GENERATION OF 2D AND 3D QUASICRYSTALLINE STRUCTURES USING THE PROJECTION METHOD

by

Peker Milas

B.S., Physics, Boğaziçi University, 2003

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Physics
Boğaziçi University

2006

GENERATION OF 2D AND 3D QUASICRYSTALLINE STRUCTURES USING THE PROJECTION METHOD

APPROVED BY:

Assist. Prof. Muhittin Mungan (Thesis Supervisor)	.....
Prof. Metin Arık	.....
Prof. Avadis Hacınlıyan	.....

DATE OF APPROVAL: 13.06.2006

ACKNOWLEDGEMENTS

Right now, I do not want to judge anybody (with any reason). Yet there is no one that I owe to represent my appreciations other than my sweet darling, my family and my best friend Burak. I believe that they deserve much more than these words as a gratitude.

First, I would like to thank to my darling Figen for her encouraging support. I was able to overcome these hard times with her endless love and mental support.

I am indebted to all of my family because of their patience and their encouragment for my work. They did not leave me alone with any kind of trouble during the past two years. They constantly tried to help me by all means.

I want to thank Burak because of his great friendship. I can not forget him and his encouragements. We were constantly in trouble for the past two years but this is the end of the hard days. I hope that the days in University of Iowa will be so nice for both him and his family.

At the end, I wish that my mom was here and could see his son earning his Ms. degree. I miss her so much. So, I gladly dedicate this thesis to my love, my mom, my dad and my brothers. They are simply the reason to live and the meaning of life.

ABSTRACT

GENERATION OF 2D AND 3D QUASICRYSTALLINE STRUCTURES USING THE PROJECTION METHOD

Two algorithms for generating 2D and 3D quasiperiodic lattices by the strip projection method are presented. For the 2D algorithm a 5D simple cubic integral lattice with unit lattice constant is used as root lattice with a rhombic icosahedron as the projection window. For the 3D case a 6D simple cubic integral lattice with unit lattice constant is used as root lattice with a rhombic triacontahedron as the projection window.

ÖZET

İKİ VE ÜÇ BOYUTLU YARİDÜZENLİ ÖRGÜLERİN İZDÜŞÜM YÖNTEMİYLE ELDE EDİLMESİ

İki ve üç boyutlu yarıdüzenli kristal yaplar, yüksek boyutlu düzenli kristal yapıların düsük boyutlara indirgenmesi yöntemiyle incelendi. Yöntemde, yüksek boyutlu düzenli kristal yapılarının yanında iz düşüm operatörleri ve iz düşüm için gerekli olan üç boyutlu pencereler yapısal olarak incelendi.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES xi
LIST OF SYMBOLS xii
1. INTRODUCTION 1
2. THEORY 2
2.1. Basics of Convetional Crystallography 2
2.2. Lattices, Dual Lattices, and The Diffraction Patterns 5
2.2.1. Lattices 5
2.2.2. Dual Lattices 6
2.2.3. The Diffraction Patterns 10
2.2.3.1. Bragg Formulation of X-ray Diffraction 10
2.2.3.2. The von Laue Formulation of X-ray Diffraction 10
2.3. Unit Cell Phenomena 12
2.3.1. Delone Cell 13
2.3.2. Voronoï Cell 17
2.4. Symmetry And Group Properties 21
2.4.1. Symmetry 21
2.4.2. Symmetry Groups 24
2.4.3. The Crystallographic Restriction 26
2.4.4. Orbits of Z-modules 28
2.5. Canonical Projection Method 29
3. QUASICRYSTALS IN ONE AND TWO DIMENSIONS 32
3.1. Quasicrystals in 1-D (Fibonacci Chain) 32
3.2. Quasicrystals in 2-D (Penrose rhombus tiling) 33
4. GENERATION AND STRUCTURAL ANALYSIS OF THREE DIMENSIONAL ICOSAHEDRAL QUASICRYSTAL 40

4.1. Quasicrystals in 3-D (The $\mathcal{I}_6$ quasicrystal) 40
4.2. Structural Properties of the Generated Quasicrystal 48
4.2.1. Layers and The Layer Densities 49
4.2.2. Bulk and Plane Coordination Numbers 51
5. CONCLUSIONS 57
APPENDIX A: ALGORITHMS 60
A.1. *Penrose.c* 60
A.2. *Icosahedral.c* 71
A.3. *Rotator.c* 82
APPENDIX B: ANALYSIS OF THE CODES 84
B.1. Penrose.c 84
B.2. Icosahedral.c 85
B.3. Rotator.c 86
REFERENCES 87

LIST OF FIGURES

Figure 2.1. Portions of orbits generated by (a) $\vec{b}_3 = (\frac{1}{15})(2\vec{b}_1 + 3\vec{b}_2)$ (b) $\vec{b}_3 = -\vec{b}_1 + 2\cos(\frac{2\pi}{5})\vec{b}_2$ (c) $\vec{b}_3 = (\sqrt{5} - \frac{\sqrt{7}}{2}\sin(\frac{4\pi}{5})\vec{b}_1 + \sqrt{7}\sin(\frac{2\pi}{5})\vec{b}_2)$ . . . 4

Figure 2.2. A a) triangular lattice, and b) its dual lattice 9

Figure 2.3. Schematic view of Bragg interpretation for diffraction 10

Figure 2.4. Schematic view of von Laue interpretation for diffraction 11

Figure 2.5. Point lattices a) square lattice, b) rectangular lattice, c) triangular lattice, d) triangular lattice with vacancies and their corresponding diffraction patterns 13

Figure 2.6. Schematic view of Delone cell for a triangular lattice 14

Figure 2.7. The elements of cubic system; a) simple cubic, b) face-centered cubic, c) body-centered cubic 14

Figure 2.8. The elements of tetragonal system; a) simple tetragonal, b) centered tetragonal 15

Figure 2.9. The elements of orthorhombic system; a) simple orthorhombic, b) face-centered orthorhombic, c) body-centered orthorhombic, d)base-centered orthorhombic 15

Figure 2.10. The elements of monoclinic system; a) simple monoclinic, b) centered monoclinic 16

Figure 2.11. The element of triclinic system 16

Figure 2.12. The element of trigonal system 17

Figure 2.13. The element of hexagonal system 17

Figure 2.14. Schematic view of Voronoï cell construction 19

Figure 2.15. the rhombic dodecahedron (a), the cube (b), 14-face truncated octahedron (c), the hexagonal prism (d), and the elongated rhombic dodecahedron (e) 20

Figure 2.16. The projection window for the $\mathcal{I}_2$ lattice 31

Figure 3.1. The projection of hypercubic lattice basis onto the (a) ε and (b) $\varepsilon^{\perp}$ 38

Figure 3.2. The projection window rhombic icosahedron 38

Figure 3.3. Rotated view perpendicular to a 2-fold axis of the rhombic-icosahedron 39

Figure 3.4. Rotated view perpendicular to a 5-fold axis of the rhombic-icosahedron 39

Figure 3.5. Real (a) and fourier space (b) images of Penrose tiling 39

Figure 4.1. The parallel space projection vectors and the corresponding icosahedron 42

Figure 4.2. The perpendicular space projection vectors 45

Figure 4.3. The polytope rhombic triacontahedron 46

Figure 4.4. View from a 2-fold symmetry direction 47

Figure 4.5. View from a 3-fold symmetry direction 47

Figure 4.6. View from a 5-fold symmetry direction 47

Figure 4.7. Projections of the generated crystal from a) 2-f plane, b) 3-fold plane, b) 5-fold plane of symmetry 48

Figure 4.8. Planes of projections of the generated quasicrystal. 49

Figure 4.9. Plane stacking in 5-fold symmetry direction 51

Figure 4.10. Heights versus densities for the 2-fold planes 51

Figure 4.11. Heights densities for the 3-fold planes 52

Figure 4.12. Heights densities for the 5-fold planes 52

Figure 4.13. The average coordination numbers as a function of the distance scales 53

Figure 4.14. Average coordination numbers per atom with respect to plane heights through a 2-fold axis 55

Figure 4.15. Average coordination numbers per atom with respect to plane heights through a 3-fold axis 56

Figure 4.16. Average coordination numbers per atom with respect to plane heights through a 5-fold axis 56

Figure 5.1. Coordination structure in 3-fold planes 59

Figure 5.2. Coordination structure in 5-fold planes 59

LIST OF TABLES

Table 4.1. Relative heights for different symmetry planes (in arbitrary units) 50

Table 4.2. Re-scaled relative heights for different symmetry planes 50

Table 4.3. The average coordination numbers for different distance scales . . . 54

LIST OF SYMBOLS

$\mathcal{A}$	Cyclic rotation matrix
C	Cylinder
C_n	Cyclic group order n
d	Lattice spacing
d_0	Average lattice spacing
D_n	Dihedral group order n
$\vec{e}_i$	ith Cartesian basis
$\|e_i^{\parallel}\rangle$	ith Parallel space basis
$\|e_i^{\perp}\rangle$	ith Perpendicular space basis
E^n	n-dimensional Euclidean space
$\vec{f}_i$	ith Facet vector
$\mathcal{F}$	Set of facet vectors
$GL\,(n, Z)$	General linear group
I	Intensity of light in diffraction
I_n	Identity matrix in n-dimensions
$\mathcal{I}_n$	Unit hypercubic lattice in n-dimensions
$\vec{k}$	Incoming plane wave vector
$\vec{k}'$	Scattered plane wave vector
K	Projection window
$\vec{K}$	Difference vector between $\vec{k}$ and $\vec{k}'$
$\mathcal{L}$	Lattice
$\mathcal{L}^*$	Dual lattice
M	Gramm matrix
$M^{\parallel}$	Parallel space projector in 3-dimensions
$M^{\perp}$	Perpendicular space projector in 3-dimensions
$N(\vec{b}_i)$	Euclidean norm of a vector
O_n	Orthogonal group order n
$\vec{p}_i$	Projection of ith cartesian basis onto parallel space
$\vec{q}_i$	Projection of ith cartesian basis onto perpendicular space

$S_{\vec{K}}$	Geometric structure factor
V_0	Voronoï cell around the origin
Z	The set of integers
δ_{ij}	Kronecker delta
Δ	Rational space in 5-dimensional projection
ε	An irrational space in projection
ε'	Another irrational space in 5-dimensional projection
$\varepsilon^{\perp}$	Perpendicular irrational space in projection
κ	Normalization constant for 3-dimensional projection matrix
λ	Wave-length
Λ	A Delone set
ξ	Real eigenvectors for 2-dimensional projection
Π	Parallel space projection operator
$\Pi^{\perp}$	Perpendicular space projection operator
ρ_i	ith Eigenvalue of cyclic rotation matrix
$\vec{\rho}_i$	ith Eigenvector of cyclic rotation matrix
τ	The *Golden number*
ϕ	Isometry
Ω	Orbit of a Z-module
τ	The *Golden number*

1. INTRODUCTION

A quasicrystal or quasiperiodic solid is a body that exhibits crystalline features such as symmetry and repeating patterns of unit cells (regular arrangements of atoms, molecules, or ions) but requires more than one type of unit cell to achieve large-scale order.In other words, the structure cannot consist of the repetition of a single cell. Quasicrystals exhibit symmetries (e.g., icosahedral and decagonal) not seen in crystals. Quasicrystals seem to forge a link between conventional crystals and materials called metallic glasses, which are solids formed when molten metals are cooled so rapidly that their constituent atoms do not have adequate time to form a crystal lattice. The first quasicrystal was discovered in a rapidly cooled sample of an aluminum-manganese alloy.

Three models have been advanced to explain the structure of quasicrystals. The Penrose model, derived from the work of Roger Penrose by Dov Levine and Paul J. Steinhardt [1], suggests that quasicrystals are composed of two or more unit cells that fit together according to specific rules. The glass model, as refined by physicists Peter W. Stephens and Alan J. Goldman [2], suggests that clusters of atoms join in a somewhat random way determined by local interactions. The random-tiling model, which combines some of the features of the other two models, suggests that the strict matching rules of the Penrose model need not be obeyed so long as local interactions leave no gaps in the structure [3].

Quasicrystals have been found to be common structures in alloys of aluminum with such metals as cobalt, iron, and nickel. Unlike their constituent elements, quasicrystals are poor conductors of electricity. Quasicrystals have stronger magnetic properties and exhibit greater elasticity at higher temperatures than crystals. Because they are extremely hard and resist deformation, quasicrystals form high-strength surface coatings, which has led to their commercial use as a surface treatment for aluminum pans.

2. THEORY

2.1. Basics of Convetional Crystallography

Although the characteristics of a nonperiodic crystal are different from conventional crystalline structure, we cannot skip the basics of crystallography which is mostly related to translational lattices, since the structures that we consider will reappear in the emerging theory as a sort of higher-dimensional periodic superstructure within which the nonperiodicities appear in a natural way [4].

We will work in n-dimensional Euclidean space (E^n) so that we have n orthonormal basis vectors $\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_n$. For the location of a specific point in this space we use coordinates $(x_1, x_2, \ldots, x_n)$, and we think of every point as a vector from the point $(0, 0, \ldots, 0)$ to it [4].

In the space E^n, k different vectors $(\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_k)$ where $k \leq n$ are linearly independent, if;

$$c_1\vec{a}_1 + c_2\vec{a}_2 + \ldots + c_k\vec{a}_k = 0 \Rightarrow c_1 = c_2 = \ldots = c_k = 0 \tag{2.1}$$

As a direct consequence, a set of $k+1$ points $a_0, a_1, \ldots, a_k$ are linearly independent if the condition Equation (2.1) holds for the vectors $\vec{a}_j = \vec{a}_j - \vec{a}_0$. As a corollary in E^n, k different vectors $(\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_k)$ where $k > n$ should be linearly dependent, so any of $\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_k$ can be expressed as a linear combination of at most n basis vectors.

Since in E^n, every vector $\vec{x}$ can be expressed as a set of k linearly independent vectors $b_0, b_1, \ldots, b_k$ as $\vec{x} = m_1\vec{b}_1 + m_2\vec{b}_2 + \ldots + m_k\vec{b}_k = 0$, we introduce a new concept;

Z-Module : A Z-module is a group that is generated by a set of vectors $b_0, b_1, \ldots, b_k$ under the operation of addition. The elements of this group is countably infinite. Thus the elements of a Z-module are the vectors of the form $\vec{x} = m_1\vec{b}_1 + m_2\vec{b}_2 +$

$\ldots + m_k\vec{b}_k = 0$, where the coefficients m_i are integers.

We need the following definition;

Orbit of a Z-Module : For a given Z-module, the point sets that are generated by the elements of this Z-module is called the orbit of the Z-module. In other words, the action of a Z-module onto a point set generate the orbit of that Z-module.

Hence, there should be some orbits for a given Z-module and we will denote any of these orbits by Ω. In Figure 2.1 we show the parts of three different Z-modules in R^2, where all of them has three elements. The elements $\vec{b}_1$ and $\vec{b}_2$ are the same for each Z-modules, so that $\vec{b}_1 = (1, 0)$ and $\vec{b}_2 = \big(\cos(\frac{2\pi}{5}), \sin(\frac{2\pi}{5})\big)$. On the other hand, $\vec{b}_3 = (\frac{1}{15})(2\vec{b}_1 + 3\vec{b}_2)$ for the Z-module in (a), $\vec{b}_3 = -\vec{b}_1 + 2\cos(\frac{2\pi}{5})\vec{b}_2$ for the Z-module in (b), and $\vec{b}_3 = \sqrt{5} - \frac{\sqrt{7}}{2}\sin(\frac{4\pi}{5})\vec{b}_1 + \sqrt{7}\sin(\frac{2\pi}{5})\vec{b}_2$ for the Z-module in (c).

Obviously these three point sets in Fig 2.1 are very different firstly by means of the densities of the points. The formation of the points here leads us to make two other definitions about the densities of the point sets.

Denseness : Let A and B be two point sets. If every neighborhood of every point which belongs to the set B contains the points of A, then the set A is dense in set B [4].

Discreteness : Let A be a point set and let B be a space that includes the set A. If every point of set A has a common neighborhood such that there is no other point which belongs to the set of A is contained by this neighborhood, the point set A is discrete in the space B [4].

Now we can classify these three point sets with respect to the 2D Euclidean space (E^2) with the help of the denseness and discreteness criterions. The orbit in Figure 2.1(a) is a discrete point set and its generators are both linearly and integrally dependent so that $\vec{b}_3$ can be written as combinations of integer multiples of $\vec{b}_2$ and $\vec{b}_1$.

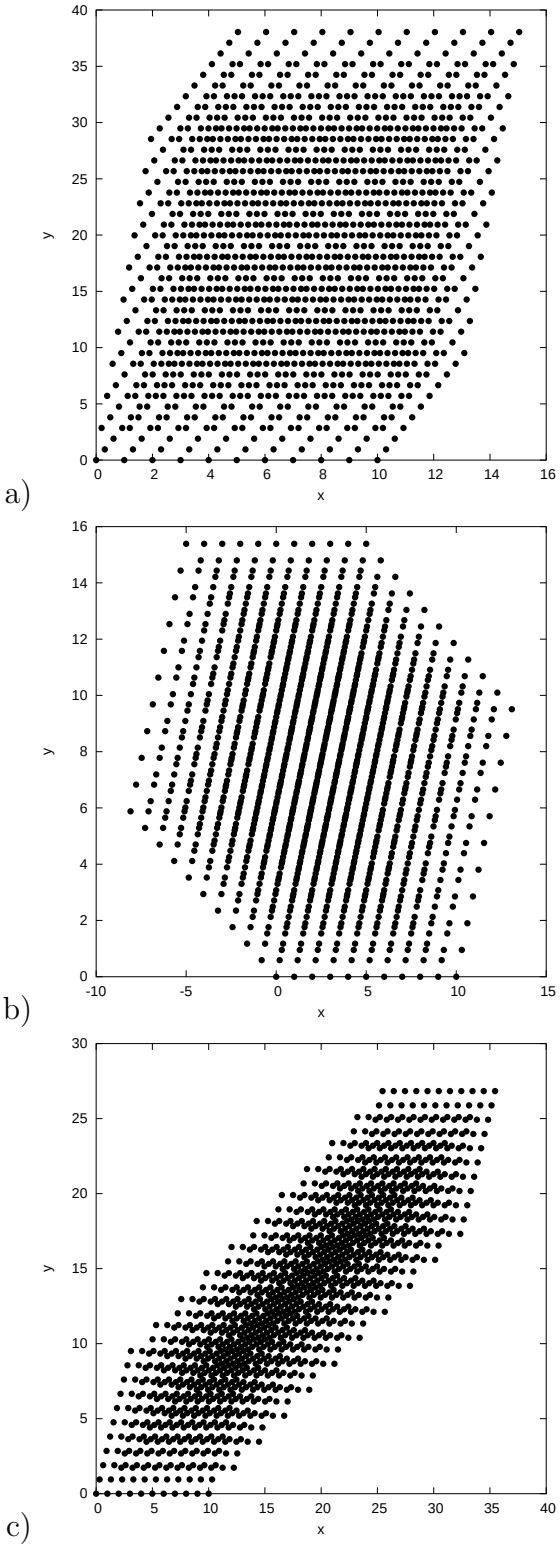


Figure 2.1. Portions of orbits generated by (a) $\vec{b}_3 = (\frac{1}{15})(2\vec{b}_1 + 3\vec{b}_2)$ (b) $\vec{b}_3 = -\vec{b}_1 + 2\cos(\frac{2\pi}{5})\vec{b}_2$ (c) $\vec{b}_3 = (\sqrt{5} - \frac{\sqrt{7}}{2}\sin(\frac{4\pi}{5})\vec{b}_1 + \sqrt{7}\sin(\frac{2\pi}{5})\vec{b}_2)$

On the other hand the point sets in Figure 2.1(b) and (c) are not integrally dependent.

Moreover the point sets in Figure 2.1(c), if all of it could be shown, is dense in the plane, while the point sets in Figure 2.1(b) somehow arranged as a family of densely filled parallel layers (lines). These point sets are representative of the general situation. *The orbit of a Z-module in E^n always consists of translations of densely filled d-dimensional subspaces where $d \in [0, n]$* [4].

A final distinction among point sets in Fig 2.1 will be their ranks. The *rank of a module* is just the number of integrally independent vectors in it. Therefore the rank of an orbit in Fig 2.1(a) is 2, whereas the ranks of (b) and (c) are 3. Furthermore when the rank of an orbit is equal to its span which is the number of vectors which span the subspace, the orbit is called a *lattice*. Here the general definition for a lattice arise;

Lattice If a Z-module in E^n is generated by n linearly independent vectors, it is a lattice. In other words the rank of every orbit of the Z-module will be equal to its span, therefore these orbits are all discrete [4].

2.2. Lattices, Dual Lattices, and The Diffraction Patterns

2.2.1. Lattices

An n-dimensional *lattice* is a discrete orbit of a Z-module (point lattice) and because of that there should be n linearly independent vectors that are its generators. The set of vectors forms a basis for the lattice $\mathcal{L}$.

Let $\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_n$ be a basis for $\mathcal{L}$. The norm $\mathrm{N}(\vec{b}_i)$ of $\vec{b}_i$ is;

$$N(\vec{b}_i) = |\vec{b}_i|^2 = \vec{b}_i \cdot \vec{b}_i, \tag{2.2}$$

Since a point lattice is discrete and the norm in Euclidean space is positive definite, $\mathcal{L}$ has a finite set of vectors of minimal or non-zero norm. These are the *short vectors* of $\mathcal{L}$.

With the help of these vectors we can construct a *generator matrix* B for $\mathcal{L}$ such that its columns are the components of the vectors $\vec{b}_i$. Then we can define det $\mathcal{L}$, the determinant of $\mathcal{L}$, by det $\mathcal{L}$ = det B; the determinant is independent of the basis or is just the volume of the *unit cell* of the point lattice. The matrix $M = B^T B$ is called the *Gramm matrix* for $\mathcal{L}$ where B^T is the transpose of B. Its (i,j)th entry is the scalar product of $\vec{b}_i \cdot \vec{b}_j$; thus its diagonal entries are the norms of the basis vectors. If for all $\vec{x} \in \mathcal{L}, N(\vec{x}) \in Z$, then $\mathcal{L}$ is said to be an *integral lattice* [4].

If the *Gramm matrix* is the $n \times n$ identity matrix the lattice associated with it will be the simple hypecubic lattice $\mathcal{I}_n$. The most important property of this lattice ($\mathcal{I}_n$) is that it is the unique n-dimensional lattice whose basis vectors are orthogonal to each other, in other words it is an orthogonal direct sum of one dimensional lattices [4].

Another feature of the *generator matrix* is that it is possible to write a new generator matrix as BA where A is an $n \times n$ integer matrix with det $A = \pm 1$. Using the identity, det BA = det B det A, we can easily prove this assertion, and hence, the volume of the original lattice that is generated by the matrix B, will be the same for the lattice that is generated by the matrix BA.

Experimentally, the properties of a lattice can be understood in terms of its diffraction pattern and it is therefore necessary to introduce the the notion of a dual lattice. Every lattice $\mathcal{L}$ has a dual lattice $\mathcal{L}^*$, the set of vectors whose scalar products with the vectors of $\mathcal{L}$ are integers [4].

2.2.2. Dual Lattices

Dual Lattice The dual $\mathcal{L}^*$ of $\mathcal{L}$ is the set of vectors $\vec{y} \in E^n$ defined by:

$\vec{y} \in E^n \Longleftrightarrow \vec{y} \cdot \vec{x} \in Z \; \forall \, \vec{x} \in \mathcal{L}$.

