimensional manifold approximation. Additionally, the algorithm stores only the cells at the boundary of the traversed region, further optimizing memory efficiency. Experimental results demonstrate that FCH outperforms state-of-the-art algorithms in terms of runtime and memory usage.

1. Introduction

Implicitly defined manifolds (level sets) appear in many contexts in the fields of mathematics and computer graphics, especially in the form of iso-valued curves and surfaces. Consequently, many algorithms and data structures have been proposed to tackle the problem of creating a representation of these shapes in a way that makes them easier to visualize and manipulate [1,2]. These methods are widely used in applications in optimization problems [3,4], finite element calculations [5], data visualization [6–8], surface reconstruction [9–12], constructive solid geometry [13,14], surface learning [15–17], among others.

One of the most universally used approaches is to represent a level set using a piecewise-linear approximation, where a cell decomposition of the domain is used and, in each cell of this decomposition, the level set is approximated by a geometric shape composed of vertices, edges, and faces. In the literature, this process has been known by various terms, such as polygonization [18], contouring [19], tracing [20], and isosurfacing [21].

A well-known algorithm is the *Marching Cubes* [22], which generates a triangular mesh in 3D representing an isosurface. In this algorithm, the 3D space is partitioned into cubes, forming a three-dimensional grid, and each cube is processed individually by examining only its vertices (to decide whether they are on the same or opposite sides of the surface). Based on the configuration of these vertices, a lookup table is consulted to determine how to connect the vertices to create the triangles inside the cubes. These triangles are then glued along their shared edges to form the resulting mesh. Alternatively, there are also methods that decompose the domain into tetrahedra instead of cubes, such as the *Marching Tetrahedra* [23–25]. Compared to cube-based methods, tetrahedron-based methods perform simpler numerical evaluations, although the space is segmented into a considerably larger number of parts.

The mathematical formalization of an implicitly defined level set is that of a *manifold*: an implicitly defined manifold $\mathcal{M}$ of dimension $(n-k)$ is defined as a level set of a function $F:\mathbb{R}^n \to \mathbb{R}^k$, with $n > k$. That is, $\mathcal{M} = \{x \in \mathbb{R}^n \mid F(x) = c\}$ for some $c \in \mathbb{R}^k$ (see Section 2 for details). In the literature, there are many methods tailored for the 3D space, but substantially fewer more general methods that can be applied to manifolds of arbitrary dimensions, as some challenges arise from the intrinsic cost of increasing dimensionality. Considering the traditional Marching Cubes, for example, Bhaniramka et al. [21] propose a generalized version for any n but limited to $k = 1$. It is observed that, although it is possible to create a complete lookup table for an arbitrary dimension, this becomes impractical in high dimensions because the total number of entries in the table is 2^{2^n}, so it is understood that generating the entries on demand would be more appropriate. On the other hand, considering a generalized version of the Marching Tetrahedra, the subdivision of the domain into n-dimensional simplices may create a factorial-order number of partitions (see Section 2.1 for details), requiring mechanisms to mitigate this cost.

☆ This article was recommended for publication by F.H. de Figueiredo.

* Corresponding author.
E-mail addresses: lucas.reia@usp.br (L.M. Reia), gameiro@math.rutgers.edu (M. Gameiro), tomasbmribeiro@usp.br (T.B.M. Ribeiro), castelo@icmc.usp.br (A. Castelo).

https://doi.org/10.1016/j.cag.2025.104237
Received 29 October 2024; Received in revised form 24 April 2025; Accepted 25 April 2025
Available online 9 May 2025

A different approach would be to employ predictor–corrector methods [26, Section 2.2], which do not subdivide the domain. Instead, these methods progressively create (cell by cell) an unstructured mesh that follows the manifold. The vertices of each cell are iteratively adjusted to be close to the manifold under a threshold of tolerance. An example is Brodzik's algorithm [27], which is valid for any $k \geq 1$ and $n \geq k+2$. These methods are often sensitive to the step size and other tolerance parameters used. Furthermore, for closed manifolds these methods require additional computations to correctly handle regions where two portions of the approximation meet. For this reason, we focus on piecewise-linear methods that operate on partitioned domains.

The most efficient algorithms for high dimensions are continuation methods [1,26], which process only the partition cells of the domain that intersect the manifold. Starting from an initial partition cell, these methods traverse to neighboring cells along the manifold in an advancing front fashion. In this sense, state-of-the-art methods for general $n > k \geq 1$ include the tracing algorithm by Boissonnat et al. [28], which leverages the so-called *permutahedral representation* to facilitate the traversal to neighboring partition cells of the domain, and the *Generalized Combinatorial Continuation Hypercubes* by Castelo et al. [29], which uses combinatorial techniques with binary labels to assist in the decomposition of the domain into n-dimensional hypercubes. Both methods are built upon the concepts of a group of triangulations usually attributed to Coxeter [30], Freudenthal [31], and Kuhn [32], which we refer to as *CFK triangulation*. It is a highly regular simplicial decomposition of the domain with a convenient set of properties.

In this paper, we introduce a new algorithm that combines the binary labeling technique from Castelo et al. [29,33] with the general approach from Boissonnat et al. [28]. This results in a continuation method with reduced computational cost, that is both faster and more memory efficient. The method generates an abstract cell complex representation of the manifold (see Section 2.4) and hence geometrical realizations of the cells are not produced. As a result, the constructed cells are not guaranteed to be homeomorphic to closed balls and their triangulations may contain self-intersections.

The rest of the paper is organized as follows: Section 2 provides a review of the necessary background and literature, detailing the concepts of implicitly defined manifolds and relevant triangulation methods. Section 3 introduces the *Fast Continuation Hypercubes* (FCH) algorithm, including implementation details. Section 4 analyzes the computational complexity of the FCH algorithm compared to existing algorithms. Section 5 presents experimental results, highlighting the performance improvements achieved by the FCH algorithm. Finally, Section 6 concludes the paper with a summary of our findings and possible directions for future research.

2. Background and literature review

In this section we present some definitions and background material, including the definitions of an *implicitly defined manifold* and of the types of cells used in this paper. For more details on these definitions and related concepts, see [26,34,35].

Definition 1. Consider a C^1 function $F : \mathbb{R}^n \to \mathbb{R}^k$ with $n \geq k$. A point $x \in \mathbb{R}^n$ is called a *regular point* of F if $\operatorname{rank}(DF(x)) = k$. If x is not a regular point of F, it is called a *critical point* of F. A point $c \in \mathbb{R}^k$ is a *regular value* of F if x is a regular point for all $x \in F^{-1}(c)$. If c is not a regular value, it is called a *singular value* of F.

Definition 2. A set $\mathcal{M} \subset \mathbb{R}^n$ is called an *implicitly defined* $(n-k)$-*dimensional manifold* if there exists a C^1 function $F : \mathbb{R}^n \to \mathbb{R}^k$ and a regular value c of F such that

$$\mathcal{M} = F^{-1}(c) = \{x \in \mathbb{R}^n \mid F(x) = c\}.$$

Without any loss of generality we can assume that

$$\mathcal{M} = F^{-1}(0) = \{x \in \mathbb{R}^n \mid F(x) = 0\}.$$

Definition 3. The points $v_0, \ldots, v_k \in \mathbb{R}^n$ are said to be *affinely independent* if the vectors $v_1 - v_0, \ldots, v_k - v_0$ are linearly independent.

Definition 4. A *simplex of dimension* k, or simply a k-*simplex*, generated by the affinely independent points

$$v_0, \ldots, v_k \in \mathbb{R}^n$$

is the set of points

$$\sigma = \left\{ v \in \mathbb{R}^n \;\middle|\; v = \sum_{i=0}^{k} \lambda_i v_i, \text{ with } \lambda_i \geq 0 \text{ and } \sum_{i=0}^{k} \lambda_i = 1 \right\},$$

and is denoted by $\sigma = [v_0, \ldots, v_k]$. A 0-simplex is also referred to as a *vertex* and a 1-simplex as an *edge*.

A k-simplex $\sigma = [v_0, \ldots, v_k]$ is the convex hull of the vertices $v_0, \ldots, v_k$ in $\mathbb{R}^n$.

Definition 5. Let $\sigma = [v_0, \ldots, v_k]$ and $\tau = [u_0, \ldots, u_m]$ be simplices of dimensions k and m, respectively. The simplex τ is a *face* of σ if $m \leq k$ and $\{u_0, \ldots, u_m\} \subseteq \{v_0, \ldots, v_k\}$. If τ is a face of σ we say that σ is a *coface* of τ.

Definition 6. A *hypercube* in $\mathbb{R}^n$ is a set of the form

$$I = \prod_{i=1}^{n} I_i \subset \mathbb{R}^n,$$

where $I_i = [a_i, b_i]$ with $a_i \leq b_i$. When $a_i = b_i$ we denote $I_i = \{a_i\}$. The *dimension* of I is $\dim(I) = \#\{i \mid I_i = [a_i, b_i] \text{ with } a_i < b_i\}$, that is, the dimension of I is the number non-trivial intervals defining I. A hypercube of dimension k is also called a k-*dimensional hypercube* or simply a k-*hypercube*. A 0-hypercube is also referred to as a *vertex* and a 1-hypercube as an *edge*.

Note that a hypercube is aligned with the axes of its domain.

Definition 7. Let $I = \prod_{i=1}^{n} I_i$ and $J = \prod_{i=1}^{n} J_i$ be hypercubes in $\mathbb{R}^n$. We say that J is a *face* of I if $\dim(J) \leq \dim(I)$ and for $i = 1, \ldots, n$ either $J_i = I_i$ or $I_i = [a_i, b_i]$ and $J_i \in \{\{a_i\}, \{b_i\}\}$. A face of dimension k is also referred to as a k-*face* of the hypercube I. If J is a face of I we say that I is a *coface* of J.

2.1. The CFK triangulation

The CFK triangulation [30–32] is used to decompose the unit hypercube into a highly regular simplicial complex and is defined as follows. Consider the canonical basis of $\mathbb{R}^n$ given by

$$e_1 = (1, 0, \ldots, 0),\ e_2 = (0, 1, 0, \ldots, 0), \ldots,\ e_n = (0, 0, \ldots, 0, 1),$$

that is, the ith entry of e_i is 1 and all the other entries are 0. Given a bijection (permutation) $\alpha : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ define

$$v_0(\alpha) := \mathbf{0}, \tag{1}$$

$$v_i(\alpha) := v_0(\alpha) + \sum_{j=1}^{i} e_{\alpha(j)}, \quad \text{for } i = 1, \ldots, n.$$

The points $v_0(\alpha), \ldots, v_n(\alpha) \in \mathbb{R}^n$ are vertices of the unit n-hypercube $I = \prod_{i=1}^{n} [0, 1]$ and all the vertices of I can be obtained this way for some choice of permutation α. These points define an n-simplex

$$\sigma(\alpha) := [v_0(\alpha), v_1(\alpha), \ldots, v_n(\alpha)] \tag{2}$$

inscribed in the hypercube I. Let Λ be the set of all permutations

$$\alpha : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}.$$

The CFK triangulation of I consists of all n-simplices generated by all permutations in Λ, that is, the CFK triangulation of I is given by the set of simplices

$$\mathcal{T} := \{\sigma(\alpha) \mid \alpha \in \Lambda\}.$$

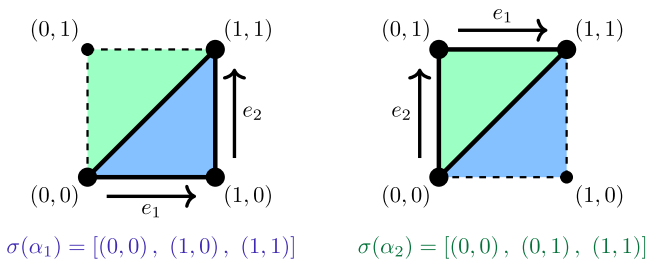

Fig. 1. CFK triangulation of the unit square decomposing it into 2 triangles corresponding to the two permutations α_1 and α_2 of $\{1,2\}$.

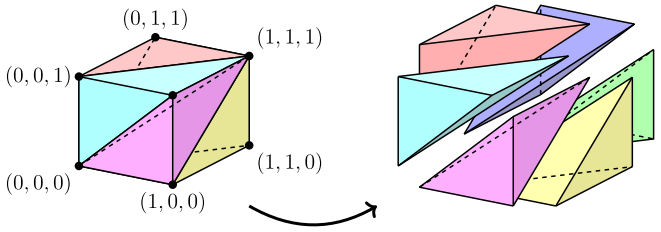

Fig. 2. CFK triangulation of the unit cube decomposing it into 6 tetrahedra.

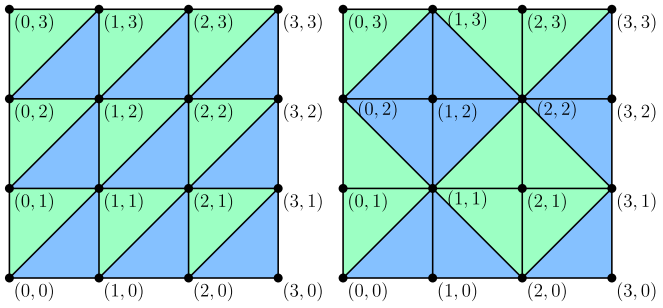

Fig. 3. CFK triangulations of a 2D domain. On the left: the K_1 triangulation. On the right: the J_1 triangulation.

The CFK triangulation $\mathcal{T}$ decomposes the unit hypercube I into $n!$ distinct simplices of dimension n. Figs. 1 and 2 illustrate this decomposition in dimensions 2 and 3, respectively.

The unit n-hypercube can be mapped to a general n-hypercube I_G by scaling and translation. Hence the CFK triangulation of the unit n-hypercube can be used to define the CFK triangulation of I_G via this mapping. Given a rectangular domain $\Omega \subset \mathbb{R}^n$, a triangulation of Ω can be obtained by first partitioning Ω into a grid of n-hypercubes, and then constructing a CFK triangulation of each of these partitions individually. To ensure consistency in the decomposition of shared faces between adjacent n-hypercubes, two main types of triangulations are considered, following the same nomenclature as Todd [36]:

- K_1 **triangulation:** constructed using translations of the unit n-hypercube;
- J_1 **triangulation:** constructed using reflections of the unit n-hypercube.

These triangulations are illustrated in Fig. 3.

When working with CFK triangulations, it is not necessary to explicitly construct each individual n-simplex that subdivides the domain. By assuming the domain is implicitly partitioned by a CFK triangulation, the specific n-hypercube and n-simplex which contain a given point can be identified via arithmetic expressions. This allows for the construction of the simplices only when they are needed.

2.2. Vertices of the approximation

To create an approximation of the manifold $\mathcal{M}$ implicitly defined by $F(\mathbf{x}) = \mathbf{0}$, where $F : \mathbb{R}^n \to \mathbb{R}^k$, the first step is to create the vertices of the approximation, and then connect these vertices to form the edges, faces, and higher-dimensional cells. In the case of piecewise linear approximations employing a triangulation, the vertices are created by detecting the intersections of $\mathcal{M}$ with the k-simplices generated by the triangulation. Thus, the main role of the triangulation is to decompose the domain into these k-simplices, enabling the detection of intersections (for further details, see [26,28]).

One of the simplest methods is to use a triangulation (such as a K_1 triangulation) to decompose the domain into n-simplices and then for each n-simplex check all of its k-faces for intersections with $\mathcal{M}$. Let τ be a k-simplex $\tau = [u_0, u_1, \dots, u_k]$. The intersection of τ with $\mathcal{M}$ can be approximated by solving the following linear system for the barycentric coordinates λ of the approximated intersection point v_{mf}:

$$\begin{bmatrix} 1 & 1 & \dots & 1 \\ F(u_0) & F(u_1) & \dots & F(u_k) \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_k \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{3}$$

The approximated intersection point v_{mf} is then given by

$$v_{mf} = \sum_{i=0}^{k} \lambda_i u_i. \tag{4}$$

If $\lambda_i > 0$ for $i = 0, \dots, k$, then v_{mf} is in the interior of τ. To ensure that the coefficient matrix in (3) is non-singular, a small random perturbation is typically applied to the coordinates of the vertices of τ, similar to the approach used in [29,33].

Although this algorithm is straightforward, it is computationally inefficient because it evaluates every k-face of every n-simplex, and hence it usually evaluates many k-faces that do not intersect $\mathcal{M}$, leading to unnecessary computations. The subsequent sections present more advanced algorithms designed to address this inefficiency and reduce the overall computational cost.

2.3. Permutahedron-based tracing algorithm

An efficient manifold tracing algorithm based on a K_1 triangulation of the domain is presented by Boissonnat et al. [28]. We refer to this algorithm as the Permutahedron-based Tracing Algorithm (PTA). Assuming that the domain is implicitly triangulated by a K_1 triangulation and that a seed k-simplex (a face) that intersects the manifold is provided, the key idea of the algorithm is to traverse the domain through only the $(k+1)$-simplices (the cofaces) that intersect the manifold and are connected to the starting k-simplex. A major component of the algorithm is the *permutahedral representation* of a simplex, a proposed data structure that enables efficient generation of all faces and cofaces of a simplex of any dimension in the triangulation of the domain. See Appendix A.1 for a step-by-step example of the PTA algorithm.

Introducing the vector $e_{n+1} = -\sum_{i=1}^{n} e_i$, from (1) and (2) we obtain

$$e_{n+1} = v_0(\alpha) - v_n(\alpha).$$

Consequently, the sequence of vectors

$$(e_{\alpha(1)}, e_{\alpha(2)}, \dots, e_{\alpha(n)}, e_{n+1})$$

provides a cyclic representation of the n-simplex $\sigma(\alpha)$ [37]:

$$v_i(\alpha) = v_{i-1}(\alpha) + e_{\alpha(i)}, \quad \text{for } i = 1, \dots, n,$$

and

$$v_n(\alpha) + e_{n+1} = v_0(\alpha).$$

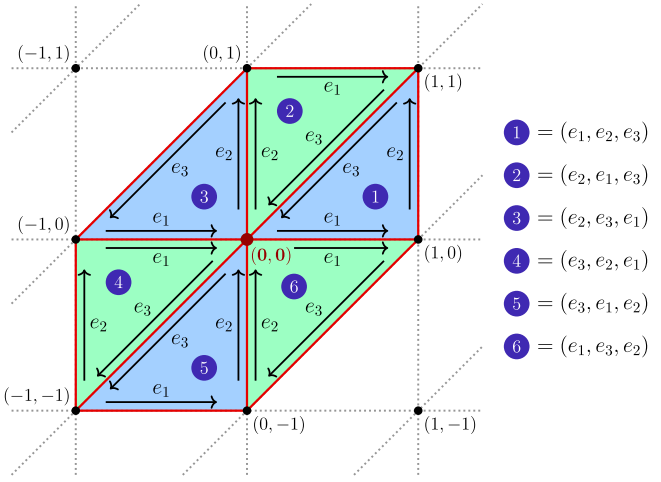

Fig. 4. The permutations α^* of $\{1,2,3\}$ in a 2D domain generate all 2-simplices containing the vertex $\mathbf{0}$.

