Vectorized algorithms for regular tessellations of d-orthotopes and their faces

François Cuvelier, Gilles Scarella

Vectorized algorithms for regular tessellations of d-orthotopes and their faces

François Cuvelier* Gilles Scarella †

2018/01/14

Abstract

Tessellation of hypercubes or orthotopes and all their faces in any dimension is a nice challenge. The purpose of this paper is to describe efficient vectorized algorithms to obtain regular tessellations made up by simplices or orthotopes. These vectorized algorithms have been implemented in array programming languages such as Matlab/Octave, Python.

Contents

*Université Paris 13, Sorbonne Paris Cité, LAGA, CNRS UMR 7539, 99 Avenue J-B Clément, F-93430 Villetaneuse, France, cuvelier@math.univ-paris13.fr

†Université Côte d'Azur, CNRS, LJAD, F-06108 Nice, France, gilles.scarella@unice.fr.

This work was supported by the ANR project DEDALES under grant ANR-14-CE23-0005.

There are many tools for generating conformal meshes in 2 or 3 dimension such as GMSH , Open CASCADE, ... They can be used, for example, to solve boundary value problems in 2D or 3D by finite element methods. In [9], vectorized algorithms are proposed to calculate some assembly matrices obtained by the $\mathbb{P}_1$-Lagrange finite element method. These algorithms are written in any dimension and they have been implemented in Matlab/Octave, Python, C++, CUDA. Subsequently, complete codes were written to solve boundary value problems (B.V.P.) in any space dimension ([6] for Matlab, [7] for Octave and [8] for python). To test these codes for dimensions greater than 3 we need simplicial meshes in dimension 4, 5, ... To our knowledge, mesh generation tools in dimensions greater than 3 are not available so we decided to write one for the simplest geometry: an d-orthotope (also called hyperrectangle or a box).

The objective of this paper is therefore to propose vectorized algorithms to build regular tessellations of a d-orthotope made up by orthotopes in section 2 and by simplicials in section 3 To solve BVP we also need to know precisely all the meshes of the faces that are part of the d-orthotope. That is why we will develop, in each section, techniques to recover all the meshes associated to the m-faces of the d-orthotope, $0 \leqslant m \leqslant d$. In section 4, the performances of these vectorized algorithms are measured with Matlab 2017a, Octave 4.2.1 and Python 3.6.3 to validate their efficiency. But first of all, we recall some usual notations and definitions.

1 Definitions and notations

In this part, we characterize the basic geometric elements that will be used later on. Some of their properties are recalled. But before we specify notations commonly used in this paper to define set of integers:

$$\begin{aligned} &[\![i,j]\!] \stackrel{\text{def}}{=} \{i,\cdots,j\}, && [\![i,j[\![\stackrel{\text{def}}{=} \{i,\cdots,j-1\}, \\ &]\!]i,j]\!] \stackrel{\text{def}}{=} \{i+1,\cdots,j\}, &&]\!]i,j[\![\stackrel{\text{def}}{=} \{i+1,\cdots,j-1\}. \end{aligned}$$

1.1 d-orthotope and d-hypercube

We first recall the definitions of a d-orthotope and a d-hypercube given in [2].

Definition 1 *In geometry, a d-**orthotope** (also called a hyperrectangle or a box) is the generalization of a rectangle for higher dimensions, formally defined as the Cartesian product of intervals.*

Definition 2 *A d-orthotope with all edges of the same length is a d-**hypercube**.*
*A d-orthotope with all edges of length one is **a unit** d-**hypercube**.*
*The hypercube $[0,1]^d$ is called the **unit reference** d-**hypercube**.*

The m-orthotopes on the boundary of a d-orthotope, $0 \leqslant m \leqslant d$, are called the m-**faces of the** d-**orthotope**.

The number of m-faces of a d-orthotope is

$$E_{m,d} = 2^{d-m}\binom{d}{m} \quad \text{where} \quad \binom{d}{m} = \frac{d!}{m!(d-m)!} \tag{1}$$

For example, the 2-faces of the unit 3-hypercube $[0,1]^3$ are the sets

$$\begin{array}{ll} \{0\}\times[0,1]\times[0,1], & \{1\}\times[0,1]\times[0,1], \\ [0,1]\times\{0\}\times[0,1], & [0,1]\times\{1\}\times[0,1], \\ [0,1]\times[0,1]\times\{0\}, & [0,1]\times[0,1]\times\{1\}. \end{array}$$

Its 1-faces are

$$\begin{array}{ll} \{0\}\times\{0\}\times[0,1], & \{0\}\times\{1\}\times[0,1], \\ \{1\}\times\{0\}\times[0,1], & \{1\}\times\{1\}\times[0,1], \\ \{0\}\times[0,1]\times\{0\}, & \{0\}\times[0,1]\times\{1\}, \\ \{1\}\times[0,1]\times\{0\}, & \{1\}\times[0,1]\times\{1\}, \\ [0,1]\times\{0\}\times\{0\}, & [0,1]\times\{0\}\times\{1\}, \\ [0,1]\times\{1\}\times\{0\}, & [0,1]\times\{1\}\times\{1\}, \end{array}$$

and its 0-faces are

$$\begin{array}{ll} \{0\}\times\{0\}\times\{0\}, & \{1\}\times\{0\}\times\{0\}, \\ \{0\}\times\{1\}\times\{0\}, & \{0\}\times\{0\}\times\{1\}, \\ \{1\}\times\{1\}\times\{0\}, & \{1\}\times\{0\}\times\{1\}, \\ \{0\}\times\{1\}\times\{1\}, & \{1\}\times\{1\}\times\{1\}. \end{array}$$

We represent in Figure 1 all the m-faces of a 3D hypercube.

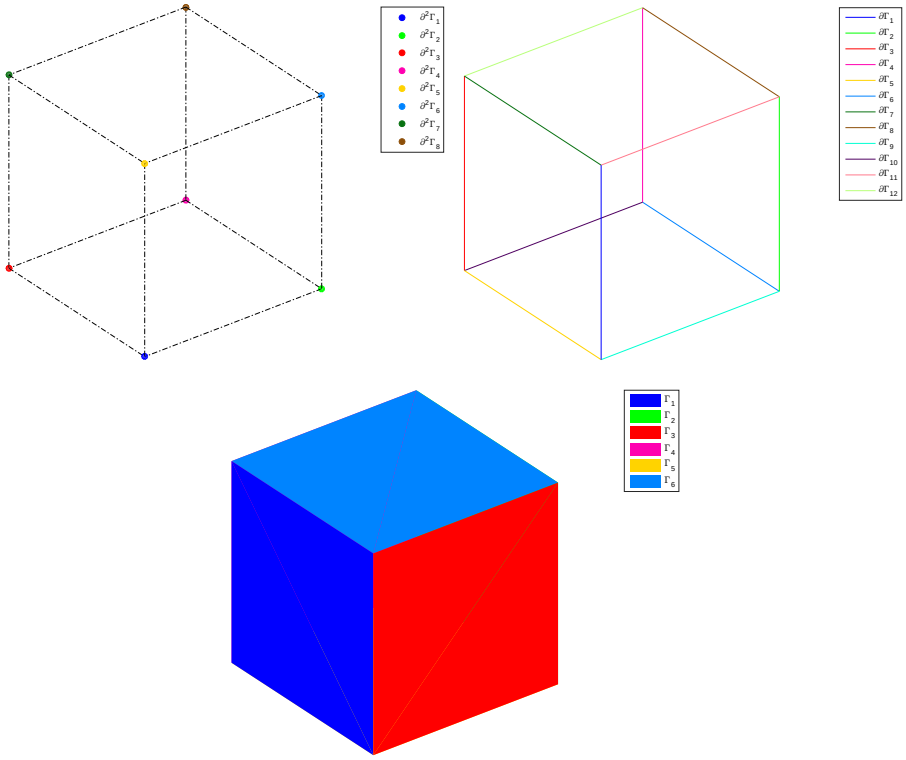


Figure 1: m-faces of a 3D hypercube : 0-faces (upper left), 1-faces (upper right) and 2-faces (bottom)

In Table 1 is given the number of m-faces for $m \in [\![0,d]\!]$ and $d \in [\![0,6]\!]$.

	m	0	1	2	3	4	5	6
d	Names	0-face	1-face	2-face	3-face	4-face	5-face	6-face
0	*Point*	1						
1	*Segment*	2	1					
2	*Square*	4	4	1				
3	*Cube*	8	12	6	1			
4	*Tesseract*	16	32	24	8	1		
5	*Penteract*	32	80	80	40	10	1	
6	*Hexeract*	64	192	240	160	60	12	1

Table 1: Number of m-faces of a d-hypercube

The identification/numbering of the m-faces is given in section2.3.

1.2 d-simplex

Definition 3 *In geometry, a* ***simplex*** *(plural: simplexes or simplices) is a generalization of the notion of a triangle or tetrahedron to arbitrary dimensions. Specifically, a d-simplex is a d-dimensional polytope which is the convex hull of its $d+1$ vertices. More formally, suppose the $d+1$ points $\mathbf{q}^0, \ldots, \mathbf{q}^d \in \mathbb{R}^d$ are affinely independent, which means $\mathbf{q}^1 - \mathbf{q}^0, \ldots, \mathbf{q}^d - \mathbf{q}^0$ are linearly independent. Then, the simplex determined by them is the set of points*

$$C = \{\theta_0 \mathbf{q}^0 + \cdots + \theta_d \mathbf{q}^d | \theta_i \geqslant 0, 0 \leqslant i \leqslant d, \sum_{i=0}^{d} \theta_i = 1\}.$$

For example, a 2-simplex is a triangle, a 3-simplex is a tetrahedron, and a 4-simplex is a 5-cell. A single point may be considered as a 0-simplex and a line segment may be considered as a 1-simplex. A simplex may be defined as the smallest convex set which contain the given vertices.

Definition 4 *Let $\mathbf{q}^0, \ldots, \mathbf{q}^d \in \mathbb{R}^d$ be the $d+1$ vertices of a d-simplex K and $\mathbb{D}_K$ be the $(d+1)$-by-$(d+1)$ matrix defined by*

$$\mathbb{D}_K = \left(\begin{array}{c:c:c} \mathbf{q}^0 & \cdots & \mathbf{q}^d \\ \hdashline 1 & \cdots & 1 \end{array} \right)$$

The d-simplex K is

- ***degenerated*** *if* $\det \mathbb{D}_K = 0$,
- ***positive oriented*** *if* $\det \mathbb{D}_K > 0$,
- ***negative oriented*** *if* $\det \mathbb{D}_K < 0$.

The m-simplices on the boundary of a d-simplex, $0 \leqslant m \leqslant d$, are called the m-**faces of the** d-**simplex**. If a d-simplex is nondegenerate, its number of m-faces, denoted by $S_{m,d}$, is given by

$$S_{m,d} = \binom{d+1}{m+1} \tag{2}$$

We give in Table 2 this number for $d \in [\![0, 6]\!]$ and $0 \leqslant m \leqslant d$.

	m	0	1	2	3	4	5	6
d	Names	0-face	1-face	2-face	3-face	4-face	5-face	6-face
0	*Point*	1						
1	*Segment*	2	1					
2	*triangle*	3	3	1				
3	*tetrahedron*	4	6	4	1			
4	4-*simplex*	5	10	10	5	1		
5	5-*simplex*	6	15	20	15	6	1	
6	6-*simplex*	7	21	35	35	21	7	1

Table 2: Number of m-faces of a nondegenerate d-simplex

2 Tessellation with d-orthotopes

2.1 The unit hypercube vertices

The unit d-dimensional hypercube $\widehat{\mathrm{H}} = [0, 1]^d$ has $n = 2^d$ vertices. Each vertex can be identified by a d-tuple $\boldsymbol{\imath} = (\imath_1, \imath_2, \cdots, \imath_d) \in [\![0, 1]\!]^d$ and we denote by $\boldsymbol{x}^{\boldsymbol{\imath}} = (\boldsymbol{x}^{\boldsymbol{\imath}}_1, \ldots, \boldsymbol{x}^{\boldsymbol{\imath}}_d)^{\mathfrak{t}} \in \mathbb{R}^d$ the vertex defined by

$$\boldsymbol{x}^{\boldsymbol{\imath}}_l = \imath_l, \quad \forall l \in [\![1, d]\!].$$

Let $\mathcal{L}$ be the function that mapping all the d-tuples $\boldsymbol{\imath} \in [\![0, 1]\!]^d$ into $[\![1, 2^d]\!]$ defined by

$$\mathcal{L}(\boldsymbol{\imath}) = 1 + \sum_{l=1}^{d} 2^{l-1} \imath_l. \tag{3}$$

We can note that $\mathcal{L}(\boldsymbol{\imath}) - 1$ has for binary representation $(\imath_d \imath_{d-1} \cdots \imath_1)_2$. Let $\widehat{\mathbf{q}}$ be the d-by-2^d array containing all the vertices of $\widehat{\mathrm{H}}$ and defined by

$$\widehat{\mathbf{q}}(:, j) \stackrel{\text{def}}{=} \boldsymbol{x}^{\mathcal{L}^{-1}(j)}, \quad \forall j \in [\![1, 2^d]\!].$$

where $\widehat{\mathbf{q}}(:, j)$ denotes the j-th column of the array $\widehat{\mathbf{q}}$.

For example, with $d = 3$, the array $\widehat{\mathbf{q}}$ is given by

$$\widehat{\mathbf{q}} \stackrel{\text{def}}{=} \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

This array can be obtained from the more general function CartesianGrid-Points , introduced in section 2.2.1 and described in Appendix B, by using

$$\widehat{\mathbf{q}} \leftarrow \text{CartesianGridPoints}(\text{Ones}(1, d))$$

In Figure 2, the numbering of the vertices is represented in 2 and 3 dimensions.