For diffraction, it is actually more convenient to define the dual lattice as follows. Let the sets $A = \{\vec{a}_1, \vec{a}_2 \ldots, \vec{a}_n\}$ and $B = \{\vec{b}_1, \vec{b}_2 \ldots, \vec{b}_n\}$ be the generators of the lattices

$\mathcal{L}$ and $\mathcal{L}^*$, respectively. Then the relation for any $\vec{b}_j \in B$ and $\vec{a}_i \in A$ is given as

$$\vec{a}_i \cdot \vec{b}_j = 2\pi\delta_{ij} \Rightarrow e^{iA\cdot B} = 1 \tag{2.3}$$

where δ_{ij} is the *Kronecker delta* defined by;

$$\delta_{\mathbf{ij}} = \begin{cases} 1 & \text{if i = j} \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

As an example, consider the case $n = 3$ with $A = \{\vec{a}_1\,, \vec{a}_2\,, \vec{a}_3\}$. Then the set $B = \left\{\vec{b}_1\,, \vec{b}_2\,, \vec{b}_3\right\}$ is given by the vectors [4, 5];

$$\vec{b}_1 = 2\pi\frac{\vec{a}_2 \times \vec{a}_3}{V} \quad \vec{b}_2 = 2\pi\frac{\vec{a}_3 \times \vec{a}_1}{V} \quad \vec{b}_3 = 2\pi\frac{\vec{a}_1 \times \vec{a}_2}{V} \tag{2.5}$$

where the V is the unit cell volume and it is $V = \vec{a}_1 \cdot (\vec{a}_2 \times \vec{a}_3)$

As it is easily seen, the length scales of generators of the reciprocal lattice are the inverse of generators of the real lattice. Thus, if $\mathcal{L}$ is the one dimensional point lattice with lattice spacing d,

$$\mathcal{L} = \{\, nd \,|\, n \in Z \,\}, \tag{2.6}$$

then $\mathcal{L}^*$ is again a one dimensional point lattice with lattice spacing $\frac{1}{d}$.

$$\mathcal{L}^* = \{\frac{m}{d} \,|\, m \in Z \,\}. \tag{2.7}$$

Then is an immediate relation between real and reciprocal lattice as;

$$(\mathcal{L}^*)^* = \mathcal{L} \tag{2.8}$$

As can be readily shown in the case of a 1-d lattice [4];

$\mathcal{L} = \{\, nd \,|\, n \in Z \,\}$, and $\mathcal{L}^* = \{\, \frac{m}{d} \,|\, m \in Z \,\} \Rightarrow (\mathcal{L}^*)^* = \{kd \,|\, k \in Z \,\}$
and because $m = \frac{1}{n} \Rightarrow k = n$. This implies $(\mathcal{L}^*)^* = \mathcal{L}$.

There is a unique lattice which is both integral and self dual. This special lattice is the standard lattice $\mathcal{I}_n$ [4, 5]. As an illustration, let us construct a 2-dimensional lattice and its dual lattice. For the sake of simplicity we take the lattice vectors as;

$$\vec{a}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \vec{a}_2 = \begin{pmatrix} \cos\left(\frac{\pi}{3}\right) \\ \sin\left(\frac{\pi}{3}\right) \end{pmatrix} \tag{2.9}$$

So we deal with the equilateral triangular lattice. The corresponding vectors can be found using the Equation (2.3);

$$\begin{aligned} \vec{a}_1 \cdot \vec{b}_1 = 2\pi \quad \vec{a}_2 \cdot \vec{b}_2 = 0 \\ \vec{a}_2 \cdot \vec{b}_1 = 0 \quad \vec{a}_2 \cdot \vec{b}_2 = 2\pi \end{aligned} \tag{2.10}$$

The calculated vectors are therefore given by;

$$\vec{b}_1 = 2\pi \begin{pmatrix} 1 \\ \frac{-1}{\sqrt{3}} \end{pmatrix}, \vec{b}_2 = 2\pi \begin{pmatrix} 0 \\ \frac{2}{\sqrt{3}} \end{pmatrix} \tag{2.11}$$

Finally, the lattice and the dual lattice belong to these vectors can be seen below;

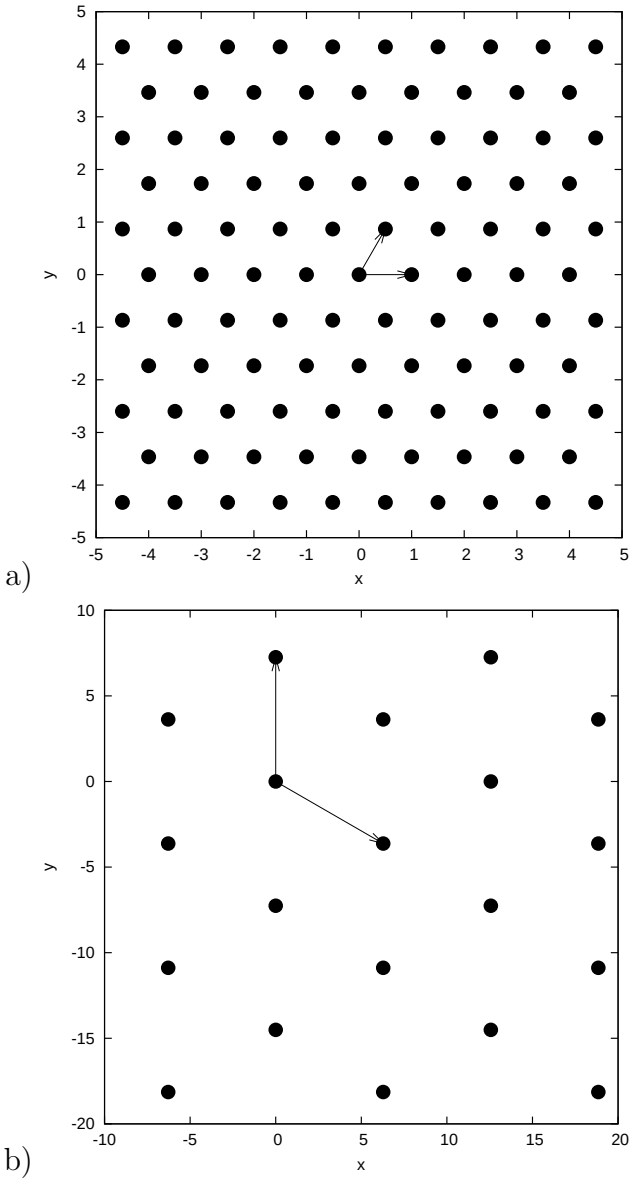


Figure 2.2. A a) triangular lattice, and b) its dual lattice

2.2.3. The Diffraction Patterns

A diffraction pattern for a crystalline structure can be treated as a multi-slit diffraction image from one source of light so that there is a coherent propagation of the light from every slit, where the slits correspond to the atoms in the lattice which actually scatter the light. The diffraction of X-rays from a crystalline structure can be understood in two different ways, hence we have two equivalent formalism.

2.2.3.1. Bragg Formulation of X-ray Diffraction. The characteristics of reflected X-ray intensity peaks, can be accounted for if the crystal structure is viewed as succesive planes of atoms which obey two assumptions:

B1. The X-rays should be specularly reflected by atoms of any one plane.
B2. The reflected rays from successive planes should interfere constructively.

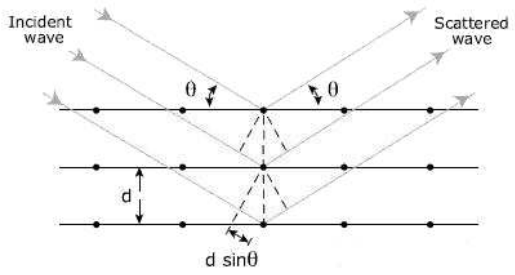


Figure 2.3. Schematic view of Bragg interpretation for diffraction

Hence, the constructive interference condition is $m\lambda = 2d\sin(\theta)$

2.2.3.2. The von Laue Formulation of X-ray Diffraction. The difference between the Bragg and the von Laue formalisms is the latter does not include the assumption B1. Hence it involves again the assumption that the X-rays scattered from atoms can interfere constructively with each other.

Let $\vec{k}$ and $\vec{k}'$ be the incident and reflected wave vectors. Then the change in wave vector is defined as $\vec{K} = \vec{k}$ - $\vec{k}'$, and the path difference between the rays scattered from

atoms at $\vec{d_i}$ and $\vec{d_j}$ will be $\vec{K} \cdot (\vec{d_i} - \vec{d_j})$. The condition for constructive interference is therefore;

$$\vec{K} \cdot \left(\vec{d_i} - \vec{d_j} \right) = 2\pi m, \, where \, m \in Z \tag{2.12}$$

The condition Equation (2.12) can be written in equivalent form;

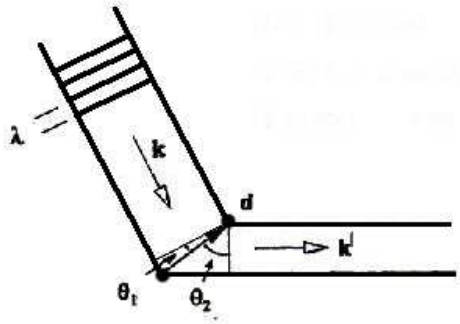


Figure 2.4. Schematic view of von Laue interpretation for diffraction

$$e^{i\vec{K} \cdot (\vec{d_i} - \vec{d_j})} = 1, \tag{2.13}$$

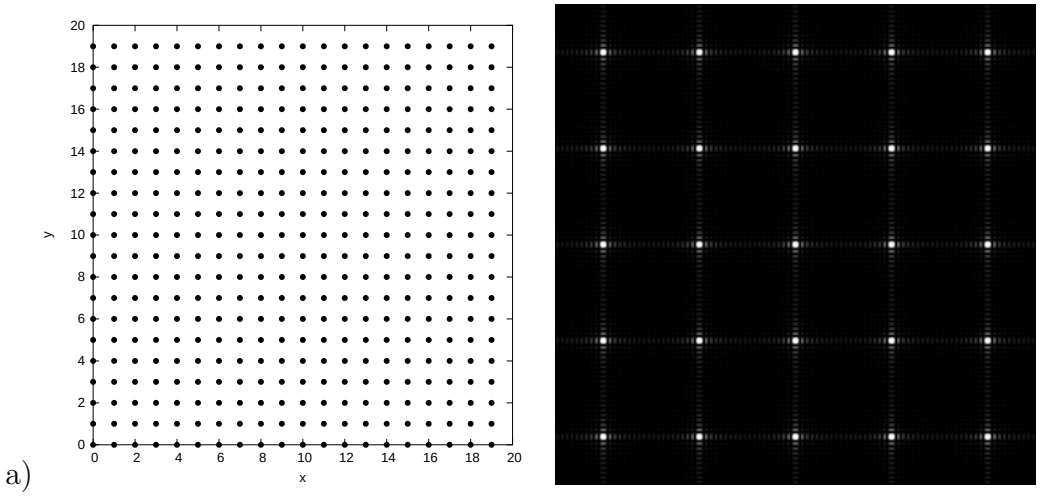
If we compare this with Equation (2.3), we see that the Laue condition for constructive interference requires the vector $\vec{K} = \vec{k} - \vec{k'}$ should be a vector of the dual lattice, since $\vec{d_i} - \vec{d_j}$ is a vector of the real space lattice. The net ray scattered by one of these elements contains *a geometrical structure factor*;

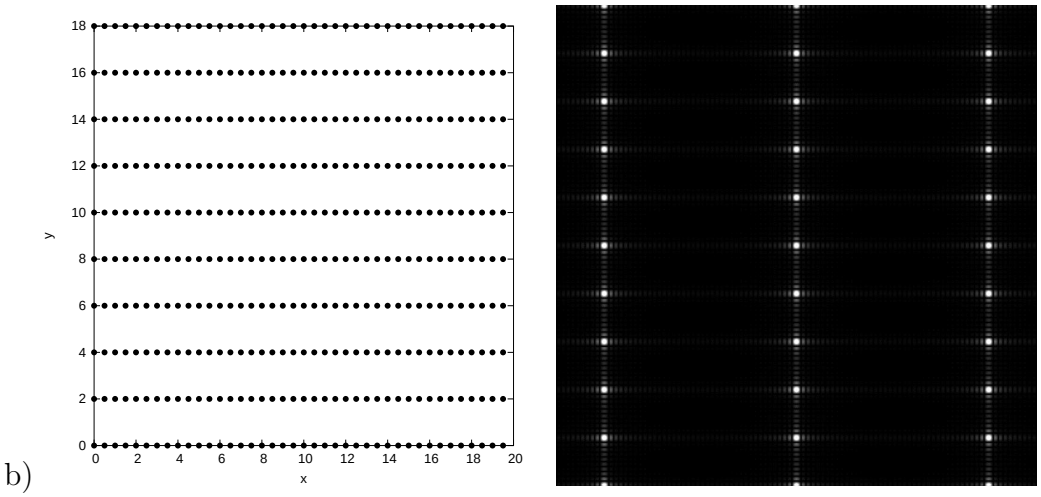
$$S_{\vec{K}} = \sum_{\vec{d} \in \mathcal{L}} e^{i\vec{K} \cdot \vec{d_j}} \tag{2.14}$$

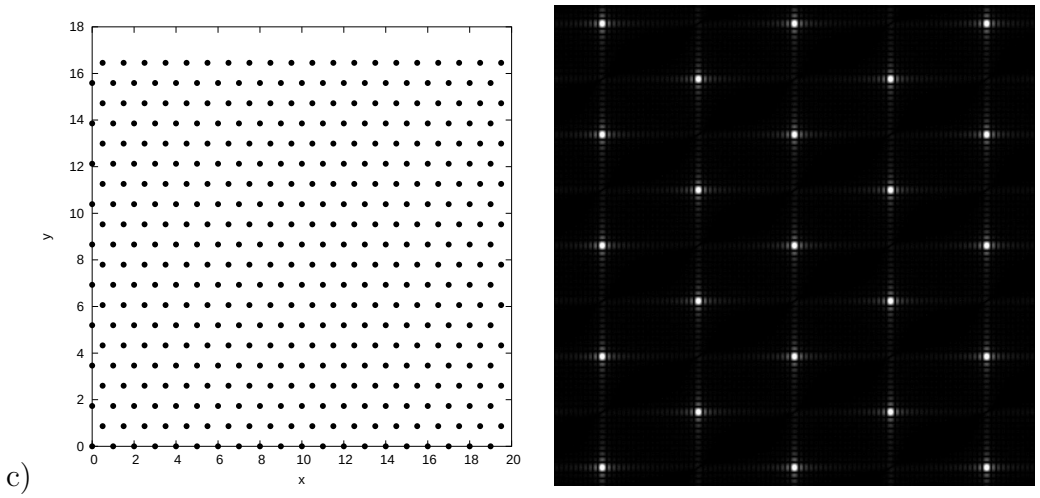
The intensities in a diffraction pattern are proportional to the square of the absolute value of the amplitude of the net ray, so;

$$I \propto | S_{\vec{K}} |^2 = S_{\vec{K}} S_{(-\vec{K})} \tag{2.15}$$

In Figure 2.5 we show a set of point lattices along with the corresponding diffraction patterns.

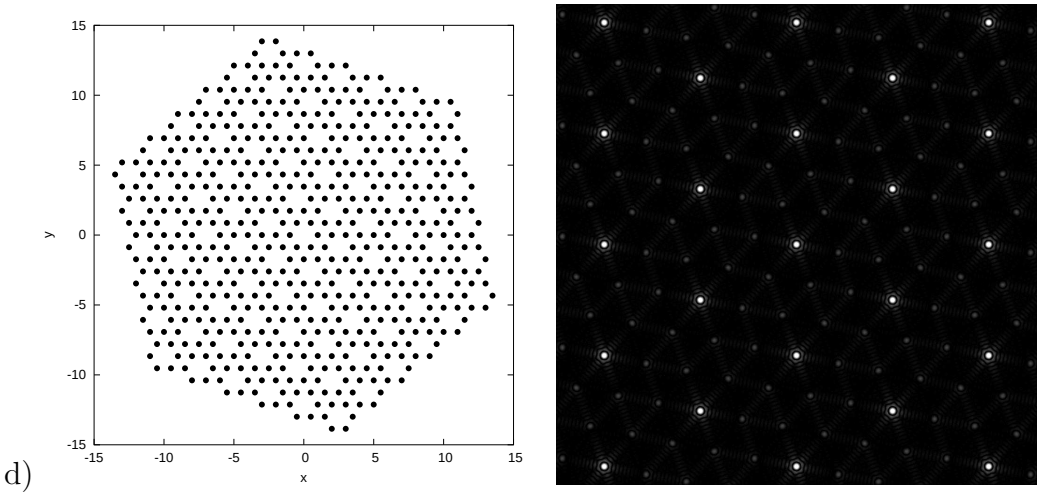






2.3. Unit Cell Phenomena

In crystallography, the symmetry properties of a crystal lattice are best understood in terms of the symmetry of the polytope (parallelpipedal block) that tiles the



d)

Figure 2.5. Point lattices a) square lattice, b) rectangular lattice, c) triangular lattice, d) triangular lattice with vacancies and their corresponding diffraction patterns

lattice. This polytope is referred as the *unit cell*. There are some of these tilings that are relevant for our purpose:

1. Every tile is a convex polytope in n-dimensional Euclidean space.
2. The points that form the lattice, (corresponding to the atoms of the crystal), can be either inside the polytope or on the vertices of it.
3. If the points are inside the polytope the unit cell is called *Voronoï Cell*, if they are on the vertices it is a *Delone Cell* [4].
4. The volume of the polytope is equal to $|\det \mathcal{L}|$, Sec. 2.2.1.

As we explained in Sec 2.2.1, every point lattice can be partitioned into unit cells in infinitely many ways which is a result of the freedom in choosing a basis.

2.3.1. Delone Cell

The *Delone Cell* is a polytope that shows the symmetry properties of the lattice. We can think of it as being constructed from connecting the lattice points nearest to an appropriately close fictitious point in the space in which the lattice embedded. In Figure 2.6, the Delone cell for a triangular lattice is shown and the fictitious point is marked by an x.

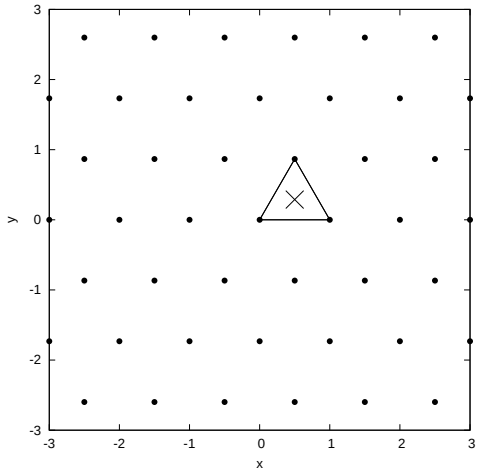
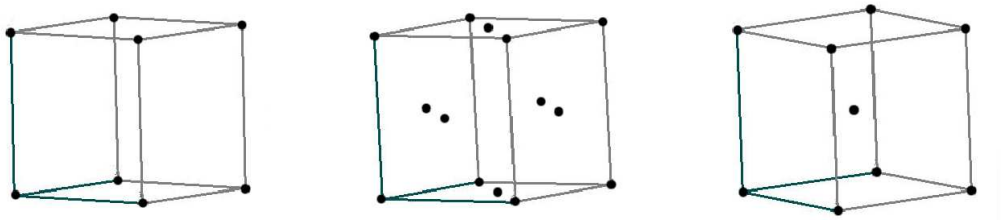


Figure 2.6. Schematic view of Delone cell for a triangular lattice

In 3-d there are seven types of Delone cells, the cubic, tetragonal, orthorhombic, monoclinic, triclinic, trigonal (rhombohedral) and hexagonal cell giving rise to a whole of 14 different types of point lattices called the 3-d *Bravais Lattices* [5].

Cubic (3) This system contains the three Bravais lattices whose Delone cell can be chosen a cube. Three Bravais lattices with nonequivalent configurations all belongs to this group. They are the *simple cubic, body-centered cubic, and face-centered cubic* lattices as shown in Figure 2.7

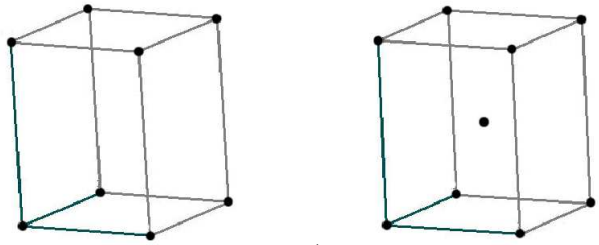


a) b) c)

Figure 2.7. The elements of cubic system; a) simple cubic, b) face-centered cubic, c) body-centered cubic

Tetragonal (2) One can reduce the symmetry of a cube by pulling on two opposite faces to stretch it into a rectangular prism with a square base, but a height not equal to the sides of the square. Thus by stretching the simple cubic Bravais lattice one obtains the *simple tetragonal* Bravais lattice, which can be characterized as a Bravais lattice generated by three mutually perpendicular primitive vectors,

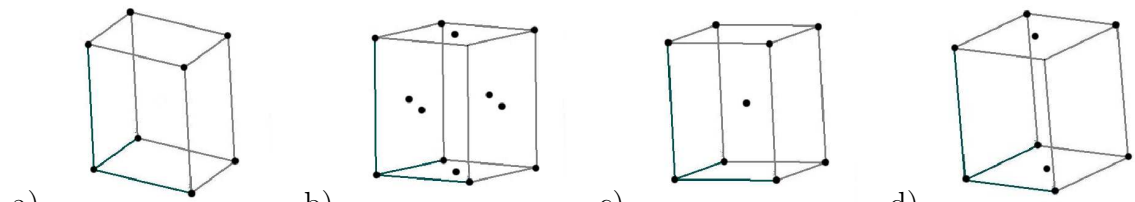
only two of which are of equal length. The axis along the long primitive vector is often referred as the *c-axis*, the slab axes are referred as *a-axis*. By similarly stretching the body-centered and face-centered cubic lattices one arrives at one more Bravais lattice, the *centered tetragonal* lattice. In Figure 2.8 2 tetragonal lattices are shown.



a) b)

Figure 2.8. The elements of tetragonal system; a) simple tetragonal, b) centered tetragonal

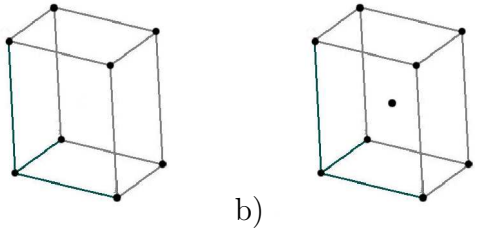
Orthorhombic (4) Continuing to still less symmetric deformations of the cube, one can reduce tetragonal symmetry by deforming the square faces of the object into rectangles, producing an object with mutually perpendicular sides of three unequal lengths. By stretching the simple tetragonal lattice along one of the *a-axis*, one produces *simple orthorhombic* Bravais lattice Figure 2.9 a). However, by stretching the simple tetragonal lattice along a square diagonal one produces a second Bravais lattice, the *base-centered orthorhombic* Figure 2.9 b). In the same way, one can reduce the point symmetry of the centered tetragonal lattice to orthorhombic in two ways, stretching either along one set of parallel lines to produce *body-centered orthorhombic* lattice Figure 2.9 c), or along one set of parallel lines, producing *face-centered orthorhombic* lattice Figure 2.9 d).



a) b) c) d)

Figure 2.9. The elements of orthorhombic system; a) simple orthorhombic, b) face-centered orthorhombic, c) body-centered orthorhombic, d)base-centered orthorhombic

Monoclinic (2) One can reduce the orthorhombic symmetry by distorting the rectangular faces perpendicular to the *c-axis* into general parallelograms. By so distorting the simple orthorhombic Bravais lattice one produces the *simple monoclinic* Bravais lattice Figure 2.10 a), which has no symmetries other than those required by the fact that it can be generated by three primitive vectors, one of which is perpendicular to the plane of the other two. Similarly, distorting the base-centered orthorhombic Bravais lattice produces a lattice with the same simple monoclinic space group. However, so distorting either the face-centered or body-centered orthorhombic Bravais lattices produces the *centered monoclinic* Bravais lattice Figure 2.10 b).



a) b)

Figure 2.10. The elements of monoclinic system; a) simple monoclinic, b) centered monoclinic

Triclinic (1) The destruction of cube is completed by tilting the *c-axis* so that it is no longer perpendicular to the other two, upon which there are no restrictions except that pairs of opposite faces are parallel. By so distorting either of the monoclinic Bravais lattices one constructs the *triclinic* Bravais lattice. This is the Bravais lattice generated by three primitive vectors with no special relationships to one another, and is therefore the Bravais lattice of minimum symmetry.

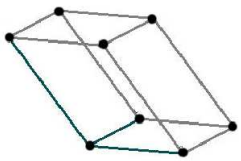


Figure 2.11. The element of triclinic system

Rhombohedral(Trigonal) (1) This type of lattice can be described by stretching

a cube along a body diagonal. This result is same for all three types of cubic lattices. It is generated by three primitive vectors of equal length that make equal angles with one another.

