Pointerless Implementation of Hierarchical Simplicial Meshes and Efficient Neighbor Finding in Arbitrary Dimensions*

F. Betul Atalay
Department of Computer Science
University of Maryland, College Park, MD
betul@cs.umd.edu.

David M. Mount
Department of Computer Science
and Institute for Advanced Computer Studies
University of Maryland, College Park, MD
mount@cs.umd.edu.

Abstract

We describe a pointerless representation of hierarchical regular simplicial meshes, based on a bisection approach proposed by Maubach. We introduce a new labeling scheme, called an LPT code, that uniquely encodes each simplex of the hierarchy. We present rules to efficiently compute the neighbors of a given simplex through the use of these codes. In addition, we show how to traverse the associated tree and how to answer point location and interpolation queries.Our system works in arbitrary dimensions.

1 Introduction

Hierarchical simplicial meshes have been widely used in various application areas such as finite element computations, scientific visualization and geometric modeling. There has been considerable amount of work in simplicial mesh refinement, particularly in 2- and 3-dimensions, and a number of different refinement techniques have been proposed [7, 24, 4, 9, 20, 21, 22, 28]. Because of

*This material is based upon work supported by the National Science Foundation under Grant No. 0098151.

the need to handle data sets with a temporal component, there is growing interest in higher dimensional meshes. In this paper, we build on the longest edge bisection method proposed by Mitchell [24], and extended to arbitrary dimensions by Maubach [22].

A hierarchical mesh is said to be *regular*, if the vertices of the mesh are regularly distributed and the process by which a cell is subdivided is identical for all cells. Maubach developed a simple bisection algorithm based on a particular ordering of vertices and presented a mathematically rigorous analysis of the geometric structure of the hierarchical regular simplicial meshes in any dimension d [22]. Each element of such a mesh is a d-simplex, that is the convex hull of $d+1$ affinely independent points [12]. The mesh is generated by a process of repeated bisection applied to a hypercube initially subdivided into $d!$ congruent simplices. The subdivision pattern repeats itself on a smaller scale at every d levels. Whenever a simplex is bisected, some of its neighboring simplices may need to be bisected as well, in order to guarantee that the entire subdivision is *compatible*. Intuitively, a *compatible* subdivision is a subdivision in which pairs of neighboring cells meet along a single common face. A compatible simplicial subdivision is also referred to as a *simplicial complex* [25]. (See Fig. 1 for a 2-dimensional example.) Compatibility is important, since otherwise, cracks occur along faces of the subdivision, which in turn present problems when using the mesh for interpolation.

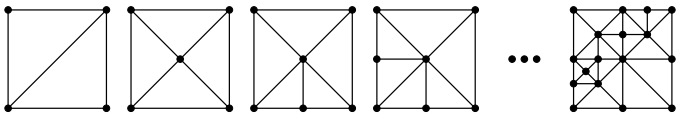


Fig. 1: Compatible simplicial mesh in the plane.

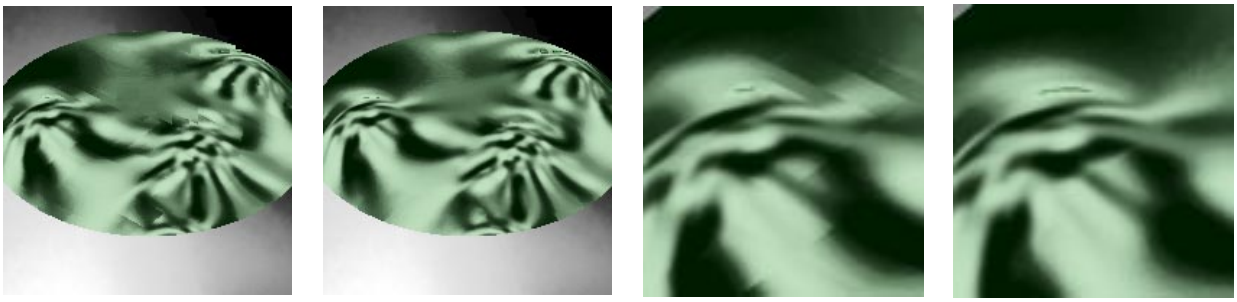
In computer graphics, adaptively refined regular meshes in 2- or 3-dimensions have been of more interest for their use in realistic surface and volume rendering [30, 14, 26]. In many such applications, efficiency of various operations such as traversal and neighbor finding on the mesh is most desired. Based on the 3-dimensional version of Maubach's method, Hebert [16] presented a more efficient symbolic implementation of regular tetrahedral meshes by introducing an addressing scheme that allows unique labeling of the tetrahedra in the mesh, and he showed how to compute face neighbors of a tetrahedron based on its label. Hebert's addressing scheme could be generalized to higher dimensions, however the neighbor finding algorithms are quite specific to 3-dimensions, and a generalization to higher dimensions is a definite challenge. In this paper, we present such an algorithm that works in arbitrary dimensions.

Our interest is in higher dimensions, motivated by another computer graphics application which aims to accelerate ray-tracing [15] based on multi-dimensional interpolation. In this application, rays in 3-space are modeled as points in a 4-dimensional parameter space, and each sample ray is traced through a scene to gather various geometric attributes that are required to compute an intensity value. Because tracing a ray through a complex scene can be computationally intensive, rather than tracing each viewing ray to compute the required attributes, we collect and store a relatively sparse set of sampled rays in a fast data structure and associate a number of continuous geometric attributes with each sample. We can then interpolate among these samples to reconstruct the value at intermediate rays [5, 6]. Because of variations in the field values, it is necessary to sample

adaptively, with denser sampling in regions of high variation and sparser sampling in regions of low variation. An adaptively refined regular simplicial mesh is constructed over the 4-dimensional domain of interest, and the field values are sampled at the vertices of this subdivision. Given a query point, we determine which cell of the subdivision contains this point, and the interpolated value is an appropriate linear or multi-linear combination of the field values at the vertices of is cell.

For interpolation purposes, compatibly refined simplicial meshes are preferable over octree-based subdivisions, since they guarantee C^0 continuous interpolants, and that they are much simpler in the sense that the interpolations are performed with a minimal number of samples. It is possible to further subdivide an octree cell to produce a simplicial complex [8, 27], but this approach does not scale well with dimension due to the exponential increase in the number of vertices and explosion of cases that need to be considered.

To illustrate the advantage of interpolation using simplicial complexes, consider the images generated from our ray-tracing application in Fig. 2. Images (a) and (c) show the result of an interpolation based on kd-trees [6], which is not a cell complex, and images (b) and (d) show the results of using the hierarchical simplicial decomposition described in this paper. (These results will be reported in a separate paper.)



(a) (b) (c) (d)

Fig. 2: Results of a ray-tracing application to produce an 800×800 image based on 4-dimensional interpolations using (a) a kd-tree based on 14,492 samples (96 CPU seconds) and (b) a simplex decomposition tree based on 6,072 samples (97 CPU seconds). Details of these images are shown in (c) and (d), respectively. Note the blocky artifacts in the kd-tree approach (c).

In addition to our own motivation, higher dimensional meshes are of interest for visualization of time-varying fields, and efficient algorithms for performing traversals and neighbor finding is required.

Thus, our main objective in this paper is to present an efficient implementation of multidimensional hierarchical regular simplicial meshes in any dimension d. Rather than representing the hierarchy explicitly as a tree using child pointers, we use a *pointerless representation* in which we access nodes through an index called a *location code*. Location codes [29, 19] have arisen as a popular alternative to standard pointer-based representations, because they separate the hierarchy from its representation, and so allow the application of very efficient access methods. The space

savings realized by not having to store pointers (to the parent, two children, and $d+1$ neighboring simplices) and simplex vertices is quite significant for large multidimensional meshes.

We present a location code, called the *LPT code*, which can be used to access nodes of this tree. Unlike Hebert's approach, which only works in 3-dimensional space and relies on enumerations and look-up tables, our approach is fully algorithmic and works in any dimension. We store the mesh in a data structure called a *simplex decomposition tree*. Our hierarchical decomposition is based on the same longest-edge bisection method given by Maubach [22]. (Note that Maubach's representation is not pointerless.) In addition to efficient computation of neighbors, we show how to perform tree traversals, point locations, and answer interpolation queries efficiently through the use of these codes.

2 Pointerless Representations and Prior Work

Regular subdivisions have the disadvantage of limiting the mesh's ability to adapt to the variational structure of the scalar field, but they provide a number of significant advantages from the perspectives of efficiency, practicality, and ease of use. The number of distinct element shapes is bounded (in our case by d), and hence it is easy to derive bounds on the geometric properties of the cells, such as aspect ratios and angle bounds. The regular structure relieves us from having to store topological information explicitly, since this information is encoded implicitly in the tree structure. Regular hierarchical decompositions can be selectively refined and coarsened, which is useful for interactive visualization. Additionally, the hierarchical structure provides a straightforward method for performing point location, which is important for answering interpolation queries.

One very practical advantage of regularity involves performance issues arising from modern memory hierarchies. It is well know that modern memory systems are based on multiple levels, ranging from registers and caches to main memory and disk (including virtual memory). The storage capacity at each level increases, and so too does access latency. There are often many orders of magnitude of difference between the time needed to access local data (which may be stored in registers or cache) versus global data (which may reside on disk) [10]. Large dynamic pointer-based data structures are particularly problematic from this perspective, because node storage is typically allocated and deallocated dynamically and, unless special care is taken, simple pointer-based traversals suffer from a nonlocal pattern of memory references. This is one of the principal motivating factors behind I/O efficient algorithms [1, 3] and cache sensitive and cache oblivious data structures and algorithms [10, 11].

In contrast with pointer-based implementations, regular spatial subdivisions support *pointerless* implementations. Pointerless versions of quadtree and its variants have been known for many years [13, 29]. The idea is to associate each node of the tree with a unique index, called a *location code*. Because of the regularity of the subdivision, given any point in space, it is possible to compute the location code of the node of a particular depth in the tree that contains this point. This can be done entirely in local memory, without accessing the data structure in global memory. Once the location code is known, the actual node containing the point can be accessed through a small number of accesses to global memory (e.g., by hashing).

The prior work in the area of pointerless regular simplicial meshes has principally been in 2- and 3-dimensions. Lee and Samet presented a pointerless hierarchical triangulation based on a four-way decomposition of equilateral triangles [19]. Hebert presented a location code for longest-edge hierarchical tetrahedral meshes and a set of rules to compute neighbors efficiently in 3-space [16]. Lee, et al. developed an alternative location code for this same tetrahedral mesh, and presented algorithms for efficient neighbor computation [18]. Both approaches are quite specific to 3-space, and are not readily generalizable to higher dimensions.

We introduce a new location code, which provides unique encoding of the simplices generated by Maubach's [22] bisection algorithm. This coding scheme works in arbitrary dimensions. We define the components required to develop a pointerless implementation based on our location code. All the geometry of the simplices and the operations required for the tree can be computed easily based solely on the code of a simplex. Our location code and the definitions of various operations on the simplex tree depend on the particular vertex ordering. We have adopted a different ordering than Maubach's system, which we feel leads to simpler formulas. Our vertex ordering is a generalization of the vertex ordering used in Hebert's 3-dimensional system.

The most challenging operation on the tree is neighbor computation. Maubach's system computes the neighbors of a simplex during construction of the tree recursively [23], and possibly stores pointers to neighbors for each simplex. We, on the other hand, are interested in efficiently computing any neighbor of any simplex directly from its code, without storing any neighbor links, and without having to traverse the path to and from the root in order to compute neighbors. This is significant gain both in terms of storage, and computational efficiency, since our approach is local and runs in $O(d)$ time—in fact in $O(1)$ time, if the operations are encoded in lookup tables. Neighbor computation is a valuable operation not only during construction of the hierarchy, but in general, for any application that requires moving between adjacent simplices of a decomposition.

3 Preliminaries

Throughout, we consider real d-dimensional space, $\Re^d$. We assume that the domain of interest has been scaled to lie within a unit *reference hypercube* of side length 2, centered at the origin, that is $[-1, 1]^d$. We shall denote points in $\Re^d$ using lower-case bold letters, and represent them as d-element row vectors, that is, $\mathbf{v} = (v_1, v_2, \ldots, v_d) = (v_i)_{i=1}^d$. We let $\mathbf{e}_i$ denote the ith unit vector. A d-simplex is represented as a $(d + 1) \times d$ matrix whose rows are the vertices of the simplex, numbered from 0 to d. Of particular interest is the *base simplex*, denoted $S_\emptyset$, whose ith vertex is $\sum_{j=1}^{i} \mathbf{e}_j - \sum_{j=i+1}^{d} \mathbf{e}_j$.

For example, in $\Re^3$ we have

$$S_\emptyset = \begin{bmatrix} -1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}. \tag{1}$$

Recall from basic geometry that two geometric objects are *congruent* if are equivalent up to translation and a possible reflection. Coordinate permutations and coordinate reflections both preserve congruence. Two objects are *similar* if they can be made congruent by a nonzero uniform scaling.

