Exact calculation of quantizer constants for arbitrary lattices

Daniel Pook-Kolb1,2, Bruce Allen1,2, and Erik Agrell3

1 Max Planck Institute for Gravitational Physics (Albert Einstein Institute), Callinstrasse 38, 30167 Hannover, Germany
2 Leibniz Universität Hannover, 30167 Hannover, Germany
3 Chalmers University of Technology, Department of Electrical Engineering, SE-41296 Gothenburg, Sweden

February 13, 2024

We present an algorithm for the exact computer-aided construction of the Voronoi cells of lattices with known symmetry group. Our algorithm scales better than linearly with the total number of faces and is applicable to dimensions beyond 12, which previous methods could not achieve. The new algorithm is applied to the Coxeter–Todd lattice $K_{12}$ as well as to a family of lattices obtained from laminating $K_{12}$. By optimizing this family, we obtain a new best 13-dimensional lattice quantizer (among the lattices with published exact quantizer constants).

Keywords—lattice quantizer, quantizer constant, normalized second moment, Coxeter–Todd lattice, laminated lattice, Voronoi cell

MSC codes—11H06, 11H56, 52B20, 52C07

1 Introduction

Lattices are central to a whole spectrum of mathematical problems. Among these are sphere packing, kissing number, covering, and quantization. A comprehensive review of these problems and descriptions of known classical lattices is given by Conway and Sloane in [15]. Lattices that provide good answers to one or more of these problems and descriptions of known classical lattices break down as the dimension increases. Examples are given in dimensions 12 and 13.

In the present paper, we discuss a new algorithm that improves upon the algorithm developed by Allen and Agrell [5] and can be used to calculate $G$ in higher dimensions. Examples are given in dimensions 12 and 13. Subsequent work [31] uses a slightly improved version of this algorithm to analyze the 16-dimensional Barnes–Wall lattice $\Lambda_{16}$. This example also highlights the improvement made possible by the groundwork laid here. Assuming at least linear scaling of the method in [5] in the total number of faces of the Voronoi cell, we estimate a reduction in running time for $\Lambda_{16}$ of about 5 orders of magnitude.

1.1 Basic notions

An $n$-dimensional lattice $\Lambda$ is a set of points in $\mathbb{R}^n$, $m \geq n$, which is generated by integral combinations of $n$ linearly independent vectors $x_i \in \mathbb{R}^n$, where $i = 1, \ldots, n$. We will adopt the convention that vectors $x$ are treated as row vectors. By a suitable rotation, it is always possible to embed $\Lambda$ in $\mathbb{R}^n$, such that the matrix $B$ of rows formed by the $n$ vectors $x_i$ is an $n \times n$ invertible square matrix. We can use $B$ to express the lattice as

$$\Lambda = \{zB : z \in \mathbb{Z}^n\} .$$  

$B$ is called a generator matrix of the lattice $\Lambda$ and $x_1, \ldots, x_n$ a set of basis vectors. Non-overlapping translated copies of the parallelepiped defined by the basis vectors cover all of $\mathbb{R}^n$, and their $n$-volume is defined as the lattice’s volume $\text{Vol}(\Lambda)$. For a square generator matrix, it is $\text{Vol}(\Lambda) = |\det B|$. Note that the volume is a characteristic of the lattice and is invariant under rotations or a change of the set of basis vectors used to build up $B$.

The main property of a lattice we are interested in is its (dimensionless) normalized second moment or quantizer constant

$$G = \frac{1}{n \text{Vol}(\Lambda)^{2/n}} .$$

Here, $E$ is the average squared distance to the closest lattice point of a uniform random distribution of points in an $n$-dimensional ball if the volume of the ball is much larger than $\text{Vol}(\Lambda)$. The normalization by a power of $\text{Vol}(\Lambda)$ makes $G$ independent of the overall scaling of $\Lambda$, whereas the factor $1/n$ makes lattices of different dimensions have comparable values. For example, the cubic

\[\text{(1)}\]

\[\text{(2)}\]
lattice $\mathbb{Z}^n$ has $G = 1/12$ in any dimension $n$. Minimizing $G$ for fixed $n$ is known as the lattice quantizer problem.

For a fixed dimension $n$, finding the globally optimal lattice is a difficult task in general. There are proofs of optimality only for lattices in dimensions up to three [23, 10], but it is believed that optimal quantizers have been found up to dimension 10 [4, 3, 1, 25]. An especially interesting case is the 9-dimensional lattice $\Lambda_9$. It was found approximately by Agrell and Eriksson in 1998 [2] using numerical optimization, and was recently reconstructed exactly by Allen and Agrell using analytic methods [5]. This lattice, conjectured to be the optimal lattice quantizer in 9 dimensions, is not one of the well-known classical lattices. This is in striking contrast to the best lattice quantizers known in lower dimensions, which are all classical.

A lower bound on $G$, known as Zador’s lower bound (or sphere bound, since it is found by calculating $G$ for an $n$-sphere), is

$$G \geq \frac{\Gamma(1 + n/2)^2/n}{(n + 2)\pi} \cdot$$

where $\Gamma$ is the gamma function. Furthermore, there are conjectured lower (Conway and Sloane [13]) and upper bounds (Zador [40, Lemma 5]) for the optimal lattice quantizer. Zador’s upper bound has been derived elegantly by Torquato [35] as the quantizer constant of a Poisson distribution of points in $\mathbb{R}^n$.

In many dimensions $n \geq 13$, no lattice was known that lies below the Zador upper bound. However, significant progress was made recently in [1], where many new lattices are constructed with quantizer constants representing new records in their respective dimension. In particular, Zador’s upper bound is now satisfied for lattices in dimensions 13, 14, 17, and 25.

A key point made in [1] is that none of the newly constructed lattices is optimal. This presents a promising avenue for further exploration.

Additional improvements were made in [23] in dimensions 14, 15, and 18–23, where Zador’s upper bound is now also satisfied in dimensions 15, 18, and 20.

One of the major challenges is the exact calculation of $G$. In the present work, we present our algorithm for solving this task and apply it to the Coxeter–Todd lattice $K_{12}$ (see [15] Section 4.9) and [16] as well as to a 13-dimensional lattice, which is obtained by laminating $K_{12}$.

The first step towards transforming [2] into a solvable problem is the following well-known observation. Due to the translational symmetry of a lattice, $G$ can be calculated by considering only a single lattice point $x \in \Lambda$ and the region

$$\Omega(x) \equiv \{ y \in \mathbb{R}^n : \|y - x\|^2 \leq \|y - x'\|^2, \; \forall x' \in \Lambda \}$$

1See [4] Footnote [31] for a correction of the original argument.

of points closer to $x$ than to any other lattice point. For brevity, we will often use $\Omega$ to denote $\Omega(0)$. This region is called the Voronoi region or Voronoi cell of $\Lambda$.

The Voronoi cells of each lattice point are translated copies of each other, centered around lattice points, and they cover $\mathbb{R}^n$ without overlap, just like the parallelepipeds formed by the basis vectors. They hence have the same volume [11] Prop. 2.2.1.

In fact, $\Omega$ is a convex polytope with facets lying halfway between the origin and nearby lattice points. The vectors connecting the origin with these points are called Voronoi-relevant vectors or simply relevant vectors and they lie perpendicular to the facets. Let $N(\Omega)$ be the set of relevant vectors. Then

$$\Omega = \{ x \in \mathbb{R}^n : 2x \cdot n \leq \|n\|^2, \; \forall n \in N(\Omega) \} \cdot$$

Using the Voronoi cell $\Omega$, the average squared distance $E$ of a random point from any lattice point can be written as the average squared distance of a point in $\Omega$ from the origin, i.e.,

$$E = \frac{1}{\text{Vol}(\Lambda)} \int_{\Omega} \|x\|^2 \, d^n x .$$

$E$ is related to the (unnormalized) second moment $U$ of $\Omega$ about the origin via $E = U/\text{Vol}(\Lambda)$.

A generalization of the second moment $U$ is the covariance matrix or second moment tensor $U$. For a $d$-dimensional body $P_d$ in $\mathbb{R}^n$, the second moment tensor about a point $x_0 \in \mathbb{R}^n$ is the $n \times n$ matrix

$$U(P_d, x_0) \equiv \int_{P_d} (x - x_0)^T (x - x_0) \, d^d x ,$$

where $d^d x$ denotes a $d$-dimensional volume element. The trace of $U \triangleq U(\Omega, 0)$ is just the second moment $U$ of the Voronoi cell.

Zamir and Feder show in [42] that a lattice that minimizes $G$ has a second moment tensor $U$ which is proportional to the identity matrix. This result has been generalized in [1], where it is shown that it holds for any locally optimal lattice $\Lambda$. The geometric interpretation of the second moment tensor being proportional to the identity matrix is that the moment of inertia about any axis placed through the origin is the same.

Throughout this work, we will denote vectors $x \in \mathbb{R}^n$ and matrices $A \in \mathbb{R}^{k \times m}$ in lower- and uppercase bold font, respectively. Finite sets will be denoted by calligraphic uppercase letters such as $\mathcal{N}$, $\mathcal{V}$, or $\mathcal{F}$, and their cardinality by, e.g., $|\mathcal{F}|$. Definitions are introduced using $\triangleq$. The rest of this paper is organized as follows. In Sec. 2 we will describe an algorithm for constructing the
Voronoi cell of an arbitrary lattice with known symmetry group. Our presentation will revolve around the data structure we use for representing the cell, which hopefully allows the reader to quickly get an overview of the main ideas and our strategy. After discussing the basic construction in four steps in a “naive” approach in Sec. 2.1, the remainder of Sec. 2 will detail the optimizations made possible by incorporating the symmetries. In particular, instead of enumerating all $K$ faces and thus having at least linear time and memory scaling in $K$ (as, e.g., the methods in [24, 37]), we show that the symmetries allow us to construct only a small fraction of all faces. The result is a hierarchy of faces, represented as objects in our algorithm, and much slower growth with $K$. Section 3 gives a brief account of the formulas that can be used to calculate the second moment scalar and tensor of the cell. In Sec. 4 we present the concept of product and laminated lattices and finally apply our algorithm to the Coxeter–Todd lattice $K_{12}$. We conclude in Sec. 5 with a summary and discussion of our method and the results.

2 Constructing the Voronoi cell

In this section, we present our algorithm for constructing the Voronoi cell of an $n$-dimensional lattice $\Lambda$. The goal is to build a data structure that allows the exact calculation of its second moment (scalar and tensor). The method presented here is derived from the one used in [5] but makes greater use of the symmetries of the lattice.

The basic idea is as follows. From (5) it is clear that points on the $(n-1)$-dimensional faces (the facets) of $\Omega$ saturate the inequality for at least one relevant vector $n \in N(\Omega)$. The facets are themselves convex polytopes, with $(n-2)$-dimensional subfaces lying at the intersections of facets. By recursively intersecting subfaces, we obtain faces of ever lower dimension until we arrive at the 0-faces, the vertices. We shall denote the set of vertices of an arbitrary polytope $P_d$ of dimension $d$ as $V(P_d)$.

Above and in the following, we use the term subface to refer to a $d$-face $F_d$ that is a proper subset of a $d'$-face $F_{d'}$, where $d' > d$. A child or child face is a subface $F_d$ of a face $F_{d'}$ of exactly one lower dimension, $d' = d + 1$. Similarly, a parent or parent face of a face $F_d$ is a face $F_{d'}$ such that $F_d$ is a child of $F_{d'}$.

The building blocks of our data structure are shown in Fig. 1. In the end, we will have a hierarchy of “Face” objects, related to each other via parent–child relationships. The recursive formulas we introduce in Section 3 for calculating the scalar and tensor second moment can then easily be applied to this data structure.

In what follows, we will often switch between mathematical statements about, e.g., equivalence classes, stabilizers, or convex polytopes and the representations of these concepts via objects in the data structure. Whenever possible, we use the same notation for mathematical objects and those in our code. For example, a $d$-face $F_d$ is a $d$-dimensional convex polytope, i.e., $F_d \subset \mathbb{R}^n$, where “c” denotes a proper subset. But it may also refer to an object of type “Face” that stores a set of, say, $N$ vertices $\{v_1, \ldots, v_N\}$ as “Vector” objects, which determine the face via their convex hull, $F_d = \text{Conv}(\{v_1, \ldots, v_N\})$. The distinction between these two cases is made explicit only when it is not clear from context. As mentioned above, (nested) sets or lists of faces will often be written as calligraphic $F$, while individual objects are always set in a non-calligraphic font. Within the pseudocode of listed algorithms, spelled-out variables names such as “faces” are sometimes used for clarity.

The three classes shown in Fig. 1 form the basis of our data structure. We defer the discussion of the full set of shown fields to the points where they become relevant. For now, we will focus on their general roles.

“Vector”: These objects represent points in $\mathbb{R}^n$ and vectors between points. The components are stored as exact expressions. Relevant vectors and vertices that define faces are stored as “Vector” objects.

“Face”: Instances of this class represent the various faces of the Voronoi cell. They are defined by a set of vertices stored as “Vector” objects. For each face, its parent faces and child faces are stored, to enable navigation in the hierarchy of faces. Note that “Face” has a subclass “Facet” that adds storage for the relevant vector, its “normal”, which defines the plane it lies in (see below).

Note that the vertices appear in Fig. 1 both as “Vector” and as “Face” objects, since the 0-face objects $F_0$ store just a single vector $v \in V(\Omega)$ in their “vertices” field. This reflects the two roles played by vertices in our data structure: Certain sets of vertices define faces, while each vertex is also a hierarchical entity with parent 1-faces.

“Group”: Symmetry groups and certain subgroups will be stored as objects of this type. It is implemented as a wrapper for groups in GAP [33], which provides a programming language for computational group theory. The various fields will be discussed further below.

The notion of symmetries and equivalence of vectors and faces will be captured by the fields “representative”, “transformation” and “class_id”, which are described in Sec. 2.2.

The majority of our code is written in the Python programming language and the interface to GAP is provided by the Python module gappy [11]. Note that the types “Set”, used in Fig. 1 as well as “List” and “Dictionary”, which we use in the algorithms below, are precisely the set, list, and dict types used in Python. For example, a “Set” object is an unordered collection of items

\footnote{Our use of GAP requires a more recent version than 4.11.1. Specifically, we use the commit [47].}
with efficient ways to compute intersections or to test for membership. Similarly, a “Dictionary” is an object that maps a set of keys to corresponding values, akin to lists mapping non-negative integers (the indices) to values. In contrast to lists, the keys of a “Dictionary” can be, e.g., strings or even objects of type “Vector” or “Face”. In Fig. 1, the type of objects stored in a “Set” is given in square brackets.