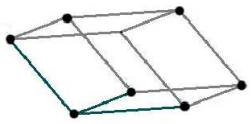


Figure 2.12. The element of trigonal system

Hexagonal (1) This type of Bravais lattice can not be made by any kind of distortion of the cube. The *simple hexagonal* Bravais lattice can be partitioned into right prisms with a regular hexagon as base,as shown in the figure Figure 2.13.

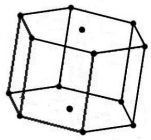


Figure 2.13. The element of hexagonal system

At this point, we leave the discussion of the symmetry features of Bravais lattices by means of their Delone cell polytopes, and go on to another partition technique namely the construction of Voronoï cells.

2.3.2. Voronoï Cell

It is useful to construct another polytope that has the same volume as the primitive cell, $|\det \mathcal{L}|$, does display the symmetries of the original lattice and, has the further advantage of being independent of the choice of basis [4]. Such a construction is the Voronoï unit cell. It is formally defined as;

Voronoï Cell Let $\Lambda \in E^n$ be any Delone set (or even a finite point set). The Voronoï cell of a point $x \in \Lambda$ is the set of points of E^n that lie at least as close as to any other point of Λ:

$$V(x) = \{u \in E^n || x - u| \leq |y - u|, \forall y \in \Lambda\}. \tag{2.16}$$

To construct the Voronoï cell of x, we connect the point x with its nearest neighbors using straight line segments. Then we construct the $(n-1)$-dimensional perpendicular bisector of each of these segments. Finally the Voronoï cell V(x) is just the smallest convex region about x bounded by these hyperplanes.

If we carry out this construction for every point of Λ, we obtain a partition of E^n into cells called the *Voronoï tessellation* induced by Λ. The main properties of the Voronoï tessellation are [4]:

(i) All the cells are convex polytopes and locked to each neighbor so they fit together along whole faces.

(ii) There is no common point which belongs any two or more cells other than their borders. No two cells have a common interior point.

(iii) For the Voronoï tessellation of an n-dimensional Delone set Λ, any of the vertices of the Voronoï cells can be thought as the center of n-dimensional spheres around these vertices. Furthermore, it is always possible to choose the radii of these spheres such that their interior does not contain any of the points of Λ.

From the definition of Voronoï cell Equation (2.16), and the above properties of Voronoï cells we need a definition for the cell structure of a tesellated lattice.

Congruency Let Λ be a Voronoï tesellated lattice and V(x) be the Voronoï cells on this lattice. For the whole lattice, if there is no gap between the Voronoï cells such that they are placed in the lattice border-to-border, the cells are called *congruent*.

The Voronoï cells associated with a lattice are congruent polytopes. The simplest example is the Voronoï cells of an n-dimensional hypercubic lattice which is the integral lattice $\mathcal{I}_n$. Since the Voronoï cells of a lattice Λ are congruent we can consider only the Voronoï cell around the lattice point $(0, 0, 0, \dots,)$ and we will refer to it V(0). This means

$$\forall x \in \mathcal{L}, V(x) = V(0) + x. \tag{2.17}$$

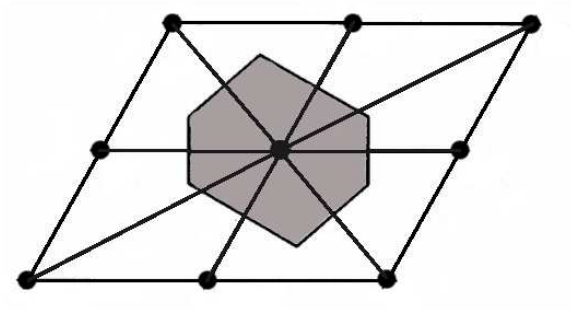


Figure 2.14. Schematic view of Voronoï cell construction

The $(n-1)$-dimensional surfaces of Voronoï cells are called *facets* of the Voronoï cell, and the vectors joining x to another lattice point whose Voronoï cell shares a facet with V(x) are called *facet vectors*. We will denote the facet vectors of a lattice Λ by $\mathcal{F}_\mathcal{L}$ or simply by $\mathcal{F}$. The set of basis vectors of a point lattice (short vectors) S are the facet vectors in most cases. Minkowski has shown the Voronoï cell of any point lattice has at least $2n$ facet vectors and at most $2(2^n - 1)$. We will write the set of facet vectors of a n-dimensional Voronoï polytope as $\mathcal{F} = \{\vec{f_1}, \vec{f_2}, \vec{f_3}, \cdots, \}$ [4].

The identification of Voronoï cell and its facet vectors is important in order to understand the structural symmetry properties of a lattice. When the density or positional correlations are calculated, it is useful to find the Voronoï cell structure. Thus let $\mathcal{L}$ be a lattice and let $x \in E^n$. Then $x \in V(0)$ and $V(0) = \left\{ \vec{x} \,|\, \vec{x} \cdot \vec{f} \leq \frac{N(\vec{f})}{2} \, \forall \, \vec{f} \in \mathcal{F} \right\}$.

Because of the inversion symmetry of the lattice Λ facets of a Voronoï cell come in parallel pairs, corresponding to pairs of facet vectors $\pm\vec{f}$. Thus the Voronoï cell of a n-dimensional point lattice is the intersection of $(n-1)$-dimensional *slabs*. And, in our context, a *slab* is a region of E^n bounded by two parallel $(n-1)$-dimensional hyperplanes.

There are many different results on the combinatorial types of Voronoï cells of lattices in different dimensional spaces. For instance in 2-dimensional lattices there can be just two different types of Voronoï cells: rectangular or hexagonal. Rectangular Voronoï cells arise in lattices whose basis vectors are orthogonal, while hexagonal Voronoï cells are associated with lattices that have non-orthogonal basis vectors. For 3-dimensional lattices there are five different combinatorial types of Voronoï cells, the cube, the rhombic dodecahedron, the elongated rhombic dodecahedron, the 14-face truncated octahedron, and the hexagonal prism see Figure 2.15. It is known that there are 52 different types of Voronoï cells for 4-dimensional lattices. However the exact numbers for the higher dimensional spaces is not known [4, 5].

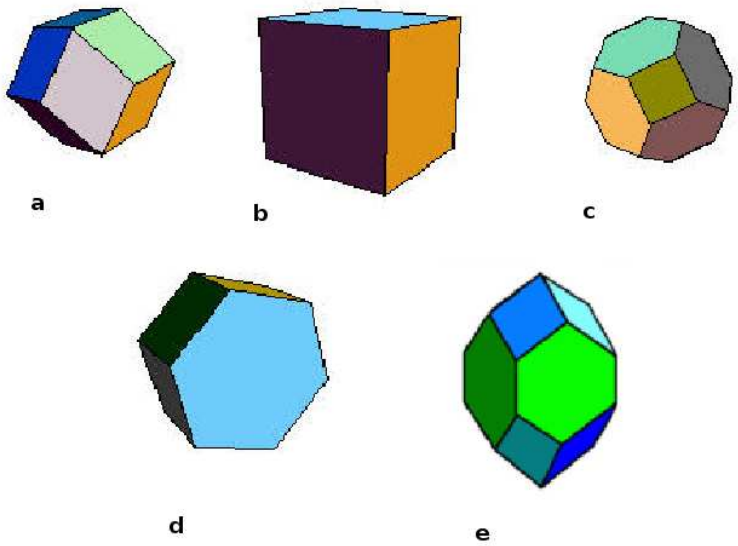


Figure 2.15. the rhombic dodecahedron (a), the cube (b), 14-face truncated octahedron (c), the hexagonal prism (d), and the elongated rhombic dodecahedron (e)

2.4. Symmetry And Group Properties

As we mentioned in the previous section, to construct structure models for different types of crystals, we need to find the symmetry properties of the physical lattices, Equations 2.14, 2.15. The fourier transforms of real lattices is a major tool to inform us about these properties, and the Voronoï tessellation of a lattice is another way to see them. We turn now to the symmetry groups of three dimensional lattices, the crystallographic groups.

2.4.1. Symmetry

An *isometry* or *congruence transformation* is any mapping of the Euclidean space E^n onto itself which preserves all the distances. Thus if x and y are the points in E^n and if ϕ is an isometry of E^n, the distance between x and y and the distance between their images under the isometry ϕ, $\phi(x)$ and $\phi(y)$ should be equal [3, 4]:

$$|\phi(x-y)| = \phi(x) - \phi(y) = |(x-y)|. \tag{2.18}$$

Let ϕ be a mapping of E^n onto itself that keeps the origin fixed, $\phi(0) = 0$. If ϕ is an isometry then it keeps both the angles and the distances fixed [4]. So it can be written as;

Linear Isometry : Let $\phi : E^n \to E^n$ and $\phi(0) = 0$. Then ϕ is a linear isometry if and only if $\vec{x} \cdot \vec{y} = \phi(\vec{x}) \cdot \phi(\vec{y}) \; \forall \; \vec{x} \cdot \vec{y} \in E^n$.

There are four main types of isometries. The classification of isometries is as follows [3];

1. *Rotation* about a point O through a given angle θ, the point is called the *center of rotation*. In the particular case when $\theta = \pi$, the line joining the points to their images under ϕ is bisected by O, and in this case the mapping is sometimes called

halfturn, central reflection, or reflection about the point O.

2. *Translation* in a given direction through a given distance.
3. *Reflection* in a given $(n-1)$-dimensional hyperplane P which is called *the mirror* or *plane of reflection.*
4. *Glide reflection* in which reflection in a plane P is combined with a translation through a given distance d parallel to P.

Note that in general not every isometry is a linear isometry. *Translation* and *glide reflection* are not linear isometries, because they have no fixed points. On the other hand, this type of isometry with no fixed points (translation, glide reflection) is either a translation or a composite of a translation and an isometry conjugate to a linear isometry. A linear isometry in E^n can be represented by $n \times n$ unimodular matrix ($\det = \pm 1$), since all of its eigenvalues must have absolute value one. This can be seen easily as follows;

The orthogonality condition for a $n \times n$ matrix ϕ is; $\phi\, \phi^{\top} = I_n$. Moreover since, $\det(\phi) = \det(\phi^{\top})$ and $\det(\phi\, \phi^{\top}) = \det(\phi)\det(\phi^{\top})$, it follows $\det(\phi)^2 = 1$, that is, $\det(\phi) = \pm 1$.

In other words a linear isometry is a generalized rotation about the origin. Since the determinant is independent of choice of basis of E^n, every linear isometry is either 'proper' or 'improper' depending on whether det $\phi = 1$ or -1. The simplest example of a proper isometry is the rotation in E^2 around the origin; that of the improper isometry will be a reflection with respect to the x-axis. Formally, this can be expressed as follows

Let $R(\theta)$ be a rotation matrix around the origin, and P be a reflection with respect to the x-axis, then $R(\theta) = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$, and $P = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$. The determinant of $R(\theta)$ is $\cos^2(\theta) + \sin^2(\theta) = 1$, and determinant of P is -1. Hence rotation is a proper isometry, but reflection is improper.

The subspace of all points fixed by a linear isometry ϕ is its *fixed subspace*. A proper subspace of E^n that is mapped to itself by ϕ but is not necessarily fixed by it, is called a stable subspace.

Linear isometries in low dimensional spaces can be easily described in terms of their fixed and stable subspaces. Returning our simple example above, a rotation in E^2 has a single fixed point (the center of rotation) but no proper stable subspace, while a rotation in E^3 has a one dimensional fixed space (the rotation axis) and a two dimensional stable subspace (the plane through the origin orthogonal to the axis). A reflection in E^2 has a fixed line and a stable line orthogonal to it; a reflection in E^3 has a fixed plane and a stable line orthogonal to it [3].

The linear isometries are commonly members of two types of groups. These groups are called *cyclic groups* and *dihedral groups*.

Cyclic Groups C_n is a group of order n that can be generated by a single element, in the sense that every other group elements can be expressed as the powers of this element, e.g., an n-fold rotational symmetry around a fixed axis.

Dihedral Groups D_n is a group of order $2n$ that both includes the cyclic group that is of order n and as many mirror symmetries as are required by the existence of an n-fold axis. It is the symmetry group of a regular n-gon with the $2n$ elements, i.e, is of order $2n$.

Let ϕ be any one to one linear transformation of E^n onto itself. By iterating ϕ and its inverse, we can generate a cyclic group of transformations whose elements are I, $\phi^{\pm 1}$, $\phi^{\pm 2}$, $\phi^{\pm 3}$,$\cdots$ If there is no integer k such that, $\phi^{\pm k} = I$ the *order* of the group (or the number of elements in the group) is infinite otherwise the order of the group is the smallest k for which $\phi^k = I$. For instance, a reflection is an isometry of order two, a rotation through $\frac{2\pi}{k}$ radians is an isometry of order k, but a translation is of infinite order.

As we asserted before any linear isometry is just an abstract rotation around the origin. Thus by a suitable choice of basis, every linear isometry in E^n can be represented as an $n \times n$ matrix reduced to block-diagonal form. Hence

$$\mathbf{A} = \begin{pmatrix} A_1 & 0 & 0 & \dots \\ 0 & A_2 & 0 & \dots \\ \vdots & \vdots & \ddots & \\ \dots & \dots & \dots & A_m \end{pmatrix} \tag{2.19}$$

where each A_j is either 1, -1, or a 2×2 matrix of the form

$$\begin{pmatrix} cos\theta & -\sin\theta \\ sin\theta & \cos\theta \end{pmatrix} \tag{2.20}$$

This matrix describes the action of ϕ on an orthonormal basis of E^n. If $A_j = \pm 1$, then the corresponding basis vector is fixed ($A_j = +1$) or stabilized (A_j = -1), while if A_j is the matrix of Equation 2.20, then two basis vectors define a plane that is stabilized (but not fixed) by the operation ϕ [4].

Finally the block structure of $\mathbf{A}$ is preserved when the matrix is raised to a power:

$$\mathbf{A^k} = \begin{pmatrix} {A_1}^k & 0 & 0 & \dots \\ 0 & {A_2}^k & 0 & \dots \\ \vdots & \vdots & \ddots & \\ \dots & \dots & \dots & {A_m}^k \end{pmatrix} \tag{2.21}$$

Thus the order of ϕ is the least common multiple of the orders of $A_1, A_2, \cdots, A_m$.

2.4.2. Symmetry Groups

The maximal group of isometries of an object that maps it onto itself (stabilizes it) is the symmetry group of it. As we have seen in the previous section, all the linear isometries are just group representations of the corresponding symmetries of a lattice,

so they can be written in the form of Equation 2.19 with respect to proper basis. However not every isometry can be written in this form with respect to the common basis. When all the stabilizers of an object can be written in the same block form with respect to a single basis, as

$$\begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} \tag{2.22}$$

where B_1 is $k \times k$, B_2 is $m \times m$, and $k > 0$, $m = n - k > 0$, then the symmetry group is said to be *reducible*; otherwise it is *irreducible* [4].

Any symmetry group which involves the linear isometries (orthogonal transformations) belonging to a lattice, and whose elements have a common fixed point can be represented as a subgroup of the *orthogonal group* $O(n)$ (by choosing the origin to be a fixed point). This is true for all finite symmetry groups, and also the symmetry groups of bounded figures. Hence the linear isometries of E^n form the group $O(n)$.

In $O(2)$ there are two infinite families of symmetry groups, cyclic rotation groups C_n and the dihedral groups D_n. The reducible subgroups of $O(3)$ have stable planes and stable or fixed lines orthogonal to them. There are several infinite families; all of them can be regarded as generalizations of C_n and D_n. The family of irreducible groups is infinite: it contains only the regular polyhedra and some of their subgroups [3, 4].

To understand the concept *crystallographic group*, we need to define another property for groups in general;

Index Of a Subgroup For the group G and the proper subgroup H (so, $H \neq G$) the index of the H defined as, $[G : H] = \frac{O(G)}{O(H)}$. Here O stands for the orders of the groups, the number of element that belong to them.

The symmetry group G of lattice points is a crystallographic group. Then G has

a maximal invariant translation subgroup (Abelian) T of finite index hence there are some translational types of actions which maps the lattice onto itself, and the system is a union of a finite number of point lattices.

The points of a lattice have integer coordinates with respect to any basis $\vec{b}_1, \ldots, \vec{b}_n$ of $\mathcal{L}$; it follows that the finite group of linear isometries P that maps T onto itself can be represented by a finite group of unimodular matrices with integer entries. In other words, P can be identified with a finite subgroup of $GL\,(n, Z)$ [4].

Although every element of P maps the lattice onto itself, neither its fixed point set nor its stable sets need to contain any lattice points other than the origin. This leads us to the criteria for a subspace to be totally irrational.

Irrationality of a Subspace Let $\mathcal{L}$ be a lattice and ε any k-dimensional subspace of E^n, where $0 < k < n$. If $\varepsilon \cap \mathcal{L} = \{0\}$ then ε is said to be totally irrational.

If we again talk about the simplest case, let's first think about a 2-dimensional integral point lattice ($\mathcal{L}$). A lower dimensional hyperspace (subspace) can be visualized as a 1-dimensional line (ε). To accomplish the irrationality condition ($\varepsilon \cap \mathcal{L} = \{0\}$) we can choose this line with a irrational slope because any line which has rational slope certainly passes through finitely many points of the lattice [4, 6].

Although the lattices that we will use in quasicrystal theory has many point symmetries (especially rotational symmetries) most of the lattices just have two types of symmetries. Namely the *identity isometry* I_n which maps the lattice onto itself and the isometry that maps every point to its image through the origin $-I_n$.

2.4.3. The Crystallographic Restriction

As we saw in Sec. 2.4.2 all the generator matrices for an integral point lattice $\mathcal{L}$ belong to the general linear group with integer entries $GL\,(n, Z)$. Thus we can use this algebraic property to prove the *crystallographic restriction* for two and three

dimensional lattices [4].

The crystallographic restriction If R is an element of the point group of a two or three dimensional lattice $\mathcal{L}$, then its order two, three, four, or six.

For the case $n = 2$ (two dimensional lattice) we can represent R as a 2×2 matrix and we can reduce it to a rotation matrix using the proper basis. So it is;

$$R = \begin{pmatrix} cos\theta & -\sin\theta \\ sin\theta & \cos\theta \end{pmatrix} \tag{2.23}$$

This matrix is an element of the $GL(n, Z)$ with $n = 2$, and all of the matrices belong to this group act as rotations with respect to different basis. It follows that, every matrix should have the same trace, and the traces of them are integers since the entries are. So the trace;

$$2\cos(\theta) \in Z \tag{2.24}$$

But $|\cos(\theta)| \leq 1$ hence $\cos(\theta) \in \{0, \pm\frac{1}{2}, \pm 1\}$. Assuming that $0 \leq \theta < 2\pi$ only possibilities are;

$$\theta \in \{0, \pi, \frac{\pi}{2}, \frac{2\pi}{3}, \frac{2\pi}{6}\} \tag{2.25}$$

For the case of $n = 3$ the matrices will be the type;

$$R = \begin{pmatrix} cos\theta & -\sin\theta & 0 \\ sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.26}$$

Then the trace will be;

$$2\cos(\theta) + 1 \in Z \tag{2.27}$$

With the same assumption the possibilities of the values of θ are $\cos(\theta) \in \{0, \pm\frac{1}{2}, \pm 1\}$.

$$\theta \in \{0, \pi, \frac{\pi}{2}, \frac{2\pi}{3}, \frac{2\pi}{6}\} \tag{2.28}$$

It can be easily seen that the other groups contain rotations of order 5, 7, or 8, ... are not the crystallographic point groups. Hence they are called as *non-crystallographic point groups*. Nevertheless these are very important for quasi-periodic structures (*quasicrystals*) [4].

2.4.4. Orbits of Z-modules

In 2.1, the orbits of Z-modules show all the possibilities for $n = 2$ and $k = 3$, because of the following theorem;

The structure of orbits of Z-modules : Let $\phi : R^k \to R^n$ be a surjective (onto) linear mapping, where $n < k$. Then there are (possibly trivial) subspaces V,W of R^n such that $R^n = V \oplus W$ where the direct sum of these means;

$$\begin{pmatrix} V & 0 \\ 0 & W \end{pmatrix} \tag{2.29}$$

and

(i) $\phi(Z^k) = \phi(Z^k) \cap W + \phi(Z^k) \cap V$,
(ii) $\phi(Z^k) \cap W$ is a discrete point lattice in W,
(iii) $\phi(Z^k) \cap V$ is a dense subgroup of V.

Hence for the Figure 2.1(a) $\vec{b}_3$ is a rational linear combinartion of $\vec{b}_1$ and $\vec{b}_2$, and an integral linear combination of $\frac{2}{15}\vec{b}_1$ and $\frac{3}{15}\vec{b}_2$. Also $V = \{0\}$ and dim $W = 2$; here we can think of V as an irrational subspace and W as a rational one. Therefor there is no dense subset for this lattice so this lattice is discrete. For the case Figure 2.1(b)

dim V = dim W = 1 because $\vec{b}_3$ is the sum of a rational multiple of $\vec{b}_1$ and an irrational multiple of $\vec{b}_2$, and we have relatively 1-dimensional lattices. Finally in Figure 2.1(c) dim $V = 2$ and $W = \{0\}$; this is due to the fact that the coefficients of $\vec{b}_3$ are irrational multiples of both $\vec{b}_1$ and $\vec{b}_2$. This is the reason for having a 2-dimensional dense lattice.

A corollary to the statement can be established as;

Every orbit Ω of a Z-module in E^n is the orthogonal projection of a point lattice in E^k onto E^n, where $k \geq n$.

Furthermore, in this way we can cover every rational and irrational subset with a suitable choice of basis. This choice thereby allows us to create quasi-periodic lattices in different dimensions as we show next [4].

2.5. Canonical Projection Method

There are many point lattices which can not be described by symmetries other than the identity, but even we can't see their symmetries at first glance some of them have unexpected symmetries. Penrose tilings do not obey the crystallographic restriction that we described in Sec. 2.4.3 [3]. The reasons for this are;

- They are not monohedral tilings so they use at least two types of tiles to cover the space.
- The congruency between these tiles have some special rules which prevent them to show translational symmetry. So they are not repetitive.

Now we know that this type of non-periodicity is caused from the irrationality of the Penrose tilings. One can deal with these type of tilings with the help of the projection method.

The projection method is a powerful technique for constructing nonperiodic patterns and tilings. In this section we describe the special case of it *canonical projection.*

Let ε be a totally irrational d-dimensional subspace of E^n, and let $\varepsilon^\perp$ be its orthogonal complement (treat it as $E^n = \varepsilon \oplus \varepsilon^\perp$ where $\varepsilon^\perp$ can be totally irrational or not). Let Π be the orthogonal projector onto ε, and $\Pi^\perp$ orthogonal projector onto $\varepsilon^\perp$. Since Π and $\Pi^\perp$ are linear maps, $\Pi(\mathcal{L})$ and $\Pi^\perp(\mathcal{L})$ are orbits of Z-modules in ε and $\varepsilon^\perp$, respectively, generated by the projections of any (every) basis of $\mathcal{L}$ [4].

When $\mathcal{L}$ is an integral lattice, then the following statemens are equivalent:

(i) $\Pi(\mathcal{L})$ is everywhere dense in ε;
(ii) $\varepsilon \cap \mathcal{L} = \{0\}$;
(iii) $\Pi^\perp|_{(\mathcal{L})}$ is one to one.

Since $\Pi(\mathcal{L})$ is dense in ε, it is not a Delone set. To obtain one, we must select a subset of points of $\mathcal{L}$ for projection.

The first step is to fix a compact subset $K \subset \varepsilon^\perp$ with nonempty interior; K will be called the *window*, or acceptance domain, for the projection. We will project onto ε, those points $x \in \mathcal{L}$ such that $\Pi(x)^\perp \in K$. These are the points that lie in the cylinder $C = K \oplus \varepsilon$. Let $X = C \cap \mathcal{L}$ [4, 7, 8]. Figure 2.16;

$\Pi(X)$ is a Delone set. Furher, if ε is totally irrational, then $\Pi(X)$ is nonperiodic.

Thus we have a simple but powerful way of constructing nonperiodic Delone sets by projecting subsets of higher-dimensional lattices:

Selection Criterion : Project $x \in \mathcal{L}$ orthogonally onto ε if and only if $\Pi^\perp(x)$ lies in the window K [4, 8].