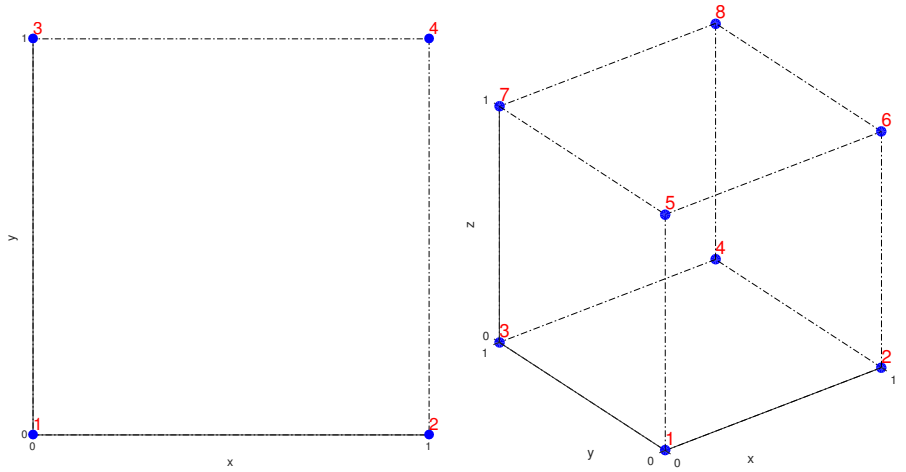


Figure 2: Vertices of the unit hypercube $[0,1]^d$, $d = 2$ (left) and $d = 3$ (right) with their indices in the array $\widehat{\mathbf{q}}$

2.2 Cartesian grid

Definition 5 *A cartesian grid in $\mathbb{R}^d$ is a tessellation where elements are unit d-hypercubes and vertices are integer lattices.*

Let $\mathbf{N} = (N_1, \ldots, N_d) \in (\mathbb{N}^*)^d$. We denote by $\mathcal{Q}_{\mathbf{N}}$ the cartesian grid of $[\![0, N_1]\!] \times \cdots \times [\![0, N_d]\!]$. The cartesian grid $\mathcal{Q}_{\mathbf{N}}$ is composed of n_{q} grid points and n_{me} unit d-hypercubes where

$$n_{\mathrm{q}} = \prod_{l=1}^{d} (N_l + 1) \quad \text{and} \quad n_{\mathrm{me}} = \prod_{l=1}^{d} N_l. \tag{4}$$

The objective of this section is to describe the construction of the vertices (or points) array $\mathbf{q}$ (section 2.2.1) and the connectivity array **me** associated with this cartesian grid (section 2.2.2). More precisely,

- $\mathbf{q}(\nu, j)$ is the ν-th coordinate of the j-th vertex, $\nu \in \{1, \ldots, d\}$, $j \in \{1, \ldots, n_{\mathrm{q}}\}$. The j-th vertex will be also denoted by $\mathbf{q}^j = \mathbf{q}(:, j)$.
- $\mathbf{me}(\beta, k)$ is the storage index of the β-th vertex of the k-th element (unit hypercube), in the array q, for $\beta \in \{1, ..., 2^d\}$ and $k \in \{1, \ldots, n_{\mathrm{me}}\}$. So $\mathbf{q}(:, \mathbf{me}(\beta, k))$ represents the coordinates of the β-th vertex in the k-th cartesian grid element.

We represent in Figure 3 two cartesian grids with the numbering of the n_{me} unit d-hypercubes. For example, on the left figure ($d = 2$), the 5-th unit hypercube is given by the vertices of numbers $6, 7, 10, 11$ and so $\mathbf{me}(:, 5) = (6, 7, 10, 11)$. On the right figure ($d = 3$), for the 9-th hypercube, we have $\mathbf{me}(:, 9) = (16, 17, 19, 20, 28, 29, 31, 32)$.

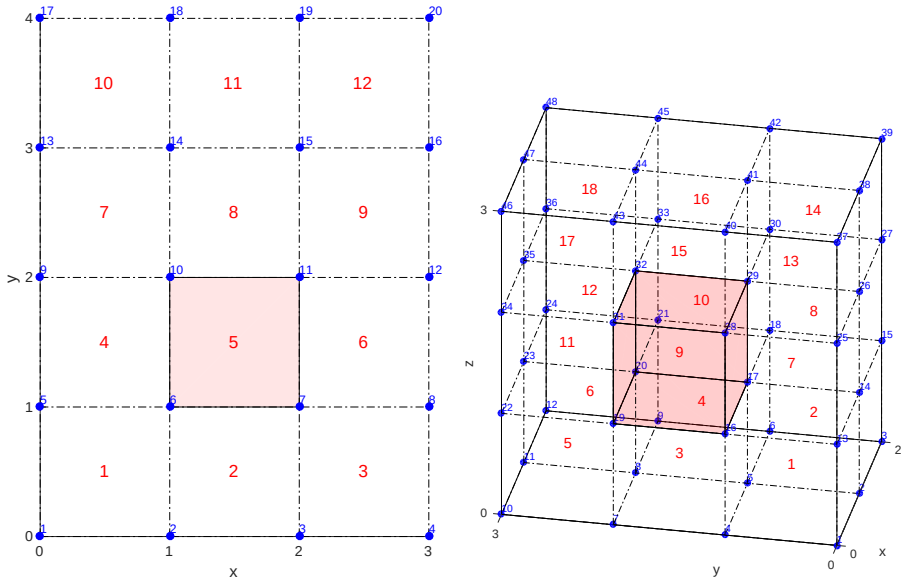


Figure 3: In blue, vertices of cartesian Grid in $\mathbb{R}^d$ with their indices in $\mathbf{q}$ array, $d = 2$ and $N_1 = 3$, $N_2 = 4$ (left), $d = 3$ and $(N_1, N_2, N_3) = (2, 3, 3)$ (right). The red numbers are the indices of unit hypercubes in the array $\mathbf{me}$.

2.2.1 Points of the cartesian grid

The grid points may be identified by a d-tuple $\boldsymbol{\imath} = (i_1, i_2, \cdots, i_d) \in [\![0, N_1]\!] \times \cdots \times [\![0, N_d]\!]$ and the corresponding grid point denoted by $\boldsymbol{x}^{\boldsymbol{\imath}}$ with integer coordinates is given by

$$\boldsymbol{x}^{\boldsymbol{\imath}} = \sum_{l=1}^{d} i_l \boldsymbol{e}^{[l]} = (i_1, i_2, \cdots, i_d)^{\mathtt{t}} = \boldsymbol{\imath} \tag{5}$$

where $\left\{\boldsymbol{e}^{[1]}, \ldots, \boldsymbol{e}^{[d]}\right\}$ denotes the standard basis of $\mathbb{R}^d$.

We want to store all the grid points in a 2D-array $\mathbf{q}$ of size d-by-n_{q}. To define an order of storage in the array $\mathbf{q}$, we will use the mapping function $\mathcal{G}$

$$\mathcal{G}(\boldsymbol{\imath}) = 1 + \sum_{l=1}^{d} i_l \beta_l = 1 + \langle \boldsymbol{\imath}, \boldsymbol{\beta} \rangle, \quad \forall \boldsymbol{\imath} \in [\![0, N_1]\!] \times \cdots \times [\![0, N_d]\!] \tag{6}$$

where $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_d) \in \mathbb{N}^d$ and

$$\beta_l = \prod_{j=1}^{l-1} (N_j + 1), \ \forall l \in [\![1, d]\!]. \tag{7}$$

The $\mathcal{G}$ function maps the tuple points set $[\![0, N_1]\!] \times \cdots \times [\![0, N_d]\!]$ to the global points index set $[\![1, n_{\mathrm{q}}]\!]$. From this function, we define the vertex array $\mathbf{q}$ as

$$\mathbf{q}(:, \mathcal{G}(\boldsymbol{\imath})) = \boldsymbol{x}^{\boldsymbol{\imath}} = \boldsymbol{\imath}, \quad \forall \boldsymbol{\imath} \in [\![0, N_1]\!] \times \cdots \times [\![0, N_d]\!] \tag{8}$$

According to the numbering choice $\mathcal{G}$, we give in Algorithm 1 the vectorized function CartesianGridPoints which returns the array $\mathbf{q}$. In Appendix B, we explain how this function was written

Algorithm 1 Function CartesianGridPoints : computes the d-by-n_{q} array $\mathbf{q}$ which contains all the points of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ (vectorized version)

Input :
$\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.

Output :
$\mathbf{q}$: array of d-by-n_{q} integers.

Function $\mathbf{q} \leftarrow$ CartesianGridPoints $(\boldsymbol{N})$
  $\boldsymbol{\beta} \leftarrow$ CGBeta$(\boldsymbol{N})$
  for $r \leftarrow 1$ **to** d **do**
    $\boldsymbol{A} \leftarrow$ Reshape(RepTile$([0 : \boldsymbol{N}(r)], \boldsymbol{\beta}(r), 1), 1, (\boldsymbol{N}(r) + 1)\boldsymbol{\beta}(r))$
    $\mathbf{q}(r, :) \leftarrow$ RepTile$(\boldsymbol{A}, 1,$ Prod$(\boldsymbol{N}(r + 1 : d) + 1))$
  end for
end Function

The function CGBeta used in previous algorithm computes the β_l, $\forall l \in [\![1, d]\!]$, by using (7). This function is given in Algorithm 2.

Algorithm 2 Function CGBeta : Computes β_l, $\forall l \in [\![1, d]\!]$, defined in (7)

Input :
$\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.

Output :
$\boldsymbol{\beta}$: array of d integers such that $\boldsymbol{\beta}(l) = \beta_l$ defined in (7)

Function $\boldsymbol{\beta} \leftarrow$ CGBeta $(\boldsymbol{N})$
  $\boldsymbol{\beta}(1) \leftarrow 1$
  for $l \leftarrow 2$ **to** d **do**
    $\boldsymbol{\beta}(l) \leftarrow \boldsymbol{\beta}(l - 1) \times (\boldsymbol{N}(l - 1) + 1)$
  end for
end Function

From the array $\mathbf{q}$ defined in (8), we can now construct the tessellation of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ with unit d-hypercubes.

2.2.2 Connectivity array of a cartesian grid

The d-dimensional cartesian grid $\mathcal{Q}_{\mathbf{N}}$ can be partitioned in n_{me} unit d-hypercubes which have as vertices the cartesian grid points. All these unit hypercubes can be uniquely identified by their vertex of minimal coordinates.

Let $\boldsymbol{\imath} \in [\![0, N_1[\![\times \cdots \times [\![0, N_d[\![$. We denote by $\mathrm{H}_{\boldsymbol{\imath}}$ the unit hypercube defined by its 2^d vertices

$$\boldsymbol{x}^{\boldsymbol{\imath}+\boldsymbol{p}}, \ \forall \boldsymbol{p} \in [\![0, 1]\!]^d.$$

We want to build the connectivity array **me** of dimensions 2^d-by-n_{me} such that $\mathbf{me}(l, k)$ is the index in array **q** of the l-th vertex of the k-th hypercube : this vertex is $\mathbf{q}(:, \mathbf{me}(l, k))$. To define an order of storage of the hypercubes in the array **me**, we will use the function $\mathcal{H}$ defined by

$$\mathcal{H}(\boldsymbol{\imath}) = 1 + \sum_{l=1}^{d} i_l \prod_{j=1}^{l-1} N_j, \quad \boldsymbol{\imath} \in [\![0, N_1[\![\times \cdots \times [\![0, N_d[\![\tag{9}$$

This bijective function maps the tuple points set $[\![0, N_1[\![\times \cdots \times [\![0, N_d[\![$ to the global points index set $[\![1, n_{\mathrm{me}}]\!]$ such that $k = \mathcal{H}(\boldsymbol{\imath})$.

The inverse function $\mathcal{H}^{-1}$ can easily be built. Indeed, if we define the d-by-n_{me} array **Hinv** by

$$\mathbf{Hinv} \leftarrow \textsc{CartesianGridPoints}(\boldsymbol{N} - 1).$$

then by construction we have

$$\mathcal{H}^{-1}(k) = \mathbf{Hinv}(:, k), \quad \forall k \in [\![1, n_{\mathrm{me}}]\!]$$

Let $k \in [\![1, n_{\mathrm{me}}]\!]$ and $\boldsymbol{\imath} = \mathcal{H}^{-1}(k)$. The k-th hypercube is $\mathrm{H}_{\boldsymbol{\imath}}$ and $\boldsymbol{x}^{\boldsymbol{\imath}}$ is its vertex of minimal coordinates. By construction of array **q** we have

$$\boldsymbol{x}^{\boldsymbol{\imath}} = \mathbf{q}(:, \mathcal{G}(\boldsymbol{\imath}))$$

From vector $\boldsymbol{\beta}$ defined in (7), we have $\mathcal{G}(\boldsymbol{\imath}) = 1 + \langle \boldsymbol{\imath}, \boldsymbol{\beta} \rangle$. Using matricial operations we can define the 1-by-n_{me} array **iBase** by

$$\mathbf{iBase} \leftarrow \boldsymbol{\beta}^{\mathrm{t}} * \mathbf{Hinv} + 1$$

such that

$$\mathcal{G}(\boldsymbol{\imath}) = \mathcal{G} \circ \mathcal{H}^{-1}(k) = \mathbf{iBase}(k). \tag{10}$$

Let $\boldsymbol{\imath} \in [\![0, N_1[\![\times \cdots \times [\![0, N_d[\![$ and $k = \mathcal{H}(\boldsymbol{\imath})$. We choose vertices local numbering in the k-th hypercube to be identical with that described ine section 2.1. That is why we take

$$\mathbf{q}(:, \mathbf{me}(l, k)) = \boldsymbol{x}^{\boldsymbol{\imath}} + \widehat{\mathbf{q}}(:, l) = \boldsymbol{x}^{\boldsymbol{\imath} + \widehat{\mathbf{q}}(:, l)}$$

where $\widehat{\mathbf{q}}$ is defined in section 2.1. So we obtain

$$\mathbf{me}(l, k) = \mathcal{G}(\boldsymbol{\imath} + \widehat{\mathbf{q}}(:, l)) \tag{11}$$

Lemma 6 *Let $\boldsymbol{\imath} \in \mathcal{Q}_{\mathbf{N}}$ and $\boldsymbol{p} \in \mathbb{Z}^d$, such that $\boldsymbol{\imath} + \boldsymbol{p} \in \mathcal{Q}_{\mathbf{N}}$. We have*

$$\mathcal{G}(\boldsymbol{\imath} + \boldsymbol{p}) = \mathcal{G}(\boldsymbol{\imath}) + \langle \boldsymbol{p}, \boldsymbol{\beta} \rangle \tag{12}$$

where $\boldsymbol{\beta}$ is defined in (7).