Exploiting this cyclical nature, any n-simplex in the K_1 triangulation containing the vertex $\mathbf{0}$ can be generated using a permutation

$$\alpha^* : \{1, 2, \dots, n, n+1\} \to \{1, 2, \dots, n, n+1\}$$

as follows. Let

$$v_0(\alpha^*) := \mathbf{0},$$
$$v_i(\alpha^*) := v_{i-1}(\alpha^*) + e_{\alpha^*(i)}, \quad \text{for } i = 1, \dots, n,$$

and define

$$\sigma(\alpha^*) := [v_0(\alpha^*), v_1(\alpha^*), \dots, v_n(\alpha^*)].$$

Notice that $v_n(\alpha^*) + e_{\alpha^*(n+1)} = v_0(\alpha^*)$ and hence $e_{\alpha^*(n+1)}$ is not needed to represent the simplex $\sigma(\alpha^*)$. However $e_{\alpha^*(n+1)}$ is useful to efficiently generate the faces and cofaces of a simplex and so it is included in the representation of $\sigma(\alpha^*)$. We represent the simplex $\sigma(\alpha^*)$ by the sequence of vectors $(e_{\alpha^*(1)}, e_{\alpha^*(2)}, \dots, e_{\alpha^*(n+1)})$ used to generate it. Considering all permutations of $\{1, 2, \dots, n, n+1\}$, there are $(n+1)!$ simplices containing $\mathbf{0}$. Fig. 4 illustrates this scenario for a 2D domain. Although the vector sequence above is based on the unit vectors of the canonical basis, it can be mapped to any n-simplex in a K_1 triangulation through translation and scaling. Furthermore, by selecting an adequate vector sequence, the reference vertex $\mathbf{0}$ can be mapped to any of the vertices of any of these n-simplices.

An $(n-1)$-face τ of the n-simplex $\sigma(\alpha^*)$ can be obtained by removing a vertex $v_j(\alpha^*)$:

$$\tau = [v_0(\alpha^*), \dots, v_{j-1}(\alpha^*), v_{j+1}(\alpha^*), \dots, v_n(\alpha^*)].$$

Given that $v_{j+1}(\alpha^*) - v_{j-1}(\alpha^*) = e_{\alpha^*(j)} + e_{\alpha^*(j+1)}$, the corresponding cyclic vector sequence for τ becomes

$$\left(e_{\alpha^*(1)}, \dots, e_{\alpha^*(j-1)}, e_{\alpha^*(j)} + e_{\alpha^*(j+1)}, e_{\alpha^*(j+2)}, \dots, e_{\alpha^*(n+1)}\right).$$

Thus, the faces of σ can be derived by manipulating the vector sequence rather than the vertices: combining two consecutive vectors $e_{\alpha^*(j)}$ and $e_{\alpha^*(j+1)}$ into a single sum in the sequence is equivalent to removing the vertex $v_j(\alpha^*)$ from σ. Conversely, the n-dimensional cofaces of τ can be obtained by the inverse operation — splitting the combined vector sum into two consecutive vectors.

Using these ideas, Boissonnat et al. [28] demonstrate that all faces and cofaces of a simplex can be generated by manipulating the vector sequence defining the simplex (see [28] for the details on these operations). They show that an m-simplex has $m+1$ faces of dimension $m-1$, and at most $2^{n-m+1}-2$ cofaces of dimension $m+1$. To enable a global representation of a simplex within the grid, the following definition is introduced.

Definition 8. Let $\tau = [u_0, \dots, u_m]$ be an m-face of σ where the relative order of the vertices from σ is maintained, and let $p \in \mathbb{Z}^n$ be the vertex in the grid that u_0 is mapped to. The *permutahedral representation* of τ is given by

$$\tau = (p, w),$$

where $w = (w_0, w_1, \dots, w_m)$ is the cyclic sequence of vectors

$$w_i := u_{i+1} - u_i, \quad \text{for } i = 0, \dots, m-1;$$
$$w_m := u_0 - u_m = -\sum_{i=0}^{m-1} w_i.$$

If p corresponds to the minimal point of τ in lexicographical order, this notation referred to as the *canonical permutahedral representation*. In this case the vertices $u_0, \dots, u_m$ are monotonically increasing with respect to the lexicographical order.

The permutahedral representation defined above implies that

$$w_i := \left\{\text{sum}\left(E_i\right) \mid E_i \subset \left\{e_1, e_2, \dots, e_n, e_{n+1}\right\}\right\}, \text{ for } i = 0, \dots, m;$$

$$E_i \cap E_j = \emptyset, \quad \text{for } i \neq j;$$

$$\bigcup_{i=0}^{m} E_i = \left\{e_1, e_2, \dots, e_n, e_{n+1}\right\}.$$

In the case of the canonical permutahedral representation we have that $e_{n+1} \in E_m$.

The canonical permutahedral representation of any simplex generated by the K_1 triangulation can be obtained by first sorting its vertices in ascending lexicographical order, and then calculating the corresponding values of p and w. This representation uniquely identifies the simplex, making it suitable for utilization in sets of unique elements (such as hash tables), while also allowing for the operations to create its faces and cofaces.

The pseudocode of the PTA algorithm is presented in Algorithm 1. In this tracing algorithm, starting from a seed k-simplex τ_{seed} that intersects the manifold, the permutahedral representation of τ_{seed} is manipulated to generate all its $(k+1)$-dimensional cofaces σ. For each σ, all k-dimensional faces τ' are subsequently generated and checked for intersection with the manifold. To avoid redundant processing of the same k-simplex, a record of all previously processed τ is maintained. The algorithm begins by initializing the empty set of k-simplices that have already been processed (Line 1) and the queue of k-simplices that are yet to be processed (Line 2). At each iteration, a k-simplex τ is dequeued from Q for processing (Line 4). The algorithm then iterates through all k-simplices adjacent to τ in the K_1 triangulation (Lines 5–6). For each adjacent k-simplex τ', it checks whether a vertex of the approximation is present in τ' (Line 8). If so, the coordinates of this approximation vertex are determined (Line 10), and τ' is enqueued to be processed in the next iterations (Line 13).

2.4. Generalized combinatorial continuation hypercubes

The Combinatorial Marching Hypercubes [33] is designed for approximating manifolds defined by functions of the form $F : \mathbb{R}^n \to \mathbb{R}$, employing combinatorial techniques to improve memory efficiency. This approach was generalized to handle functions of the form $F : \mathbb{R}^n \to \mathbb{R}^k$ with $n > k$, leading to the development of the Generalized Combinatorial Marching Hypercubes (GCMH) and its continuation variant, the Generalized Combinatorial Continuation Hypercubes (GCCH) [29].

These algorithms utilize a simplicial decomposition of the domain via a K_1 triangulation, where each n-hypercube is processed independently. For each n-hypercube, the method avoids processing all simplices in the n-hypercube by analyzing only those contained in its k-faces. See Appendix A.2 for a step-by-step example of the GCCH algorithm.

Algorithm 1: Permutahedron-based Tracing Algorithm (PTA).

Input : A function $F : \mathbb{R}^n \to \mathbb{R}^k$ defining the manifold $\mathcal{M} = F^{-1}(0)$. A seed k-simplex τ_{seed} for the continuation algorithm.
Output: A list of vertices of the approximation and the respective k-simplices.

```
S ← { };                                   // initialize set
Q ← [τ_seed];                              // initialize queue
while Q is not empty do
    τ ← remove(Q);
    foreach (k+1)-coface σ of τ do
        foreach k-face τ′ of σ do
            if τ′ ∉ S then
                λ ← solve system for τ′;                    // Eq. (3)
                if min(λ) > 0 then
                    v_mf ← τ′ · λ;                          // Eq. (4)
                    Save {τ′, v_mf};
                    Insert τ′ in S;
                    Insert τ′ in Q;
```

As mentioned before, an important aspect of these algorithms is that the hypercube and the simplicial decomposition of the domain are combinatorial in nature and do not need to be explicitly constructed. Each face of the hypercube and its simplicial decomposition can be created independently and as needed. A binary labeling system is employed in order to efficiently obtain the adjacency and membership relationships among the faces of the n-hypercube.

Definition 9. Let $I = \prod_{i=1}^{n} I_i$ be an m-face of the unit n-hypercube in $\mathbb{R}^n$. This implies that either $I_i = [0,1]$ or $I_i \in \{\{0\},\{1\}\}$. The *label* of the face I is the integer ℓ with $2n$ binary digits defined as

$$\ell := \sum_{j=0}^{2n-1} d_j 2^j = \left[d_{2n-1}\ d_{2n-2}\ \dots\ d_0\right]_2,$$

where for $i = 1, \dots, n$ the digit pairs $(d_{i-1},\ d_{i-1+n})$ are given by

$$(d_{i-1},\ d_{i-1+n}) := \begin{cases} (0,\ 0), & \text{if } I_i = [0,1] \\ (1,\ 0), & \text{if } I_i = \{0\} \\ (0,\ 1), & \text{if } I_i = \{1\} \end{cases}$$

Definition 10. Let $I = \prod_{i=1}^{n} I_i$ be a m-face of the unit n-hypercube in $\mathbb{R}^n$. If $I_i \in \{\{0\},\{1\}\}$, we say that the coordinate i of the face is *fixed*. If $I_i = [0,1]$, we say that the coordinate i is *free*.

Fig. 5 illustrates the labeling of all 2-faces of the unit cube in 3D. In this binary labeling system, each set (nonzero) bit of ℓ corresponds to a coordinate of the face that is fixed at either 0 or 1. Consequently, all possible k-faces of the unit n-hypercube can be generated by manipulating ℓ using combinatorial techniques. Since a coordinate cannot be fixed at both 0 and 1 simultaneously, there will be exactly $(n-m)$ set bits in an m-face. Additionally, because a general n-hypercube is combinatorially equivalent to the unit n-hypercube, this labeling system can be used to manipulate the faces of a general n-hypercube.

Consider a K_1 triangulation of the domain and let τ be a k-simplex in a k-face of an n-hypercube decomposing the domain. The coordinates of the vertices of τ corresponding to the fixed coordinates of the k-face are constant. Hence only the vectors of the canonical basis corresponding to the free coordinates of the k-face are needed to generate all the vertices of τ. Furthermore, the initial vertex of all k-simplices in this k-face is determined by the n most significant bits of the label ℓ of the k-face. More precisely, let I be a k-face of the unit n-hypercube and let ℓ be the label of this k-face I. The k-simplices τ of I are generated as follows. Let $\left[d_{2n-1}\ d_{2n-2}\ \dots\ d_0\right]_2$ be the binary representation of ℓ and notice that the set of free coordinates of I is given by

$$\mathcal{I}_\ell = \left\{i \mid d_{i-1} + d_{i-1+n} = 0\right\}. \tag{5}$$

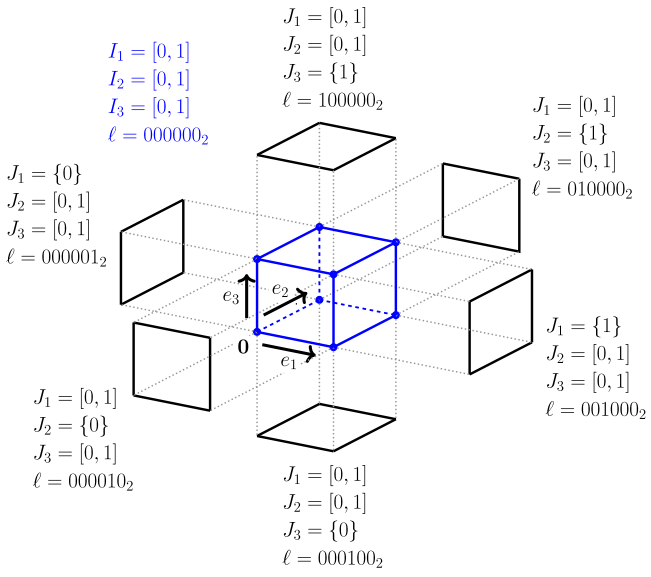

Fig. 5. Decomposition of the unit 3-hypercube into all of its 2-faces and their corresponding labels.

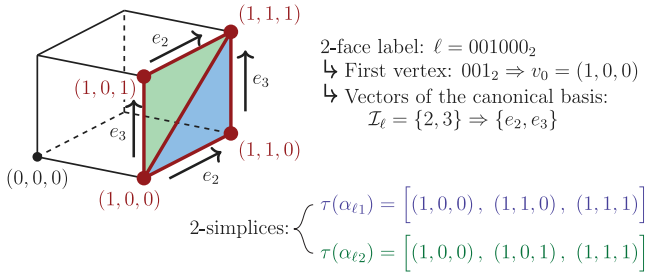

Fig. 6. Triangulating a 2-face of the unit 3-hypercube based on its label.

Let

$$\alpha_\ell : \{1, 2, \dots, k\} \to \mathcal{I}_\ell$$

be a bijection and define

$$v_0(\alpha_\ell) = (d_n,\ d_{n+1},\ \dots\ ,\ d_{2n-1})$$

and

$$v_i(\alpha_\ell) = v_0(\alpha_\ell) + \sum_{j=1}^{i} e_{\alpha_\ell(i)}, \quad \text{for } i = 1, \dots, k.$$

The points $v_0(\alpha_\ell), \dots, v_k(\alpha_\ell)$ are vertices of I and define the k-simplex

$$\tau(\alpha_\ell) = [v_0(\alpha_\ell), v_1(\alpha_\ell), \dots, v_k(\alpha_\ell)]$$

in the face I. The set of all k-simplices in the face I is obtained by considering all bijections $\alpha_\ell : \{1, 2, \dots, k\} \to \mathcal{I}_\ell$. The k-simplices in a k-face of a general n-hypercube can be obtained from the k-simplices of the unit n-hypercube by scaling and translation.

Fig. 6 provides an example of applying the CFK triangulation to a 2-face of a hypercube, which is analogous to the decomposition in Fig. 1. The GCMH algorithm approximates the manifold by processing all k-simplices τ of all k-faces of each n-hypercube and creating vertices of the approximation within those τ which intersect the manifold. Once they are created, the vertices are connected based on adjacency rules, forming the edges of the approximation (for more details, see [29,33]). In the continuation variant GCCH, the algorithm initiates with a seed n-hypercube and enqueues, for subsequent processing, the neighboring n-hypercubes that share faces intersecting the manifold.

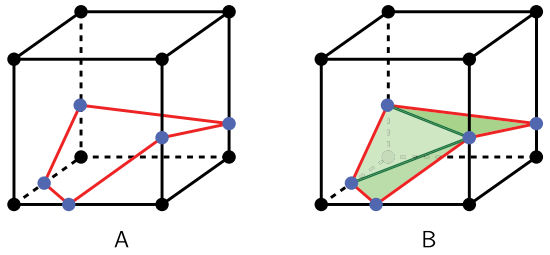

Fig. 7. (A) Combinatorial cell represented by a list of vertices (blue) and edges (red). (B) The combinatorial cell in (A) is decomposed into convex affine cells (green triangles).

The pseudocode for the GCCH algorithm is presented in Algorithm 2. To avoid redundancy, a record of the indices of the n-hypercubes that have been processed is maintained. The algorithm begins by initializing the empty set of n-hypercubes that have already been processed (Line 1) and the queue of n-hypercubes to be processed (Line 2). At each iteration, an n-hypercube h is dequeued from $\mathcal{Q}$ for processing (Line 4). The algorithm then initializes a list that will store the approximation vertices found in h (Line 7), the corresponding list of k-face labels for each vertex (Line 8), and a binary label marking all neighbors of h that share approximation vertices (Line 9). It proceeds by iterating through all k-simplices on the k-faces of h in the K_1 triangulation (Lines 10–11). For each k-simplex τ, the algorithm checks for the presence of an approximation vertex (Line 12) and, if found, determines its coordinates (Line 14) and sets the neighboring n-hypercubes that share face ℓ (Line 17). Finally, the algorithm inserts these neighbors into $\mathcal{Q}$ for subsequent processing (Line 18).

The same adjacency rules used within an n-hypercube to create the edges of the approximation are also applied to connect edges to form faces, connect faces to form volumes, and to construct higher-dimensional cells up to dimension $n-k$. This set of rules is referred to as the Combinatorial Skeleton [29,33], which can be constructed in a subsequent step, after the vertices and edges of the approximation have been generated.

While the cells generated by the Combinatorial Skeleton do not form a cellular complex in the geometric sense (they represent distorted polytopes — see Fig. 7), they do constitute an abstract cell complex (a cellular complex in the topological sense). A 2D cell has a boundary composed of 1D cells (edges); a 3D cell has a boundary composed of 2D cells; and this pattern continues up to the final polytope, an $(n-k)$-dimensional cell with a boundary composed of $(n-k-1)$-dimensional cells. As the dimension increases, the structure gets more complex and there is no guarantee that each m-dimensional cell is topologically equivalent (homeomorphic) to an m-dimensional closed ball. These distorted polytopes do not intersect each other, as each polytope is contained within its own n-hypercube, and the shared faces between adjacent polytopes are coincident (identical, without gaps). However, a simplicial decomposition of a polytope may contain self-intersections. Consequently, a simplicial decomposition of the generated approximation will not necessarily constitute a manifold. For visualization purposes within a rendering pipeline, it is sufficient to decompose only the 2D cells into triangles using a triangle fan or similar approach, but this does not guarantee the absence of intersections between triangles.

3. Fast continuation hypercubes algorithm

In our proposed algorithm, called Fast Continuation Hypercubes (FCH), the goal is to mix the binary labeling of the GCCH with the concepts of the PTA algorithm. Starting from an initial k-face from an n-hypercube, the traversal proceeds through the grid of the domain via the $(k+1)$-dimensional cofaces, while adhering to the manifold. The cofaces are decomposed into $(k+1)$-simplices using the K_1 triangulation,

Algorithm 2: Generalized Combinatorial Continuation Hypercubes (GCCH).