It is common in Python that assigning an object to a variable effectively creates a pointer to the underlying object. This mechanic is implied in all algorithms shown here. One consequence of this is that, e.g., the set of vertices each “Face” object stores consists of “Vector” objects from the common pool of all vertices in $\mathcal{V}(\Omega)$. That is, no copies are created and stored.

We now start without considering symmetries and present a conceptually simple method of constructing the face hierarchy of the Voronoi cell. This will introduce the basic tasks and highlight the most expensive steps. Our algorithm then builds upon this method and introduces efficient alternatives.

### 2.1 The naive approach

An explicit construction of the Voronoi cell may be carried out using the following four steps, where steps 1, 3, and 4 are very similar to those presented in [5].

**Step 1.** One first finds all relevant vectors $n_i \in \mathcal{N}(\Omega)$. An efficient algorithm, which is based on [15] Chapter 21, Theorem 10, is described in [3]. The relevant vectors lie orthogonal to the facets, which in turn lie in the hyper-planes

$$E_i \triangleq \{ x \in \mathbb{R}^n : 2x \cdot n_i = \|n_i\|^2 \} .$$

It was shown [30, 38] that there are at most $2(2^n - 1)$ relevant vectors.

Without considering any symmetries for now, we store each of the vectors $n_i$ as “Vector” object with no data for the fields “representative”, “transformation”, and “class_id”. On the other hand, the facets are stored as objects of type “Facet”. At this point, they contain data in their “normal” and “dim” fields, the latter being set to $n - 1$, which is the dimension of the facets of an $n$-dimensional polytope.

**Step 2.** The next step is to find all vertices $\mathcal{V}(\Omega)$ of the Voronoi cell. It is easy to see that the vertices of $\Omega$ are holes of $\Lambda$, i.e., points that have a locally maximal distance to the nearest lattice points. In principle, they can be obtained by taking all possible combinations of $n$ linearly independent vectors $n_i \in \mathcal{N}(\Omega)$ and intersecting the corresponding planes $E_i$. A point $x$ obtained in this way is a vertex if and only if it is an element of the Voronoi cell, i.e., if

$$2x \cdot n_i \leq \|n_i\|^2, \ \forall n_i \in \mathcal{N}(\Omega) .$$  

4By [6], n planes $E_i$ intersect in the points $\{x \in \mathbb{R}^n : 2xM^T = \text{Diag}(MM^T)\}$, where the $n \times n$ matrix $M$ consists of rows formed by the $n$ relevant vectors $n_i$ corresponding to the planes $E_i$. These planes intersect in exactly one point if and only if $M$ has full rank, i.e., if the relevant vectors are linearly independent.
As in step 1, the vertices are stored as “Vector” objects without any symmetry-related data.

Step 3. We next assign each facet its set of vertices by evaluating \( \Omega \) for fixed \( i \) and all \( x \in \mathcal{V}(\Omega) \), and checking for equality, i.e.,

\[
x \in \mathcal{V}(\mathcal{F}_n^{i-1}) \iff 2x \cdot n_i = ||n_i||^2,
\]

where \( \mathcal{F}_n^{i-1} \) is the facet associated with \( n_i \). The resulting sets of vertices then populate the “vertices” field of the corresponding facet objects.

Step 4. The \( (n-2) \)-dimensional faces of the Voronoi cell are obtained by intersecting the sets of vertices of pairs of facets. This is easy to do with the data structure in Fig. 1. We intersect the “vertices” fields of the two facet objects and assign the result to the “vertices” field of a new face object.

In many cases, however, this is not guaranteed to produce a child face. For example, the intersection will be empty if the two facets do not touch. Similarly, their intersection may be a subface that is not a (direct) child. Such a case is visualized in Fig. 2 where two 2-dimensional facets meet in a 0-face. We therefore evaluate the dimension of the convex hull of the new smaller set of vertices. This is done by constructing all vectors connecting one of these vertices with the others and counting the number of linearly independent vectors. If the dimension is \( n-2 \), the hull is a child face and we populate its “parents” field and update the respective facets’ “children” fields. If the dimension is less than \( n-2 \), the face is discarded.

It is possible that distinct combinations of faces intersect in the same child face. For example, the 0-face \( F_0^2 \) in Fig. 2 is the intersection of any pair of the 1-faces \( F_1^2, \ldots, F_2^3 \). We check for that case by comparing the sets of vertices of any newly found child face \( F_{d-1}^i \) against previously found ones. If there is a match \( F_{d-1}^j \), the “parents” field of \( F_{d-1}^i \) and the “children” field of the intersected faces are updated, respectively, and the duplicate \( F_{d-1}^i \) is discarded.

Recursively intersecting child faces as above produces the hierarchy of all lower-dimensional faces.

There are two main problems of this naive approach. First, steps 2 and 4 involve explicit iteration over all possible combinations of quantities which may greatly grow in number with the dimension \( n \) of the lattice. As an example, \( AE_9 \) has 370 relevant vectors and in step 2 one would need to check more than \( 3 \times 10^{17} \) intersections of \( n = 9 \) vectors \( n_i \in \mathcal{N}(\Omega) \) for linear independence and then intersect the planes \( E_i \). Similarly, step 4 would involve computing and checking about \( 7 \times 10^{12} \) intersections of two faces to construct the full hierarchy of 7836067 faces. This gets much worse in higher dimensions. \( K_{12} \), for example, has 4788 relevant vectors and hence would require checking almost \( 3 \times 10^{35} \) combinations in step 2.

The other problem is that the total number of faces may become too large to feasibly perform any recursive calculations involving all those faces. Even though it is still possible to recursively calculate the second moment for the nearly 8 million faces of \( AE_9 \), this becomes exceedingly difficult in higher dimensions.

In the next sections, we will therefore show how this naive method can be improved significantly. The main theme of these improvements will be to use symmetries of the Voronoi cell to greatly reduce the number of objects we need to explicitly track.

### 2.2 Symmetries of a lattice

When viewed as a discrete subset of \( \mathbb{R}^n \), we can ask which rotations \( O(n) \) take the lattice \( \Lambda \) into itself. The subset of \( O(n) \) that does this is called the symmetry group or automorphism group \( \text{Aut}(\Lambda) \) of the lattice, and we write it as

\[
\mathcal{G} \triangleq \text{Aut}(\Lambda) \triangleq \{ g \in O(n) : g\Lambda = \Lambda \}.
\]

The symmetry group of a lattice is always finite (see, e.g., [15, Section 3.4.1] and [27, Theorem 1.4.2]).

The (left) action of a symmetry \( g \in \mathcal{G} \) on a row vector \( x \in \mathbb{R}^n \) will be written as

\[
gx \triangleq xM_g,
\]

where \( M_g \) is the \( n \times n \) matrix representing \( g \) on \( \mathbb{R}^n \), i.e., \( g \) acts via \( M_g^T \) on column vectors on the left.

The action of \( \mathcal{G} \) on faces and any other set of points in \( \mathbb{R}^n \) is defined analogously by applying \( g \) to each point. For a convex \( d \)-dimensional polytope \( P_d \) with vertices \( \mathcal{V}(P_d) \), this immediately implies that

\[
gP_d = \text{Conv}(g\mathcal{V}(P_d)) \).
\]

\(^5\)We use the libraries SymPy [28] and GAP [33], which work with left actions and right actions, respectively. In our case, it is trivial to convert between the conventions by simply transposing the matrices. Hence, the transpose of the matrix representing \( g \) is multiplied from the right in \([12]\) to define the left action here.
The orbit of a vector \( x \) under \( G \) is the set
\[
Gx \triangleq \{ gx : g \in G \}.
\]
(14)

We say that two vectors \( x \) and \( x' \) are equivalent under \( G \) if and only if they lie in the same orbit, i.e., if there exists a \( g \in G \) such that \( x' = gx \). In that case we will write \( x \sim x' \). In the present work, all equivalence relations will originate from group actions, and so we will use the terms “orbit” and “(equivalence) class” interchangeably.

The notion of orbits and equivalence generalizes in the obvious way to any set acted on by \( G \), in particular to the faces of the Voronoi cell.

To see that a face always remains within \( \Omega \) under \( G \), consider a symmetry \( g \in G \). Clearly, since by definition \( g\Lambda = \Lambda \), the set of all Voronoi cells remains invariant as well. In fact, if \( x \in \Lambda \) is a lattice point, then \( g\Omega(x) = \Omega(gx) \). However, since the origin is fixed by \( g \), we have \( g\Omega(0) = \Omega(0) \) and so \( g \) is a symmetry of \( \Omega \). This implies that the complete sets of vertices, relevant vectors, and faces are individually invariant under \( G \).

In our data structure, the symmetry group \( G \) is represented by the “Group” class shown in Fig. 1. We store a set of matrices that generate the full group in the “generators” field. Since any group element \( g \in G \) can be written as a finite product of such generators, this holds the complete information necessary to construct the group. In practice, this set can be relatively small even for large groups. For example, the Coxeter–Todd lattice \( K_{12} \) has a symmetry group of order 78 382 080 [15, p. 129], which we generate with just three matrices (see Sec. 4.1).

The generator matrices are used to construct a GAP group object using the already mentioned GAP system via . This object is stored in the “gap_mat_grp” field of our “Group” class. Since some of the operations in GAP are implemented much more efficiently for permutation groups than for matrix groups, we also let GAP construct a permutation group \( \mathcal{P} \) that is isomorphic to \( G \). This is stored in the “gap_perm_grp” field alongside the isomorphism and its inverse.

In our method, only a small number of instances of the “Group” class will be created. In particular, the full symmetry group \( G \) is such an instance. We furthermore need several subgroups \( \mathcal{U}_i \) of \( G \), which are discussed in Sec. 2.4. These are stored as “Group” instances as well. However, elements \( g \) of the full group or of a subgroup will just be stored as matrices.

The equivalence of vertices, relevant vectors, and faces is represented as follows. Each vector and face has fields “representative”, “transformation”, and “class_id”. For any equivalence class, only one arbitrary element (usually the first one encountered) is chosen as the canonical representative of its class. This element will get a unique integer as “class_id” and no data for the other two fields. Any equivalent item will then store a pointer to this chosen representative and the group element (in matrix form) taking the representative into the item. It will have data in the “representative” and “transformation” fields and no data in “class_id”.

In a concrete implementation, one may choose to have two subclasses of, e.g., the class “Vector”, one for the representative and one for the transformed equivalent vectors. This would avoid reserving redundant storage for the unused fields. For the purposes of discussing our algorithm, however, we will assume that the classification information is stored as shown in Fig. 1.

### 2.3 Finding the vertices

Given the description of \( \Omega \) by the relevant vectors \( \mathcal{N}(\Omega) \), determining the vertices \( \mathcal{V}(\Omega) \) is a representation conversion problem. Standard approaches to this problem are described in [32]. Other methods, which make use of symmetries, are given in [9, 12, 18, 19] and publicly available code [20].

We take a different, more pragmatic approach. Instead of computing the full set of vertices \( \mathcal{V}(\Omega) \) given \( \mathcal{N}(\Omega) \), we find a subset of vertices using a random search and use \( G \) as well as the translational symmetries of the lattice to determine additional vertices. As discussed below, this alone does not ensure that we find the full set of vertices. However, the subsequent steps of our analysis provide a proof of completeness. In practice, this has turned out to be an efficient method to determine the full set \( \mathcal{V}(\Omega) \) with reasonable computational cost.

To find a subset of \( \mathcal{V}(\Omega) \), we do not form intersections of facets as in step 2 of Sec. 2.1. Instead, it is more efficient to search within \( \Omega \) for local maxima of \( ||x||^2 \). This can be done using linear programming, with \( \Omega \) characterized by \( \mathcal{N}(\Omega) \), and yields exact expressions of the vertices. As in [5, Section 2], we solve \( \max_{x \in \Omega}(c \cdot x) \) multiple times, but rather than choosing \( c \) uniformly, we find it more efficient to set it to a relevant vector \( n \in \mathcal{N}(\Omega) \) with a small random perturbation.

Further vertices are obtained by utilizing the translational symmetries of \( \Lambda \) as follows: Let \( v \) be a vertex and let \( \mathcal{M}(v) \) be the set of nearest lattice points of \( v \). These can be found using the algorithms presented in [3]. Since \( v \) is a vertex of the Voronoi cell, the origin is one of those lattice points. More generally, \( v \) is a vertex of all the Voronoi cells \( \Omega(n) \) for any \( n \in \mathcal{M}(v) \). However, since all the Voronoi cells are congruent translated copies of each other, we have
\[
v \in \mathcal{V}(\Omega) \implies v - n \in \mathcal{V}(\Omega), \quad \forall n \in \mathcal{M}(v).
\]

(15)

Vertices found via (15) need not be equivalent under \( G \), as we have used the translational symmetries here.
These are, by definition, not included in the automorphism group \( G \) since that fixes the origin. Any new vertex thus leads to further vertices by constructing its orbit under \( G \).

Whenever we construct the orbit of a vertex, we first define that vertex as the fixed representative of its class and assign it a new “class id”. Upon applying the group to this representative, we store the representative and the group element in the respective fields of our “Vector” class (see Fig. 1). We use the sets of vertices in the orbits found thus far in order to quickly check if any newly found vertex represents a new or a known class. Vertices in known classes are then discarded.

The search for vertices is continued with random initial conditions until no new classes of vertices appear. We do not have a strict criterion for stopping this search. However, when many iterations produce only known classes, one may proceed with the analysis under the assumption that all vertices have been found. At a later point, a side product of our calculations is the volume of the convex hull of the vertices. Comparing this volume with the known volume \( \text{Vol} (\Lambda) = |\det B| \) provides an unambiguous consistency check, since the convex hull of every proper subset of the full set of vertices also has a strictly smaller volume than the full Voronoi cell. This technique of testing for completeness has been used previously in [37]. However, the test can not be carried out during or immediately after the vertex search, because the volume is only available near the end of the second moment computations. So although the overall computation tests that the vertex classes are complete, that test is far more expensive than the vertex search.

Our random search for vertices makes the task of determining all vertices feasible even for higher dimensions. For example, the symmetry group of laminated \( K_{12} \) (see Sec. 4.2) partitions the 52 351 632 vertices of the Voronoi cell into 482 different classes. By using (15), one can find representatives of each of these classes from just 74 vertices. This means that a random search needs to find at least one of each of the 74 subsets of vertices, which in practice is accomplished on a single core within less than a day.

A different method for finding vertices is given in [21] Section 3.2. After finding a single initial vertex using a cutting-plane algorithm [21] Section 3.3, it is deterministic and finds all vertices of \( \Omega \) up to automorphisms and translations by lattice vectors.

The method [21] Section 3.3] of finding the initial vertex shares certain aspects with our random search for vertices. However, the two methods scale differently with the number of vertices found. While our method is faster when finding many vertices, as needed here, the algorithm in [21] Section 3.3] is faster when only a very small number of vertices is required.

