German Research School
for Simulation Sciences

Classifying Skewed Lattices for Quantum Cluster Calculations

Master's Thesis

Cica Gustiani

27 May 2015

Supervisor
Prof. Dr. Erik Koch

Examiner
Prof. Dr. Erik Koch

Co-Examiner
Prof. Dr. Eva Pavarini

JÜLICH
FORSCHUNGSZENTRUM

RWTHAACHEN
UNIVERSITY

Abstract

Practical calculations for infinite lattices are limited to finite systems, usually supercells with periodic boundary conditions. We intend to study such supercells by systematically enumerating all inequivalent choices of a given size using the Hermite normal form. Using the symmetries of the underlying lattice, we eliminate supercells that are equivalent by symmetry. With the help of integer matrix methods like the Lenstra-Lenstra-Lováz (LLL) algorithm, we reduce given basis to its most compact form and analyze its properties using the criteria given by Betts and collaborators. We finally turn to the properties in k-space and investigate the tight-binding states on the clusters using periodic, antiperiodic, and open boundary conditions.

Contents

Chapter 1

Introduction

In order to understand the properties of a material, simulating an infinite lattice system has been a challenge in the electronic structure theory. One of the approach is by truncating an infinite lattice into a finite size of cluster. However, this introduces a large fraction of surface atoms. For example, in a $10 \times 10 \times 10$ cluster, 488 of the 1000 atoms are on the surface. One may remove this surface effects by imposing periodicity. Hence, the electrons that passing to the border will enter back in the opposite side. By construction, we have an infinite number of electrons. However, since the number of electrons inside the cell is limited to N, only N electrons have independent degrees of freedom. Then imposing periodicity basically will suppress the fluctuation. We can not avoid periodicity in order to have infinite lattice, but we may choose good supercells that are useful for extrapolating the properties of the infinite lattice.

There are infinitely many choices of supercells. One might try to find the supercells that describe a property of an infinite lattice by trying out many supercells, but this is very ineffective. However, we might try to find some properties of a supercell that may owned by the infinite lattice.

Let us approach this problem from the 2-dimensional lattices, *e.g.* the calculation of quantum spin systems that has been frequently discussed in literature as a topic of highly correlated electron system. In these calculations, all supercells were based on square shapes until Haan *et al.* [1] showed that non-squares supercells can be also good to use. Latter, Betts *et al.* [2, 3] initiated some works classifying the goodness of supercells by introducing some criteria that may grade them. They introduced some geometrical properties and topological properties of a supercell that may quantify the divergence of a supercell to the infinite lattice. Here we elaborate the criteria of good supercells introduced by Betts *et al.* [2, 3].

One might ask the number of supercells that should be tried in order to find out the best ones. Fortunately there are finite number of supercells that contains N lattice points. There might be infinitely many, but almost all of them are equivalent to each other. For a given number of N, we list all the unique supercells then we grade each of them by using Betts criteria. This will be very helpful for guiding the choice of supercells for 2-dimensional calculations.

Chapter 2

Lattice and Sublattice

2.1 Lattice

A d-dimensional lattice is an infinite set of points defined by all linear combinations over $\mathbb{Z}$ of a set of linearly independent vectors $\mathbf{a}_i \in \mathbb{R}^d$. The vectors $\mathbf{a}_i$ are said to generate or span lattice $\mathcal{L}$ and are called primitive vectors of $\mathcal{L}$.

$$\mathcal{L} = \left\{ \mathbf{r}_{n_1,\ldots,n_d} = \sum_{i=1}^{d} n_i \mathbf{a}_i \;\middle|\; n_i \in \mathbb{Z} \right\} \tag{2.1}$$

We arrange the (column) vectors $\mathbf{a}_i$ into matrix $\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \ldots)$. For instance, for the case of $d = 3$ the primitive vectors take form of a 3×3 matrix

$$\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3) = \begin{pmatrix} \mathbf{a}_{1_x} & \mathbf{a}_{2_x} & \mathbf{a}_{3_x} \\ \mathbf{a}_{1_y} & \mathbf{a}_{2_y} & \mathbf{a}_{3_y} \\ \mathbf{a}_{1_z} & \mathbf{a}_{2_z} & \mathbf{a}_{3_z} \end{pmatrix} . \tag{2.2}$$

Thus, any point $\mathbf{r_n} \in \mathbb{R}^d$ in lattice $\mathcal{L}$ can be expressed as matrix-vector product $\mathbf{r_n} = \mathbf{An}$ where $\mathbf{n} \in \mathbb{Z}^d$. Vector $\mathbf{n}$ contains the indices of the lattice points, and $\mathbf{n}$ can be considered as the representation of a lattice point in basis $\mathbf{A}$, in other words $\mathbf{A}$ maps lattice points from $\mathbb{Z}^d$ into $\mathbb{R}^d$. For example, Figure 2.1 shows how matrix $\mathbf{A}$ maps from index space into real space. In practice we can work either in coordinate space or with its integer indices, nevertheless there are many advantages to work with integer matrices in the sense of math and programming.

Likewise, if an arbitrary point $\mathbf{r} = \mathbf{Af}$, then the coordinate in basis $\mathbf{A}$ is $\mathbf{f} = \mathbf{A}^{-1}\mathbf{r}$. Vector $\mathbf{f}$ is not an integer vector unless $\mathbf{r}$ is in lattice $\mathcal{L}$. It is called fractional coordinates of $\mathbf{r}$. The primitive lattice cell defined by $\mathbf{A}$ is the set of all points $\mathbf{r}$ with fractional coordinate $\mathbf{f} \in [0, 1)^d$. Its determinant $|\det(\mathbf{A})|$ has a natural geometric interpretation as volume of parallelepiped in $\mathbb{R}^d$, with edges $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d$. Thence, it is the volume of primitive lattice cell $V_c = |\det(\mathbf{A})|$. An example of a primitive cell in two dimensions is illustrated in Figure 2.2 with its volume indicate with the grey area.

However, the primitive vectors $\mathbf{A}$ are not the only primitive vectors that span lattice $\mathcal{L}$. We

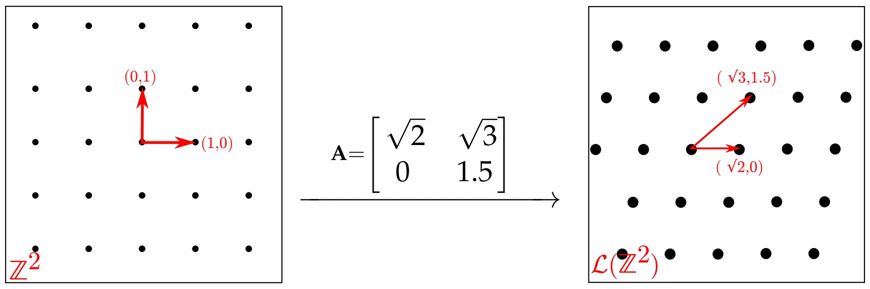

(0,1)
(1,0)
$\mathbb{Z}^2$
$\mathbf{A}=\begin{bmatrix}\sqrt{2} & \sqrt{3}\\ 0 & 1.5\end{bmatrix}$
$(\sqrt{3},1.5)$
$(\sqrt{2},0)$
$\mathcal{L}(\mathbb{Z}^2)$

Figure 2.1: Matrix $\mathbf{A}$ maps integer points on the left side into lattice points on the right side.

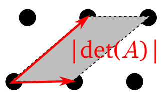


Figure 2.2: Example of determinant representation primitive lattice cell.

can construct other primitive vectors $\tilde{\mathbf{A}}$ by adding to primitive vector $\mathbf{a}_i$ any integer multiple of $\mathbf{a}_{j\neq i}$, such that $|\det(\tilde{\mathbf{A}})| = |\det(\mathbf{A})|$. In the trivial case, for $d = 1$ the lattice $\mathcal{L}$ is generated by a nonzero real number $\mathbf{a}$ consisting of integer multiples $\mathbf{a}$. In this case, the only other possible primitive vectors is $\tilde{\mathbf{a}} = -\mathbf{a}$. If $d \geq 2$, then there can be infinitely many $\tilde{\mathbf{A}}$.

If $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d$ and $\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2, \ldots, \tilde{\mathbf{a}}_d$ are two sets of primitive vectors that span the same lattice $\mathcal{L}$, thence every $\mathbf{a}_i$ belongs to the lattice with primitive vectors $\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2, \ldots, \tilde{\mathbf{a}}_d$ likewise every $\tilde{\mathbf{a}}_i$ belongs to the lattice with primitive vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d$. It follows that

$$\mathbf{a}_i = \sum_{j=1}^{n} m_{ij}\tilde{\mathbf{a}}_j \qquad \tilde{\mathbf{a}}_i = \sum_{j=1}^{n} \tilde{m}_{ij}\mathbf{a}_\mathbf{j} \qquad i = 1, 2, \ldots, d, \tag{2.3}$$

Where both $\mathbf{M} = \mathbf{m}_{ij}$ and $\tilde{\mathbf{M}} = \tilde{\mathbf{m}}_{ij}$ are square matrices with integer entries. Writing those equations in matrix form gives $\mathbf{A} = \tilde{\mathbf{A}}\mathbf{M}$ and $\tilde{\mathbf{A}} = \mathbf{A}\tilde{\mathbf{M}}$, hence $\mathbf{A} = \mathbf{A}\tilde{\mathbf{M}}\mathbf{M}$ and $\tilde{\mathbf{A}} = \tilde{\mathbf{A}}\mathbf{M}\tilde{\mathbf{M}}$. Since primitive vectors are linearly independent, both matrices $\tilde{\mathbf{A}}$ and $\mathbf{A}$ are invertible, therefore $\mathbf{M}\tilde{\mathbf{M}} = \mathbf{I}$ and $\tilde{\mathbf{M}}\mathbf{M} = \mathbf{I}$, and so $\det(\mathbf{M})\det(\tilde{\mathbf{M}}) = 1$.