Proof: We have

$$\begin{aligned}\mathcal{G}(\boldsymbol{\imath}+\boldsymbol{p}) &\overset{\text{def}}{=} 1+\sum_{s=1}^{d}(i_s+p_l)\prod_{j=1}^{s-1}(N_j+1)\\ &= 1+\sum_{s=1}^{d} i_s\prod_{j=1}^{s-1}(N_j+1)+\sum_{s=1}^{d} p_s\prod_{j=1}^{s-1}(N_j+1)\\ &= \mathcal{G}(\boldsymbol{\imath})+\sum_{s=1}^{d} p_s\beta_s.\end{aligned}$$

□ From Lemma 6 and definition of $\boldsymbol{\beta}$ in (7), we obtain

$$\mathcal{G}(\boldsymbol{\imath}+\widehat{\mathbf{q}}(:,l)) = \mathcal{G}(\boldsymbol{\imath})+\sum_{s=1}^{d}\widehat{\mathbf{q}}(s,l)\beta_s = \mathcal{G}(\boldsymbol{\imath})+\langle\widehat{\mathbf{q}}(:,l),\boldsymbol{\beta}\rangle$$

From (11), we have

$$\mathbf{me}(l,k) \leftarrow \mathbf{iBase}(k)+\langle\widehat{\mathbf{q}}(:,l),\boldsymbol{\beta}\rangle, \quad \forall l\in[\![1,d]\!]$$

or in a vectorized form

$$\mathbf{me}(l,:) \leftarrow \mathbf{iBase}+\langle\widehat{\mathbf{q}}(:,l),\boldsymbol{\beta}\rangle$$

So we can easily write the function CGTessHyp in Algorithm 3 which computes the **q** and **me** arrays.

Algorithm 3 Function CGTessHyp : computes the vertices array **q** and the connectivity array **me** obtained from a tesselation of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ with unit hypercube.

Input :
N : array of d integers, $\boldsymbol{N}(i)=N_i$.

Output :
q : vertices array of d-by-n_{q} integers.
me : connectivity array of 2^d-by-N_h integers. $\mathbf{me}(l,k)$ is the index in array **q** of the l-th vertex of the k-th hypercube : this vertex is $\mathbf{q}(:,\mathbf{me}(l,k))$.

```
Function [q,me] ← CGTessHyp (N)
  q ← CartesianGridPoints(N)
  Hinv ← CartesianGridPoints(N − 1)
  q̂ ← CartesianGridPoints(Ones(1,d))
  β ← CGbeta(N)
  iBase ← β^t ∗ Hinv + 1
  for l ← 1 to 2^d do
    me(l,:) ← iBase + ⟨β, q̂(:,l)⟩
  end for
end Function
```

2.3 Numbering of the m-faces of the unit d-hypercube

Let $m \in [\![0, d]\!]$. As introduced in section 1, the m-faces of the unit d-hypercube $[0,1]^d$ are unit m-hypercubes in $\mathbb{R}^d$ defined by the product of d intervals where $d-m$ intervals are reduced to the singleton $\{0\}$ or $\{1\}$ (called reduced dimension)

We have $n_c = \binom{d}{m}$ possible choices to select the index of the $d-m$ reduced dimensions (combination of d elements taken $d-m$ at a time) and for each selected dimension 2 choices : $\{0\}$ or $\{1\}$.

So if $l \in [\![1, d]\!]$ is the index of a reduced dimension then vertices $\boldsymbol{x}^{\boldsymbol{i}} (= \boldsymbol{i} = (i_1, \ldots, i_d))$ is such that $i_l = 0$ (minimum value) or $i_l = 1$ (maximum value).

Let $\mathbb{L}^{[d,m]}$ be the n_c-by-$(d-m)$ array given by

$$\mathbb{L}^{[d,m]} \leftarrow \textsc{Combs}([\![1, d]\!], d-m).$$

Then each row of $\mathbb{L}^{[d,m]}$ contains the index of the $d-m$ reduced dimensions of an m-face sorted by lexicographical order (see Combs function description in Appendix A)

Let $\mathbb{S}^{[d-m]}$ be the $(d-m)$-by-2^{d-m} array given by

$$\mathbb{S}^{[d-m]} \leftarrow \textsc{CartesianGridPoints}(\textsc{Ones}(1, d-m)).$$

This array contains all the possible choices of the constants for the $d-m$ reduced dimensions (2 choices per dimension) : values are 0 for constant minimal value or 1 for maximal value.

Definition 7 *Let $l \in [\![1, n_c]\!]$, $r \in [\![1, 2^{d-m}]\!]$ and $k = 2^{d-m}(l-1) + r$. The k-th m-faces of the unit reference d-hypercube is defined by*

$$\left\{\boldsymbol{x} \in [0,1]^d \text{ such that } \boldsymbol{x}(\mathbb{L}^{[d,m]}(l, s)) = \mathbb{S}^{[d-m]}(s, r),\ \forall s \in [\![1, d-m]\!]\right\}$$

or in a vectorized form

$$\left\{\boldsymbol{x} \in [0,1]^d \text{ such that } \boldsymbol{x}(\mathbb{L}^{[d,m]}(l, :)) = \mathbb{S}^{[d-m]}(:, r)\right\} \tag{13}$$

For example, to obtain the ordered 2-faces of the unit 3-hypercube we compute

$$\mathbb{L}^{[3,2]} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad \mathbb{S}^{[1]} = \begin{pmatrix} 0 & 1 \end{pmatrix}$$

and then we have

2-face number	Set
1	$\{\boldsymbol{x} \in [0,1]^3$ such that $x_1 = 0\}$
2	$\{\boldsymbol{x} \in [0,1]^3$ such that $x_1 = 1\}$
3	$\{\boldsymbol{x} \in [0,1]^3$ such that $x_2 = 0\}$
4	$\{\boldsymbol{x} \in [0,1]^3$ such that $x_2 = 1\}$
5	$\{\boldsymbol{x} \in [0,1]^3$ such that $x_3 = 0\}$
6	$\{\boldsymbol{x} \in [0,1]^3$ such that $x_3 = 1\}$

To obtain the ordered 1-faces of the unit 3-hypercube we compute

$$\mathbb{L}^{[3,1]} = \begin{pmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 3 \end{pmatrix} \quad \text{and} \quad \mathbb{S}^{[2]} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

and then we have

1-face number	Set
1	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 0,\ x_2 = 0\}$
2	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 1,\ x_2 = 0\}$
3	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 0,\ x_2 = 1\}$
4	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 1,\ x_2 = 1\}$
5	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 0,\ x_3 = 0\}$
6	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 1,\ x_3 = 0\}$
7	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 0,\ x_3 = 1\}$
8	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 1,\ x_3 = 1\}$
9	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_2 = 0,\ x_3 = 0\}$
10	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_2 = 1,\ x_3 = 0\}$
11	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_2 = 0,\ x_3 = 1\}$
12	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_2 = 1,\ x_3 = 1\}$

To obtain the ordered 0-faces of the unit 3-hypercube we compute

$$\mathbb{L}^{[3,0]} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \quad \text{and} \quad \mathbb{S}^{[3]} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

and then we have

1-face number	Set
1	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 0,\ x_2 = 0,\ x_3 = 0\}$
2	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 1,\ x_2 = 0,\ x_3 = 0\}$
3	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 0,\ x_2 = 1,\ x_3 = 0\}$
4	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 1,\ x_2 = 1,\ x_3 = 0\}$
5	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 0,\ x_2 = 0,\ x_3 = 1\}$
6	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 1,\ x_2 = 0,\ x_3 = 1\}$
7	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 0,\ x_2 = 1,\ x_3 = 1\}$
8	$\{\boldsymbol{x} \in [0,1]^3 \text{ such that } x_1 = 1,\ x_2 = 1,\ x_3 = 1\}$

2.4 m-faces tessellations of a cartesian grid

Let $\mathcal{Q}_{\mathbf{N}}$ be the d-dimensional cartesian grid defined in section 2.2. So as not to confuse notations, we denote by $\mathcal{Q}_{\mathbf{N}}.\mathbf{q}$ and $\mathcal{Q}_{\mathbf{N}}.\mathbf{me}$ respectively the vertices and connectivity arrays of the tessellation with unit hypercubes of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$.

Let $m \in [\![0, d[\![$ and $k \in [\![1, E_{m,d}]\!]$. We want to determine $\mathcal{Q}^m_{\mathbf{N}}(k)$ the tessellation obtained from the restriction of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ to its k-th m-face where the numbering of the m-faces is specified in section 2.3. We denote by

- $\mathcal{Q}^m_{\mathbf{N}}(k).\mathbf{q}$, the (local) vertex array
- $\mathcal{Q}^m_{\mathbf{N}}(k).\mathbf{me}$, the (local) connectivity array
- $\mathcal{Q}^m_{\mathbf{N}}(k)$.toGlobal, the global indices such that

$$\mathcal{Q}^m_{\mathbf{N}}(k).\mathbf{q} \equiv \mathcal{Q}_{\mathbf{N}}.\mathbf{q}(:, \mathcal{Q}^m_{\mathbf{N}}(k).\text{toGlobal}).$$

By construction, $\mathcal{Q}^m_{\mathbf{N}}(k)$ is the tessellation of an m-hypercube in $\mathbb{R}^d$ with unit m-hypercubes .

Let $l \in [\![1, n_c]\!]$, $r \in [\![1, 2^{d-m}]\!]$ and $k = 2^{d-m}(l-1)+r$. The cartesian grid point $\boldsymbol{x} = (x_1, \ldots, x_d)$ is on the k-th m-face $\mathcal{Q}^m_{\mathbf{N}}(k)$ if and only if for all $s \in [\![1, d-m]\!]$ and with $j = \mathbb{L}^{[d,m]}(l, s)$ we have

$$x_j = \begin{cases} 0 & \text{if } \mathbb{S}^{[d-m]}(s, r) == 0, \quad \text{(minimum value)} \\ N_j & \text{otherwise } (\mathbb{S}^{[d-m]}(s, r) == 1), \quad \text{(maximum value)} \end{cases}$$

So we obtain

$$x_j = N_j \times \mathbb{S}^{[d-m]}(s, r)$$

or, in a vectorized form using element-wise multiplication operator **.***:

$$\boldsymbol{x}(\mathbb{L}^{[d,m]}(l, :)) = \boldsymbol{N}(\mathbb{L}^{[d,m]}(l, :)) \mathbin{.*} \mathbb{S}^{[d-m]}(:, r). \tag{14}$$

Definition 8 *Let $l \in [\![1, n_c]\!]$, $r \in [\![1, 2^{d-m}]\!]$ and $k = 2^{d-m}(l-1) + r$. Then, the k-th m-faces of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ is defined as the set*

$$\left\{ \boldsymbol{x} \in \mathcal{Q}_{\mathbf{N}} \text{ such that } \boldsymbol{x}(\mathbb{L}^{[d,m]}(l, :)) = \boldsymbol{N}(\mathbb{L}^{[d,m]}(l, :)) \mathbin{.*} \mathbb{S}^{[d-m]}(:, r) \right\} \tag{15}$$

2.4.1 Case $m = 0$.

If $m = 0$, the m-faces are the 2^d corner points of the cartesian grid. Then we have $\mathbb{L}^{[d,0]} = [\![1, d]\!]$ and $\mathbb{S}^{[d]}$ is an d-by-2^d array.

From (15), we obtain that $\forall k \in [\![1, 2^d]\!]$ the k-th 0-face of $\mathcal{Q}_{\mathbf{N}}$ is reduced to the point

$$\boldsymbol{x} = \boldsymbol{N} \mathbin{.*} \mathbb{S}^{[d]}(:, k)^{\mathsf{t}}$$

and it is also the k-th column of the array Q of dimensions d-by-2^d given by

$$Q \leftarrow \begin{pmatrix} N_1 & 0 & \ldots & \ldots & 0 \\ 0 & N_2 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \ldots & \ldots & 0 & N_d \end{pmatrix} \mathbb{S}^{[d]}$$

So we obtain

$$\begin{aligned} \mathcal{Q}^0_{\mathbf{N}}(k).\mathbf{q} &= Q(:, k) \\ \mathcal{Q}^0_{\mathbf{N}}(k).\mathbf{me} &= 1 \\ \mathcal{Q}^0_{\mathbf{N}}(k).\text{toGlobal} &= \langle \boldsymbol{\beta}, Q(:, k) \rangle + 1 \end{aligned}$$

2.4.2 Case $m > 0$

Let $l \in [\![1, n_c]\!]$, $r \in [\![1, 2^{d-m}]\!]$ and $k = 2^{d-m}(l-1) + r$. To construct $\mathcal{Q}^m_{\mathbf{N}}(k)$ we first set a tessellation without the m constant dimensions given in $\mathbf{idc} = \mathbb{L}(l, :)$ (i.e. only with nonconstant dimensions in $\mathbf{idnc} = [\![1, d]\!] \backslash \mathbf{idc}$):

$$[\mathbf{q}^w, \mathbf{me}^w] \leftarrow \text{CGTessHyp}(\boldsymbol{N}(\mathbf{idnc}))$$

The dimension of the array $\mathbf{q}^w$ is m-by-n_{q}^l where $n_{\mathrm{q}}^l = \prod_{i \in \mathbf{idnc}} (N_i + 1)$. Then the nonconstant rows are

$$\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}(\mathbf{idnc}(i), :) \leftarrow \mathbf{q}^w(i, :), \quad \forall i \in [\![1, m]\!]$$

and the constants rows

$$\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}(\mathbf{idc}(i), :) \leftarrow \boldsymbol{N}(\mathbf{idc}(i)) * \mathbb{S}^{[d-m]}(i, r) * \textsc{Ones}(1, n_{\mathrm{q}}^l), \quad \forall i \in [\![1, d-m]\!]$$

In a vectorized way, we have

$$\begin{aligned} \mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}(\mathbf{idnc}, :) &\leftarrow \mathbf{q}^w \\ \mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}(\mathbf{idc}, :) &\leftarrow \left(\boldsymbol{N}(\mathbf{idc})^{\mathbf{t}} \mathbin{.*} \mathbb{S}^{[d-m]}(:, r)\right) * \textsc{Ones}(1, n_{\mathrm{q}}^l) \end{aligned}$$

We immediately have the connectivity array

$$\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{me} = \mathbf{me}^w.$$

There still remains to compute $\mathcal{Q}_{\mathbf{N}}^m(k)$.toGlobal. For that we use the mapping function $\mathcal{G}$ defined in section 2.2.1 and more particularly (8). Indeed, for all $j \in [\![1, n_{\mathrm{q}}^l]\!]$, we can identify the point $\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}(:, j)$ by the d-tuple $\boldsymbol{\imath}$ and use it with the mapping function $\mathcal{G}$ to obtain the index in array $\mathcal{Q}_{\mathbf{N}}.\mathbf{q}$ of the point $\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}(:, j)$. So we have

$$\boldsymbol{\imath} = \mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}(:, j) = \mathcal{Q}_{\mathbf{N}}.\mathbf{q}(:, \mathcal{G}(\boldsymbol{\imath}))$$

and then

$$\mathcal{Q}_{\mathbf{N}}^m(k).\mathrm{toGlobal}(j) = \mathcal{G}(\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}(:, j)).$$

In a vectorized way, we set

$$\mathcal{Q}_{\mathbf{N}}^m(k).\mathrm{toGlobal} \leftarrow 1 + \boldsymbol{\beta}^{\mathbf{t}} * \mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}$$

with the vector $\boldsymbol{\beta}$ defined in (7).