In both vertex-finding algorithms, the complexity is dominated by solving linear programs (LP) and finding all closest points (ACP) in a lattice. Let \( C_{\text{LP}} (n, k) \) denote the complexity of solving an LP in \( n \) dimensions with \( k \) constraints. Let \( C_{\text{ACP}} (n) \) denote complexity of solving an ACP problem in \( n \) dimensions. These functions depend not only on \( n \) and \( k \) but also on the lattice, implementation, and hardware.

For our algorithm, the relevant vectors are determined in a preprocessing step using [3]. This requires solving \( 2^n - 1 \) LP problems. To find one vertex when the \( r \) relevant vectors are known, an LP with \( r/2 \) constraints is solved. For a single vertex, including the preprocessing, the complexity is thus

\[
(2^n - 1) C_{\text{ACP}} (n) + C_{\text{LP}} (n, r/2)
\]

and per vertex it is

\[
C_{\text{PAA}} = \frac{2^n - 1}{M} C_{\text{ACP}} (n) + C_{\text{LP}} (n, r/2)
\]

when \( M \) vertices are found in total.

In contrast, the algorithm in [21] Section 3.3] requires no preprocessing. To find one vertex, an iterative process is applied, where each step consists of solving one LP and one ACP problem. The number of constraints increases in every iteration from \( 2n \) in the first LP. The number of iterations needed varies with different vertices for the same lattice, and so does the increasing sequence of the number of constraints. If \( L \) iterations are needed to find a certain vertex, and the numbers of constraints are \( c_1, \ldots, c_L \), where \( c_1 = n/2 \), then the complexity of the algorithm in [21] Section 3.3] to find that vertex is

\[
L C_{\text{ACP}} (n) + \sum_{i=1}^{L} C_{\text{LP}} (n, c_i) .
\]

The average complexity per vertex is therefore

\[
C_{\text{DSV}} = \mathbb{E}[L] C_{\text{ACP}} (n) + \mathbb{E} \left[ \sum_{i=1}^{L} C_{\text{LP}} (n, c_i) \right],
\]

where the expectations are over different sequences of \( L \) and \( c_1, \ldots, c_L \), depending on the starting points.

We illustrate the difference with \( K_{12} \) in \( n = 12 \) dimensions. It has \( r = 4788 \) relevant vectors. We found \( M = 1000 \) vertices using the algorithm in [21] Section 3.3] and obtained sequences of \( L \) and \( c_1, \ldots, c_L \) for each vertex. The number of iterations \( L \) was on average 40.9, ranging from 29 to 52. The 40 900 LPs used on average 70.5 constraints, ranging from 24 to 330.

We implemented both algorithms in Mathematica on a Macbook Pro laptop. The CPU time measured in one test is plotted in Fig. 3. Here, \( C_{\text{DSV}} \) averaged 590 ms per vertex, whereas \( C_{\text{PAA}} \) averaged 55 ms per vertex for \( M = 1000 \) vertices and approaches 37 ms per vertex for large \( M \). While the numerical results will vary for different
2.4 Constructing the hierarchy of faces

The most challenging of the steps to optimize is the construction of the hierarchy of faces, i.e., step 4 of the naive approach in Sec. 2.1. We will describe our solution to this problem in two parts. The first part is discussed in this section and focuses on optimizing the structure of the hierarchy, given knowledge about the equivalence of faces. This is followed in Sec. 2.5 by an algorithm for evaluating that equivalence.

2.4.1 Strategy for constructing the face hierarchy

Instead of building the full hierarchy of all faces as is done in [21, Section 3.3], we will show that for the calculation of the second moment tensor [7] it is sufficient to explicitly consider only a small subset of faces, exploiting the symmetries.

Our method is similar to that of [21] where symmetries are used to simplify the problem. However, we unconditionally compute all classes of faces in dimensions \(0, \ldots, n\). Furthermore, our method of constructing child faces via intersections of sets of vertices is different and we use a different algorithm for determining the equivalence of faces than those proposed in [21, Section 3.5].

Our basic construction is visualized in Fig. 4. It shows the faces of various dimensions \(d\) as boxes in different rows. Within each row, faces that are equivalent under \(G\) lie adjacent to each other, while small gaps demarcate inequivalent subsets. For example, the figure shows nine \((n-2)\)-faces partitioned into three classes. In each class, we arbitrarily select one face as representative, shown as a box with gray background. The children of these representative faces are indicated by solid lines. In fact, except for the top layer \(F_n\), the sets \(F_d\) of \(d\)-faces that we construct consist of only those children; no other face is needed.

We point out a few important properties of this structure:

(i) The sets of children of each chosen representative face is complete, i.e., the face hierarchy contains all their child faces (solid lines in Fig. 4).

(ii) At least one face of each class of faces is constructed.

(iii) At least one parent is a representative. However, in general we do not have a full set of parents.

(iv) In general, only few of the children of faces that are not chosen as representative are constructed (dashed lines in Fig. 4), even though they clearly have the same number of children as all others in their class. There may even be cases, like the last face in the class of \(F_{n-2}^1\), where none of its children are constructed.

Points (i) and (ii) are essential for the correctness of our results. After describing our algorithm in detail, we will therefore give proofs that the resulting structure possesses these properties.

One might be worried that the arbitrary choice of a representative in each class of faces may lead to intersections being missed. For example, consider two inequivalent representatives \(F\) and \(F'\) and their orbits under \(G\). If \(g_1 F\) intersects with \(g_2 F'\) in a child face \(F_C\), i.e.,

\[
F_C = g_1 F \cap g_2 F',
\]

(20)
Algorithm 1 Setup routine for building the face hierarchy.

```
1 procedure ConstructLattice()
2     n ← dimension of the lattice
3     gens ← Set of generator matrices of \( \mathcal{G} \)
4     \( \mathcal{N} \) ← Set of relevant vectors (step 1 of Sec. 2.1)
5     \( \mathcal{V} \) ← Set of vertices (Sec. 2.3)
6 return ConstructFaceHierarchy(n, gens, \( \mathcal{N} \), \( \mathcal{V} \))
```

for some \( g_1, g_2 \in \mathcal{G} \), then how do we know that the children of \( F \) and \( F' \) contain a face equivalent to \( F_C \)? Properties [(i)] and [(ii)] precisely say that this is the case. In this particular situation, we can simply transform \( F_C \) with \( g_1 \) to see that

\[
y_1^{-1}F_C \subset F
\]

and so \( F_C \) is equivalent to \( g_1^{-1}F_C \), which is a child of \( F \).

The above construction greatly reduces the total number of faces to consider compared to the full set of faces. For example, the 9-dimensional lattice \( \text{AE}_9 \) has 7836067 faces in total which fall into 170 classes. By including only the child faces of the representatives, our method results in about 2000 faces to construct.

Note that we lose easy access to the information about the total number of \( d \)-faces of the Voronoi cell. In principle, this number can be calculated by computing the sizes of orbits of the faces under \( \mathcal{G} \). Since the orbit size of a face \( F \) equals \([\mathcal{G}]/|\text{Stab}(F)|\), where \( \text{Stab}(F) \) is the stabilizer of \( F \), the total number of faces can be calculated from their stabilizers. However, since neither the orbits nor the stabilizers of all faces are needed for our method, a combinatorial argument that obtains this number from the information we do have would be desirable. Such an argument may involve information about the subfaces that occur in intersections but are discarded due to having too low dimension (see step 4 in Sec. 2.1). However, we have not been able to resolve this question, which is not critical for the computation of quantizer constants.

2.4.2 The basic algorithms

We will now introduce our algorithms that construct the hierarchy of faces. They are split into several procedures, which are shown in this section and Sec. 2.3. Fig. 3 shows how the various procedures are related and in which algorithm they can be found.

Several preparatory steps are required before the face hierarchy can be built. For definiteness, they are shown in Alg. 1 and build on the results of the previous steps 1 and 2 of Sec. 2.1 with step 2 being replaced by Sec. 2.3. Alg. 1 is specific to each lattice we analyze. For the Coxeter–Todd lattice \( K_{12} \), \( n \) is set to 12 in line 2 and

\[
g \equiv \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}
\]

the generator matrices in line 3 are those given below in (29). With the preparations done, the final step is calling the generic procedure \( \text{ConstructFaceHierarchy} \), which is shown in Alg. 2 and implements our algorithm for constructing the hierarchy of faces.

Alg. 2 itself proceeds in the following way. We start by constructing the symmetry group \( \mathcal{G} \) from the given generator matrices. The relevant vectors \( \mathcal{N} \) and vertices \( \mathcal{V} \) are then used to construct the “Facet” objects (step 3 of Sec. 2.1). Lines 5–8 create the \( n \)-face \( F_n \). On line 10 we initialize the result list \( \mathcal{F} \), which will contain sets \( \mathcal{F}_d \) of \( d \)-faces for each dimension \( d = 0, \ldots, n \). At this point, only the facets \( (d = n - 1) \) and the cell itself \( (d = n) \) are known. The loop on line 11 then iterates from the \( (n-2) \)-faces down to the 0-faces. In each dimension \( d \), we classify the parents into equivalence classes, which arbitrarily select one representative face in each class. Then in line 12 we intersect each of these representative faces with all children of one of its \( \mathcal{G} \)-representatives parents using Alg. 3. As we prove below, this will create all child faces of those representatives, which are stored in \( \mathcal{F}_d \).

The classification itself includes identifying equivalences and finding transformations between equivalent faces. It is carried out by the procedure \( \text{ClassifyFaces} \) shown in Alg. 4. As a first step, in line 2 it splits the given list of \textit{faces} into multiple smaller lists, each containing only faces that share certain properties that are

Figure 5: Call graph showing the relationships between the procedures we present in this paper. Arrows point from the calling to the called procedure. The labeled boxes indicate in which of our algorithm listings the procedures can be found.
Algorithm 2 Construct a hierarchy of faces. See the main text for details.

1 procedure CONSTRUCTFACEHIERARCHY(n, gens, N, V)
2 \( G \leftarrow \text{new GapGroup} \) Create symmetry group object
3 \( G \text{.generators} \leftarrow \text{gens} \)
4 \( \text{faces} \leftarrow \text{faces using } N \text{ and } V \text{ as in step 3 of Sec. 2.1} \)
5 \( F_0 \leftarrow \text{new Face} \) Create n-face and populate its fields
6 \( F_0 \text{.dim} \leftarrow n \)
7 \( F_0 \text{.class_id} \leftarrow 0 \)
8 \( F_0 \text{.vertices} \leftarrow V \)
9 \( F_0 \text{.children} \leftarrow \text{faces} \) \( F \) is a list of sets \( F_0, F_1, \ldots, F_n \):
10 \( F \leftarrow \{0\}, \ldots, \{0\}, F_0, \text{children}, \{F_0\} \)
11 for \( d \leftarrow n - 2 \) to 0 do Iterate from \( n - 2 \) to 0-faces
12 \( \text{CLASSIFYFACES}(F_{d+1}, G, \text{faces}) \)
13 \( \text{children} \leftarrow \text{INTERSECTFACES}(F_{d+1}) \)
14 \( F_d \leftarrow \text{children} \) Store d-faces in result list
15 return \( F \)

Algorithm 3 Create child faces of representatives.

1 procedure INTERSECTFACES(faces)
2 \( \text{children} \leftarrow \text{new empty Dictionary} \)
3 Outer loop over representatives:
4 for all \( P_2 \in \text{REPS}(\text{faces}) \) do
5 \( F \leftarrow \text{NULL} \) Storage for a potential child face
6 Intersect the sets of vertices:
7 \( V \leftarrow P_2 \text{.vertices} \cap P_2 \text{.vertices} \)
8 if \( V \in \text{children} \) then Child is already known
9 \( F \leftarrow \text{children}[V] \) Get the face object
10 add \( P_1, P_2 \) to \( F \text{.parents} \) Add new parents
11 else if dimension of \( V = P_1 \text{.dim} - 1 \) then
12 Affine space has the correct dimension.
13 Create a new “Face” object:
14 \( F \leftarrow \text{new Face} \)
15 \( F \text{.dim} \leftarrow P_1 \text{.dim} - 1 \)
16 \( F \text{.vertices} \leftarrow V \)
17 \( F \text{.parents} \leftarrow \{P_1, P_2\} \)
18 store that we know this set of vertices:
19 \( \text{children}[V] \leftarrow F \)
20 if \( F \neq \text{NULL} \) then
21 Register \( F \) as child of the two parents:
22 add \( F \) to \( P_1 \text{.children} \) and \( P_2 \text{.children} \)
23 \( \text{children} \leftarrow \text{Set values in } \text{children} \text{ dictionary} \)
24 return \( \text{children} \)

Collects representatives of already classified faces:

19 procedure REPS(faces)
20 \( F \text{\textsuperscript{rep}} \leftarrow \text{new empty List} \)
21 for all \( F \in \text{faces} \) do
22 \( \text{if } F \text{.representative} = \text{NULL} \) then
23 \( \text{add } F \text{ to } F \text{\textsuperscript{rep}} \)
24 return \( F \text{\textsuperscript{rep}} \)

Collects children of (any) one representative parent:

25 procedure RELEVANTSIBLINGS(F)
26 for all \( F_{d+1} \in F \text{\textsuperscript{rep}} \text{.parents} \) do
27 \( \text{if } F_{d+1} \text{.representative} = \text{NULL} \) then
28 return \( F_{d+1} \text{.children} \)
29 \( \text{error “Face has no representative parent”} \)

Algorithm 4 Classify a set of faces (mutates \text{faces}).