The *canonical* choice for the windows is $\Pi(V(0))$, where V(0) is the Voronoï cell of the origin of $\mathcal{L}$.

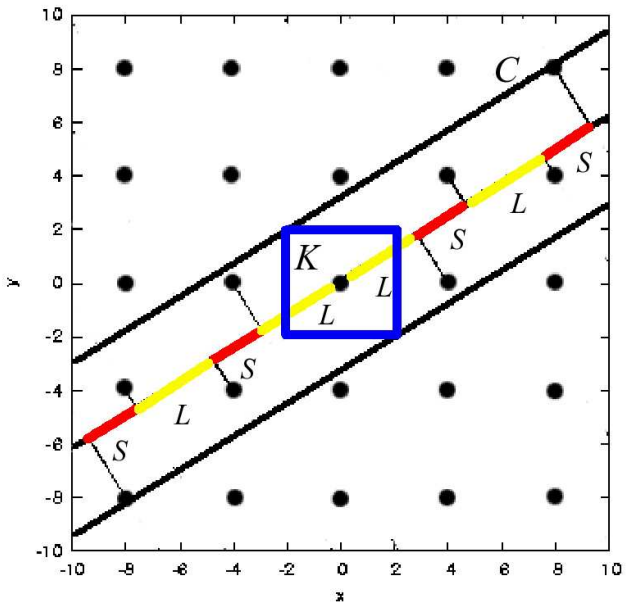


Figure 2.16. The projection window for the $\mathcal{I}_2$ lattice

For the canonical projection (K = $\Pi(V(0))$) the following statements are equivalent [4].

(i) $x \in X$;

(ii) $\Pi^{\perp}(x) \in \Pi^{\perp}(V(0))$;

(iii) $\varepsilon \cap V(x) \neq \varnothing$.

From Figure 2.16, and the definition of X ($X = C \cap \mathcal{L}$), if the satatement i is true for some x then the point x should be inside of the cylider (C) or the strip. Hence the perpendicular distance of (indeed its y coordinate after the projection) x to the plane ε should be smaller than the width of the strip, and the width of strip arranged by the vertices of V(0). For the next statement, let us call the projection of x onto ε space (as we can think of it as a vector from the origin) $\vec{x}^{\parallel}$, for any x if there is a vector $\vec{x}^{\parallel}$ such that $\vec{x} + \vec{x}^{\parallel} \in V(0)$ then $\vec{e} \in V(x)$.

3. QUASICRYSTALS IN ONE AND TWO DIMENSIONS

3.1. Quasicrystals in 1-D (Fibonacci Chain)

The simplest quasicrystal can be constructed from a 2-dimensional cubic lattice. To construct the structure we have to find two subspaces for the 2-d space so that these correspond the subspaces ε and $\varepsilon^{\perp}$.

The reasonable choice for the space ε is a line with an irrational slope. The other subspace $\varepsilon^{\perp}$ is another line which is perpendicular to the space ε. The Voronoï cell around the origin is again a square and its vertices are in the points $\left(\pm\frac{1}{2}, \pm\frac{1}{2}\right)$.

To have the Fibonacci chain we choose the slope as $\frac{1}{\tau}$ for the subspace ε [4]. Therefore the slope for the $\varepsilon^{\perp}$ subspace is $-\tau$. When we select the points that are in the cylinder C, their projections on to the subspace ε give two different length scales. These length scales correspond to two vectors. Hence the constructed lattice is a one dimensional but the Z-module has two vectors.

All the procedure that we explained above are illustrated in Figure 2.16. Two length scales are shown as S and L. The scale S corresponds to a length 1.376382 in arbitrary units, while the L corresponds 2.227033 in arbitrary units, for the 2-dimensional cubic lattice with the unit lattice spacing. The resultant chain is shown below in Equation 3.1, which starts from the origin and goes to the infinity symmetrically in both directions.

$$\{L,\ S,\ L,\ S,\ L,\ L,\ S,\ L,\ S,\ L,\ L\ \ldots\} \tag{3.1}$$

3.2. Quasicrystals in 2-D (Penrose rhombus tiling)

The main property of all the Penrose type tilings is that although they do not have any translational symmetry, all of them have five-fold rotational symmetry. This means the Z-modules corresponding to the lattices have five vectors. To reach the five-fold symmetrical basis we chose $\mathcal{I}_5$ [4, 9].

Now the cartesian basis in $\mathcal{I}_5$ are;

$$\vec{e}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \vec{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \vec{e}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \vec{e}_4 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \vec{e}_5 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.2}$$

We have to find another set of basis to express the five-fold symmetry. We therefore choose a new set of basis using the symmetry of the Voronoï cell which is a 5-dimensional hypercube. The five-fold rotation about the body diagonal (so called $\vec{w} = (1, 1, 1, 1, 1)$ is a symmetry of the hypercube, since the rotation permutes the five basis vectors $\vec{e}_1, \vec{e}_1, \cdots$ cyclically. With respect to this basis, the cyclic rotation matrix is

$$\mathcal{A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.3}$$

Utilizing the eigen-basis of this matrix we can find a new set of basis vectors that

decompose the 5-d space into 5-fold symmetric subspaces.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ \rho \\ \rho^2 \\ \rho^3 \\ \rho^4 \end{pmatrix} = \begin{pmatrix} \rho^4 \\ 1 \\ \rho \\ \rho^2 \\ \rho^3 \end{pmatrix} = \frac{1}{\rho} \begin{pmatrix} \rho^5 \\ \rho \\ \rho^2 \\ \rho^3 \\ \rho^4 \end{pmatrix} \tag{3.4}$$

So, the eigenvalues of $\mathcal{A}$ are fifth roots of unity;

$\rho = e^{\frac{2n\pi i}{5}}$ where $n = 0, 1, 2, 3, 4$. And the normalized eigenvectors are

$$\vec{\rho_1} = \frac{1}{\sqrt{5}} \begin{pmatrix} 1 \\ e^{\frac{2\pi}{5}i} \\ e^{\frac{4\pi}{5}i} \\ e^{\frac{6\pi}{5}i} \\ e^{\frac{8\pi}{5}i} \end{pmatrix}, \vec{\rho_2} = \frac{1}{\sqrt{5}} \begin{pmatrix} 1 \\ e^{\frac{4\pi}{5}i} \\ e^{\frac{8\pi}{5}i} \\ e^{\frac{2\pi}{5}i} \\ e^{\frac{6\pi}{5}i} \end{pmatrix}, \vec{\rho_3} = \frac{1}{\sqrt{5}} \begin{pmatrix} 1 \\ e^{\frac{6\pi}{5}i} \\ e^{\frac{2\pi}{5}i} \\ e^{\frac{8\pi}{5}i} \\ e^{\frac{4\pi}{5}i} \end{pmatrix},$$

$$\vec{\rho_4} = \frac{1}{\sqrt{5}} \begin{pmatrix} 1 \\ e^{\frac{8\pi}{5}i} \\ e^{\frac{6\pi}{5}i} \\ e^{\frac{4\pi}{5}i} \\ e^{\frac{2\pi}{5}i} \end{pmatrix}, \vec{\rho_5} = \frac{1}{\sqrt{5}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{3.5}$$

The only real eigenvalue is 1; the others are complex and occur in conjugate pairs, ρ , ρ^4 and ρ^2 , ρ^3. This means E^5 decomposes into two two-dimensional planes as ε and

$\varepsilon^{\perp}$, both stable under the action of these set of basis, both totally irrational, and both orthogonal to each other and the fixed space Δ generated by $\vec{w}$.

The vectors $\vec{e}_j$ project to the vertices of a regular pentagon in both planes, but in one of them (say, ε) the angle between the projections of $\vec{e}_j$ and $\vec{e}_{j+1}$ is $\frac{2\pi}{5}$ while in the other ($\varepsilon^{'}$), it is $\frac{4\pi}{5}$. So we have $\varepsilon^{\perp} = \varepsilon^{'} \oplus \Delta$. And because of Δ the perpendicular subspace is not totally irrational. We postpone the schematic confirmations of our assertion until the construction of projection matrix ϕ.

Now we have the eigenvalues and eigenvectors, but to get an real lattice we need the real vectors which define $\varepsilon^{\perp}$ and ε. Taking the appropriate compositions of the vectors and then re-normalize them, we find that $\vec{\xi_1} = \vec{\rho}_1 + \vec{\rho}_4$, $\vec{\xi_2} = i(\vec{\rho}_1 - \vec{\rho}_4)$, $\vec{\xi_3} = \vec{\rho}_2 + \vec{\rho}_3$, $\vec{\xi_4} = i(\vec{\rho}_2 - \vec{\rho}_3)$, so that;

$$\vec{\xi_1} = \frac{2\sqrt{2}}{5}\begin{pmatrix} 1 \\ -\cos(\frac{3\pi}{5}) \\ -\cos(\frac{\pi}{5}) \\ -\cos(\frac{\pi}{5}) \\ -\cos(\frac{3\pi}{5}) \end{pmatrix}, \vec{\xi_2} = \frac{2\sqrt{2}}{5}\begin{pmatrix} 0 \\ -\sin(\frac{3\pi}{5}) \\ -\sin(\frac{\pi}{5}) \\ \sin(\frac{\pi}{5}) \\ \sin(\frac{3\pi}{5}) \end{pmatrix}, \vec{\xi_3} = \frac{2\sqrt{2}}{5}\begin{pmatrix} 1 \\ -\cos(\frac{\pi}{5}) \\ -\cos(\frac{3\pi}{5}) \\ -\cos(\frac{3\pi}{5}) \\ -\cos(\frac{\pi}{5}) \end{pmatrix},$$

$$\vec{\xi_4} = \frac{2\sqrt{2}}{5}\begin{pmatrix} 0 \\ -\sin(\frac{\pi}{5}) \\ \sin(\frac{3\pi}{5}) \\ -\sin(\frac{3\pi}{5}) \\ \sin(\frac{\pi}{5}) \end{pmatrix}, \vec{\xi_5} = \frac{1}{\sqrt{5}}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{3.6}$$

Then the projection matrix ϕ^{-1} could be written as the inverse of the matrix ϕ whose

column vectors are $\vec{\xi_1}$, $\vec{\xi_2}, \ldots, \vec{\xi_5}$. The matrices ϕ and ϕ^{-1} are as follows

$$\phi = \frac{2\sqrt{2}}{5} \begin{pmatrix} 1 & 0 & 1 & 0 & \frac{\sqrt{5}}{2\sqrt{2}} \\ -\cos(\frac{3\pi}{5}) & -\sin(\frac{3\pi}{5}) & -\cos(\frac{\pi}{5}) & -\sin(\frac{\pi}{5}) & \frac{\sqrt{5}}{2\sqrt{2}} \\ -\cos(\frac{\pi}{5}) & -\sin(\frac{\pi}{5}) & -\cos(\frac{3\pi}{5}) & \sin(\frac{3\pi}{5}) & \frac{\sqrt{5}}{2\sqrt{2}} \\ -\cos(\frac{\pi}{5}) & \sin(\frac{\pi}{5}) & -\cos(\frac{3\pi}{5}) & -\sin(\frac{3\pi}{5}) & \frac{\sqrt{5}}{2\sqrt{2}} \\ -\cos(\frac{3\pi}{5}) & \sin(\frac{\pi}{5}) & -\cos(\frac{\pi}{5}) & \sin(\frac{\pi}{5}) & \frac{\sqrt{5}}{2\sqrt{2}} \end{pmatrix} \tag{3.7}$$

$$\phi^{-1} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -\cos(\frac{3\pi}{5}) & -\cos(\frac{\pi}{5}) & -\cos(\frac{\pi}{5}) & -\cos(\frac{3\pi}{5}) \\ 0 & -\sin(\frac{3\pi}{5}) & -\sin(\frac{\pi}{5}) & \sin(\frac{\pi}{5}) & \sin(\frac{3\pi}{5}) \\ 1 & -\cos(\frac{\pi}{5}) & -\cos(\frac{3\pi}{5}) & -\cos(\frac{3\pi}{5}) & -\cos(\frac{\pi}{5}) \\ 0 & -\sin(\frac{\pi}{5}) & \sin(\frac{3\pi}{5}) & -\sin(\frac{3\pi}{5}) & \sin(\frac{\pi}{5}) \\ \frac{\sqrt{2}}{\sqrt{5}} & \frac{\sqrt{2}}{\sqrt{5}} & \frac{\sqrt{2}}{\sqrt{5}} & \frac{\sqrt{2}}{\sqrt{5}} & \frac{\sqrt{2}}{\sqrt{5}} \end{pmatrix} \tag{3.8}$$

The assertion about the decomposition of the space E^5 can be proven with a similarity transformation of the rotation matrix $\mathcal{A}$ with respect to the projection matrices ϕ and its inverse ϕ^{-1}. The transformed matrix is in the form that we proposed in Sec. 2.4.1 and illustrated in Equations 2.19 and 2.20. This is shown in Equation 3.9

$$\phi^{-1}\mathcal{A}\phi = \begin{pmatrix} \cos\left(\frac{-2\pi}{5}\right) & -\sin\left(\frac{-2\pi}{5}\right) & 0 & 0 & 0 \\ \sin\left(\frac{-2\pi}{5}\right) & \cos\left(\frac{-2\pi}{5}\right) & 0 & 0 & 0 \\ 0 & 0 & \cos\left(\frac{-4\pi}{5}\right) & -\sin\left(\frac{-4\pi}{5}\right) & 0 \\ 0 & 0 & \sin\left(\frac{-4\pi}{5}\right) & \cos\left(\frac{-4\pi}{5}\right) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.9}$$

To project the lattice onto ε we act ϕ^{-1} on all the lattice points, then we drop the final three coordinates, and for the projection on to the $\varepsilon^{\perp}$ we should drop the first

two coordinates. Similarly the hypercubic basis vectors $\vec{e}_1\,,\vec{e}_2\,,\ldots\,,\vec{e}_5$ can be projected onto these subspaces. The projections of the cartesian basis onto the ε and $\varepsilon^{\perp}$ spaces form two new types of vector families. We call these new vectors $\vec{p}_1,\vec{p}_2,\cdots,\vec{p}_5$ and $\vec{q}_1,\vec{q}_2,\cdots,\vec{q}_5$. These two sets of vectors form two pentagons in each space with a slight difference as follows, while in the projection onto the ε space the angle between the vertex vectors are $\frac{2\pi}{5}$, onto the $\varepsilon^{\perp}$ space they are $\frac{4\pi}{5}$. Hence the vertex coordinates of pentagons in ε and $\varepsilon^{\perp}$ space coordinates are;

$$\vec{p}_1=\begin{pmatrix}\frac{1}{\sqrt{2}}\\0\end{pmatrix},\vec{p}_2=\begin{pmatrix}-\frac{1}{\sqrt{2}}\cos\left(\frac{3\pi}{5}\right)\\-\frac{1}{\sqrt{2}}\sin\left(\frac{3\pi}{5}\right)\end{pmatrix},\vec{p}_3=\begin{pmatrix}-\frac{1}{\sqrt{2}}\cos\left(\frac{\pi}{5}\right)\\-\frac{1}{\sqrt{2}}\sin\left(\frac{\pi}{5}\right)\end{pmatrix},$$

$$\vec{p}_4=\begin{pmatrix}-\frac{1}{\sqrt{2}}\cos\left(\frac{\pi}{5}\right)\\\frac{1}{\sqrt{2}}\sin\left(\frac{\pi}{5}\right)\end{pmatrix},\vec{p}_5=\begin{pmatrix}-\frac{1}{\sqrt{2}}\cos\left(\frac{3\pi}{5}\right)\\\frac{1}{\sqrt{2}}\sin\left(\frac{3\pi}{5}\right)\end{pmatrix}. \tag{3.10}$$

$$\vec{q}_1=\begin{pmatrix}\frac{1}{\sqrt{2}}\\0\end{pmatrix},\vec{q}_2=\begin{pmatrix}-\frac{1}{\sqrt{2}}\cos\left(\frac{\pi}{5}\right)\\-\frac{1}{\sqrt{2}}\sin\left(\frac{\pi}{5}\right)\end{pmatrix},\vec{q}_3=\begin{pmatrix}-\frac{1}{\sqrt{2}}\cos\left(\frac{3\pi}{5}\right)\\\frac{1}{\sqrt{2}}\sin\left(\frac{3\pi}{5}\right)\end{pmatrix},$$

$$\vec{q}_4=\begin{pmatrix}-\frac{1}{\sqrt{2}}\cos\left(\frac{3\pi}{5}\right)\\-\frac{1}{\sqrt{2}}\sin\left(\frac{3\pi}{5}\right)\end{pmatrix},\vec{q}_5=\begin{pmatrix}-\frac{1}{\sqrt{2}}\cos\left(\frac{\pi}{5}\right)\\\frac{1}{\sqrt{2}}\sin\left(\frac{\pi}{5}\right)\end{pmatrix}. \tag{3.11}$$

The next step is identification of the Voronoï cell ($V(0)$). Construction of the Voronoï cell, includes the action of the projection matrix ϕ on to the 5D hypercubic unit cell which is centered at the origin $(0,0,0,0,0)$. The matrix projects the vertices of the unit cell whose coordinates $\left(\pm\frac{1}{2}\,,\pm\frac{1}{2}\,,\pm\frac{1}{2}\,,\pm\frac{1}{2}\,,\pm\frac{1}{2}\right)$ such that the last three

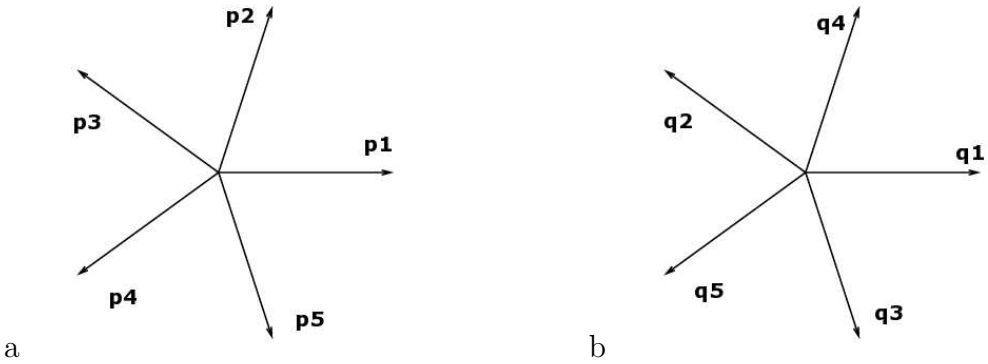


Figure 3.1. The projection of hypercubic lattice basis onto the (a) ε and (b) $\varepsilon^{\perp}$

coordinates denote the vertices of a polytope in 3D which is a *rhombic icosahedron*, which is shown in Figure 3.2.

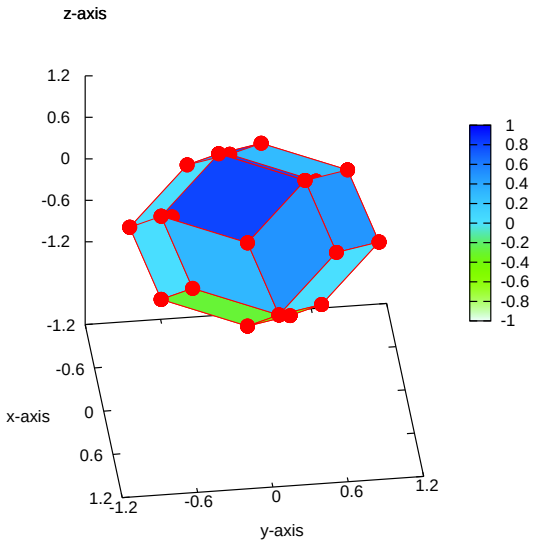


Figure 3.2. The projection window rhombic icosahedron

Because the hypercubic lattice is already highly symmetric including the 5-fold and 2-fold symmetries, the polytope again has the same symmetries with the hypercubic lattice, see Figures 3.2 and 3.2. Furthermore, the polytope is convex and every face of it is a rhombus as a result from the projection of the hypercubic lattice. These faces come in symmetric pairs with respect to the origin. As a final remark, the polytope has twenty two vertices, forty edges and twenty faces [9, 10].

After finding the projection window, with the help of the edge vectors we find

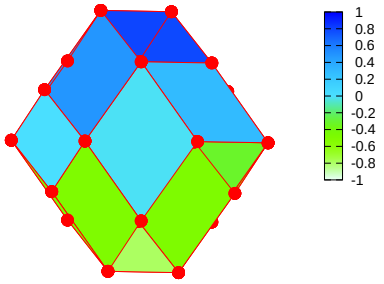


Figure 3.3. Rotated view perpendicular to a 2-fold axis of the rhombic-icosahedron

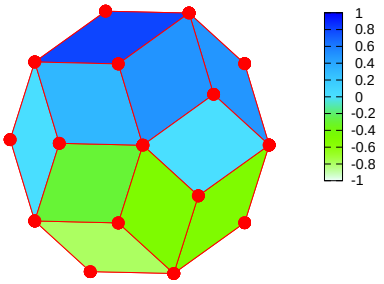
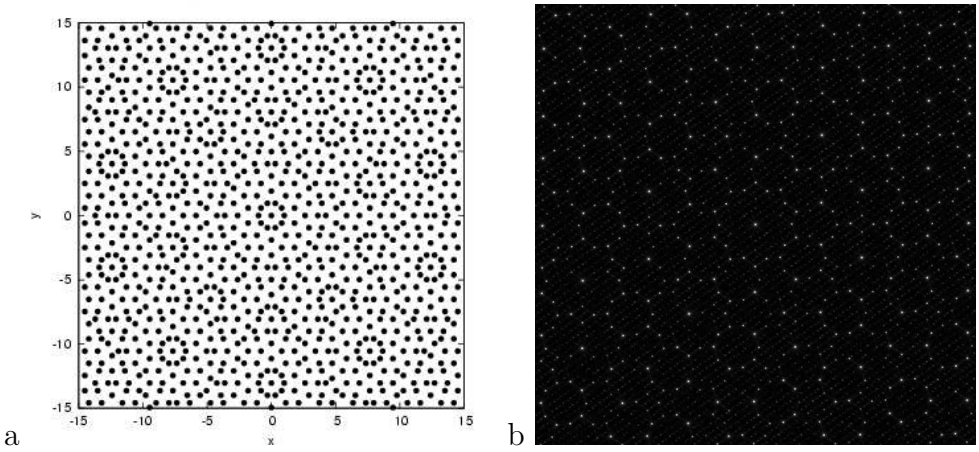


Figure 3.4. Rotated view perpendicular to a 5-fold axis of the rhombic-icosahedron

the surface normals. Not surprisingly they are related such that for any facet vector $\vec{f}$ there is another vector of the same magnitude but with its direction reversed.

After obtaining the projection of the Voronoï cell $V(0)$ we find the canonical projection window. The elements of the generated hypercubic lattice which are in the canonical projection window *rhombic icosahedron* constitute the vertex set of the Penrose rhomb tiling.



a b

Figure 3.5. Real (a) and fourier space (b) images of Penrose tiling

4. GENERATION AND STRUCTURAL ANALYSIS OF THREE DIMENSIONAL ICOSAHEDRAL QUASICRYSTAL

4.1. Quasicrystals in 3-D (The $\mathcal{I}_6$ quasicrystal)

For the 2 dimensional quasicrystal we used a 5D hypercubic lattice and then we found two sets of basis vectors that decompose a by-product of the hyperspace into two orthogonal subspaces (row vectors of ϕ^{-1}). The basis vectors of the E^5 is projected onto the ε and $\varepsilon^{\perp}$ with the help of these sets of vectors, see Figure 3.1. These projections give two new sets of vectors that span the spaces ε ($\vec{\rho}_i$) and $\varepsilon^{\perp}$ ($\vec{\xi}_j$). These basis vectors also carry the symmetry of the resulting lattice.

As we stated before in Sec. 3.2, the number of vectors in the Z-module of the lattice signs the dimensionality of the hypercubic lattice. Therefore, in the construction of icosahedrally symmetric quasicrystal we start with the simple integral hypercubic lattice in six-dimensions ($\mathcal{I}_6$). Then the cartesian basis vectors are given by;

$$\vec{e}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \vec{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \vec{e}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

$$\vec{e}_4 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \vec{e}_5 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \vec{e}_6 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{4.1}$$

For the 3-dimensional quasi crystal, we follow a different interpretation of the Canonical projection method. In 2-d we first find a new set of basis vectors that have the 5-fold symmetry with the utilization of the rotation matrix $\mathcal{A}$. But in the 3-d case, we first utilize the vectors $\vec{p}_i$ and $\vec{q}_i$, Equations 3.10, and 3.11, that are the projections of the cartesian basis in E^6 onto the ε and $\varepsilon^{\perp}$.