One can also compute the connectivity array of $\mathcal{Q}_{\mathbf{N}}^m(k)$ associated with global vertices array $\mathcal{Q}_{\mathbf{N}}.\mathbf{q}$ which is given by $\mathcal{Q}_{\mathbf{N}}^m(k).\mathrm{toGlobal}(\mathbf{me}^w)$.

We give in Algorithm 4 the function CGTessFaces which computes $\mathcal{Q}_{\mathbf{N}}^m(k)$, $\forall k \in [\![1, 2^{d-m} n_c]\!]$.

Algorithm 4 Function CGTessFaces : computes all m-faces tessellations of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ with unit m-hypercubes.

Input :
$\boldsymbol{N}$: 1-by-d array of integers, $\boldsymbol{N}(i) = N_i$.
m : integer, $0 \leqslant m < d$

Output :
$\mathcal{Q}_{\mathbf{N}}^m$: array of tessellations of all m-faces of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$.
Its length is $E_{m,d} = 2^{d-m}\binom{d}{m}$.

Function $s\mathcal{Q}_{\mathbf{N}} \leftarrow$ CGTessFaces $(\boldsymbol{N}, m)$
  $\boldsymbol{\beta} \leftarrow$ CGBeta$(\boldsymbol{N})$
  if $m == 0$ **then**
    $\mathbb{Q} \leftarrow$ Diag$(\boldsymbol{N}) *$ CartesianGridPoints(Ones$(1, d)$)
    for $k \leftarrow 1$ **to** 2^d **do**
      $\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q} \leftarrow \mathbb{Q}(:, k)$
      $\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{me} \leftarrow 1$
      $\mathcal{Q}_{\mathbf{N}}^m(k)$.toGlobal $\leftarrow 1 + \langle \boldsymbol{\beta}, \mathbb{Q}(:, k) \rangle$
    end for
  else
    $n_c \leftarrow \binom{d}{m}$
    $\mathbb{L} \leftarrow$ Combs$([\![1, d]\!], d - m)$
    $\mathbb{S} \leftarrow$ CartesianGridPoints(Ones$(1, d - m)$)
    $k \leftarrow 1$
    for $l \leftarrow 1$ **to** n_c **do**
      idc $\leftarrow \mathbb{L}(l, :)$
      idnc $\leftarrow [\![1, d]\!] \backslash$ **idc**
      $[\mathbf{q}^w, \mathbf{me}^w] \leftarrow$ CGTessHyp$(\boldsymbol{N}($**idnc**$))$
      $n_{\mathrm{q}}^l \leftarrow \prod_{s=1}^{m} (\boldsymbol{N}($**idnc**$(s)) + 1)$ ▷ or length of $\mathbf{q}^w$
      for $r \leftarrow 1$ **to** 2^{d-m} **do**
        $\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}($**idnc**$, :) \leftarrow \mathbf{q}^w$
        $\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}($**idc**$, :) \leftarrow \left(\boldsymbol{N}(\mathbf{idc})^{\mathbf{t}} \mathbin{.*} \mathbb{S}(:, r)\right) *$ Ones$(1, n_{\mathrm{q}}^l)$
        $\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{me} \leftarrow \mathbf{me}^w$
        $\mathcal{Q}_{\mathbf{N}}^m(k)$.toGlobal $\leftarrow 1 + \boldsymbol{\beta}^{\mathbf{t}} * \mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}$
        $k \leftarrow k + 1$
      end for
    end for
  end if
end Function

2.5 Tessellation of a d-orthotope with d-orthotopes

Let $\mathcal{O}_d$ be the d-orthotope $[a_1, b_1] \times \cdots \times [a_d, b_d]$. To construct a regular grid on $\mathcal{O}_d$ with $N_i + 1$ points in $\boldsymbol{e}^{[i]}$ direction, $i \in [\![1, d]\!]$, we use an affine transformation of the cartesian grid $\mathcal{Q}_{\mathbf{N}} = [\![0, N_1]\!] \times \cdots \times [\![0, N_d]\!]$ to $\mathcal{O}_d$. Let $\boldsymbol{a} = (a_1, \ldots, a_d)$, $\boldsymbol{b} = (b_1, \ldots, b_d)$ and $\boldsymbol{h} = (h_1, \ldots, h_d)$ with $h_i = (b_i - a_i)/N_i$ be three vectors of $\mathbb{R}^d$. Let $\mathbb{H} \in \mathcal{M}_d(\mathbb{R})$ be the diagonal matrix with $\boldsymbol{h}$ as diagonal. Then the affine

transformation is given by

$$\begin{array}{rccc} \mathcal{A} & : & \mathcal{Q}_{\mathbf{N}} & \longrightarrow & \mathcal{O}_d \\ & & \boldsymbol{x} & \longmapsto & \boldsymbol{y} = \boldsymbol{a} + \mathbb{H}\boldsymbol{x} \end{array}$$

Let $\boldsymbol{N} \leftarrow [N_1, \ldots, N_d]$. The tessellation of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ is obtained by

$$[\mathbf{q}, \mathbf{me}] \leftarrow \text{CGTessHyp}(\boldsymbol{N})$$

To obtain the tessellation of the orthotope $\mathcal{O}_d$ we only have to apply the affine transformation $\mathcal{A}$ to array $\mathbf{q}$. In a vectorized form, one can write for all $i \in [\![1, d]\!]$

$$\mathbf{q}(i,:) \leftarrow \boldsymbol{a}(i) + (\boldsymbol{b}(i) - \boldsymbol{a}(i))/\boldsymbol{N}(i) * \mathbf{q}(i,:)$$

This operation is done by the function boxMapping given in Algorithm 5.

Algorithm 5 Function boxMapping : mapping points of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ to the d-orthotope $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Input :
- $\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.
- $\mathbf{q}$: d-by-n_{q} array of integer obtained from $[\mathbf{q}, \mathbf{me}] \leftarrow \text{CGTessHyp}(\boldsymbol{N})$
- $\boldsymbol{a}$, $\boldsymbol{b}$: arrays of d reals, $\boldsymbol{a}(i) = a_i$, $\boldsymbol{b}(i) = b_i$ with $a_i < b_i$

Output :
- $\mathbf{q}$: vertices array of d-by-n_{q} reals.

```
Function q ← boxMapping (q, a, b, N)
  for i ← 1 to d do
    h ← (b(i) − a(i))/N(i)
    q(i,:) ← a(i) + h ∗ q(i,:)
  end for
end Function
```

The function OrthTessOrth , which returns the arrays $\mathbf{q}$ and $\mathbf{me}$ corresponding to the regular tessellation of $\mathcal{O}_d$ with d-orthotopes, is presented in Algorithm 6.

Algorithm 6 Function OrthTessOrth : d-orthotope regular tessellation with orthotopes

Input :
- $\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.
- $\boldsymbol{a}$, $\boldsymbol{b}$: arrays of d reals, $\boldsymbol{a}(i) = a_i$, $\boldsymbol{b}(i) = b_i$ with $a_i < b_i$

Output :
- $\mathbf{q}$: array of d-by-n_{q} reals.
- $\mathbf{me}$: array of 2^d-by-n_{me} integers.

```
Function [q, me] ← OrthTessOrth (N, a, b)
  [q, me] ← CGTessHyp(N)
  q ← boxMapping(q, a, b)
end Function
```

2.6 m-faces tessellations of a d-orthotope

As seen in section 2.5, we only have to apply the function BOXMAPPING to each array $\mathcal{Q}_{\mathbf{N}}^m(k).\mathbf{q}$ of the tessellations of the m-faces of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$. This is the object of the function ORTHTESSFACES given in Algorithm 7.

Algorithm 7 Function ORTHTESSFACES : computes the conforming tessellations of all the m-faces of the d-orthotope $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Input :
- $\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.
- $\boldsymbol{a}$, $\boldsymbol{b}$: arrays of d reals, $\boldsymbol{a}(i) = a_i$, $\boldsymbol{b}(i) = b_i$
- m : integer, $0 \leqslant m < d$

Output :
- $s\mathcal{O}_{\boldsymbol{h}}$: array of the tessellations of each m-faces of the orthotope. Its length is $E_{m,d} = 2^{d-m}\binom{d}{m}$.

Function $s\mathcal{O}_{\boldsymbol{h}} \leftarrow$ ORTHTESSFACES $(\boldsymbol{N}, \boldsymbol{a}, \boldsymbol{b}, m)$
  $s\mathcal{O}_{\boldsymbol{h}} \leftarrow$ CGTESSFACES$(\boldsymbol{N}, m)$
  for $k \leftarrow 1$ **to** LEN$(s\mathcal{O}_{\boldsymbol{h}})$ **do**
    $s\mathcal{O}_{\boldsymbol{h}}(k).\mathbf{q} \leftarrow$ BOXMAPPING$(s\mathcal{O}_{\boldsymbol{h}}(k).\mathbf{q}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{N})$
  end for
end Function

3 Tessellation with d-simplices

The goal of this section is to obtain a *conforming* triangulation or tessellation of a d-orthotope named $\mathcal{O}_d$ with d-simplices.

The basic principle selected here is to start from a tesselation of a cartesian grid with unit hypercubes as obtained in section 2.2. Then by using the Kuhn's decomposition of an hypercube in simplices, we build in section 3.2 a tesselation of a cartesian grid with simplices and we explain how to obtain all its m-faces in section 3.3. Finally, ...

3.1 Kuhn's decomposition of a d-hypercube

Kuhn's subdivision (see [1, 11, 12]) is a good way to divide a d-hypercube into d-simplices ($d \geqslant 2$). We recall that a d-simplex is made of $(d+1)$ vertices.

Definition 9 *Let* $\mathrm{H} = [0,1]^d$ *be the unit* d*-hypercube in* $\mathbb{R}^d$*. Let* $\boldsymbol{e}^{[1]}, \ldots, \boldsymbol{e}^{[d]}$ *be the standard unit basis vectors of* $\mathbb{R}^d$ *and denote by* S_d *the permutation group of* $[\![1, d]\!]$*. For all* $\pi \in S_d$*, the simplex* K_π *has for vertices* $\{\boldsymbol{x}_\pi^{[0]}, \ldots, \boldsymbol{x}_\pi^{[d]}\}$ *defined by*

$$\boldsymbol{x}_\pi^{[0]} = (0, \ldots, 0)^t, \quad \boldsymbol{x}_\pi^{[j]} = \boldsymbol{x}_\pi^{[j-1]} + \boldsymbol{e}^{[\pi(j)]}, \ \forall j \in [\![1, d]\!]. \tag{16}$$

The set $\mathcal{K}(\mathrm{H})$ *defined by*

$$\mathcal{K}(\mathrm{H}) = \{K_\pi \mid \pi \in S_d\} \tag{17}$$

is called the ***Kuhn's subdivision*** *of* H *and its cardinality is* $d!$.

For example, we give in Figure 4 the Kuhn'subdivision of an d-hypercube with $d = 2$ and $d = 3$. We choose the **positive orientation** for all the d simplices. The corresponding vertex array **q** and the connectivity array **me** are given by (préciser comment **me** est ordonné):

- for $d = 2$,

$$\mathbf{q} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \ \mathbf{me} = \begin{pmatrix} 4 & 1 \\ 3 & 2 \\ 1 & 4 \end{pmatrix}$$

- for $d = 3$,

$$\mathbf{q} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}, \ \mathbf{me} = \begin{pmatrix} 1 & 8 & 8 & 1 & 1 & 8 \\ 5 & 3 & 5 & 3 & 2 & 2 \\ 7 & 7 & 6 & 4 & 6 & 4 \\ 8 & 1 & 1 & 8 & 8 & 1 \end{pmatrix}$$

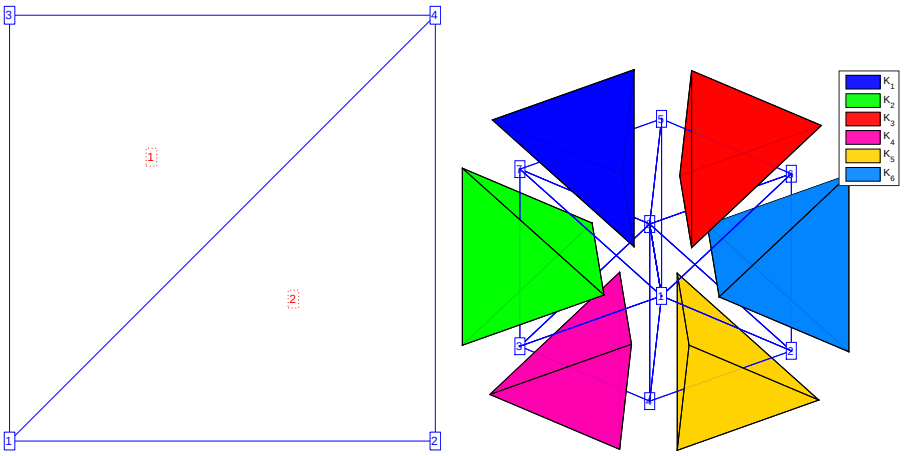


Figure 4: Kuhn's subdivision

Let K_{ref} be the *base simplex* or *reference simplex* with vertices denoted by $\{\boldsymbol{x}^{[0]}, \ldots, \boldsymbol{x}^{[d]}\}$ and such that

$$\boldsymbol{x}^{[0]} = (0, \ldots, 0)^t, \quad \boldsymbol{x}^{[j]} = \boldsymbol{x}^{[j-1]} + \boldsymbol{e}^{[j]}, \ \forall j \in [\![1, d]\!]. \tag{18}$$

Let $\pi \in S_n$ and $\pi(\boldsymbol{x})$ indicate the application of permutation π to the coordinates of vertex $\boldsymbol{x}$. The vertices of the simplex K_π defined in (16) can be derived from the reference simplex K_{ref} by

$$\boldsymbol{x}_\pi^{[j]} = \pi(\boldsymbol{x}^{[j]}), \quad \forall j \in [\![0, d]\!]. \tag{19}$$

Let $\pi(K_{\text{ref}})$ denote the application of permutation to each vertex of K_{ref}. Then we have

$$\pi(K_{\text{ref}}) = K_\pi \tag{20}$$

Lemma 10 ([1], Lemma 4.1) *The **Kuhn's subdivision** $\mathcal{K}(\mathrm{H})$ of the unit d-hypercube* H *has the following properties:*

1. 0^d *and* 1^d *are common vertices of all elements* $K_\pi \in \mathcal{K}(\mathrm{H})$.