Input : A function $F:\mathbb{R}^n \to \mathbb{R}^k$ defining the manifold $\mathcal{M} = F^{-1}(0)$.
A seed n-hypercube h_{seed} for the continuation algorithm.
Output: A list of vertices of the approximation and the respective face labels for each n-hypercube.

```
1  H ← { };                          // initialize set
2  Q ← [h_seed];                     // initialize queue
3  while Q is not empty do
4      h ← remove(Q);
5      if h ∉ H then
6          Insert h in H;
7          V ← [ ];                  // initialize list
8          L ← [ ];                  // initialize list
9          ℓ_n ← 0;                  // initialize binary label
10         foreach label ℓ of a k-face of h do
11             foreach k-simplex τ of ℓ do
12                 λ ← solve system for τ;                    // Eq. (3)
13                 if min(λ) > 0 then
14                     v_mf ← τ · λ;                         // Eq. (4)
15                     Insert v_mf in V;
16                     Insert ℓ in L;
17                     ℓ_n ← ℓ_n OR ℓ;                       // set neighbors
18         Insert in Q the neighbors indicated by ℓ_n;
19         Save {h, L, V};
```

Line 17: OR is the bitwise OR operator.

and only those simplices that intersect the manifold are evaluated. To facilitate this process, a global notation for the binary labels of the faces is established, uniquely identifying each face in the domain, upon which the operations for generating cofaces are conducted. See Appendix A.3 for a step-by-step example of the FCH algorithm.

3.1. Canonical face notation

The face label from Definition 9 provides a local representation of a face, depending on the specific n-hypercube under consideration. To globally identify a face, we introduce the following definition.

Definition 11. Let I be an n-hypercube on a grid in $\mathbb{R}^n$ with vertices on the integer lattice $\mathbb{Z}^n$. Let $p \in \mathbb{Z}^n$ be the vertex on the grid that corresponds to the minimal vertex of I in lexicographical order. Let ℓ be the label of an m-face J of I. The *global face representation* of the face J is defined as

$$J := (p, \ell).$$

If p is a point contained in J, this representation is the *canonical face representation*.

The global face representation is the hypercube equivalent of the permutahedral representation. Note that, in the canonical face representation, the point p is also the minimal point of J in lexicographical order, and the n most significant bits of ℓ are unset. Since a face is shared among neighboring n-hypercubes, it can be expressed from the perspective of each n-hypercube that contains it. In the example of Fig. 8, we have:

$$\underbrace{\underbrace{(5,1)}_{\substack{\text{2-hypercube}\\ \text{position}}}, \underbrace{0011_2}_{\substack{\text{face}\\ \text{label}}}}_{\substack{\text{canonical face}\\ \text{representation}}} = (4,1), 0110_2 = (5,0), 1001_2 = (4,0), 1100_2.$$

The fact that the n most significant bits of the label are always 0 in the canonical face representation simplifies the generation of cofaces and the calculation of relationships between faces and cofaces, as

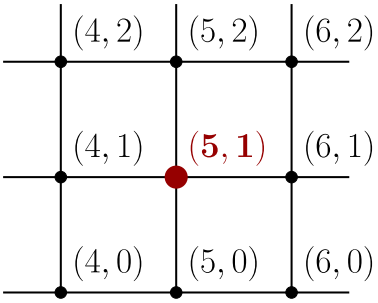

Fig. 8. Example of a 0-face (the vertex $(5,1)$) shared among four 2-hypercubes in the 2D domain.

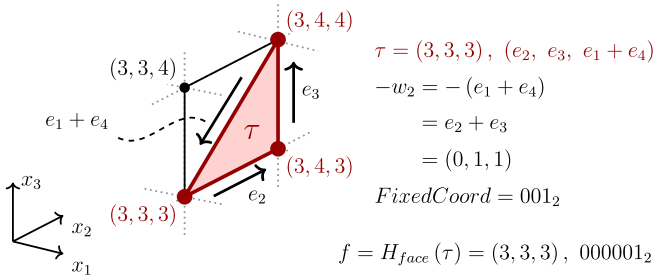

Fig. 9. Determining the hypercube 2-face that contains a 2-simplex in the 3D domain.

discussed in the following sections. For this reason, the canonical face representation is always used to represent hypercubes in our algorithm.

3.2. Simplex face

In a way akin to the PTA algorithm, which traverses the $(k+1)$-simplices of the domain, our algorithm operates by traversing the $(k+1)$-hypercubes of the domain. Initially, a seed k-simplex τ that intersects the manifold is provided. The algorithm then determines the hypercube k-face f containing τ. Starting from f, the method iteratively moves to all $(k+1)$-cofaces of f, systematically tracing the manifold in the process. Therefore, the first step is to determine f from τ.

As in the PTA algorithm, an initial seed k-simplex τ must be provided. However the FCH algorithm has the additional requirement that the τ must be contained in a hypercube k-face. One way to determine τ is to run the GCMH algorithm until the first intersecting k-simplex is encountered. This process can be guided, for example, by prioritizing k-simplices whose vertices are closer to $F(\mathbf{x}) = \mathbf{0}$. If the manifold contains multiple disconnected components, it is necessary to find one seed simplex in each component, and execute the algorithm for each component individually. In the canonical permutahedral representation, τ is represented by:

$$\tau = \underbrace{\underbrace{(p_\tau)}_{\substack{\text{reference}\\ \text{vertex}}}\ ,\ \underbrace{(w_0,\ w_1,\ \ldots\ ,\ w_k)}_{\substack{\text{vector}\\ \text{sequence}}}}_{\text{canonical permutahedral representation}}\ .$$

The fact that $-w_k$ is the vector that translates the minimal point of τ (which is p_τ) into the maximal point of τ in lexicographical order can be utilized to identify the coordinates of f that are fixed. The fixed coordinates are represented by the bits of an n-digit bit mask $FixedCoord$, defined as follows:

$$FixedCoord \quad := \quad \sum_{j=0}^{n-1} \left(1 + w_k \cdot e_{j+1}\right) 2^j \quad \Rightarrow$$

$$\Rightarrow \quad FixedCoord_j \quad = \quad \begin{cases} 0, & \text{if } w_k \cdot e_{j+1} \neq 0 \\ 1, & \text{otherwise.} \end{cases}$$

Since the vertices of τ are monotonically increasing from p_τ, making p_τ the minimal point of both τ and f, it follows that p_τ is also the reference point for the n-hypercube containing f, and each fixed coordinate i of f takes the value $\{p_{\tau i}\}$. Consequently, f is given in terms of its canonical face representation H_{face} by:

$$f \ = \ H_{face}(\tau) \quad := \underbrace{\underbrace{(p_\tau)}_{\substack{\text{reference}\\ n\text{-hypercube}}}\ ,\ \underbrace{\underbrace{00\ldots0}_{\substack{n \text{ most}\\ \text{significant}\\ \text{bits}}}\ \underbrace{FixedCoord}_{\substack{n \text{ least}\\ \text{significant}\\ \text{bits}}}}_{2n \text{ bits}}}\ .$$

Because f represents a k-face, $FixedCoord$ must contain exactly $(n-k)$ set bits. Fig. 9 provides an example for the case when $n = 3$ and $k = 2$, showing how f is obtained from τ. Having determined the canonical face representation of the hypercube k-face f that contains the seed k-simplex τ, the next step is to generate the $(k+1)$-cofaces of f.

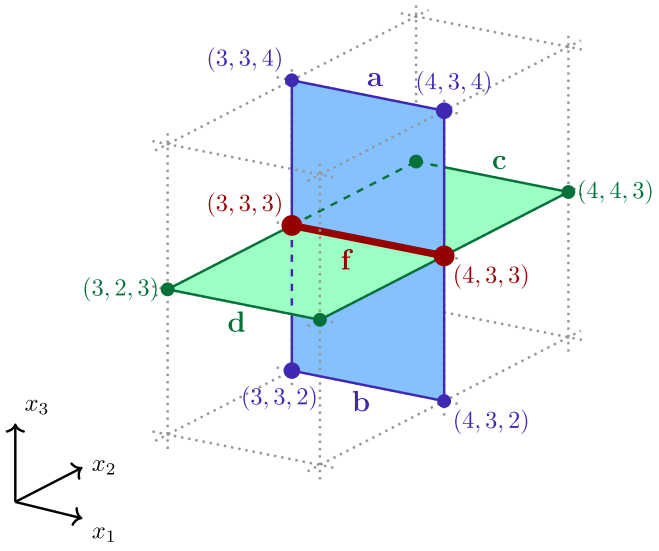

Fig. 10. Example of an 1-face (f) in the 3D domain, and all of its cofaces in the Cartesian grid.

3.3. Generating the cofaces from a k-face

Let J be an m-hypercube where the coordinate i is fixed. Then $J_i = \{a_i\}$, with $a_i \in \mathbb{R}$. Let I be a $(m+1)$-coface of J where the coordinate i is free. Then there are only two possibilities for I: one where $I_i = [b_i, a_i]$, $a_i > b_i$, and another where $I_i = [a_i, b_i]$, $b_i > a_i$. This implies that if the ith coordinate of a k-face f is fixed, then f represents the intersection of two distinct $(k+1)$-cofaces along the x_i axis. In Fig. 10, the third coordinate of the face f is fixed, which means that f is the intersection of the cofaces a and b. Similarly, since the second coordinate of f is also fixed, f is the intersection of the cofaces c and d. These intersections can be expressed as Cartesian products as

$$a = [3,4] \times \{3\} \times [3,4]\,, \qquad b = [3,4] \times \{3\} \times [2,3]\,,$$

$$c = [3,4] \times [3,4] \times \{3\}\,, \qquad d = [3,4] \times [2,3] \times \{3\}\,,$$

$$a \cap b = c \cap d = [3,4] \times \{3\} \times \{3\} = f\,.$$

Since each set bit in the label of f corresponds to a fixed coordinate, we can generate all the cofaces of f by unsetting each bit individually. For each unset bit j (where $0 \leq j < n$) there will be two cofaces: one in the same n-hypercube as f, and another in the previous n-hypercube

along the direction of the x_{j+1} axis. These cofaces are denoted by $c_{f_{x^+_{j+1}}}$ and $c_{f_{x^-_{j+1}}}$, respectively:

$$c_{f_{x^+_{j+1}}} = \underbrace{(p_\tau)}_{\text{reference } n\text{-hypercube}}, \underbrace{00\ldots0}_{n \text{ most significant bits}} \underbrace{(FixedCoord \ \ \mathsf{XOR} \ \ 2^j)}_{n \text{ least significant bits}},$$

$$c_{f_{x^-_{j+1}}} = \underbrace{(p_\tau - e_{j+1})}_{\text{reference } n\text{-hypercube}}, \underbrace{00\ldots0}_{n \text{ most significant bits}} \underbrace{(FixedCoord \ \ \mathsf{XOR} \ \ 2^j)}_{n \text{ least significant bits}},$$

where XOR is the bitwise XOR operator.

As a result, the total number of $(k+1)$-cofaces generated from any given k-face is $2(n-k)$. The cofaces of f in Fig. 10 are

$$\begin{aligned} & f && = && (3,3,3), 000110_2, \\ a \ = \ & c_{f_{x^+_3}} && = && (3,3,3), 000010_2, \\ b \ = \ & c_{f_{x^-_3}} && = && (3,3,2), 000010_2, \\ c \ = \ & c_{f_{x^+_2}} && = && (3,3,3), 000100_2, \\ d \ = \ & c_{f_{x^-_2}} && = && (3,2,3), 000100_2. \end{aligned}$$

After identifying each $(k+1)$-coface c_f of the k-face f, we can apply a manifold tracing algorithm inside each of these $(k+1)$-hypercubes, as described in the next section.

3.4. Tracing the manifold approximation edge in a coface

Tracing the manifold inside a single hypercube of dimension $k+1$ means that we are essentially tracing the isocurve $\mathcal{M}^*$ of an implicit function

$$F^* : \mathbb{R}^{k+1} \to \mathbb{R}^k,$$

where $\mathcal{M}^*$ is the intersection of the original manifold $\mathcal{M}$ and this $(k+1)$-hypercube. We can exploit this fact by using a tracing algorithm specific to curves embedded in high dimensions. More specifically, our implementation is based on the algorithm described by Allgower and Georg [26].

This algorithm works by traversing the $(k+1)$-simplices inside the $(k+1)$-hypercube while following $\mathcal{M}^*$. It starts at the k-simplex τ and continues until $\mathcal{M}^*$ leaves the $(k+1)$-hypercube through another k-simplex. We must first identify the starting $(k+1)$-simplex σ, which is the $(k+1)$-simplex containing τ inside the $(k+1)$-hypercube. This means that σ contains all the vertices of τ plus one additional vertex v, which we have to determine.

Let f be a hypercube k-face and c_f a $(k+1)$-coface of f. Let ℓ_f and ℓ_{c_f} be the labels of f and c_f, respectively. With respect to Eq. (5), if c_f was created by unfixing the coordinate i of f, then i is a value in $\mathcal{I}_{\ell_{c_f}}$ that is not in $\mathcal{I}_{\ell_f}$. Since σ must keep the same vertices of τ and follow the rules of the CFK triangulation, the only possible values for the additional vertex v that needs to be determined are either $\tau_0 - e_i$ or $\tau_k + e_i$ (which also reflects the fact that there are two cofaces for each fixed coordinate). In the first case, v becomes the minimal point of σ, which is emphasized by a change of the reference n-hypercube. In the canonical permutahedral representation, if p_{c_f} is the reference n-hypercube of c_f, then σ is given by

$$\sigma_{c_f} = \begin{cases} \left(p_{c_f}\right), (w_0, w_1, \ldots, w_{(k-1)}, e_i, (w_k - e_i)), \text{if } p_{c_f} = p_\tau; \\ \left(p_{c_f}\right), (e_i, w_0, w_1, \ldots, w_{(k-1)}, (w_k - e_i)), \text{if } p_{c_f} = p_\tau - e_i. \end{cases} \tag{6}$$

Fig. 11 illustrates the σ corresponding to each c_f in Fig. 10:

$$\tau = (3,3,3), \left(e_1,\ e_2 + e_3 + e_4\right);$$

$$\sigma_{c_{f_{x^+_3}}} = (3,3,3), \left(e_1,\ e_3,\ e_2 + e_4\right);$$

$$\sigma_{c_{f_{x^-_3}}} = (3,3,2), \left(e_3,\ e_1,\ e_2 + e_4\right);$$

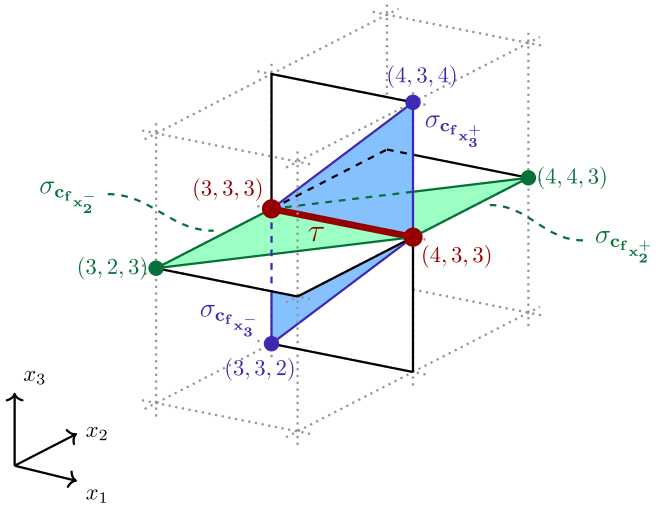

Fig. 11. Highlight of the $(k+1)$-simplices σ calculated for each coface in Fig. 10.

$$\sigma_{c_{f_{x^+_2}}} = (3,3,3), \left(e_1,\ e_2,\ e_3 + e_4\right);$$

$$\sigma_{c_{f_{x^-_2}}} = (3,2,3), \left(e_2,\ e_1,\ e_3 + e_4\right).$$

To perform the traversal inside the hypercube we use the following result, known as the Door-in/Door-out principle.

Proposition 1. *A $(k+1)$-simplex has exactly 0 or 2 k-faces that intersect $\mathcal{M}$.*

The proof is provided by Allgower and Georg [26]. If a $(k+1)$-simplex σ has two k-faces τ and τ' intersecting the manifold, we say that the manifold enters σ through τ and leaves it through τ'. In our case, the input k-simplex τ is known, and Eq. (6) gives σ, so now we want to compute the output k-simplex τ'. To do this, we use a technique equivalent to one iteration of Bittner's Generalized Regula Falsi algorithm [38], which was used in a similar way in [20].

Assuming that $[u_0,\ u_1,\ \ldots\ ,\ u_k]$ are the vertices of τ, we first compute λ using Eq. (3). Note that $\lambda_i > 0$, for $i = 0, \ldots, k$, because τ is a k-simplex that intersects $\mathcal{M}$. The vertex v, which is the vertex in σ that is not in τ, is used in the following system to obtain μ:

$$\begin{bmatrix} 1 & 1 & \ldots & 1 \\ F(u_0) & F(u_1) & \ldots & F(u_k) \end{bmatrix} \begin{bmatrix} \mu_0 \\ \mu_1 \\ \vdots \\ \mu_k \end{bmatrix} = \begin{bmatrix} 1 \\ F(v) \end{bmatrix}. \tag{7}$$

Knowing λ and μ, we find the index i where:

$$\frac{\lambda_i}{\mu_i} = \min\left\{ \frac{\lambda_l}{\mu_l}, \quad \mu_l > 0 \right\}. \tag{8}$$

The output k-simplex τ' is finally obtained by replacing the vertex u_i with v in τ. In other words,

$$\tau' = [u_0,\ \ldots\ ,\ u_{i-1},\ v,\ u_{i+1},\ \ldots\ ,\ u_{k-1},\ u_k].$$

The value λ' is computed by applying Eq. (3) to τ', and the vertices of the approximation are created by applying Eq. (4) to the pairs (τ, λ) and (τ', λ'). Connecting these vertices gives us the edge of the approximation that is inside σ, thus tracing a manifold edge inside a single $(k+1)$-simplex.