3.1 Permutations and Reflections

Let $\mathrm{Sym}(d)$ denote the *symmetric group* of all $d!$ permutations over $\{1, 2, \ldots, d\}$. We denote a permutation $\Pi \in \mathrm{Sym}(d)$ by a tuple of distinct integers $[\pi_1\ \pi_2\ \cdots\ \pi_d]$, where $\pi_i \in \{1, 2, \ldots, d\}$. We can interpret such a permutation as a linear function that maps the unit vector $\mathbf{e}_i$ to the $\mathbf{e}_{\pi_i}$, or equivalently as a coordinate permutation given by a $d \times d$ matrix whose ith row is the unit vector $\mathbf{e}_{\pi_i}$. For example, for $\Pi = [2\ 3\ 1]$,

$$S_\emptyset\Pi \;=\; \begin{bmatrix} -1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \;=\; \begin{bmatrix} -1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

It is well known that the collection of simplices $\{S_\emptyset\Psi : \Psi \in \mathrm{Sym}(d)\}$ fully subdivides the reference hypercube, and further that this subdivision is compatible (is a simplicial complex) [2]. These $d!$ simplices form the starting point of our hierarchical simplicial mesh. The *composition* of two permutations $\Pi \circ \Psi$, defined as $S(\Pi \circ \Psi) = (S\Psi)\Pi$ is given by the matrix product $\Psi\Pi$. Note that the notation $[2\ 3\ 1]$ is not a vector in $\Re^d$, but merely a convenient shorthand for a permutation matrix. Throughout, vectors will be denoted with parentheses, and square brackets will be used for objects that are to be interpreted as linear transformations, or equivalently a shorthand for a matrix. Another useful class of transformations are coordinate reflections, which can be expressed as a d-tuple $R = [r_1\ r_2\ \cdots\ r_d]$ where $r_i \in \{\pm 1\}$, and is interpreted as a linear transformation represented by the diagonal matrix $\mathit{diag}(r_1, r_2, \ldots, r_d)$.

It will simplify notation to combine the composition of a permutation and a reflection using a unified notation. We define a *signed permutation* to be a d-tuple of integers $[r_i\pi_i]_{i=1}^d$, where $[\pi_i]_{i=1}^d$ is a permutation and $[r_i]_{i=1}^d$ is a reflection. This is interpreted as a linear transformation that maps the ith unit vector to $r_i\mathbf{e}_{\pi_i}$. For example, in $\Re^3$, the composition of the reflection $R = [-1\ -1\ +1]$ and the permutation $\Pi = [2\ 3\ 1]$ is expressed as the signed permutation $[-2\ -3\ +1]$, which is just a shorthand for the matrix product $R\Pi$, that is

$$R\Pi \;=\; \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & +1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \;=\; \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ +1 & 0 & 0 \end{bmatrix}.$$

An intuitive way to interpret the meaning of a signed permutation is as an operation involving a selective negation followed by a subsequent permutation of some of the components of a row vector or the columns of a matrix. For example the signed permutation $[-2\ -3\ +1]$ can be interpreted as negating the first and second components of a vector, and then mapping the first, second, and

third components of the resulting vector to positions 2, 3, and 1, respectively. Thus, the image of (v_1, v_2, v_3) under this transformation is $(v_3, -v_1, -v_2)$.

We define the following functions that act on a signed permutation $\Pi = [\pi_i]_{i=1}^d$. The first, *perm*(Π), extracts the permutation part of Π, the second, *refl*(Π), extracts the (unpermuted) reflection part as a vector in $\{\pm 1\}^d$, and the third, *orth*(Π), returns the permutation of *refl*(Π) under Π. More formally,

$$\textit{perm}(\Pi) = [\,|\pi_i|\,]_{i=1}^d \qquad \textit{refl}(\Pi) = (\textit{sign}(\pi_i))_{i=1}^d \qquad \textit{orth}(\Pi) = \textit{refl}(\Pi)\textit{perm}(\Pi) = (\textit{sign}(\pi_i^{-1}))_{i=1}^d.$$

For example, if $\Pi = [-2\ -3\ +1]$ then *perm*$(\Pi) = [2\ 3\ 1]$, *refl*$(\Pi) = (-1, -1, +1)$, and *orth*$(\Pi) = (+1, -1, -1)$. Note that *refl*(Π) and *orth*(Π) are vectors. The associated transformation matrices are *diag*(*refl*(Π)) and *diag*(*orth*(Π)), respectively. The following is an easy consequence.

Lemma 3.1 *Let Π be a signed permutation. Then $\Pi =$ diag(refl(Π))perm(Π) = perm(Π)diag(orth(Π)).*

3.2 The Simplex Decomposition Tree

Recall that the initial simplicial complex is formed from the $d!$ permutations of the base simplex, that is, $S_\emptyset\Psi$ for $\Psi \in \mathrm{Sym}(d)$. Simplices are then refined a process of repeated subdivision, called *bisection*, in which a simplex is bisected by splitting its longest edge [22]. (Details will be given below.) The resulting *child* simplices are labeled 0 and 1. By applying the process repeatedly, each simplex in this hierarchy is uniquely identified by its *path*, which is a string over $\{0, 1\}$. The resulting collection of trees is called the *simplex decomposition tree*. It consists of $d!$ separate binary trees, which conceptually are joined under a common super-root. Each simplex of this tree is uniquely identified by a *permutation-path pair* as $S_{\Psi,p}$, where Ψ is the initial permutation of the base simplex, and $p \in \{0, 1\}^*$ is the path string. When starting with the base simplex (Ψ is the identity permutation) we may omit explicit reference to Ψ. By symmetry, it suffices to describe the bisection process on just the base simplex $S_\emptyset$. The ordering of the rows, that is, the numbering of vertices, will be significant.

Maubach [22] showed that with every d consecutive bisections, the resulting simplices are similar copies of their d-fold grandparent, subject to a uniform scaling by $1/2$. Thus, the pattern of decomposition repeats every d levels in the decomposition. Define the *level*, ℓ, of a simplex S_p to be the path length modulo the dimension, that is, $\ell = (|p| \bmod d)$, where $|p|$ denotes the length of p. The 0-child S_{p0} and 1-child S_{p1} of a simplex are computed as follows:

$$S_p = \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_{\ell-1} \\ \mathbf{v}_\ell \\ \mathbf{v}_{\ell+1} \\ \vdots \\ \mathbf{v}_d \end{bmatrix} \qquad S_{p0} = \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_{\ell-1} \\ (\mathbf{v}_\ell + \mathbf{v}_d)/2 \\ \mathbf{v}_{\ell+1} \\ \vdots \\ \mathbf{v}_d \end{bmatrix} \qquad S_{p1} = \begin{bmatrix} \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_{\ell-1} \\ (\mathbf{v}_\ell + \mathbf{v}_d)/2 \\ \mathbf{v}_\ell \\ \vdots \\ \mathbf{v}_{d-1} \end{bmatrix}.$$

A portion of the tree is illustrated in Fig. 3. Note that in both cases the first ℓ vertices are unchanged. The new ℓth vertex is the midpoint of the edge between the ℓth and last vertices. The remaining $d - \ell$ vertices are a subsequence of the original vertices, shifted by one position relative to each other.

Equivalently, we can define $S_{p0} = B_{\ell,0} S_p$ and $S_{p1} = B_{\ell,1} S_p$, where $B_{\ell,0}$ and $B_{\ell,1}$ are $(d+1) \times (d+1)$ matrices whose ℓth row (starting from row 0) has the value $1/2$ in columns ℓ and d (starting from column 0), and all other rows are unit vectors. For example, in dimension $d = 4$ and for $\ell = 2$ we have

$$B_{\ell,0} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad B_{\ell,1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Our bisection scheme is geometrically equivalent to the one defined by Maubach [22], but we order the vertices differently from Maubach and reverse the names of the 0- and 1-children. Although the differences are theoretically insignificant, our ordering results in somewhat simpler and more regular formulas for computing descendents and neighbors.

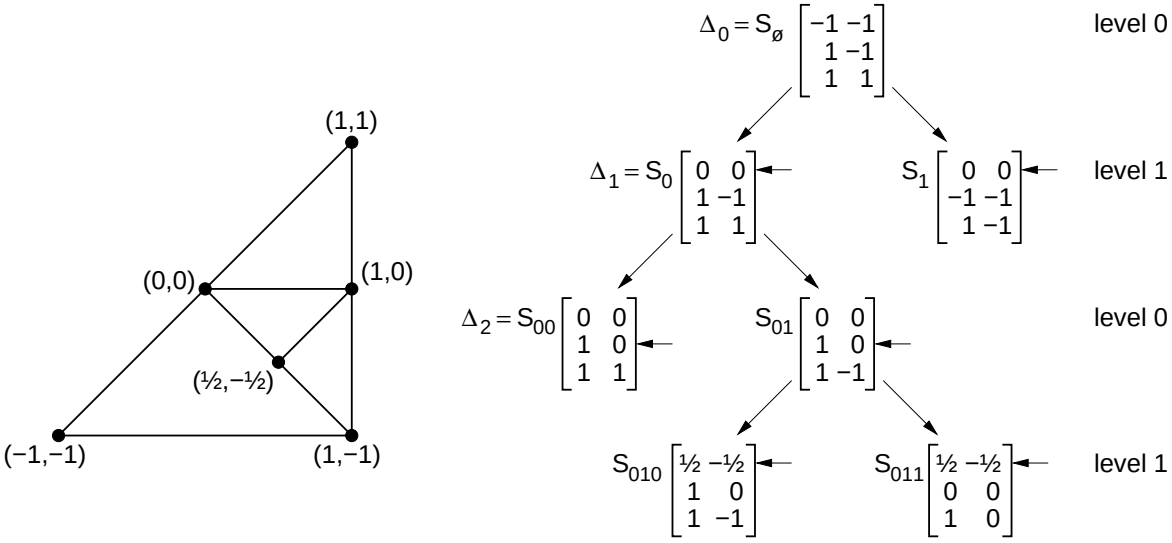


Fig. 3: The simplex decomposition tree. The corresponding bisected simplex is shown on the left. The newly created vertex is indicated by an arrow in each case. The reference simplices Δ_i are indicated as well.

3.3 Reference Simplices and the Reference Tree

Since with every d consecutive bisections, the simplices are similar to, but half the size, of their d-fold grandparent, we can partition the nodes of the decomposition tree into a collection of isomorphic, disjoint subtrees of height d. The roots of these subtrees are the nodes whose depths are multiples of d (where the root starts at depth 0). It suffices to analyze the structure of just one of these trees, in particular, the subtree of height d starting at the root. We call this the *reference tree*. Since the two children of any simplex are congruent, it follows that all the simplices at any given depth of the decomposition tree are congruent to each other. Thus, all the similarity classes are represented by d canonical simplices, called the *reference simplices*. These are defined to be $S_{(0^k)}$, for $0 \leq k < d$, and denoted by Δ_k. (See Fig. 3.) Although it is not a reference simplex, we also define $\Delta_d = S_{(0^d)}$, since it is useful in our proofs.

For example, in $\Re^3$ the 3 reference simplices together with Δ_3 are

$$\underset{(S_\emptyset)}{\Delta_0} = \begin{bmatrix} -1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \underset{(S_0)}{\Delta_1} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & -1 & -1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \underset{(S_{00})}{\Delta_2} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \underset{(S_{000})}{\Delta_3} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

4 The LPT code

So far we have defined an infinite decomposition tree and a procedure for generating the simplices of this tree from the top down. In order to provide pointerless implementation of the hierarchical mesh, we define a *location code*, which uniquely identifies encodes each simplex of the hierarchy. The most direct location code is combination consisting of the initial permutation Ψ followed by the binary encoding of the tree path p. Unfortunately, it is not easy to compute basic properties of the simplex such as neighbors from this code. Nonetheless, Lee, De Floriani, and Samet showed how to compute neighbors from the path code in the 3-dimensional case [18]. Instead we modify an approach presented by Hebert [16] for the 3-dimensional case, by defining a location code that more directly encodes the geometric relationship between the each simplex and the reference simplex at the same level. We call this the *LPT code*, since it encodes for each simplex its *Level*, its signed *Permutation*, and its *Translation* relative to some reference simplex. We shall show that it is possible to compute tree relations (children and parents) as well as neighbors in the simplicial complex using this code.

Given any simplex $S_{\Psi,p}$ in the hierarchy, the *LPT code* is a 3-tuple (ℓ, Π, Φ), where $\ell = |p| \bmod d$ is the simplex's level, Π is a signed permutation relating $S_{\Psi,p}$ to its reference simplex, and Φ is a list of vectors, called the orthant list, which is used to derive the translation relative to the reference simplex. The permutation part $\Pi = \Pi_{\Psi,p}$ and and orthant list $\Phi = \Phi_{\Psi,p}$ are defined below as functions of Ψ and p. Correctness will be established in Theorem 4.1 below.