2. $\mathcal{K}(\mathrm{H})$ *is a consistent/conforming triangulation of* H.

3. $\mathcal{K}(\mathrm{H})$ *is compatible with translation, i.e., for each vector* $\boldsymbol{v} \in [\![0,1]\!]^d$ *the union of* $\mathcal{K}(\mathrm{H})$ *and* $\mathcal{K}(\boldsymbol{v}+\mathrm{H})$ *is a consistent/conforming triangulation of the set* $\mathrm{H} \cup (\boldsymbol{v}+\mathrm{H})$.

4. *For any affine transformation* $\mathcal{F}$*, the Kuhn's triangulation of* $\mathcal{F}(\mathrm{H})$ *is defined by* $\mathcal{K}(\mathcal{F}(\mathrm{H})) \stackrel{\text{def}}{=} \mathcal{F}(\mathcal{K}(\mathrm{H}))$.

To explicitly obtain a Kuhn's triangulation $\mathcal{K}(\mathrm{H})$ of the unit d-hypercube H we must build the connectivity array, denoted by **me**, associated with the vertex array **q**. The dimension of the array **me** is $(d+1)$-by-$d!$.

Let $\mathbf{q}^{\text{ref}}$ be the d-by-$(d+1)$ array of vertex coordinates of reference d-simplex K^{ref} :

$$\mathbf{q}^{\text{ref}} = \left(\begin{array}{c|c|c|c|c} \boldsymbol{x}^{[0]} & \boldsymbol{x}^{[1]} & \dots & \dots & \boldsymbol{x}^{[d]} \end{array} \right) = \left(\begin{array}{c|cccc} 0 & 1 & \dots & \dots & 1 \\ \vdots & 0 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 1 \end{array} \right)$$

Let $\mathbf{P}$ be the d-by-$d!$ array of all permutations of the set $[\![1,d]\!]$ and $\pi = \mathbf{P}(:,k)$ the k-th permutation. The array $\mathbf{P}$ is obtained by using the function PERMS defined in Appendix A.2. We use (19) and (20) to build the vertices of K_π. So the j-th vertex of K_π is given by

$$\boldsymbol{x}_\pi^{[j-1]} \leftarrow \mathbf{q}^{\text{ref}}(\mathbf{P}(:,k),j)$$

To find which column in array **q** corresponds to $\boldsymbol{x}_\pi^{[j-1]}$ we use the mapping function $\mathcal{L}$ defined in (3) and we set

$$\mathbf{me}(j,k) \leftarrow \mathcal{L}(\mathbf{q}^{\text{ref}}(P(:,k),j)) = \left\langle \begin{pmatrix} 2^0 \\ \vdots \\ 2^{d-1} \end{pmatrix}, \mathbf{q}^{\text{ref}}(\mathbf{P}(:,k),j) \right\rangle + 1$$

If the k-th d-simplex has a negative orientation, one can permute the index of the first and the last points to obtain a positive orientation:

$$\mathbf{me}(1,k) \leftrightarrow \mathbf{me}(d+1,k).$$

In Algorithm 8, we give the function KUHNTRIANGULATION which returns the points array **q** and the connectivity array **me** where all the d-simplices have a positive orientation.

Algorithm 8 Kuhn's triangulation of the unit d-hypercube $[0,1]^d$ with $d!$ simplices (positive orientation)

Input :
d : space dimension

Output :
q : vertices array of d-by-2^d integers.
me : connectivity array of $(d+1)$-by-$d!$ integers

Function $[\mathbf{q}, \mathbf{me}] \leftarrow$ KuhnTriangulation (d)
  $\mathbf{q} \leftarrow$ CartesianGridPoints(Ones$(1,d)$)
  $\mathbf{q}^{\text{ref}} \leftarrow \begin{pmatrix} 0 & 1 & \dots & \dots & 1 \\ \vdots & 0 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}$ ▷ a d-by-$(d+1)$ array
  $\mathbf{P} \leftarrow$ perms$(1:d)$ ▷ see Appendix A.2
  $\mathbf{me} \leftarrow \mathbb{0}_{d+1,d!}$
  $\boldsymbol{a} \leftarrow [2^0, 2^1, \dots, 2^{d-2}, 2^{d-1}]$
  for $k \leftarrow 1$ **to** $d!$ **do**
    for $j \leftarrow 1$ **to** $d+1$ **do**
      $\mathbf{me}(j,k) \leftarrow$ dot$(\boldsymbol{a}, \mathbf{q}^{\text{ref}}(\mathbf{P}(:,k),j)) + 1$
    end for
    if det$([q(:, me(:,k)); \text{ones}(1,d+1)]) < 0$ **then**
      $t \leftarrow \mathbf{me}(1,k)$, $\mathbf{me}(1,k) \leftarrow \mathbf{me}(d+1,k)$, $\mathbf{me}(d+1,k) \leftarrow t$
    end if
  end for
end Function

From this tesselation of the unit reference d-hypercube, we will see how to get a regular tessellation of a cartesian grid with simplices.

3.2 Cartesian grid tesselation with simplices

Let $\mathcal{Q}_{\mathbf{N}}$ be the d-dimensional cartesian grid defined in section 2.2. As before, so as not to confuse notations, we denote by $\mathcal{Q}_{\mathbf{N}}.\mathbf{q}$ and $\mathcal{Q}_{\mathbf{N}}.\mathbf{me}$ respectively the vertices and connectivity arrays of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$. There are $N_h = \prod_{i=1}^{d} N_i$ unit hypercubes in this tessellation.

Let $\mathcal{I} = [\![0, N_1[\![\times \dots \times [\![0, N_d[\![$. We have

$$\mathcal{Q}_{\mathbf{N}} = \bigcup_{\boldsymbol{\imath} \in \mathcal{I}} \mathrm{H}_{\boldsymbol{\imath}}$$

where $\mathrm{H}_{\boldsymbol{\imath}}$ is the unit hypercube with $\boldsymbol{x}^{\boldsymbol{\imath}} = \boldsymbol{\imath}$ vertex of minimal coordinates.

From Lemma 10, the triangulation

$$\mathcal{T}_{\mathbf{N}} = \bigcup_{\boldsymbol{\imath} \in \mathcal{I}} \mathcal{K}(\mathrm{H}_{\boldsymbol{\imath}})$$

is a conforming triangulation of $\mathcal{Q}_{\mathbf{N}}$ with $n_{\text{me}} = d! \times N_h$ d-simplices and by construction the vertices of $\mathcal{T}_{\mathbf{N}}$ are the vertices of $\mathcal{Q}_{\mathbf{N}}$:

$$\mathcal{T}_{\mathbf{N}}.\mathbf{q} = \mathcal{Q}_{\mathbf{N}}.\mathbf{q}.$$

It thus remains to calculate the connectivity array $\mathbf{me}$ of $\mathcal{T}_{\mathbf{N}}$ also denoted by $\mathcal{T}_{\mathbf{N}}.\mathbf{me}$. This is a $(d+1)$-by-n_{me} array. For a given hypercube $\mathrm{H}_{\boldsymbol{\imath}}$ we store consecutively in the array $\mathbf{me}$, the $d!$ simplices given by $\mathcal{K}(\mathrm{H}_{\boldsymbol{\imath}})$

The Kuhn's triangulation for the reference hypercube $[0,1]^d$ can be obtained from the function KuhnTriangulation :

$$[\mathbf{q}_{\mathrm{K}}, \mathbf{me}_{\mathrm{K}}] \leftarrow \textsc{KuhnTriangulation}(d)$$

Let $\boldsymbol{\imath} \in \mathcal{I}$ and $k = \mathcal{H}(\boldsymbol{\imath})$ where $\mathcal{H}$ is defined by (9). Let $l \in [\![1, d!]\!]$. We choose to store the l-th simplex of $\mathcal{K}(\mathrm{H}_{\boldsymbol{\imath}})$ in $\mathbf{me}(:, d!(k-1)+l)$.

Let $j \in [\![1, d+1]\!]$. The j-th vertex of the l-th simplex of $\mathcal{K}(\mathrm{H}_{\boldsymbol{\imath}})$ is stored in $\mathbf{q}(:, \mathbf{me}(j, d!(k-1)+l))$ and its coordinates are given by

$$\boldsymbol{x}^{\boldsymbol{\imath}} + \mathbf{q}_{\mathrm{K}}(:, \mathbf{me}_{\mathrm{K}}(j,l)) = \boldsymbol{\imath} + \mathbf{q}_{\mathrm{K}}(:, \mathbf{me}_{\mathrm{K}}(j,l))$$

So we want to determine the index $\mathbf{me}(j, d!(k-1)+l)$. From (8), we obtain

$$\mathbf{me}(j, d!(k-1)+l) = \mathcal{G}(\boldsymbol{\imath} + \mathbf{q}_{\mathrm{K}}(:, \mathbf{me}_{\mathrm{K}}(j,l))).$$

By using Lemma 6, we deduce that

$$\mathbf{me}(j, d!(k-1)+l) = \mathcal{G}(\boldsymbol{\imath}) + \langle \mathbf{q}_{\mathrm{K}}(:, \mathbf{me}_{\mathrm{K}}(j,l)), \boldsymbol{\beta} \rangle$$

Then, with (10), the array $\mathbf{me}$ is given by: $\forall l \in [\![1, d!]\!]$, $\forall j \in [\![1, d+1]\!]$, $\forall k \in [\![1, N_h]\!]$,

$$\mathbf{me}(j, d!(k-1)+l) = \mathbf{iBase}(k) + \langle \mathbf{q}_{\mathrm{K}}(:, \mathbf{me}_{\mathrm{K}}(j,l)), \boldsymbol{\beta} \rangle .$$

This formula can be vectorized in k: with $\mathbf{Idx} \leftarrow d![0 : N_h - 1] + l$ then

$$\mathbf{me}(j, \mathbf{Idx}) \leftarrow \mathbf{iBase} + \langle \mathbf{q}_{\mathrm{K}}(:, \mathbf{me}_{\mathrm{K}}(j,l)), \boldsymbol{\beta} \rangle .$$

We give in Algorithm 9 the function CGTriangulation which computes the triangulation of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$.

Algorithm 9 Function CGTriangulation : computes the triangulation of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$

Input :
$\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.

Output :
q : vertices array of the triangulation of $\mathcal{Q}_{\mathbf{N}}$.
d-by-n_{q} array of reals (integer in fact) where $n_{\mathrm{q}} = \prod_{i=1}^{d}(N_i+1)$.
me : connectivity array of the triangulation of $\mathcal{Q}_{\mathbf{N}}$.
$(d+1)$-by-n_{me} array of integers where $n_{\mathrm{me}} = d!\prod_{i=1}^{d} N_i$. .

Function $[\mathbf{q}, \mathbf{me}] \leftarrow$ CGTriangulation $(\boldsymbol{N})$
  $\mathbf{q} \leftarrow$ CartesianGridPoints$(\boldsymbol{N})$
  Hinv $\leftarrow$ CartesianGridPoints$(\boldsymbol{N}-1)$
  $[\mathbf{q}_{\kappa}, \mathbf{me}_{\kappa}] \leftarrow$ KuhnTriangulation(d)
  $\boldsymbol{\beta} \leftarrow$ CGBeta$(\boldsymbol{N})$
  iBase $\leftarrow \boldsymbol{\beta}^{\mathrm{t}} *$ **Hinv** $+ 1$
  Idx $\leftarrow d! * [0 : (N_h - 1)]$
  for $l \leftarrow 1$ **to** $d!$ **do**
    Idx $\leftarrow$ **Idx** $+ 1$
    for $j \leftarrow 1$ **to** $d+1$ **do**
      $\mathbf{me}(j,$**Idx**$) \leftarrow$ **iBase** $+ \langle \mathbf{q}_{\kappa}(:, \mathbf{me}_{\kappa}(j,l)), \boldsymbol{\beta} \rangle$
    end for
  end for
end Function

3.3 m-faces tessellations of a cartesian grid

Let $\mathcal{Q}_{\mathbf{N}}$ be the d-dimensional cartesian grid defined in section 2.2. As before, we denote by $\mathcal{T}_{\boldsymbol{N}}.\mathbf{q}$ and $\mathcal{T}_{\boldsymbol{N}}.\mathbf{me}$ respectively the vertices and connectivity arrays of the tessellation of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ with d-simplices obtained from CGTriangulation function and described in Algorithm 9.