Note that $\mathbf{M}$ and $\tilde{\mathbf{M}}$ are integer matrices, so either $\det(\mathbf{M}) = \det(\tilde{\mathbf{M}}) = 1$ or $\det(\mathbf{M}) = \det(\tilde{\mathbf{M}}) = -1$. Both $\mathbf{M}$ and $\tilde{\mathbf{M}}$ are *unimodular* matrices, *i.e.* such an integer matrix that has determinant ± 1. The inverse of unimodular matrix is an unimodular matrix, therefore the unimodular operators applied to matrix $\mathbf{A}$ preserve the lattice. Now we can conveniently confirm that for two arbitrary sets of primitive vectors $\mathbf{A}$ and $\tilde{\mathbf{A}}$ the following relation holds.

$$\tilde{\mathbf{A}}^{-1}\mathbf{A} = \mathbf{M} \qquad \mathbf{A}^{-1}\tilde{\mathbf{A}} = \tilde{\mathbf{M}} \tag{2.4}$$

where transformation matrices $\mathbf{M}$ and $\tilde{\mathbf{M}}$ are unimodular, then both primitive vectors span the same lattice. At this point we can prove that the volume of the primitive lattice cell is independent of the choice of primitive vectors,

$$|\det(\mathbf{A})| = |\det(\tilde{\mathbf{A}}\mathbf{M})| = |\det(\tilde{\mathbf{A}})\det(\mathbf{M})| = |\det(\tilde{\mathbf{A}})(\pm 1)| = |\det(\tilde{\mathbf{A}})| \tag{2.5}$$

since both $\mathbf{A}$ and $\tilde{\mathbf{A}}$ are arbitrary sets of primitive vectors, this completes the proof. For a thorough understanding, let us consider Example 1.

Example 1 *Calculate determinant of a primitive lattice cell with different choices of primitive vectors*

$$\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2] = \begin{bmatrix} \sqrt{2} & \sqrt{3} \\ 0 & 1.5 \end{bmatrix}$$

Generate other arbitrary primitive vectors that span the same lattice as $\mathbf{A}$

$$\mathbf{A}_a = [\mathbf{a}_1 + 2\mathbf{a}_2, \mathbf{a}_2] = \begin{bmatrix} \sqrt{2} + 2\sqrt{3} & \sqrt{3} \\ 3 & 1.5 \end{bmatrix}$$

$$\mathbf{A}_b = [\mathbf{a}_1, \mathbf{a}_2 + \mathbf{a}_1] = \begin{bmatrix} \sqrt{2} & \sqrt{3} + \sqrt{2} \\ 0 & 1.5 \end{bmatrix}$$

$$\mathbf{A}_c = [\mathbf{a}_1 + \mathbf{a}_2, \mathbf{a}_2 + 2(\mathbf{a}_1 + \mathbf{a}_2)] = \begin{bmatrix} \sqrt{2} + \sqrt{3} & 3\sqrt{3} + 2\sqrt{2} \\ 1.5 & 4.5 \end{bmatrix}$$

$\det(\mathbf{A}) = \det(\mathbf{A}_a) = \det(\mathbf{A}_b) = \det(\mathbf{A}_c) \approx 2.1213243$

For each set of primitive vectors above, their primitive cells are depicted in Figure 2.3.

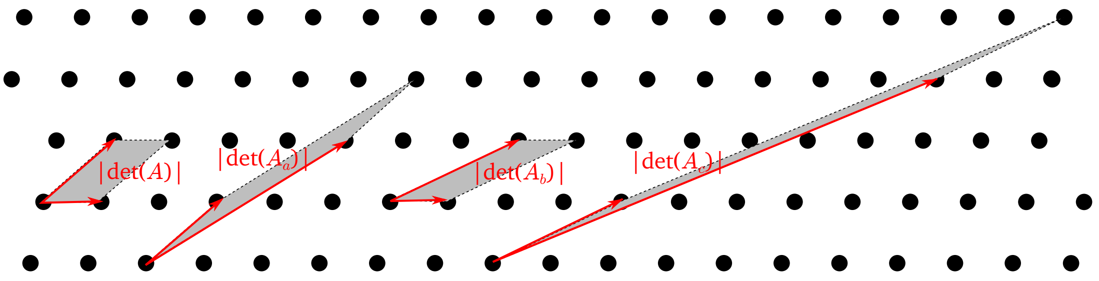


Figure 2.3: $\mathbf{A}, \mathbf{A_a}, \mathbf{A_b}$, and $\mathbf{A_c}$ span the same lattice, denoted by black circles. Each primitive lattice cells determinant is illustrated by grey area.

2.1.1 Additional Examples of Lattices

It is convenient to introduce some examples of simple lattices which are very useful for studies, namely the square lattice and the hexagonal lattice. Square lattice is a rectangular lattice with nearest neighbors having the same distances along horizontal and vertical lines. Hexagonal lattice is also a simple lattice with six nearby points on 6-fold axis.

A square lattice is shown in Figure 2.4 with some possible compact primitive lattice cell vectors. The term "compact" here means that the primitive vectors are as short as possible. Figure 2.4 provides two distinct primitive vectors ($(1,0)(0,1)$ and $(1,0)(1,1)$) while the rest are simply obtained by rotating these vectors on $n\pi/4$. In same manner one can consider the case of hexagonal lattice (see Figure 2.5). Here, we also provide two distinct sets of primitive vectors

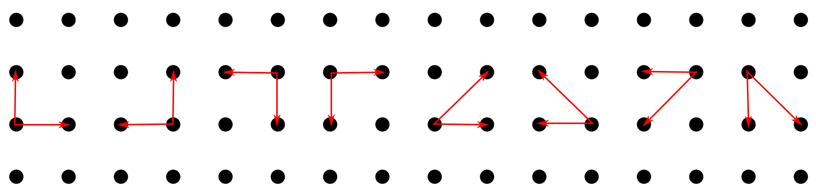


Figure 2.4: Square lattice with some possible compact primitive vectors. There are two distinct primitive cells (the first four and the last four are not related). Among the four identical primitive vectors, they are related by symmetry operation.

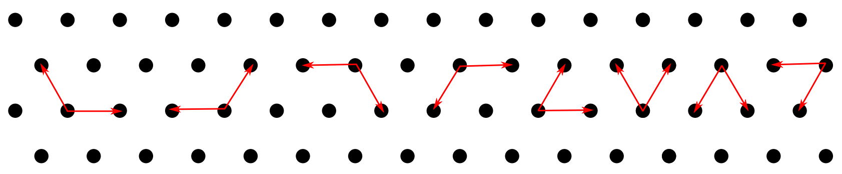


Figure 2.5: Hexagonal lattice with some possible compact primitive vectors. The first four and the last four primitive cells are equivalent. Among the four identical primitive vectors, they are related by symmetry operation.

$((1,0)(-1/2,\sqrt{3}/2)$ and $(1,0)(1/2,\sqrt{3}/2))$, the others are also obtained by rotating these two vectors on $n\pi/6$.

Basically, the primitive vectors that we picked for the hexagonal lattice are equivalent since each primitive vectors has the same length which equals one. The case of the square lattice is different, where $(1,0)(0,1)$ has shorter vectors than $(1,0)(1,1)$. In the end, we conclude that the choice of primitive vectors are unique up to symmetries of lattice which is called point symmetries, and it will be discussed in more details in Section 4.1.

2.1.2 Construct `Class Lattice`

Here we are going to construct class `Lattice` that creates object lattice. From the definition of a lattice, a set of d linearly independent primitive vectors is required. The constructor of class `Lattice` is shown in Listing 2.1. This class has class variable `eps` for epsilon of computation and takes a matrix `A` as primitive vectors. Its instances have properties `vol` for volume, `d` for dimension, and `A` for basis. Functions `assert` in lines `7,9` inquire whether `A` is a square matrix and linearly independent (det $\neq 0$).

Listing 2.1: Constructor of Lattice

```
1 import numpy as np
2
3 class Lattice :
4   eps = 1e-10  #global to Lattice
5   def __init__(self, A):
6     n = A.shape[0]
7     assert(A.shape == (n,n)), "Require a square matrix"
8     det = np.linalg.det(A)
9     assert (det), "Require nonsingular matrix"
```

```
    self.A, self.d, self.vol, eps = A, n, abs(det) #set attributes
```

Following the Equation 2.4, we can compare one set of primitive vectors to another and check if they span the same lattice. Namely, for two arbitrary primitive vectors **A** and **B**, if the relation $\mathbf{M} = \mathbf{B}^{-1}\mathbf{A}$ holds with **M** a unimodular matrix, then **A** and **B** span the same lattice.

We define this equality as a static private method of `Lattice` (see Listing 2.2) which is called `_eq`. We also can add other methods to confirm if one lattice is a subset or a superset to another lattice.

If a lattice that spanned by **A** can be represented by primitive vectors **B**, then the lattice that spanned by **B** is a subset to the lattice that spanned by **A**. It is defined by method `_le`. The comparison subset or equal is defined by method `_le`. The auxiliary method `_is_intm` is used to check if a matrix is an integer matrix, and `_is_unimodular` is used to check if a matrix is unimodular. The comparison methods are set as private static method because it will be invoked by other methods and they will be inherited to the other classes later.

Listing 2.2: Comparison of primitive cells

```
@staticmethod
def _eq(A, B):    #Check if A and B span the same lattice
  return Lattice._is_unimodular(np.linalg.inv(B)*A)

@staticmethod
def _le(A, B):    #Check if A subset of B
  return Lattice._is_intm(np.linalg.inv(A)*B)

@staticmethod
def _ge(A, B):   #Check if A superset of B
  return Lattice._is_intm(np.linalg.inv(B)*A)

@staticmethod
def _is_intm(M): #Check if M is an integer matrix
  return np.amax(abs(np.around(M) - M)) < Lattice.eps

@staticmethod
def _is_unimodular(M):  #Check if M is a unimodular matrix
  return abs(abs(np.linalg.det(M))-1) < Lattice.eps and Lattice._is_intm(M)
```

Now define special methods invoking previous static methods in Listing 2.3, therefore we can easily use mathematical operators >, >=, <, <=, and ==. As shown in the Listing, `__eq__` defines operator ==, `__le__` defines operator <=, `__lt__` defines operator <, `__ge__` defines operator >=, and `__gt__` defines operator >.