Hence, to construct ϕ^{-1} we utilize a set of basis vectors that are vectors corresponding to six vertices of icosahedron, whence they are the symmetry carriers of a regular icosahedron as shown in Figure 4.1. These vertex vectors are projections of the cartesian basis in six dimensional space onto the three dimensional subspace ε, like in the case of 2-dimensional quasicrystal [11, 12]. As a remark, the subspace ε and $\varepsilon^{\perp}$ are called the *physical space* and the *dual space*. The vectors which span these spaces are called $|e_i^{\parallel}\rangle$ and $|e_i^{\perp}\rangle$. The vectors which arise from projection are as follows

$$\vec{p}_1 = \begin{pmatrix} 1 \\ \tau \\ 0 \end{pmatrix}, \vec{p}_2 = \begin{pmatrix} \tau \\ 0 \\ 1 \end{pmatrix}, \vec{p}_3 = \begin{pmatrix} 0 \\ 1 \\ \tau \end{pmatrix},$$

$$\vec{p}_4 = \begin{pmatrix} -1 \\ \tau \\ 0 \end{pmatrix}, \vec{p}_5 = \begin{pmatrix} \tau \\ 0 \\ -1 \end{pmatrix}, \vec{p}_6 = \begin{pmatrix} 0 \\ -1 \\ \tau \end{pmatrix} \tag{4.2}$$

where the number τ is the *golden mean* $\frac{1+\sqrt{5}}{2}$.

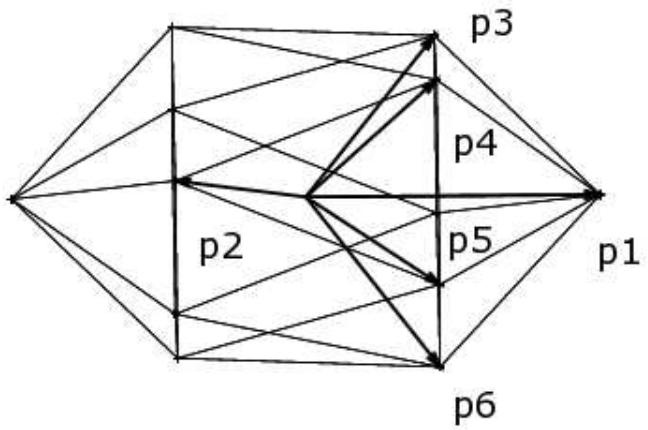


Figure 4.1. The parallel space projection vectors and the corresponding icosahedron

Using the Dirac notation for the vectors, let us find the orthogonal basis for the physical space

$$\begin{aligned}
\left\langle e_1^{\parallel} \mid e_1 \right\rangle = 1, \left\langle e_2^{\parallel} \mid e_1 \right\rangle = \tau, \left\langle e_3^{\parallel} \mid e_1 \right\rangle = 0, \\
\left\langle e_1^{\parallel} \mid e_2 \right\rangle = \tau, \left\langle e_2^{\parallel} \mid e_2 \right\rangle = 0, \left\langle e_3^{\parallel} \mid e_2 \right\rangle = 1, \\
\left\langle e_1^{\parallel} \mid e_3 \right\rangle = 0, \left\langle e_2^{\parallel} \mid e_3 \right\rangle = 1, \left\langle e_3^{\parallel} \mid e_3 \right\rangle = \tau, \\
\left\langle e_1^{\parallel} \mid e_4 \right\rangle = -1, \left\langle e_2^{\parallel} \mid e_4 \right\rangle = \tau, \left\langle e_3^{\parallel} \mid e_4 \right\rangle = 0, \\
\left\langle e_1^{\parallel} \mid e_5 \right\rangle = \tau, \left\langle e_2^{\parallel} \mid e_5 \right\rangle = 0, \left\langle e_3^{\parallel} \mid e_5 \right\rangle = -1, \\
\left\langle e_1^{\parallel} \mid e_6 \right\rangle = 0, \left\langle e_2^{\parallel} \mid e_6 \right\rangle = -1, \left\langle e_3^{\parallel} \mid e_6 \right\rangle = \tau,
\end{aligned} \tag{4.3}$$

The orthonormality condition is

$$\left\langle e_i^{\parallel} \mid e_j^{\parallel} \right\rangle = \delta_{ij} \Rightarrow \sum_{k=1}^{6} \left\langle e_i^{\parallel} \mid e_k \right\rangle \left\langle e_k \mid e_j^{\parallel} \right\rangle = \delta_{ij}. \tag{4.4}$$

The normalization constant is

$$\begin{aligned}
\left\langle e_1^{\parallel} \mid e_1^{\parallel} \right\rangle = \sum_{k=1}^{6} \left\langle e_1^{\parallel} \mid e_k \right\rangle \left\langle e_k \mid e_1^{\parallel} \right\rangle = 2 + 2\tau^2, \\
\left\langle e_2^{\parallel} \mid e_2^{\parallel} \right\rangle = \sum_{k=1}^{6} \left\langle e_2^{\parallel} \mid e_k \right\rangle \left\langle e_k \mid e_2^{\parallel} \right\rangle = 2 + 2\tau^2, \\
\left\langle e_3^{\parallel} \mid e_3^{\parallel} \right\rangle = \sum_{k=1}^{6} \left\langle e_3^{\parallel} \mid e_k \right\rangle \left\langle e_k \mid e_3^{\parallel} \right\rangle = 2 + 2\tau^2.
\end{aligned} \tag{4.5}$$

All the other combinations are equal to zero, for instance

$$\left\langle e_1^{\parallel} \mid e_2^{\parallel} \right\rangle = \sum_{k=1}^{6} \left\langle e_1^{\parallel} \mid e_k \right\rangle \left\langle e_k \mid e_2^{\parallel} \right\rangle = 0. \tag{4.6}$$

As a result, the normalization constant will be $\kappa = \frac{1}{\sqrt{2(1+\tau^2)}}$,and we can construct the real (parallel) space projector $M^{\parallel}$ as;

$$M^{\parallel} = \sum_{i=1}^{3} \sum_{j=1}^{6} \mid e_i^{\parallel} \rangle \left\langle e_i^{\parallel} \mid e_j \right\rangle \langle e_j \mid . \tag{4.7}$$

We finally obtain the projector $M^{\parallel}$ as

$$\mathbf{M}^{\parallel} = \kappa \begin{pmatrix} 1 & \tau & 0 & -1 & \tau & 0 \\ \tau & 0 & 1 & \tau & 0 & -1 \\ 0 & 1 & \tau & 0 & -1 & \tau \end{pmatrix} \tag{4.8}$$

In the same spirit we can construct the perpendicular (dual) space projector $M^{\perp}$. Indeed we just need to find a set of vectors that are mutually orthogonal to the parallel space vectors, Equations 2.3, and 4.2, except that the factor 2π. A convenient choice is

$$\vec{q}_1 = \begin{pmatrix} -\tau \\ 1 \\ 0 \end{pmatrix}, \vec{q}_2 = \begin{pmatrix} 1 \\ 0 \\ -\tau \end{pmatrix}, \vec{q}_3 = \begin{pmatrix} 0 \\ -\tau \\ 1 \end{pmatrix}, \tag{4.9}$$

$$\vec{q_4} = \begin{pmatrix} \tau \\ 1 \\ 0 \end{pmatrix}, \vec{q_5} = \begin{pmatrix} 1 \\ 0 \\ \tau \end{pmatrix}, \vec{q_6} = \begin{pmatrix} 0 \\ \tau \\ 1 \end{pmatrix} \tag{4.10}$$

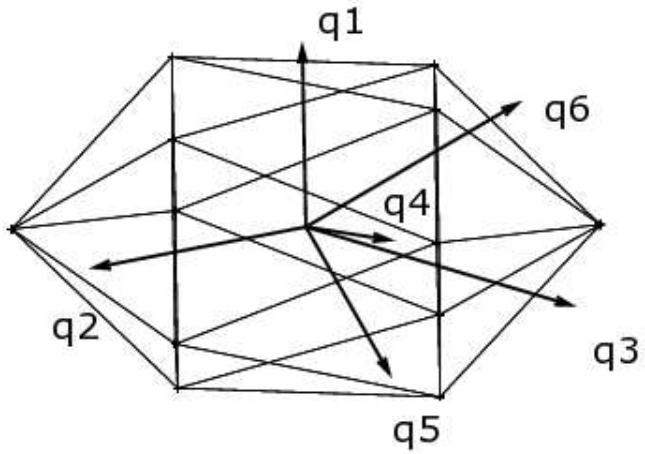


Figure 4.2. The perpendicular space projection vectors

The procedure for the next steps is as before, and we obtain the perpendicular space projector $M^{\perp}$ as

$$\mathbf{M}^{\perp} = \kappa \begin{pmatrix} -\tau & 1 & 0 & \tau & 1 & 0 \\ 1 & 0 & -\tau & 1 & 0 & \tau \\ 0 & -\tau & 1 & 0 & \tau & 1 \end{pmatrix} \tag{4.11}$$

The projection matrix ϕ^{-1} is obtained by summing these two projectors as;

$$\phi^{-1} = \left(\sum_{i=1}^{3} \mid e_i^{\parallel} \rangle \langle e_i^{\parallel} \mid + \sum_{j=1}^{3} \mid e_i^{\perp} \rangle \langle e_i^{\perp} \mid \right) \tag{4.12}$$

$$\phi^{-1} = \kappa \begin{pmatrix} 1 & \tau & 0 & -1 & \tau & 0 \\ \tau & 0 & 1 & \tau & 0 & -1 \\ 0 & 1 & \tau & 0 & -1 & \tau \\ -\tau & 1 & 0 & \tau & 1 & 0 \\ 1 & 0 & -\tau & 1 & 0 & \tau \\ 0 & -\tau & 1 & 0 & \tau & 1 \end{pmatrix} \tag{4.13}$$

Using the matrix ϕ we transform the lattice, to get the new coordinates of hypercube centered at the origin (0,0,0,0,0,0). After the projection onto the $\varepsilon^{\perp}$, we get four types of vertices with respect to the distance from the origin. The twelve distances have the largest moduli $0.1441\ldots$ $(\frac{\tau}{2\sqrt{5}})$ that belongs to the vertices of type $\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{-1}{2}\right)$ and the twenty next largest distances $0.6967\ldots$ $(\sqrt{0.3(\tau+2)})$ which belongs the vertices of the type $\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{-1}{2}, \frac{-1}{2}\right)$ construct the vertices of a *rhombic triacontahedron*, which will be our Voronoï cell [12, 13]. The polytope *rhombic triacontahedron* has thirty two vertices, sixty edges, and thirty faces. A picture is provided in Figure 4.3.

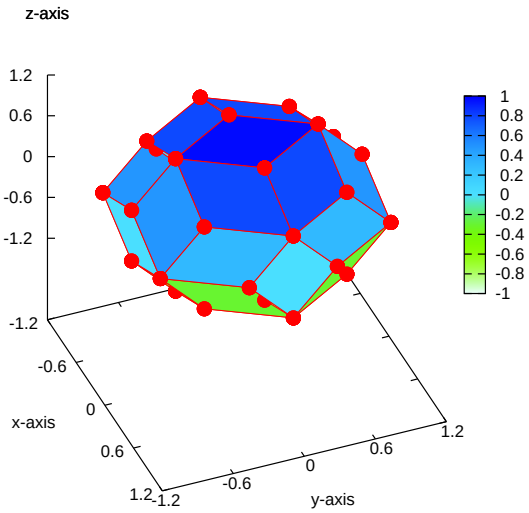


Figure 4.3. The polytope rhombic triacontahedron

After the construction of the Voronoï cell, we project the whole lattice under the projector ϕ^{-1} and we choose the vertices that are inside of the Voronoï cell. These set of vertices give us an icosahedrally symmetric 3-dimensional lattice. For the symmetries of

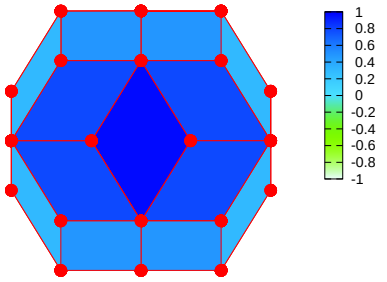


Figure 4.4. View from a 2-fold symmetry direction

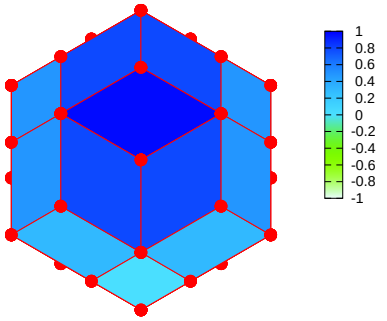


Figure 4.5. View from a 3-fold symmetry direction

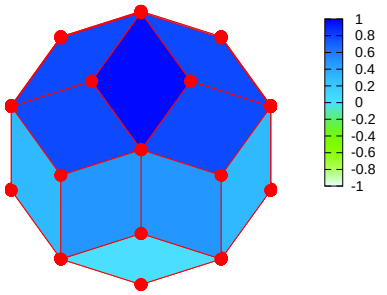


Figure 4.6. View from a 5-fold symmetry direction

the generated lattice, we utilize the perpendicular vector set $(\vec{q}_1, \ldots, \vec{q}_6)$. The reasoning behind this is as follows; any element of this set is identifiable as being parallel to the 5-fold axes of a suitable icosahedron. Hence we can observe in the direction of any of these vectors the stacking of the 5-fold symmetric planes. Likewise, for the 3-fold symmetric stackings we can utilize the vectors $\vec{q}_1 + \vec{q}_2 - \vec{q}_4$ or $\vec{q}_2 + \vec{q}_3 - \vec{q}_5$, etc. and finally for the 2-fold symmetric stacking $\vec{q}_1 + \vec{q}_2$, $\vec{q}_2 + \vec{q}_3$, etc. can be identified as the normal vectors [15].

In order to compare the generated quasicrystal with the real structure we placed

the aluminum atoms into the calculated coordinates. The software Crystal-Maker is utilized for the purpose [16]. The plane projected views are shown in the Figure 4.7 a) from 2-fold, Figure 4.7 b) from 3-fold, Figure 4.7 c) from 5-fold symmetry planes.

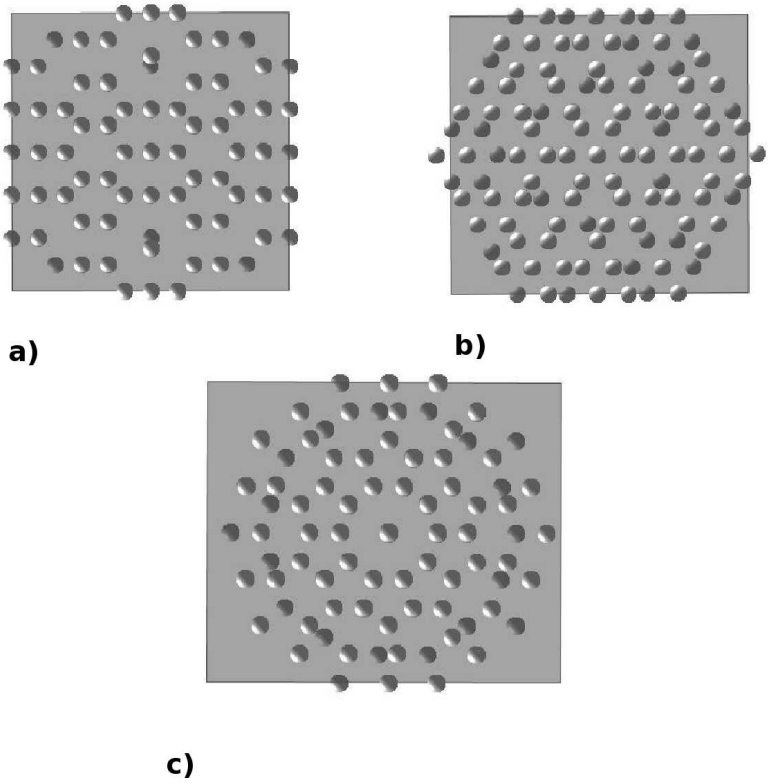


Figure 4.7. Projections of the generated crystal from a) 2-f plane, b) 3-fold plane, b) 5-fold plane of symmetry

We can also show the relative plane orientations as follows in Figure 4.8.

4.2. Structural Properties of the Generated Quasicrystal

Having generated the icosahedral quasicrystalline structure, we can investigate its physical properties. Here we will focus on the densities and the layer stackings of the generated quasicrystal.

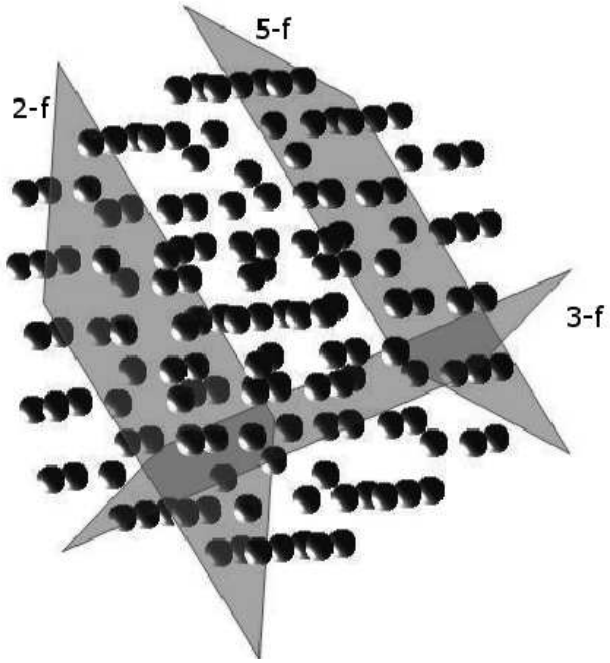


Figure 4.8. Planes of projections of the generated quasicrystal.

4.2.1. Layers and The Layer Densities

Aperiodicity in all directions is one of the main properties of icosahedral structure. Although, we can find the symmetry planes in different directions, every plane of symmetry has aperiodicity in itself. Indeed, through the symmetry axis, their positions are again aperiodic. This property inspired us to think of the structure as including isotropic aperiodicity. Morever, when we consider the generator matrix Equation 4.13, [12, 13], this seems a natural consequence for the generated structure.

Visualising the 3-dimensional quasicrystal as a stacking of layers that exhibit either a 5-fold, 3-fold or 2-fold symmetry, we find for each stacking that the spacing between the planes takes one of three values S, M or L as shown below in Table 4.1. Moreover the sequence of stackings appears to be aperiodic. Below we show a sequence of distances for the stacking in terms of 5-fold symmetry planes.

$$\{\, L, L, S, M, S, L, L, L, S, M, S, L, L, L, L, S, M, S, L, L, L \ldots \}$$

The aperiodicity in layer spacings has been approximated as the Fibonacci chain in some works [18], but this is not a proven property.

Table 4.1. Relative heights for different symmetry planes (in arbitrary units)

	2-fold	3-fold	5-fold
S	0.09	0.13	0.08
M	0.14	0.17	0.24
L	0.23	0.30	0.32

The directions for the symmetry axes can be easily found with the help of the basis vectors that we present in the previous section. For the calculation of plane densities, we project the quasicrystal along the symmetry axis, and we take a cubic part from the structure. We obtain the square sections for each layer of side length 30 units. The areal density is obtained from calculating the mean number of atoms in the layer.

As we already mentioned, these densities are in arbitrary units. But, we can use the experimental distance for the scaling of the quasicrystal such that for 5-fold planes 4.561 Å [14]. When we do this, we see the results are in a good agreement with other similar works on different models and the real crystal (Al-Pd-Mn) [14]. The relative seperations between successive layers are shown in Table 4.2 and the Figure 4.9. Figures 4.10, 4.11, 4.12 show the variations of the layer densities as a function of heiht for the 2, 3, and 5-fold symmetric layers.

Table 4.2. Re-scaled relative heights for different symmetry planes

	2-fold	3-fold	5-fold
S	0.581 Å	0.838 Å	0.516 Å
M	0.903 Å	1.097 Å	1.548 Å
L	1.484 Å	1.935 Å	2.064 Å

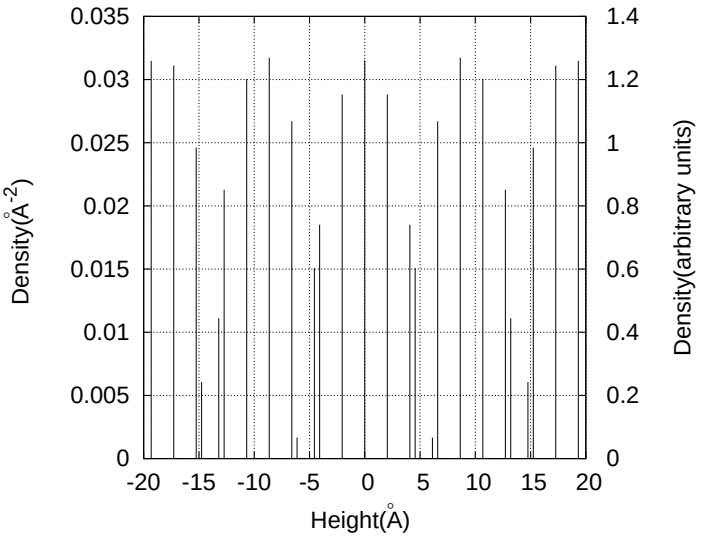


Figure 4.9. Plane stacking in 5-fold symmetry direction

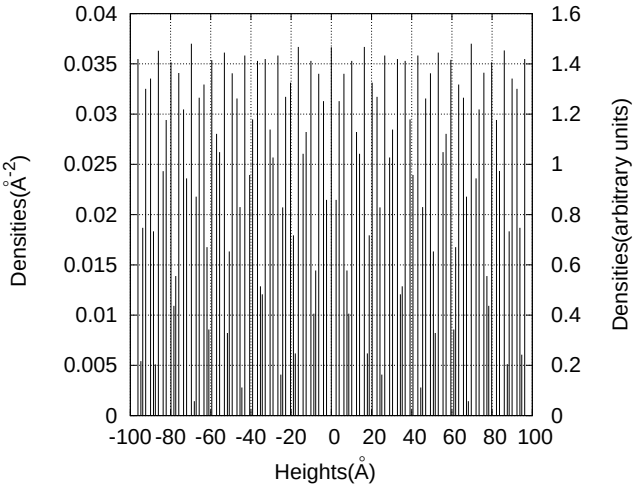


Figure 4.10. Heights versus densities for the 2-fold planes

4.2.2. Bulk and Plane Coordination Numbers

The coordination number is simply the number of nearest neighbors bonded to a cenral atom. The crystalline structures can be decomposed the successive parallel layers by means of their symmetry axes. In fact, with this procedure there arise another concept, namely plane coordination numbers. The coordination numbers of the whole crystal referred to the bulk coordination numbers are smaller than the plane coordination numbers. Maybe the easiest example to visualize is the body centered cubic lattice and the layer stacking of it through the $(1, 0, 0)$ direction. Here the average

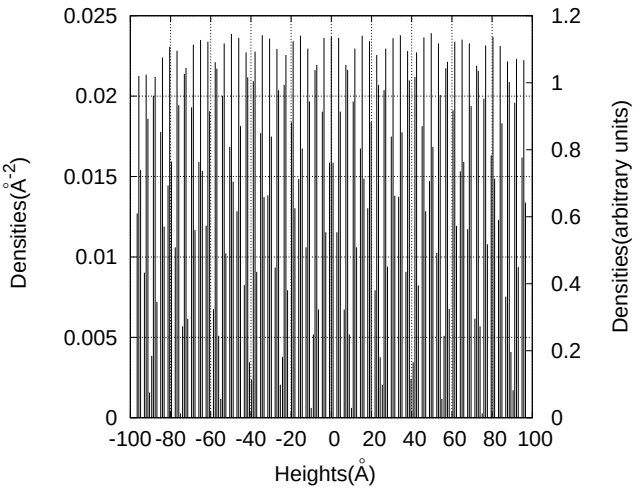


Figure 4.11. Heights densities for the 3-fold planes

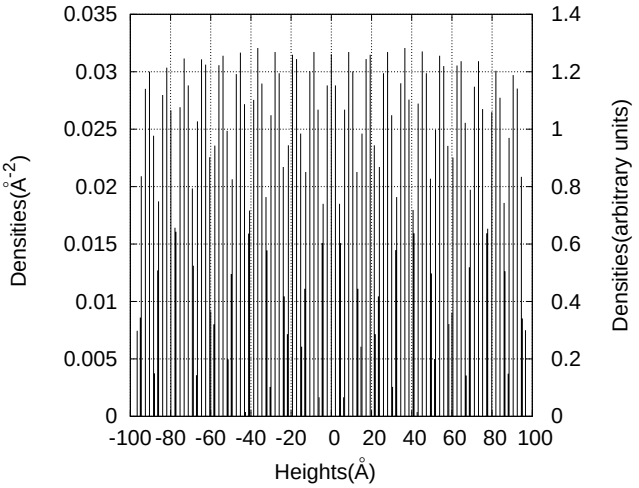


Figure 4.12. Heights densities for the 5-fold planes

bulk coorination numbers for the structure is 8, but the coordination numbers for the above defined planes are just 4. At this point, the quasicrystals start to deviate from the conventional crystal structures. As we shown before the densities change for both the positions and the number of atoms of the layers. These facts already give us clues about these two types of structures i.e crystals and quasicrystals.