When the manifold leaves σ through τ', it enters another $(k+1)$-simplex σ' through τ'. To find σ', we need to determine the additional vertex v' needed to generate σ' from τ', that is, σ' is generated by

$$\{u_0,\ \ldots\ ,\ u_{i-1},\ v,\ u_{i+1},\ \ldots\ ,\ u_{k-1},\ u_k\} \cup \{v'\}.$$

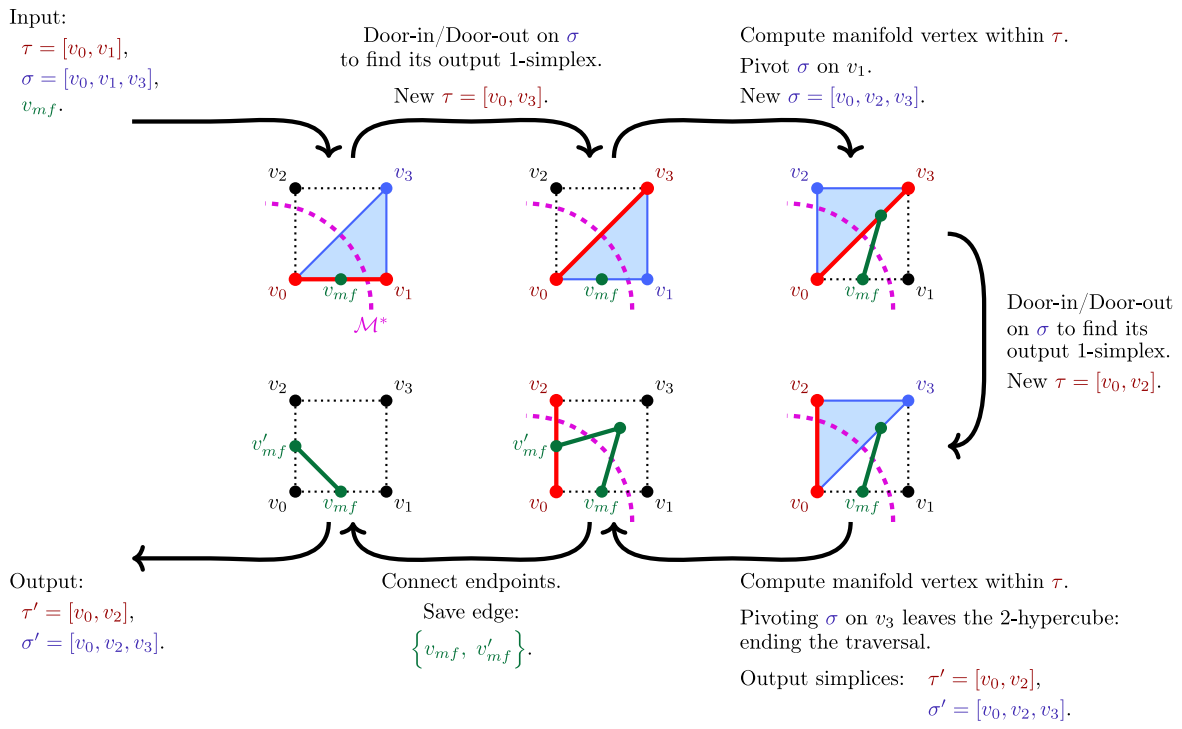

Fig. 12. Steps to trace an edge $\{v_{mf},\ v'_{mf}\}$ in a 2-dimensional hypercube.

Using the CFK triangulation, v' can be easily obtained by pivoting σ on u_i. Let $[\tilde{u}_0, \ldots, \tilde{u}_{k+1}]$ be the vertices of σ sorted in ascending lexicographical order, and let $(w_0, \ldots, w_k)$ be the vector sequence given by Definition 8, which generates σ starting from $\tilde{u}_0$. Pivoting σ on vertex $\tilde{u}_j$, with $1 \leq j \leq k$, means swapping the positions of vectors w_{j-1} and w_j in the vector sequence, effectively replacing $\tilde{u}_j$ while keeping all the other vertices.

By repeating this combination of the Door-in/Door-out principle plus pivoting on the subsequent $(k+1)$-simplices, we can traverse the entire $(k+1)$-hypercube, crossing only the k-simplices that intersect the manifold, until the manifold exits c_f. A step-by-step example of this tracing algorithm is provided in Fig. 12, showing how an edge of the approximation is traced inside a 2-hypercube.

If the index found by Eq. (8) corresponds to pivoting on the first or the last vertex of the $(k+1)$-hypercube (minimal or maximal points in lexicographical order), it means that the manifold has left the coface and the traversal wants to move to a neighboring $(k+1)$-hypercube, so the traversal ends. Since we are only interested in the vertices inside the k-faces of the n-hypercube, we only store the endpoints of the traversal: if the manifold enters the $(k+1)$-hypercube through $\{\tau, \sigma\}$ and leaves it through $\{\tau', \sigma'\}$, the vertices are created in τ and τ' and connected to form a single edge in the entire $(k+1)$-hypercube, thus creating one edge of the resulting approximation.

Once we have identified the k-simplex τ' from which $\mathcal{M}^*$ leaves the $(n+1)$-hypercube, we can then restart the process on τ', tracing the manifold across all $(k+1)$-hypercubes that contain τ'. By repeating this procedure iteratively, the algorithm generates the approximation of the entire manifold within the domain. The resulting approximation consists of the same edges that are created by the GCCH algorithm. Hence, the Combinatorial Skeleton (see Section 2.4) can also be used to create the higher-dimensional cells of the approximation.

Observe that Eqs. (3) and (7) can be rewritten to reduce the overall cost as follows:

$$\begin{bmatrix} F(u_1) - F(u_0) & \ldots & F(u_k) - F(u_0) \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_k \end{bmatrix} = \begin{bmatrix} -F(u_0) \end{bmatrix} \tag{9}$$

and

$$\begin{bmatrix} F(u_1) - F(u_0) & \ldots & F(u_k) - F(u_0) \end{bmatrix} \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_k \end{bmatrix} = \begin{bmatrix} F(v) - F(u_0) \end{bmatrix}, \tag{10}$$

with $\lambda_0 = 1 - \lambda_1 - \cdots - \lambda_k$ and $\mu_0 = 1 - \mu_1 - \cdots - \mu_k$. Noting that the same coefficient matrix is used in Eqs. (9) and (10), both systems can be solved at the same time. Typically, these systems are solved using LU decomposition with partial pivoting, which presents an asymptotic complexity of $O(k^3)$, though other methods could also be employed.

3.5. Memory storage

The Door-in/Door-out principle ensures that we can avoid creating duplicate edges in the approximation. The reason is that only the two $(k+1)$-simplices σ and σ' (which represent the entry and exit locations of a $(k+1)$-hypercube) need to be added to a set of already traversed $(k+1)$-simplices to effectively prevent the respective $(k+1)$-hypercube from being processed more than once. Each time a new σ is about to be traversed, this set is checked for redundancy.

To optimize memory usage, this approach is equivalent to storing information about which cofaces of τ have yet to be traversed. Instead of storing each of the $2(n-k)$ cofaces of τ separately, we can store a single integer C for each τ. The initial value of C is given by the function C_n:

$$C = C_n(\tau) := (FixedCoord \cdot 2^n) + FixedCoord.$$

Each set bit in C indicates a coface of τ to be traversed. When the algorithm traverses the coface $\sigma_{c_{f_{x_{j+1}^+}}}$ starting from τ, the jth bit of C is unset, indicating that this coface has already been traversed. Similarly, when the coface $\sigma_{c_{f_{x_{j+1}^-}}}$ is traversed, the $(j+n)$th bit of C is unset. Given that $f = (p_f, \ell_f)$ represents the hypercube k-face that contains τ, and $c_f = (p_{c_f}, \ell_{c_f})$ represents its coface that contains σ, we can update C

by using a bit mask U as follows:

$$U\left(f, c_f\right) = \begin{cases} \ell_f \ \text{XOR} \ \ell_{c_f}, & \text{if } p_f = p_{c_f} \\ \left(\ell_f \ \text{XOR} \ \ell_{c_f}\right) \cdot 2^n, & \text{otherwise} \end{cases}$$

$C = C \ \text{XOR} \ U(f, c_f)$.

We use a dictionary that stores key–value pairs, where the keys are the k-simplices τ and the values are the corresponding integers C. In the step where the cofaces of f are being generated, the algorithm verifies the C value of τ and only proceeds to the tracing step if the bit corresponding to the coface has not yet been unset in C. After creating an edge of the approximation, we have f and c_f (which contain τ and σ), and we calculate f' and c'_f (which contain τ' and σ') and update the C integers of both τ and τ' in the dictionary accordingly.

3.6. Boundary tracking

The process of traversing the grid while following the manifold is similar to an advancing front algorithm. Hence it is not necessary to store all cells already traversed within the domain; it is sufficient to store only the boundary of the traversed region.

Based on the storage method described in the previous section, when all cofaces originating from a simplex τ are traversed, τ is not revisited because access to τ is blocked by the neighboring simplices τ' that are added to the dictionary. Therefore, when the integer C associated with a simplex τ reaches zero, τ can be removed from the dictionary, leading to further memory savings.

A double-ended queue is used to perform the traversal in a breadth-first manner: the τ to be processed is removed from the front of the queue, while any τ' resulting from traversing the cofaces are inserted at the back. To ensure that the queue contains only unique elements, the algorithm first checks if τ' is already in the queue and inserts it only if it is not.

For a manifold of dimension $n-k$, the boundary of the explored region has dimension $n-k-1$, so the memory usage is proportional to this reduced dimension.

3.7. Algorithm pseudocode

Algorithm 3 presents the complete pseudocode of the FCH algorithm. The algorithm begins by initializing the queue of k-simplices to be processed (Line 1) and a dictionary that records the boundary of $(k+1)$-simplices to be processed (Line 2). At each iteration, a k-simplex τ is removed from the front of $\mathcal{Q}$ for processing (Line 4). The algorithm identifies the k-hypercube f that contains τ (Line 5) and computes λ for τ (Line 6). It then determines the coordinates of the approximation vertex in τ (Line 7). Next, the algorithm iterates through all cofaces c_f of f in the grid (Line 8) and, in each c_f, identifies the $(k+1)$-simplex σ that contains τ (Line 10). The manifold is traced inside c_f as discussed in Section 3.4 (Line 11). The algorithm then determines the k-hypercube f' that contains the output k-simplex τ' (Line 12) and identifies the $(k+1)$-coface of f' that contains σ' (Line 13). The dictionary is updated accordingly (Lines 14–19) and τ' is enqueued to be processed in the next iterations (Line 20). Finally, the algorithm computes λ' for τ' (Line 21) and determines the coordinates of its approximation vertex (Line 22).

3.8. Implementation and tests

We implemented the three algorithms (PTA, GCCH, and FCH) in the C++ programming language and ran various test cases (see Section 5) to measure the memory usage and execution time of the traversal. The generated cells are progressively saved to disk, with only the necessary data for the traversal of each algorithm being kept in RAM. This allows the generation of bigger approximations that cannot fit entirely in RAM.

Algorithm 3: Fast Continuation Hypercubes (FCH)

Input : A function $F : \mathbb{R}^n \to \mathbb{R}^k$ defining the manifold $\mathcal{M} = F^{-1}(0)$.
A seed k-simplex τ_{seed} for the continuation algorithm.
Output: A list of edges of the approximation with each respective pair of k-simplices.

1 $\mathcal{Q} \leftarrow [\tau_{\text{seed}}]$; // initialize queue
2 $\mathcal{B} \leftarrow \{ [\tau_{\text{seed}}, C_n(\tau_{\text{seed}})] \}$; // initialize dictionary
3 **while** $\mathcal{Q}$ *not empty* **do**
4 $\tau \leftarrow$ popfront$(\mathcal{Q})$;
5 $f \leftarrow H_{face}(\tau)$;
6 $\lambda \leftarrow$ solve system for τ; // Eq. (3)
7 $v_{mf} \leftarrow \tau \cdot \lambda$; // Eq. (4)
8 **foreach** *coface* c_f *of* f **do**
9 **if** $U(f, c_f)$ *AND* $\mathcal{B}(\tau)$ **then**
10 $\sigma \leftarrow \tau \cup \{v\}$; // $(k+1)$-simplex
11 $\tau', \sigma' \leftarrow$ Trace(τ, σ); // trace edge
12 $f' \leftarrow H_{face}(\tau')$;
13 $c'_f \leftarrow H_{face}(\sigma')$;
14 **if** $\mathcal{B}$ *contains* τ' **then**
15 $\mathcal{B}(\tau') \leftarrow \mathcal{B}(\tau') \ \text{XOR} \ U\left(f', c'_f\right)$;
16 **else**
17 $key \leftarrow \tau'$;
18 $value \leftarrow C_n(\tau') \ \text{XOR} \ U\left(f', c'_f\right)$;
19 Insert $[key, value]$ pair in $\mathcal{B}$;
20 Insert τ' at the back of $\mathcal{Q}$;
21 $\lambda' \leftarrow$ solve system for τ'; // Eq. (3)
22 $v'_{mf} \leftarrow \tau' \cdot \lambda'$; // Eq. (4)
23 Save edge $\left\{\tau, v_{mf}, \tau', v'_{mf}\right\}$;
24 Remove τ from $\mathcal{B}$;

Line 9: AND is the bitwise AND operator.
Lines 15, 18: XOR is the bitwise XOR operator.

All methods were implemented by the same programmer and tested on the same machine, ensuring consistency in the evaluation. The specifications of the test machine are as follows:

- Processor: AMD Ryzen 9 3900X 12-Core;
- Memory: 4× 16 GB DIMM DDR4 3200 MHz;
- Storage: 223.57 GB SSD.

The runtime of each algorithm was measured internally, using the high-resolution clock from the C++ standard library. Memory usage was measured externally, using a utility tool from the operating system. Because the operating system always allocates a minimum amount of memory for any given program, the memory usage presents a very noticeable lower bound.

Our C++ implementation of all three algorithms has been made available in an online repository, along with a tool to visualize the generated approximations.[1]

4. Computational complexity per iteration

Since the total runtime of an algorithm depends on the manifold being approximated, here we present the time complexity equations of the PTA, GCCH, and FCH algorithms for the traversal of the domain as a function of the number of structures traversed (the number of iterations executed by each algorithm). For a more direct comparison, in Appendix B we offer a discussion on the upper bound on computational cost for each algorithm, and a worst-case comparison.

[1] https://github.com/lucasmreia/fch.

4.1. PTA algorithm

The computational complexity of the PTA algorithm, as calculated by Boissonnat et al. [28], is given by:

$$S_k \cdot \underbrace{\left(2^{(n-k+1)}-2\right)\cdot(k+1)}_{O\left(k2^{(n-k)}\right)} \cdot \mathcal{I}, \tag{11}$$

where:

- S_k is the total number of k-simplices processed (number of iterations);
- $\left(2^{(n-k+1)}-2\right)$ is the maximum number of $(k+1)$-dimensional cofaces a k-simplex can have;
- $(k+1)$ is the number of k-faces (k-simplices) in each coface;
- $\mathcal{I}$ is the cost of solving the $k \times k$ system of Eq. (9) to obtain the λ for each k-simplex.

Note that, since the PTA algorithm processes every k-face of each coface, there will be many processed k-simplices which do not intersect the manifold. However, the Door-in/Door-out principle can be applied to directly find the output k-simplex in each coface, which reduces the complexity of the middle term to $O\left(2^{(n-k)}\right)$.

4.1.1. GCCH algorithm

The computational complexity of the GCCH algorithm, as calculated by Castelo et al. [29], is given by:

$$\mathcal{H}_n \cdot \underbrace{\binom{n}{n-k}\cdot 2^{(n-k)}\cdot k!}_{O\left(\frac{n!}{(n-k)!}2^{(n-k)}\right)} \cdot \mathcal{I}, \tag{12}$$

where:

- $\mathcal{H}_n$ is the total number of n-hypercubes processed (number of iterations);
- $\binom{n}{n-k}\cdot 2^{(n-k)}$ is the number of k-faces of each n-hypercube, where $\binom{n}{n-k}$ represents the number of possible combinations of coordinates that can be fixed, and $2^{(n-k)}$ represents the number of possible values for each set of fixed coordinates (each coordinate can be fixed at either 0 or 1);
- $k!$ is the number of k-simplices in each k-face;
- $\mathcal{I}$ is the cost of solving the $k \times k$ system of Eq. (9) to obtain the λ for each k-simplex.

As the GCCH algorithm processes only the k-simplices on the k-faces of the n-hypercube, and not those in the interior of the n-hypercube, the resulting number of vertices and edges of the approximation is significantly smaller than that of the PTA algorithm. The memory usage is also lower for the GCCH algorithm because it records only the position of the traversed n-hypercubes. On the other hand, the GCCH algorithm always processes all k-simplices of all k-faces, which leads to unnecessary computations.

4.2. FCH algorithm

The computational complexity of the FCH algorithm is given by:

$$\mathcal{H}_k \cdot \underbrace{2(n-k)\cdot\left(S_{k+1}+1\right)}_{O\left((n-k)(k+1)!\right)} \cdot \mathcal{I}, \tag{13}$$

where:

- $\mathcal{H}_k$ is the total number of k-hypercubes processed (number of iterations);
- $2(n-k)$ is the number of $(k+1)$-cofaces for one k-hypercube, as discussed in Section 3.3;
- S_{k+1} is the number of $(k+1)$-simplices processed when tracing the manifold inside a coface, as explained in Section 3.4. In the best case we have $S_{k+1} = 1$, and in the worst case we have $S_{k+1} = (k+1)!$;
- $\mathcal{I}$ is the cost of solving the two $k \times k$ systems of Eqs. (9) and (10), used to determine λ and μ for each time the Door-in/Door-out principle is applied. An additional $\mathcal{I}$ is added at the end to calculate λ' for the output τ'.

Although Eq. (13) contains factorial terms, the total cost of the FCH algorithm is greatly reduced by restricting the evaluation to the same k-simplices as the GCCH, but only to those that do intersect the manifold thanks to the Door-in/Door-out principle.

5. Results

The performance results obtained from the implementation and evaluation of the three methods – PTA, GCCH, and FCH – are presented in Figs. 13–18. Figs. 13–15 show the performance as a function of the resolution of the grid that subdivides the domain (the number of subdivisions or partitions per dimension) for the following test cases:

- Hypersphere S^4 generated by:

$$F:\mathbb{R}^5 \to \mathbb{R}, \qquad F(\mathbf{x}) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 1.$$

- Klein Bottle in $\mathbb{R}^5$ generated by:

$$F:\mathbb{R}^5 \to \mathbb{R}^3,$$
$$F(\mathbf{x}) = \begin{bmatrix} x_1 - \left(3 + cos\left(\frac{x_4}{2}\right) sin\left(x_5\right) - sin\left(\frac{x_4}{2}\right) sin\left(2x_5\right)\right) cos\left(x_4\right) \\ x_2 - \left(3 + cos\left(\frac{x_4}{2}\right) sin\left(x_5\right) - sin\left(\frac{x_4}{2}\right) sin\left(2x_5\right)\right) sin\left(x_4\right) \\ x_3 - sin\left(\frac{x_4}{2}\right) sin\left(x_5\right) + cos\left(\frac{x_4}{2}\right) sin\left(2x_5\right) \end{bmatrix}$$

- Circle S^1 embedded in $\mathbb{R}^9$ generated by:

$$F:\mathbb{R}^9 \to \mathbb{R}^8, \qquad F(\mathbf{x}) = \begin{bmatrix} x_1^2 + x_9^2 - 1.0001 \\ x_2^2 - 1.0002 \\ x_3^2 - 1.0003 \\ x_4^2 - 1.0004 \\ x_5^2 - 1.0005 \\ x_6^2 - 1.0006 \\ x_7^2 - 1.0007 \\ x_8^2 - 1.0008 \end{bmatrix}.$$

  In this case, the level set corresponds to several disconnected curves, but only the central circumference was traced due to the chosen starting point.