Listing 2.3: Special methods of `Lattice`

```
def __eq__(self, latB):  #Check if self lattice equal to lattice(B)
  return self._eq(self.A, latB.A)

def __le__(self, latB):  #Check if self lattice is subset of lattice(B)
  return self._le(self.A, latB.A)

def __lt__(self, latB):  #Check if self lattice is proper subset of lattice(B)
  return self._le(self.A, latB.A) and self.vol < latB.vol #smaller volume also

```

```
def __ge__(self, latB):  #Check if self lattice is superset of lattice(B)
  return self._ge(self.A, latB.A)

def __gt__(self, latB):  #Check if self lattice is proper superset of lattice(B)
  return self._ge(self.A, latB.A) and self.vol > latB.vol #smaller volume also
```

Consider some example of the output of our program. `L1` is a lattice generated by an arbitrary square real matrix, `L2` is generated by integer multiplication of basis vectors of `L1`. Thence, `L1` is sublattice of `L2`.

```
>> L1 = Lattice(np.matrix([[1.2,5.5,2.3],[3,1.5,2.2],[6.7,8,9]]))
>> L2 = Lattice(np.column_stack((L1.A[:,0]*2,L1.A[:,1]*3,L1.A[:,2])))
>> L1.A, L2.A
>> (matrix([[ 1.2,  5.5,  2.3],
           [ 3. ,  1.5,  2.2],
           [ 6.7,  8. ,  9. ]]),
    matrix([[  2.4,  16.5,   2.3],
           [  6. ,   4.5,   2.2],
           [ 13.4,  24. ,   9. ]])

>> L1 < L2, L1 > L2, L1 == L2, L2 >= L1, L2 <= L1
>> (True, False, False, True, False)
```

2.2 Sublattice

Let $\mathcal{L} \subset \mathbb{R}^d$ be a lattice with primitive vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d$. Suppose that vectors $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_d \in \mathcal{L}$ are linearly independent and generate lattice $\mathcal{L}_s$. We call $\mathcal{L}_s$ *sublattice* of $\mathcal{L}$ and it can be written as $\mathcal{L}_s \subseteq \mathcal{L}$. Each of the vectors $\mathbf{s}_i$ belongs to lattice $\mathcal{L}$. Then, it follows that

$$\mathbf{s}_i = \sum_{j=1}^{n} c_{ij}\mathbf{a}_j \qquad i = 1, 2, \ldots, d, \tag{2.6}$$

where $\mathbf{C} = c_{ij}$ is a nonsingular integer square matrix, and in matrix notation it is written as $\mathbf{S} = \mathbf{AC}$. Taking determinant on both sides gives

$$\det(\mathbf{S}) = \det(\mathbf{A})\det(\mathbf{C}), \quad \det(\mathbf{C}) = \frac{\det(\mathbf{S})}{\det(\mathbf{A})}. \tag{2.7}$$

The primitive cell corresponding to $\mathbf{S}$ contains $|\det \mathbf{C}|$ copies of the primitive cell corresponding to $\mathbf{A}$. It is called a *supercell* of the lattice spanned by $\mathbf{A}$. To elaborate more we shall consider Example 2.

Example 2 *Let us calculate the determinant of sublattice cells (supercell).*

$$\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2] = \begin{bmatrix} \sqrt{2} & \sqrt{3} \\ 0 & 1.5 \end{bmatrix}$$

$$\mathbf{S}_1 = [3\mathbf{a}_1, 2\mathbf{a}_2] = \begin{bmatrix} 3\sqrt{2} & 2\sqrt{3} \\ 0 & 3 \end{bmatrix} \qquad \mathbf{S}_2 = [3\mathbf{a}_1 + 2\mathbf{a}_2, 2\mathbf{a}_2] = \begin{bmatrix} 3\sqrt{2} + 2\sqrt{3} & 2\sqrt{3} \\ 3 & 3 \end{bmatrix}$$

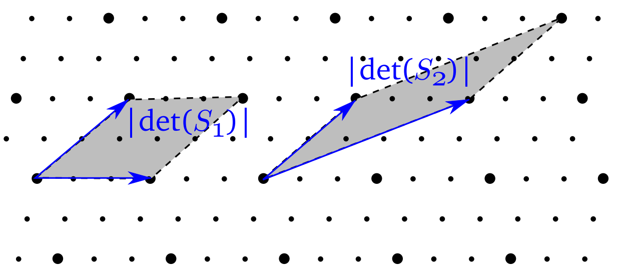


Figure 2.6: Lattice points spanned by $\mathbf{A}$ are noticed by small points, the supercells spanned by $\mathbf{S}_1$ and $\mathbf{S}_2$ are noticed by bigger points. There are six lattice points inside the both supercells.

$\det(\mathbf{S}_1) = \det(\mathbf{S}_2) \approx 12.727922 = 6\det(\mathbf{A})$.

Vectors $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *span the same lattice, which is illustrated in Figure 2.6.*

Matrix $\mathbf{A}$ generates lattice $\mathcal{L}$ and matrix $\mathbf{S}$ generates lattice $\mathcal{L}_s$. Let $\mathcal{L}_s \subseteq \mathcal{L}$, then $\mathbf{S} = \mathbf{AC}$, with $\mathbf{C} \in \mathbb{Z}^{d\times d}$ and $\mathbf{C}$ is not unique. However, we can easily check whether some arbitrary sublattices are the same lattice by checking if they are related by unimodular matrix.

2.2.1 Construct Subclass `Sublattice`

Note that a sublattice is basically a lattice, therefore it is convenient to construct a subclass `Sublattice` that will inherit methods from `Lattice`. From Equation 2.6, definition of a primitive vectors $\mathbf{A}$ and an integer square matrix $\mathbf{C}$ are required to generate a sublattice, thence we create a constructor that requires as arguments matrix `A` and a matrix `C` (see Listing 2.4). An instance of `Sublattice` has attributes `A` for square matrix, `C` for nonsingular square integer matrix, and `vol` for volume of sublattice. In order to check, that matrix `C` consists of integer elements (see line 11), it is sufficient to to check only its first element, since class `numpy.matrix` stores elements of the same data type.

Listing 2.4: Constructor of `Sublattice`

```
 1 from lattice import Lattice   #Inherited from parent class Lattice
 2 import numpy as np
 3
 4 class Sublattice(Lattice):
 5   def __init__(self, A, C):
 6     n = A.shape[0]
 7     assert(A.shape == (n,n)), "Require square matrices"
 8     assert(A.shape == C.shape), "Matrices A and C must have the same shape"
 9     det_a, det_c = np.linalg.det(A), np.linalg.det(C)
10     assert(det_a and det_c), "Require non-singular matrices"
11     assert(isinstance(C[0,0]),int), "Require integer matrix C"
12     self.A, self.C, self.vol, self.d = A, C, abs(det_a*det_c), n
```

Now it is easy to construct methods for simple mathematical comparison by inheriting methods from superclass `Lattice` shown in Listing 2.5. We compare $\mathbf{S} = \mathbf{AC}$ in the same manner with lattice.

Listing 2.5: Special methods of `Sublattice`

```
def __eq__(self, sublB): #Sublattice self and sublattice(B) span the same lattice
  assert(isinstance(sublB,Sublattice)), "Argument is a sublattice"
  return self._eq(self.A*self.C, self.A*sublB.C)

def __le__(self, sublB): #Sublattice self is subset of sublattice(B)
  assert(isinstance(sublB,Sublattice)), "Argument is a sublattice"
  return self._le(self.A*self.C, self.A*sublB.C)

def __lt__(self, sublB): #Sublattice self is proper subset of sublattice(B)
  assert(isinstance(sublB,Sublattice)), "Argument is a sublattice"
  return self._le(self.A*self.C, self.A*sublB.C) and self.vol < sublB.vol #smaller vol

def __ge__(self, sublB): #Sublattice self is superset of sublattice(B)
  assert(isinstance(sublB,Sublattice)), "Argument is a sublattice"
  return self._ge(self.A*self.C, self.A*sublB.C)

def __gt__(self, sublB): #Sublattice self is proper superset of sublattice(B)
  assert(isinstance(sublB,Sublattice)), "Argument is a sublattice"
  return self._ge(self.A*self.C, self.A*sublB.C) and self.vol > sublB.vol #smaller vol
```

To make sure that it works, let us run an example. Firstly, we create primitive vectors `A`, then randomly create integer square matrices `C1` and `C2`. Create sublattice `S1` and `S2`, from `A`, and `C1` and `C2`. Now we can apply the comparison operators. The program output is in the following.

```
>> run lattice.py
>> A  = np.matrix([[1.2,5.5,2.3],[3,1.5,2.2],[6.7,8,9]])
>> C1 = np.matrix(np.random.randint(-5,8,size=(3,3)))
>> C2 = np.column_stack((C1[:,0]*2,C1[:,1]*3,C1[:,2]))
>> C1, C2
>> (matrix([[-5,  5,  2],
           [ 0,  1,  7],
           [-3,  5,  5]]),
    matrix([[-10,  15,   2],
           [  0,   3,   7],
           [ -6,  15,   5]]))
>> S1 = Sublattice(A,C1); S2 = Sublattice(A, C2)
>> S1 < S2, S1 > S2, S1 == S2, S2 >= S1, S2 <= S1
>> (True, False, False, True, False)
```

Chapter 3

List the Supercells

3.1 Hermite Normal Form Matrix

A Hermite Normal Form matrix is defined as a nonsingular integer square matrix with upper triangular form fulfilling the properties of Definition 1.

$$\mathbf{H} = \begin{pmatrix} h_{11} & \dots & \dots & \dots & \dots & h_{1n} \\ & \ddots & & & & \vdots \\ & & h_{ii} & \dots & h_{ij} & \\ & & & \ddots & & \vdots \\ & & & & \ddots & \\ & & & & & h_{nn} \end{pmatrix}$$

Definition 1 *A nonsingular integer matrix* $\mathbf{H} \in \mathbb{Z}$ *is Hermite Normal Form if*

1. $h_{ii} > 0$*, diagonal elements are positive*
2. $0 \leq h_{ij} < h_{ii}$ *where* $i < j$*, off-diagonal elements are less than diagonal element in the same row*
3. $h_{ij} = 0$ *where* $i > j$*, upper triangular matrix*

There are two steps to reduce an arbitrary nonsingular integer matrix to HNF. At first we apply the extended Euclidean algorithm for column operations to obtain an upper triangular matrix with positive elements, then we reduce the off-diagonal elements. The first reduction is based on Euclidean algorithm to find the *greatest common divisor* (gcd)

$$\gcd(a,b) = \begin{cases} |a| & \text{if } b = 0 & \text{(operation of type i)} \\ \gcd(b,a) & \text{if } |a| < b & \text{(operation of type ii)} \\ \gcd(a - \lfloor \frac{a}{b} \rfloor b, b) & \text{if } 0 < |b| \leq a & \text{(operation of type iii)} \end{cases} \cdot \tag{3.1}$$

For each row vector $\mathbf{a_i'} = (a_{i1}, \dots, a_{in})$, apply the sequence of elementary column operations $\sigma(\mathbf{a}')$ such that $\sigma(\mathbf{a_i'}) = (0, \dots, 0, g, \dots, a_{i1})$, where g is the gcd, then we apply σ to the entire column. Basically, these operations are unimodular column operations given in Definition 2.