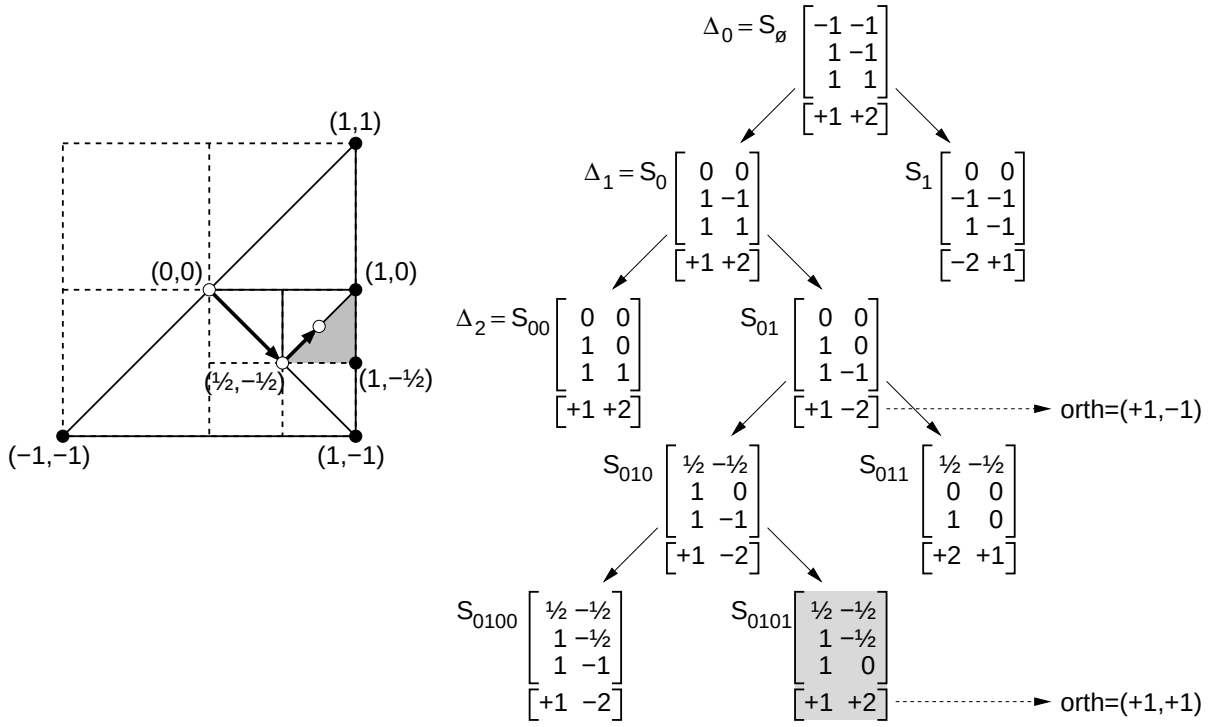


Fig. 4: The signed permutations $\Pi_{\Psi,p}$ associated with each simplex are shown below each simplex matrix, and the entries of the orthant list are shown for the shaded simplex S_{0101}. The LPT code for this simplex is $(0, [+1\ +2], \langle(+1, -1), (+1, +1)\rangle)$.

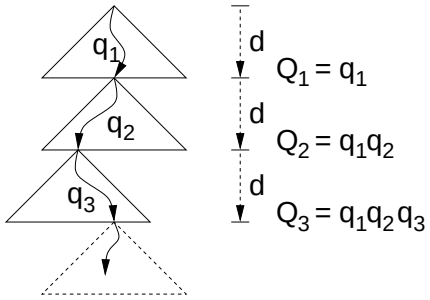
Permutation Part: The signed permutation $\Pi_{\Psi,p}$ is defined recursively as follows for a base permutation Ψ and binary path p:

$$\Pi_{\Psi,\emptyset} \;=\; \Psi \qquad\qquad \Pi_{\Psi,p0} \;=\; \Pi_{\Psi,p} \qquad\qquad \Pi_{\Psi,p1} \;=\; \Pi_{\Psi,p} \circ \Sigma_\ell, \tag{2}$$

where Σ_ℓ is the permutation that cyclically shifts the last $d-\ell$ elements to the right and negates the element that is wrapped around. That is, $\Sigma_\ell = [1\ 2\ \cdots\ \ell\ (-d)\ (\ell+1)\ (\ell+2)\ \cdots\ (d-1)]$. A portion of the simplex decomposition tree, and the associated permutation values are shown in Fig. 4. For example, observe that S_1 is related to Δ_1 by the signed permutation $[-2\ +1]$, which negates the first column of Δ_1 and then swaps the two columns.

Orthant List: Recall that with every d levels of descent in the decomposition tree, the resulting simplices decrease in size by a factor of $1/2$. The bounding hypercube of the resulting descendent is one of the 2^d hypercubes that would result from a quadtree-like decomposition (indicated by broken lines on the left side of Fig. 4). Depending on the level within the tree, the translation of the descendent hypercube relative to its ancestor will be some power of $(1/2)$ times a d-vector over $\{\pm 1\}$. Such a vector defines the *orthant* containing the descendent hypercube relative to the central vertex of its ancestor. Consider, for example, the shaded simplex in Fig. 4. Its translation relative to the base simplex is $\frac{1}{2}(+1,-1) + \frac{1}{4}(+1,+1)$, indicated by the arrowed lines on the left side of the figure. The orthant list encodes these two vectors.

To define the orthant list, we first remove the last ℓ symbols of p, leaving a multiple of d symbols (possibly empty). We then partition the remaining symbols into $L = \lfloor |p|/d \rfloor$ substrings, $q_1 q_2 \ldots q_L$, where $|q_i| = d$. Since the reference tree structure repeats every d levels, each q_i can be viewed as a complete path in one of these subtrees of height d. Let Q_i denote the concatenation of the first i substrings. For $1 \le i \le L$, define $\Gamma_{\Psi,p}[i]$ to be the signed permutation for path Q_i, that is Π_{Ψ,Q_i}. Define the *orthant list* for the pair (Ψ, p) to be the sequence of L vectors whose ith element is *orth*$(\Gamma_{\Psi,p}[i])$, that is



$$\Phi_{\Psi,p} = \langle \textit{orth}(\Gamma_{\Psi,p}[1]), \textit{orth}(\Gamma_{\Psi,p}[2]), \ldots, \textit{orth}(\Gamma_{\Psi,p}[L]) \rangle\,. \tag{3}$$

The orthant list can be computed incrementally along with the permutation part of the code as follows. Given the LPT code (ℓ, Π, Φ) for a simplex $S_{\Psi,p}$, first observe that the orthant list only changes for the children if the current level is $d-1$. If so, we compute the child's permutation Π' from Eq. (2) and append *orth*(Π') to the current list. Observe that given the level ℓ and the orthant list Φ for any simplex, we can derive the length of the associated tree path p as $\ell + d \cdot \textit{length}(\Phi)$.

The computation of the LPT code is summarized in the procedure *LPTcode*. The code for the simplex $S_{\Psi,p}$ is computed by the call *LPTcode*$(p, (0, \Psi, \emptyset))$. We may now state the main result of this section, called the *LPT Theorem*, which establishes the geometric meaning of our LPT code by relating each simplex of the decomposition tree to its associated reference simplex. Hebert [16]

LPTcode$(p, (\ell, \Pi, \Phi))$
if $(p = \emptyset)$ **return** (ℓ, Π, Φ)
Express p as xq, for $x \in \{0, 1\}$
$\ell \leftarrow (\ell + 1) \bmod d$
if $(x = 1)$ $\Pi \leftarrow \Pi \circ \Sigma_\ell$
if $(\ell = 0)$ $\Phi \leftarrow \Phi +$ *orth*(Π)
return *LPTcode*$(q, (\ell, \Pi, \Phi))$

proved the analogous result for his 3-dimensional bisection system. Let $\mathbf{1}_{d+1}^T$ denote a $(d+1)$-column vector of 1's. The following theorem makes use of the observation that, for any d-row vector **v**, the matrix product $\mathbf{1}_{d+1}^T \cdot \mathbf{v}$ is a $(d+1) \times d$ vector whose rows are all equal to **v**, and hence adding this to any simplex matrix is equivalent to a translation by **v**.

Theorem 4.1 (LPT Theorem) *Let $S_{\Psi,p}$ be the simplex of the decomposition tree associated with some initial permutation Ψ and binary path p. Let (ℓ, Π, Φ) be the LPT code for this simplex, defined above. Then $S_{\Psi,p}$ is related to Δ_ℓ, the reference simplex at this level, by the following similarity transformation:*

$$S_{\Psi,p} = \frac{1}{2^L}\Delta_\ell \Pi + \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i}\Phi[i].$$

where $L = \lfloor\, |p|/d \rfloor$.

Before proving this theorem, we will prove the following technical lemmas.

Lemma 4.1 *Given the reference simplex Δ_ℓ, $0 \le \ell < d$,*

$$B_{\ell,0}\Delta_\ell = B_{\ell,1}\Delta_\ell \Sigma_\ell^{-1},$$

where Σ_ℓ is as defined in Section 4.

Proof: $\Sigma_\ell^{-1} = \Sigma_\ell^T = [e_1^T \ldots e_\ell^T \quad -e_d^T \quad e_{\ell+1}^T \ldots e_{d-1}^T]$, since Σ_ℓ is an orthogonal matrix. When a matrix is postmultiplied by Σ_ℓ^{-1}, the last column is negated, and then the last $(d-\ell)$ columns are cyclically shifted to the right. Consider the general form of a reference simplex and its two children as shown in Figure 5. It can be observed that, if we negate the last column of the 1-child of Δ_ℓ, and cyclically shift the last $(d-\ell)$ columns to the right, we get the 0-child of Δ_ℓ. □

Lemma 4.2 *Given the reference simplex Δ_ℓ, $0 \le \ell < d$, and a signed permutation $\Pi_{\Psi,p}$,*

$$B_{\ell,1}\Delta_\ell \Pi_{\Psi,p} = \Delta_{\ell+1}\Pi_{\Psi,p1}$$

Proof: By definition, $\Pi_{\Psi,p1} = \Sigma_\ell \Pi_{\Psi,p}$, that is $\Pi_{\Psi,p} = \Sigma_\ell^{-1}\Pi_{\Psi,p1}$.

$$B_{\ell,1}\Delta_\ell \Pi_{\Psi,p} = \Delta_{\ell+1}\Sigma_\ell \Pi_{\Psi,p} = \Delta_{\ell+1}\Sigma_\ell\Sigma_\ell^{-1}\Pi_{\Psi,p1} = \Delta_{\ell+1}\Pi_{\Psi,p1}$$

□

Proof: (Theorem 4.1) We will prove Theorem Theorem 4.1 by induction. Recall that $\Pi = \Pi_{\Psi,p}$, $\Phi = \Phi_{\Psi,p}$, $l = |p| \bmod d$ and $L = \lfloor\, |p|/d \rfloor$.

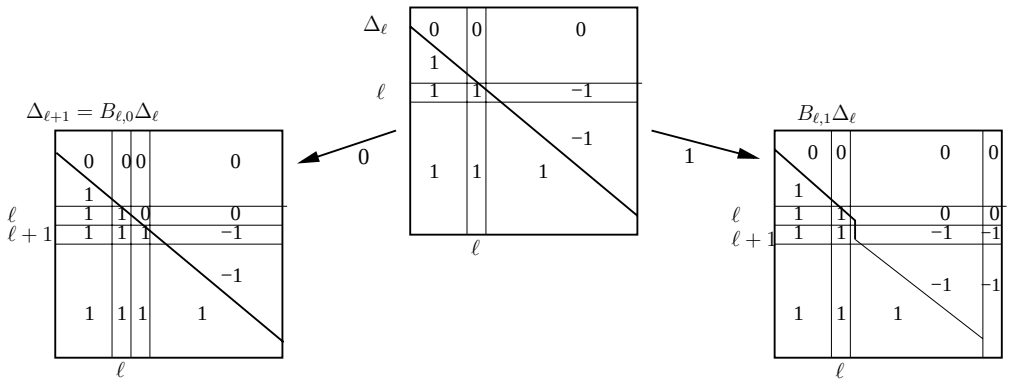


Fig. 5: The two children of a reference simplex.

Induction Basis: The hypothesis holds for all root simplices, $S_{\Psi,\emptyset}$. Since $L = \lfloor |p|/d \rfloor = 0$, and $\ell = 0$ at root level,

$$\begin{aligned} S_{\Psi,\emptyset} &= \tfrac{1}{2^0}\Delta_0\Pi_{\Psi,\emptyset} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{0} \frac{1}{2^i}\Phi_{\Psi,\emptyset}[i] \\ &= \Delta_0\Pi_{\Psi,\emptyset} \quad \text{(holds by definition)}. \end{aligned}$$

Induction Step: Assume that the inductive hypothesis holds for $S_{\Psi,p}$, at *level* $\ell = |p| \bmod d$. We will show that, it holds for the 0- and 1-children of $S_{\Psi,p}$. In addition to the above lemmas, we will make use of the following equalities:

$$\Delta_{\ell+1} = B_{\ell,0}\Delta_\ell.$$

Let $T = \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p}[i]$. Note that, $T = \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p0}[i] = \mathbf{1}_{d+1}^T \sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p1}[i]$ as well.

Also, note that all rows are T are equal to each other. For such a matrix T, the following equalities hold,

$$B_{\ell,0}T = T, \quad B_{\ell,1}T = T.$$

In the induction, there are two cases to be distinguished depending on ℓ:

1. $0 \le \ell < d-1$

   - First, consider $S_{\Psi,p0}$. By definition, $\Pi_{\Psi,p0} = \Pi_{\Psi,p}$.

$$\begin{aligned} S_{\Psi,p0} &= B_{\ell,0}S_{\Psi,p} = B_{\ell,0}(\tfrac{1}{2^L}\Delta_\ell\Pi_{\Psi,p} + T) && \text{(by induction hyp.)} \\ &= \tfrac{1}{2^L}\Delta_{\ell+1}\Pi_{\Psi,p} + T = \tfrac{1}{2^L}\Delta_{\ell+1}\Pi_{\Psi,p0} + T. \end{aligned}$$

   This completes the induction for $S_{\Psi,p0}$, since $\lfloor |p0|/d \rfloor = L$ for $0 \le \ell < d-1$.