Figs. 16–18 show the performance as a function of the dimension of the manifold for the following test cases:

- Hypersphere $S^{(7-k)}$ embedded in $\mathbb{R}^7$, for varying k, generated by:

$$F:\mathbb{R}^7 \to \mathbb{R}^k, \qquad F(\mathbf{x}) = \begin{bmatrix} 1 - \sum_{i=1}^{7-k+1} {x_i}^2 \\ x_{7-k+2} \\ x_{7-k+3} \\ \vdots \\ x_7 \end{bmatrix}.$$

- Hypersphere $S^{(15-k)}$ embedded in $\mathbb{R}^{15}$, for varying k, generated by:

$$F:\mathbb{R}^{15} \to \mathbb{R}^k, \qquad F(\mathbf{x}) = \begin{bmatrix} 1 - \sum_{i=1}^{15-k+1} {x_i}^2 \\ x_{15-k+2} \\ x_{15-k+3} \\ \vdots \\ x_{15} \end{bmatrix}.$$

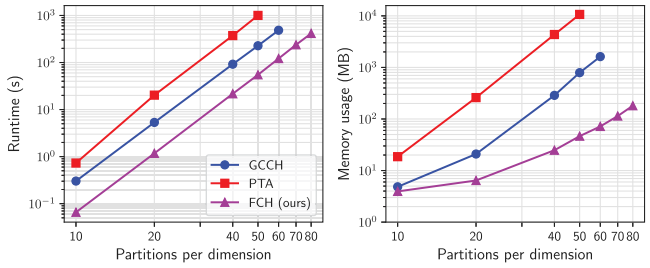

Fig. 13. Hypersphere S^4 ($F : \mathbb{R}^5 \to \mathbb{R}$).

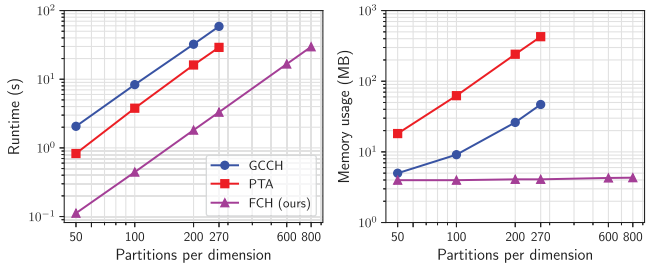

Fig. 14. Klein Bottle in $\mathbb{R}^5$ ($F : \mathbb{R}^5 \to \mathbb{R}^3$).

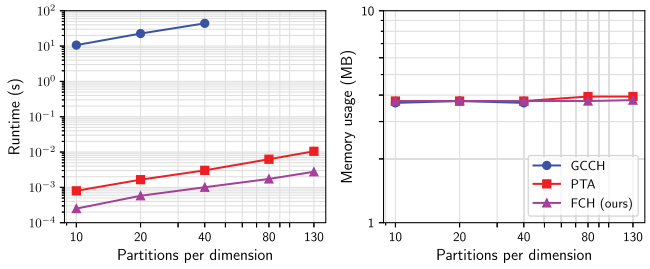

Fig. 15. Circle S^1 embedded in $\mathbb{R}^9$ ($F : \mathbb{R}^9 \to \mathbb{R}^8$).

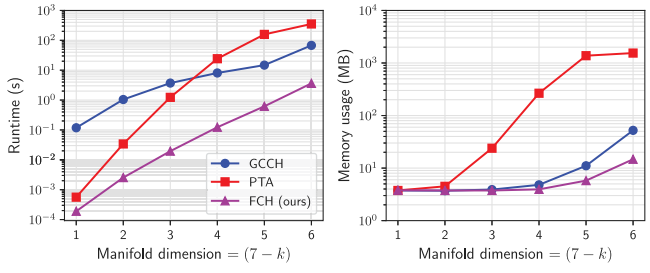

Fig. 16. Hypersphere $S^{(7-k)}$ embedded in $\mathbb{R}^7$ ($F : \mathbb{R}^7 \to \mathbb{R}^k$, varying k), with 10 partitions per dimension.

- Cartesian product of S^1 k times, $\left(S^1\right)^k$, for varying k, generated by:

$$F : \mathbb{R}^{2k} \to \mathbb{R}^k, \qquad F(\mathbf{x}) = \begin{bmatrix} x_1^2 + x_2^2 - 1 \\ x_3^2 + x_4^2 - 1 \\ \vdots \\ x_{2k-1}^2 + x_{2k}^2 - 1 \end{bmatrix}.$$

In all cases, the GCCH algorithm consumed less memory than the PTA algorithm, which was expected since the GCCH algorithm stores only the indices of the traversed n-hypercubes. The PTA algorithm presented the highest memory usage, as it stores all the evaluated k-simplices. In contrast, the FCH algorithm presented the lowest memory

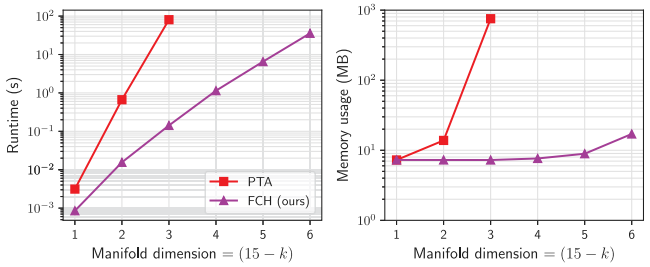

Fig. 17. Hypersphere $S^{(15-k)}$ embedded in $\mathbb{R}^{15}$ ($F : \mathbb{R}^{15} \to \mathbb{R}^k$, varying k), with 10 partitions per dimension.

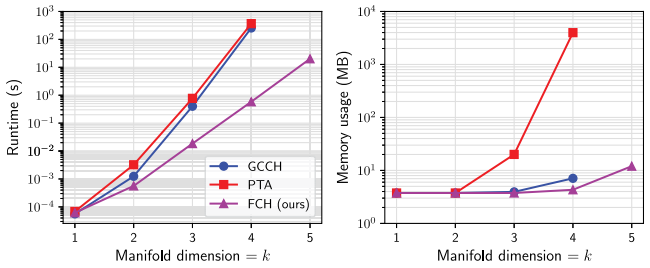

Fig. 18. Cartesian product of S^1 taken k times, $\left(S^1\right)^k$ ($F : \mathbb{R}^{2k} \to \mathbb{R}^k$, varying k), with 5 partitions per dimension.

usage, as it stores only the boundary of the $(k+1)$-simplices that still need to be traversed. In many cases, the memory usage of an algorithm was lower than the minimum allocated by the operating system, which is particularly evident in Figs. 14 and 15, making the memory usage appear constant.

In terms of runtime, Figs. 14–16 show that the PTA algorithm is faster than the GCCH algorithm when the value of k is close to n. When k and n are significantly different, the cost to traverse all $(k+1)$-simplices that intersect the manifold resulted in longer execution times for the PTA algorithm. Because of this, the GCCH algorithm seems better suited than the PTA algorithm for situations where k is low.

Fig. 18 shows that the PTA and GCCH algorithms present relatively similar growth in runtime when both k and $n-k$ are increased by 1. One explanation for this behavior is that, while greater k means that the GCCH will process more k-simplices, greater $n-k$ means that the PTA will process more $(k+1)$-simplices, and these terms have similar influence in the total number of evaluated simplices.

It is important to note that in all three methods, the data structures used to track the traversed regions are implemented as hash tables using the C++ standard library. The higher the memory usage of a method, the more prone it is to hash collisions, with the PTA algorithm being the most affected by this issue. In addition, since the cells generated by the algorithms are stored progressively on disk, methods that generate more vertices and edges are expected to be slowed down by disk write operations, which also penalizes the PTA algorithm the most.

Table 1 presents the performance metrics related to the usage of the hash table for each algorithm. To minimize memory consumption in general scenarios, the structure was designed to expand dynamically and perform rehash operations as needed (on demand). Collisions are tracked during both element insertion and rehashing: a collision is recorded whenever a new element is inserted into a bucket that already contains one or more elements. The FCH algorithm resulted in the lowest number of collisions and rehashes, although close to the GCCH algorithm.

Finally, the FCH algorithm produced an output identical to that of the GCCH algorithm, which consists of edges of affine cells, but this

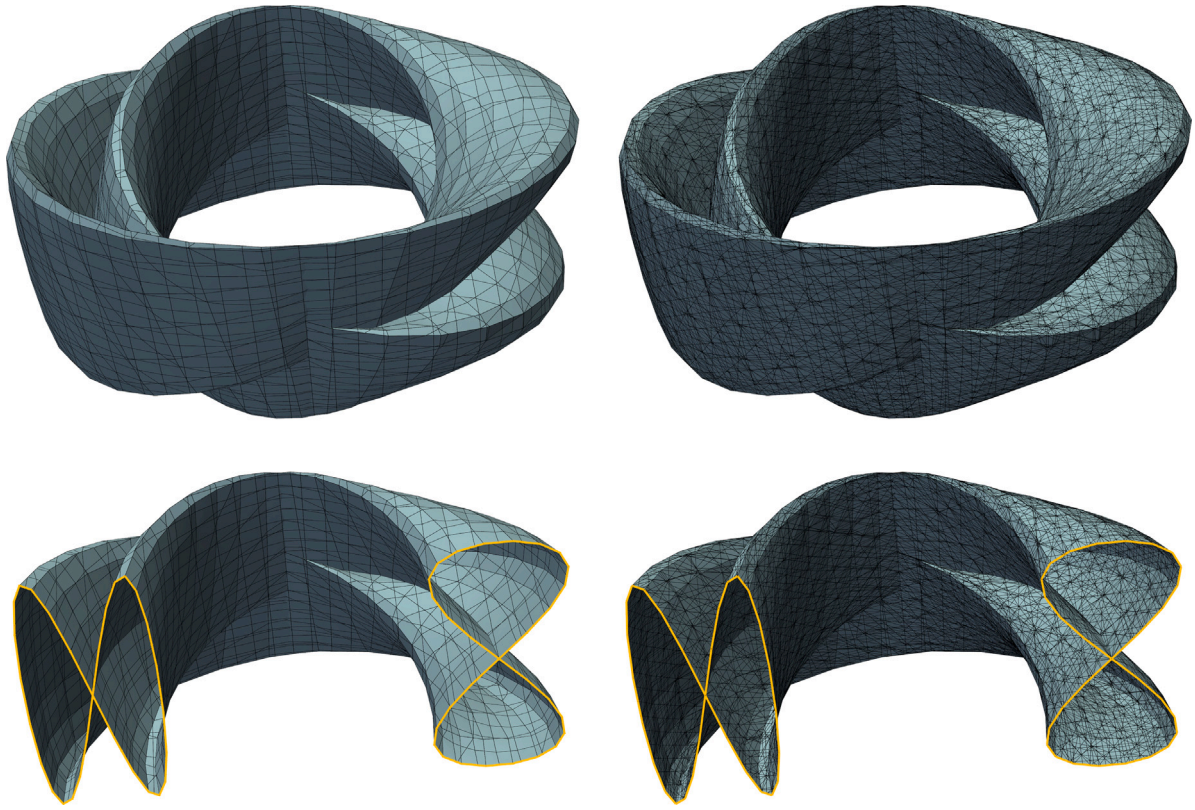

Fig. 19. 3D projections of the approximations generated for the Klein Bottle in $\mathbb{R}^5$ with 30 partitions per dimension. Left column: approximation created by GCCH and FCH. Right column: approximation created by PTA. Top row: entire approximation. Bottom row: cross-sectional cut to reveal the interior of the 3D projection and its auto-intersection, which forms a figure-eight shape. Notice that in $\mathbb{R}^5$ the shape is a manifold, but its projection in $\mathbb{R}^3$ is not (the projection contains self-intersections).

Table 1
Performance metrics of the hash table in different settings.

		PTA	GCCH	FCH
Hypersphere S^4	Rehashes	14	11	9
(10 parts/dim)	Collisions	132.015	14.062	9.417
Klein bottle in $\mathbb{R}^5$	Rehashes	14	10	6
(50 parts/dim)	Collisions	140.879	12.613	12.381
Circle S^1 in $\mathbb{R}^9$	Rehashes	1	1	1
(40 parts/dim)	Collisions	98	11	10

was achieved with significantly less execution time and memory usage. Across all scenarios tested, the FCH algorithm was the most efficient, with the lowest execution time and memory usage. Fig. 19 provides an example of the generated approximations, showing that the approximations generated by the GCCH and FCH algorithms contain significantly fewer cells compared to those generated by the PTA algorithm.

High-dimensional modeling, combined with an appropriate projection to lower dimensions, enables the generation of more complex geometric shapes compared to traditional low-dimensional modeling. In the original domain of F, the level set $F(\mathbf{x}) = \mathbf{0}$ is expected to satisfy the properties of a manifold. However, when this manifold is projected to lower dimensions, no constraints are imposed, allowing the creation of shapes with self-intersections, as illustrated in Fig. 19. Additionally, this approach supports the modeling of shapes of different dimensions – such as curves, surfaces, and volumes – demonstrated in Fig. 20. The implicit equations used to generate the shapes in Fig. 20 are:

- **A:**

$$F(x,y,z) = \begin{bmatrix} x - r_t\ \cos(\omega\ z) \\ y - r_t\ \sin(\omega\ z) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix};$$

- **B,C:**

$$F(x,y,z) = \sqrt{\left(x - r_t\ \cos(\omega\ z)\right)^2 + \left(y - r_t\ \sin(\omega\ z)\right)^2} - r_\gamma = 0;$$

- **D,E:**

$$F(x,y,z,\gamma) = \sqrt{\left(x - r_t\ \cos(\omega\ z)\right)^2 + \left(y - r_t\ \sin(\omega\ z)\right)^2} - \gamma\ r_\gamma = 0.$$

where ω, r_t and r_γ are constants, and $\gamma \in [0,1]$.

6. Conclusion

In this work, we introduce the Fast Continuation Hypercubes (FCH), an algorithm that combines the strengths of the binary labeling technique used in the Generalized Combinatorial Continuation Hypercubes (GCCH) [29] with the general tracing approach of the Permutahedron-based Tracing Algorithm (PTA) [28]. Through a series of experiments, we demonstrate that the FCH algorithm provides a more efficient method to generate piecewise-linear approximations of implicitly defined manifolds compared to both the PTA and GCCH algorithms.

Our results show that the FCH algorithm not only reduces memory usage by limiting memory storage to only the boundary of the region being traversed, but also significantly decreases runtime by limiting the evaluation to only the k-simplices contained in the k-faces of the n-hypercubes that subdivide the domain. The application of the Door-in/Door-out principle effectively avoids processing simplices that do not intersect the manifold, which makes the FCH algorithm particularly suitable for applications where both memory efficiency and computational speed are desired, and underscores its potential as a method for high-dimensional manifold approximation.

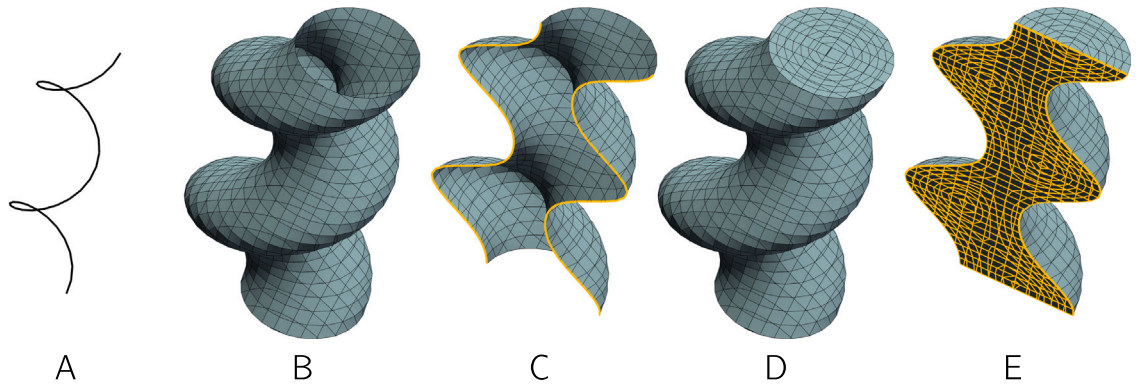

Fig. 20. 3D projections of approximations generated for helical shapes with varying dimensions. A: 1D helix. B,C: 2D tube-shaped helix. D,E: 3D solid helix. While A, B and D display the full approximations, C and E include a cross-sectional cut to reveal the interior structure. Note that, topologically, A is composed of 1D edges; B,C are composed of 2D polygons; and D,E are composed of 3D polyhedra.

For future research, an interesting direction would be to develop adaptive and parallel methods based on the proposed approach. In a general sense, adaptive methods could dynamically allocate computational resources based on the complexity of different regions, improving efficiency. Additionally, parallelization across multiple processors could significantly decrease computation time, especially for high-dimensional problems. These advancements could improve the scalability of the algorithm, making it more suitable for large-scale applications.

CRediT authorship contribution statement

Lucas Martinelli Reia: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Marcio Gameiro:** Writing – review & editing, Supervision, Methodology, Investigation, Formal analysis. **Tomás Bueno Moraes Ribeiro:** Visualization. **Antonio Castelo:** Writing – review & editing, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

L.M.R., M.G. and A.C. were partially supported by São Paulo Research Foundation (FAPESP), Brazil, grants #2013/07375-0 and #2019/07316-0. L.M.R. was also partially supported by São Paulo Research Foundation (FAPESP), Brazil, grant #2023/12195-2. M.G. was also partially supported by DARPA, USA contract HR0011-16-2-0033, National Institutes of Health, USA award R01 GM126555, and Air Force Office of Scientific Research, USA under award number FA9550-23-1-0011 and FA9550-23-1-0400. M.G. and A.C. were also partially supported by São Paulo Research Foundation (FAPESP), Brazil, grant #2024/06833-9. T.B.M.R. was partially supported by the National Council for Scientific and Technological Development (CNPq), Brazil, grant #2023-1713.