1 procedure CLASSIFYFACES(faces, G, faces)
2 \( \text{faces_list} \leftarrow \text{SPLITBYINVARIANTS}(\text{faces}, \text{faces}) \)
3 \( F \) is a list of faces sharing certain invariants:
4 for all \( F \in \text{faces_list} \) do
5 \( \text{this will contain the representatives of all classes} \)
6 \( F \text{\textsuperscript{rep}} \leftarrow \text{new empty List} \)
7 \( \text{for all } F \in F \text{ do} \)
8 \( \text{for all } F \text{\textsuperscript{rep}} \in F \text{\textsuperscript{rep}} \) do
9 \( g \leftarrow \text{FINDTRANSFORMATION}(F \text{\textsuperscript{rep}}, F) \)
10 \( \text{if } g \neq \text{NULL} \text{ then} \)
11 \( F \text{.representative} \leftarrow F \text{\textsuperscript{rep}} \)
12 \( F \text{.transformation} \leftarrow g \)
13 \( \text{break} \) No need to check other \( F \text{\textsuperscript{rep}} \)
14 \( \text{if } F \text{.representative} = \text{NULL} \text{ then} \)
15 \( F \) is in a new class; mark it as representative:
16 \( \text{add } F \text{ to } F \text{\textsuperscript{rep}} \)
17 \( F \text{\textsuperscript{rep}} \text{.class_id} \leftarrow \text{next available integer} \)
18 \( \text{faces_by_invariants} \leftarrow \text{new empty Dictionary} \)
19 for all \( F \in \text{faces} \) do
20 \( \mathcal{N} \leftarrow \text{GETNORMALS}(F, \text{faces}) \)
21 \( \text{Calculate invariants } inv \text{ (see Sec. 2.5)} \)
22 \( inv \leftarrow \text{PERCLASSCOUNT}(F, \text{vertices} \cup \mathcal{N}) \)
23 \( \text{if } inv \notin \text{faces_by_invariants} \text{ then} \)
24 \( \text{This value of } inv \text{ is new; initialize a list for it:} \)
25 \( \text{faces_by_invariants}[inv] \leftarrow \{F\} \)
26 \( \text{else} \)
27 \( \text{append } F \text{ to faces_by_invariants}[inv] \)
28 return \( \text{List of values in faces_by_invariants dictionary} \)

invariant under \( G \). Which of these invariants we use is discussed in more detail in Sec. 2.5. For now, it is sufficient to know that faces with different invariants are guaranteed to be inequivalent under \( G \).

A very valuable property of the loop in line 3 is that each iteration is fully independent of any other iteration. In particular, no change is made to the data structure outside the faces in the sub-list \( F \) of the current iteration. Problems with this property are called “embarrassingly parallel” since they can in principle be run in parallel across many cores or even computers in a cluster.

The procedure \text{FINDTRANSFORMATION} used in line 7 is responsible for evaluating whether two faces \( F \) and \( F' \) are equivalent under \( G \). If they are, it returns one of the group elements (there may be multiple) such that \( F' = gF \). This highly non trivial task is discussed in detail in Sec. 2.5.5, where an algorithmic solution is given.

2.4.3 Proving important properties

As mentioned above, it is important that Alg. 2 produces results having the properties (i) and (ii) (see Sec. 2.4.1), which we will now prove. We start with (i), i.e., that Alg. 2 yields all children of each representative face.

We argue by recursion. First, (i) holds trivially in dimension \( d = n \), since all children of the (only) \( n \)-face \( F_0 \) are generated. Recall that these are precisely the facets, which are all constructed in line 4 of Alg. 2. Next, assume
it is true in dimension $d+1$: the set of constructed faces of dimension $d$ includes all children of all representative $(d+1)$-faces. Within this set of $d$-faces, Alg. 2 arbitrarily selects one representative per subset of equivalent $d$-faces. Let $F$ be one such representative $d$-face. By construction, $F$ has at least one parent $P$ that is a representative face. By assumption, Alg. 2 includes all children of $P$ among the faces of dimension $d$. However, it is clear that all child faces of a face $F$ may be constructed by picking any parent $P$ of $F$ and intersecting $F$ with all other children of $P$. It follows immediately that all children of $F$ are constructed.

We continue with the proof of (6) which states that the set $\mathcal{F}_{d+1}^{\text{rep}} \triangleq \text{Rep}(\mathcal{F}_d)$ of representatives of the set of $d$-faces $\mathcal{F}_d$ constructed by Alg. 2 is complete: the union of the orbits

$$ \bigcup_{F_d^{\text{rep}} \in \mathcal{F}_{d+1}^{\text{rep}}} GF_d^{\text{rep}} \tag{22} $$

is the full set of $d$-faces of the Voronoi cell $\Omega$.

Here, the proof is recursive and by contradiction. Assume that the union provides a complete face set in dimension $d+1$, but that the Voronoi cell includes a $d$-face $F'$ which is not constructed by Alg. 2 and is also not equivalent to one of the representative $d$-faces which is constructed. Since $F'$ is a $d$-face, it must have a parent, which by assumption can be written as $P = gF_d^{\text{rep}}$, where $g$ is a group element and $F_d^{\text{rep}}$ a representative $(d+1)$-face. In this case, by symmetry, $g^{-1}P$ must have $g^{-1}F'$ as a child. Since $g^{-1}P = F_{d+1}^{\text{rep}}$ is a representative face, and Alg. 2 constructs all children of representative faces, then $g^{-1}F'$ must have been constructed. But this contradicts our assumption, since $g^{-1}F'$ is in the same orbit as $F'$ and hence equivalent to it.

### 2.4.4 Iterated classification

The classification step in Alg. 2 can be further optimized as follows. For a given set of faces, one may perform the classification via ClassifyFaces(faces, $U$, facets) using subgroups $U \subset G$. More precisely, let

$$ U_1, U_2, \ldots, U_N \tag{23} $$

be a list of proper subgroups of $G$. Generally, we shall order them such that $|U_i| \leq |U_{i+1}|$ but without implying inclusion relations.

The classification using the subgroups at the beginning of this list is usually much faster than with the subgroups at the end or with the full group $G$. We therefore produce classes of faces by starting the classification with $U_1$ and working up to $U_N$ followed by $G$. In each step, only one representative of the previous classification is considered. If two of the representatives turn out to be equivalent, then the whole sets they represent can be merged.

Note that it is possible for faces in a single class under $U_i$ to be inequivalent under $U_{i+1}$. This is not a problem, as equivalence is clearly restored under $G$.

The choice of subgroups may have a large influence on the resulting computational cost. In practice, it turned out to be sufficient to use a subset of stabilizers of the facets and $(n-2)$-dimensional faces.8 The stabilizer of an element $X$ (for example a face or a vector) acted on by $G$ is the set

$$ \text{Stab}(X) \triangleq \{ g \in G : gX = X \} . \tag{24} $$

It is easy to construct such a stabilizer using GAP. For individual vectors, on which the matrix group acts directly, one may use the pre-defined action “OnPoints”

```gap
gap> Stabilizer(G, x, OnPoints);
```

where $G$ is the matrix group and $x$ the vector. This is shown here with GAP syntax and is translated accordingly when used in Python with gappy. To compute the stabilizer of a face, we take a set of vectors defining the face (see Sec. 2.5) and use the pre-defined action “OnSets”. For example, for a face defined by three vectors $x_1, x_2, x_3$, one may use

```gap
gap> vectors = AsSet([x1, x2, x3]);
gap> Stabilizer(G, vectors, OnSets);
```

### 2.4.5 Selective use of floating-point calculations

Several steps during the construction of the face hierarchy can be made faster by utilizing floating-point calculations. The final results are still exact. An example is collecting the vertices belonging to a facet via (10). Using a small numerical tolerance $\varepsilon$, i.e.,

$$ x \in \mathcal{V}(F_{n-1}) \iff \|2x \cdot n_i - \|n_i\|^2\| \leq \varepsilon , \tag{25} $$

we can quickly find the subset of vertices contained in the facet $F_{n-1}$. A second example is evaluating the dimension of the convex hull of the intersection of two faces, which is used in line 10 of Alg. 3.

Even the search for vertices itself, described in Sec. 2.3 may be carried out with floating-point operations. To obtain an exact expression for an inexact vertex $v$, one can use (25) to collect all relevant vectors $n_i \in \mathcal{N}(\Omega)$ of the facets it lies in. Since a vertex lies in at least $n$ facets, this set will contain at least $n$ vectors. If it is larger, we select $n$ linearly independent ones. The intersection of the corresponding planes $E_i$ is formed as in step 2 of Sec. 2.3, yielding the exact expression of the vertex $v$.

We emphasize that such floating-point calculations are optional: the algorithm can be implemented using exact operations only.

---

8We constructed stabilizers for the representatives of the $(n-1)$- and $(n-2)$-faces. For laminated $K_{12}$, we used only some of these stabilizers. This is explained in more detail in Sec. 4.2.
2.5 Evaluating the equivalence of faces

The above strategy of constructing the hierarchy of faces of $Ω$ hinges on an efficient way to evaluate if two faces are equivalent. In Alg. 4 line 7, this is performed by the call to `FindTransformation`. This section presents our approach to solving this problem (see [21] Section 3.5 and [31] Section V.A.1 for other approaches). We start in Sec. 2.5.1 and 2.5.2 by discussing two possible sets of vectors one may use, followed by our decision criterion in Sec. 2.5.3. Recall that in Alg. 4, the equivalence test is only performed with faces that share certain invariants. Since the specific invariants we use are important for our algorithm, they are presented next in Sec. 2.5.4 before we introduce our method (Sec. 2.5.5) and the algorithm itself (Sec. 2.5.6). Finally, Sec. 2.5.7 solves the problem of determining all transformations in $G$ taking one vector into another, which is a key ingredient in our method.

2.5.1 Equivalence using vertices

We first identify the quantities one may use to evaluate if two faces $F_d$ and $F_d'$ are equivalent. Recall that $F_d \sim F_d'$ if and only if there exists a $g \in G$ such that $F_d' = gF_d$. From [13], it is clear that

$$F_d' = gF_d \iff V(F_d') = gV(F_d), \quad (26)$$

that is, we can use the sets of vertices of the two faces.

For faces in higher dimensions, however, the number of vertices can become too large for computationally evaluating equivalence. For example, the 10-faces of $K_{12}$ contain between 104 and 6978 vertices each.

2.5.2 Equivalence using relevant vectors

An alternative to the vertices is the set of relevant vectors of the faces within which a face lies. For a $d$-face $F_d$, where $d < n$, this set is defined by

$$\mathcal{N}(F_d) \triangleq \{ n_i \in \mathcal{N}(\Omega) : F_d \subset E_i \}, \quad (27)$$

where $\mathcal{N}(\Omega)$ is the set of all relevant vectors and $E_i$ are the planes (8) the corresponding facets lie in. The vectors $n_i \in \mathcal{N}(F_d)$ are all orthogonal to $F_d$. In fact, since all faces of dimension $d < n$ are ultimately obtained by repeated intersections of facets, (27) collects the relevant vectors of the facets whose intersection yields $F_d$, i.e.,

$$F_d = \bigcap_{n \in \mathcal{N}(F_d)} F^n. \quad (28)$$

Here, $F^n$ is the facet belonging the relevant vector $n \in \mathcal{N}(\Omega)$.

Note that for $d = n$, the only face is the Voronoi cell itself, $F_n = \Omega$, and so (27) would result in the empty set. We hence restrict this definition to $d < n$ and define $\mathcal{N}(F_n)$ as the full set of all relevant vectors.

Algorithm 5 Compute the set $\mathcal{N}(F_d)$.

1. procedure GetNormals($F_d$, facets)
2. if $F_d.f\text{acet\_normals} \neq \text{NULL}$ then
3. return $F_d.f\text{acet\_normals}$
4. $\mathcal{N} \leftarrow \text{new empty Set}$
5. for all $F_{n-1} \in \text{facets}$ do
6. if $F_d.\text{vertices} \subseteq F_{n-1}.\text{vertices}$ then
7. $F_{j}$ is a subface of $F_{n-1};$ collect its normal:
8. $\mathcal{N} \leftarrow \mathcal{N} \cup \{F_{n-1}.\text{normal}\}$
9. $F_d.f\text{acet\_normals} \leftarrow \mathcal{N}$
10. return $\mathcal{N}$

The number of vectors in $\mathcal{N}(F_d)$ depends on the dimension $d$ of the face. To see this, consider $k$ linearly independent vectors $n_i \in \mathcal{N}(\Omega)$. The intersection of the corresponding $E_i$ is an affine space of dimension $n - k$. This means that $\mathcal{N}(F_d)$ will contain at least $k = n - d$ vectors, since $n - k = d$. $\mathcal{N}(F_d)$ may contain more elements, as can be seen in Fig. 2. For the face $F_d$, we have $n = 3$ and $d = 0$, but the set $\mathcal{N}(F_d)$ contains the four normals of the facets $F_1^2, \ldots, F_2^2$.

An important observation for us is that

$$F_d' = gF_d \iff \mathcal{N}(F_d') = g\mathcal{N}(F_d). \quad (29)$$

To prove this, first note that by (8), we have

$$gE_i = \{ g x \in \mathbb{R}^n : 2 x \cdot n_i = \| n_i \|^2 \} = \{ x \in \mathbb{R}^n : 2(g^{-1} x) \cdot n_i = \| n_i \|^2 \} = \{ x \in \mathbb{R}^n : 2x \cdot (gn_i) = \| n_i \|^2 \} = E_i', \quad (30)$$

where $E_i'$ is the plane corresponding to the transformed normal $gn_i$. Let now $F_d$ be a $d$-face, where $d < n$, and $g \in G$. Then, since $F_d \subset E_i \iff gF_d \subset gE_i$ and using the definition (27) and (30), we get $\mathcal{N}(gF_d) = g\mathcal{N}(F_d)$. For the other direction, let $F_d$ and $F_d'$ be two $d$-faces. Since $gF^n = F'^n$, we have with (28)

$$gF_d = \bigcap_{n \in \mathcal{N}(F_d)} F^n = \bigcap_{n \in g\mathcal{N}(F_d)} F^n. \quad (31)$$

If now $g\mathcal{N}(F_d) = \mathcal{N}(F_d')$, this implies $gF_d = F_d'$.

Together, (29) and (31) mean that we are free to use either $V(F_d)$ or $\mathcal{N}(F_d)$ to evaluate equivalence. In the example of $K_{12}$, the set (27) has exactly 2 elements for all 10-faces.

We use Alg. 3 to obtain $\mathcal{N}(F_d)$ in our code. This is made particularly efficient by our data structure. We simply iterate over all facets and check if the “vertices” field of $F_d$ is a subset of the “vertices” field of the respective facet (line 6). Recall from Fig. 1 that facets are instances of the subclass “Facet” and thus store their relevant vector in the “normal” field. Since $\mathcal{N}(F_d)$ is used multiple times in our algorithm, we store the normals in the “facet\_normals” field for later re-use.
We remark that Alg. 5 could have been implemented by recursively collecting all parent faces up to dimension \( n - 1 \). However, this does not work in our case since the information about a face’s parents is not guaranteed to be complete and so some parents may be missed.

The next steps require that the fields “representative”, “transformation”, and “class_id” of the relevant vectors \( \mathcal{N}(\Omega) \) have been populated. As described in Sec. 2.3, the vertices have already been classified as part of the finding process, i.e., these fields are fully populated. For the much smaller number of relevant vectors, we will assume that this has been done in a similar way.