Definition 2 *A* ***unimodular column operation*** *on a matrix is one of the following elementary column operations:*

i multiply any column with -1

ii interchange any two columns

iii add a column with integer multiplication of other columns

Now we have attained the definition of HNF in Definition 1 for point 1 and 3. Then to achieve point 2, each element a_{ij} in upper off-diagonal that is larger than the corresponding diagonal elements a_{ii} is replaced by its modulo by a_{ii}. Again, this operation is applied toward entire columns.

3.1.1 Construct `method HNF`

Here we add a new method to the `Sublattice` class that reduces an integer matrix into its HNF form. Listing 3.1 shows the method of class `Sublattice` invoking the first step of reduction, the extended Euclidean algorithm to find greatest common divisor called `gcd`. The auxiliary method called `_swap` is for exchanging position between two columns. This method `gcd` is quite straightforward following Definition3.1. It basically manipulates the corresponding matrix in place. These methods are used to transform the corresponding matrix to upper triangular form.

Listing 3.1: Euclidean method

```python
#Extended Euclidean algorithm to find the greatest common divisor that is applied to C
#All operations involved are the elementary column operations
  def gcd(self, C, i_r, i_a, i_b):
    a,b = C[i_r,i_a], C[i_r,i_b]  #Assign a,b as the corresponding columns
    if abs(b) == 0:    #Stop if the corresponding row has form (0,...0,g,...,a_ij)
      if a < 0 : C[:,i_a] *= -1    #Operation type i
      self._swap(C, i_a, i_b)        #Operation type ii
    elif abs(a) < abs(b):
      self._swap(C, i_a,i_b)         #Operation type ii
      self.gcd(C, i_r, i_a, i_b)
    else :
      C[:,i_a] -= int(a//b)*C[:,i_b] #Operation type iii
      self.gcd(C, i_r, i_a, i_b)

#Swap column i_a and i_b of matrix C
  def _swap(self, C, i_a, i_b):
    si = range(self.d)                    #Generate new indices
    si[i_a], si[i_b] = si[i_b], si[i_a] #Swap the necessary indices
    C[:] = C[:,si]           #Arrange column matrix C following si
```

The second step of this transformation is reducing the difference between diagonal and off-diagonal by replacing the off-diagonal elements with its modulo to diagonal elements that lay on the same row, it is invoked by the main function. We construct the full HNF method in function `setHNF` (see Listing 3.2) that sets the HNF form of `C` as property `HNF`. This function consists of two main loops, the first loop reduces the matrix into upper triangular form, then the second loop reduces the difference between main diagonal elements and off-diagonal elements.

Notice that both loops are going through columns in backward direction, it is because the final matrix formed into an upper triangular matrix.

Listing 3.2: HNF method

```
#Set property self.HNF by convert self.C into its HNF form
  def setHNF(self):
    self.HNF = self.C.copy()
    #First reduction to upper triangular matrix
    for i in range(self.d-1, 0, -1): #Loop backward
      for j in range (i):
        self.gcd(self.HNF, i, j, j+1)
    if(self.HNF[0,0] < 0):   #Last part is missing (the most top-left one)
        self.HNF[:,0] *= -1
    for i in range(self.d-2,-1,-1): #Second reduction, reduce differece diag and off-diag
      for j in range (i+1,self.d):
        self.HNF[:,j] -= int(self.HNF[i,j]//self.HNF[i,i])*self.HNF[:,i]
```

Let us put some test to our HNF code. Firstly, create a nonsingular square matrix `A`, a random integer matrix `C1`, and an equivalent basis `C2`. Then `C1` and `C2` should span the same sublattice. The reduced matrices should form into identical HNF form both for `S1.C` and `S2.C`.

```
>> run lattice.py
>> L1 = np.matrix([[1.2,5.5,2.3],[3,1.5,2.2],[6.7,8,9]])
>> C1 = np.matrix(np.random.randint(-8,8,size=(3,3)))
>> C2 = np.column_stack((C1[:,0]+2*C1[:,1],C1[:,1],C1[:,2]+3*C1[:,0]))
>> C1,C2
>> (matrix([[ 0,  5, -5],
            [ 6,  3,  7],
            [-1,  6, -3]]),
    matrix([[10,  5, -5],
            [12,  3, 25],
            [11,  6, -6]]))
>> S1 = Sublattice(A,C1); S2=Sublattice(A,C2)
>> S1.toHNF(), S2.toHNF()
>> S1.HNF, S2.HNF
>> (matrix([[140, 115, 130],
            [  0,   1,   0],
            [  0,   0,   1]]),
    matrix([[140, 115, 130],
            [  0,   1,   0],
            [  0,   0,   1]]))
```

3.1.2 Compare Sublattices by their HNF

We have already had methods for comparing two sublattices in Listing 2.5. Alternatively, we also can check the equality between two sublattices by reducing their integer matrices to HNF form and check if they agree. The advantage of the HNF is that we only need to deal with integer numbers, thence we can simply apply the comparison operator without worrying about numerical errors.

The updated method `__eq__` is shown in Listing 3.3, we comment out the `return` of previous code.

Listing 3.3: Equality by checking its HNF

```
def __eq__(self, sublB):
  self.toHNF(); sublB.toHNF()   #Reduce both integer matrix into HNF
  return np.all(self.C == sublB.C) #Directly compare every elements in matrices B and C
  #return self._eq(self.C, sublB.C) #This line is replaced
```

3.2 Listing the Possible Sublattices

At this point, we are able to point out primitive vectors that span the same lattice. For the sublattice, it is even more convenient as we can reduce the coefficient matrix **C** into HNF form and check if they agree. Recall that the volume of a unit cell is independent of the choice of primitive vectors, it means that each lattice has injective relation with its volume. Thence we may list all possible supercells in HNF form for a given number of lattice points inside supercell.

Referring to Definition 1, in order to construct an HNF matrix, we need to form a triangular matrix with their off-diagonal entries smaller than its corresponding diagonal elements. The determinant of a triangular matrix is simply product of all its diagonal elements. In principle for a given volume we can distribute its prime factors as diagonal elements and place any numbers that smaller than its diagonal, then we will have a long list of HNF matrices. Before constructing the possible HNF matrices, firstly we need to list the possible diagonals. Here will be shown two different codes to generate possible diagonals with given volume and dimension. The first method is shown in Listing 3.4 and the second method is shown in Listing 3.5.

In Listing 3.4, the main function `list_diag` has arguments `det` and `dim` that mean determinant and dimension consecutively. An auxiliary function called `prime_factor` with argument `n` returns a *generator* for a list of prime factors of `n`. The main function `list_diag` creates list of prime factors `prime_list` which is multiplication prime factors of `det` and distribute them through diagonals. The initial diagonal is a vector with entries one, then we place one by one the prime numbers for all the possibilities, which is stored in an array `prime_list`. In this case, some existed combinations might encountered but line `9` checks every combination in the list `prime_list` and eliminate if the combination has already existed. At last generate combination that has product equal to determinant (line `11-12`).

Listing 3.4: First code: lists the possible diagonals

```
 1 # List of the possible diagonals
 2 def list_diag(det,dim) :
 3   diag_list = [[1]*dim]  #Initial configuration [1,1,..1]
 4   for fact in prime_factor(det):    #Loop over all factors, try all combinations
 5     for d in range(len(diag_list)):
 6       for i in range(dim):
 7         temp = diag_list[d][:]
 8         temp[i]*= fact
 9         if temp not in diag_list: #Check if combination has not existed
10           diag_list.append(temp)
11           if reduce(lambda a,b: a*b, temp) == det: #Generate if determinant correct
12             yield temp
13
14 #List the prime factors of n, output [f1,f2,...]
```

```
def prime_factor(n):
  f=2 #trial factor
  while f*f <= n:
    while n%f == 0 :
      n/=f
      yield f
    f+=1
  if n>1 : yield n
```

The first code may be quite brief and simple, but always keeping track of the combination of diagonals (at line 9) costs some time. Fortunately there is some more efficient way for listing the diagonals by distributing each prime factor to the right places. In this way it will be more predictable but less simple, this second code is shown in Listing 3.5

There the main function `list_diag2` has the same arguments with `list_diag` from the first code. The auxiliary function `prime_factor_mul` lists the prime factors with its multiplication number. It is similar to function `prime_factor` but different format of return, for example a return `[[2,3],[5,2]]` means $2 \cdot 2 \cdot 2 \cdot 5 \cdot 5$.