Moreover, in a more realistic approach the coordination number should be related with the diameters of the atoms that constitute the whole structure. But, here we can not identify the atomic types that belong to the quasicrystal. Corollary, the only way to figure out the coordination number concept can be the evaluation of minimum distance

among the predicted atomic positions. Hence we try to find an average distance and we try to observe the average layer coordination numbers with respect to this distance.

Now, the first thing to do here is the calculation of minimum distance among the atoms. Unfortunately, in quasicrystals this approach can not be directly used. The reason is simply that there are at least 3 different length scales for layer structures with respect to their symmetries. When we analyse the projected views that belong to different symmetry axes, we can easily see that the minimum distance is contributed by the 2-fold symmetry planes. For the other (3-fold and 5-fold) planes we have other length scales. Hence we should start with an approximation so that we use a common length scale which gives a true average coordination number for the bulk.

We evaluate the structure in that spirit and we start to calculate the average coordination number for the bulk from the minimum distance (0.4 in arbitrary units) to a distance (2.3 in arbitrary units) where the average coordination number comes to an unrealistic point (around 203 per atom), see Table 4.3 and Figure 4.2.2.

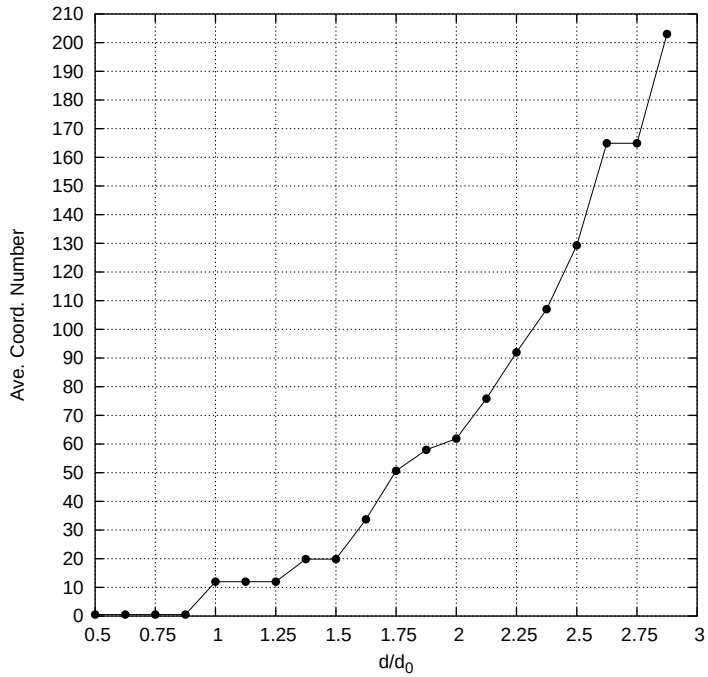


Figure 4.13. The average coordination numbers as a function of the distance scales

Table 4.3. The average coordination numbers for different distance scales

distance	Ave. Coord. Number
0.4	0.50653
0.5	0.50653
0.6	0.50653
0.7	0.50653
0.8	11.9696
0.9	11.9696
1.0	11.9696
1.1	19.8405
1.2	19.8405
1.3	33.7222
1.4	50.6497
1.5	57.9836
1.6	61.9085
1.7	75.8609
1.8	91.9605
1.9	107.034
2.0	129.297
2.1	164.889
2.2	164.889
2.3	203.038

Now we use another approximation for the choice of the common length scale, because we have an icosahedrally symmetric quasicrystal, the coordination structure of the atoms constitutes either icosahedron or pentagonal bipyramid. Hence, for the icosahedral coordinations we count the coordination number as 12 and for the pentagonal bipyramid it will be 10. By this way, we choose the length scale which corresponds the average coordination number of bulk that is between 10 and 12. This classification seems acceptable when we investigate the possible vertex structure of icosahedral quasicrystals [17].

Based on above approximation the choice will be the distance which corresponds to 11.96 average coordination number per atom. This ratio means that the icosahedral coordinations cover the 98 % of the structure while the pentagonal bipyramid coordinations occur in 2 % of the structure. Then the distance taken to be 0.8 in arbitrary units.

Now we handle another part of the coordination number calculations, namely the plane coordination numbers. For the purpose we again use the same approximation but in this time another constraint comes into play that is the plane coordination numbers can not be greater than the bulk coordination numbers.

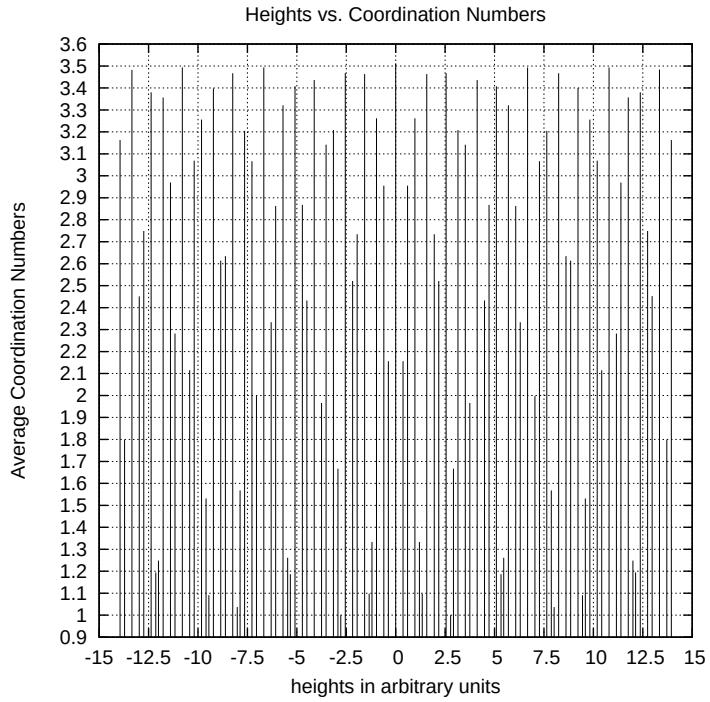


Figure 4.14. Average coordination numbers per atom with respect to plane heights through a 2-fold axis

Figures 4.14, 4.15 and 4.16 show the variations of the average plane coordination number as a function of the height of the plane. The average planar coordination number is defined to be the number of atoms in a layer that are at most a distance d_0 away from an atom. We choose $d_0 = 0.8$ in arbitrary units.

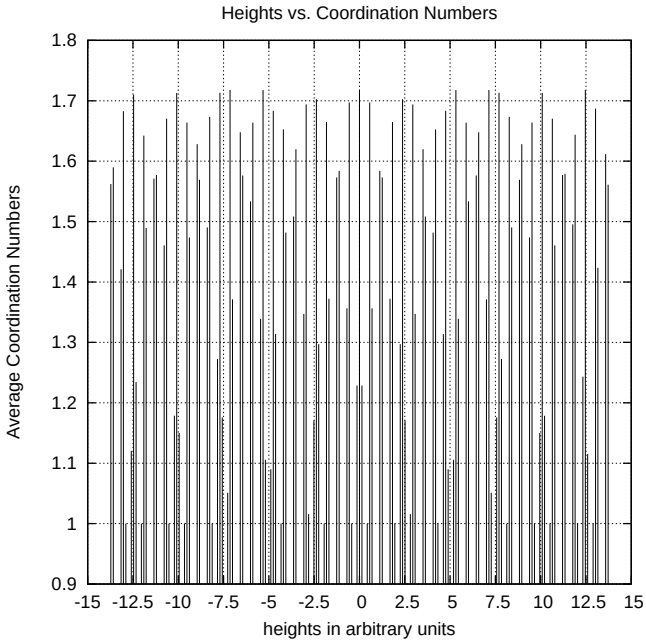


Figure 4.15. Average coordination numbers per atom with respect to plane heights through a 3-fold axis

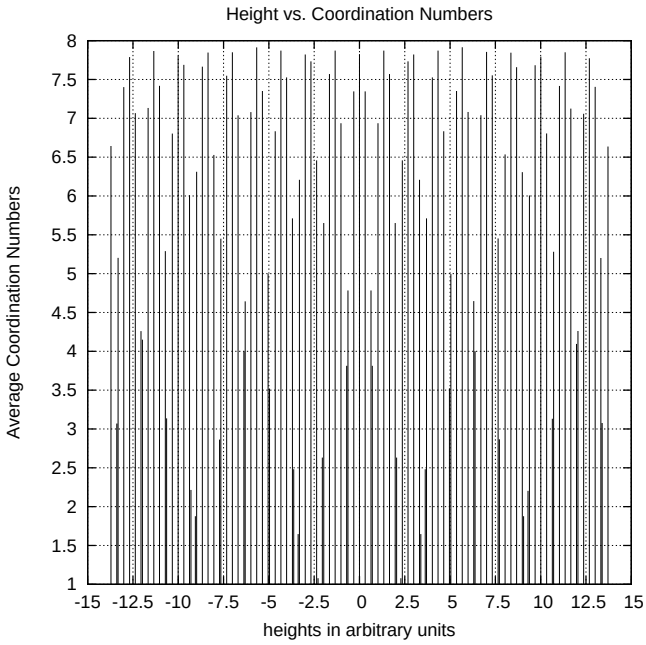


Figure 4.16. Average coordination numbers per atom with respect to plane heights through a 5-fold axis

5. CONCLUSIONS

In this thesis, we discuss the aperiodic point sets and the canonical projection method which is a widely used method for generation of the icosahedral structures (quasicrystals) . We next generated 1-d, 2-d and 3-d quasicrystals. We finally analyze the structural properties of the icosahedral 3-d quasicrystal.

We have shown that the generation methods are based on the Z-modules of the point sets. Any orbit of a Z-module which belongs to a aperiodic set of points has a greater rank than the rank of the space that includes the set. Thus aperiodic sets need a higner dimensional embedding space where they become periodic. Thus a quasicrystal turn out to be a conventional crystal that is translationaly symmetric with the help of extra dimensions.

Canonical projection is a technique that utilizes the embedding concept, to construct the aperiodic point sets in different dimensions. The distinguishing feature of the canonical projection is that its projection window is the Voronoï cell of the lattice around the origin.

As we have shown in our 2-d and 3-d examples, the canonical projection method can be interpreted in different ways. In the construction of 2-d structure we utilized the symmetry of the lattice, while the real and the dual subspaces were not defined primarily. Nevertheless in 3-d construction, we firstly define these subspaces. This is the mere difference between the two interpretations. The remaining parts of the procedure is same for both of them.

After generating from mathematical model the 3-d icosahedral quasicrystal, we performed an analysis of its physical properties. We start with the plane densities of the icosahedral structure through different symmetry axes. Naturally, the structure has 2,3, and 5-fold symmetry axes, so every calculation is made with respect to all of these symmetry planes. As a starting point, we scale the length scale for the 5-fold axis

as 4.561 Å (to be compatible with an icosahedral Al-Mn-Pd quasicrystal). Then we use a more physical sample for our calculations. The outputs of the density calculations are consistent with some other similar work on different models [14], where the plane densities belong to the 5-fold symmetric planes are calculated 0.06 $Å^{-2}$ as maximum. The results are interesting, because in 2-fold and 5-fold directions the plane densities fluctuate considerably, in sharp contrast to the 3-fold planes where there are again some fluctuations but not as big. Another point about the densities is that the 3-fold planes seems to be dilute relative to the other two types of planes (nevertheless, their densities are closer to each other through the lattice planes). As a result, the 2-fold and 5-fold planes seem to more densely populated and we therefore expect the atoms in these planes to be more strongly bound to each other.

As a final analysis of the structure we consider the average coordination numbers for the whole lattice and the symmetry planes. Here, we used an average coordination distance (0.8 in arbitrary units or 5.16 Å) that we can determine from the bulk lattice. In this calculation we make an approximation for the coordination types, and we use two types of coordinations. These are namely an icosahedral type which corresponds to 12 coordination on the average, and the pentagonal bipyramid type which induces 10 coordination on the average. Then the final result turns out to be there is an average coordination number per atom 11.96 and this implies that 98 % of the lattice points have 12 coordinations (icosahedral type).

In the next step we analysed the plane coordinations. We know from the density analysis that the 3-fold planes are less dense relative to the others likewise the average coordination numbers are again smaller than the others. This situation mimics us for the 3-fold planes as there are at least two types of in-plane coordination structures (say, triangular and line-like) but these are homogeneously distributed throughout the plane. This shown in Figure 5.1

For other types of the plane coordinations, for instance the 5-fold planes induce an average coordination number around 5.5 and 6.5. So, under the distance d_0 the average coordination number predicts that in the 5-fold symmetric planes there should

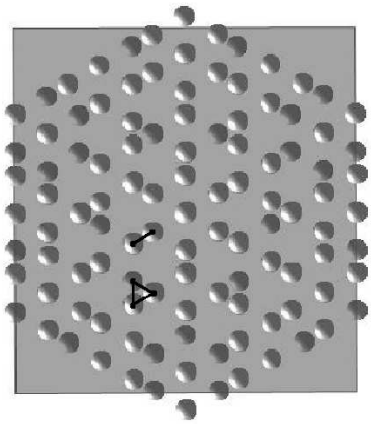


Figure 5.1. Coordination structure in 3-fold planes

be more than five neighbors for each of the atoms. This result is can be visualized in Figure 5.2 and this isteresting result have been seen in another simulation work, See. [19].