### 2.5.3 Deciding which set of vectors to use

We can now construct a defining set \( \mathcal{D}(F_d) \) for each face, which will be used for evaluating equivalence. A simple choice for \( \mathcal{D}(F_d) \) would be the smaller of the two sets \( \mathcal{V}(F_d) \) and \( \mathcal{N}(F_d) \). However, for reasons that will become clear below, we instead take the following approach. First, both sets \( \mathcal{V}(F_d) \) and \( \mathcal{N}(F_d) \) are sorted such that equivalent vectors are adjacent and the classes in the same order. We then count the number of permutations that keep equivalent vectors adjacent and the classes in the same order. The set with the smaller number of these permutations is chosen as \( \mathcal{D}(F_d) \). In case both sets have the same number of those permutations, we choose the smaller set, or, for definiteness, \( \mathcal{N}(F_d) \) if that number is also equal. We do not present this logic in a formal algorithm. It will be used later in Alg. 7 line 31.

The criterion for the choice of \( \mathcal{D}(F_d) \) is based purely on invariants under \( \mathcal{G} \): two equivalent faces necessarily have the same number of vertices and relevant vectors in each class. In general, \( \mathcal{D}(F_d) = \mathcal{V}(F_d) \) for small \( d \) and \( \mathcal{D}(F_d) = \mathcal{N}(F_d) \) for large \( d \), but it is not only a function of \( d \). For two equivalent \( d \)-faces \( F_d \) and \( F'_d \), it may happen that \( \mathcal{D}(F_d) = \mathcal{N}(F_d) \) but \( \mathcal{D}(F'_d) = \mathcal{V}(F'_d) \).

### 2.5.4 Inexpensive invariants of faces

Before deciding if \( F'_d \sim F_d \), it is possible to perform several inexpensive tests that can show if equivalence is excluded. Any property of a face that is invariant under \( \mathcal{G} \) can be used for this purpose.

With our data structure, several such invariants of a face \( F_d \) are easily calculated. Obvious ones are the total number of vertices \( |\mathcal{V}(F_d)| \) and of facet normals \( |\mathcal{N}(F_d)| \).

The number of children of a face is another possible invariant. However, it is not available in our algorithm, because the classification is performed prior to constructing the child faces.

An additional invariant is the per-class count of elements in \( \mathcal{V}(F_d) \) and \( \mathcal{N}(F_d) \), which is calculated by Alg. 6. The per-class count contains all the information needed to evaluate the criterion for the choice of \( \mathcal{D}(F_d) \) described at the end of Sec. 2.5.3. In fact, we use the per-class count of the union \( \mathcal{V}(F_d) \cup \mathcal{N}(F_d) \) in line 19 of Alg. 4 to partition the faces as a first step in their classification.

### 2.5.5 A strategy for finding a transformation

Our goal now is to determine if \( \mathcal{D}(F'_d) \) and \( \mathcal{D}(F_d) \) are equivalent, and, in case they are, to find a symmetry \( g \in \mathcal{G} \) such that \( \mathcal{D}(F'_d) = g\mathcal{D}(F_d) \). We will assume that the two faces \( F_d \) and \( F'_d \) share the same per-class count discussed above. This means that the defining sets \( \mathcal{D}(F_d) \) and \( \mathcal{D}(F'_d) \) have an equal number of vectors per equivalence class and we can form \( N \) pairs of individually equivalent vectors. Here, \( N \triangleq |\mathcal{D}(F_d)| = |\mathcal{D}(F'_d)| \) is the number of vectors in each of the defining sets.

Our method is illustrated in Fig. 6. The sets \( \mathcal{D}(F_d) \) and \( \mathcal{D}(F'_d) \) are each split into subsets of equivalent vectors. In our algorithms, the partition \( \mathcal{X} \) of \( \mathcal{D}(F_d) \) is represented as a list of lists \( \mathcal{X}_j \). For a fixed \( j \), the vectors \( x \in \mathcal{X}_j \) are all equivalent. We have in this example \( N = 7 \), \( |\mathcal{X}_j| = 3 \), and \( |\mathcal{X}_j| = 2, 2, 3 \) for \( j = 1, 2, 3 \), respectively. Similarly, \( \mathcal{Y} \) is a partition of \( \mathcal{D}(F'_d) \) with subsets \( \mathcal{Y}_j \). We first order the subsets \( \mathcal{X}_j \) and \( \mathcal{Y}_j \) by the “class_id” field of the contained vectors. Since the per-class counts are the same, the vectors in \( \mathcal{X}_j \) and those in \( \mathcal{Y}_j \) are in the same class for each \( j \). We further order the subsets such that

\[
|\mathcal{X}_j| = |\mathcal{Y}_j| \leq |\mathcal{X}_{j+1}| = |\mathcal{Y}_{j+1}| .
\]
This is done with a stable sorting algorithm so that for each \( j \) the vectors in \( X_j \) and \( Y_j \) remain in the same class, even when \(|X_j| = |X_{j+1}|\).

The main idea of our algorithm is now as follows. Let \( \sigma \) be a permutation of \( 1, \ldots, N \), which keeps the pairs equivalent, \( x_i \sim y_{x(i)} \) for all \( i = 1, \ldots, N \). In other words, \( \sigma \) permutes the vectors \( y_i \) only within the individual subsets \( Y_j \), which is indicated by the arrows inside the \( Y \) row in Fig. 6. The single permutation \( \sigma \) is equivalent to a set \( \{\sigma_1, \ldots, \sigma_{|Y|}\} \) of permutations acting on the individual subsets \( Y_j \). Let further

\[
T_k^i \triangleq \{ g \in G : y_k = g x_i \}
\]

be the set of all group elements taking the single vector \( x_i \) into the single vector \( y_k \). Clearly, if there exists a permutation \( \sigma \) such that

\[
\bigcap_{i=1}^N T_i^\sigma(i) \neq \emptyset,
\]

then \( F_d \sim F_d' \) and any group element \( g \) in this intersection takes \( F_d \) into \( F_d' \). If the intersection is empty for all permutations, then \( F_d \) and \( F_d' \) are inequivalent.

One property of this method that may become a limitation is the scaling with the factorial of the sizes of the subsets \( Y_j \). By permuting only within the subsets, the number of permutations to check is reduced from \( N! \) to

\[
\prod_j |Y_j|!.
\]

We argue below that this number is a worst case only and we discuss several means by which we can often avoid most of these permutations.

Despite this, our algorithm may fail if the subset sizes \(|Y_j|\) become too large and thus too many permutations need to be checked. For such cases, one may switch to a different algorithm. Alternatives include the GAP function “RepresentativeAction”, which would replace the call to \textsc{FindTransformation} in Alg. 2. However, in the cases we encountered for the 12 and 13-dimensional lattices discussed in Sec. 4, the method described here was superior.\(^{10}\)

### 2.5.6 An algorithm for finding a transformation

In this section, we describe our implementation of the method outlined in Sec. 2.5.5

\(^{10}\)As an example, we consider the classification of 1,068 different 8-faces of \( K_{12} \) that are children of the 15 representative 9-faces. They lie in 40 classes with respect to the full group \( G \). The classification required in total 1,122 equivalence tests, with an average of 2.36 seconds per test. This is about 30 times faster than “RepresentativeAction”, which took about 71 seconds per test on average. We observed that this difference increases with the number of vectors \( |D| \).

---

```
Algorithm 7 Evaluate if \( F_d \sim F_d' \). Return a group element \( g \in G \) such that \( F_d' = gF_d \) if \( F_d \sim F_d' \), else NULL.

1 procedure 
2 \textsc{FindTransformation}(F_d, F_d')
3 Split vectors by class and order them as in Fig. 6.
4 \textsc{PartitionDefiningSet}(F_d)
5 \textsc{PartitionDefiningSet}(F_d')
6 Start recursion with full group \( G \) and subset pair \((X_1, Y_1)\).
7 \textsc{T} \leftarrow \textsc{FilterTransforms}(X, Y, G, 1)
8 if \( \textsc{T} \neq \emptyset \) then
9 \( \textsc{FilterTransforms}(X, Y, \textsc{T}, 1) \)
10 else return one arbitrary element \( g \in \textsc{T} \)
11 return NULL
12 procedure \textsc{FilterTransforms}(X, Y, \textsc{T}, j)
13 \textsc{TransformsOfSubsets}(X_j, \sigma_j Y_j, \textsc{T}_j-1)
14 if \( \textsc{T}_j \neq \emptyset \) then
15 Recurse to next pair of subsets \((X_{j+1}, Y_{j+1})\):
16 \( \textsc{T}_j \leftarrow \textsc{FilterTransforms}(X, Y, \textsc{T}_j, j+1) \)
17 \textsc{Recursion stopping criterion}
18 \( \textsc{T}_j \leftarrow \textsc{TransformsOfSubsets}(X_j, \sigma_j Y_j, \textsc{T}_j-1) \)
19 \textbf{else}
20 \( \textsc{T}_j \leftarrow \emptyset \)
21 procedure \textsc{TransformsOfSubsets}(X_j, Y_j, \textsc{T}_j-1)
22 \textsc{TransformsOfSubsets}(X_j, Y_j, \textsc{T}_j-1)
23 Iterate over all pairs of vectors in \( X_j \) and \( Y_j \):
24 for all \( \sigma_j \in \text{Perm}(1, \ldots, |X_j|) \) do
25 \( x \leftarrow k \)-th vector of the list \( X_j \)
26 \( y \leftarrow k \)-th vector of the list \( Y_j \)
27 Construct all transformations of the \( k \)-th pair:
28 \( T_{xy} \leftarrow \{ \sigma \} \)
29 \( \textsc{TransformsOfSubsets}(X_j, Y_j, \textsc{T}_j-1) \)
30 \( \textsc{T}_j \leftarrow \textsc{FilterTransforms}(X, Y, \textsc{T}_j, j+1) \)
31 Pool is not empty, go to next pair.
32 return \( \textsc{T}_j \)
33 procedure \textsc{PartitionDefiningSet}(F)
34 \( D \leftarrow \text{defining set of } F \) as in Sec. 2.5.3
35 We store \( \mathcal{X} \) as list \([X_1, X_2, \ldots] \) of lists \( X_j \).
36 \( X \leftarrow \text{partition } D \) based on the “class id” of the vectors
37 sort \( \mathcal{X} \) based on the “class id” of the vectors in \( X_j \)
38 stably sort \( \mathcal{X} \) based on the sizes \(|X_j|\)
39 return \( \mathcal{X} \)
```

A key point is to keep track of a pool of remaining transformations, defined as follows. Let \( \mathcal{P}_0 = G \) be the
full group. We set

\[ \mathcal{P}_{j_1 \ldots j_L} = \prod_{i=1}^L \mathcal{T}_{i}^{\sigma(i)}, \]

where \( L = \sum_{j=1}^J |\mathcal{X}_j| \) is the number of vectors up to and including the \( j \)th pair of subsets \((\mathcal{X}_j, \mathcal{Y}_j)\) and where \( \sigma \) corresponds to the permutation of the first \( L \) vectors in \( \mathcal{D}(\mathcal{F}_d) \) defined by the permutations \( \sigma_1, \ldots, \sigma_j \) of the first \( j \) subsets \( \mathcal{Y}_k \). For brevity, we will drop the explicit dependency of the pool on the permutations \( \sigma_1, \ldots, \sigma_j \) and write \( \mathcal{P}_j \).

Our method now recursively goes through all pairs \((\mathcal{X}_j, \mathcal{Y}_j), j = 1, \ldots, |\mathcal{X}| \). Generally, we move down to the next pair as soon as we find a permutation \( \sigma_j \) such that \( \mathcal{P}_j \) is nonempty. We move back up one level to try the next permutation \( \sigma_j-1 \) if none are found in the current level.

To go from \( j \) to \( j+1 \) we first initialize \( \mathcal{P}_{j+1} \rightarrow \mathcal{P}_j \). For each pair \((x_i, y_{\sigma(i)})\) of vectors \( x_i \in \mathcal{X}_{j+1} \) and \( y_{\sigma(i)} \in \mathcal{Y}_{j+1} \), we then construct the set of transformations \( \mathcal{T}_{i}^{\sigma(i)} \) and update \( \mathcal{P}_{j+1} \rightarrow \mathcal{P}_{j+1} \cap \mathcal{T}_{i}^{\sigma(i)} \).

Two cases can occur during this process. First, \( \mathcal{P}_j \) may become empty at some point when going through the pairs of vectors in the \( j \)th pair of subsets. In that case, we iterate over the permutations \( \sigma_j \) of \( \mathcal{Y}_j \) and each time repeat this process from the beginning of the \( j \)th pair of subsets. This is done until a permutation \( \sigma_j \) is found where \( \mathcal{P}_j \) remains nonempty. If one is found, we go to the next pair \((\mathcal{X}_{j+1}, \mathcal{Y}_{j+1})\). If none is found, we go back to the pair \( j-1 \) and try the next permutation of \( \mathcal{Y}_{j-1} \). Finally, if the permutations of \( \mathcal{Y}_j \) are exhausted, the faces are inequivalent. The second case is that we end up with a nonempty set of remaining transformations \( \mathcal{P}_j \) after going through all pairs of subsets. This immediately ends the search, since any of its elements takes \( F_d \) into \( \mathcal{F}^n \).

Alg. \ref{alg} implements this idea as follows (the implicit dependency of the procedures on the full group \( \mathcal{G} \) is suppressed). Lines \ref{line3} and \ref{line4} construct the partitions \( \mathcal{X} \) and \( \mathcal{Y} \) (i.e., the boxes in Fig. \ref{fig:6}) ordered by their size and such that the vectors in \((\mathcal{X}_j, \mathcal{Y}_j)\) are equivalent for each \( j \). We then enter the recursive procedure FilterTransforms. It takes as arguments the procedure \( \mathcal{X} \) and \( \mathcal{Y} \) as well as the pool \( \mathcal{P}_{j-1} \) of remaining transformations and subset index \( j \) to work on next. The recursion is started in line \ref{line9} with the full group \( \mathcal{P}_0 = \mathcal{G} \) as pool and \( j = 1 \) to begin with the first pair of subsets. Line \ref{line10} recurses by advancing to the next pair of subsets.

Note that, in general, not all permutations \( \sigma \) need to be checked, not even those that respect the partition. First, if the faces are equivalent, then this equivalence can be manifest in many of the permutations. On the other hand, even if the faces are inequivalent, in many cases the algorithm does not need to check all permutations. To see this, consider again Fig. \ref{fig:6}. There are two vector pairs in the first subset and hence only two permutations of \( \mathcal{Y}_1 \). If \( T^1 \cap T^2 \) and \( T^2 \cap T^3 \) are both empty, then none of the permutations of \( \mathcal{Y}_2 \) or \( \mathcal{Y}_3 \) are considered. This is the reason for ordering the subsets \( \mathcal{X}_j \) and \( \mathcal{Y}_j \) by increasing size: the larger subsets are only looked at if transformations exist for the smaller ones.