Let $m \in [\![0, d[\![$ and $k \in [\![1, E_{m,d}]\!]$ where $E_{m,d}$ is the number of m-faces defined in (1). We want to determine $\mathcal{T}^m_{\boldsymbol{N}}(k)$, the tessellation obtained from the restriction of $\mathcal{T}_{\boldsymbol{N}}$ to its k-th m-face where the numbering of the m-faces is specified in section 2.3. We denote by

- $\mathcal{T}^m_{\boldsymbol{N}}(k).\mathbf{q}$, the (local) vertex array
- $\mathcal{T}^m_{\boldsymbol{N}}(k).\mathbf{me}$, the (local) connectivity array
- $\mathcal{T}^m_{\boldsymbol{N}}(k)$.toGlobal, the global indices such that

$$\mathcal{T}^m_{\boldsymbol{N}}(k).\mathbf{q} \equiv \mathcal{T}_{\boldsymbol{N}}.\mathbf{q}(:, \mathcal{T}^m_{\boldsymbol{N}}(k).\text{toGlobal}).$$

By construction, $\mathcal{T}^m_{\boldsymbol{N}}(k)$ is the triangulation by m-simplices of an m-hypercube in $\mathbb{R}^d$.

The only difference with the construction of $\mathcal{Q}^m_{\mathbf{N}}(k)$ given in section 2.4 is on the $\mathbf{me}^w$ array. For $\mathcal{Q}^m_{\mathbf{N}}(k)$, we had

$$[\mathbf{q}^w, \mathbf{me}^w] \leftarrow \text{CGTessHyp}(\boldsymbol{N}(\mathbf{idnc}))$$

whereas for $\mathcal{T}_{\boldsymbol{N}}^m(k)$ we must have instead

$$[\mathbf{q}^w, \mathbf{me}^w] \leftarrow \text{CGTriangulation}(\boldsymbol{N}(\mathbf{idnc}))$$

So only one line changes in the Algorithm 4 to obtain the new one given in Algorithm 10 where the function CGTriFaces computes $\mathcal{T}_{\boldsymbol{N}}^m(k)$, $\forall k \in 2^{d-m} n_c$.

Algorithm 10 Function CGTriFaces : computes all m-faces tessellations of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$ with m-simplices

---

Input :
$\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.
m : integer, $0 \leqslant m < d$

Output :
$\mathcal{T}_{\boldsymbol{N}}^m$: array of triangulations of all m-faces comming from the cartesian grid triangulation $\mathcal{T}_{\boldsymbol{N}}$.
The length of $\mathcal{T}_{\boldsymbol{N}}^m$ is $E_{m,d} = 2^{d-m}\binom{d}{m}$ (number of m-faces).

Function $\mathcal{T}_{\boldsymbol{N}}^m \leftarrow$ CGTriFaces $(\boldsymbol{N}, m)$
  $\boldsymbol{\beta} \leftarrow$ CGBeta$(\boldsymbol{N})$
  if $m == 0$ **then**
    $\mathbb{Q} \leftarrow$ Diag$(\boldsymbol{N}) *$ CartesianGridPoints(Ones$(1, d)$)
    for $k \leftarrow 1$ **to** 2^d **do**
      $\mathcal{T}_{\boldsymbol{N}}^m(k).\mathbf{q} \leftarrow \mathbb{Q}(:, k)$
      $\mathcal{T}_{\boldsymbol{N}}^m(k).\mathbf{me} \leftarrow 1$
      $\mathcal{T}_{\boldsymbol{N}}^m(k).\text{toGlobal} \leftarrow 1 + \langle \boldsymbol{\beta}, \mathbb{Q}(:, k) \rangle$
    end for
  else
    $n_c \leftarrow \binom{d}{m}$
    $\mathbb{L} \leftarrow$ Combs$([\![1, d]\!], d - m)$
    $\mathbb{S} \leftarrow$ CartesianGridPoints(Ones$(1, d - m)$)
    $k \leftarrow 1$
    for $l \leftarrow 1$ **to** n_c **do**
      idc $\leftarrow \mathbb{L}(l, :)$
      idnc $\leftarrow [\![1, d]\!] \backslash$ **idc**
      $[\mathbf{q}^w, \mathbf{me}^w] \leftarrow$ CGTriangulation$(\boldsymbol{N}(\mathbf{idnc}))$
      $n_{\mathrm{q}}^l \leftarrow \prod_{s=1}^{m} (\boldsymbol{N}(\mathbf{idnc}(s)) + 1)$ ▷ or length of $\mathbf{q}^w$
      for $r \leftarrow 1$ **to** 2^{d-m} **do**
        $\mathcal{T}_{\boldsymbol{N}}^m(k).\mathbf{q}(\mathbf{idnc}, :) \leftarrow \mathbf{q}^w$
        $\mathcal{T}_{\boldsymbol{N}}^m(k).\mathbf{q}(\mathbf{idc}, :) \leftarrow \left(\boldsymbol{N}(\mathbf{idc})^{\mathbf{t}} \mathbin{.*} \mathbb{S}(:, r)\right) *$ Ones$(1, n_{\mathrm{q}}^l)$
        $\mathcal{T}_{\boldsymbol{N}}^m(k).\mathbf{me} \leftarrow \mathbf{me}^w$
        $\mathcal{T}_{\boldsymbol{N}}^m(k).\text{toGlobal} \leftarrow 1 + \boldsymbol{\beta}^{\mathbf{t}} * \mathcal{T}_{\boldsymbol{N}}^m(k).\mathbf{q}$
        $k \leftarrow k + 1$
      end for
    end for
  end if
end Function

3.4 d-orthotope tessellation with d-simplices

Let $\mathcal{O}_d$ be the d-orthotope $[a_1, b_1] \times \cdots \times [a_d, b_d]$.

The mechanism is similar to that seen in section 2.5 while taking as a starting point the cartesian grid triangulation.

Algorithm 11 Function OrthTriangulation : regular tessellation with simplices of a d-orthotope

---

Input :
- $\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.
- $\boldsymbol{a}$, $\boldsymbol{b}$: arrays of d reals, $\boldsymbol{a}(i) = a_i$, $\boldsymbol{b}(i) = b_i$

Output :
- $\mathbf{q}$: vertices array with d-by-n_q reals.
- $\mathbf{me}$: connectivity array with $(d+1)$-by-n_me integers.

Function $[\mathbf{q}, \mathbf{me}] \leftarrow$ OrthTriangulation $(\boldsymbol{N}, \boldsymbol{a}, \boldsymbol{b})$
  $[\mathbf{q}, \mathbf{me}] \leftarrow$ CGTriangulation$(\boldsymbol{N})$
  $\mathbf{q} \leftarrow$ boxMapping$(\mathbf{q}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{N})$
end Function

3.5 m-faces tessellations of a d-orthotope with d-simplices

As seen in section 2.5, we only have to apply the function boxMapping to each vertices array $\mathcal{T}^m_{\boldsymbol{N}}(k).\mathbf{q}$ corresponding to the k-th m-faces tessellations of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$. This is the object of the function OrthTriFaces given in Algorithm 12.

Algorithm 12 Function OrthTriFaces : computes the conforming tessellations with simplices of all m-faces of the d-orthotope $[a_1, b_1] \times \cdots \times [a_d, b_d]$

Input :
- $\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.
- $\boldsymbol{a}$, $\boldsymbol{b}$: arrays of d reals, $\boldsymbol{a}(i) = a_i$, $\boldsymbol{b}(i) = b_i$
- m : integer, $0 \leqslant m < d$

Output :
- $\mathcal{T}^m_{\boldsymbol{N}}$: array of the tessellations with simplices of all m-faces of the orthotope. Its length is $E_{m,d} = 2^{d-m} \binom{d}{m}$.

Function $\mathcal{T}^m_{\boldsymbol{N}} \leftarrow$ OrthTriFaces $(\boldsymbol{N}, \boldsymbol{a}, \boldsymbol{b}, m)$
  $\mathcal{T}^m_{\boldsymbol{N}} \leftarrow$ CGTriFaces$(\boldsymbol{N}, m)$
  for $k \leftarrow 1$ **to** len$(\mathcal{T}^m_{\boldsymbol{N}})$ **do**
    $\mathcal{T}^m_{\boldsymbol{N}}(k).\mathbf{q} \leftarrow$ boxMapping$(\mathcal{T}^m_{\boldsymbol{N}}(k).\mathbf{q}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{N})$
  end for
end Function

---

4 Efficiency of the algorithms

Based on previous algorithms, a Matlab toolbox [3], an Octave package [4] and a python package [5] were developed. They contain a simple class object `OrthMesh`

from which can be obtained, in any dimension $d \geqslant 1$ a simplicial or orthotope mesh with all its m-faces, $0 \leqslant m < d$. It is also possible with the method function `plot` of the class object `OrthMesh` to represent a mesh or its m-faces for $d \leqslant 3$.

In the folowing section, the class object `OrthMesh` is presented. Thereafter some warning statements on the memory used by these objects in high dimension are given. Finally computation times for orthotope meshes and simplicial meshes are given in dimension $d \in [\![1, 5]\!]$.

4.1 Class object `OrthMesh`

The aim of the class object `OrthMesh` is to use previous algorithms to create an object which contains a mesh of a d-orthotope and all its m-face meshes. An elementary mesh class object `EltMesh` is used to store only one mesh, the main mesh as well as any of the m-face meshes. The class `EltMesh` also simplifies writing code. Its fields are the following:

- d, space dimension
- m, kind of mesh ($m = d$ for the main mesh and $m < d$ for m-faces mesh)
- type, 0 for simplicial mesh or 1 for orthotope mesh
- n_{q}, number of vertices
- $\mathbf{q}$, vertices array of dimension d-by-n_{q}
- n_{me}, number of mesh elements
- $\mathbf{me}$, connectivity array of dimension $(d+1)$-by-n_{me} for simplices elements or 2^d-by-n_{me} for orthotopes elements
- toGlobal, index array linking local array $\mathbf{q}$ to the one of the main mesh
- label, name/number of this elementary mesh
- color, color of this elementary mesh (for plotting purpose)

Let the d-orthotope be defined by $[a_1, b_1] \times \cdots \times [a_d, b_d]$. The class object `OrthMesh` corresponding to this d-orthotope contains the main mesh and all its m-face meshes, $0 \leqslant m < d$. Its fields are the following

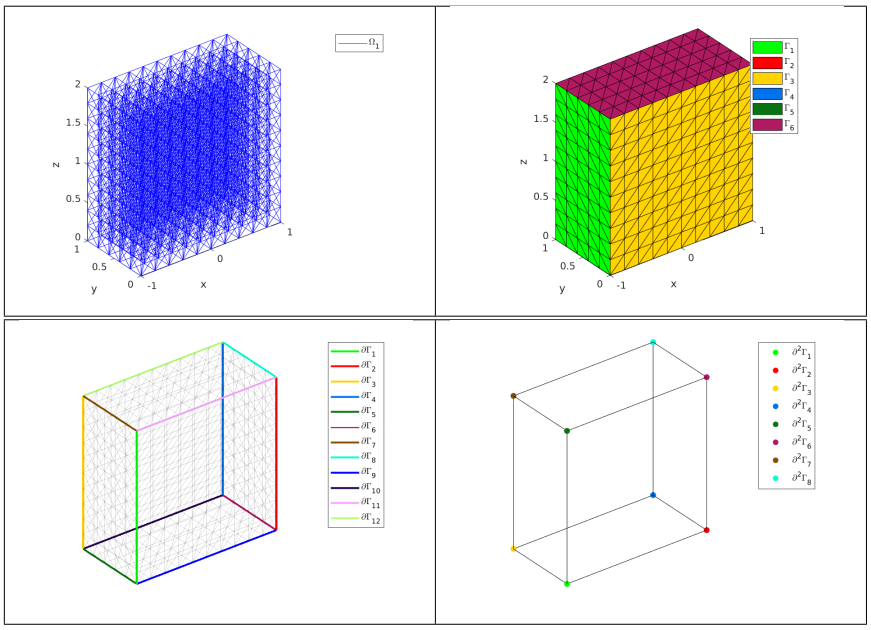
- d: space dimension
- type: string 'simplicial' or 'orthotope' mesh
- Mesh: main mesh as an `EltMesh` object
- Faces: list of arrays of `EltMesh` objects such that Faces(1) is an array of all the $(d-1)$-face meshes, Faces(2) is an array of all the $(d-2)$-face meshes, and so on
- box: a d-by-2 array such that $\mathrm{box}(i, 1) = a_i$ and $\mathrm{box}(i, 2) = b_i$.

The `OrthMesh` constructor is

$$Oh \leftarrow \textsc{OrthMesh}(d, \mathbf{N}, < \text{box} >, < \text{type} >)$$

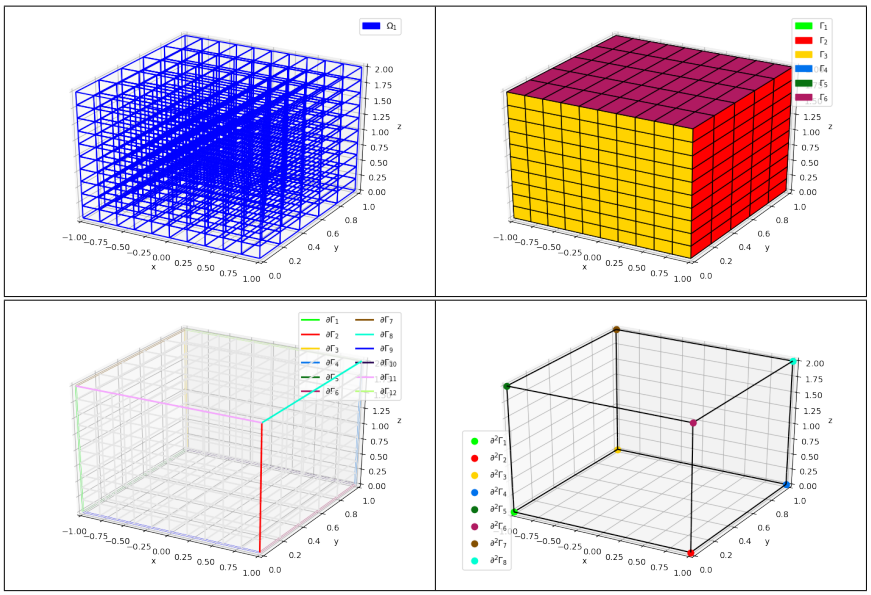
where $\mathbf{N}$ is either a 1-by-d array such that $\mathbf{N}(i)+1$ is the number of grid points discretising $[a_i, b_i]$ or either an integer if the the number of discretization is the same in all space directions. The optional parameter box previously described as for default value $a_i = 0$ and $b_i = 1$. The default value for optional parameter type is 'simplicial', otherwise 'orthotope' can be used.