- Now, consider $S_{\Psi,p1}$.

$$\begin{aligned} S_{\Psi,p1} &= B_{\ell,1}S_{\Psi,p} = B_{\ell,1}(\tfrac{1}{2^L}\Delta_\ell\Pi_{\Psi,p} + T) && \text{(by induction hyp.)} \\ &= \tfrac{1}{2^L}\Delta_{\ell+1}\Pi_{\Psi,p1} + T. && \text{(by Lemma 4.2)} \end{aligned}$$

This completes the induction for $S_{\Psi,p1}$, since $\lfloor |p1|/d \rfloor = L$ for $0 \leq \ell < d-1$.

2. $\ell = d-1$, the children of $S_{\Psi,p}$ will be at *level* 0.

- First, consider $S_{\Psi,p0}$. By definition, $\Pi_{\Psi,p0} = \Pi_{\Psi,p}$.

$$\begin{aligned} S_{\Psi,p0} &= B_{d-1,0}S_{\Psi,p} \\ &= B_{d-1,0}(\tfrac{1}{2^L}\Delta_{d-1}\Pi_{\Psi,p} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p}[i]) && \text{(by induction hyp.)} \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p}[i] \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p0}[i] \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p0}[i] - \frac{1}{2^{L+1}}\mathbf{1}_{d+1}^T\Phi_{\Psi,p0}[L+1] \end{aligned}$$

Since $\Delta_d = \frac{\Delta_0 + [\mathbf{1}]_{(d+1)\times d}}{2}$ where $[\mathbf{1}]_{(d+1)\times d}$ is a matrix of 1's and $\Phi_{\Psi,p0}[L+1] = \mathit{orth}(\Pi_{\Psi,p0})$,

$$\begin{aligned} S_{\Psi,p0} &= \tfrac{1}{2^L}\tfrac{(\Delta_0 + [\mathbf{1}]_{(d+1)\times d})}{2}\Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p0}[i] - \frac{1}{2^{L+1}}\mathbf{1}_{d+1}^T\mathit{orth}(\Pi_{\Psi,p0}) \\ &= \tfrac{1}{2^{L+1}}\Delta_0\Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p0}[i] \\ &\quad + \tfrac{1}{2^{L+1}}([\mathbf{1}]_{(d+1)\times d}\Pi_{\Psi,p0} - \mathbf{1}_{d+1}^T\mathit{orth}(\Pi_{\Psi,p0})) \end{aligned}$$

By Lemma 3.1,

$$\begin{aligned} \mathbf{1}_{d+1}^T\mathit{orth}(\Pi_{\Psi,p0}) &= \mathbf{1}_{d+1}^T\mathit{refl}(\Pi_{\Psi,p0})\mathit{perm}(\Pi_{\Psi,p0}) \\ &= [\mathbf{1}]_{(d+1)\times d}\mathit{diag}(\mathit{refl}(\Pi_{\Psi,p0}))\mathit{perm}(\Pi_{\Psi,p0}) \\ &= [\mathbf{1}]_{(d+1)\times d}\Pi_{\Psi,p0}. \end{aligned}$$

And so, we see that, the third term above is 0, yielding

$$S_{\Psi,p0} = \tfrac{1}{2^{L+1}}\Delta_0\Pi_{\Psi,p0} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p0}[i].$$

This completes the induction for $S_{\Psi,p0}$, since $\lfloor |p0|/d \rfloor = L+1$ for $\ell = d-1$.

- Next, consider $S_{\Psi,p1}$.

$$\begin{aligned} S_{\Psi,p1} &= B_{\ell,1} S_{\Psi,p} = B_{\ell,1}(\tfrac{1}{2^L}\Delta_{d-1}\Pi_{\Psi,p} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p}[i]) \quad \text{(by induc. hyp.)} \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p1} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L} \frac{1}{2^i}\Phi_{\Psi,p1}[i] \qquad \text{(by Lemma 4.2)} \\ &= \tfrac{1}{2^L}\Delta_d\Pi_{\Psi,p1} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p1}[i] - \frac{1}{2^{L+1}}\mathbf{1}_{d+1}^T\Phi_{\Psi,p1}[L+1] \end{aligned}$$

  Applying the same derivations as in the previous case, this can be reduced to

$$S_{\Psi,p1} = \tfrac{1}{2^{L+1}}\Delta_0\Pi_{\Psi,p1} + \mathbf{1}_{d+1}^T \textstyle\sum_{i=1}^{L+1} \frac{1}{2^i}\Phi_{\Psi,p1}[i].$$

  This completes the induction for $S_{\Psi,p1}$, since $\lfloor |p1|/d \rfloor = L+1$ for $\ell = d-1$.

□

Implementation Issues: We can now describe a pointerless implementation of a simplex decomposition tree. For each simplex $S_{\Psi,p}$ in the tree, we create an node that is indexed by an appropriate encoding of the associated LPT code. Theorem 4.1 implies that the geometry of this simplex is determined entirely from the LPT code, and, if desired, it can be computed from the code in time proportional to the code length. In addition to the index, this node may also contain application-specific data. These objects are then stored in any index structure that supports rapid look-ups, for example, a hash table.

There are a number of practical observations that can be made in how to encode LPT codes efficiently in low dimensional spaces. Let D denotes the maximum depth of any node in the tree. Each of the $d!$ permutations of $\mathrm{Sym}(d)$ can be encoded as a integer with $\log_2 d!$ bits [17]. A d-element reflection vector over $\{\pm 1\}$ can be represented as a d-element bit string (e.g., by the mapping $+1 \to 0$ and $-1 \to 1$). Thus, a signed permutation Π then can be encoded by a pair of integers. A convenient way to encode the vectors of the orthant list is map them to bit strings and to store them as d separate lists, one for each coordinate. (The advantage of this representation will be discussed in Section 6.) The final code consists of the level ℓ, expressed with $\lceil \log_2 d \rceil$ bits, the permutation and reflection, represented using $\lceil \log_2(d!) \rceil + d$ bits, and finally the orthant list, represented using $d \cdot \mathit{length}(\Phi)$ bits, which is at most $d\lfloor D/d \rfloor \leq D$. The total number of bits needed to represent the code for a simplex at depth D is $D + \log_2(d!) + O(d)$. This is close to optimal in the worst case, since there are $2^D d!$ simplices at depth D in a full tree. If we assume that the machine's word size is $\Omega((D/d) + \log_2 d!)$, then the permutation part of the code can be stored in a constant number of machine words and the orthant lists can be stored in $O(d)$ machine words.

Also, note that for small d, the multiplication tables for the various signed permutations (such as Σ_ℓ of Eq. (2) and the neighbor permutations of Section 6 below) can be precomputed and stored in tables. This allows very fast evaluation of permutation operations by simple table look-up.

5 Decomposition Tree Operations

In this section we present methods for performing useful tree access operations based on manipulations of LPT codes, including tree traversal, point location and interpolation queries, and computing neighbors in the simplicial complex.

5.1 Tree Traversal

Consider a simplex $S_{\Psi,p}$ of the tree whose LPT code is (ℓ, Π, Φ). Let us consider how to compute the children and parent of this simplex in the tree. The LPT codes of the children of this simplex can be computed in $O(d)$ time by applying the recursive rules used to define the LPT code, given in Section 4. We can compute the parent from the LPT code by inverting this process, but in order to do so we need to know whether the simplex is a left child, a right child, or the root. A root simplex is distinguished by having an empty orthant list and level $\ell = 0$. Otherwise, we make use of the following lemma.

Lemma 5.1 *Consider a nonroot simplex S of the decomposition tree with LPT code (ℓ, Π, Φ), and let S' be its nearest proper ancestor at level 0. Let $\Pi = [\pi_i]_{i=1}^d$ be the signed permutation of S, let $\mathbf{o} = (o_i)_{i=1}^d$ be the last entry of the orthant list of S', and let $\ell^* = 1 + ((\ell - 1) \bmod d)$. Then S is a 0-child if and only if sign$(\pi_{\ell^*}) =$ sign$(o_{|\pi_{\ell^*}|})$.*

Proof: In order to prove Lemma 5.1, we prove the following more general lemma, which characterizes the child relations for a simplex's ancestors, up to the next 0th level.

Lemma 5.2 *Let $S_{\Psi,p}$ be a nonroot simplex, and let $S_{\Psi,t}$ be its nearest proper ancestor of level 0. Let $\mathbf{o} =$ orth$(\Pi_{\Psi,t})$ be the last orthant list entry of $S_{\Psi,t}$. Let $b_1 b_2 \ldots b_{\ell^*}$ denote the path from $S_{\Psi,t}$ to $S_{\Psi,p}$, where $\ell^* = 1 + ((\ell - 1) \bmod d)$. Let $\Pi_{\Psi,p} = [\pi_i]_{i=1}^d$ and $\mathbf{o} = (o_i)_{i=1}^d$. Then*

$$b_i = \begin{cases} 0 & \textit{if} \quad \textit{sign}(\pi_i) = \textit{sign}(o_{|\pi_i|}) \\ 1 & \textit{if} \quad \textit{otherwise.} \end{cases},$$

Proof: We do not know what $\Pi_{\Psi,t}$ is, but since we know **o**, we know the signs of each coordinate axis in $\Pi_{\Psi,t}$. We can determine $b_1 b_2 \ldots b_{\ell^*}$ by finding out which axes changed signs as we go down the tree from $S_{\Psi,t}$ to $S_{\Psi,p}$.

Consider the step, when we descend down from $S_{\Psi,tb_1\ldots b_{i-1}}$ to $S_{\Psi,tb_1\ldots b_i}$. $\Pi_{\Psi,tb_1\ldots b_{i-1}}$ and $\Pi_{\Psi,tb_1\ldots b_i}$ denote the associated permutations.

If $b_i = 0$, we follow the 0-path, and $\Pi_{\Psi,tb_1\ldots b_i}$ will be identical to $\Pi_{\Psi,tb_1\ldots b_{i-1}}$. Thus, the i^{th} entry in $\Pi_{\Psi,tb_1\ldots b_i}$ remains with its original sign. On the other hand, if $b_i = 1$, we follow the 1-path, and so the d^{th} entry in $\Pi_{\Psi,tb_1\ldots b_{i-1}}$ is negated and cyclically shifted to the i^{th} position in $\Pi_{\Psi,tb_1\ldots b_i}$. Thus, the i^{th} entry in $\Pi_{\Psi,tb_1\ldots b_i}$ has changed its original sign. Since the subsequent steps apply cyclical shifts only to the last $(d - i)$ entries of the permutation, the i^{th} location remains the same until we

descend down to $S_{\Psi,p}$. And so, looking at whether the i^{th} entry in $\Pi_{\Psi,p}$ has changed its sign or not, we can determine b_i.

$$\overbrace{[+1-4-3+2]}^{S_{\psi,t}} \xrightarrow{1} [-2+1-4-3] \xrightarrow{0} [-2+1-4-3] \xrightarrow{1} \overbrace{[\underbrace{-2}_{b_1=1}\ \underbrace{+1}_{b_2=0}\ \underbrace{+3}_{b_3=1}\ -4]}^{S_{\psi,p}}$$

$$(o_i)_1^d = (+1,+1,-1,-1), \quad \ell = 3 \qquad\qquad b_1b_2b_3 = 101$$

Consider the above example. o_2 had a positive sign, following the 1-path, it was negated, and became the first element in $\Pi_{\Psi,p}$, because after it was shifted to the first location, it was fixed. Similarly, following the 0-path, 1 remained positive and got fixed at the second location, and following 1-path 3 was negated and placed at the third location. And so, the path from $S_{\Psi,t}$ to $S_{\Psi,p}$ is 101. □

Now Lemma 5.1 follows as an immediate corollary since $S_{\Psi,p}$ is a 0-child, if and only if $b_{\ell^*} = 0$. □

Lemma 5.1 can be applied as follows to determine the LPT code for the parent of a nonroot simplex S. Given S's LPT code, (ℓ, Π, Φ), we distinguish two cases, depending on its level. If ℓ is nonzero, then its parent's level is $\ell' = \ell - 1$ and otherwise its parent's level is $\ell' = d - 1$. If ℓ is nonzero, then the orthant vector **o** of the lemma is the last entry of Φ. We apply this lemma to determine whether S is a 0- or 1-child. From Eq. (2) and Theorem 4.1 we know that, if it is a 0-child, it has the same permutation code as its parent, and otherwise its parent's permutation code is $\Pi \circ \Sigma_{\ell'}^{-1}$. Its parent has the same orthant list. On the other hand, if $\ell = 0$ then **o** is the second to last entry of Φ. Again we apply the lemma to determine whether S is a 0- or 1-child, and derive its parent's permutation code. The last entry of S's orthant list is removed to form the orthant list of its parent. This can be computed in $O(d)$ time. The computation of the parent is summarized in the procedure *parent*. The parent of the simplex $S_{\Psi,p}$ with LPT code (ℓ, Π, Φ) is computed by the call *parent*$(p, (\ell, \Pi, \Phi))$.

parent$(p, (\ell, \Pi, \Phi))$
  if $(p = \emptyset)$ **return** $\emptyset$
  Express p as $tb_1b_2 \ldots b_{\ell^*}$.
  $\ell' \leftarrow (\ell - 1) \bmod d$
  if $(\ell = 0)$ $\Phi' \leftarrow \Phi - \langle \Phi[L] \rangle$
  else $\Phi' \leftarrow \Phi$
  if $(b_{\ell^*} = 0)$ $\Pi' \leftarrow \Pi$
  else $\Pi' \leftarrow \Pi \circ \Sigma_{\ell'}^{-1}$
  return (ℓ', Π', Φ')

5.2 Point Location and Interpolation Queries

In this section we consider how to compute the LPT code of the leaf simplex of the decomposition tree that contains a given query point $\mathbf{q} = (q_i)_{i=1}^d$. We assume that **q** lies in the base hypercube, that is, $-1 \leq q_i \leq 1$. If **q** lies on a face between two simplices, we will choose one arbitrarily.