Listing 3.5: Second code: lists the possible diagonals

```
#List of possible diagonals
def list_diag2(det,dim):
  pind = []   #Prime indices
 #Produce list of where to put prime factors
  for p,m in prime_factor(det):
    lind=[[i] for i in range(dim)] #Initially, factor can be placed anywhere
    for mm in range(m-1):          #Place another multiplied factor
      lnew=[]
      for l in lind:
        for i in range(l[-1],dim): #Other factor, placed from previous site and after
          ln=l[:]
          ln.append(i)
          lnew.append(ln)
      lind=lnew
    pind.append([p,lind])   # [prime_factor, list of indices]
  #Now produce list of all diagonals
  diag=[[1]*dim]    #Initial diagonal [[1],...,[1]]
  for p, lind in pind:
    dnew=[]
    for dd in diag:  #Placing each diagonal at lind ([i_1,i_2,..])
      for l in lind:
        dn=dd[:]
        for i in l:
          dn[i]*=p
        dnew.append(dn)
    diag=dnew
  return diag

#List the prime factors with multiplicity, output format [[f1,m1],..,[fn,mn]]
def prime_factor_mul(n):
  f=2    #trial factor
  while f*f <= n:
    m=0
    while n%f == 0 :
      m+=1
      n/=f
    if(m) : yield [f,m]
    f+=1
  if n>1 : yield [n,1]
```

Basically distributing prime factors through the diagonal is similar to distributing boson particles. The main function `list_diag2` (see Listing 3.5) consists of two main steps, producing a list which stores positions in diagonal of the matrix to place the prime factors and then producing the proper diagonals.

The first step consists of four nested loops, it places the prime factors anywhere in diagonal, except for the factors that appear more than once: we must place the next factor in prior or in the same place. As it might be somewhat puzzling, then let us see some example.

For a given factor with multiplicity 2 that is distributed on a 3-dimensional diagonal, we list the index to place the prime factors: initially the factor can be placed anywhere: index `[0]`, `[1]` and `[2]`. Then, we need to place another one because the multiplicity is 2. For the one that is placed at `[0]`, the possibility is only `[0,0]`, for the one at `[1]`, the possibilities are `[1,0]` and `[1,1]`, for the one at `[2]`, the possibilities are `[2,0]`, `[2,1]` and `[2,2]`. Thence, the complete list is `[[0,0], [1,0], [1,1], [2,0], [2,1], [2,2]]`. Now that we have both of the codes, lets run some test and both must produce the same results.

```
>> zip([a for a in list_diag(8,3)],list_diag2(8,3))
>> [([8, 1, 1], [8, 1, 1]),
    ([4, 2, 1], [4, 2, 1]),
    ([4, 1, 2], [4, 1, 2]),
    ([2, 4, 1], [2, 4, 1]),
    ([2, 2, 2], [2, 2, 2]),
    ([2, 1, 4], [2, 1, 4]),
    ([1, 8, 1], [1, 8, 1]),
    ([1, 4, 2], [1, 4, 2]),
    ([1, 2, 4], [1, 2, 4]),
    ([1, 1, 8], [1, 1, 8])]
```

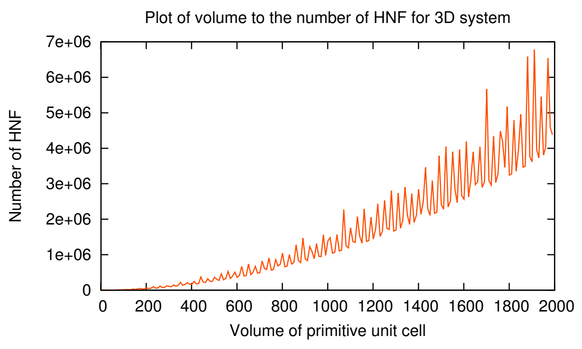
Note that the first code returns a generator while the second one returns a complete list, to be clear it is only the matter of programming taste.

The final step is to construct all possible HNF matrices based on Definition 1, point 2. Before writing the code, it is helpful to know how many HNF matrices will be produced. Firstly we need to count the number diagonals that will be generated. Note that it is the same way with counting the configuration of distributing bosons. For each prime factor the possibility of different distribution is simply equal to $\binom{m+d-1}{m}$, for m as number of factor multiplication and d for dimension. Then the number of total possible diagonal N_{diag} is given by

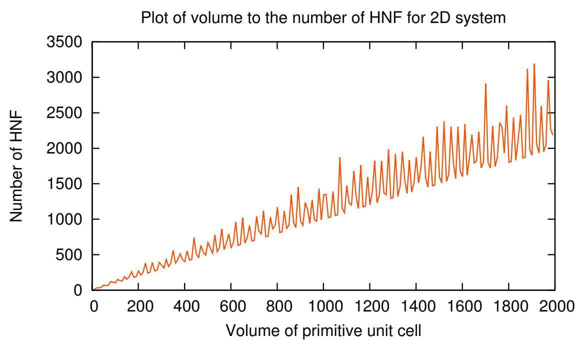
$$N_{diag} = \prod_{i=1}^{n} \binom{m+d-1}{m}, \qquad n = \text{number of prime factors}. \tag{3.2}$$

Secondly, the number of HNFs depends on the volume and dimension. For a d-dimensional coordinate system, with all configurations of diagonal elements $a_1, \ldots, a_d$, the number of possible HNF is shown by Equation 3.3.

$$N_{HNF} = \sum_{(a_1,\ldots,a_d)} a_1^{d-1} \times a_2^{d-2} \ldots \times a_{d-1}. \tag{3.3}$$



(a)



(b)

Figure 3.1: Number of possible HNFs increases to the volume of primitive cells. (a) Three-dimensional system. (b) Two-dimensional system.

Then it is easy to show that the number of possible HNFs will increase exponentially to the volume and dimension. Some examples of the relation between the number of HNFs and volume are shown in Figure 3.1a and Figure 3.1b for 3-dimension and 2-dimension respectively.

Figure 3.1a and 3.1b illustrate that the possible number of HNF increases quickly with dimension. If we compare both figures for determinants up to 2000, for the case of 3-*d*, the number of HNFs develops in scale of millions, on the other hand, in 2-*d* the number of HNFs almost linear to the determinant. Some samples of the number of HNFs as a function of volume are shown in Tables 3.1 and 3.2 for 3-*d* and 2-*d* system respectively.

Table 3.1: Count of distinct HNF matrices for given volume of primitive cell for 3-*d* system.

Volume of supercell	**Number of HNF**
1	1
2	7
3	13
4	35
5	31
6	91
7	57
8	155
9	130
10	217
11	133
12	455

Now we are ready to list all the possibilities of HNF matrices (see Listing 3.6). The methods `numpy.diag` and `itertools.product` are needed for creating diagonal square matrices and for invoking recursive loops consecutively. The main function `list_hnf` requires the determinant

Table 3.2: Count of distinct HNF matrices for given volume of primitive cell for 2-*d* system.

Volume of supercell	Number of HNF
1	1
3	4
5	6
7	8
9	13
11	12
13	14
15	24
17	18
19	20
21	32
23	24

and the dimension as arguments. Principally, it loops through all possible diagonals, then calls function `iterate` that replaces each row of possible off-diagonal recursively.

Listing 3.6: List all possible HNF matrices

```python
from numpy import diag
from itertools import product

#List all possible HNF
def list_hnf(det, dim):
  ldiag, lhnf = list_diag2(det,dim), []  #Get the list of diagonal firstly
  for vdiag in ldiag:                    #Vector diagonal
    m = diag(vdiag)                     #Create diagonal matrix from vdiag
    iterate(m, 0, dim, lhnf)            #Loop for all possible off-diagonal element
  return lhnf

#Recursive loop, list all possible off diagonals from
def iterate(mat, row, dim, lst):
  for a in product(range(mat[row,row]),repeat=dim-row-1): #Possible numbers: 0-(diag-1)
    mat[row,row+1:] = a
    if(dim-2 > row):
      iterate(mat, row+1, dim, lst)      #Move to the next row
    if(row == dim-2):                   #When reach second last row, it is done
      lst.append(mat.copy())
```