A further significant reduction of the number of permutations is achieved by the iterated classification described in Sec. \ref{sec:2.4.4}. Recall that in this method, we use proper subgroups \( \mathcal{U}_i \subset \mathcal{G} \) (usually stabilizers of the \((n-1)\)- and \((n-2)\)-faces) to evaluate if two faces are equivalent. This creates subsets of equivalent faces, each of which is represented by only one face. This classification is repeated with different subgroups, where in each iteration we consider only those representatives. In the last iteration, the full group \( \mathcal{G} \) is used on a now much smaller set of faces representing the already collected subsets of equivalent faces.

The number of permutations is generally much smaller than for the smaller subgroups \( \mathcal{U}_i \). The reason is that vectors that are equivalent under \( \mathcal{G} \) may become inequivalent under \( \mathcal{U}_i \). Consider the case shown in Fig. \ref{fig:6}. A subgroup \( \mathcal{U}_i \) may partition \( \mathcal{D}(\mathcal{F}_d) \) and \( \mathcal{D}(\mathcal{F}_d) \) into, e.g., four or five subsets \( \mathcal{X}_j \) and \( \mathcal{Y}_j \), respectively, which can drastically decrease the number of permutations \( (35) \).

The maximal values of \( |\mathcal{X}_j| \) across all face classes in all dimensions are 6 for \( \mathcal{AE}_5 \) and 10 for both \( K_{12} \) and

| \( d \) | \( |\mathcal{D}(\mathcal{F}_d)| \) | \( |\mathcal{X}_1|, \ldots \) | \( \Pi_j |\mathcal{X}_j|! \) | checked | equiv. |
|---|---|---|---|---|---|
| 5 | 14 | (1, 6, 7) | 3628800 | 169908 | yes |
| 5 | 14 | (1, 6, 7) | 3628800 | 152903 | yes |
| 4 | 10 | (10) | 3628800 | 127811 | yes |
| 5 | 14 | (1, 6, 7) | 3628800 | 30738 | yes |
| 4 | 10 | (10) | 3628800 | 12553 | yes |
| 5 | 12 | (4, 8) | 967680 | 14563 | yes |
| 5 | 12 | (4, 8) | 967680 | 9553 | yes |
| 5 | 12 | (4, 8) | 967680 | 9073 | yes |
| 5 | 13 | (1, 5, 7) | 604800 | 1772 | yes |
| 5 | 13 | (1, 6, 6) | 518400 | 735 | yes |
laminated $K_{12}$. For laminated $K_{12}$, we list in Tab. 1 the worst cases of numbers of permutations occurring in invocations of FindTransformation. We note that for inequivalent faces, there were four cases where all 241 920 permutations were checked. On average, in the roughly 8.5 million invocations, about 6.1 of 34.8 permutations had to be iterated over for equivalent faces and about 9.2 of 23.1 permutations for inequivalent faces.

2.5.7 Constructing the full set of transformations between two vectors

The only remaining calculation to describe is the construction of the full set of transformations $T_{i}^{x(i)}$ taking $x_{i}$ into $y_{x(i)}$ for a given $i$. In Alg. 1, this is done in line 25 where we changed the notation for simplicity, i.e., we consider the pair of vectors $(x, y)$ and the set $T_{xy}$ of all transformations taking $x$ to $y$. Luckily, since the fields “representative” and “transformation” of each vector object are populated, we always know one11 element $g_{xy} \in G$ such that $y = g_{xy}x$. Then $T_{xy}$ is given by the left coset

$$T_{xy} = g_{xy} \text{Stab}(x) ,$$

where $\text{Stab}(x)$ is the stabilizer of $x$.

Note that if the stabilizer $\text{Stab}(x)$ of the (arbitrary but fixed) representative vector $v_{\text{rep}}$ of a given class has been calculated and stored, then the stabilizer of any other vector $x = g_{x}v_{\text{rep}}$ in the same class can efficiently be generated via conjugation

$$\text{Stab}(x) = g_{x} \text{Stab}(v_{\text{rep}}) g_{x}^{-1} .$$

We make use of this fact by caching the stabilizers of the representative vectors $v_{\text{rep}}$. The stabilizer of a vector depends not only on the vector itself but also on the group. Since we use several groups $G_{i}$, in the iterated classification described in Sec. 2.4.4, a single vector generally has multiple stabilizers we need to keep track of. This is easily achieved by storing them in a dictionary field of each individual group object (see the “Group” class in Fig. 1). The “stabilizer” cache has the vectors as keys and the stabilizers as values. Whenever a stabilizer of a vector $x = g_{x}v_{\text{rep}}$ is requested from a group object, we query its cache for the $v_{\text{rep}}$ key. If it is not found, the stabilizer is calculated (via GAP) and stored in the cache. Then we calculate $\text{Stab}(x)$ using (38).

3 Calculating the second moment

With the face hierarchy built as discussed in the previous section, it is possible to calculate the second moment

$$\text{tensor} [7] \text{ of the Voronoi cell } \Omega \text{ using explicit recursion relations. The strategy for our calculations closely follows the method described in Section 3 of [5], which corrects and extends the original relations derived in Section IV-C of [37].}$$

In our case, however, we can greatly reduce the computational cost of evaluating these relations by making use of the symmetries of the faces. When a quantity has been computed for a representative in $F_{d}^{\text{rep}}$, we immediately obtain this quantity for all equivalent faces. Scalar quantities are equal for all faces within a class, while vector or tensor quantities can be transformed using the known group element. This significantly reduces the number of steps in our calculations.

Let $F_{d} \in F_{d}^{\text{rep}}$ be the selected representative of one class of $d$-faces. Its centroid $c(F_{d})$ is defined as the mean of its vertices $V(F_{d})$,

$$c(F_{d}) \triangleq \frac{1}{|V(F_{d})|} \sum_{v \in V(F_{d})} v ,$$

and, due to the convexity of $F_{d}$, the centroid is guaranteed to lie in $F_{d}$. The volume of $F_{d}$ is defined by

$$\text{Vol}(F_{d}) \triangleq \int_{F_{d}} d^{d} x .$$

A recursion relation for the volume is

$$\text{Vol}(F_{d}) = \frac{1}{d} \sum_{F \in \mathcal{C}(F_{d})} h_{c}(F) \text{Vol}(F) ,$$

where $\mathcal{C}(F_{d}) \subseteq F_{d-1}$ is the set of all child faces of $F_{d}$ and $h_{c}(F)$ is the height of the parent’s centroid, $c(F_{d})$, above the plane containing its child face $F$.

The height $h_{c}(F)$ is calculated in either of two ways described in [5]. The first is to project the difference $\Delta c_{F} \triangleq c(F) - c(F_{d})$ onto the space orthogonal to the child face $F$ and then calculating the norm. This requires the orthogonalization and normalization of a linearly independent set of, e.g., relevant vectors $\mathcal{N}(F)$ orthogonal to $F$.

The second method of computing $h_{c}(F)$ is to construct any set of vectors $v_{1}, v_{2}, \ldots, v_{d-1}$, spanning the $(d-1)$-dimensional plane the child $F$ lies in.12 Then we have

$$h_{c}(F)^{2} = \frac{\text{det Gram}(v_{1}, \ldots, v_{d-1}, \Delta c_{F})}{\text{det Gram}(v_{1}, \ldots, v_{d-1})} ,$$

where Gram$(\cdot)$ is the Gram matrix of the vectors in its argument.
As in [3], the decision on which method to use is based on the dimension $d$ of the face. For low dimensions, calculating the spanning vectors and Gram determinants turned out to be faster while in higher dimensions, the projections are less expensive. We switch the method usually close to $n/2$, although the precise threshold dimension seemed not to become a major factor up to $n = 13$.

Being a scalar quantity that is invariant under $G$, we calculate the volume once for each face in $F_{d-1}^{\text{rep}}$ and subsequently obtain $\text{Vol}(F)$ of each child face of the $d$-face $F_d$ from its respective representative, i.e., $\text{Vol}(F) = \text{Vol}(F_{d-1}^{\text{rep}})$ if $F \sim F_{d-1}^{\text{rep}}$. The recursion starts with the 0-faces $F_0 \in F_0^{\text{rep}}$, which have a 0-volume of 1.

Similarly, the barycenter of $F_d$,

$$ b(F_d) \equal\frac{1}{\text{Vol}(F_d)} \int_{F_d} x \, d^d x, $$

satisfies a recursion relation [5, Eq. (3.5)]

$$ b(F_d) = \frac{1}{d+1} \left( c(F_d) + \frac{1}{\text{Vol}(F_d)} \sum_{F \in C(F_d)} h_c(F) \text{Vol}(F)b(F) \right). $$

(43)

Again, the barycenter $b(F)$ of a child face $F$ can be obtained from the representative of its class. Since it is a vector quantity, however, it needs to be transformed via

$$ b(F) = gb(F_{d-1}^{\text{rep}}), $$

(45)

where $F = gF_{d-1}^{\text{rep}}$ and $F_{d-1}^{\text{rep}} \in F_{d-1}^{\text{rep}}$. The second moment tensor [7] can be calculated using

$$ U(F_d) = \frac{1}{d+2} \sum_{F \in C(F_d)} h_b(F) \left[ U(F) + (\Delta b_F)^T \Delta b_F \text{Vol}(F) \right]. $$

(46)

where $h_b(F)$ is the height of the barycenter $b(F)$ above the plane of $F$ and we use the abbreviation $\Delta b_F \equal b(F_d) - b(F)$. The heights $h_b(F)$ are calculated in the same way as $h_c(F)$.

The second moment tensors $U(F)$ of child faces are obtained from those of their representative faces as

$$ U(F) = M_g^T U(F_{d-1}^{\text{rep}}) M_g, $$

(47)

where, as in [12], $M_g^T$ is the matrix representing the transformation $g \in G$ on $\mathbb{R}^n$.

Finally, taking the trace of (46) provides a recursion for the scalar second moment

$$ U(F_d) = \frac{1}{d+2} \sum_{F \in C(F_d)} h_b(F) \left[ U(F) + \| \Delta b_F \|^2 \text{Vol}(F) \right]. $$

(48)

Figure 7: Classes used for implementing the calculations. Each “FaceProperties” instance stores a calculator used for basic operations that make use of some symbolic (e.g., SymPy [28]) or numerical (e.g., NumPy [36]) library. This makes the implementations of the second moment, volume, and the other properties independent of the underlying library. For certain dimensions, there are specializations of “FaceProperties” implemented as subclasses. See the main text for details.
class.

4 Analyzing families of lattices

In this section, we apply our algorithm to calculate the second moment \( G \) of an individual lattice and to minimize \( G \) in a one-parameter family of lattices. In 9 dimensions, such an optimization problem has led to the conjectured optimal lattice quantizer \( AE_9 \) [5], which is obtained by laminating the root lattice \( D_8 \). We will first review the closely related construction of product lattices and then the lamination procedure. This is then applied to optimize a family of 13-dimensional lattices.

Product lattices are analyzed in detail in [1]. If \( \Lambda_1 \) and \( \Lambda_2 \) are lattices of dimension \( n_1 \) and \( n_2 \), respectively, then the one-parameter family of product lattices of dimension \( n = n_1 + n_2 \) is defined via

\[
\Lambda(a) \triangleq \Lambda_1 \times a\Lambda_2,
\]

where \( a > 0 \) is a real parameter. Let \( G(a) \) be the quantizer constant of \( \Lambda(a) \) and let further \( \mathbf{B}_1, V_1, \) and \( G_1 \) be, respectively, a generator matrix, the volume, and the quantizer constant of \( \Lambda_1 \). Then a generator matrix for \( \Lambda(a) \) is

\[
\mathbf{B}(a) \triangleq \begin{bmatrix} 1 & 0 \\ 0 & a\mathbf{B}_2 \end{bmatrix}.
\]

The quantizer constant of \( \Lambda(a) \) is minimal when [1, Corollary 5]

\[
a = a_{\text{opt}} \triangleq \sqrt{\frac{V_1}{V_2}} \sqrt{\frac{G_1}{G_2}},
\]

for which \( G(a) \) satisfies

\[
G^n(a_{\text{opt}}) = G_1^n G_2^n.
\]

From a collection of different lattices with known values for \( G \), Agrell and Allen form more than 30 product lattices whose quantizer constants are smaller than any previously published ones in their respective dimension. Additional improved product lattices are constructed in [25] in dimensions 15, 19, 21, and 23. Starting at low dimensions \( n \), the first one found in [1] is a product of the Coxeter–Todd lattice \( K_{12} \) and the one-dimensional lattice \( \mathbb{Z} \), i.e., \( \Lambda = K_{12} \times a\mathbb{Z} \), which has

\[
G(a_{\text{opt}}) \approx 0.071034583.
\]

The following key result motivates investigating \( K_{12} \times a\mathbb{Z} \) further. If \( \Lambda_1 \) and \( \Lambda_2 \) are two lattices with generator matrices \( \mathbf{B}_1 \) and \( \mathbf{B}_2 \) as before, then we define \( \Lambda_{\mathbf{H}}(a) \) as the lattice generated by

\[
\mathbf{B}_{\mathbf{H}}(a) \triangleq \begin{bmatrix} \mathbf{B}_1 & 0 \\ \mathbf{H} & a\mathbf{B}_2 \end{bmatrix},
\]

where \( \mathbf{H} \) is an \( n_2 \times n_1 \) matrix. Let \( G_{\mathbf{H}}(a) \) be the quantizer constant of \( \Lambda_{\mathbf{H}}(a) \). Then it is shown in [1, Theorem 7] that

\[
G_{\mathbf{H}}(a) \leq G(a), \quad \forall \mathbf{H}.
\]

This means that in general, any product lattice [49] can be further optimized by taking \( \mathbf{H} \neq 0 \). In particular, it should be possible to get better than [53] in 13 dimensions.

The special case of the construction (54) with \( n_2 = 1 \) (i.e., \( \Lambda_2 = \mathbb{Z} \)) is called a lamination of \( \Lambda_1 \). For that case, we will write

\[
\mathbf{B}_h(a) \triangleq \begin{bmatrix} \mathbf{B}_1 & 0 \\ \mathbf{h} & a \end{bmatrix},
\]

where \( \mathbf{h} \in \mathbb{R}^{n_1} \) is the offset vector. We will henceforth assume that \( \mathbf{h} \) is fixed and drop the subscript of the quantities \( \mathbf{B} \) and \( G \) of the laminated lattice \( \Lambda(a) \).

Note that for \( \mathbf{h} \neq 0 \), we no longer have a general closed form expression for the optimal value \( a_{\text{opt}} \) that minimizes \( G(a) \). For example, \( AE_9 \) is obtained by laminating \( D_8 \) with \( \mathbf{h} \) chosen as a vertex of the Voronoi cell of \( D_8 \) that is most distant from \( 0 \). Such a vertex is called a deep hole. It is shown in [5] that the optimal value of \( a = 0.573\ldots \) is an algebraic number whose square is a root of a 9th order polynomial.