Appendix A. Step-by-step example

In order to make the difference between the three algorithms more evident, in this section we present a short step-by-step example of isosurfacing using each algorithm. In this case, we have a 2D manifold embedded in the 3D domain, which is generated by an implicit function $F : \mathbb{R}^3 \to \mathbb{R}$.

A.1. PTA

Fig. A.21 illustrates the steps of the PTA algorithm. Each step is described below.

- **A:** Isosurface (in orange) and the 1D edge of the K_1 triangulation (in dark red) that will serve as the seed τ for the continuation method.
- **B:** Generation of all 2D triangular faces σ of the K_1 triangulation that contain τ.
- **C:** The intersection of the isosurface with a triangular face forms a curve (in magenta). These curves will be approximated by line segments.
- **D:** Generation of the edges of the approximation (in bright red) that lie on each σ. The edges of the approximation are formed by one input vertex (that lie on τ) and one output vertex (that lie on a different edge τ' of σ).
- **E:** The process is restarted from step A at the τ' of each σ (in dark red). This is repeated until the entire isosurface is covered.
- **F:** Final surface mesh generated.

Note that the generated approximation in panel F is not a triangulation, as it may contain shapes other than triangles.

A.2. GCCH

Fig. A.22 illustrates the steps of the GCCH algorithm. Each step is described below.

- **A:** Isosurface (in orange) and the 3D cube (in red) that will be the seed for the continuation method.
- **B:** Generation of all 1D edges τ (in dark red) of the initial 3D cube.
- **C:** Generation of the vertices of the approximation (in bright red) that lie in each τ that intersects the isosurface.
- **D:** Generation of the edges of the approximation (in bright red) using the Combinatorial Skeleton.
- **E:** The process is restarted from step A at the neighboring 3D cubes that share vertices of the approximation. This is repeated until the entire isosurface is covered.
- **F:** Final surface mesh generated.

Comparing panel F of Fig. A.22 with panel F of Fig. A.21, we can see that the GCCH algorithm generates an approximation with fewer vertices and edges, and its vertices are a subset of the vertices generated by the PTA algorithm. In both cases, the generated approximations are not triangulations.

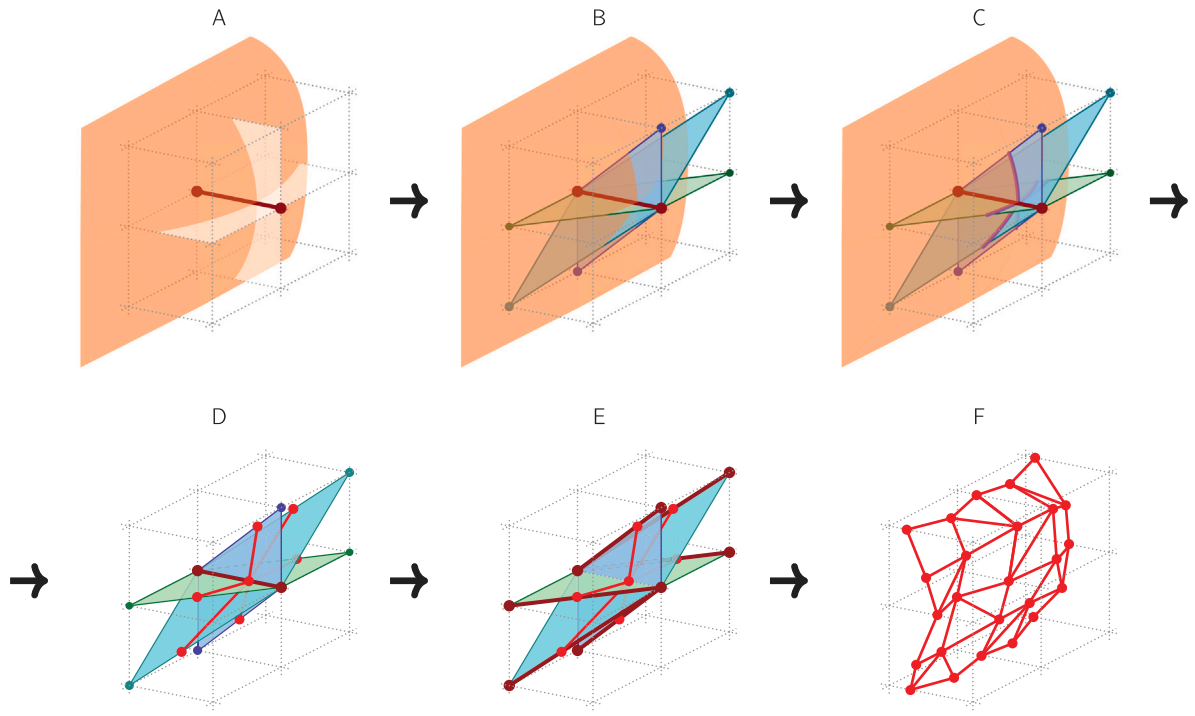

Fig. A.21. Step-by-step example of the PTA algorithm for isosurfacing in the 3D domain. Panels A to F depict each step.

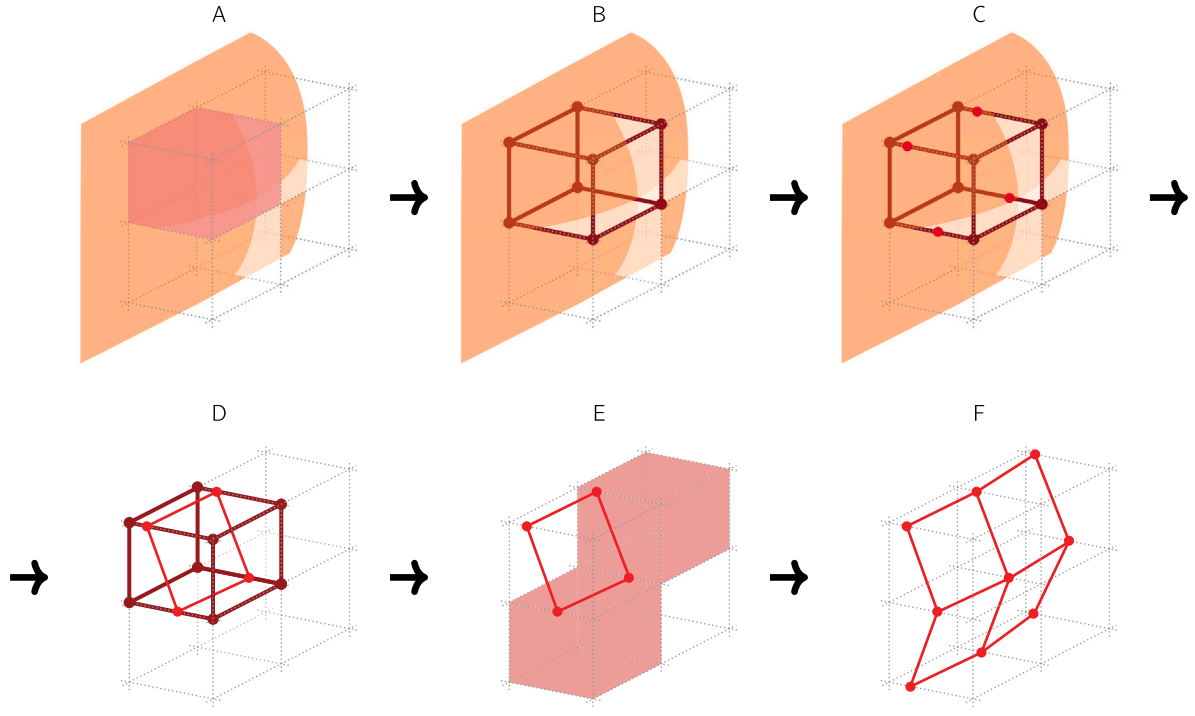

Fig. A.22. Step-by-step example of the GCCH algorithm for isosurfacing in the 3D domain. Panels A to F depict each step.

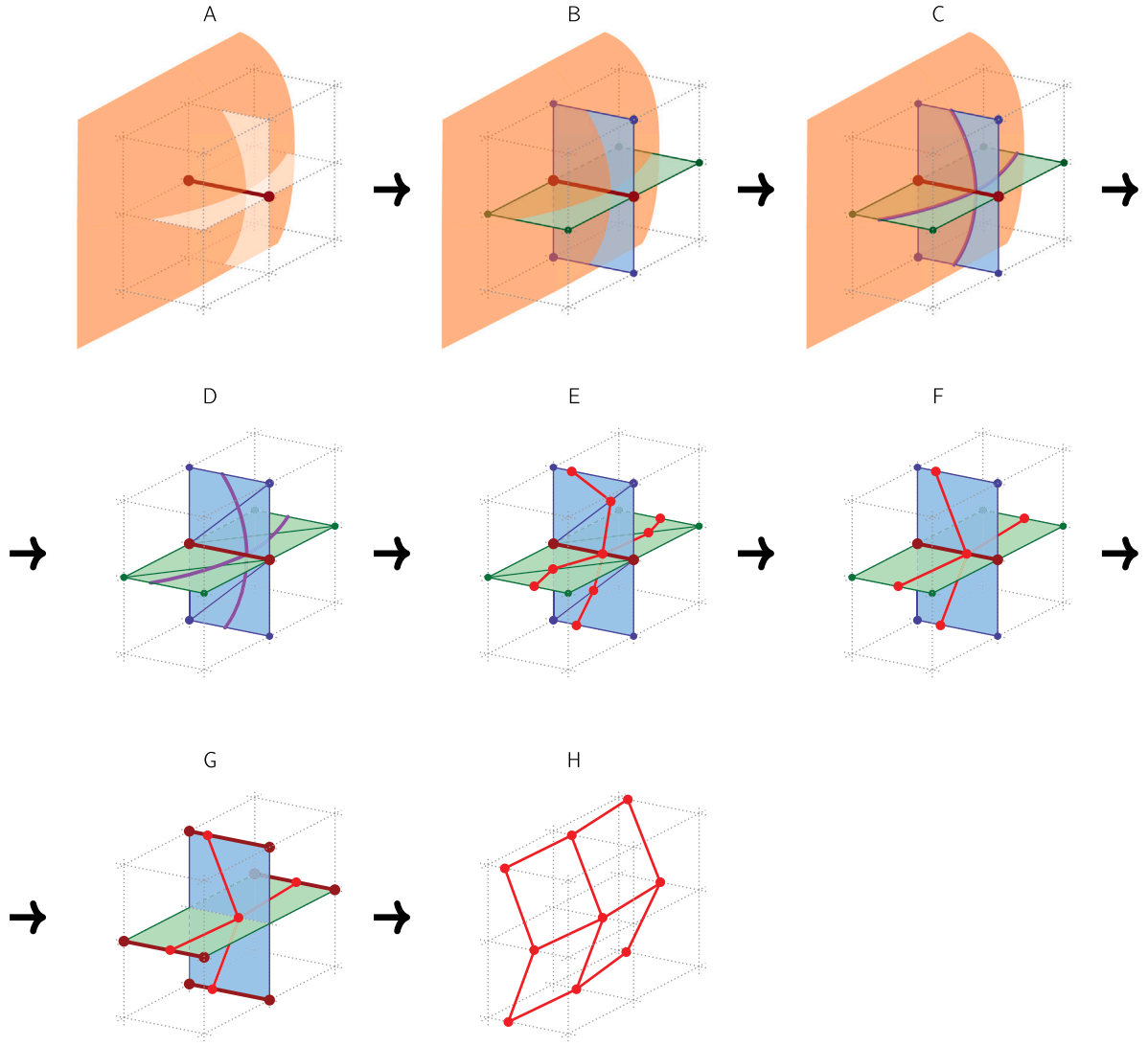

Fig. A.23. Step-by-step example of the FCH algorithm for isosurfacing in the 3D domain. Panels A to H depict each step.

A.3. FCH

Fig. A.23 illustrates the steps of the FCH algorithm. Each step is described below.

- **A:** Isosurface (in orange) and the 1D edge of a 3D cube (in dark red) that will serve as the seed τ for the continuation method.
- **B:** Generation of all 2D square faces of the that contain τ.
- **C:** The intersection of the isosurface with a square face forms a curve (in magenta). These curves will be approximated by line segments.
- **D:** Each square face is subdivided into triangles using the K_1 triangulation.
- **E:** Traversal of each square face, generating the edges the approximation (in bright red) that lie on each triangle.
- **F:** The intermediary vertices of the approximation that lie inside each square face are removed. The edges of the approximation are then formed by one input vertex (that lie on τ) and one output vertex (that lie on a different edge τ' of the square face).
- **G:** The process is restarted from step A at the τ' of each square face (in dark red). This is repeated until the entire isosurface is covered.
- **H:** Final surface mesh generated.

The FCH algorithm employs an approach similar to that of the PTA algorithm, but generates an approximation identical to the one produced by the GCCH algorithm.

Appendix B. Discussion on the computational cost

B.1. Upper bound on the computational cost

Assuming that every structure in the K_1 triangulation that *could* intersect the manifold *will* be processed, the largest possible number of processed structures per n-hypercube sets an upper bound on the computational cost. In all three methods, each processed structure requires solving a $k\times k$ linear system, so this upper bound is directly comparable across the methods. Thus, we will treat the upper bound on the number of processed structures as an upper bound on the computational cost, with the terms $\mathcal{U}_{PTA}$, $\mathcal{U}_{GCCH}$, and $\mathcal{U}_{FCH}$ being used to denote the upper bounds for the PTA, GCCH, and FCH, respectively.

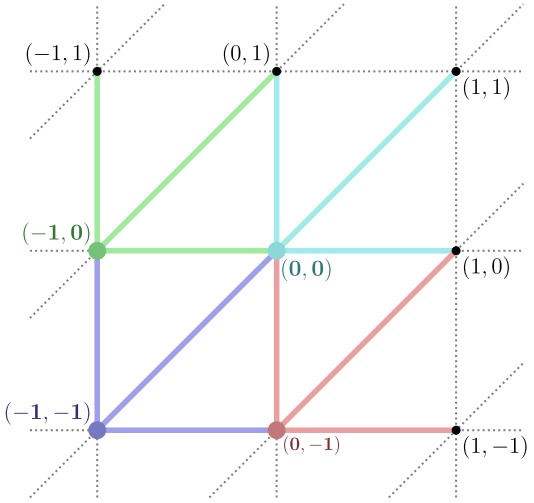

Fig. B.24. Example of a 2D domain triangulated by the K_1 triangulation. The different colors highlight the 1-simplices that would be taken into account for each different 2-hypercube if we were to count all the 1-simplices generated.

B.1.1. PTA

The PTA algorithm assumes that every n-hypercube in the domain is partitioned into $n!$ n-simplices via the K_1 triangulation, and that the k-faces of these n-simplices might intersect the manifold. The application of the Door-in/Door-out principle means that, to find the k-faces that intersect the manifold, we can process the $(k+1)$-dimensional cofaces instead of the k-faces. Therefore, the processed structures on the PTA algorithm are the $(k+1)$-simplices, and we need to compute the total number of $(k+1)$-simplices that the CFK triangulation generates in an n-hypercube.

Note that in the K_1 triangulation, adjacent n-hypercubes may share $(k+1)$-simplices. Simply counting all $(k+1)$-simplices within a single n-hypercube could lead to duplicates when considering the entire domain. However, the PTA algorithm ensures each $(k+1)$-simplex is processed only once, so we should guarantee that there are no redundant counts in our computation. To achieve this, for a given n-hypercube I we take the minimal vertex v_0 as a reference vertex and, using the permutahedral representation, we generate all $(k+1)$-simplices where this permutahedral representation is canonical. This means that all generated $(k+1)$-simplices lie within I and are connected to v_0. Only these $(k+1)$-simplices, which connect to v_0, are considered in the cost calculation.

For example, if we were to count all the 1-simplices generated by the K_1 triangulation in a 2D domain, Fig. B.24 highlights the 1-simplices that would be taken into account for each 2-hypercube: the edges were painted with 4 different colors (blue, red, green and cyan), where each color represents a different square, to indicate which edges would be considered for each square. We can see that all 1-simplices in the domain would be taken into account, except those that lie on the upper boundary of each dimension of the domain. The set representing the upper boundary of the domain is 1 dimension smaller than the domain itself, so in terms of asymptotic complexity the lack of these 1-simplices does not affect the result.

Leveraging the construction rules of the canonical permutahedral representation, we can compute the total number of $(k+1)$-simplices connected to v_0 using purely combinatorial techniques. In this representation of a $(k+1)$-simplex, the $n+1$ vectors $e_1, e_2, \dots, e_n, e_{n+1}$ are grouped into $k+2$ non-empty ordered groups, with the constraint that the vector e_{n+1} must always belong to the last group. This means that the vectors $e_1, \dots, e_n$ can be freely distributed among the $k+2$ groups, provided that each group contains at least one element. The number of ways these $n+1$ vectors can be distributed among these groups (i.e. the number of valid arrangements) gives us the total number of $(k+1)$-simplices connected to v_0.

If we choose to leave only e_{n+1} in the last group, there will be n vectors to be distributed among the first $k+1$ groups. Alternatively, if we choose to place m vectors together with e_{n+1} in the last group, then $n-m$ vectors remain to be distributed among the first $k+1$ groups. Since each group must contain at least one element, we have that $n-m \geq k+1 \Rightarrow 0 \leq m \leq n-k-1$.

For a given m, there are $\binom{n}{m}$ ways to select the vectors that will be placed in the last group. Now, if we were to consider the first $k+1$ groups to be unordered, the number of ways to distribute the remaining $n-m$ vectors into $k+1$ unordered groups can be determined using the Stirling number of the second kind [39,40], $S(n-m, k+1)$, where $S(\cdot,\cdot)$ can be defined as

$$S(a,b) = \frac{1}{b!}\sum_{i=0}^{b}(-1)^{b-i}\binom{b}{i}i^a.$$

Since the groups are ordered, we must account for the permutations of these $k+1$ groups: there are $S(n-m, k+1)$ ways to distribute $n-m$ vectors among $k+1$ non-empty groups, but each configuration can be permuted $(k+1)!$ times. This gives a total of $(k+1)!\ S(n-m, k+1)$ arrangements for the first $k+1$ groups. Thus, for a given m, the total number of arrangements across all $k+2$ groups is:

$$\binom{n}{m}\,(k+1)!\ S(n-m, k+1).$$

Finally, the upper bound on the computational cost of the PTA algorithm, which includes all possible values of m where $0 \leq m \leq n-k-1$, is given by the following sum:

$$(k+1)!\sum_{m=0}^{n-k-1}\binom{n}{m}S(n-m, k+1).$$

A change of variable $j = n-m$ (so $m = n-j$) gives

$$(k+1)!\sum_{j=k+1}^{n}\binom{n}{n-j}S(j, k+1),$$

and, with the equality $\binom{n}{n-j} = \binom{n}{j}$, we have

$$(k+1)!\sum_{j=k+1}^{n}\binom{n}{j}S(j, k+1). \tag{B.1}$$

This expression can be simplified using the following Stirling number identity [39, Section 6.1]:

$$S(a+1, b+1) = \sum_{j=b}^{a}\binom{a}{j}S(j, b). \tag{B.2}$$

Comparing equation (B.1) with (B.2), we can see that $a = n$ and $b = k+1$. Consequently, we obtain the final simplified expression

$$\mathcal{U}_{PTA} = (k+1)!\ S(n+1, k+2). \tag{B.3}$$

It is worth noting that this result could have been obtained directly by considering the distribution of the $n+1$ vectors into $k+2$ non-empty groups (yielding $S(n+1, k+2)$), and then permuting only the first $k+1$ groups (resulting in $(k+1)!$ permutations).