We have tried to list all possible diagonals for determinant 8 and dimension 3. Applying Equation 3.3, counted that there are 155 possible HNF matrices, the output is the following.

```
>> L = list_hnf(8,3)
>> len(L)
>> 155
>> for l in L : print L
>> 8  0  0         8  1  6         8  3  4         8  5  2         8  7  0         4  0  3
   0  1  0         0  1  0         0  1  0         0  1  0         0  1  0         0  2  0
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  0  1         8  1  7         8  3  5         8  5  3         8  7  1         4  0  3
   0  1  0         0  1  0         0  1  0         0  1  0         0  1  0         0  2  1
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  0  2         8  2  0         8  3  6         8  5  4         8  7  2         4  1  0
   0  1  0         0  1  0         0  1  0         0  1  0         0  1  0         0  2  0
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  0  3         8  2  1         8  3  7         8  5  5         8  7  3         4  1  0
   0  1  0         0  1  0         0  1  0         0  1  0         0  1  0         0  2  1
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  0  4         8  2  2         8  4  0         8  5  6         8  7  4         4  1  1
   0  1  0         0  1  0         0  1  0         0  1  0         0  1  0         0  2  0
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  0  5         8  2  3         8  4  1         8  5  7         8  7  5         4  1  1
   0  1  0         0  1  0         0  1  0         0  1  0         0  1  0         0  2  1
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  0  6         8  2  4         8  4  2         8  6  0         8  7  6         4  1  2
   0  1  0         0  1  0         0  1  0         0  1  0         0  1  0         0  2  0
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  0  7         8  2  5         8  4  3         8  6  1         8  7  7         4  1  2
   0  1  0         0  1  0         0  1  0         0  1  0         0  1  0         0  2  1
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  1  0         8  2  6         8  4  4         8  6  2         4  0  0         4  1  3
   0  1  0         0  1  0         0  1  0         0  1  0         0  2  0         0  2  0
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  1  1         8  2  7         8  4  5         8  6  3         4  0  0         4  1  3
   0  1  0         0  1  0         0  1  0         0  1  0         0  2  1         0  2  1
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  1  2         8  3  0         8  4  6         8  6  4         4  0  1         4  2  0
   0  1  0         0  1  0         0  1  0         0  1  0         0  2  0         0  2  0
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  1  3         8  3  1         8  4  7         8  6  5         4  0  1         4  2  0
   0  1  0         0  1  0         0  1  0         0  1  0         0  2  1         0  2  1
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  1  4         8  3  2         8  5  0         8  6  6         4  0  2         4  2  1
   0  1  0         0  1  0         0  1  0         0  1  0         0  2  0         0  2  0
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1

   8  1  5         8  3  3         8  5  1         8  6  7         4  0  2         4  2  1
   0  1  0         0  1  0         0  1  0         0  1  0         0  2  1         0  2  1
   0  0  1         0  0  1         0  0  1         0  0  1         0  0  1         0  0  1
```

```
4  2  2     4  0  0     4  3  0     2  1  0     2  1  0     1  0  0
0  2  0     0  1  0     0  1  0     0  4  0     0  2  0     0  8  4
0  0  1     0  0  2     0  0  2     0  0  1     0  0  2     0  0  1

4  2  2     4  0  1     4  3  1     2  1  0     2  1  0     1  0  0
0  2  1     0  1  0     0  1  0     0  4  1     0  2  1     0  8  5
0  0  1     0  0  2     0  0  2     0  0  1     0  0  2     0  0  1

4  2  3     4  0  2     4  3  2     2  1  0     2  1  1     1  0  0
0  2  0     0  1  0     0  1  0     0  4  2     0  2  0     0  8  6
0  0  1     0  0  2     0  0  2     0  0  1     0  0  2     0  0  1

4  2  3     4  0  3     4  3  3     2  1  0     2  1  1     1  0  0
0  2  1     0  1  0     0  1  0     0  4  3     0  2  1     0  8  7
0  0  1     0  0  2     0  0  2     0  0  1     0  0  2     0  0  1

4  3  0     4  1  0     2  0  0     2  1  1     2  0  0     1  0  0
0  2  0     0  1  0     0  4  0     0  4  0     0  1  0     0  4  0
0  0  1     0  0  2     0  0  1     0  0  1     0  0  4     0  0  2

4  3  0     4  1  1     2  0  0     2  1  1     2  0  1     1  0  0
0  2  1     0  1  0     0  4  1     0  4  1     0  1  0     0  4  1
0  0  1     0  0  2     0  0  1     0  0  1     0  0  4     0  0  2

4  3  1     4  1  2     2  0  0     2  1  1     2  1  0     1  0  0
0  2  0     0  1  0     0  4  2     0  4  2     0  1  0     0  4  2
0  0  1     0  0  2     0  0  1     0  0  1     0  0  4     0  0  2

4  3  1     4  1  3     2  0  0     2  1  1     2  1  1     1  0  0
0  2  1     0  1  0     0  4  3     0  4  3     0  1  0     0  4  3
0  0  1     0  0  2     0  0  1     0  0  1     0  0  4     0  0  2

4  3  2     4  2  0     2  0  1     2  0  0     1  0  0     1  0  0
0  2  0     0  1  0     0  4  0     0  2  0     0  8  0     0  2  0
0  0  1     0  0  2     0  0  1     0  0  2     0  0  1     0  0  4

4  3  2     4  2  1     2  0  1     2  0  0     1  0  0     1  0  0
0  2  1     0  1  0     0  4  1     0  2  1     0  8  1     0  2  1
0  0  1     0  0  2     0  0  1     0  0  2     0  0  1     0  0  4

4  3  3     4  2  2     2  0  1     2  0  1     1  0  0     1  0  0
0  2  0     0  1  0     0  4  2     0  2  0     0  8  2     0  1  0
0  0  1     0  0  2     0  0  1     0  0  2     0  0  1     0  0  8

4  3  3     4  2  3     2  0  1     2  0  1     1  0  0
0  2  1     0  1  0     0  4  3     0  2  1     0  8  3
0  0  1     0  0  2     0  0  1     0  0  2     0  0  1
```

Chapter 4

Symmetry

4.1 Symmetry of Lattice

In Section 2.1.1, we have shown some examples of symmetries of rotation for the square and hexagonal lattice. Generally speaking, rotations in Euclidean space of 2 or higher dimension constitute a continuous, infinite set which includes all arbitrary infinitesimal rotation operations. Moreover, the two sequential rotation operations are equivalent to a distinct rotation. A group with this essential feature is called *Lie group*.

We must pick the symmetry rotations within Lie group that associate with our lattice. In the following we formulate the criteria of these symmetries. Consider an orthogonal transformation matrix **P** which is a symmetry operation that keeps the origin fixed (point symmetry) and a given set of primitive vectors **A**. Note that **P** can be any symmetry operations which is not restricted to only symmetry rotation. Matrix $\tilde{\mathbf{A}}$ is the transformed primitive vectors of **A**, then it follows that

$$\begin{aligned} \mathbf{PA} &= \tilde{\mathbf{A}} \\ \mathbf{PA} &= \mathbf{AM} \\ \mathbf{A}^{-1}\mathbf{PA} &= \mathbf{M}. \end{aligned} \tag{4.1}$$

Lattices that constructed by $\tilde{\mathbf{A}}$ and **A** are equivalent, $\mathcal{L}(\tilde{\mathbf{A}}) \equiv \mathcal{L}(\mathbf{A})$ if there exists a unimodular matrix **M** such that $\tilde{\mathbf{A}} = \mathbf{AM}$. Matrix **M** is a matrix column operation that applied to **A** to produce $\tilde{\mathbf{A}}$, hence if **M** is unimodular then both **A** and $\tilde{\mathbf{A}}$ span the same lattice. Correspondingly, we list the symmetry operation that associate to **A** by operating all possible symmetries and pick ones that produce a unimodular matrix **M**. Here we list all symmetry operations that associate with square, rectangular, and hexagonal lattice.

Tables 4.1, 4.2, and 4.3 show the symmetries of a square lattice for three different choices of primitive vectors. There are eight symmetry operations that associate to rectangular lattice, these operations belong to group symmetry of C_{4v} (or D_4). Notice, that the symmetry operations **P** here are also associated with the signed permutation group. These symmetry operations consists of four rotations and four reflections. The unit operation is denoted by E which is equivalent to rotation of 2π, the rotations of $\pi/2$, π, and $3\pi/2$ are denoted by C_4, C_4^2, and C_4^3 respectively. The reflections along vertical, diagonal, horizontal, and anti-diagonal axes are denoted by σ_y, $C_4\sigma_y$, $C_4^2\sigma_y$, and $C_4^3\sigma_y$ respectively.

Table 4.1 lists the symmetries of a square lattice with primitive vectors a unit cell matrix $(1,0)(0,1)$. Hence, we can expect that matrix $\mathbf{M}$ will be equal to the transformation matrix since $\mathbf{M} = \mathbf{A}^{-1}\mathbf{P}\mathbf{A} = \mathbf{P}$. On the other hand, we list the symmetries of the square lattice with non-compact choice of primitive vectors in order to observe the direct relation between $\mathbf{A}$ and $\mathbf{M}$, but we could not find any proof.

Table 4.1: Symmetries of square lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
C_4	$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$
C_4^2	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
C_4^3	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$
σ_y	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$
$C_4\sigma_y$	$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$
$C_4^2\sigma_y$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
$C_4^3\sigma_y$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

Table 4.2: Symmetries of square lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

C_4	$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 \\ 2 & 1 \end{bmatrix}$
C_4^2	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ -1 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
C_4^3	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ -2 & -1 \end{bmatrix}$
σ_y	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 1 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$
$C_4\sigma_y$	$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 \\ 0 & 1 \end{bmatrix}$
$C_4^2\sigma_y$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ -1 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ -2 & -1 \end{bmatrix}$
$C_4^3\sigma_y$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix}$

Table 4.3: Symmetries of square lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 4 & 1 \\ 3 & 1 \end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 4 & 1 \\ 3 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
C_4	$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -3 & -1 \\ 4 & 1 \end{bmatrix}$	$\begin{bmatrix} -7 & -2 \\ 25 & 7 \end{bmatrix}$
C_4^2	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -4 & -1 \\ -3 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
C_4^3	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 3 & 1 \\ -4 & -1 \end{bmatrix}$	$\begin{bmatrix} 7 & 2 \\ -25 & -7 \end{bmatrix}$
σ_y	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -4 & -1 \\ 3 & 1 \end{bmatrix}$	$\begin{bmatrix} -7 & -2 \\ 24 & 7 \end{bmatrix}$
$C_4\sigma_y$	$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -3 & -1 \\ -4 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ -7 & -1 \end{bmatrix}$

$C_4^2\sigma_y$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 4 & 1 \\ -3 & -1 \end{bmatrix}$	$\begin{bmatrix} 7 & 2 \\ -24 & -7 \end{bmatrix}$
$C_4^3\sigma_y$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 3 & 1 \\ 4 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 7 & 1 \end{bmatrix}$

Now let us consider the point group symmetries of rectangular lattices. It is known that the square lattice (2-*d* hypercube) possesses the highest symmetry among the other parallelograms. Hence, we might estimate that the symmetries of rectangular lattice must be a subgroup of symmetries of square lattice.

Correspondingly, we construct symmetries of rectangular lattices by applying all elements of the symmetry point group of square lattice and eliminate the ones that have a non-unimodular matrix **M**. Tables 4.4 and 4.5 list the symmetries of the compact and non-compact choice of primitive vectors respectively. It appears that the rectangular lattice has point group symmetry of C_{2v} (or D_2). The symmetry operations consists of two rotations π and 2π denoted by C_2 and E respectively, and two reflections along y-axis and x-axis denoted by σ_y and $C_2\sigma_y$ respectively.

Table 4.4: Symmetries of rectangular lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$.

Symmetry operation	**P**	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
C_2 (C_4^2)	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -2 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
σ_y	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 2 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$
$C_2\sigma_y$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & -2 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

Table 4.5: Symmetries of rectangular lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$.

Symmetry operation	**P**	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

C_2	$\begin{bmatrix}-1 & 0\\ 0 & -1\end{bmatrix}$	$\begin{bmatrix}-1 & -1\\ 0 & -2\end{bmatrix}$	$\begin{bmatrix}-1 & 0\\ 0 & -1\end{bmatrix}$
σ_y	$\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix}$	$\begin{bmatrix}-1 & -1\\ 0 & 2\end{bmatrix}$	$\begin{bmatrix}-1 & -2\\ 0 & 1\end{bmatrix}$
$C_2\sigma_y$	$\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$	$\begin{bmatrix}1 & 1\\ 0 & -2\end{bmatrix}$	$\begin{bmatrix}1 & 2\\ 0 & -1\end{bmatrix}$

Table 4.6: Symmetries of rectangular lattice with primitive vectors $\mathbf{A} = \begin{bmatrix}1 & 2\\ 2 & 6\end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 2\\ 2 & 6\end{bmatrix}$	$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$
C_2	$\begin{bmatrix}-1 & 0\\ 0 & -1\end{bmatrix}$	$\begin{bmatrix}-1 & -2\\ -2 & -6\end{bmatrix}$	$\begin{bmatrix}-1 & 0\\ 0 & -1\end{bmatrix}$
σ_y	$\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix}$	$\begin{bmatrix}-1 & -2\\ 2 & 6\end{bmatrix}$	$\begin{bmatrix}-5 & -12\\ 2 & 5\end{bmatrix}$
$C_2\sigma_y$	$\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$	$\begin{bmatrix}1 & 2\\ -2 & -6\end{bmatrix}$	$\begin{bmatrix}5 & 12\\ -2 & -5\end{bmatrix}$

At last, we list the symmetry operations that are associated with the hexagonal lattice (see Table 4.7). There are more of symmetries in the hexagonal lattice than the square lattice since there are six lattice points on the nearest neighbor.