We begin by locating the root simplex, $S_{\Psi,\emptyset}$ that contains **q**. It is easy to see that a point **q** in the base hypercube lies in the base reference simplex, Δ_0, if and only if its coordinate vector is sorted in decreasing order. It follows that determining the permutation Ψ of the root simplex reduces to sorting the coordinates of **q** in decreasing order and setting Ψ to the permutation that

produces this sorted order. Let us assume that we have a function *sortDescending* that computes this permutation.

Letting $\mathbf{v}_i$ denote the ith vertex of the root simplex $S_{\Psi,\emptyset}$ that contains $\mathbf{q}$, the *barycentric coordinates* of $\mathbf{q}$ with respect to this simplex is the unique $d+1$ vector $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$, $0 \le \alpha_i \le 1$, such that $\sum_i \alpha_i = 1$ and $\mathbf{q} = \sum_i \alpha_i \mathbf{v}_i$. Because of the special structure of Δ_0, it is easy to verify that the procedure *findRoot* shown in Fig. 6 computes these coordinates.

After this initialization, we recursively descend the hierarchy until finding a leaf simplex. We use the barycentric coordinates of $\mathbf{q}$ relative to the current simplex to determine in which child it resides. Then we generate the barycentric coordinates of $\mathbf{q}$ with respect to this child. This is done with the aid of the following lemma, which is proved in the appendix. The descent algorithm is presented in Fig. 6 and its correctness follows from Lemma 5.3. To simplify the presentation, we have omitted the orthant list processing, but it is essentially the same as in the code block just prior to Theorem 4.1.

```
findRoot(q)
    Ψ ← sortDescending((q)_{i=1}^d)
    α_0 ← (1 − q_{ψ_1})/2
    α_d ← (1 + q_{ψ_d})/2
    for (0 < i < d)   α_i ← (q_{ψ_i} − q_{ψ_{i+1}})/2
    return(Ψ, α)
```

```
search(q, (ℓ, Π), α)
    if ((ℓ, Π) is a leaf) return (ℓ, Π)
    α′ ← α
    if (α_ℓ ≤ α_d)
        α′_ℓ ← 2α_ℓ;   α′_d ← α_d − α_ℓ
        return search(q, ((ℓ + 1) mod d, Π), α′)
    else
        α′_d ← 2α_d;   α′_ℓ ← α_ℓ − α_d
        return search(q, ((ℓ + 1) mod d, Π ∘ Σ_ℓ), α′Σ′_ℓ)
```

Fig. 6: The procedures *findRoot* and *search*, which are used to locate a query point $\mathbf{q}$ in the hierarchy. The permutation Σ'_ℓ is defined in Lemma 5.3 and the permutation Σ_ℓ was given in Section 4, Eq. 2.

Lemma 5.3 *Consider a nonleaf simplex $S_{\Psi,p}$ of the hierarchy at level ℓ with the associated permutation code $\Pi_{\Psi,p} = [\pi_i]_{i=1}^d$. Suppose that $\mathbf{q}$ lies within this simplex with the barycentric coordinates $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$.*

- *If $\alpha_\ell \le \alpha_d$, then $\mathbf{q}$ lies in the 0-child. Let $\boldsymbol{\alpha}'$ be the $(d+1)$-vector that is identical $\boldsymbol{\alpha}$ except that $\alpha'_\ell = 2\alpha_\ell$ and $\alpha'_d = \alpha_d - \alpha_\ell$. Then the barycentric coordinate vector of $\mathbf{q}$ relative to this child is $\boldsymbol{\alpha}'$.*
- *Otherwise, $\mathbf{q}$ lies in the 1-child. Let Σ'_ℓ be a $(d+1)$-permutation that shifts the last $d+1-\ell$ coordinates circularly one position to the right. Let $\boldsymbol{\alpha}'$ be the $(d+1)$-vector that is identical to $\boldsymbol{\alpha}$ except that $\alpha'_d = 2\alpha_d$ and $\alpha'_\ell = \alpha_\ell - \alpha_d$. Then the barycentric coordinate vector of $\mathbf{q}$ relative to this child is $\boldsymbol{\alpha}'\Sigma'_\ell$.*

Proof: Let $S_{\Psi,p}$ be the simplex of the hierarchy at level ℓ that contains $\mathbf{q}$. Let $\Pi_{\Psi,p} = [\pi_i]_{i=1}^d$ be the associated permutation vector. Let $\boldsymbol{\alpha} = (\alpha_i)_{i=0}^d$ denote $\mathbf{q}$'s barycentric coordinates with respect to

$S_{\Psi,p}$. Let $\mathbf{v}_i$ denote the ith vertex of $S_{\Psi,p}$. Recall that $\mathbf{m} = \frac{\mathbf{v}_\ell + \mathbf{v}_d}{2}$ is the newly created vertex that bisects this simplex and that

$$S_{\Psi,p}=[\mathbf{v}_0 \ldots \mathbf{v}_\ell \ldots \mathbf{v}_d]^T, \quad S_{\Psi,p0}=[\mathbf{v}_0 \ldots \mathbf{v}_{\ell-1}\ \mathbf{m}\ \mathbf{v}_{\ell+1} \ldots \mathbf{v}_d]^T, \quad S_{\Psi,p1}=[\mathbf{v}_0 \ldots \mathbf{v}_{\ell-1}\ \mathbf{m}\ \mathbf{v}_\ell \ldots \mathbf{v}_{d-1}]^T$$

And so, $\mathbf{v}_\ell = 2\mathbf{m} - \mathbf{v}_d$, and $\mathbf{v}_d = 2\mathbf{m} - \mathbf{v}_\ell$. Thus, $\mathbf{q}$ can be written in terms of barycentric coordinates as,

$$\begin{aligned} \mathbf{q} &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_\ell\mathbf{v}_\ell + \ldots + \alpha_d\mathbf{v}_d \\ &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_{\ell-1}\mathbf{v}_{\ell-1} + \alpha_\ell(2\mathbf{m} - \mathbf{v}_d) + \alpha_{\ell+1}\mathbf{v}_{\ell+1} + \ldots + \alpha_d\mathbf{v}_d \\ &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_{\ell-1}\mathbf{v}_{\ell-1} + 2\alpha_\ell\mathbf{m} + \alpha_{\ell+1}\mathbf{v}_{\ell+1} + \ldots + (\alpha_d - \alpha_\ell)\mathbf{v}_d \end{aligned}$$

and similarly,

$$\begin{aligned} \mathbf{q} &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_\ell\mathbf{v}_\ell + \ldots + \alpha_d\mathbf{v}_d \\ &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_{\ell-1}\mathbf{v}_{\ell-1} + \alpha_\ell\mathbf{v}_\ell + \ldots + \alpha_d(2\mathbf{m} - \mathbf{v}_\ell) \\ &= \alpha_0\mathbf{v}_0 + \ldots + \alpha_{\ell-1}\mathbf{v}_{\ell-1} + 2\alpha_d\mathbf{m} + (\alpha_\ell - \alpha_d)\mathbf{v}_\ell + \ldots + \alpha_{d-1}\mathbf{v}_{d-1} \end{aligned}$$

And so, if $(\alpha_d - \alpha_\ell) \geq 0$, it follows that $\mathbf{q}$ resides in $S_{\Psi,p0}$, and otherwise it resides in $S_{\Psi,p1}$. From the above equations, we can also see the barycentric coordinates when $\mathbf{q}$ resides in $S_{\Psi,p0}$ or in $S_{\Psi,p1}$. □

Given the query point $\mathbf{q}$, the point location procedure first calls *findRoot* to find the appropriate root simplex Ψ of the decomposition tree and the barycentric coordinates $\boldsymbol{\alpha}$. Then it invokes the recursive procedure *search*$(0, \Psi, \boldsymbol{\alpha})$ to locate $\mathbf{q}$ within the appropriate root simplex. Once the point has been located, we can answer the interpolation query for this point. We access the stored vector field values at each of the simplex vertices, and then weight these values according to the barycentric coordinates of $\mathbf{q}$. The result is a piecewise linear, continuous interpolant.

This simple sequential search makes as many memory accesses as the depth of the final leaf simplex that contains $\mathbf{q}$. A more efficient procedure in terms of memory accesses would be to employ a doubling binary search, which computes (using only local memory) the LPT codes for the simplices at depths 0, 1, 2, 4, 8, and so on, until first finding a depth whose simplex does not exist in the hierarchy. We then use standard binary search to locate the exact depth of the leaf simplex that contains $\mathbf{q}$. Although the computation of the LPT codes is performed sequentially in time linear in the depth of the final simplex, the number accesses to the simplex decomposition tree is only logarithmic in the final depth. Thus, the running time is $O(dD)$, where D is the maximum depth of the tree, and $O(\log D)$ global memory accesses are made.

6 Neighbors in the Simplicial Complex

As we mentioned earlier, when simplices of the decomposition tree are bisected, it is necessary to bisect some of its neighbors in order to guarantee that the final subdivision is a simplicial complex. Maubach has shown how to perform bisections so this condition is satisfied [22]. (We omit discussion of how to do this for our system, but it is very similar to his.) Henceforth, let us assume

that the simplex tree decomposition has been constructed so that the underlying subdivision is a simplicial complex. In order to know what additional simplices must be bisected, it is necessary to compute neighbors within the complex. Two simplices are *neighbors* if they share a common $(d-1)$-dimensional face.

Hebert [16] presented rules for computing neighbors in the 3-dimensional case. Lee, et al. [18] presented an efficient neighbor finding algorithm for the 3-dimensional case based on the path codes. Maubach [23] considered the case of arbitrary dimensions and presented a recursive algorithm for computing neighbors, assuming that the neighbors of the parent simplex are known. This leads to a neighbor algorithm whose running time is proportional to the depth of the simplex in the decomposition tree. In this section we provide rules for computing facet neighbors based solely on their LPT codes. In all but one of the cases, the neighbor can be computed in $O(d)$ time, independent of the depth of the simplex. In the case where the computation may require time proportional to the depth in the tree, we show that this computation can be sped up by a factor of d times the machine's word size, and so is nearly constant time for practical purposes.

Consider a simplex S in the complex defined by the decomposition tree. For $0 \leq i \leq d$, let $\mathbf{v}_i$ denote its ith vertex. Exactly one $(d-1)$-face of S does not contain $\mathbf{v}_i$. If this face is not on the boundary of the base hypercube, its neighbor exists in the complex. If so, we define $N^{(i)}(S)$ to be the neighboring simplex to S lying on the opposite side of this face. Let (ℓ, Π, Φ) denote the LPT code for S and let $(\ell^{(i)}, \Pi^{(i)}, \Phi^{(i)})$ denote the LPT code for $N^{(i)}(S)$. We present rules here for computing LPT codes of these neighbors. The proof of their correctness is based on a straightforward but lengthy induction argument. The proof is presented in the appendix.

The rules compute the LPT code for the neighbor simplex at the same depth as S, and hence $\ell^{(i)} = \ell$. Of course, this simplex need not be in the decomposition tree because its parent may not yet have been bisected. In fact, in a compatible subdivision, a $(d-1)$-face neighbor of S could also appear at one level higher or one level lower than S. We will also show how to compute the LPT codes of those neighbors.

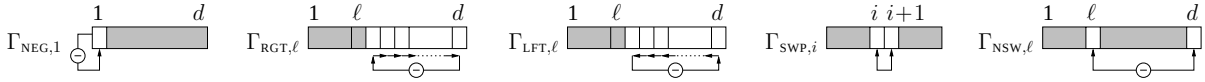


Fig. 7: Neighbor permutations. (The circle with a minus sign indicates that the element is negated.)