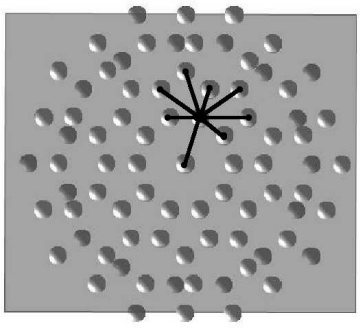


Figure 5.2. Coordination structure in 5-fold planes

APPENDIX A: ALGORITHMS

A.1. *Penrose.c*

```
1. #include < stdio.h >
2. #include < stdlib.h >
3. #include < math.h >
4. #define d 5
5. #define N 16
6. #define n 2 // #of steps to construct the unitcube
7. #define A 1.0
8. #define pi 4.0 ∗ atanl(1.0)
9. #define C (1.0/sqrt(5.0))
10. #define beta (2.0 ∗ pi)/5.0
11. #define tau 0.5 ∗ (1.0 + sqrt(5.0))
12. #define uver 32 // #of vertices of unitcube
13. #define ver 22 // #of vertices of rhombicicosahedron
14. #define faces 20 // #of faces of rhombicicosahedron
15. #define poss 216 // #of possibilities to constructable surfaces
16. #define eps 0.00001
17. #define delta 0.00001
18. int neighs [d ∗ uver] , listlen [uver] , surfprev [poss] , surface [poss] , surfacefin [faces ∗ 4] ,
    symm [faces] ;
19. float x [uver] [d] , xnew [uver] [d] , xx [d] , xxnew [d] ;
20. float lines [uver] , distance [uver] , vec [d] [d] ;
21. float ranvec [ver] [4] , center [faces] [3] , surfvec [2 ∗ faces] [3] , surfnormal [faces] [3] ;
22. float check [faces] ;
23. float Sqr(float x)
24. {
25.     float i;
26.     i = x ∗ x;
27.     returni;
28. }
29. float normal(float x1, float x2, float x3, float x4, float x5)
30. {
31.     float j;
32.     j = sqrt(Sqr(x1) + Sqr(x2) + Sqr(x3) + Sqr(x4) + Sqr(x5));
33.     returnj;
34. }
35. void rotvectors()
36. {
37.     vec [0] [0] = C ∗ 1.0;
38.     vec [0] [1] = C ∗ sqrt(2.0);
39.     vec [0] [2] = C ∗ 0.0;
40.     vec [0] [3] = C ∗ sqrt(2.0);
```

```
41.     vec [0] [4] = C * 0.0;
42.     vec [1] [0] = C * 1.0;
43.     vec [1] [1] = C * sqrt(2.0) * cos(beta);
44.     vec [1] [2] = -C * sqrt(2.0) * sin(beta);
45.     vec [1] [3] = C * sqrt(2.0) * cos(3.0 * beta);
46.     vec [1] [4] = C * sqrt(2.0) * sin(3.0 * beta);
47.     vec [2] [0] = C * 1.0;
48.     vec [2] [1] = C * sqrt(2.0) * cos(3.0 * beta);
49.     vec [2] [2] = C * sqrt(2.0) * sin(3.0 * beta);
50.     vec [2] [3] = C * sqrt(2.0) * cos(beta);
51.     vec [2] [4] = C * sqrt(2.0) * sin(beta);
52.     vec [3] [0] = C * 1.0;
53.     vec [3] [1] = C * sqrt(2.0) * cos(3.0 * beta);
54.     vec [3] [2] = -C * sqrt(2.0) * sin(3.0 * beta);
55.     vec [3] [3] = C * sqrt(2.0) * cos(beta);
56.     vec [3] [4] = -C * sqrt(2.0) * sin(beta);
57.     vec [4] [0] = C * 1.0;
58.     vec [4] [1] = C * sqrt(2.0) * cos(beta);
59.     vec [4] [2] = C * sqrt(2.0) * sin(beta);
60.     vec [4] [3] = C * sqrt(2.0) * sin(3.0 * beta);
61.     vec [4] [4] = -C * sqrt(2.0) * sin(3.0 * beta);
62. }
63. void normalize()
64. {
65.     int i, j, k;
66.     float norm, sum;
67.     for(i = 0; i <= d - 1; i + +){
68.         sum = 0.0;
69.         for(j = 0; j <= d - 1; j + +){
70.             sum = sum + Sqr(vec [i] [j]);
71.         }
72.         norm = sqrt(sum);
73.         for(k = 0; k <= d - 1; k + +){
74.             vec [i] [k] = vec [i] [k] /norm;
75.         }
76.     }
77. }
78. void makerotatedunitcube()
79. {
80.     int i, j, k, l, m, r, s, p;
81.     float ii, jj, kk, ll, mm, sum;
82.     p = 0;
83.     for(i = 0; i <= n - 1; i + +){
84.         for(j = 0; j <= n - 1; j + +){
85.             for(k = 0; k <= n - 1; k + +){
86.                 for(l = 0; l <= n - 1; l + +){
87.                     for(m = 0; m <= n - 1; m + +){
88.                         ii = (float )i;
```

```
89.                                 jj = (float )j;
90.                                 kk = (float )k;
91.                                 ll = (float )l;
92.                                 mm = (float )m;
93.                                 x [p] [0] = (ii − 0.5) ∗ A;
94.                                 x [p] [1] = (jj − 0.5) ∗ A;
95.                                 x [p] [2] = (kk − 0.5) ∗ A;
96.                                 x [p] [3] = (ll − 0.5) ∗ A;
97.                                 x [p] [4] = (mm − 0.5) ∗ A;
98.                                 for(r = 0; r <= d − 1; r + +){
99.                                     sum = 0.0;
100.                                    for(s = 0; s <= d − 1; s + +){
101.                                        sum = sum + x [p] [s] ∗ vec [s] [r] ;
102.                                    }
103.                                    xnew [p] [r] = sum;
104.                                }
105.                                p + +;
106.                            }
107.                        }
108.                    }
109.                }
110.            }
111.    }
112.    void findvertices()
113.    {
114.        int i, j;
115.        float ordistcal, ordist, rr;
116.        for(i = 0; i <= uver − 1; i + +){
117.            ordistcal = Sqr(xnew [i] [0]) + Sqr(xnew [i] [3]) + Sqr(xnew [i] [4]);
118.            ordist = sqrt(ordistcal);
119.            distance [i] = ordist;
120.            lines [i] = distance [i] ;
121.        }
122.        for(i = 0; i <= uver − 1; i + +){
123.            for(j = 0; j <= uver − 2; j + +){
124.                if(distance [j + 1] > distance [j]){
125.                    rr = distance [j] ;
126.                    distance [j] = distance [j + 1] ;
127.                    distance [j + 1] = rr;
128.                }
129.                else{
130.                    distance [j] = distance [j] ;
131.                    distance [j + 1] = distance [j + 1] ;
132.                }
133.            }
134.        }
135.    }
136.    void findneighbours()
```

```
{
    int p, q, r, t, i, j, k;
    float distcal, dist;
    float a, b, c;
    p = 0;
    for(i = 0; i <= uver − 1; i + +){
        for(j = 0; j <= uver − 1; j + +){
            distcal = 0.0;
            for(k = 0; k <= d − 1; k + +){
                distcal = distcal + Sqr(x [i] [k] − x [j] [k]);
            }
            dist = sqrt(distcal);
            if(i! = j){
                if((dist − 1.0) < eps){
                    a = lines [i] ;
                    b = distance [ver − 1] ;
                    c = lines [j] ;
                    if((a ≥ b)&&(c ≥ b)){
                        neighs [p] = j;
                        p + +;
                    }
                    else{
                        neighs [p] = uver;
                        p + +;
                    }
                }
            }
        }
    }
    p = 0;
    q = 0;
    r = 0;
    for(i = 0; i <= uver − 1; i + +){
        t = 0;
        p = i ∗ d;
        for(j = 0; j <= d − 1; j + +){
            q = neighs [p + j] ;
            if(q! = uver){
                neighs [p + t] = q;
                t + +;
            }
        }
        listlen [r] = t;
        if(t! = d){
            for(k = p + t; k <= p + d − 1; k + +){
                neighs [k] = uver;
            }
        }
```

```
185.            r++;
186.        }
187.    }
188.    void identifysurfaces()
189.    {
190.        int p, q, r, s, t, i, j, k, l, s1, s2, p1, p2, t1, t2, pivot, pivot1, pivot2;
191.        p = 0;
192.        q = 0;
193.        s = 0;
194.        r = 0;
195.        for(i = 0; i <= uver - 1; i++){
196.            s = listlen[i];
197.            if(s! = 0){
198.                for(j = 0; j <= s - 1; j++){
199.                    q = i * d;
200.                    p = q + j;
201.                    pivot = neighs[q];
202.                    if(j < s - 1){
203.                        pivot1 = neighs[p];
204.                        pivot2 = neighs[p + 1];
205.                        s1 = listlen[pivot1];
206.                        s2 = listlen[pivot2];
207.                        for(k = 0; k <= s1 - 1; k++){
208.                            p1 = pivot1 * d + k;
209.                            for(l = 0; l <= s2 - 1; l++){
210.                                p2 = pivot2 * d + l;
211.                                t1 = neighs[p1];
212.                                t2 = neighs[p2];
213.                                if(t1 == t2){
214.                                    if(t1! = i){
215.                                        surfprev[r] = i;
216.                                        surfprev[r + 1] = pivot1;
217.                                        surfprev[r + 2] = pivot2;
218.                                        surfprev[r + 3] = t1;
219.                                        r = r + 4;
220.                                    }
221.                                }
222.                            }
223.                        }
224.                    }
225.                    else{
226.                        pivot1 = neighs[p];
227.                        pivot2 = pivot;
228.                        s1 = listlen[pivot1];
229.                        s2 = listlen[pivot2];
230.                        for(k = 0; k <= s1 - 1; k++){
231.                            p1 = pivot1 * d + k;
232.                            for(l = 0; l <= s2 - 1; l++){
```

```
233.                                    p2 = pivot2 * d + l;
234.                                    t1 = neighs[p1];
235.                                    t2 = neighs[p2];
236.                                    if(t1 == t2){
237.                                        if(t1! = i){
238.                                            surfprev[r] = i;
239.                                            surfprev[r + 1] = pivot1;
240.                                            surfprev[r + 2] = pivot2;
241.                                            surfprev[r + 3] = t1;
242.                                            r = r + 4;
243.                                        }
244.                                    }
245.                                }
246.                            }
247.                        }
248.                    }
249.                }
250.            }
251.            p = 0;
252.            for(i = 0; i <= (poss/4) - 1; i++){
253.                p = i * 4;
254.                for(j = 0; j <= 4 - 1; j++){
255.                    surface[p + j] = surfprev[p + j];
256.                }
257.            }
258.            p = 0;
259.            q = 0;
260.            for(i = 0; i <= (poss/4) - 1; i++){
261.                p = i * 4;
262.                for(k = 0; k <= d; k++){
263.                    for(j = 0; j <= 3 - 1; j++){
264.                        q = p + j;
265.                        pivot1 = surfprev[q];
266.                        pivot2 = surfprev[q + 1];
267.                        if(pivot1 > pivot2){
268.                            t = surfprev[q];
269.                            surfprev[q] = surfprev[q + 1];
270.                            surfprev[q + 1] = t;
271.                        }
272.                    }
273.                }
274.            }
275.            p = 0;
276.            q = 0;
277.            for(i = 0; i <= (poss/4) - 2; i++){
278.                p = i * 4;
279.                for(j = i + 1; j <= (r/4) - 1; j++){
280.                    q = j * 4;
```

```
281.                if(surfprev [p] == surfprev [q]){
282.                    if(surfprev [p + 1] == surfprev [q + 1]){
283.                        if(surfprev [p + 2] == surfprev [q + 2]){
284.                            if(surfprev [p + 3] == surfprev [q + 3]){
285.                                for(k = 0; k <= 4 - 1; k + +){
286.                                    surfprev [q + k] = uver;
287.                                }
288.                            }
289.                        }
290.                    }
291.                }
292.            }
293.        }
294.        p = 0;
295.        q = 0;
296.        for(i = 0; i <= (poss/4) - 1; i + +){
297.            p = i * 4;
298.            if(surfprev [p]! = uver){
299.                surfacefin [q] = surface [p] ;
300.                surfacefin [q + 1] = surface [p + 1] ;
301.                surfacefin [q + 2] = surface [p + 2] ;
302.                surfacefin [q + 3] = surface [p + 3] ;
303.                q = q + 4;
304.            }
305.        }
306.  }
307.  void plotrhombicicosahedron()
308.  {
309.        int i, p, q, pivot1, pivot2, pivot3, pivot4;
310.        FILE * fp;
311.        fp = fopen("point s.txt","w");
312.        p = 0;
313.        q = 0;
314.        for(i = 0; i <= (poss/4) - 1; i + +){
315.            p = i * 4;
316.            if(surfprev [p]! = uver){
317.                pivot1 = surface [p] ;
318.                pivot2 = surface [p + 1] ;
319.                pivot3 = surface [p + 2] ;
320.                pivot4 = surface [p + 3] ;
321.                fprint f(fp,"%f\ t%f\ t%f\ t1\ n", xnew [pivot1] [0] , xnew [pivot1] [3] , xnew [pivot1] [4]);
322.                fprint f(fp,"%f\ t%f\ t%f\ t2\ n", xnew [pivot2] [0] , xnew [pivot2] [3] , xnew [pivot2] [4]);
323.                fprint f(fp,"\ n");
324.                fprint f(fp,"%f\ t%f\ t%f\ t1\ n", xnew [pivot3] [0] , xnew [pivot3] [3] , xnew [pivot3] [4]);
325.                fprint f(fp,"%f\ t%f\ t%f\ t2\ n", xnew [pivot4] [0] , xnew [pivot4] [3] , xnew [pivot4] [4]);
326.                fprint f(fp,"\ n");
327.                fprint f(fp,"\ n");
328.            }
```

```
329.        }
330.        fclose(fp);
331.        printf("setterminalX11\n");
332.        printf("splot'point s.txt'withlinespoint spt7ps2\n");
333.        printf("save'rhombicicosahedron.gnu'\n");
334.  }
335.  void findsymmetries()
336.  {
337.        int p, q, i, j, k, nok, pivot1, pivot2, pivot3, pivot4;
338.        p = 0;
339.        q = 0;
340.        for(i = 0; i <= faces - 1; i++){
341.              p = i * 4;
342.              pivot1 = surfacefin [p] ;
343.              pivot2 = surfacefin [p + 1] ;
344.              pivot3 = surfacefin [p + 2] ;
345.              pivot4 = surfacefin [p + 3] ;
346.              center [q] [0] = (xnew [pivot1] [0] + xnew [pivot4] [0])/2.0;
347.              center [q] [1] = (xnew [pivot1] [3] + xnew [pivot4] [3])/2.0;
348.              center [q] [2] = (xnew [pivot1] [4] + xnew [pivot4] [4])/2.0;
349.              q++;
350.        }
351.        q = 0;
352.        for(i = 0; i <= faces - 1; i++){
353.              for(j = i; j <= faces - 1; j++){
354.                    nok = 0;
355.                    for(k = 0; k <= 3 - 1; k++){
356.                          if(sqrt(Sqr(center [i] [k] - (-1.0 * center [j] [k]))) < eps){
357.                                nok++;
358.                          }
359.                    }
360.                    if(nok == 3){
361.                          symm [q] = i;
362.                          symm [q + 1] = j;
363.                          q = q + 2;
364.                    }
365.              }
366.        }
367.  }
368.  void surfacenormals()
369.  {
370.        int p, q, r, s, i, j, k, pivot1, pivot2, pivot3, pivot4, pivot5, pivot6, cc;
371.        float sum, norm;
372.        p = 0;
373.        r = 0;
374.        s = 0;
375.        for(i = 0; i <= (faces/2) - 1; i++){
376.              p = i * 2;
```

```
377.            r = symm [p] ;
378.            s = symm [p + 1] ;
379.            pivot1 = surfacefin [4 * r] ;
380.            pivot2 = surfacefin [4 * r + 1] ;
381.            pivot3 = surfacefin [4 * r + 2] ;
382.            pivot4 = surfacefin [4 * s] ;
383.            pivot5 = surfacefin [4 * s + 1] ;
384.            pivot6 = surfacefin [4 * s + 2] ;
385.            surfvec [2 * r] [0] = xnew [pivot2] [0] - xnew [pivot1] [0] ;
386.            surfvec [2 * r] [1] = xnew [pivot2] [3] - xnew [pivot1] [3] ;
387.            surfvec [2 * r] [2] = xnew [pivot2] [4] - xnew [pivot1] [4] ;
388.            surfvec [2 * r + 1] [0] = xnew [pivot3] [0] - xnew [pivot1] [0] ;
389.            surfvec [2 * r + 1] [1] = xnew [pivot3] [3] - xnew [pivot1] [3] ;
390.            surfvec [2 * r + 1] [2] = xnew [pivot3] [4] - xnew [pivot1] [4] ;
391.            surfvec [2 * s + 1] [0] = xnew [pivot5] [0] - xnew [pivot4] [0] ;
392.            surfvec [2 * s + 1] [1] = xnew [pivot5] [3] - xnew [pivot4] [3] ;
393.            surfvec [2 * s + 1] [2] = xnew [pivot5] [4] - xnew [pivot4] [4] ;
394.            surfvec [2 * s] [0] = xnew [pivot6] [0] - xnew [pivot4] [0] ;
395.            surfvec [2 * s] [1] = xnew [pivot6] [3] - xnew [pivot4] [3] ;
396.            surfvec [2 * s] [2] = xnew [pivot6] [4] - xnew [pivot4] [4] ;
397.        }
398.        p = 0;
399.        q = 0;
400.        for(i = 0; i <= faces - 1; i + +){
401.            q = i * 2;
402.            cc = 1;
403.            if(i == 5){ cc* = 0; }
404.            if(i == 6){ cc* = 0; }
405.            if(i == 9){ cc* = 0; }
406.            if(i == 10){ cc* = 0; }
407.            if(i == 14){ cc* = 0; }
408.            if(i == 15){ cc* = 0; }
409.            if(i == 18){ cc* = 0; }
410.            if(cc == 1){
411.                surfnormal [p] [0] = ((surfvec [q] [1]) * (surfvec [q + 1] [2])) - ((surfvec [q] [2]) * (surfvec [q + 1] [1]));
412.                surfnormal [p] [1] = -((surfvec [q] [0]) * (surfvec [q + 1] [2])) + ((surfvec [q] [2]) * (surfvec [q + 1] [0]));
413.                surfnormal [p] [2] = ((surfvec [q] [0]) * (surfvec [q + 1] [1])) - ((surfvec [q] [1]) * (surfvec [q + 1] [0]));
414.                p + +;
415.            }
416.            else{
417.                surfnormal [p] [0] = -1.0 * (((surfvec [q] [1]) * (surfvec [q + 1] [2])) - ((surfvec [q] [2]) * (surfvec [q + 1] [1])));
418.                surfnormal [p] [1] = -1.0 * (-((surfvec [q] [0]) * (surfvec [q + 1] [2])) + ((surfvec [q] [2]) * (surfvec [q + 1] [0])));
419.                surfnormal [p] [2] = -1.0 * (((surfvec [q] [0]) * (surfvec [q + 1] [1])) - ((surfvec [q] [1]) * (surfvec [q + 1] [0])));
420.                p + +;
421.            }
422.        }
423.    }
424.    void surfaceconstants()
```

```
425. {
426.     int i, j, k;
427.     float length, temp;
428.     float nterm [faces] [3] = {0.0};
429.     for(i = 0; i <= faces − 1; i + +){
430.         temp = 0.0;
431.         length = (Sqr(surfnormal [i] [0]) + Sqr(surfnormal [i] [1]) + Sqr(surfnormal [i] [2]));
432.         for(j = 0; j <= 3 − 1; j + +){
433.             temp = temp + (center [i] [j] ∗ surfnormal [i] [j])/length;
434.         }
435.         nterm [i] [0] = surfnormal [i] [0] ∗ temp;
436.         nterm [i] [1] = surfnormal [i] [1] ∗ temp;
437.         nterm [i] [2] = surfnormal [i] [2] ∗ temp;
438.     }
439.     for(i = 0; i <= faces − 1; i + +){
440.         surfnormal [i] [0] = nterm [i] [0] ;
441.         surfnormal [i] [1] = nterm [i] [1] ;
442.         surfnormal [i] [2] = nterm [i] [2] ;
443.     }
444. }
445. void constructlattice()
446. {
447.     int p, q, t, i, j, k, l, m, r, s, a, b, c, nok, u;
448.     float ii, jj, kk, ll, mm, sum, temp1, temp2;
449.     FILE ∗ fp;
450.     fp = fopen("point s.txt", "w");
451.     for(i = 0; i <= N − 1; i + +){
452.         for(j = 0; j <= N − 1; j + +){
453.             for(k = 0; k <= N − 1; k + +){
454.                 for(l = 0; l <= N − 1; l + +){
455.                     for(m = 0; m <= N − 1; m + +){
456.                         ii = (float )i;
457.                         jj = (float )j;
458.                         kk = (float )k;
459.                         ll = (float )l;
460.                         mm = (float )m;
461.                         xx [0] = ii ∗ A − ((N − 1)/2) ∗ A;
462.                         xx [1] = jj ∗ A − ((N − 1)/2) ∗ A;
463.                         xx [2] = kk ∗ A − ((N − 1)/2) ∗ A;
464.                         xx [3] = ll ∗ A − ((N − 1)/2) ∗ A;
465.                         xx [4] = mm ∗ A − ((N − 1)/2) ∗ A;
466.                         for(r = 0; r <= d − 1; r + +){
467.                             sum = 0.0;
468.                             for(s = 0; s <= d − 1; s + +){
469.                                 sum = sum + xx [s] ∗ vec [s] [r] ;
470.                             }
471.                             xxnew [r] = sum;
472.                         }
```

```
473.                                nok = 0;
474.                                for(q = 0; q <= faces − 1; q + +){
475.                                    check [q] = 0.0;
476.                                }
477.                                for(p = 0; p <= faces − 1; p + +){
478.                                    temp1 = 0.0;
479.                                    temp2 = Sqr(xxnew [0]) + Sqr(xxnew [3]) + Sqr(xxnew [4]);
480.                                    for(u = 0; u <= 3 − 1; u + +){
481.                                        temp1 = temp1 + Sqr(surfnormal [p] [u]);
482.                                    }
483.                                    check [p] = sqrt(temp1) − sqrt(temp2);
484.                                }
485.                                for(t = 0; t <= faces − 1; t + +){
486.                                    if(check [t] > 0.){
487.                                        nok + +;
488.                                    }
489.                                }
490.                                if(nok == faces){
491.                                    fprint f(fp, "%f\ t%f\ n", xxnew [1] , xxnew [2]);
492.                                }
493.                            }
494.                        }
495.                    }
496.                }
497.            }
498.            fclose(fp);
499.    }
500.    main()
501.    {
502.            int i;
503.            rotvectors();
504.            normalize();
505.            makerotatedunitcube();
506.            findvertices();
507.            findneighbours();
508.            identifysurfaces();
509.            //plotrhombicicosahedron();
510.            findsymmetries();
511.            surfacenormals();
512.            surfaceconstants();
513.            constructlattice();
514.    }
```