Upper and lower bounds for $S(\cdot,\cdot)$ can be found in [41]. Using the upper bound

$$S(a,b) \leq \frac{b^a}{b!}$$

gives us

$$\mathcal{U}_{PTA} \leq (k+2)^n,$$

and we can say the asymptotic cost of the PTA algorithm is $O\left((k+2)^n\right)$. However, this approximation is most accurate when k is close to 1:

$$\mathcal{U}_{PTA(k\approx 1)} \approx 3^n. \tag{B.4}$$

When $k \approx n$ (meaning k close to $n-1$), we have

$$(k+1)!\ S(n+1,k+2) \approx n!\ S(n+1,n+1) = n!,$$

therefore

$$\mathcal{U}_{PTA(k\approx n)} \approx n!. \tag{B.5}$$

To create an estimate for the intermediary case $k \approx \frac{n}{2}$, let us consider the lower bound [41]:

$$S(a,b) \geq \binom{a}{b}\left(\frac{b}{2}\right)^{a-b}$$

which gives us

$$\mathcal{U}_{PTA} \geq \frac{(n+1)!}{(n-k-1)!}\ \frac{(k+2)^{n-k-2}}{2^{n-k-1}}. \tag{B.6}$$

The Stirling's approximation [39, Section 9.3]

$$a! \approx \sqrt{2\pi a}\left(\frac{a}{e}\right)^a$$

can be used to approximate both factorials, yielding

$$\mathcal{U}_{PTA} \gtrsim \sqrt{\frac{n+1}{n-k-1}}\ \frac{(k+2)^{n-k-2}}{2^{n-k-1}}\ \frac{(n+1)^{n+1}}{(n-k-1)^{n-k-1}}\ e^{-(k+2)}.$$

Simplifying this expression for $k \approx \frac{n}{2}$ and high n gives us

$$\mathcal{U}_{PTA\left(k\approx\frac{n}{2}\right)} \gtrsim \sqrt{2}\left(\frac{n^2}{2e}\right)^{\frac{n}{2}}, \tag{B.7}$$

which can later be compared against the other algorithms.

B.1.2. GCCH

The GCCH algorithm always processes all k-simplices that lie on the k-faces of an n-hypercube. Hence, the processed structures in the GCCH algorithm are these k-simplices, and the upper bound on the computational cost is already given by Eq. (12):

$$\mathcal{U}_{GCCH} = \frac{n!}{(n-k)!}\ 2^{n-k}. \tag{B.8}$$

We have the inequality

$$\mathcal{U}_{GCCH} \leq 2^{n-k}\ n^k,$$

indicating that the algorithm has asymptotic complexity of $O\left(2^{n-k}\ n^k\right)$, which is a more accurate approximation when k is close to 1. Assuming high n:

$$\mathcal{U}_{GCCH(k\approx 1)} \approx 2^n\ n. \tag{B.9}$$

For values of $n-k$ close to 1, the expression (B.8) simplifies to

$$\mathcal{U}_{GCCH(k\approx n)} \approx 2\ n!. \tag{B.10}$$

For the general case, the Stirling's approximation can be employed on both factorials from Eq. (B.8), yielding

$$\mathcal{U}_{GCCH} \approx \sqrt{\frac{n}{n-k}}\left(\frac{2n}{n-k}\right)^{n-k}\left(\frac{n}{e}\right)^k.$$

Then, for the intermediary case $k \approx \frac{n}{2}$, we have

$$\mathcal{U}_{GCCH\left(k\approx\frac{n}{2}\right)} \approx \sqrt{2}\left(\frac{4n}{e}\right)^{\frac{n}{2}}. \tag{B.11}$$

B.1.3. FHC

The FCH algorithm assumes that every n-hypercube in the domain is partitioned into $n!$ n-simplices via the K_1 triangulation, but only the k-simplices that lie on the $(k+1)$-faces of an n-hypercube might intersect the manifold. Here, the same approach is used as in PTA: for a given n-hypercube I, only the $(k+1)$-faces connected to the minimal vertex v_0 are taken into account, and since the Door-in/Door-out principle is applied, we can count the $(k+1)$-simplices instead of the k-simplices. Thus, the processed structures in the FCH algorithm are these $(k+1)$-simplices connected to v_0.

The label for a $(k+1)$-face of an n-hypercube consists of $2n$ bits, where $n-k-1$ bits are set (equal to 1) and the remaining bits are unset (equal to 0). To count only the $(k+1)$-faces connected to v_0, the n most significant bits of the label must be 0, which implies that $n-k-1$ of the n least significant bits must be set. The number of $(k+1)$-faces connected to v_0 in an n-hypercube is then given by the number of ways to choose $n-k-1$ unique elements out of n:

$$\binom{n}{n-k-1}.$$

Using the K_1 triangulation, each $(k+1)$-face of an n-hypercube is subdivided into $(k+1)!$ $(k+1)$-simplices that contain v_0. Therefore, the total number of $(k+1)$-simplices connected to v_0 is

$$\binom{n}{n-k-1}(k+1)!.$$

Simplifying this expression, we obtain the upper bound on the computational cost of the FCH algorithm:

$$\mathcal{U}_{FCH} = \frac{n!}{(n-k-1)!}. \tag{B.12}$$

It follows that

$$\mathcal{U}_{FCH} \leq n^{k+1},$$

which implies an asymptotic cost of $O\left(n^{k+1}\right)$, particularly when k is small:

$$\mathcal{U}_{FCH(k\approx 1)} \approx n^2. \tag{B.13}$$

Conversely, for $n-k$ close to 1, the expression (B.12) simplifies to

$$\mathcal{U}_{FCH(k\approx n)} \approx n!. \tag{B.14}$$

Applying the Stirling's approximation to (B.12) yields

$$\mathcal{U}_{FCH} \approx \sqrt{\frac{n}{n-k-1}}\left(\frac{n}{n-k-1}\right)^{n-k-1}\left(\frac{n}{e}\right)^{k+1},$$

which lets us derive an approximation for the intermediary case $k \approx \frac{n}{2}$ (assuming high n):

$$\mathcal{U}_{FCH\left(k\approx\frac{n}{2}\right)} \approx \sqrt{2}\left(\frac{2n}{e}\right)^{\frac{n}{2}}. \tag{B.15}$$

B.2. Upper bound comparison

Eqs. (B.3), (B.8), and (B.12) give us

$$\mathcal{U}_{PTA} = (k+1)!\ S(n+1,k+2),$$
$$\mathcal{U}_{GCCH} = \frac{n!}{(n-k)!}\ 2^{n-k},$$
$$\mathcal{U}_{FCH} = \frac{n!}{(n-k-1)!}.$$

Based on inequality (B.6), we have:

$$\mathcal{U}_{PTA} \geq \frac{(n+1)!}{(n-k-1)!}\ \frac{(k+2)^{n-k-2}}{2^{n-k-1}}.$$

It can be verified that for all n and k such that $k \geq 2$ and $n-k \geq 2$, the following relation holds:

$$\mathcal{U}_{PTA} \geq \frac{(n+1)!}{(n-k-1)!}\ \frac{(k+2)^{n-k-2}}{2^{n-k-1}} > \mathcal{U}_{GCCH} > \mathcal{U}_{FCH}.$$

We begin by verifying that $\mathcal{U}_{PTA} > \mathcal{U}_{GCCH}$. Substituting the expressions:

$$\mathcal{U}_{PTA} \geq \frac{(n+1)!}{(n-k-1)!}\ \frac{(k+2)^{n-k-2}}{2^{n-k-1}} > \frac{n!}{(n-k)!}\ 2^{(n-k)} = \mathcal{U}_{GCCH}.$$

Letting $m = n-k$, we rewrite the inequality as:

$$\frac{(n+1)!}{(m-1)!}\ \frac{(k+2)^{m-2}}{2^{m-1}} > \frac{n!}{m!}\ 2^m.$$

Simplifying:

$$m(n+1)(k+2)^{m-2} > 2^{2m-1}.$$

Table B.2
Upper bound on the computational cost and approximations assuming high n.

	PTA		GCCH		FCH
Cost equation	$(k+1)!\ S(n+1,k+2)$		$2^{n-k}\ \frac{n!}{(n-k)!}$		$\frac{n!}{(n-k-1)!}$
Approximation for $k \approx 1$	3^n		$2^n\ n$		n^2
Approximation for $k \approx \frac{n}{2}$	$\sqrt{2}\left(\frac{n^2}{2e}\right)^{\frac{n}{2}}$		$\sqrt{2}\left(\frac{4n}{e}\right)^{\frac{n}{2}}$		$\sqrt{2}\left(\frac{2n}{e}\right)^{\frac{n}{2}}$
Approximation for $k \approx n$	$n!$		$2\ n!$		$n!$
Relationship when $k \geq 2$ and $n-k \geq 2$	$\mathcal{U}_{PTA}$	$>$	$\mathcal{U}_{GCCH}$	$>$	$\mathcal{U}_{FCH}$

Now, using $n = m + k$, the inequality becomes:

$$\underbrace{m(m+k+1)(k+2)^{m-2}}_{\text{LHS}} > \underbrace{2^{2m-1}}_{\text{RHS}}.$$

The left-hand side (LHS) will be analyzed first. If $k \geq 2$, then:

$$(k+2)^{m-2} \geq (2+2)^{m-2} = \left(2^2\right)^{m-2} = 2^{2m-4}. \tag{B.16}$$

Also, if $k \geq 2$ and $m \geq 2$:

$$m(m+k+1) \geq 2(2+2+1) > 2(2+2) = 2^3. \tag{B.17}$$

So, combining (B.16) and (B.17), for the left-hand side we get:

$$m(m+k+1)(k+2)^{m-2} > 2^3\ 2^{2m-4} = 2^{2m-1},$$

which matches the right-hand side (RHS). Therefore, the inequality $\mathcal{U}_{PTA} > \mathcal{U}_{GCCH}$ holds for all $k \geq 2$ and $n-k \geq 2$.

Now let us verify that $\mathcal{U}_{GCCH} > \mathcal{U}_{FCH}$. Substituting the expressions:

$$\mathcal{U}_{GCCH} = \frac{n!}{(n-k)!}\ 2^{n-k} > \frac{n!}{(n-k-1)!} = \mathcal{U}_{FCH}.$$

Again, letting $m = n - k$, it simplifies to:

$$2^m > m,$$

which is true for all $m \geq 1$. Thus, the inequality $\mathcal{U}_{GCCH} > \mathcal{U}_{FCH}$ holds whenever $n-k \geq 1$, and we conclude that $\mathcal{U}_{PTA} > \mathcal{U}_{GCCH} > \mathcal{U}_{FCH}$ is valid for all $k \geq 2$ and $n-k \geq 2$. When $n-k$ is close to 1, we already have the approximations given in Eqs. (B.5), (B.10), and (B.14); and for k close to 1, the approximations are provided in Eqs. (B.4), (B.9), and (B.13).

These results show that, based on the upper bound analysis, the FCH algorithm is the most efficient among the three. Table B.2 summarizes the approximations of each algorithm, showing that:

- When k is close to n, all three algorithms have very similar costs, approaching $n!$;
- As k decreases, FCH becomes the most efficient algorithm, with the lowest computational complexity.

B.3. Worst-case scenario

While the calculated upper bound establishes a theoretical limit for the computational cost, reaching this limit may not be feasible due to inherent mathematical constraints. For instance, the Door-in/Door-out principle limits to 0 or 2 the number of k-faces that can intersect the manifold in a $(k+1)$-simplex. In this section, we present a method to artificially construct a feasible case with a high computational cost, though without guaranteeing that it represents the absolute worst-case scenario. Consequently, the actual worst-case cost will fall somewhere between this feasible configuration and the previously calculated upper bound. The objective of this analysis is to assess how tight the upper bounds calculated in Appendix B.1 are, and to verify if the relationships in Table B.2 hold true for actually feasible cases.

As in the previous section, our focus remains on calculating the number of processed structures within the unit n-hypercube that are connected to v_0.

B.3.1. PTA

Our goal is to artificially construct a function $F : \mathbb{R}^n \to \mathbb{R}^k$ that results in a high number of processed structures, which requires F to produce a large number of intersections between k-simplices and the manifold. Since F is evaluated only at the vertices of a n-hypercube, we will define F exclusively at these points for our artificial case.

As discussed in Section 2.2, a k-simplex intersects the manifold if all components of λ, obtained from Eq. (3), are positive. Note that it is always possible to select $k+1$ distinct values in $\mathbb{R}^k$ (the codomain of F) that satisfy this condition. Therefore, by assigning these selected values to the vertices of a k-simplex, we ensure that this k-simplex will intersect the manifold. Conversely, if F at the vertices of a k-simplex can yield only these $k+1$ distinct values, we can verify manifold intersection by checking whether each vertex corresponds to a unique value.

Let $\{L_0, L_1, \ldots, L_k\}$ be a set of $k+1$ distinct values in the codomain of F that satisfy the intersection condition. We will assign these values to all vertices of the unit n-hypercube, meaning that F evaluated at any vertex will yield one of these values. For simplicity, we treat each L_i as a label that is assigned to a vertex, allowing us to work with generic labels instead of specific values in $\mathbb{R}^k$.

In the simplices generated by the CFK triangulation, we define a k-simplex as *fully-labeled* if it contains $k+1$ distinct labels, which implies that this simplex intersects the manifold. Furthermore, a $(k+1)$-simplex containing $k+1$ distinct labels has exactly two k-faces that are fully-labeled, following the Door-in/Door-out principle. We also refer to such a $(k+1)$-simplex as *fully-labeled*.

With this framework, we can pose the problem as follows: given an assignment of $k+1$ labels across the vertices of the unit n-hypercube, how many fully-labeled $(k+1)$-simplices connected to v_0 exist? The answer corresponds to the number of processed structures for the PTA algorithm.

As detailed in Section 2.1, the n-simplices generated by the CFK triangulation of the unit n-hypercube are formed through permutations of α in a sequence of vectors:

$$\underbrace{\mathbf{0}}_{v_0(\alpha)} \xrightarrow{e_{\alpha(1)}} \underbrace{v_0(\alpha) + e_{\alpha(1)}}_{v_1(\alpha)} \xrightarrow{e_{\alpha(2)}} \underbrace{v_1(\alpha) + e_{\alpha(2)}}_{v_2(\alpha)} \xrightarrow{e_{\alpha(3)}} \ldots$$

$$\ldots \xrightarrow{e_{\alpha(n)}} \underbrace{\left(v_{n-1}(\alpha) + e_{\alpha(n)}\right)}_{v_n(\alpha)}$$

Because all generated n-simplices share the same geometric shape but are placed at different positions within the n-hypercube, we can view them as mirrored versions of a standard n-simplex defined by a specific α. To simplify both label assignment and computational analysis, we will first assign labels to the vertices of this standard n-simplex, and the labels for all other n-simplices can then be derived through the corresponding permutations of α. For example, when $n = 4$ and $k = 2$, a possible label configuration is:

$$\underbrace{v_0}_{L_0} \xrightarrow{e_{\alpha(1)}} \underbrace{v_1}_{L_1} \xrightarrow{e_{\alpha(2)}} \underbrace{v_2}_{L_1} \xrightarrow{e_{\alpha(3)}} \underbrace{v_3}_{L_2} \xrightarrow{e_{\alpha(4)}} \underbrace{v_4}_{L_1}$$

and the permutations of α will automatically determine the label assignments for all vertices within the unit n-hypercube.

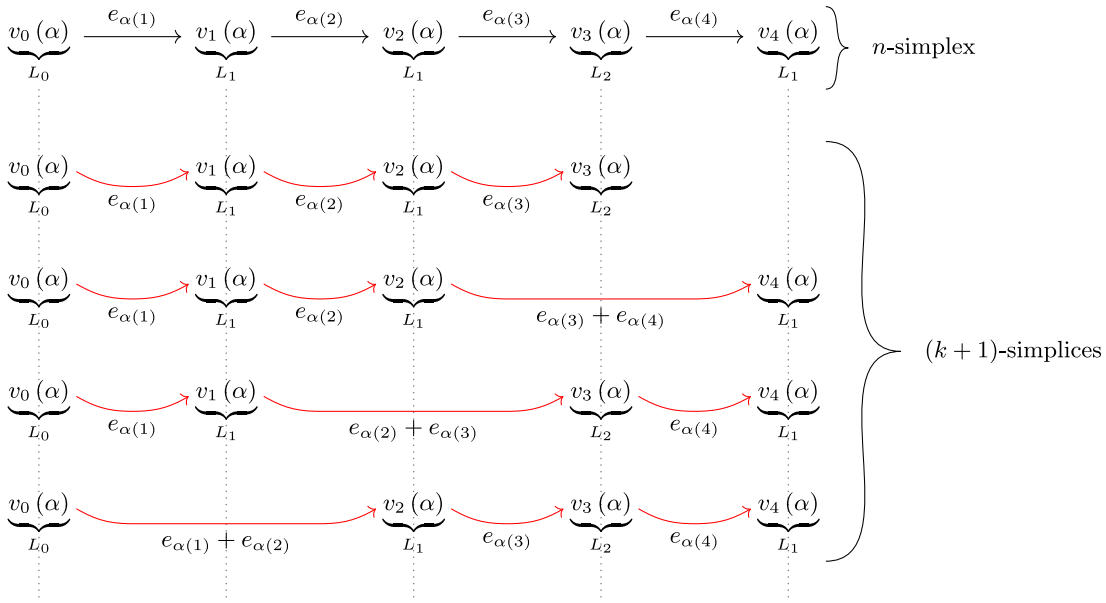

Fig. B.25. Diagram that illustrates all possible 3-faces containing vertex v_0 that can be formed from a 4-simplex.