6.1 Neighbor Permutation Code

Each neighbor's permutation code is determined by applying one of a set of special signed permutations to Π. The permutation depends on whether S is a 0-child or a 1-child, which can be determined using the test given in Section 5.1. These permutations are illustrated in Fig. 7, and include the following: $\Gamma_{\text{NEG},1}$, negates the first element, $\Gamma_{\text{RGT},\ell}$ (resp., $\Gamma_{\text{LFT},\ell}$), shifts the last $d-\ell$ elements cyclically one position to the right (resp., left) and negates the element that was wrapped around, $\Gamma_{\text{SWP},i}$, swaps elements i and $i+1$, and $\Gamma_{\text{NSW},\ell}$, swaps and negates elements ℓ and d. The neighbor rules are given in Theorem 6.1. A number of the rules involve the parent's level, and so

to condense notation, we define $\ell^- = (\ell - 1) \bmod d$ and $\ell^* = \ell^- + 1$. Observe that $\ell^- = \ell - 1$ and $\ell^* = \ell$, except when $\ell = 0$, in which case they are larger by d. These can be computed in $O(d)$ time, and in fact in $O(1)$ time if permutations are encoded in look-up tables.

Theorem 6.1 *Let S denote a simplex at level ℓ, and let Π denote the permutation code for S.*

$$
\begin{array}{lllll}
\textbf{\textit{if }}(S \textit{ is a 0-child}): & N^{(0)}(S): & & \Pi^{(0)} & = \Pi \circ \Gamma_{\text{NEG},1} \\
& N^{(i)}(S): & (0 < i < d) & \Pi^{(i)} & = \Pi \circ \Gamma_{\text{SWP},i} \\
& N^{(d)}(S): & & \Pi^{(d)} & = \Pi \circ \Gamma_{\text{RGT},\ell^-} \\
\textbf{\textit{if }}(S \textit{ is a 1-child}): & N^{(0)}(S): & & \Pi^{(0)} & = \Pi \circ \Gamma_{\text{NEG},1} \\
& N^{(\ell^*)}(S): & & \Pi^{(\ell^*)} & = \Pi \circ \Gamma_{\text{LFT},\ell^-} \\
& N^{(i)}(S): & (0 < i < d, i \neq \ell^*) & \Pi^{(i)} & = \Pi \circ \Gamma_{\text{SWP},i} \\
& N^{(d)}(S): & (d \neq \ell^*) & \Pi^{(d)} & = \Pi \circ \Gamma_{\text{NSW},\ell}
\end{array}
$$

6.2 Neighbor Orthant

If $\ell \neq 0$ or $1 \leq i < d$, $N^{(i)}(S)$, is in the same final orthant as S, and so $\Phi^{(i)} = \Phi$. If $\ell = 0$, $N^{(d)}$ is in a different orthant than S, but, $N^{(d)}$ is the sibling of S in this case. Thus, Φ and $\Phi^{(d)}$ differ only in their last element, which is *orth*$(\Pi^{(d)})$ in $\Phi^{(d)}$. Thus the orthant list can be updated in either $O(1)$ or $O(d)$ time in these cases.

The only remaining case is $\Phi^{(0)}$. This case is the most complex because the final enclosing quadtree box of $N^{(0)}(S)$ is disjoint from S's final quadtree box. Further, it may be arbitrarily far away, in the sense that the least common ancestor of the two nodes may be the root of the tree. In this case, we can apply known methods for computing neighbors in linear quadtrees to compute $\Phi^{(0)}$ [29].

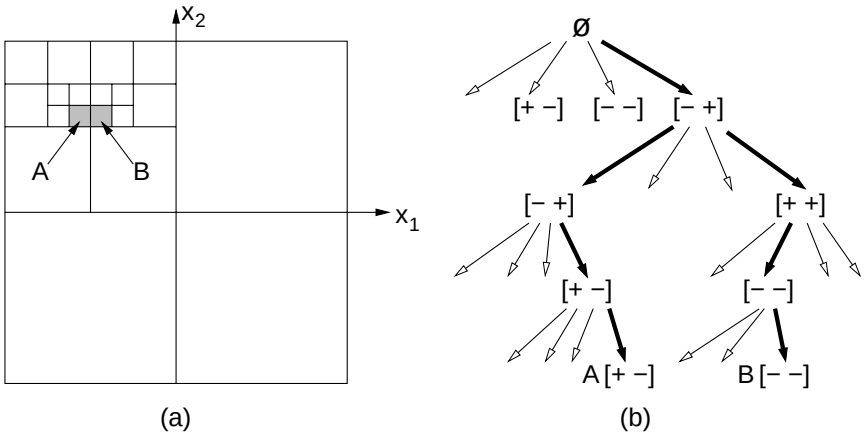


Fig. 8:

How to compute $\Phi^{(0)}$: Similar to the method of finding neighbor quadrants in a quadtree, all we need is to compute the path to the neighbor orthant. In our representation, the path from the root to the orthant is the list of orthants in Φ. Consider a 2-dimensional example. In Figure 8(a), orthants A and B are neighbors, and their associated orthant lists, written as column vectors are:

$$\Phi_A = \left\langle \begin{bmatrix} - \\ + \end{bmatrix} \begin{bmatrix} - \\ + \end{bmatrix} \begin{bmatrix} + \\ - \end{bmatrix} \begin{bmatrix} + \\ - \end{bmatrix} \right\rangle \qquad \Phi_B = \left\langle \begin{bmatrix} - \\ + \end{bmatrix} \begin{bmatrix} + \\ + \end{bmatrix} \begin{bmatrix} - \\ - \end{bmatrix} \begin{bmatrix} - \\ - \end{bmatrix} \right\rangle.$$

It is easy to see that, paths to A and B have a common prefix corresponding to their common ancestors, that is the orthant $[- +]$ in the example. Orthant entries are identical for the remainder of the paths except that one coordinate (in our example, X_1) is complemented. Fig. 8(b) illustrates the paths to A and B. The axis which has to be complemented depends on which neighbor we are looking for. This generalizes to a d-cube which is subdivided in a quadtree-like manner.

Given an orthant A whose path from the root is represented by Φ_A, and a direction defined as a 2-tuple (D, X_i) where X_i is the i^{th} coordinate axis, and $D \in \{-, +\}$ represents the direction of X_i, find the neighbor orthant B of equal size located in the given direction with respect to A.

Similar to the algorithm described for quadtrees by Samet [29], the algorithm to find the neighbor orthant B is a two-step process: Let the direction of the neighbor be (D, X_i). In terms of the tree representation, we first perform a bottom-up traversal starting from A, until we find the closest ancestor, C such that C is the parent of the lowest ancestor of A whose i^{th} coordinate is the complement of D. This is the desired common ancestor of A and B. If no such ancestor exists, then the desired neigbor B is outside the bounding box, and so, it does not exist. Otherwise, let the path from C to A be denoted as P_{CA}. In the next step, we complement the X_i coordinates in P_{CA}, to get the path from C to B, P_{CB}. And since the path from the root to C, Φ_C is common for both A and B, $\Phi_B = \Phi_C + P_{CB}$.

Finding the common ancestor C by bottom-up traversal corresponds to processing Φ_A back-to-front, complementing the X_i coordinate of each orthant, until we come across an orthant whose X_i coordinate is $-D$. We complement this coordinate as well. This completes the complementing part. Rest of the list remain the same. The resulting list is Φ_B.

Now, if we consider the original problem of computing $\Phi^{(0)}$ corresponding to the 0^{th} neighbor of simplex S, we can use the algorithm explained above, if we know which direction $N^{(0)}(S)$ is located with respect to S. Consider the Π and $\Pi^{(0)}$ corresponding to S and $N^{(0)}(S)$ respectively. By the given neighbor rules, these two permutation-reflection codes differ only in the sign of their first element. This is the sign corresponding to the $X_{|\pi_1|}$ axis, given that $\Pi = [\pi_i]_1^d$ is the code for S. The sign of π_1 determines in which direction of $X_{|\pi_1|}$ axis S resides in its final orthant. And so, the neighbor $N^{(0)}(S)$ is also in that direction. Thus, the axis component of the direction is $X_{|\pi_1|}$, and the sign component of the direction is *sign*(π_1).

Implementation Issues: This operation can be implemented very rapidly through a simple trick with bit manipulations. The neighbor computation [29] essentially involves an operation, which is applied to a bit string that consists of the ith coordinate of each entry of the orthant list. Recall from our earlier discussion of implementation issues, that the orthant list is stored as d separate

bit strings, one per coordinate, and packed into machine words as binary numbers. The key operation needed for the neighbor computation involves complementing a maximal trailing sequence of matching bits. For example, given a bit string of the form $w10^k$, for $w \in \{0,1\}^*$, the desired result is $w01^k$ (and vice versa). By packing these bits into a single word, we can compute this function with a single arithmetic operation by subtracting (or adding) 1 from the resulting binary number. (This trick has been applied elsewhere in the context of neighbor finding in quadtree-like structures [18].) Under the assumption that the machine's word size is $\Omega(D/d)$, where D is the maximum depth of any simplex, it follows that the orthant list for the neighbor can be computed in $O(1)$ time.

7 Compatible Refinement and the Simplicial Complex

A subdivision is said to be *compatible*, if each simplex S in the subdivision shares a $(d-1)$-face with exactly one neighbor simplex. A simplex decomposition tree is a simplicial complex if all of its simplices are compatible. Compatibility is important, since otherwise, cracks occur along faces of the subdivision, which in turn present problems when using the mesh for interpolation. In order to keep the subdivision compatible at all times, whenever a simplex is bisected a series of bisections will be triggered in other simplices. Hebert [16] and Maubach [22] describe the process for their systems.

For completeness we include a short description here as well. Consider a simplex S which is about to be bisected, and let e denote the next edge of S to be split. The the simplices of the subdivision that share this edge, denoted $E_e(S)$, must be bisected as well. The rules given in Section 6 provide a means to locate neighboring simplices that share a common $(d-1)$-face with S, that is, the *facet neighbors* of S. Let $N_e(S)$ denote the facet neighbors of S that contain the edge e, or equivalently, the facet neighbors lying opposite all the $d-1$ vertices of S other than the endpoints of e. In order to access all the simplices of $E_e(S)$ we compute facet neighbors recursively. The algorithm was given by Maubach [22], and is shown as the recursive function *compatBisect* in the code block. The procedure *simpleBisect* performs the basic bisection step described in Section 3.2.

```
compatBisect(S, e)
    mark S as pending
    for (S' ∈ N(S))
        if ( S' does not exist )
            compatBisect(parent(S'))  // now S' exists
        if ( S' is a leaf and not marked as pending )
            compatBisect(S')  // bisect S' and its neighbors
    simpleBisect(S)
```

Maubach proved that in a compatible subdivision, the facet neighbors of S needed in this refinement, either appear at the same depth as S or one level closer to the root [22]. For this reason, if the *compatBisect* procedure does not find a simplex S' in the tree, then it knows that its parent exists, and bisecting the parent will being S' into existence. Recall that our neighbor rules compute only the Note that the bisection of the parent may trigger recursive bisections on levels $\ell-1$ and $\ell-2$, and so on.

8 Neighbors at different depths

Neighbor rules of Theorem 6.1 provide the LPT code for the same depth neighbors. However, in a compatible subdivision, a neighbor could possibly appear one level closer or one level further from the root, that is, some neighbors of a simplex S_p at depth $|p|$, could appear at depths $|p|-1$ or $|p|+1$. We can categorize the neighbors of a simplex into two groups: neighbors that share the edge-to-be-bisected, and neighbors that do not. Maubach already proved that a neighbor sharing the edge-to-be-bisected is either at depth $|p|$ or at depth $|p|-1$, and that a neighbor at depth $|p|-1$ is the parent of the same depth neighbor which did not come into existence yet. And so, for a neighbor at depth $|p|-1$, we first compute the LPT code for the same depth neighbor by the above rules, and if the same depth neighbor does not exist in the tree, we compute its parent's LPT code as described in Section 5.

In addition, any (d-1)-face neighbor of S_p which does not share the edge-to-be-bisected could possibly be at depth $|p|+1$. Specifically, same depth neighbors $N^{\ell}(S_p)$ and $N^{(d)}(S_p)$, might have been bisected without triggering bisection of S_p, and so, one of their children will now share a face with S_p. Moreover, the child of $N^{\ell}(S_p)$ or $N^{(d)}(S_p)$ that shares a face with S_p, is the same depth neighbor of one of the children of S_p. So, we can compute a neighbor at depth $|p|+1$ by computing the appropriate same depth neighbor of one of the children of S_p. Formally,

if($N^{\ell}(S_p)$ is a bisected simplex)
    $N^{\ell}(S_{p0})$ is the neighbor of S_p across vertex $\mathbf{v}_{\ell}$,
if($N^{(d)}(S_p)$ is a bisected simplex)
    $N^{\ell}(S_{p1})$ is the neighbor of S_p across vertex $\mathbf{v}_d$.

It can be easily proven that these neighbors cannot exist at depths higher than $|p|+1$. Intuitively, same depth neighbor $N^{\ell}(S_p)$ ($N^{(d)}(S_p)$) have exactly one vertex different from S_p. Let that vertex be **u**. It can be shown that when $N^{\ell}(S_p)$ ($N^{(d)}(S_p)$) is bisected, **u** is one of the endpoints of the bisected edge. So, one of the children of $N^{\ell}(S_p)$ ($N^{(d)}(S_p)$) will have two vertices different from S_p, and cannot be a neighbor. The other child has exactly one vertex (**u**) different from S_p, thus is a neighbor of S_p. If that child is further bisected however, its children will have an additional new vertex created by bisection of an edge which does not contain **u**, hence these children at depth $|p|+2$ cannot be neighbors of S_p.