In the following, we will try a similar strategy in 13 dimensions and laminate \( K_{12} \) in the direction of a deep hole. This will require calculating \( G \) as a function of \( a \) and then finding the minimum of \( G(a) \). Our method for doing so is discussed in Sec. 4.2. However, in order to verify our code and to explore the possible symmetries of laminated \( K_{12} \), we shall first apply our algorithm to \( K_{12} \) itself.

4.1 The Coxeter–Todd lattice \( K_{12} \)

The Coxeter–Todd lattice \( K_{12} \) is a well studied lattice that was first described by Coxeter and Todd in 1953 [17]. Its quantizer constant was calculated numerically in [14] and exactly in [21]. Until better quantizers were recently discovered in [4], it was conjectured to be the optimal quantizer in 12 dimensions [15, p. 13]. Note that some of the properties we state below were not previously published and may be useful as verification of potential future methods.

A square generator matrix for \( K_{12} \) is [15, Section 4.9]

\[
\mathbf{B} \triangleq \begin{bmatrix} 2\mathbf{A} & 0 & 0 & 0 & 0 & 0 \\ 0 & 2\mathbf{A} & 0 & 0 & 0 & 0 \\ 0 & 0 & 2\mathbf{A} & 0 & 0 & 0 \\ \mathbf{A} & \mathbf{W} & \mathbf{W} & \mathbf{A} & 0 & 0 \\ \mathbf{W} & \mathbf{A} & \mathbf{W} & 0 & \mathbf{A} & 0 \\ \mathbf{W} & \mathbf{W} & \mathbf{A} & 0 & 0 & \mathbf{A} \end{bmatrix},
\]

where

\[
\mathbf{A} \triangleq \frac{1}{2} \begin{bmatrix} 2 & 0 \\ -1 & \sqrt{3} \end{bmatrix}, \quad \mathbf{W} \triangleq \frac{1}{2} \begin{bmatrix} -1 & \sqrt{3} \\ -1 & -\sqrt{3} \end{bmatrix}.
\]
Table 2: Representatives of the relevant vectors $\mathbf{n}_i \in \mathcal{N}(\Omega)$ (first two rows) and vertices $\mathbf{v}_i \in \mathcal{V}(\Omega)$ (remaining eight rows) of the Voronoi cell of $K_{12}$ in order of decreasing length. Shown are the components, squared lengths, and sizes of the orbits under $G$. The vertex $\mathbf{v}_1$ has the largest squared length and thus lies furthest from any lattice point. It is used in Sec. 4.2 as the offset vector for laminating $K_{12}$.

| vector | components | $\| \cdot \|^2$ | orbit size |
|--------|------------|----------------|-------------|
| $\mathbf{n}_1$ | $\frac{1}{2} (0 \ 0 \ 0 \ 0 \ 2\sqrt{3} \ 1 \ \sqrt{3} \ 1 \ \sqrt{3} \ 2 \ 0)$ | 6 | 4032 |
| $\mathbf{n}_2$ | $(0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ -\sqrt{3})$ | 4 | 756 |
| $\mathbf{v}_1$ | $\frac{1}{4} (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 2\sqrt{3} \ 0 \ 2\sqrt{3})$ | 4 | 20412 |
| $\mathbf{v}_2$ | $\frac{1}{3} (0 \ 0 \ 0 \ 0 \ 3 \ -19\sqrt{3} \ 6 \ -12\sqrt{3} \ 12 \ 2\sqrt{3} \ 9 \ -5\sqrt{3})$ | 5 | 108864 |
| $\mathbf{v}_3$ | $\frac{1}{15} (0 \ 0 \ 0 \ 0 \ 4\sqrt{3} \ 6 \ 4\sqrt{3} \ 12 \ -6\sqrt{3} \ 9 \ -\sqrt{3})$ | 12 | 653184 |
| $\mathbf{v}_4$ | $\frac{1}{15} (0 \ 0 \ 0 \ 0 \ 4\sqrt{3} \ 3 \ 5\sqrt{3} \ 15 \ 3\sqrt{3} \ 9 \ \sqrt{3})$ | 15 | 653184 |
| $\mathbf{v}_5$ | $\frac{1}{8} (0 \ 0 \ 0 \ 0 \ 3 \ -5\sqrt{3} \ 3 \ \sqrt{3} \ 3 \ -3\sqrt{3} \ 3 \ \sqrt{3})$ | 8 | 326592 |
| $\mathbf{v}_6$ | $\frac{1}{8} (0 \ 0 \ 0 \ 0 \ 2\sqrt{3} \ 3 \ 3\sqrt{3} \ 6 \ -4\sqrt{3} \ 6 \ 0)$ | 8 | 3265920 |
| $\mathbf{v}_7$ | $\frac{1}{6} (0 \ 0 \ 0 \ 0 \ 2\sqrt{3} \ 3 \ -\sqrt{3} \ 3 \ 3\sqrt{3} \ 9 \ 3\sqrt{3})$ | 6 | 4032 |
| $\mathbf{v}_8$ | $\frac{1}{6} (0 \ 0 \ 0 \ 0 \ 2\sqrt{3} \ 3 \ -\sqrt{3} \ 3 \ -\sqrt{3} \ 6 \ 0)$ | 6 | 4032 |

Its symmetry group is described in [15, Section 4.9] and [16] and it has order $|G| = 78\,382\,080$. These references also discuss the 4788 relevant vectors and the deep holes. Tab. 2 lists the two representatives $\mathbf{n}_1$, $\mathbf{n}_2$ of the relevant vectors and the single representative deep hole $\mathbf{v}_1$ along with representatives $\mathbf{v}_2, \ldots, \mathbf{v}_8$ of all other vertices of the Voronoi cell of $K_{12}$. We obtained the full set of 5,685,372 vertices using the method discussed in Sec. 2.3.

With the help of GAP, we determined three matrices that together generate the full group $G$, namely

\[
M_1 \triangleq \begin{bmatrix} 0 & I_2 & 0 & 0 & 0 & 0 & 0 \\ I_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & I_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & I_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & I_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & I_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & S' \\ 0 & 0 & 0 & 0 & 0 & S & 0 \end{bmatrix},
\]

\[
M_2 \triangleq \begin{bmatrix} 0 & 0 & 0 & 0 & S & 0 & 0 \\ 0 & 0 & 0 & S & 0 & 0 & 0 \\ 0 & 0 & S & 0 & 0 & 0 & 0 \\ S & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},
\]

\[
M_3 \triangleq \frac{1}{2} \begin{bmatrix} I_2 & V & -I_2 & 0 & V & 0 \\ V^T & I_2 & V^T & 0 & -I_2 & 0 \\ -I_2 & V & I_2 & 0 & V & 0 \\ 0 & 0 & 0 & 2I_2 & 0 & 0 \\ V^T & -I_2 & V^T & 0 & I_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2I_2 \end{bmatrix}, \quad (59)
\]

where $I_k$ is the $k \times k$ identity matrix and

\[
S \triangleq \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad V \triangleq \frac{1}{2} \begin{bmatrix} 1 & \sqrt{3} \\ -\sqrt{3} & 1 \end{bmatrix}. \quad (60)
\]

The construction of the hierarchy of faces is done as described in Sec. 2. We employ the iterated classification using proper subgroups $\mathcal{U}_i \subset G$ followed by the full group $G$. One could easily obtain such subgroups by taking any proper subset of the three generator matrices $\{M_1, M_2, M_3\}$ as generators of a subgroup. However, the subgroups generated by a single of these matrices each have order 2, while the subgroups generated by any pair of matrices have orders 8 ($M_1$ and $M_2$) or 12 (the other pairs). These subgroups are too small to efficiently reduce the number of faces by consolidating them into classes.

Larger subgroups $\mathcal{U}_i \subset G$ are, e.g., the stabilizers of relevant vectors—and thus of the facets—or more generally of any subface $F_d$. We chose the stabilizers of the two representative facets, i.e., of $\mathbf{n}_1$ and $\mathbf{n}_2$ shown in Tab. 2 with sizes 19,440 and 103,680, respectively. Using these two subgroups and $G$, we classify the children of the representative facets, yielding six classes of 10-faces. The stabilizers of these six representatives have sizes 240, 480, 648, 1,296, 2,592, and 103,680, where we note that the two subgroups of size 103,680 are indeed distinct. We now have eight subgroups $\mathcal{U}_i$ and the full group $G$ with which to perform the iterated classification in all lower dimensions.

The resulting face hierarchy contains 809 classes of faces. From dimension 1 through 12, the number of classes is 8, 22, 48, 93, 149, 185, 154, 86, 40, 15, 6, 2, and 1, respectively.

We calculate an exact value for the quantizer constant of $K_{12}$,

\[
G = \frac{797\,361\,941}{6\,567\,561\,000\sqrt{3}} \approx 0.0700956, \quad (61)
\]

which agrees with the value reported in [21]. We also
calculate the (unnormalized) second moment tensor
\[ U = \frac{797,361,941}{243,243,000} I_{12} \approx 3.2780468 I_{12} \tag{62} \]
Recall that due to (4), a locally optimal lattice quantizer necessarily has a second moment tensor proportional to the identity matrix. Therefore, (62) is compatible with \( K_{12} \) being a local optimum, although we know from (4) that it is not globally optimal.

A comprehensive catalog of all 809 face classes with exact expressions for volumes, second moment scalars and hierarchical information is available as a supplementary online resource [44].

4.2 Laminated \( K_{12} \)

In this section, we apply our algorithm to a lamination \( \Lambda(a) \) of \( K_{12} \). We take as offset vector \( h \) a deep hole of \( K_{12} \), \( v_1 \) (see Tab. 2), i.e., we have the generator matrix
\[ B(a) = \begin{bmatrix} B_1 & 0 \\ v_1 & a \end{bmatrix}, \tag{63} \]
where \( B_1 \) is the \( 12 \times 12 \) generator matrix \([57]\) of \( K_{12} \).

Note that it is unclear whether our choice of \( h = v_1 \) is optimal. However, as we shall see below, with an optimal \( a \) this choice leads to a second moment tensor \( U \) that is proportional to the identity matrix, supporting that \( \Lambda(a) \) might at least be a locally optimal lattice quantizer.

Since we are interested in finding the minimum of \( G \), we begin our analysis by numerically estimating \( G(a) \) via Monte Carlo integration for several values of \( a \). A resulting rough estimate \( a_0 \approx a_{\text{opt}} \) is then used for constructing the Voronoi cell (Sec. 2) and performing the calculations (Sec. 3). In the present case of laminated \( K_{12} \), we used \( a_0 = 34/33 \approx 1.03 \).

We next carry out the full analysis at \( a_0 \) and determine \( G \) as well as the volumes \( \text{Vol}(F_d) \) of all faces \( F_d \) as functions of \( a \). This can be done via symbolic calculations of the equations in Sec. 3. We remark that instead of symbolically calculating with an unknown \( a \), one may speed up the calculations substantially by substituting an (exact) rational value for \( a \). Following the discussion in Sec. 6, we used \( n + 3 \) rational values close to \( a_0 \) to initially determine \( G(a) \) and later verified our results with full symbolic calculation with unknown \( a \).

At this point, we do not yet know for which values of \( a \) these results hold. As \( a \) changes, vertices may merge or split, resulting in changes to the hierarchy of faces. At these critical values of \( a \), not only may the functional dependence of \( G \) on \( a \) change (phase transition), the whole data structure we build for one value of \( a \) becomes invalid.

We perform the following steps to determine the domain of \( a \) where our results apply. This is done after the data structure has been calculated at \( a = a_0 \) and the volumes of all \( d \)-faces, \( d = 0, \ldots, n \), have been determined as function of \( a \). First, we find an interval \( I = (a_-, a_+) \) around \( a_0 \) where the expressions for all the volumes evaluate to strictly positive values. Our data structure evaluated at any \( a \in I \) then represents a convex polytope \( P(a) \) with faces having positive volumes. In particular, the volume \( \text{Vol}(P(a)) \) is correctly calculated by (41). We have \( P(a_0) = \Omega(a_0) \), where \( \Omega(a) \) is the Voronoi cell of \( \Lambda(a) \), but away from \( a_0 \), \( P(a) \) does not need to agree with \( \Omega(a) \). In principle, vertices of \( \Omega(a) \) can split in this interval and new faces may appear, so that the vertices of \( P(a) \) need not coincide with those of \( \Omega(a) \) for all \( a \in I \).

Therefore, we additionally check the following two conditions: (i) The set of relevant vectors found at \( a_0 \) is the (full) set of relevant vectors of \( \Omega(a) \) for any \( a \in I \). (ii) The representative vertices are still inside \( \Omega(a) \) and contained in the same facets as at \( a = a_0 \). This is verified using (9) and (10). Then, \( P(a) \subseteq \Omega(a) \) and if furthermore \( \text{Vol}(P(a)) = | \det B | \), then clearly \( P(a) = \Omega(a) \).

For laminated \( K_{12} \) analyzed at \( a_0 = 34/33 \approx 1.03 \), we find vanishing volumes of 1-faces of \( P(a) \) at \( a = 1 \) and \( a = \sqrt{17/15} \approx 1.06 \). Faces have non-vanishing volumes between these two values and the calculated volume of \( \Omega \) is always \( | \det B | = 27a \). Also, the additional conditions (i) and (ii) mentioned above are satisfied in that range. This means that the formulas we report below are valid for
\[ 1 \leq a \leq \sqrt{17/15}. \tag{64} \]

The analysis itself starts by finding all the relevant vectors, which is step 1 of Sec. 2.1 Using the algorithm presented in [3], we obtain a set of 7706 vectors \( \mathcal{N}(\Omega) \). At this point, they are not yet classified, since we do not know the symmetry group \( \mathcal{G} \) of laminated \( K_{12} \).