In Listing 1, an `OrthMesh` object is built under Octave for the orthotope $[-1, 1] \times [0, 1] \times [0, 2]$ with simplicial elements and $\mathbf{N} = (10, 5, 10)$. The main mesh and all the m-face meshes of the resulting object are plotted. In Listing 2, similar operations are done under Python with orthotope elements.



Listing 1: 3D simplicial `OrthMesh` object with Matlab 2017a, main mesh (upper left), 2-face meshes (upper right), 1-face meshes (bottom left) and 0-face meshes (bottom right)

```
Oh=OrthMesh(3,[10,5,10],'box',[-1,1;0,1;0,2])
%  plot the main mesh
figure(1)
Oh.plotmesh('legend',true)
axis equal;xlabel('x');ylabel('y');zlabel('z')
% plot the 2-face meshes
figure(2)
Oh.plotmesh('m',2,'legend',true)
axis equal;xlabel('x');ylabel('y');zlabel('z')
% plot the 1-face meshes
figure(3)
Oh.plotmesh('m',2,'color',[0.8,0.8,0.8],'EdgeAlpha',0.2, 'FaceColor','none')
hold on
Oh.plotmesh('m',1,'Linewidth',2,'legend',true)
axis equal;axis off
% plot the 0-face meshes
figure(4)
Oh.plotmesh('m',1,'color','k')
hold on
Oh.plotmesh('m',0,'legend',true)
axis equal;axis off
```



Listing 2: 3D orthotope `OrthMesh` object with Python 3.6.3, main mesh (upper left), 2-face meshes (upper right), 1-face meshes (bottom left) and 0-face meshes (bottom right)

```
import matplotlib.pyplot as plt
from fc_hypermesh import OrthMesh

oTh=OrthMesh(3,[10,5,10],type='orthotope', box=[[-1,1],[0,1],[0,2]])
# plot the main mesh
plt.ion()
plt.figure(1)
plt.clf()
oTh.plotmesh(legend=True)
# plot the 2-face meshes
plt.figure(2)
plt.clf()
oTh.plotmesh(m=2,legend=True,edgecolor=[0,0,0])
# plot the 1-face meshes
plt.figure(3)
plt.clf()
oTh.plotmesh(m=2,edgecolor=[0.9,0.9,0.9],facecolor=None)
oTh.plotmesh(m=1,legend=True,linewidth=2)
# plot the 0-face meshes
plt.figure(4)
plt.clf()
oTh.plotmesh(m=1,color='black')
oTh.plotmesh(m=0,legend=True,s=55)
```

Of course, the `plotmesh` method doesn't work in dimension $d > 3$!

4.2 Memory consuming

Beware when using theses codes of memory consuming : the number of points n_{q} and the number of elements quickly increase according to the space dimension d. If $(N+1)$ points are taken in each space direction, we have

$$n_{\mathrm{q}} = (N+1)^d, \quad \text{for both tessellation and triangulation}$$

and

$$\begin{array}{rcll} n_{\mathrm{me}} & = & N^d, & \text{for tessellation by orthotopes} \\ n_{\mathrm{me}} & = & d!N^d, & \text{for tessellation by simplices.} \end{array}$$

If the array **q** is stored as *double* (8 bytes) then

$$\text{mem. size of } \mathbf{q} = d \times n_{\mathrm{q}} \times 8 \text{ bytes}$$

and if the array **me** as *int* (4 bytes) then

$$\text{mem. size of } \mathbf{me} = \begin{cases} 2^d \times n_{\mathrm{me}} \times 4 \text{ bytes} & \text{(tessellation by orthotopes)} \\ (d+1) \times n_{\mathrm{me}} \times 4 \text{ bytes} & \text{(tessellation by simplices)} \end{cases}$$

For $N = 10$ and $d \in [\![1, 8]\!]$, the values of n_{q} and n_{me} are given in Table 3. The memory usage for the corresponding array **q** and array **me** is available in Table 4.

d	$n_{\mathrm{q}} = (N+1)^d$	$n_{\mathrm{me}} = N^d$ (orthotopes)	$n_{\mathrm{me}} = d!N^d$ (simplices)
1	11	10	10
2	121	100	200
3	1 331	1 000	6 000
4	14 641	10 000	240 000
5	161 051	100 000	12 000 000
6	1 771 561	1 000 000	720 000 000
7	19 487 171	10 000 000	50 400 000 000
8	214 358 881	100 000 000	4 032 000 000 000

Table 3: Number of vertices n_{q} and number of elements n_{me} for the tessellation of an orthotope by orthotopes and by simplices according to the space dimension d and with $N = 10$.

d	**q**	**me** (orthotopes)	**me** (simplices)
1	88 B	80 B	80 B
2	1 KB	1 KB	2 KB
3	31 KB	32 KB	96 KB
4	468 KB	640 KB	4 MB
5	6 MB	12 MB	288 MB
6	85 MB	256 MB	20 GB
7	1 GB	5 GB	1 612 GB
8	13 GB	102 GB	145 152 GB

Table 4: Memory usage of the array **q** and the array **me** for the tessellation of an orthotope by orthotopes and by simplices according to the space dimension d and with $N = 10$.

In the following pages, computational costs of the `OrthMesh` constructor will be presented.

4.3 Computational times

For all the following tables, the computational costs of the `OrthMesh` constructor are given for the orthotope $[-1, 1]^d$ under Matlab R2017a, Octave 4.2.1 and Python 3.6.0. The computations were done on a laptop with `Core i7-4800MQ` processor and 16Go of RAM under Ubuntu 14.04 LTS (64bits).

In Table 5, some computational costs of the `OrthMesh` constructor

$$Oh \leftarrow \textsc{OrthMesh}(d, N, [-1;1]^d, \texttt{'orthotope'})$$

are given for $d \in [\![2,5]\!]$. Computational costs for tessellations with simplices are presented in Table 6 for $d \in [\![2,5]\!]$. In Appendix C, more detailed tables are given.

d	N	n_{q}	n_{me}	Python	Matlab	Octave
2	4000	16 008 001	16 000 000	1.307 (s)	0.388 (s)	1.473 (s)
3	250	15 813 251	15 625 000	1.896 (s)	0.718 (s)	2.782 (s)
4	62	15 752 961	14 776 336	2.804 (s)	1.321 (s)	5.403 (s)
5	27	17 210 368	14 348 907	4.485 (s)	2.511 (s)	10.781 (s)

Table 5: Tessellation of $[-1,1]^d$ by orthotopes with approximatively 15 millions elements. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

d	N	n_{q}	n_{me}	Python	Matlab	Octave
2	5000	25 010 001	50 000 000	4.362 (s)	2.000 (s)	4.148 (s)
3	180	5 929 741	34 992 000	3.517 (s)	2.202 (s)	4.098 (s)
4	40	2 825 761	61 440 000	4.175 (s)	4.204 (s)	9.798 (s)
5	12	371 293	29 859 840	2.394 (s)	2.788 (s)	8.119 (s)

Table 6: Tessellation of $[-1,1]^d$ with tens of millions of simplices. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

5 Conclusion

In [9], vectorized algorithms are proposed to compute some assembly matrices obtained by the $\mathbb{P}_1$-Lagrange finite element method and this in any space dimension. Furthermore, complete codes were written to solve boundary value problems (B.V.P.) in any space dimension ([6] for Matlab, [7] for Octave and [8] for python). To test these codes for dimensions greater than 3 we need simplicial meshes in dimension 4, 5, ... Meshing softwares, as GMSH or Open CASCADE, do not provide tools to generate kind of meshes. So we have developed vectorized algorithms for the simplest geometry: a d-orthotope. These algorithms are proved to be particularly efficient as they make possible to obtain meshes with tens of millions of elements in a few seconds with either Matlab, Octave or Python. The least performance of Octave are probably due to non-optimal choices during compilation. We work on this point by following several tracks: use of the Intel MKL library, ...

The codes in Matlab, Octave and Python, referenced as `fc_hypermesh`, can be obtained on

`http://www.math.univ-paris13.fr/~cuvelier/software/`

The Python package `fc_hypermesh` is also available on PyPI [10].

A Vectorized algorithmic language

A.1 Common operators and functions

We also provide below some common functions and operators of the vectorized algorithmic language used in this article which generalize the operations on scalars to higher dimensional arrays, matrices and vectors:

$\mathbb{A} \leftarrow \mathbb{B}$	Assignment
$\mathbb{A} * \mathbb{B}$	matrix multiplication,
$\mathbb{A}$ **.*** $\mathbb{B}$	element-wise multiplication,
$\mathbb{A}$ **./** $\mathbb{B}$	element-wise division,
$\mathbb{A}(:)$	all the elements of $\mathbb{A}$, regarded as a single column.
$[,]$	Horizontal concatenation,
$[;]$	Vertical concatenation,
$\mathbb{A}(:, J)$	J-th column of $\mathbb{A}$,
$\mathbb{A}(I, :)$	I-th row of $\mathbb{A}$,
$\textsc{Sum}(\mathbb{A}, dim)$	sums along dimension dim,
$\textsc{Prod}(\mathbb{A}, dim)$	product along dimension dim,
$\mathbb{I}_n$	n-by-n identity matrix,
$\mathbb{1}_{m\times n}$ (or $\mathbb{1}_n$)	m-by-n (or n-by-n) matrix or sparse matrix of ones,
$\mathbb{0}_{m\times n}$ (or $\mathbb{0}_n$)	m-by-n (or n-by-n) matrix or sparse matrix of zeros,
$\textsc{Ones}(m, n)$	m-by-n array/matrix of ones,
$\textsc{Zeros}(m, n)$	m-by-n array/matrix of zeros,
$\textsc{RepTile}(\mathbb{A}, m, n)$	tiles the p-by-q array/matrix $\mathbb{A}$ to produce the $(m \times p)$-by-$(n \times q)$ array composed of copies of $\mathbb{A}$,
$\textsc{Reshape}(\mathbb{A}, m, n)$	returns the m-by-n array/matrix whose elements are taken columnwise from $\mathbb{A}$.

A.2 Combinatorial functions

$\textsc{Perms}(\boldsymbol{V})$	where $\boldsymbol{V}$ is an array of length n. Returns a $n!$-by-n array containing all permutations of $\boldsymbol{V}$ elements. The lexicographical order is chosen.
$\textsc{Combs}(\boldsymbol{V}, k)$	where $\boldsymbol{V}$ is an array of length n and $k \in [\![1, n]\!]$. Returns a $\frac{n!}{k!(n-k)!}$-by-k array containing all combinations of n elements taken k at a time. The lexicographical order is chosen.

B Function CartesianGridPoints

The objective is to explain how to obtain the vectorized function CartesianGridPoints given in Algorithm 1, section 2.2.1. This function returns the vertex array $\mathbf{q}$ of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$. The dimension of $\mathbf{q}$ is d-by-n_{q} with $n_{\mathrm{q}} = \prod_{i=1}^{d}(N_i + 1)$.

According to the numbering choice described in section 2.2.1 the Algorithm 13 gives the most simple presentation of $\mathbf{q}$ computated column by column.

Algorithm 13 Building $\mathbf{q}$ the d-by-n_{q} array of cartesian grid points

```
k ← 1
for i_d ← 0 to N_d do
  for i_{d-1} ← 0 to N_{d-1} do
    ⋱
    for i_2 ← 0 to N_2 do
      for i_1 ← 0 to N_1 do
        ı ← [i_1, i_2, ..., i_{d-1}, i_d]
        q(:, k) ← ı                          ▷ By construction k ≡ G(ı)
        k ← k + 1
      end for
    end for
    ⋰
  end for
end for
```

To vectorize this algorithm we need to rewrite it with computed line by line. For that we write this algorithm, with an explicit for loop on the coordinates: it is given by Algorithm 14.

Algorithm 14 Building $\mathbf{q}$ the d-by-n_{q} array of cartesian grid points

```
k ← 1
for i_d ← 0 to N_d do
  for i_{d-1} ← 0 to N_{d-1} do
    ⋱
    for i_2 ← 0 to N_2 do
      for i_1 ← 0 to N_1 do
        for r ← 1 to d do
          q(r, k) ← i_r
        end for
        k ← k + 1
      end for
    end for
    ⋰
  end for
end for
```

Let $r \in [\![1, d]\!]$. From Algorithm 14, we deduce Algorithm 15 which only computes the component r of the cartesian grid point $\mathcal{Q}_{\mathbf{N}}$ (i.e. the values $\mathbf{q}(r, k)$, $\forall k \in [\![1, n_{\mathrm{q}}]\!]$)

Algorithm 15 Computes component $r \in [\![1, d]\!]$ of cartesian grid points in the d-by-n_{q} array $\mathbf{q}$.

```
Let r ∈ [[1, d]]
k ← 1
for i_d ← 0 to N_d do
  ⋱
  for i_r ← 0 to N_r do
    for i_{r−1} ← 0 to N_{r−1} do
      ⋱
      for i_1 ← 0 to N_1 do
        q(r, k) ← i_r
        k ← k + 1
      end for
      ⋰
    end for
  end for
  ⋰
end for
```

One can replace the for loops i_1 to i_{r-1} by a for loop in j with number of iterations equal to $(N_1 + 1) \times \cdots \times (N_{r-1} + 1) = \beta_{r-1}$. This is done in Algorithm 16.

Algorithm 16 Computes component $r \in [\![1, d]\!]$ of cartesian grid points in the d-by-n_{q} array $\mathbf{q}$.

```
Let r ∈ [[1, d]]
k ← 1
for i_d ← 0 to N_d do
  ⋱
  for i_r ← 0 to N_r do
    for j ← 1 to β_r do
      q(r, k) ← i_r
      k ← k + 1
    end for
  end for
  ⋰
end for
```

We can replace the for loops in i_r and j by a call to the function BuildPA given in Algorithm 17 which returns the array containing the β_{r+1} values stored in array $\mathbf{q}$ by these two loops. The modified code using this function is given in Algorithm 18.