9 Summary of Implementation

We have implemented the Simplex Decomposition Tree and utilized it for interpolating in 4-dimensional space in a ray-tracing application. The data structure could be used for other applications given a black-box sampling procedure and a measure of discrepancy between samples. Additional information for detecting discontinuties would also be useful, for example in the ray-tracing application, we make use of the surface patches hit by sample rays to detect discontinuties.

Sampling: Samples are collected at simplex vertices adaptively such that a simplex is bisected only when the desired accuracy of interpolation cannot be achieved within that simplex, and also when the bisection of a neighboring simplex triggers its bisection. When bisection is performed, a new sample is generated at the newly created vertex position. Through a hash of vertices, we make sure that a vertex is sampled only once.

$LPT\ code$ **Storage and Manipulation:**

- In our implementation, we treat the signed permutation component as a reflection and a permutation separately, as in the initial description given in Section 3.1. Note that the reflection could be one of 2^d reflections, and the permutation could be one of $d!$ permutations. Both the reflection and the permutation are represented by a unique integer-id. Each of the $d!$ permutations is assigned a unique integer-id in $\{0, 1, 2, \ldots, (d!-1)\}$. A reflection $R = [r_1\ r_2\ \cdots\ r_d]$ where $r_i \in \{\pm 1\}$, is also assigned a unique integer-id, $r \in \{0, 1, 2, \ldots, (2^d - 1)\}$, such that

$$r = \sum_{k=1}^{d} \frac{(r_k + 1)}{2} \times 2^{d-k}$$

  And so, the permutation-reflection component of the $LPT\ code$ is stored as two integers ($\lceil \log_2(d!) \rceil + d$ bits).

- The operations defined on the permutation-reflection component such as cyclical shifts and swaps are performed through use of precomputed tables. Each possible operation is also given a unique integer-id. We precompute two tables, one for permutations, and one for reflections. There is an entry for each possible permutation/reflection and each possible operation combination. Then, the permutation/reflection integer-id and the operation-id could be used as indices to these tables to get the integer-id of the resulting permutation/reflection. By these tables, all operations are done in $O(1)$ time.

- As mentioned in Section 6.2, orthant lists are stored a d separate bit strings, one for each coordinate.

- To access a simplex, its $LPT\ code$ is used to index a hashtable.

Querying and Interpolation: Once the query points is located in one of the leaf simplices, as explained in Section 5.2, the barycentric coordinates with respect to the simplex vertices are already known. The output value for the query point is then computed by barycentric interpolation of the $d + 1$ simplex vertices.

However, if the the data structure is being built dynamically, such that the simplices are generated on-demand, only when they are required by interpolation, then the interpolation query would cause more bisections and sampling until the desired accuracy is achieved. In this case, the simplex is first bisected and then the traversal down the tree is continued with one of its children as usual.

10 Conclusion

We have presented a representation of hierarchical regular simplicial meshes based on Maubach's [22] simplex bisection algorithm. Unlike Maubach's approach, which requires the use of recursion or an explicit tree structure, our representation is pointerless, that is, the simplices of the mesh are uniquely identified through a location code, called the LPT code. We have shown how to use this code to traverse the hierarchy, compute neighbors, and to answer point location and interpolation queries.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[2] E. Allgower and K. Georg. Generation of triangulations by reflection. *Utilitas Mathematica*, 16:123–129, 1979.

[3] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[4] D.N. Arnold, A. Mukherjee, and L.Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM J. Sci. Comput.*, 22(2):431–448, 2001.

[5] F. B. Atalay and D. M. Mount. Ray interpolants for fast ray-tracing relections and refractions. *J. of WSCG*, 10(3):1–8, 2002. Proc. Int. Conf. in Central Europe on Comp. Graph., Visual. and Comp. Vision.

[6] F. B. Atalay and D. M. Mount. Interpolation over light fields with applications in computer graphics. In *Proc. of the 5th Workshop on Algorithm Engineering and Experiments (ALENEX 2003)*, pages 56–68. SIAM, 2003.

[7] R.E. Bank, A.H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing*, pages 3–17, 1983.

[8] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *J. Comput. Syst. Sci.*, 48:384–409, 1994.

[9] J. Bey. Tetrahedral grid refinement. *Computing*, 55:355–378, 1995.

[10] T. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Programming Languages Design and Implementation*, 1999.

[11] E. D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, 2002.

[12] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.

[13] I. Gargantini. An effective way to represent quad-trees. *Commun. ACM*, 25(12):905–910, 1982.

[14] T. Gerstner and M. Rumpf. Multiresolutional parallel isosurface exraction based on tetrahedral bisection. In *Proc. Symp. Volume Visualization*, 1999.

[15] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press, San Diego, 1989.

[16] D. J. Hebert. Symbolic local refinement of tetrahedral grids. *J. of Symbolic Comput.*, 17:457–472, 1994.

[17] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[18] M. Lee, L. De Floriani, and H. Samet. Constant-time neighbor finding in hierarchical tetrahedral meshes. In *Proc. Int. Conf. on Shape Modelling*, pages 286–295, 2001.

[19] M. Lee and H. Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Trans. on Computer Graphics*, 19:79–121, 2000.

[20] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM J. Sci. Comput.*, 16:1269–1291, 1995.

[21] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision. *Math. of Comput.*, 65(215):1183–1200, 1996.

[22] J. M. Maubach. Local bisection refinement for N-simplicial grids generated by reflection. *SIAM J. Sci. Stat. Comput.*, 16:210–227, 1995.

[23] J. M. Maubach. The efficient location of neighbors for locally refined n-simplicial grids. In *5th Int. Meshing Roundable*, 1996.

[24] W. F. Mitchell. Optimal multilevel iterative methods for adaptive grids. *SIAM J. Sci. Stat. Comput.*, 13:146–167, 1992.

[25] J. R. Munkres. *Topology: A first course*. Prentice Hall, Englewood Cliffs, NJ, 1975.

[26] M. Ohlberger and M. Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 56:365–385, 1997.

[27] R. Pajarola. Overview of quadtree-based terrain triangulation and visualization. Technical report, UCI-ICS-02-01, Information & Computer Science, University of California Irvine, 2002.

[28] M.C. Rivara. Local modification of meshes for adaptive and/or multigrid finite-element methods. *J. Comput. Appl. Math.*, 36:79–89, 1991.

[29] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.

[30] Y. Zhou, B. Chen, and A. Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In *Proc. IEEE Visualization'97*, pages 135–142, 1997.

Appendix

Proof of Theorem 6.1

The following notation will be used throughout the proof. S denotes any simplex.
$S^{(i)} = N^{(i)}(S)$, i.e. the i^{th} neighbor of S.
Π and $\Pi^{(i)}$ denote the signed permutation code associated with S and $S^{(i)}$ respectively.
S_0 and S_1 denote the 0- and 1-children of S, respectively.
$S_0^{(i)}$ and $S_1^{(i)}$ denote the 0- and 1-children of $S^{(i)}$, respectively.
Π_0 and Π_1 denote the signed permutation code associated with S_0 and S_1, respectively.
$\Pi_0^{(i)}$ and $\Pi_1^{(i)}$ denote the signed permutation code associated with $S_0^{(i)}$ and $S_1^{(i)}$, respectively.
$(S_0)^{(i)}$ denote the i^{th} neighbor of S_0. $(\Pi_0)^{(i)}$ denotes the code for $(S_0)^{(i)}$.
$(S_1)^{(i)}$ denote the i^{th} neighbor of S_1. $(\Pi_1)^{(i)}$ denotes the code for $(S_1)^{(i)}$.
m and **m**′ are used to denote the new vertex generated by bisection.
u is used for the vertex that differs in the neighbor simplex.

Inductive Hypothesis: Let $S = [\mathbf{v}_0 \dots \mathbf{v}_\ell \dots \mathbf{v}_d]^T$ be a simplex at level $\ell = |p| \bmod d$. Let $\ell^- = (\ell - 1) \bmod d$ and $\ell^* = \ell^- + 1$.

The rules of the theorem could be stated more explicitly as:
if(S is a 0-child)

$$S^{(0)} = [\mathbf{u} \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T \qquad \Pi^{(0)} = \Pi \circ \Gamma_{\text{NEG},1},$$
$$S^{(i)} = [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{u} \quad \mathbf{v}_{i+1} \dots \mathbf{v}_d]^T, \quad (0 < i < d) \qquad \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i},$$
$$S^{(d)} = [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \quad \mathbf{u} \quad \mathbf{v}_{\ell^-+1} \dots \mathbf{v}_{d-1}]^T \qquad \Pi^{(d)} = \Pi \circ \Gamma_{\text{RGT},\ell^-},$$

if(S is a 1-child)

$$S^{(0)} = [\mathbf{u} \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T \qquad \Pi^{(0)} = \Pi \circ \Gamma_{\text{NEG},1},$$
$$S^{(\ell^*)} = [\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \quad \mathbf{v}_{\ell^-+2} \dots \mathbf{v}_d \quad \mathbf{u}]^T \qquad \Pi^{(\ell^*)} = \Pi \circ \Gamma_{\text{LFT},\ell^-},$$
$$S^{(i)} = [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{u} \quad \mathbf{v}_{i+1} \dots \mathbf{v}_d]^T, \quad (0 < i < d, i \neq \ell^*) \qquad \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i},$$

$$S^{(d)} = [\mathbf{v}_0 \dots \mathbf{v}_{d-1} \quad \mathbf{u}]^T, \quad (d \neq \ell^*) \qquad\qquad \Pi^{(d)} = \Pi \circ \Gamma_{\text{NSW},\ell}.$$

Basis of Induction: We will show that the neighbor rules hold for the $d!$ root simplices. Note that the *level* of a root simplex is 0, and the rules are the same whether the simplex is a 0-child, or a 1-child. Let S denote any root simplex, with $LPT\ code\ \Pi = [\pi_1 \dots \pi_i \quad \pi_{i+1} \dots \pi_d]$.

- For all root simplices, $S^{(0)} = \emptyset$, and $S^{(d)} = \emptyset$, that is, the 0^{th} and the d^{th} neighbors do not exist, since they are outside the *reference hypercube*.
- Other neighbors, $S^{(i)}$, $0 < i < d$, should be obtainable by swaps. Recall that the *base simplex* $S_\emptyset$ could be represented as,

$$S_\emptyset = [\mathbf{y}_1 \dots \mathbf{y}_d], \quad \mathbf{y}_i = \begin{bmatrix} y_{i,0} \\ \vdots \\ y_{i,i-1} \\ y_{i,i} \\ \vdots \\ y_{i,d} \end{bmatrix} = \begin{bmatrix} -1 \\ \vdots \\ -1 \\ 1 \\ \vdots \\ 1 \end{bmatrix},$$

And, any root simplex S could be written as,

$$S = [\mathbf{y}'_1 \dots \mathbf{y}'_d], \quad \mathbf{y}'_j = \mathbf{y}_i = \begin{bmatrix} y_{i,0} \\ \vdots \\ y_{i,i-1} \\ y_{i,i} \\ \vdots \\ y_{i,d} \end{bmatrix} = \begin{bmatrix} -1 \\ \vdots \\ -1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad \text{iff} \quad \pi_i = j$$

If $\pi_i = j$, and $\pi_{i+1} = k$, then $\mathbf{y}'_j = \mathbf{y}_i$ and $\mathbf{y}'_k = \mathbf{y}_{i+1}$.

$$\Pi = [\pi_1 \dots \pi_{i-1} \quad j \quad k \quad \pi_{i+2} \dots \pi_d]$$

Note that swapping columns $\mathbf{y}'_j$ and $\mathbf{y}'_k$ of S, will give us another valid root simplex, S' which differs from S only in the i^{th} row, that is the i^{th} vertex. So, S' is basically the i^{th} neighbor of S, that is $S' = S^{(i)}$. Let Π' denote the signed permutation for S'. Then,

$$\Pi' = [\pi_1 \dots \pi_{i-1} \quad k \quad j \quad \pi_{i+2} \dots \pi_d].$$

This shows that the i^{th} and the $(i+1)^{th}$ entries in Π is swapped to get Π', And so,

$$\Pi^{(i)} = \Pi' = \Pi \circ \Gamma_{\text{SWP},i}.$$

Induction Step: Let S be a simplex at *level* ℓ^- such that the inductive hypothesis holds. We will show that the inductive hypothesis holds for the two children of S.

1. First, consider the 0-child of S, that is S_0. Let ℓ denote the level of S_0. Note that $\ell^- = (\ell - 1) \bmod d$. Let $\ell^* = \ell^- + 1$. Letting i denote the neighbor number, there are multiple cases to be distinguished.

   (a) $i = 0$

   If $0 < \ell^- \leq d - 1$,

$$\begin{aligned} S&=[\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, & S_0&=[\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \\ S^{(0)}&=[\ \mathbf{u} \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, & S_0^{(0)}&=[\ \mathbf{u} \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \end{aligned}$$

   Else if $\ell^- = 0$

$$\begin{aligned} S&=[\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T, & S_0&=[\mathbf{m} \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T \\ S^{(0)}&=[\ \mathbf{u} \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T, & S_0^{(0)}&=[\mathbf{m}' \quad \mathbf{v}_1 \dots \mathbf{v}_d]^T \end{aligned}$$

   In either case, $(S_0)^{(0)} = S_0^{(0)}$. And so, $(\Pi_0)^{(0)} = \Pi_0^{(0)} = \Pi^{(0)} = \Pi \circ \Gamma_{\text{NEG},1} = \Pi_0 \circ \Gamma_{\text{NEG},1}$.