However, the relevant vectors themselves help in finding \( \mathcal{G} \) as follows. We take the symmetry group \( \mathcal{G}_{12} \) of \( K_{12} \) and embed it in \( O(13) \) via
\[ \mathcal{G}_{12} \to 13 \triangleq \left\{ \begin{bmatrix} M_0^T & 0 \\ 0 & 1 \end{bmatrix} : g \in \mathcal{G}_{12} \right\}, \tag{65} \]
where, as before, \( M_0^T \) is the \( 12 \times 12 \) matrix representing \( g \in \mathcal{G}_{12} \). This group is generated by the matrices
\[ \begin{bmatrix} M_1^{(12)} & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} M_2^{(12)} & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} M_3^{(12)} & 0 \\ 0 & 1 \end{bmatrix}, \tag{66} \]
where \( M_i^{(12)} \) are the generators \([59]\) of \( K_{12} \). We add a fourth generator
\[ \begin{bmatrix} I_{12} & 0 \\ 0 & -1 \end{bmatrix} \tag{67} \]
to create a group \( \mathcal{G} \) of twice the order of \( \mathcal{G}_{12} \). Then, we use GAP to compute the stabilizer
\[ \mathcal{G} \triangleq \left\{ g \in \mathcal{G} : g \mathcal{N}(\Omega) = \mathcal{N}(\Omega) \right\}. \tag{68} \]
of the set of relevant vectors of $\Omega$ and produce a generating set for $G$. This way, we obtain the three symmetries

$$M_1 \triangleq \frac{1}{2} \begin{bmatrix} 0 & I_2 & 0 & V & -I_2 & -V & 0 \\ 0 & -V & V & 0 & -I_2 & I_2 & 0 \\ 2V^T & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -I_2 & V & V & I_2 & 0 \\ 0 & I_2 & V & -I_2 & V & 0 & 0 \\ 0 & V & I_2 & I_2 & 0 & V & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix},$$

$$M_2 \triangleq \frac{1}{2} \begin{bmatrix} 0 & -S & 0 & Y & -S & Y & 0 \\ 2S & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & Y' & 0 & 0 & -S & -Y' & S \\ 0 & 0 & -2S & 0 & 0 & 0 & 0 \\ 0 & Y' & 0 & S & -Y' & -S & 0 \\ 0 & -S & 0 & -Y & S & Y & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix},$$

$$M_3 \triangleq \begin{bmatrix} I_8 & 0 \\ 0 & -I_5 \end{bmatrix},$$

where $S$ and $V$ are given in (60) and

$$Y \triangleq \frac{1}{2} \begin{bmatrix} -1 & -\sqrt{3} & 0 \\ -\sqrt{3} & 1 & 0 \end{bmatrix}, \quad Y' \triangleq \frac{1}{2} \begin{bmatrix} -1 & \sqrt{3} & 0 \\ \sqrt{3} & 1 & 0 \end{bmatrix}.$$  

(72)

This group $G$, generated by $M_1$, $M_2$, and $M_3$, has order 622,080 and we use it as the symmetry group of laminated $K_{12}$ for the next steps. In principle, more symmetries might exist, in particular for specific values of $a$, resulting in a larger symmetry group. However, we were not able to find any for generic $a$.

Using $G$, the 7,706 relevant vectors are partitioned into 8 classes, representatives of which are listed in Tab. 3. The search for vertices of $\Omega$ (step 2) is done as described in Sec. 2.3 and yields 52,351,632 vertices in 482 classes.

As for $K_{12}$, we employ the iterated classification method and therefore need subgroups $U_i$ of $G$. The subgroups generated by only $M_1$, $M_2$, and $M_3$ have orders 30, 10, and 2, respectively. Those generated by the pairs $\{M_1, M_2\}$ and $\{M_1, M_3\}$ both have order 311,040, while the pair $\{M_2, M_3\}$ generates a group of order 320. Again, instead of these, we use the stabilizers of the representative faces and $(n-2)$-faces as subgroups $U_i$. The 8 facets (see Tab. 3) have stabilizer sizes ranging from 240 to 3,840 and the 81 faces of dimension 11 have stabilizers with sizes between 4 and 3,840. Of these in total 89 subgroups, we selected 13 with sizes 12, 48, 480, 1,152, 1,920, 2,880 (twice), and 3,840 (six times) and used these for the iterated classification. We remark that the decision on how to construct and choose the subgroups can likely be optimized further.

Our construction of the hierarchy of faces resulted in 430,051 classes. From dimensions 0 to 13, the number of classes is 482, 3,599, 15,656, 45,473, 87,511, 110,578, 92,074, 50,820, 18,590, 4,477, 701, 81, and 1.

The total number of face classes of laminated $K_{12}$ is much larger than for $K_{12}$ (which has 809 face classes) in part due to the much smaller symmetry group. If the group were larger, more faces would potentially be equivalent. On the other hand, the smaller symmetry group also leads to more but smaller subsets $X_i$ of the defining sets $D(F_i)$ of vectors we use for the classification (see Sec. 2.5.3 and 2.5.5). This reduces the classification cost for an individual face. However, the much larger number of classes in the end leads to a much more expensive construction of the face hierarchy than for $K_{12}$.

For a full catalog of face classes, the reader is again referred to the supplementary online material [44], where we list properties and hierarchical information of all face classes.

Using the formulas of Sec. 3, the unnormalized second moment $U$ as a function of $a$ is

$$U = \frac{18347048081953}{2701535532000} - \frac{13214046803972}{5741014400} + \frac{993489988977}{184704898757} = \frac{621694889957}{4702948000} + \frac{390355352800}{14108774400} - \frac{136946795267}{8843457559120000},$$

(73)

With the volume $\text{Vol}(\Omega) = 27a$ and using $E = U/\text{Vol}(\Omega)$ it is easy to calculate $G(a)$ via (2). The condition $G'(a) = 0$ can be turned into a polynomial equation $f(v) = 0$, where $v \triangleq a^2$ and

$$f(v) \triangleq \frac{1239510953a^{14}}{41534172000} - \frac{13694679267a^{13}}{2988699840} + \frac{110439594323a^{12}}{344834115200} - \frac{232261582151a^{11}}{17240657600} + \frac{5355403365271a^{10}}{8405264600} - \frac{6089067326571a^9}{14108774400} + \frac{20641917523a^8}{1881199920} - \frac{7501504391287a^7}{6394984800} + \frac{228910566635a^6}{3212426756633a^5} - \frac{124535835146037a^4}{3428342179200} + \frac{3914170754080a^3}{2513056574800} - \frac{61858884194560a^2}{8843457559120000} + \frac{99697144277617}{23085368000} - \frac{796717144277617}{23085368000}.$$

(74)

If $v_0$ denotes the smallest positive root of $f$, then the minimum of $G(a)$ is attained at

$$a_{\text{opt}} = \sqrt{v_0} \approx 1.0149980107,$$

which lies in the range $[64]$. The resulting value of the quantizer constant is

$$G(a_{\text{opt}}) \approx 0.0699012856.$$
This lies well below the currently known best second moment $\frac{1}{53}$ in 13 dimensions [1] Table I, [25] Table I].

The second moment tensor is the diagonal $13 \times 13$ matrix

$$U = \alpha(a) I_{13} + \beta(a) Z_{13},$$

(77)

where $Z_{13} = \text{Diag}(0, \ldots, 0, 1)$,

$$\alpha(a) = \frac{1239510953a^{27}}{81885808000} + \frac{4417638557a^{25}}{135862272000} - \frac{155482113a^{23}}{65978203040} + \frac{5404897975a^{21}}{98019200} - \frac{18447040154a^{19}}{7054387200} + \frac{62169488057a^{17}}{12093235200} - \frac{2064198597a^{15}}{28217548800} + \frac{88885278720}{648936908800} + \frac{283204405037a^5}{4288621452800} + \frac{164338533146037a^5}{33941478540800} + \frac{2085093939502000}{79428305983447200000},$$

(78)

and

$$\beta(a) = \frac{f(a^2)}{a}.$$ 

(79)

As was done for $G(a)$ initially, we obtained (86) by following [5, Sec. 6] to infer $U$ as a function of $a$ from $n + 3$ exact results for rational $a$.

Note that

$$\beta(a) \propto a^{28} G'(a),$$

(80)

which indicates that $U$ is proportional to the identity matrix if and only if $G'(a) = 0$. This is satisfied at $a = a_{\text{opt}}$.

5 Conclusions

For a lattice whose symmetry group is known, the algorithm presented in this work provides a way to explicitly construct a part of the full face hierarchy of its Voronoi cell. This partial structure contains sufficient information for exactly calculating, among other properties, its quantizer constant $G$. In addition, one obtains a full classification of all faces of the cell, despite most faces never being constructed in this process.

A key property of our approach is the possibility to parallelize the algorithm, which enables the work to be distributed over many processor cores. By carefully planning the steps needed to evaluate the recursion formulas for $G$, parallelization can be employed not only during construction of the face hierarchy, but also in the subsequent computations.

We have applied our algorithm to reproduce the results known for $AE_9$ and $K_{12}$ and then analyzed a new 13-dimensional family of lattices obtained by laminating $K_{12}$. This led to a new currently best known lattice quantizer in 13 dimensions. With 430,051 different classes of faces, analyzing this family with our method proved to still be computationally feasible.

The most expensive part is determining whether two faces are equivalent under the lattice’s symmetry group and obtaining a group element transforming one into the other. If a more efficient method can be found for this step, then most of the strategy discussed in this paper is still applicable and may enable the analysis of lattices in significantly higher dimensions.

Although the algorithm presented here builds upon that developed for the analysis of $AE_9$ [5], it contains major improvements in several key aspects. Most significantly, it exploits the symmetries of a lattice during construction of the face hierarchy, which is not done in [5]. This has been crucial for the analysis in higher dimensions. For example, for $K_{12}$, only about 15,000 of the in total 11,971,907,593 faces are constructed in the new method. It also improves upon [5] by utilizing the translational symmetries of the lattice in the search for vertices.

Some of our methods have similarities with those of Dutour Sikirić, Schürmann, and Vallentin [21]. Both use the automorphisms combined with the translational symmetries of a lattice to find representatives of all vertices of the Voronoi cell. Furthermore, both methods calculate
the scalar second moment and the second moment tensor (which in [21] p. 1725) is called $L_2(P)$.

There are also a number of differences. One example is the already-discussed different method to test for the equivalence of faces. Another example is our search for vertices. We find all representatives of vertices via a random search using a carefully chosen distribution of initial vectors. This is optimized to run many times by pre-computing all relevant vectors and using these to find a vertex in a single linear program. In contrast, [21] finds one random vertex using multiple linear programs for vertices. We find all representatives of vertices via a random search using multiple linear programs without them. On an implementation level, a major difference is that we use floating point operations to speed up certain steps, although our method does not depend upon floating point operations and can be implemented without them. On an implementation level, a major difference is that our method can be used with families of lattices parametrized by one or more parameters.

In principle, our algorithm could be improved by adopting some of the methods of [21]. By selectively performing a simplicial decomposition as well as incorporating the translational symmetries in the classification of faces, the number of face classes to construct can be reduced. This may lead to a lower computational cost for a given lattice, at the cost of potentially more expensive equivalence tests. We leave this possible extension of our method to future work.

References

[1] E. Agrell and B. Allen. “On the best lattice quantizers”. In: IEEE Trans. Inf. Theory 69.12 (2023), pp. 7650–7658. DOI: 10.1109/TIT.2023.3291313

[2] E. Agrell and T. Eriksson. “Optimization of lattices for quantization”. In: IEEE Trans. Inf. Theory 44.5 (1998), pp. 1814–1828. DOI: 10.1109/18.705561.

[3] E. Agrell et al. “Closest point search in lattices”. In: IEEE Trans. Inf. Theory 48.8 (2002), pp. 2201–2214. DOI: 10.1109/TIT.2002.800499.

[4] E. Agrell, D. Pook-Kolb, and B. Allen. Glued lattices are better quantizers than $K_{12}$. 2023. arXiv: 2312.0048

[5] B. Allen and E. Agrell. “The optimal lattice quantizer in nine dimensions”. In: Annalen der Physik 533.12 (2021), p. 2100259. URL: https://doi.org/10.1002/andp.202100259

[6] B. Allen. “Optimal template banks”. In: Phys. Rev. D 104 (2021), p. 042005. DOI: 10.1103/PhysRevD.104.042005.

[7] B. Allen. “Performance of random template banks”. In: Phys. Rev. D 105 (2022), p. 102003. DOI: 10.1103/PhysRevD.105.102003.

[8] B. Allen and A. A. Shoom. “Template banks based on $Z^r$ and $A_r$ lattices”. In: Phys. Rev. D 104 (2021), p. 122007. DOI: 10.1103/PhysRevD.104.122007.

[9] D. Avis, D. Bremner, and A. Deza. Polyhedral Computation. Vol. 48. American Mathematical Society (AMS), 2009. DOI: 10.1090/crm/048.

[10] E. S. Barnes and N. J. A. Sloane. “The optimal lattice quantizer in three dimensions”. In: SIAM J. Alg. Disc. Meth. 4.1 (1983), pp. 30–41. URL: https://doi.org/10.1137/0604005.

[11] E. M. Bray. gappy – a Python interface to GAP, Version 0.1.0a4. URL: https://github.com/embray/gappy.

[12] D. Bremner, M Dutour Sikirić, and A. Schürmann. “Polyhedral representation conversion up to symmetries”. In: CRM Proceedings and Lecture Notes. Vol. 48. American Mathematical Society, 2009, pp. 45–72.

[13] J. H. Conway and N. J. A. Sloane. “A lower bound on the average error of vector quantizers”. In: IEEE Trans. Inf. Theory IT-31.1 (1985), pp. 106–109. DOI: 10.1109/TIT.1985.1066993.

[14] J. H. Conway and N. J. A. Sloane. “On the Voronoi regions of certain lattices”. In: SIAM J. Alg. Disc. Meth. 5.3 (1984), pp. 294–305. URL: https://doi.org/10.1137/0605031.

[15] J. H. Conway and N. J. A. Sloane. Sphere Packings, Lattices and Groups. 3rd. New York, NY: Springer, 1999. URL: https://doi.org/10.1007/978-1-4757-6568-7.

[16] J. H. Conway and N. J. A. Sloane. “The Coxeter–Todd lattice, the Mitchell group, and related sphere packings”. In: Mathematical Proceedings of the Cambridge Philosophical Society. Vol. 93. Cambridge University Press, 1983, pp. 421–440. DOI: 10.1017/S030500410006076.

[17] H. S. M. Coxeter and J. A. Todd. “An extreme dually convex form”. In: Canadian Journal of Mathematics 5 (1953), pp. 384–392. DOI: 10.4153/CJM-1953-043-4.

[18] M. Deza and M. Dutour Sikirić. “Enumeration of the facets of cut polytopes over some highly symmetric graphs”. In: International Transactions in Operational Research 23.5 (2016), pp. 853–860. DOI: 10.1111/itor.12194.

[19] M. Dutour Sikirić. “The Birkhoff polytope of the groups $F_4$ and $H_4$”. In: Proceedings of the 4th Croatian Combinatorial Days. Ed. by T. Došljić, S. Majstorović, and L. Podrug. Faculty of Civil Engineering, University of Zagreb, 2022, pp. 21–26. DOI: 10.5592/CD/CD.2022.03.

[20] M. Dutour Sikirić. Polyhedral, a C++ package. 2020. URL: https://github.com/MathieuButSik/polyhedra1_common.