It appears that the point symmetry of the hexagonal lattice belongs to group C_{6v} (or D_6). The symmetry operations consists of six rotations and six reflections. The notations C_6, C_6^2, C_6^3, C_6^4, and C_6^5 denote the rotations of $\pi/3$, $2\pi/3$, π, $4\pi/3$, and $5\pi/3$ respectively. The notations σ_y, $C_6\sigma_y$, and $C_6^2\sigma_y$ denote the reflections along the axes between edges, and $C_6^3\sigma_y$, $C_6^4\sigma_y$, and $C_6^5\sigma_y$ denote the reflections along the axes between vertices.

Table 4.7: Symmetries of hexagonal lattice with primitive vectors $\mathbf{A} = \begin{bmatrix}1 & -1/2\\ 0 & \sqrt{3}/2\end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$

E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & -1/2 \\ 0 & \sqrt{3}/2 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
C_6	$\begin{bmatrix} 1/2 & -\sqrt{3}/2 \\ \sqrt{3}/2 & 1/2 \end{bmatrix}$	$\begin{bmatrix} 1/2 & -1 \\ \sqrt{3}/2 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$
C_6^2	$\begin{bmatrix} -1/2 & -\sqrt{3}/2 \\ \sqrt{3}/2 & -1/2 \end{bmatrix}$	$\begin{bmatrix} -1/2 & -1/2 \\ \sqrt{3}/2 & -\sqrt{3}/2 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 \\ 1 & -1 \end{bmatrix}$
C_6^3	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & 1/2 \\ 0 & -\sqrt{3}/2 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
C_6^4	$\begin{bmatrix} -1/2 & \sqrt{3}/2 \\ -\sqrt{3}/2 & -1/2 \end{bmatrix}$	$\begin{bmatrix} -1/2 & 1 \\ -\sqrt{3}/2 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & 1 \\ -1 & 0 \end{bmatrix}$
C_6^5	$\begin{bmatrix} 1/2 & \sqrt{3}/2 \\ -\sqrt{3}/2 & 1/2 \end{bmatrix}$	$\begin{bmatrix} 1/2 & 1/2 \\ -\sqrt{3}/2 & \sqrt{3}/2 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix}$
σ_y	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 1/2 \\ 0 & \sqrt{3}/2 \end{bmatrix}$	$\begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix}$
$C_6\sigma_y$	$\begin{bmatrix} -1/2 & -\sqrt{3}/2 \\ -\sqrt{3}/2 & 1/2 \end{bmatrix}$	$\begin{bmatrix} -1/2 & -1/2 \\ -\sqrt{3}/2 & \sqrt{3}/2 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ -1 & 1 \end{bmatrix}$
$C_6^2\sigma_y$	$\begin{bmatrix} 1/2 & -\sqrt{3}/2 \\ -\sqrt{3}/2 & -1/2 \end{bmatrix}$	$\begin{bmatrix} 1/2 & -1 \\ -\sqrt{3}/2 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$
$C_6^3\sigma_y$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & -1/2 \\ 0 & -\sqrt{3}/2 \end{bmatrix}$	$\begin{bmatrix} 1 & -1 \\ 0 & -1 \end{bmatrix}$
$C_6^4\sigma_y$	$\begin{bmatrix} 1/2 & \sqrt{3}/2 \\ \sqrt{3}/2 & -1/2 \end{bmatrix}$	$\begin{bmatrix} 1/2 & 1/2 \\ \sqrt{3}/2 & -\sqrt{3}/2 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix}$
$C_6^5\sigma_y$	$\begin{bmatrix} -1/2 & \sqrt{3}/2 \\ \sqrt{3}/2 & 1/2 \end{bmatrix}$	$\begin{bmatrix} -1/2 & 1 \\ \sqrt{3}/2 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

4.2 Rotated Lattice

Now we might ask ourselves what happens when our lattice is slightly rotated? All symmetries of rotation will remain but not the reflections. Imagine that symmetries of rotation do not depend on alignment of axis, but reflection apparently depend on how the axis is aligned. However, when the lattice is rotated, the reflection axes are rotated as well. Thus, we shall consider the reflection operations about the non-perpendicular axes.

In the following, we list some examples of square and rectangular lattices that is rotated by 10°. Tables 4.8 and 4.9 list the symmetries for tilted square lattices, and Tables 4.10 and 4.11 list the symmetries of tilted rectangular lattice. Notice that all symmetries of rotation are maintained but none of the reflections corresponding to the prior tables. Instead, the reflection operations that are associated with rotated lattices are the reflections about non-perpendicular axes. We might find some peculiar matrix transformations on the Tables, which are notated by $\sigma_{y(10^\circ)}$, $C_4\sigma_{y(10^\circ)}$, $C_4^2\sigma_{y(10^\circ)}$, and $C_4^3\sigma_{y(10^\circ)}$ denote the reflections about 10° rotated vertical axis, 10° rotated diagonal axis, 10° rotated horizontal axis, and 10° rotated anti-diagonal axis respectively.

$$\mathbf{A} = \begin{bmatrix} 0.985 & -0.174 \\ 0.174 & 0.985 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Table 4.8: Symmetry of 10° tilted square lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 0.985 & -0.174 \\ 0.174 & 0.985 \end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.985 & -0.174 \\ 0.174 & 0.985 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
C_4	$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -0.174 & -0.985 \\ 0.985 & -0.174 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$
C_4^2	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -0.985 & 0.174 \\ -0.174 & -0.985 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
C_4^3	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0.174 & 0.985 \\ -0.985 & 0.174 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$
$\sigma_{y(10^\circ)}$	$\begin{bmatrix} -0.940 & -0.342 \\ -0.342 & 0.940 \end{bmatrix}$	$\begin{bmatrix} -0.985 & -0.174 \\ -0.174 & 0.985 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$
$C_4\sigma_{y(10^\circ)}$	$\begin{bmatrix} 0.342 & -0.940 \\ -0.940 & -0.342 \end{bmatrix}$	$\begin{bmatrix} 0.174 & -0.985 \\ -0.985 & -0.174 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$
$C_4^2\sigma_{y(10^\circ)}$	$\begin{bmatrix} 0.940 & 0.342 \\ 0.342 & -0.940 \end{bmatrix}$	$\begin{bmatrix} 0.985 & 0.174 \\ 0.174 & -0.985 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
$C_4^3\sigma_{y(10^\circ)}$	$\begin{bmatrix} -0.342 & 0.940 \\ 0.940 & 0.342 \end{bmatrix}$	$\begin{bmatrix} -0.174 & 0.985 \\ 0.985 & 0.174 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

$$\mathbf{A} = \begin{bmatrix} 0.985 & -0.174 \\ 0.174 & 0.985 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Table 4.9: Symmetry of 10° tilted square lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 0.811 & -0.174 \\ 1.158 & 0.985 \end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.811 & -0.174 \\ 1.158 & 0.985 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
C_4	$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1.158 & -0.985 \\ 0.811 & -0.174 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 \\ 2 & 1 \end{bmatrix}$
C_4^2	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -0.811 & 0.174 \\ -1.158 & -0.985 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
C_4^3	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1.158 & 0.985 \\ -0.811 & 0.174 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ -2 & -1 \end{bmatrix}$
$\sigma_{y(10^\circ)}$	$\begin{bmatrix} -0.940 & -0.342 \\ -0.342 & 0.940 \end{bmatrix}$	$\begin{bmatrix} -1.158 & -0.174 \\ 0.811 & 0.985 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$
$C_4\sigma_{y(10^\circ)}$	$\begin{bmatrix} 0.342 & -0.940 \\ -0.940 & -0.342 \end{bmatrix}$	$\begin{bmatrix} -0.811 & -0.985 \\ -1.158 & -0.174 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 \\ 0 & 1 \end{bmatrix}$
$C_4^2\sigma_{y(10^\circ)}$	$\begin{bmatrix} 0.940 & 0.342 \\ 0.342 & -0.940 \end{bmatrix}$	$\begin{bmatrix} 1.158 & 0.174 \\ -0.811 & -0.985 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ -2 & -1 \end{bmatrix}$
$C_4^3\sigma_{y(10^\circ)}$	$\begin{bmatrix} -0.342 & 0.940 \\ 0.940 & 0.342 \end{bmatrix}$	$\begin{bmatrix} 0.811 & 0.985 \\ 1.158 & 0.174 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix}$

$$\mathbf{A} = \begin{bmatrix} 0.985 & -0.174 \\ 0.174 & 0.985 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

Table 4.10: Symmetry of 10° tilted rectangular lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 0.985 & -0.347 \\ 0.174 & 1.970 \end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.985 & -0.347 \\ 0.174 & 1.970 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
C_4^2	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -0.985 & 0.347 \\ -0.174 & -1.970 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$

$\sigma_{y(10^\circ)}$	$\begin{bmatrix} -0.940 & -0.342 \\ -0.342 & 0.940 \end{bmatrix}$	$\begin{bmatrix} -0.985 & -0.347 \\ -0.174 & 1.970 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$
$C_4^2\sigma_{y(10^\circ)}$	$\begin{bmatrix} 0.940 & 0.342 \\ 0.342 & -0.940 \end{bmatrix}$	$\begin{bmatrix} 0.985 & 0.347 \\ 0.174 & -1.970 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

$$\mathbf{A} = \begin{bmatrix} 0.985 & -0.174 \\ 0.174 & 0.985 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$$

Table 4.11: Symmetry of 10° tilted rectangular lattice with primitive vectors $\mathbf{A} = \begin{bmatrix} 0.985 & 0.638 \\ 0.174 & 2.143 \end{bmatrix}$.