A.2. *Icosahedral.c*

```
#include < stdio.h >
#include < stdlib.h >
#include < math.h >
#define d 6
#define N 6
#define n 2 // #of steps to construct the unitcube
#define A 1.0
#define pi 4.0 ∗ atanl(1.0)
#define C (1.0/sqrt(2.0 ∗ (1.0 + (tau ∗ tau))))
#define beta (2.0 ∗ pi)/5.0
#define tau 0.5 ∗ (1.0 + sqrt(5.0))
#define uver 64 // #of vertices of unitcube
#define ver 32 // #of vertices of rhombicicosahedron
#define faces 30 // #of faces of rhombicicosahedron
#define poss 352 // #of possibilities to constructable surfaces
#define eps 0.00001
#define min (a, b)((a) < (b)?(a) : (b))
int neighs [uver ∗ d] , listlen [uver] , surfprev [poss] , surface [poss] , surfacefin [faces ∗ 4] , symm [faces] , tria [ver] , vert1 [3] , vert2 [3] ;
float x [uver] [d] , xnew [uver] [d] , xx [d] , xxnew [d] ;
float lines [uver] , distance [uver] , vec [d] [d] ;
float ranvec [ver] [4] , center [faces] [3] , surfvec [2 ∗ faces] [3] , surfnormal [faces] [3] , reldist [3] ;
float check [faces] , relative [ver ∗ (ver − 1)] , figen [5 ∗ ver] , unit [ver] [3] ;
float Sqr(float x)
{
    float i;
    i = x ∗ x;
    return i;
}
void rotvectors()
{
    vec [0] [0] = C ∗ 1.0;
    vec [0] [1] = C ∗ tau;
    vec [0] [2] = C ∗ 0.0;
    vec [0] [3] = C ∗ (−1.0 ∗ tau);
    vec [0] [4] = C ∗ 1.0;
    vec [0] [5] = C ∗ 0.0;
    vec [1] [0] = C ∗ tau;
    vec [1] [1] = C ∗ 0.0;
    vec [1] [2] = C ∗ 1.0;
    vec [1] [3] = C ∗ 1.0;
    vec [1] [4] = C ∗ 0.0;
    vec [1] [5] = C ∗ (−1.0 ∗ tau);
    vec [2] [0] = C ∗ 0.0;
    vec [2] [1] = C ∗ 1.0;
    vec [2] [2] = C ∗ tau;
```

```
46.     vec [2] [3] = C * 0.0;
47.     vec [2] [4] = C * (-1.0 * tau);
48.     vec [2] [5] = C * 1.0;
49.     vec [3] [0] = C * (-1.0);
50.     vec [3] [1] = C * tau;
51.     vec [3] [2] = C * 0.0;
52.     vec [3] [3] = C * tau;
53.     vec [3] [4] = C * 1.0;
54.     vec [3] [5] = C * 0.0;
55.     vec [4] [0] = C * tau;
56.     vec [4] [1] = C * 0.0;
57.     vec [4] [2] = C * (-1.0);
58.     vec [4] [3] = C * 1.0;
59.     vec [4] [4] = C * 0.0;
60.     vec [4] [5] = C * tau;
61.     vec [5] [0] = C * 0.0;
62.     vec [5] [1] = C * (-1.0);
63.     vec [5] [2] = C * tau;
64.     vec [5] [3] = C * 0.0;
65.     vec [5] [4] = C * tau;
66.     vec [5] [5] = C * 1.0;
67. }
68. void normalize()
69. {
70.     int i, j, k;
71.     float norm, sum;
72.     for(i = 0; i <= d - 1; i++){
73.         sum = 0.0;
74.         for(j = 0; j <= d - 1; j++){
75.             sum = sum + Sqr(vec [i] [j]);
76.         }
77.         norm = sqrt(sum);
78.         for(k = 0; k <= d - 1; k++){
79.             vec [i] [k] = vec [i] [k] /norm;
80.         }
81.     }
82. }
83. void makerotatedunitcube()
84. {
85.     int i, j, k, l, m, r, s, p, q;
86.     float ii, jj, kk, ll, mm, qq, sum;
87.     p = 0;
88.     for(i = 0; i <= n - 1; i++){
89.         for(j = 0; j <= n - 1; j++){
90.             for(k = 0; k <= n - 1; k++){
91.                 for(l = 0; l <= n - 1; l++){
92.                     for(m = 0; m <= n - 1; m++){
93.                         for(q = 0; q <= n - 1; q++){
```

```
                                    ii = (float )i;
                                    jj = (float )j;
                                    kk = (float )k;
                                    ll = (float )l;
                                    mm = (float )m;
                                    qq = (float )q;
                                    x [p] [0] = (ii − 0.5) ∗ A;
                                    x [p] [1] = (jj − 0.5) ∗ A;
                                    x [p] [2] = (kk − 0.5) ∗ A;
                                    x [p] [3] = (ll − 0.5) ∗ A;
                                    x [p] [4] = (mm − 0.5) ∗ A;
                                    x [p] [5] = (qq − 0.5) ∗ A;
                                    for(r = 0; r ≤ d − 1; r + +){
                                    sum = 0.0;
                                        for(s = 0; s ≤ d − 1; s + +){
                                            sum = sum + x [p] [s] ∗ vec [s] [r] ;
                                        }
                                        xnew [p] [r] = sum;
                                    }
                                    p + +;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    void findvertices()
    {
        int i, j;
        float ordistcal, ordist, rr;
        for(i = 0; i ≤ uver − 1; i + +){
            ordistcal = Sqr(xnew [i] [0]) + Sqr(xnew [i] [1]) + Sqr(xnew [i] [2]);
            ordist = sqrt(ordistcal);
            distance [i] = ordist;
            lines [i] = distance [i] ;
        }
        for(i = 0; i ≤ uver − 1; i + +){
            for(j = 0; j ≤ uver − 2; j + +){
                if(distance [j + 1] > distance [j]){
                    rr = distance [j] ;
                    distance [j] = distance [j + 1] ;
                    distance [j + 1] = rr;
                }
                else{
                    distance [j] = distance [j] ;
                    distance [j + 1] = distance [j + 1] ;
                }
```

```
142.            }
143.        }
144.    }
145.    void findneighbours()
146.    {
147.        int p, q, r, t, i, j, k, l, m, s;
148.        float distcal, dist;
149.        float a, b, c;
150.        for(i = 0; i ≤ uver − 1; i + +){
151.            p = 0;
152.            q = i ∗ d;
153.            for(j = 0; j ≤ uver − 1; j + +){
154.                distcal = 0.0;
155.                for(k = 0; k ≤ d − 1; k + +){
156.                    distcal = distcal + Sqr(x [i] [k] − x [j] [k]);
157.                }
158.                dist = sqrt(distcal);
159.                if(i! = j){
160.                    if((dist − 1.0) < eps){
161.                        if(distance [ver − 1] ≤ lines [i] &&distance [ver − 1] ≤ lines [j]){
162.                            neighs [q + p] = j;
163.                            p + +;
164.                        }
165.                    }
166.                }
167.            }
168.            listlen [i] = p;
169.            if(p! = d){
170.                for(s = p + q; s ≤ q + d − 1; s + +){
171.                    neighs [s] = uver;
172.                }
173.            }
174.        }
175.    }
176.    void identifysurfaces()
177.    {
178.        int p, q, r, s, t, i, j, k, l, s1, s2, p1, p2, t1, t2, pivot, pivot1, pivot2;
179.        p = 0;
180.        q = 0;
181.        s = 0;
182.        r = 0;
183.        for(i = 0; i ≤ uver − 1; i + +){
184.            s = listlen [i] ;
185.            if(s! = 0){
186.                for(j = 0; j ≤ s − 1; j + +){
187.                    q = i ∗ d;
188.                    p = q + j;
189.                    pivot = neighs [q] ;
```

```
190.                    if(j < s - 1){
191.                        pivot1 = neighs[p];
192.                        pivot2 = neighs[p + 1];
193.                        s1 = listlen[pivot1];
194.                        s2 = listlen[pivot2];
195.                        for(k = 0; k <= s1 - 1; k++){
196.                            p1 = pivot1 * d + k;
197.                            for(l = 0; l <= s2 - 1; l++){
198.                                p2 = pivot2 * d + l;
199.                                t1 = neighs[p1];
200.                                t2 = neighs[p2];
201.                                if(t1 == t2){
202.                                    if(t1! = i){
203.                                        surfprev[r] = i;
204.                                        surfprev[r + 1] = pivot1;
205.                                        surfprev[r + 2] = pivot2;
206.                                        surfprev[r + 3] = t1;
207.                                        r = r + 4;
208.                                    }
209.                                }
210.                            }
211.                        }
212.                    }
213.                    else{
214.                        pivot1 = neighs[p];
215.                        pivot2 = pivot;
216.                        s1 = listlen[pivot1];
217.                        s2 = listlen[pivot2];
218.                        for(k = 0; k <= s1 - 1; k++){
219.                            p1 = pivot1 * d + k;
220.                            for(l = 0; l <= s2 - 1; l++){
221.                                p2 = pivot2 * d + l;
222.                                t1 = neighs[p1];
223.                                t2 = neighs[p2];
224.                                if(t1 == t2){
225.                                    if(t1! = i){
226.                                        surfprev[r] = i;
227.                                        surfprev[r + 1] = pivot1;
228.                                        surfprev[r + 2] = pivot2;
229.                                        surfprev[r + 3] = t1;
230.                                        r = r + 4;
231.                                    }
232.                                }
233.                            }
234.                        }
235.                    }
236.                }
237.            }
```

```
238.    }
239.    p = 0;
240.    for(i = 0; i ≤ (poss/4) − 1; i + +){
241.        p = i ∗ 4;
242.        for(j = 0; j ≤ 4 − 1; j + +){
243.            surface [p + j] = surfprev [p + j] ;
244.        }
245.    }
246.    p = 0;
247.    q = 0;
248.    for(i = 0; i ≤ (poss/4) − 1; i + +){
249.        p = i ∗ 4;
250.        for(k = 0; k ≤ d; k + +){
251.            for(j = 0; j ≤ 3 − 1; j + +){
252.                q = p + j;
253.                pivot1 = surfprev [q] ;
254.                pivot2 = surfprev [q + 1] ;
255.                if(pivot1 > pivot2){
256.                    t = surfprev [q] ;
257.                    surfprev [q] = surfprev [q + 1] ;
258.                    surfprev [q + 1] = t;
259.                }
260.            }
261.        }
262.    }
263.    p = 0;
264.    q = 0;
265.    for(i = 0; i ≤ (poss/4) − 2; i + +){
266.        p = i ∗ 4;
267.        for(j = i + 1; j ≤ (r/4) − 1; j + +){
268.            q = j ∗ 4;
269.            if(surfprev [p] == surfprev [q]){
270.                if(surfprev [p + 1] == surfprev [q + 1]){
271.                    if(surfprev [p + 2] == surfprev [q + 2]){
272.                        if(surfprev [p + 3] == surfprev [q + 3]){
273.                            for(k = 0; k ≤ 4 − 1; k + +){
274.                                surfprev [q + k] = uver;
275.                            }
276.                        }
277.                    }
278.                }
279.            }
280.        }
281.    }
282.    p = 0;
283.    q = 0;
284.    for(i = 0; i ≤ (poss/4) − 1; i + +){
285.        p = i ∗ 4;
```

```
286.            if(surfprev [p]! = uver){
287.                surfacefin [q] = surface [p] ;
288.                surfacefin [q + 1] = surface [p + 1] ;
289.                surfacefin [q + 2] = surface [p + 2] ;
290.                surfacefin [q + 3] = surface [p + 3] ;
291.                q = q + 4;
292.            }
293.        }
294.  }
295.  void plotrhombictriacontahedron()
296.  {
297.        int i, p, q, pivot1, pivot2, pivot3, pivot4;
298.        FILE * fp;
299.        fp = fopen("point s.txt","w");
300.        p = 0;
301.        q = 0;
302.        for(i = 0; i <= (poss/4) - 1; i + +){
303.            p = i * 4;
304.            if(surfprev [p]! = uver){
305.                pivot1 = surface [p] ;
306.                pivot2 = surface [p + 1] ;
307.                pivot3 = surface [p + 2] ;
308.                pivot4 = surface [p + 3] ;
309.                fprint f(fp,"%f\ t%f\ t%f\ t1\ n", xnew [pivot1] [0] , xnew [pivot1] [1] , xnew [pivot1] [2]);
310.                fprint f(fp,"%f\ t%f\ t%f\ t2\ n", xnew [pivot2] [0] , xnew [pivot2] [1] , xnew [pivot2] [2]);
311.                fprint f(fp,"\ n");
312.                fprint f(fp,"%f\ t%f\ t%f\ t1\ n", xnew [pivot3] [0] , xnew [pivot3] [1] , xnew [pivot3] [2]);
313.                fprint f(fp,"%f\ t%f\ t%f\ t2\ n", xnew [pivot4] [0] , xnew [pivot4] [1] , xnew [pivot4] [2]);
314.                fprint f(fp,"\ n");
315.                fprint f(fp,"\ n");
316.            }
317.        }
318.        fclose(fp);
319.        print f("setterminalx11\ n");
320.        print f("settitle\ "Rhombictriacontahedron\ "\ n");
321.        print f("splot′point s.txt′withlinespoint spt7ps2\ n");
322.        print f("save′rhombictriacontahedron.gnu′\ n");
323.  }
324.  void findsymmetries()
325.  {
326.        int p, q, i, j, k, nok, pivot1, pivot2, pivot3, pivot4;
327.        p = 0;
328.        q = 0;
329.        for(i = 0; i <= faces - 1; i + +){
330.        p = i * 4;
331.            pivot1 = surfacefin [p] ;
332.            pivot2 = surfacefin [p + 1] ;
333.            pivot3 = surfacefin [p + 2] ;
```

```
334.            pivot4 = surfacefin [p + 3] ;
335.            center [q] [0] = (xnew [pivot1] [0] + xnew [pivot4] [0])/2.0;
336.            center [q] [1] = (xnew [pivot1] [1] + xnew [pivot4] [1])/2.0;
337.            center [q] [2] = (xnew [pivot1] [2] + xnew [pivot4] [2])/2.0;
338.            q + +;
339.        }
340.        q = 0;
341.        for(i = 0; i ≤ faces − 1; i + +){
342.            for(j = i; j ≤ faces − 1; j + +){
343.                nok = 0;
344.                for(k = 0; k ≤ 3 − 1; k + +){
345.                    if(sqrt(Sqr(center [i] [k] − (−1.0 ∗ center [j] [k]))) < eps){
346.                        nok + +;
347.                    }
348.                }
349.                if(nok == 3){
350.                    symm [q] = i;
351.                    symm [q + 1] = j;
352.                    q = q + 2;
353.                }
354.            }
355.        }
356. }
357. void surfacenormals()
358. {
359.        int p, q, r, s, i, j, k, pivot1, pivot2, pivot3, pivot4, pivot5, pivot6, cc;
360.        float sum, norm;
361.        p = 0;
362.        r = 0;
363.        s = 0;
364.        for(i = 0; i ≤ (faces/2) − 1; i + +){
365.            p = i ∗ 2;
366.            r = symm [p] ;
367.            s = symm [p + 1] ;
368.            pivot1 = surfacefin [4 ∗ r] ;
369.            pivot2 = surfacefin [4 ∗ r + 1] ;
370.            pivot3 = surfacefin [4 ∗ r + 2] ;
371.            pivot4 = surfacefin [4 ∗ s] ;
372.            pivot5 = surfacefin [4 ∗ s + 1] ;
373.            pivot6 = surfacefin [4 ∗ s + 2] ;
374.            surfvec [2 ∗ r] [0] = xnew [pivot2] [0] − xnew [pivot1] [0] ;
375.            surfvec [2 ∗ r] [1] = xnew [pivot2] [1] − xnew [pivot1] [1] ;
376.            surfvec [2 ∗ r] [2] = xnew [pivot2] [2] − xnew [pivot1] [2] ;
377.            surfvec [2 ∗ r + 1] [0] = xnew [pivot3] [0] − xnew [pivot1] [0] ;
378.            surfvec [2 ∗ r + 1] [1] = xnew [pivot3] [1] − xnew [pivot1] [1] ;
379.            surfvec [2 ∗ r + 1] [2] = xnew [pivot3] [2] − xnew [pivot1] [2] ;
380.            surfvec [2 ∗ s + 1] [0] = xnew [pivot5] [0] − xnew [pivot4] [0] ;
381.            surfvec [2 ∗ s + 1] [1] = xnew [pivot5] [1] − xnew [pivot4] [1] ;
```

```
382.            surfvec [2 * s + 1] [2] = xnew [pivot5] [2] − xnew [pivot4] [2] ;
383.            surfvec [2 * s] [0] = xnew [pivot6] [0] − xnew [pivot4] [0] ;
384.            surfvec [2 * s] [1] = xnew [pivot6] [1] − xnew [pivot4] [1] ;
385.            surfvec [2 * s] [2] = xnew [pivot6] [2] − xnew [pivot4] [2] ;
386.        }
387.        p = 0;
388.        q = 0;
389.        for(i = 0; i ≤ faces − 1; i + +){
390.            q = i * 2;
391.            surfnormal [p] [0] = ((surfvec [q] [1]) * (surfvec [q + 1] [2])) − ((surfvec [q] [2]) * (surfvec [q + 1] [1]));
392.            surfnormal [p] [1] = −((surfvec [q] [0]) * (surfvec [q + 1] [2])) + ((surfvec [q] [2]) * (surfvec [q + 1] [0]));
393.            surfnormal [p] [2] = ((surfvec [q] [0]) * (surfvec [q + 1] [1])) − ((surfvec [q] [1]) * (surfvec [q + 1] [0]));
394.            p + +;
395.        }
396.        for(i = 0; i ≤ faces − 1; i + +){
397.            sum = 0.0;
398.            for(j = 0; j ≤ 3 − 1; j + +){
399.                sum = sum + ((surfnormal [i] [j]) * center [i] [j]);
400.            }
401.            if(sum >= 0.0){
402.                for(j = 0; j ≤ 3 − 1; j + +){
403.                    surfnormal [i] [j] = surfnormal [i] [j] ;
404.                }
405.            }
406.            else{
407.                for(j = 0; j ≤ 3 − 1; j + +){
408.                    surfnormal [i] [j] = (−1.0) * surfnormal [i] [j] ;
409.                }
410.            }
411.        }
412.  }
413.  void surfaceconstants()
414.  {
415.        int i, j, k;
416.        float length, temp;
417.        float nterm [faces] [3] = {0.0};
418.        for(i = 0; i ≤ faces − 1; i + +){
419.            temp = 0.0;
420.            length = (Sqr(surfnormal [i] [0]) + Sqr(surfnormal [i] [1]) + Sqr(surfnormal [i] [2]));
421.            for(j = 0; j ≤ 3 − 1; j + +){
422.                temp = temp + (center [i] [j] * surfnormal [i] [j])/length;
423.            }
424.            nterm [i] [0] = surfnormal [i] [0] * temp;
425.            nterm [i] [1] = surfnormal [i] [1] * temp;
426.            nterm [i] [2] = surfnormal [i] [2] * temp;
427.        }
428.        for(i = 0; i ≤ faces − 1; i + +){
429.            surfnormal [i] [0] = nterm [i] [0] ;
```

```
430.            surfnormal [i] [1] = nterm [i] [1] ;
431.            surfnormal [i] [2] = nterm [i] [2] ;
432.        }
433.  }
434.  void constructlattice()
435.  {
436.        int p, q, t, i, j, k, l, m, r, s, a, b, c, nok, u, o;
437.        float ii, jj, kk, ll, mm, qq, sum, temp1, temp2;
438.        FILE * fp;
439.        fp = fopen("point s.txt", "w");
440.        for(i = 0; i <= N - 1; i + +){
441.            for(j = 0; j <= N - 1; j + +){
442.                for(k = 0; k <= N - 1; k + +){
443.                    for(l = 0; l <= N - 1; l + +){
444.                        for(m = 0; m <= N - 1; m + +){
445.                            for(q = 0; q <= N - 1; q + +){
446.                                ii = (float )i;
447.                                jj = (float )j;
448.                                kk = (float )k;
449.                                ll = (float )l;
450.                                mm = (float )m;
451.                                qq = (float )q;
452.                                xx [0] = ii * A - ((N - 1)/2) * A;
453.                                xx [1] = jj * A - ((N - 1)/2) * A;
454.                                xx [2] = kk * A - ((N - 1)/2) * A;
455.                                xx [3] = ll * A - ((N - 1)/2) * A;
456.                                xx [4] = mm * A - ((N - 1)/2) * A;
457.                                xx [5] = qq * A - ((N - 1)/2) * A;
458.                                for(r = 0; r <= d - 1; r + +){
459.                                    sum = 0.0;
460.                                    for(s = 0; s <= d - 1; s + +){
461.                                        sum = sum + xx [s] * vec [s] [r] ;
462.                                    }
463.                                    xxnew [r] = sum;
464.                                }
465.                                nok = 0;
466.                                for(o = 0; o <= faces - 1; o + +){
467.                                    check [o] = 0.0;
468.                                }
469.                                for(p = 0; p <= faces - 1; p + +){
470.                                    temp1 = 0.0;
471.                                    temp2 = Sqr(xxnew [0]) + Sqr(xxnew [1]) + Sqr(xxnew [2]);
472.                                    for(u = 0; u <= 3 - 1; u + +){
473.                                        temp1 = temp1 + Sqr(surfnormal [p] [u]);
474.                                    }
475.                                    check [p] = sqrt(temp1) - sqrt(temp2);
476.                                }
477.                                for(t = 0; t <= faces - 1; t + +){
```

```
478.                                        if(check [t] >= 0.){
479.                                            nok + +;
480.                                        }
481.                                    }
482.                                    if(nok == faces){
483.                                        fprint f(fp, "%f\ t%f\ t%f\ n", xxnew [3] , xxnew [4] , xxnew [5]);
484.                                    }
485.                                }
486.                            }
487.                        }
488.                    }
489.                }
490.        }
491.        fclose(fp);
492.  }
493.  main()
494.  {
495.        int i;
496.        rotvectors();
497.        normalize();
498.        makerotatedunitcube();
499.        findvertices();
500.        findtriacontahedron();
501.        findneighbours();
502.        identifysurfaces();
503.        //plotrhombictriacontahedron();
504.        findsymmetries();
505.        surfacenormals();
506.        surfaceconstants();
507.        constructlattice();
508.  }
```

A.3. *Rotator.c*

```
#include < stdio.h >
#include < stdlib.h >
#include < math.h >
#include < string.h >
#define N203751
#define delta0.0001
float x [N] , y [N] , z [N] , dot1, dot2;
main()
{
    int i, j, k, p = 1, d [N] = {0};
    float gx, gy, gz, n [3] , n1 [3] , n2 [3] , rp [3] , dist1, dist2;
    FILE ∗ fp;
    FILE ∗ out;
    FILE ∗ out2;
    fp = fopen("point s30.txt", "r");
    for(i = 0; i <= N − 1; i + +){
        fscanf(fp, "%f\ t%f\ t%f\ n", &x [i] , &y [i] , &z [i]);
    }
    fclose(fp);
    n [0] = 1.;
    n [1] = 0.;
    n [2] = (1. + sqrt(5.))/2.;
    gx = sqrt(n [0] ∗ n [0] + n [1] ∗ n [1] + n [2] ∗ n [2]);
    n [0] = n [0] /gx;
    n [1] = n [1] /gx;
    n [2] = n [2] /gx;
    print f("%f%f%f\ n", n [0] , n [1] , n [2]);
    out = fopen("out130.txt", "w");
    out2 = fopen("out230.txt", "w");
    for(i = 0; i < N − 1; i + +){
        if(d [i] == 0){
            d [i] = p;
            dist1 = x [i] ∗ n [0] + y [i] ∗ n [1] + z [i] ∗ n [2] ;
            k = 0;
            rp [0] = x [i] − dist1 ∗ n [0] ;
            rp [1] = y [i] − dist1 ∗ n [1] ;
            rp [2] = z [i] − dist1 ∗ n [2] ;
            if(i == 0)
            {
                n1 [0] = rp [0] ;
                n1 [1] = rp [1] ;
                n1 [2] = rp [2] ;
                gx = sqrt(n1 [0] ∗ n1 [0] + n1 [1] ∗ n1 [1] + n1 [2] ∗ n1 [2]);
                n1 [0] = n1 [0] /gx;
                n1 [1] = n1 [1] /gx;
```

```
46.                     n1 [2] = n1 [2] /gx;
47.                     n2 [0] = n [1] ∗ n1 [2] − n [2] ∗ n1 [1] ;
48.                     n2 [1] = n [2] ∗ n1 [0] − n [0] ∗ n1 [2] ;
49.                     n2 [2] = n [0] ∗ n1 [1] − n [1] ∗ n1 [0] ;
50.                     gx = sqrt(n2 [0] ∗ n2 [0] + n2 [1] ∗ n2 [1] + n2 [2] ∗ n2 [2]);
51.                     n2 [0] = n2 [0] /gx;
52.                     n2 [1] = n2 [1] /gx;
53.                     n2 [2] = n2 [2] /gx;
54.                 }
55.                 dot1 = n1 [0] ∗ rp [0] + n1 [1] ∗ rp [1] + n1 [2] ∗ rp [2] ;
56.                 dot2 = n2 [0] ∗ rp [0] + n2 [1] ∗ rp [1] + n2 [2] ∗ rp [2] ;
57.                 fprint f(out, "%f%f%f%d\ n", dot1, dot2, dist1, p);
58.                 for(j = i + 1; j < N; j + +){
59.                     if(d [j] == 0){
60.                         dist2 = x [j] ∗ n [0] + y [j] ∗ n [1] + z [j] ∗ n [2] ;
61.                         if(fabs(dist1 − dist2) <= delta){
62.                             d [j] = p;
63.                             k + +;
64.                             rp [0] = x [j] − dist2 ∗ n [0] ;
65.                             rp [1] = y [j] − dist2 ∗ n [1] ;
66.                             rp [2] = z [j] − dist2 ∗ n [2] ;
67.                             dot1 = n1 [0] ∗ rp [0] + n1 [1] ∗ rp [1] + n1 [2] ∗ rp [2] ;
68.                             dot2 = n2 [0] ∗ rp [0] + n2 [1] ∗ rp [1] + n2 [2] ∗ rp [2] ;
69.                             fprint f(out, "%f%f%f%d\ n", dot1, dot2, dist2, p);
70.                             if(p == 92)
71.                             {
72.                                 fprint f(out2, "%f%f\ n", dot1, dot2);
73.                             }
74.                         }
75.                     }
76.                 }
77.                 p + +;
78.                 print f("%f\ t%d\ t%d\ n", dist1, p − 1, k);
79.             }
80.         }
81.         fclose(out);
82.         print f("%dplanes\ n", p − 1);
83.         fp = fopen("point s30rot.txt", "w");
84.             for(i = 0; i < N; i + +){
85.                 fprint f(fp, "%f\ t%f\ t%f\ t%d\ n", x [i] , y [i] , z [i] , d [i]);
86.             }
87.             fclose(fp);
88.         }
89. }
```

APPENDIX B: ANALYSIS OF THE CODES

B.1. Penrose.c

The 2-dimensional quasicrystal can be obtained using the Penrose.c code. Functions and variables are in the code can be explained as follow:

- Some of the header files that the code needs are included in the first three lines.
- Lines 4-17 are declarations of the user defined constants are given to the code as inputs.
- Lines 18-22 are declarations of the user defined variables with their types and sizes that should be allocated in memory.
- Function *Sqr* (lines 23-28) takes the square of an floating point number if it is needed.
- Function *rotvectors* (lines 35-62) corresponds to the projection matrix as column vectors. The vectors are embedded into the code as a global input.
- Function *normalize* (lines 63-77) normalizes the vectors if that is needed.
- Function *makerotatedunitcube* (78-111) costructs the unit hypercube, and tarnsforms it under the operation of vectors that are defined in the function *rotvectors*.
- Function *findvertices* (lines 112-135) finds the vertices of the *rhombic icosahedron* using the distances relative to the origin.
- Function *findneighbours* (lines 136-187) takes the coordinates of vertices that are obtained from the function *findvertices*, as the input. Utilizing these coordinates it calculates the coordinations of the vertices of the *rhombic icosahedron*.
- Function *identifysurfaces* (lines 188-306) takes the neighbor list from the function *findvertices* and identify the surfaces of the *rhombic icosahedron*.
- Function *plotrhombicicosahedron* (lines 307-328) gives an output as a text file that can be utilized to draw the *rhombic icosahedron* with the help of any plotting software.
- Function *findsymmetries* (lines 335-367) identifies the symmetric faces of the *rhombic icosahedron*. It calculates the coordinates of the centers of those faces

also.

- Function *surfacenormals* (lines 368-423) calculates the surface normals.
- Function *surfaceconstants* (lines 424-444) calculates the positions of faces with respect to the origin.
- Function *constructlattice* (lines 445-499)generates the hypercubic lattice. It transforms the lattice and check whether a generated point inside of the *rhombic icosahedron* or not. If it is inside, it records the transformed coordinate of this point to a text file.
- In the function *main* (lines 500-514), other functions are organized and optional outputs (e.g plotting of the *rhombic icosahedron*) can be obtained.

B.2. Icosahedral.c

The code icosahedral.c can be utilized in order to obtain the 3-d Quasicrystal. Functions and variables are in the code can be explained as follow:

- Some of the header files that the code needs are included in the first three lines.
- Lines 4-18 are declarations of the user defined constants are given to the code as inputs.
- Lines 19-23 are declarations of the user defined variables with their types and sizes that should be allocated in memory.
- Function *Sqr* (lines 23-28) takes the square of an floating point number if it is needed.
- Function *rotvectors* (lines 29-67) corresponds to the projection matrix as column vectors. The vectors are embedded into the code as a global input.
- Function *normalize* (lines 68-82) normalizes the vectors if that is needed.
- Function *makerotatedunitcube* (83-120) costructs the unit hypercube, and tarnsforms it under the operation of vectors that are defined in the function *rotvectors*.
- Function *findvertices* (lines 121-144) finds the vertices of the *rhombic triacontahedron* using the distances relative to the origin.
- Function *findneighbours* (lines 145-175) takes the coordinates of vertices that are obtained from the function *findvertices*, as the input. Utilizing these coordinates

it calculates the coordinations of the vertices of the *rhombic triacontahedron*.

- Function *identifysurfaces* (lines 176-294) takes the neighbor list from the function *findvertices* and identify the surfaces of the *rhombic triacontahedron*.
- Function *plotrhombictriacontahedron* (lines 295-323) gives an output as a text file that can be utilized to draw the *rhombic icosahedron* with the help of any plotting software.
- Function *findsymmetries* (lines 324-356) identifies the symmetric faces of the *rhombic triacontahedron*. It calculates the coordinates of the centers of those faces also.
- Function *surfacenormals* (lines 357-412) calculates the surface normals.
- Function *surfaceconstants* (lines 413-433) calculates the positions of faces with respect to the origin.
- Function *constructlattice* (lines 434-492) generates the hypercubic lattice. It transforms the lattice and check whether a generated point inside of the *rhombic icosahedron* or not. If it is inside, it records the transformed coordinate of this point to a text file.
- In the function *main* (lines 500-514), other functions are organized and optional outputs (e.g plotting of the *rhombic icosahedron*) can be obtained.

B.3. Rotator.c

The code Rotator.c starts with including some of the necessary header files of *C* (lines 1-4). It reads the generated coordinates as an input from a chosen text file at the beginning of the function *main* (lines 10-18). It takes a vector as an input from the user and normalizes it (lines 20-26). With the help of this vector and the lattice coordinates it generates two perpendicular vectors to the formerly given vector (lines 27-54). As a next step, it calculates the rotated coordinates with respect to these three vectors (lines 55-81). Finally rotated coordinates are written to a given text file.

REFERENCES

1. Levine, D. and P. J. Steinhardt, "Quasicrystals: A new class of ordered structures", *Physical Review Letters*, Vol.53, No.26, pp.2477-2480, 1984.

2. Stephens, P. W. and A. J. Goldman, "Sharp diffraction maxima from an icosahedral glass", *Physical Review Letters*, Vol.56, No.11, pp.1168-1171, 1986.

3. Grünbaum, B. and G. C. Shephard, *Tilings and Patterns*, W. H. Freeman & Go, New York, 1987.

4. Senechal, M., *Quasicrystals and Geometry*, Cambridge University Press, 1995.

5. Ashcroft, N. W. and N. D. Mermin, *Solid State Physics*, Brooks Cole, New York, 1976.

6. Hippert, F. and D. Gratias, *Lectures on Quasicrystals*, Editions de Physique, 1994.

7. Gähler, F. and J. Rhyner, "Equivalance of the generalised grid and projection methods for the construction of quasiperiodic tilings", *Journal of Physics A: Mathematical and general*, Vol.19, pp.267-277, 1986.

8. Levine, D. and P. J. Steinhardt, "Quasicrystals. I: Definition and structure", *Physical Review B*, Vol.34, No.2, pp.596-616, 1986.

9. Pavlovitch, A. and M. Kleman, "Generalised 2D Penrose tilings: Structural properties", *Journal of Physics A: Mathematical and general*, Vol.20, pp.687-702, 1987.

10. Repetowitcz, P., U. Grimm and M. Schreiber, "High-temprature expansion for Ising Models on quasiperiodic tilings", *Journal of Physics A: Mathematical and general*, Vol.32, pp.4397-4418, 1999.

11. Gratias, D., M. Quinquandon and A. Katz, *Introduction to Icasahedral Quasicrys-*

tals, World Scientific, 2002.

12. Vogg, U. and P. L. Ryder, "A general algorithm for generating quasiperiodic lattices by the strip projection method", *Journal of Non-Crystalline Solids*, Vol.194, pp.135-144, 1996.

13. Conway, J. H. and K. M. Knowles, "Quasiperiodic tiling in two and three dimension", *Journal of Physics A: Mathematical and general*, Vol.19, pp.3645-3653, 1986.

14. Papadopolos, Z. and G. Kasner, "Thick atomic layers of maximum density as bulk terminations of quasicrystals", *Physical Review B*, Vol.72, pp.4397-4418, 2005.

15. Mermin, N. D, "The space groups of icosahedral quasicrystals and cubic, orthorombic, monoclinic, and triclinic crystals", *Reviews of Modern Physics*, Vol.64, No.1, 1992.

16. Crystal Maker Software Ltd., *http://www.crystalmaker.com*

17. Henley, C. L., "Cell geometry for Cluster-Based quasicrystal models", *Physical Review B*, Vol.43 , pp.993-1020, 1991.

18. Kats, E. I. and A. R. Muratov, "Three-wave vibrational mode broadening for Fibonacci One-dimensional quasicrystals", *Journal of Physics: Condensed Matter*, Vol.17, No.43, pp.6849-6869, 2005.

19. Dimitrienko, V. E., M. Erbudak and M. Kleman, "Growth, melting, and surface structure of icosahedral quasicrystals: Monte Carlo simulations", *Aperiodic Structures*, p.41, 2001.