Algorithm 17 Computes the array containing the β_{r+1} values stored in array $\mathbf{q}$ by the for loops in i_r and j.

Input :
$\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.
r : $r \in [\![1, d]\!]$

Output :
$\boldsymbol{A}$: array of $\beta_{r+1} = (N_r + 1)\beta_r$ integers.

```
Function A ← BuildPA (N, r)
  β_r ← ∏_{l=1}^{r-1} (N(l) + 1), k ← 1,
  s ← 1
  for i ← 0 to N_r do
    for j ← 1 to β_r do
      A(s) ← i
      s ← s + 1
    end for
  end for
end Function
```

Algorithm 18 Computes component $r \in [\![1, d]\!]$ of cartesian grid points in the d-by-n_{q} array $\mathbf{q}$.

```
Let r ∈ [[1, d]]
I ← 1 : β_{r+1}
for i_d ← 0 to N(d) do
  ⋱
  for i_{r+1} ← 0 to N(r + 1) do
    q(r, I) ← BuildPA(N, r)
    I ← I + β_{r+1}
  end for
  ⋰
end for
```

As we can see, the BuildPA call in Algorithm 18 does not depend on the for loops i_d to i_{r+1}. Using this property and replacing the for loops i_d to i_{r+1} by a for loop in i with a number of iterations equal to $(N_d + 1) \times \cdots \times (N_{r+1} + 1)$ gives the first writable code in Algorithm 19.

Algorithm 19 Computes component $r \in [\![1, d]\!]$ of cartesian grid points in the d-by-n_{q} array $\mathbf{q}$

```
Let r ∈ [[1, d]]
I ← 1 : β_{r+1}
A ← BuildPA(N, r)
for i ← 1 to (N(d) + 1) × ⋯ × (N(r + 1) + 1) do
  q(r, I) ← A
  I ← I + β_{r+1}
end for
```

We can now write a complete nonvectorized function

Algorithm 20 Function CartesianGridPointsv0 : computes the d-by-n_{q} array $\mathbf{q}$ which contains all the points of the cartesian grid $\mathcal{Q}_{\mathbf{N}}$. (non vectorized version)

Input :
$\boldsymbol{N}$: array of d integers, $\boldsymbol{N}(i) = N_i$.

Output :
$\mathbf{q}$: array of d-by-n_{q} array of integers.

Function $\mathbf{q} \leftarrow$ CartesianGridPointsv0 $(\boldsymbol{N})$
  $\boldsymbol{\beta} \leftarrow$ CGBeta$(\boldsymbol{N})$
  for $r \leftarrow 1$ **to** d **do**
    $\boldsymbol{I} \leftarrow 1 : \beta_{r+1}$
    $\boldsymbol{A} \leftarrow$ BuildPA$(\boldsymbol{N}, r)$
    for $i \leftarrow 1$ **to** $(\boldsymbol{N}(d)+1) \times \cdots \times (\boldsymbol{N}(r+1)+1)$ **do**
      $\mathbf{q}(r, \boldsymbol{I}) \leftarrow \boldsymbol{A}$
      $\boldsymbol{I} \leftarrow \boldsymbol{I} + \beta_{r+1}$
    end for
  end for
end Function

To obtain a vectorized function, we must *work* on the for i loop and on the construction of the array $\boldsymbol{A}$.

We first vectorize the computation of array $\boldsymbol{A}$. Let us define the β_r-by-(N_r+1) array

$$\mathbb{A} = \begin{pmatrix} 0 & 1 & \dots & \boldsymbol{N}(r) \\ 0 & 1 & \dots & \boldsymbol{N}(r) \\ \vdots & \vdots & & \vdots \\ 0 & 1 & \dots & \boldsymbol{N}(r) \end{pmatrix}$$

obtained by copying array $[0 : \boldsymbol{N}(r)]$ on each row of $\mathbb{A}$ from

$$\mathbb{A} \leftarrow \text{RepTile}([0 : \boldsymbol{N}(r)], \beta_r, 1)$$

So array $\boldsymbol{A}$ can be obtained with the command

$$\boldsymbol{A} \leftarrow \text{Reshape}(\mathbb{A}, 1, (\boldsymbol{N}(r)+1)\beta_r)$$

or directly by

$$\boldsymbol{A} \leftarrow \text{Reshape}(\text{RepTile}([0 : \boldsymbol{N}(r)], \beta_r, 1), 1, (\boldsymbol{N}(r)+1)\beta_r)$$

We can easily vectorize the for i loop in function CartesianGridPointsv0 by using the RepTile function as follows

$$\mathbf{q}(r, :) \leftarrow \text{RepTile}(\boldsymbol{A}, 1, \text{Prod}(\boldsymbol{N}(r+1 : d)+1))$$

With these two vectorizations we obtain the function CartesianGridPoints given in Algorithm 1.

C Computational costs

In this section, computational costs of the `OrthMesh` constructor are presented for tessellations of the orthotope $[-1;1]^d$ with orthotopes and simplices. The computations were done on a laptop with `Core i7-4800MQ` processor and 16Go of RAM under Ubuntu 14.04 LTS (64bits).

C.1 Tessellation with orthotopes

Under Matlab 2017a, Octave 4.2.0 and Python 3.6.3, the computational costs of the `OrthMesh` constructor

$$Oh \leftarrow \textsc{OrthMesh}(d, N, [-1;1]^d, \texttt{'orthotope'})$$

are given in tables 8 to 11, respectively for $d = 2$ to $d = 5$.

N	n_{q}	n_{me}	Python	Matlab	Octave
1000	1 002 001	1 000 000	0.146 (s)	0.311 (s)	0.186 (s)
2000	4 004 001	4 000 000	0.339 (s)	0.14 (s)	0.381 (s)
3000	9 006 001	9 000 000	0.755 (s)	0.255 (s)	0.893 (s)
4000	16 008 001	16 000 000	1.307 (s)	0.388 (s)	1.473 (s)
5000	25 010 001	25 000 000	2.018 (s)	0.58 (s)	2.254 (s)

Table 8: Tessellation of $[-1,1]^2$ with orthotopes. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

N	n_{q}	n_{me}	Python	Matlab	Octave
50	132 651	125 000	0.158 (s)	0.353 (s)	0.274 (s)
100	1 030 301	1 000 000	0.229 (s)	0.117 (s)	0.385 (s)
150	3 442 951	3 375 000	0.454 (s)	0.211 (s)	0.741 (s)
200	8 120 601	8 000 000	1.061 (s)	0.402 (s)	1.610 (s)
250	15 813 251	15 625 000	1.896 (s)	0.718 (s)	2.782 (s)
300	27 270 901	27 000 000	3.166 (s)	1.198 (s)	4.514 (s)
350	43 243 551	42 875 000	4.892 (s)	1.838 (s)	8.237 (s)

Table 9: Tessellation of $[-1,1]^3$ with orthotopes. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

N	n_{q}	n_{me}	Python	Matlab	Octave
10	14 641	10 000	0.238 (s)	0.394 (s)	0.571 (s)
20	194 481	160 000	0.243 (s)	0.132 (s)	0.618 (s)
30	923 521	810 000	0.325 (s)	0.172 (s)	0.805 (s)
40	2 825 761	2 560 000	0.546 (s)	0.321 (s)	1.323 (s)
50	6 765 201	6 250 000	1.424 (s)	0.652 (s)	4.122 (s)
62	15 752 961	14 776 336	2.804 (s)	1.321 (s)	5.403 (s)

Table 10: Tessellation of $[-1,1]^4$ with orthotopes. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

N	n_{q}	n_{me}	Python	Matlab	Octave
5	7 776	3 125	0.385 (s)	0.513 (s)	1.405 (s)
10	161 051	100 000	0.400 (s)	0.243 (s)	1.511 (s)
15	1 048 576	759 375	0.594 (s)	0.365 (s)	2.055 (s)
20	4 084 101	3 200 000	1.213 (s)	0.765 (s)	3.722 (s)
25	11 881 376	9 765 625	4.750 (s)	1.832 (s)	9.395 (s)
27	17 210 368	14 348 907	4.485 (s)	2.511 (s)	10.781 (s)

Table 11: Tessellation of $[-1,1]^5$ with orthotopes. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

C.2 Tessellation with d-simplices

Under Matlab 2017a, Octave 4.2.0 and Python 3.6.3, the computational costs of the `OrthMesh` constructor

$$Oh \leftarrow \textsc{OrthMesh}(d, N, [-1;1]^d, \texttt{'simplicial'})$$

are given in tables 12 to 15, respectively for $d = 2$ to $d = 5$.

N	n_{q}	n_{me}	Python	Matlab	Octave
1000	1 002 001	2 000 000	0.190 (s)	0.381 (s)	0.269 (s)
2000	4 004 001	8 000 000	0.522 (s)	0.384 (s)	0.687 (s)
3000	9 006 001	18 000 000	1.145 (s)	0.783 (s)	1.619 (s)
4000	16 008 001	32 000 000	1.944 (s)	1.305 (s)	2.701 (s)
5000	25 010 001	50 000 000	4.362 (s)	2.000 (s)	4.148 (s)

Table 12: Tessellation of $[-1,1]^2$ with simplices. Computational times in seconds for Python 3.6.3, Matlab 2017aand Octave 4.2.1.

N	n_{q}	n_{me}	Python	Matlab	Octave
40	68 921	384 000	0.165 (s)	0.393 (s)	0.295 (s)
60	226 981	1 296 000	0.203 (s)	0.143 (s)	0.406 (s)
80	531 441	3 072 000	0.285 (s)	0.226 (s)	0.598 (s)
100	1 030 301	6 000 000	0.396 (s)	0.370 (s)	0.847 (s)
120	1 771 561	10 368 000	0.592 (s)	0.618 (s)	1.282 (s)
140	2 803 221	16 464 000	0.871 (s)	0.940 (s)	1.838 (s)
160	4 173 281	24 576 000	1.266 (s)	1.341 (s)	2.677 (s)
180	5 929 741	34 992 000	3.517 (s)	2.202 (s)	4.098 (s)

Table 13: Tessellation of $[-1,1]^3$ with simplices. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

N	n_{q}	n_{me}	Python	Matlab	Octave
10	14 641	240 000	0.250 (s)	0.461 (s)	0.609 (s)
20	194 481	3 840 000	0.462 (s)	0.333 (s)	1.068 (s)
25	456 976	9 375 000	0.794 (s)	0.689 (s)	1.804 (s)
30	923 521	19 440 000	1.471 (s)	1.353 (s)	3.335 (s)
35	1 679 616	36 015 000	2.524 (s)	4.104 (s)	6.017 (s)

Table 14: Tessellation of $[-1,1]^4$ with simplices. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

N	n_{q}	n_{me}	Python	Matlab	Octave
2	243	3 840	0.365 (s)	0.557 (s)	1.456 (s)
4	3 125	122 880	0.372 (s)	0.227 (s)	1.420 (s)
6	16 807	933 120	0.496 (s)	0.310 (s)	1.653 (s)
8	59 049	3 932 160	0.617 (s)	0.517 (s)	2.163 (s)
10	161 051	12 000 000	1.048 (s)	1.156 (s)	3.400 (s)
12	371 293	29 859 840	2.394 (s)	2.788 (s)	8.119 (s)

Table 15: Tessellation of $[-1,1]^5$ with simplices. Computational times in seconds for Python 3.6.3, Matlab 2017a and Octave 4.2.1.

List of algorithms

List of Tables

References

[1] Jürgen Bey. Simplicial grid refinement: on freudenthal's algorithm and the optimal number of congruence classes. *Numerische Mathematik*, 85(1):1–29, 2000.

[2] H.S.M. Coxeter. *Regular Polytopes*. Dover books on advanced mathematics. Dover Publications, 1973.

[3] F. Cuvelier. fc_hypermesh: a object-oriented Matlab toolbox to mesh any d-orthotopes (hyperrectangle in dimension d) and their m-faces with simplices or orthotopes. `http://www.math.univ-paris13.fr/~cuvelier/software/`, 2017. User's Guide.

[4] F. Cuvelier. fc_hypermesh: a object-oriented Octave package to mesh any d-orthotopes (hyperrectangle in dimension d) and their m-faces with simplices or orthotopes. `http://www.math.univ-paris13.fr/~cuvelier/software/`, 2017. User's Guide.

[5] F. Cuvelier. fc_hypermesh: a object-oriented Python package to mesh any d-orthotopes (hyperrectangle in dimension d) and their m-faces with simplices or orthotopes. `http://www.math.univ-paris13.fr/~cuvelier/software/`, 2017. User's Guide.

[6] F. Cuvelier. fc_vfem$\mathbb{P}_1$: a object-oriented Matlab toolbox to solve scalar and vector boundary value problems by $\mathbb{P}_1$-lagrange finite element method in any space dimension. `http://www.math.univ-paris13.fr/~cuvelier/software/`, 2017. User's Guide.

[7] F. Cuvelier. fc_vfem$\mathbb{P}_1$: a object-oriented Octave package to solve scalar and vector boundary value problems by $\mathbb{P}_1$-lagrange finite element method in any space dimension. `http://www.math.univ-paris13.fr/~cuvelier/software/`, 2017. User's Guide.

[8] F. Cuvelier. fc_vfem$\mathbb{P}_1$: a object-oriented Python package to solve scalar and vector boundary value problems by $\mathbb{P}_1$-lagrange finite element method in any space dimension. `http://www.math.univ-paris13.fr/~cuvelier/software/`, 2017. User's Guide.

[9] François Cuvelier, Caroline Japhet, and Gilles Scarella. An efficient way to assemble finite element matrices in vector languages. *BIT Numerical Mathematics*, 56(3):833–864, dec 2015.

[10] Python Software Foundation. Pypi, the python package index, 2003.

[11] H. W. Kuhn. Some combinatorial lemmas in topology. *IBM Journal of Research and Development*, 4:518–524, 1960.

[12] K. Weiss. *Diamond-Based Models for Scientific Visualization*. PhD thesis, University of Maryland, 2011.