As described in Section 2.3 and Appendix B.1.1, a $(k+1)$-simplex that includes v_0 is formed by a sequence of $k+1$ vectors originating from v_0. Each vector in this sequence can be a combination of multiple subsequent vectors from the parent n-simplex. To illustrate this, Fig. B.25 presents all possible ways to generate sequences of vectors yielding $(k+1)$-simplices for $n=4$ and $k=2$.

From these sequences, we will consider only those that generate fully-labeled $(k+1)$-simplices, meaning sequences that include vertices with all $k+1$ distinct labels. To compute the total number of fully-labeled $(k+1)$-simplices, we must account for all permutations of α. Let u be a vector in a sequence; if u comprises m vectors from the canonical basis, the number of configurations for u equals the number of ways to select m vectors from the available canonical basis vectors. Consequently, the total number of arrangements for each sequence represents the total number of fully-labeled $(k+1)$-simplices connected to v_0. Continuing with the example where $n=4$ and $k=2$, the diagram in Fig. B.26 illustrates the total number of possible arrangements that generate fully-labeled $(k+1)$-simplices. In this example, we observe that the total number of processed structures is 48.

To summarize, our artificial case involves assigning a set of $k+1$ distinct labels to the vertices of a standard n-simplex and identifying the sequences that form fully-labeled $(k+1)$-simplices. Through experimentation, we found that a high number of processed structures can be achieved when the labels are distributed among the vertices of the standard n-simplex as follows:

- L_0 is assigned exclusively to v_0.
- Let $\#(L_i)$ denote the number of times label L_i appears in the sequence. Then:

$$\#(L_0) = 1;$$
$$\#(L_i) = \left\lfloor \frac{n+1-\sum_{j=0}^{i-1}\#(L_j)}{k+1-i} \right\rfloor, \quad \text{for } 1 \le i \le k.$$

- Labels L_1 to L_k are assigned in increasing order to the vertices v_1 through v_n.

As an example, Table B.3 shows the label assignments for $n=10$ and $1 \le k \le 9$.

To identify sequences that generate fully-labeled $(k+1)$-simplices, we implemented a recursive search algorithm that explores all possible sequences in the n-simplex, checking whether each sequence is fully labeled. For each valid sequence, we then compute the total number of possible arrangements of vectors, as previously discussed. The obtained results are presented in Appendix B.4.

Table B.3
Label assignment used for $n=10$.

	Label sequence in the n-simplex
$k=1$	0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
$k=2$	0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2
$k=3$	0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3
$k=4$	0, 1, 1, 2, 2, 3, 3, 3, 4, 4, 4
$k=5$	0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5
$k=6$	0, 1, 2, 3, 3, 4, 4, 5, 5, 6, 6
$k=7$	0, 1, 2, 3, 4, 5, 5, 6, 6, 7, 7
$k=8$	0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 8
$k=9$	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 9

B.3.2. GCCH

The GCCH processes a fixed number of structures per n-hypercube, regardless of the manifold being approximated. Therefore, its worst-case scenario is identical to the upper bound calculated in Appendix B.1.2.

B.3.3. FCH

In the FCH, evaluations are restricted to the $(k+1)$-hypercubes that are faces of the n-hypercube and that are connected to v_0. Applying the same analysis used for the PTA to each of these $(k+1)$-hypercubes individually, we can demonstrate that all $(k+1)$-simplices within these hypercubes will be evaluated. Consequently, the worst-case scenario for the FCH is equal to the upper bound calculated in Appendix B.1.3.

B.4. Worst-case comparison

Fig. B.27 illustrates the worst-case comparison of the three methods – PTA, GCCH, and FCH – for the specific values of $n \in \{3, 4, 5, 10, 20, 40\}$ and $1 \le k \le n-1$. It is important to highlight that the data in Fig. B.27 refer to a single n-hypercube, meaning they represent a cost normalized by the number of n-hypercubes in the domain. While the theoretical upper bounds for GCCH and FCH (calculated in Appendix B.1) exactly match their worst-case costs, the data shown in Fig. B.27 indicate that

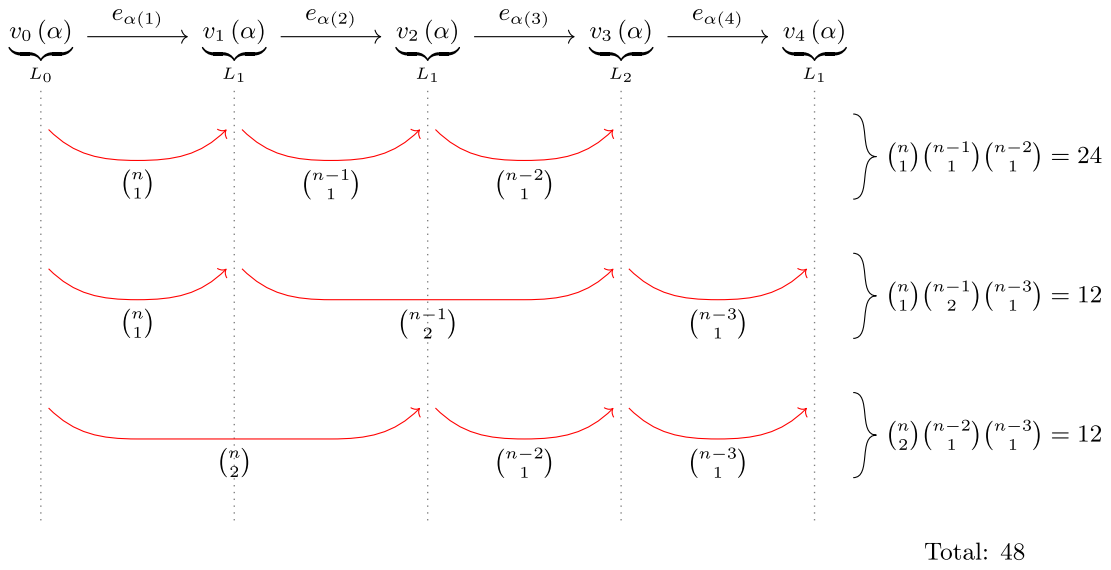

Fig. B.26. Given a particular label assignment, this diagram illustrates the number of all possible fully-labeled 3-faces containing vertex v_0 that can be formed from a 4-simplex.

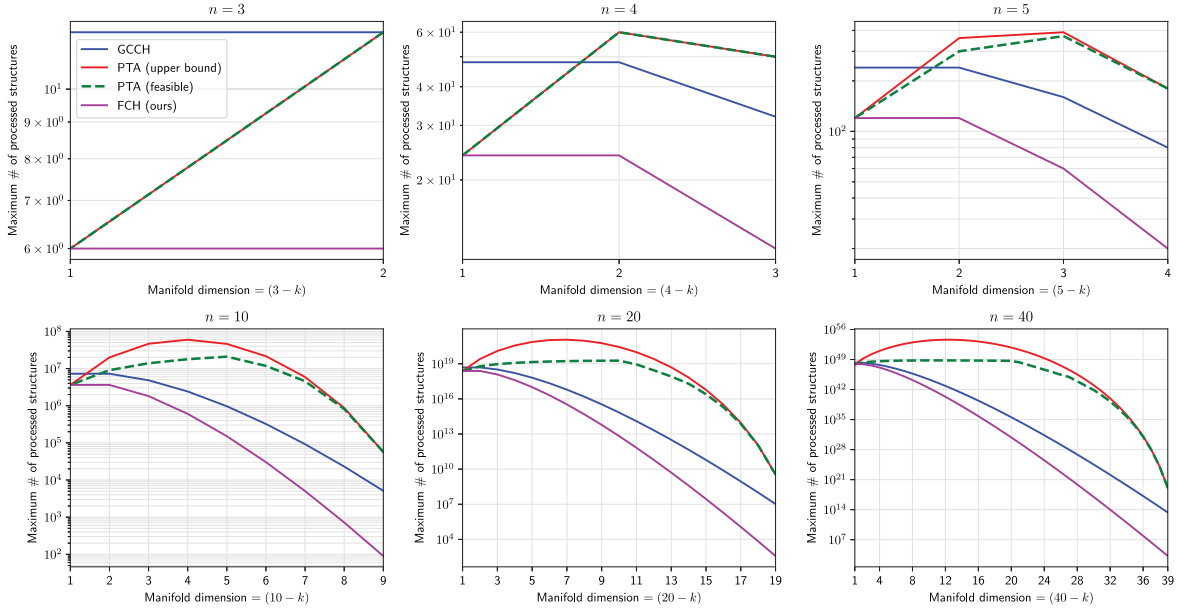

Fig. B.27. Worst-case comparison of the computational cost between the three methods for $n \in \{3, 4, 5, 10, 20, 40\}$ and $1 \leq k \leq n-1$. For the PTA, its worst-case lies somewhere in between its upper bound and the feasible case.

the bound for PTA is also a reasonable approximation for its worst-case cost. This is further reinforced by the consistent cost relationships observed between the algorithms when comparing Fig. B.27 with Table B.2. For other comparisons in situations that are not guaranteed to be worst-case, see the experimental results in Section 5.

To conclude the comparisons, we can say that the FCH has lower complexity compared to the PTA because it is more selective regarding the $(k+1)$-simplices it processes — the FCH can approximate the entire manifold by processing only the $(k+1)$-simplices located on the $(k+1)$-faces of the n-hypercubes, unlike the PTA, which processes all $(k+1)$-simplices that intersect the manifold. Furthermore, the FCH also exhibits lower complexity compared to the GCCH because the application of the Door-in/Door-out principle ensures that structures not intersecting the manifold are not processed, reducing the total number of processed structures.

Data availability

No data was used for the research described in the article.

References

[1] Gomes A, Voiculescu I, Jorge J, Wyvill B, Galbraith C. Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms. Springer London; 2009, http://dx.doi.org/10.1007/978-1-84882-406-5.

[2] Büscher KJ, Degel JP, Oellerich J. A comprehensive survey of isocontouring methods: Applications, limitations and perspectives. Algorithms 2024;17(2). http://dx.doi.org/10.3390/a17020083.

[3] Allgower EL, Schmidt PH. Piecewise linear approximation of solution manifolds for nonlinear systems of equations. In: Hammer G, Pallaschke D, editors. Selected topics in operations research and mathematical economics. Berlin, Heidelberg:

Springer Berlin Heidelberg; 1984, p. 339–47. http://dx.doi.org/10.1007/978-3-642-45567-4_26.
[4] Zhou M, Xiao M, Zhang Y, Gao J, Gao L. Marching cubes-based isogeometric topology optimization method with parametric level set. Appl Math Model 2022;107:275–95. http://dx.doi.org/10.1016/j.apm.2022.02.032.
[5] Anderson JT, Williams DM, Corrigan A. Surface and hypersurface meshing techniques for space–time finite element methods. Comput.- Aided Des 2023;163:103574. http://dx.doi.org/10.1016/j.cad.2023.103574.
[6] Wu J, Zhang D, Yi X, Luo F, Zhang T. Improved marching cubes algorithm for 3d multi-slice spiral computed tomography in the diagnosis of bone and joint diseases. J Med Imaging Heal Inform. 2019;9(5):962–8. http://dx.doi.org/10.1166/jmihi.2019.2684.
[7] Nugroho PA, Basuki DK, Sigit R. 3D heart image reconstruction and visualization with marching cubes algorithm. In: 2016 international conference on knowledge creation and intelligent computing. KCIC, IEEE; 2016, p. 35–41. http://dx.doi.org/10.1109/KCIC.2016.7883622.
[8] Cirne MVM, Pedrini H. Marching cubes technique for volumetric visualization accelerated with graphics processing units. J Braz Comput Soc 2013/09/01;19(3):223–33. http://dx.doi.org/10.1007/s13173-012-0097-z.
[9] Carr JC, Beatson RK, Cherrie JB, Mitchell TJ, Fright WR, McCallum BC, Evans TR. Reconstruction and representation of 3D objects with radial basis functions. In: Proceedings of the 28th annual conference on computer graphics and interactive techniques. SIGGRAPH '01, New York, NY, USA: Association for Computing Machinery; 2001, p. 67–76. http://dx.doi.org/10.1145/383259.383266.
[10] Ohtake Y, Belyaev A, Alexa M, Turk G, Seidel H-P. Multi-level partition of unity implicits. ACM Trans Graph 2003;22(3):463–70. http://dx.doi.org/10.1145/882262.882293.
[11] Kazhdan M, Hoppe H. Screened poisson surface reconstruction. ACM Trans Graph 2013;32(3). http://dx.doi.org/10.1145/2487228.2487237.
[12] Incahuanaco F, Paiva A. Surface reconstruction method for particle-based fluids using discrete indicator functions. Comput Graph 2023.
[13] Breen DE, Mauch S, Whitaker RT. 3D scan conversion of CSG models into distance volumes. In: IEEE symposium on volume visualization (cat. no.989EX300). SVV-98, IEEE; 1998, p. 7–14. http://dx.doi.org/10.1109/SVV.1998.729579.
[14] Fang S, Srinivasan R. Volumetric-CSG – a model-based volume visualization approach. J WSCG 1998;6.
[15] Chen Z, Tagliasacchi A, Funkhouser T, Zhang H. Neural dual contouring. ACM Trans Graph 2022;41(4):1–13. http://dx.doi.org/10.1145/3528223.3530108.
[16] Liao Y, Donné S, Geiger A. Deep marching cubes: Learning explicit surface representations. In: 2018 IEEE/CVF conference on computer vision and pattern recognition. IEEE; 2018, p. 2916–25. http://dx.doi.org/10.1109/CVPR.2018.00308.
[17] Shen T, Munkberg J, Hasselgren J, Yin K, Wang Z, Chen W, Gojcic Z, Fidler S, Sharp N, Gao J. Flexible isosurface extraction for gradient-based mesh optimization. ACM Trans Graph 2023;42(4). http://dx.doi.org/10.1145/3592430.
[18] Bloomenthal J. Polygonization of implicit surfaces. Comput Aided Geom Design 1988;5(4):341–55. http://dx.doi.org/10.1016/0167-8396(88)90013-1.
[19] Schroeder W, Maynard R, Geveci B. Flying edges: A high-performance scalable isocontouring algorithm. In: 2015 IEEE 5th symposium on large data analysis and visualization. LDAV, IEEE; 2015, p. 33–40. http://dx.doi.org/10.1109/LDAV.2015.7348069.
[20] Dobkin DP, Wilks AR, Levy SVF, Thurston WP. Contour tracing by piecewise linear approximations. ACM Trans Graph 1990;9(4):389–423. http://dx.doi.org/10.1145/88560.88575.
[21] Bhaniramka P, Wenger R, Crawfis R. Isosurfacing in higher dimensions. In: Proceedings visualization 2000. VIS 2000 (cat. no.00CH37145). IEEE; 2000, p. 267–73. http://dx.doi.org/10.1109/VISUAL.2000.885704.
[22] Lorensen WE, Cline HE. Marching cubes: A high resolution 3D surface construction algorithm. SIGGRAPH Comput Graph 1987;21(4):163–9. http://dx.doi.org/10.1145/37402.37422.
[23] Doi A, Koide A. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. IEICE Trans Inf Syst 1991;74(1):214–24.
[24] Guéziec A, Hummel R. Exploiting triangulated surface extraction using tetrahedral decomposition. IEEE Trans Vis Comput Graphics 1995;1(4):328–42. http://dx.doi.org/10.1109/2945.485620.
[25] Treece GM, Prager RW, Gee AH. Regularised marching tetrahedra: improved iso-surface extraction. Comput Graph 1999.
[26] Allgower EL, Georg K. Numerical continuation methods. Heidelberg: Springer Berlin; 1990, http://dx.doi.org/10.1007/978-3-642-61257-2,
[27] Brodzik ML. The computation of simplicial approximations of implicitly defined p-dimensional manifolds. Comput Math Appl 1998;36(6):93–113. http://dx.doi.org/10.1016/S0898-1221(98)00164-3.
[28] Boissonnat J-D, Kachanovich S, Wintraecken M. Tracing isomanifolds in $\mathbb{R}^d$ in time polynomial in d using Coxeter–Freudenthal–Kuhn triangulations. SIAM J Comput 2023;52(2):452–86. http://dx.doi.org/10.1137/21M1412918.
[29] Castelo A, Nakassima G, Bueno LM, Gameiro M. A generalized combinatorial marching hypercube algorithm. Comput Appl Math 2024;43(3):1–23. http://dx.doi.org/10.1007/s40314-024-02627-4.
[30] Coxeter HSM. Discrete groups generated by reflections. Ann Math 1934;35(3):588–621.
[31] Freudenthal H. Simplizialzerlegungen von Beschränkter Flachheit. Ann Math 1942;43(3):580–2.
[32] Kuhn HW. Simplicial approximation of fixed points. Proc Natl Acad Sci 1968;61(4):1238–42. http://dx.doi.org/10.1073/pnas.61.4.1238.
[33] Castelo A, Moutinho Bueno L, Gameiro M. A combinatorial marching hypercubes algorithm. Comput Graph 2022.
[34] Spivak M. Calculus On Manifolds. CRC Press; 1965, http://dx.doi.org/10.1201/9780429501906.
[35] Edelsbrunner H. Geometry and topology for mesh generation. Cambridge monographs on applied and computational mathematics, Cambridge University Press; 2001, http://dx.doi.org/10.1017/cbo9780511530067.
[36] Todd MJ. The computation of fixed points and applications. Springer Berlin Heidelberg; 1976, http://dx.doi.org/10.1007/978-3-642-50327-6,
[37] Eaves BC. A course in triangulations for solving equations with deformations. Heidelberg: Springer Berlin; 1984, http://dx.doi.org/10.1007/978-3-642-46516-1,
[38] Bittner L. Simplicial methods for the solution of systems of nonlinear equations. ZAMM - J Appl Math Mech / Z Angew Math Mech 1976;56(2–3):65–73. http://dx.doi.org/10.1002/zamm.19760560202.
[39] Graham RL, Knuth DE, Patashnik O. Concrete Mathematics: A Foundation for Computer Science. second ed.. USA: Addison-Wesley Longman Publishing Co. Inc.; 1994.
[40] Moll VH. Numbers and functions. Student mathematical library, Providence, RI: American Mathematical Society; 2012.
[41] Gismatullin J, Tardivel P. Beta distribution and associated stirling numbers of the second kind. Probab Math Statist 2024;44(1):119–32. http://dx.doi.org/10.37190/0208-4147.00156.