   (b) $i = d$

   By definition of bisection rules, d^{th} neighbor of S_0 is its sibling, that is S_1, and

$$(\Pi_0)^{(d)} = \Pi_1 = \Pi_0 \circ \Gamma_{\text{RGT},\ell^-}.$$

   (c) $0 < i < \ell^-$

$$\begin{aligned} S&=[\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \quad \mathbf{v}_{i+1} \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, \\ S^{(i)}&=[\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{u} \quad \mathbf{v}_{i+1} \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, \\ S_0&=[\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \quad \mathbf{v}_{i+1} \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T, \\ S_0^{(i)}&=[\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{u} \quad \mathbf{v}_{i+1} \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \end{aligned}$$

   Then, $(S_0)^{(i)} = S_0^{(i)}$. And so, $(\Pi_0)^{(i)} = \Pi_0^{(i)} = \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i} = \Pi_0 \circ \Gamma_{\text{SWP},i}$.

   (d) $i = \ell^-$, $\ell^- \neq 0$

      i. If S is a 0-child

$$\begin{aligned} S&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, & S_0&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \dots \mathbf{v}_d]^T \\ S^{(\ell^-)}&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{y} \dots \mathbf{v}_d]^T, & S_0^{(\ell^-)}&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m}' \dots \mathbf{v}_d]^T \end{aligned}$$

      Then, $(S_0)^{(\ell^-)} = S_0^{(\ell^-)}$. And so, $(\Pi_0)^{(\ell^-)} = \Pi_0^{(\ell^-)} = \Pi^{(\ell^-)} = \Pi \circ \Gamma_{\text{SWP},\ell^-} = \Pi_0 \circ \Gamma_{\text{SWP},\ell^-}$.

ii. If S is a 1-child,

$$\begin{aligned} S{=}[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^-} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \ , \qquad & S_0{=}[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \\ S^{(\ell^-)}{=}[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^*} \dots \dots \mathbf{v}_d \quad \mathbf{u}]^T, \quad & S_1^{(\ell^-)}{=}[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m}' \quad \mathbf{v}_{\ell^*} \dots \mathbf{v}_d]^T \end{aligned}$$

Then, $(S_0)^{(\ell^-)} = S_1^{(\ell^-)}$.

$$\begin{aligned} \Pi_0 = \Pi &= [\pi_1 \dots \pi_d], \\ \Pi^{(\ell^-)} &= [\pi_1 \dots \pi_{\ell^- -1} \quad \pi_{\ell^*} \dots \pi_d \quad -\pi_{\ell^-}], \\ \Pi_1^{(\ell^-)} &= [\pi_1 \dots \pi_{\ell^- -1} \quad \pi_{\ell^*} \quad \pi_{\ell^-} \quad \pi_{\ell^*+1} \dots \pi_d] \end{aligned}$$

And so, $(\Pi_0)^{(\ell^-)} = \Pi_1^{(\ell^-)} = \Pi_0 \circ \Gamma_{\text{SWP},\ell^-}$.

(e) $\ell^- < i < d$

$$\begin{aligned} S{=}[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_d]^T, \quad & S_0{=}[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_d]^T \\ S^{(i)}{=}[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-1} \quad \mathbf{y} \dots \mathbf{v}_d]^T, \quad & S_0^{(i)}{=}[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{i-1} \quad \mathbf{y} \dots \mathbf{v}_d]^T \end{aligned}$$

Then, $(S_0)^{(i)} = S_0^{(i)}$. And so, $(\Pi_0)^{(i)} = \Pi_0^{(i)} = \Pi^{(i)} = \Pi \circ \Gamma_{\text{SWP},i} = \Pi_0 \circ \Gamma_{\text{SWP},i}$.

2. Next, consider the 1-child of S, that is S_1. Let ℓ denote the level of S_1. Note that $\ell^- = (\ell - 1) \bmod d$. Let $\ell^* = \ell^- + 1$. Letting i denote the neighbor number, again, there are multiple cases.

(a) $i = 0$

i. $\ell^- = 0$

$$\begin{aligned} S{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T, \quad & S_1{=}[\mathbf{m} \quad \mathbf{v}_0 \dots \mathbf{v}_{d-1}]^T \\ S^{(d)}{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{d-1} \quad \mathbf{u}]^T, \quad & S_1^{(d)}{=}[\mathbf{m}' \quad \mathbf{v}_0 \dots \mathbf{v}_{d-1}]^T \end{aligned}$$

Then, $(S_1)^{(0)} = S_1^{(d)}$.

$$\begin{aligned} \Pi &= [\pi_1 \dots \pi_d], & \Pi_1 &= [-\pi_d \quad \pi_1 \dots \pi_{d-1}], \\ \Pi^{(d)} &= [\pi_1 \dots \pi_{d-1} \quad -\pi_d], & \Pi_1^{(d)} &= [\pi_d \quad \pi_1 \dots \pi_{d-1}] \end{aligned}$$

And so, $(\Pi_1)^{(0)} = \Pi_1^{(d)} = \Pi_1 \circ \Gamma_{\text{NEG},1}$.

ii. $\ell^- \neq 0$

$$\begin{aligned} S{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, \quad & S_1{=}[\mathbf{v}_0 \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \\ S^{(0)}{=}[\ \mathbf{y} \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, \quad & S_1^{(0)}{=}[\ \mathbf{y} \quad \mathbf{v}_1 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \end{aligned}$$

Then, $(S_1)^{(0)} = S_1^{(0)}$.

$$\begin{array}{ll} \Pi = [\pi_1 \dots \pi_d], & \Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}], \\ \Pi^{(0)} = [-\pi_1 \dots \pi_d], & \Pi_1^{(0)} = [-\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}] \end{array}$$

And so, $(\Pi_1)^{(0)} = \Pi_1^{(0)} = \Pi_1 \circ \Gamma_{\text{NEG},1}$.

(b) $0 < i < \ell^-$

$$\begin{array}{rrl} S = & [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_{\ell^--1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, & S_1 = [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_{\ell^--1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \\ S^{(i)} = & [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{y} \dots \mathbf{v}_{\ell^--1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_d]^T, & S_1^{(i)} = [\mathbf{v}_0 \dots \mathbf{v}_{i-1} \quad \mathbf{y} \dots \mathbf{v}_{\ell^--1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \end{array}$$

Then, $(S_1)^{(i)} = S_1^{(i)}$.

$$\begin{array}{l} \Pi = [\pi_1 \dots \pi_d], \\ \Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}], \\ \Pi^{(i)} = [\pi_1 \dots \pi_{i-1} \quad \pi_{i+1} \quad \pi i \quad \pi_{i+2} \dots \pi_d], \\ \Pi_1^{(i)} = [\pi_1 \dots \pi_{i-1} \quad \pi_{i+1} \quad \pi i \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}] \end{array}$$

And so, $(\Pi_1)^{(i)} = \Pi_1^{(i)} = \Pi_1 \circ \Gamma_{\text{SWP},i}$.

(c) $i = \ell^-$

i. If S is a 0-child

$$\begin{array}{rrl} S = & [\mathbf{v}_0 \dots \mathbf{v}_{\ell^--1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T, & S_1 = [\mathbf{v}_0 \dots \mathbf{v}_{\ell^--1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \\ S^{(d)} = & [\mathbf{v}_0 \dots \mathbf{v}_{\ell^--1} \quad \mathbf{y} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T, & S_0^{(d)} = [\mathbf{v}_0 \dots \mathbf{v}_{\ell^--1} \quad \mathbf{m}' \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \end{array}$$

Then, $(S_1)^{(\ell^-)} = S_0^{(d)}$.

$$\begin{array}{l} \Pi = [\pi_1 \dots \pi_d], \\ \Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}], \\ \Pi_0^{(d)} = \Pi^{(d)} = [\pi_1 \dots \pi_{\ell^--1} \quad -\pi_d \quad \pi_{\ell^-} \dots \pi_{d-1}] \end{array}$$

And so, $(\Pi_1)^{(\ell^-)} = \Pi_0^{(d)} = \Pi_1 \circ \Gamma_{\text{SWP},\ell^-}$.

ii. If S is a 1-child,

$$\begin{array}{rrl} S = & [\mathbf{v}_0 \dots \mathbf{v}_{\ell^--1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T, & S_1 = [\mathbf{v}_0 \dots \mathbf{v}_{\ell^--1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \\ S^{(d)} = & [\mathbf{v}_0 \dots \mathbf{v}_{\ell^--1} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1} \quad \mathbf{u}]^T, & S_1^{(d)} = [\mathbf{v}_0 \dots \mathbf{v}_{\ell^--1} \quad \mathbf{m}' \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-1}]^T \end{array}$$

Then, $(S_1)^{(\ell^-)} = S_1^{(d)}$.

$$\begin{aligned}
\Pi &= [\pi_1 \dots \pi_d],\\
\Pi_1 &= [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}],\\
\Pi^{(d)} &= [\pi_1 \dots \pi_{\ell^- -1} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1} \quad -\pi_{\ell^-}],\\
\Pi_1^{(d)} &= [\pi_1 \dots \pi_{\ell^- -1} \quad -\pi_d \quad \pi_{\ell^-} \dots \pi_{d-1}]
\end{aligned}$$

And so, $(\Pi_1)^{(\ell^-)} = \Pi_1^{(d)} = \Pi_1 \circ \Gamma_{\text{SWP},\ell^-}$.

(d) $i = \ell^*$

By definition of bisection rules, $(\ell^*)^{th}$ neighbor of S_1 is its sibling, that is S_0, and

$$(\Pi_1)^{(\ell^*)} = \Pi_0 = \Pi_1 \circ \Gamma_{\text{LFT},\ell^-}.$$

(e) $\ell^* < i \le d, \quad \ell \neq 0$

i. $\ell^* < i < d$, $\ell \neq 0$,

$$\begin{aligned}
S&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-2} \quad \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T,\\
S^{(i-1)}&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-2} \quad \mathbf{u} \quad \mathbf{v}_i \dots \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T,\\
S_1&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-2} \quad \mathbf{v}_{i-1} \quad \mathbf{v}_i \dots \mathbf{v}_{d-1}]^T,\\
S_1^{(i-1)}&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{i-2} \quad \mathbf{u} \quad \mathbf{v}_i \dots \mathbf{v}_{d-1}]^T
\end{aligned}$$

Then, $(S_1)^{(i)} = S_1^{(i-1)}$.

$$\begin{aligned}
&\Pi = [\pi_1 \dots \pi_{\ell^-} \dots \pi_{i-1} \quad \pi_i \dots \pi_d],\\
&\Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{i-1} \quad \pi_i \dots \pi_{d-1}], \quad \text{since} \quad i-1 > \ell^-\\
&\Pi^{(i-1)} = [\pi_1 \dots \pi_{i-2} \quad \pi_i \quad \pi_{i-1} \quad \pi_{i+1} \dots \pi_d],\\
&\Pi_1^{(i-1)} = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{i-2} \quad \pi_i \quad \pi_{i-1} \quad \pi_{i+1} \dots \pi_{d-1}].
\end{aligned}$$

And so, $(\Pi_1)^{(i)} = \Pi_1^{(i-1)} = \Pi_1 \circ \Gamma_{\text{SWP},i}$.

ii. $i = d$, $\ell \neq 0$

$$\begin{aligned}
S&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-2} \quad \mathbf{v}_{d-1} \quad \mathbf{v}_d]^T,\\
S^{(d-1)}&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-2} \quad \mathbf{u} \quad \mathbf{v}_d]^T,\\
S_1&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-2} \quad \mathbf{v}_{d-1}]^T,\\
S_1^{(d-1)}&=[\mathbf{v}_0 \dots \mathbf{v}_{\ell^- -1} \quad \mathbf{m} \quad \mathbf{v}_{\ell^-} \dots \mathbf{v}_{d-2} \quad \mathbf{u}]^T
\end{aligned}$$

Then, $(S_1)^{(d)} = S_1^{(d-1)}$.

$$\begin{aligned}
&\Pi = [\pi_1 \dots \pi_{\ell^-} \dots \pi_{d-1} \quad \pi_d],\\
&\Pi_1 = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_d \quad \pi_{\ell^*} \dots \pi_{d-1}], \quad \text{since} \quad d-1 > \ell^-\\
&\Pi^{(d-1)} = [\pi_1 \dots \pi_{d-2} \quad \pi_d \quad \pi_{d-1}],\\
&\Pi_1^{(d-1)} = [\pi_1 \dots \pi_{\ell^-} \quad -\pi_{d-1} \quad \pi_{\ell^*} \dots \pi_{d-2} \quad \pi_d].
\end{aligned}$$

And so, $(\Pi_1)^{(d)} = \Pi_1^{(d-1)} = \Pi_1 \circ \Gamma_{\text{NSW},\ell^*} = \Pi_1 \circ \Gamma_{\text{NSW},\ell}$.