Symmetry operation	$\mathbf{P}$	$\tilde{\mathbf{A}} = \mathbf{PA}$	$\mathbf{M} = \mathbf{A}^{-1}\tilde{\mathbf{A}}$
E	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.985 & 0.638 \\ 0.174 & 2.143 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
C_4^2	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -0.985 & -0.638 \\ -0.174 & -2.143 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
$\sigma_{y(10^\circ)}$	$\begin{bmatrix} -0.940 & -0.342 \\ -0.342 & 0.940 \end{bmatrix}$	$\begin{bmatrix} -0.985 & -1.332 \\ -0.174 & 1.796 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 \\ 0 & 1 \end{bmatrix}$
$C_4^2\sigma_{y(10^\circ)}$	$\begin{bmatrix} 0.940 & 0.342 \\ 0.342 & -0.940 \end{bmatrix}$	$\begin{bmatrix} 0.985 & 1.332 \\ 0.174 & -1.796 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}$

4.3 Alignment of Primitive Vectors

One lattice and another may be identical but in different orientations. In the case of sublattices it is convenient to confirm if two sublattices are identical by reducing its integer matrix into HNF form. In most of the case of primitive vectors, there is no such HNF reduction. For instance the problem arises when one of two identical lattices for example rotated by some degree. In general, if we apply our comparison program that is given in Listing 2.2 will return `False`, though the two lattices are really the same up to a rotation or a reflection.

This is the main reason of our exploration into symmetries of lattices. However, we may try to align the primitive vectors such that the representations have triangular matrix form, and then we shall list the symmetries again after alignment. This implies that we make some standard in listing symmetries and make the choice of primitive vectors as compact as possible. The complete steps are following.

For some arbitrary set of primitive vectors $\mathbf{A} = (\mathbf{a_1}, \dots, \mathbf{a_d})$ where $\mathbf{A} \in \mathbb{R}^{d\times d}$, find some orthonormal new set of vectors $\mathbf{Q} = (\mathbf{q_1}, \dots, \mathbf{q_d})$ where $\mathbf{Q} \in \mathbb{R}^{d\times d}$, such that $\mathbf{A}' = \mathbf{Q}^{-1}\mathbf{A}$ is upper triangular matrix. At this point neither $\mathbf{Q}$ nor $\mathbf{A}'$ are known but we construct them.

Firstly, take $\mathbf{q}_1$ as a unit vector $\mathbf{q}_1 = \mathbf{a}_1/\|\mathbf{a}_1\|$, then the first element of $\mathbf{A}'$ is magnitude of vector $\mathbf{a}_1$, and the rest of elements are zeros, thence $\mathbf{a}'_1 = (\|\mathbf{a}_1\|, 0, \dots, 0)^\intercal$. Secondly, take an orthogonal vector

$$\mathbf{a}^\perp = \mathbf{a}_2 - \left\lfloor \frac{\langle \mathbf{a}_1, \mathbf{a}_2\rangle}{\langle \mathbf{a}_1, \mathbf{a}_1\rangle} \right\rceil \mathbf{a}_1 \qquad \mathbf{q}_2 = \frac{\mathbf{a}^\perp}{\|\mathbf{a}^\perp\|}$$

this is by construction orthogonal to $\mathbf{q}_1$. Note that the coefficient of $\mathbf{a}_1$ is rounded to nearest integer in order to maintain this operation as an elementary column operation. There are two nonzero elements of $\mathbf{a}_2$ which are the projection of $\mathbf{a}_2$ into $\mathbf{q}_1$ and $\mathbf{q}_2$, then as a result $\mathbf{a}'_2 = (\langle \mathbf{a}_2, \mathbf{q}_1\rangle, \langle \mathbf{a}_2, \mathbf{q}_2\rangle, 0, \dots, 0)^\intercal$. Then, take orthogonal vector toward $\mathbf{q}_1$ and $\mathbf{q}_2$ by operation

$$\mathbf{a}^{\perp\!\!\perp} = \mathbf{a}_3 - \left\lfloor \frac{\langle \mathbf{a}_3, \mathbf{a}_1\rangle}{\langle \mathbf{a}_1, \mathbf{a}_1\rangle} \right\rceil \mathbf{a}_1 - \left\lfloor \frac{\langle \mathbf{a}_3, \mathbf{a}^\perp\rangle}{\langle \mathbf{a}^\perp, \mathbf{a}^\perp\rangle} \right\rceil \mathbf{a}^\perp \qquad \mathbf{q}_3 = \frac{\mathbf{a}^{\perp\!\!\perp}}{\|\mathbf{a}^{\perp\!\!\perp}\|}.$$

We obtain three nonzero entries of $\mathbf{a}'_3 = (\langle \mathbf{a}_3, \mathbf{q}_1\rangle, \langle \mathbf{a}_3, \mathbf{q}_2\rangle, \langle \mathbf{a}_3, \mathbf{q}_3\rangle, 0, \dots, 0)^\intercal$. By repeating these steps d-times, we obtain matrix $\mathbf{A}'$ in the following form.

$$\mathbf{A}' = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{q}_1\rangle & \langle \mathbf{a}_2, \mathbf{q}_1\rangle & \dots & \langle \mathbf{a}_d, \mathbf{q}_1\rangle \\ & \langle \mathbf{a}_2, \mathbf{q}_2\rangle & \dots & \langle \mathbf{a}_d, \mathbf{q}_2\rangle \\ & & \ddots & \vdots \\ & & & \langle \mathbf{a}_d, \mathbf{q}_d\rangle \end{bmatrix}$$

Following the steps above, we can easily construct the code to align primitive vectors (see Listing 4.1). The main method is called `aligned`, also three auxiliary static methods `_proj`, `_dot`, and `_norm` are for projection, dot product, and norm calculation consecutively. These methods belong to class `Lattice`.

Listing 4.1: Aligning primitive vectors

```
#Aligning primitive vectors
  def aligned(self):
    Q = np.matrix(np.zeros((self.d, self.d)))  #Create dxd zero matrix
    for i in range(self.d):
      Q[:,i] = self.A[:,i]     #Reduce each column from zero to d
      for j in range(i):
        Q[:,i] -= self._proj(self.A[:,j],Q[:,i])
      Q[:,i]/=self._norm(Q[:,i])
    self.Aa = np.linalg.inv(Q)*self.A

  @staticmethod
  def _proj(u,v):   #Projection of v to u
    return Lattice._dot(u,v)/Lattice._dot(u,u)*u

  @staticmethod
  def _dot(v,u):    #Dot product of v and u
    return sum([ev*eu for ev,eu in zip(u,v)])[0,0]

  @staticmethod
  def _norm(v):     #Find the norm of v
    return np.sqrt(sum([e*e for e in v])[0,0])
```

Applying this code to the rotated square lattices from Tables 4.8 and 4.9, we obtain the aligned primitive vectors identical to the untransformed one. The same is true for applying the given code to rectangular lattices from Tables 4.10 and 4.11.

Chapter 5

LLL Algorithm

5.1 LLL Primitive Vectors Reduction

We have been trying to find the shortest primitive vectors in order to get the unit cell as compact as possible. There is no best algorithm yet known to find the shortest vector in lattice, but there is a very famous algorithm called LLL (Lenstra Lenstra Lovász) that is able to solve SVP (shortest vector problem). The LLL algorithm does not necessarily find the shortest vector in lattice, but it finds $\gamma \cdot \lambda_1$ with approximation factor γ for d-dimension lattice and λ_1 as shortest vector.

The LLL algorithm is used to get an approximation of SVP in arbitrary high dimension and this approximation is sufficient in many applications. It is widely used in many computer areas such as cryptanalysis of public-key encryption, RSA, MIMO detection algorithm, and so forth. Our main goal here is applying LLL to solve our lattice problem which are 2-d and 3-d problems, however the code will be generally written for any dimension.

The main idea of primitive vector reduction is the following, for a lattice $\mathcal{L}$ with primitive vectors $\mathbf{A}$ change $\mathbf{A}$ into a shorter primitive vectors $\tilde{\mathbf{A}}$ such that $\mathcal{L}$ remains the same. We have known that we can attain other sets of primitive vectors by applying unimodular column operations as defined in Definition 2. Before going to d-dimensions SVP, let us consider the case of 2-dimensions to grasp the main idea.

5.2 2-d Primitive Vectors Reduction

Primitive vector reduction of 2-d lattice is easy to understand yet plays a pivotal role in the LLL algorithm. For a given a set of primitive vectors $\{\mathbf{a}_1, \mathbf{a}_2\}$ we shall consider its reduction. The intuitive approach to solve SVP here is firstly to find the shortest vector, for example $\mathbf{a}_1$ then subtract $\mathbf{a}_2$ from z integer multiple of $\mathbf{a}_1$. Thus we get a new vector $\tilde{\mathbf{a}}_2 = \mathbf{a}_2 - z\mathbf{a}_1$. Choose z such that $\tilde{\mathbf{a}}_2$ will be as short as possible. In order to find z, take coefficient $c = \frac{\langle \mathbf{a}_1, \mathbf{a}_2 \rangle}{\langle \mathbf{a}_1, \mathbf{a}_1 \rangle}$ which is the orthogonal projection of $\mathbf{a}_2$ to $\mathbf{a}_1$ and round it to the nearest integer. Repeat this process until the primitive vectors can no longer be reduced.

Figure 5.1 shows how to find vector $\mathbf{a}_2^*$ that is orthogonal to $\mathbf{a}_1$, also in fact $\mathbf{a}_2^*$ has the shortest distance from $\mathbf{a}_2$ to span$(\mathbf{a}_1)$. However, coefficient c is not always integer hence this transformation is not unimodular, therefore we need to round c to the nearest integer. The new vector $\tilde{\mathbf{a}}_2 = \mathbf{a}_2 - \lfloor c \rceil \mathbf{a}_1$ is not always the same as $\mathbf{a}_2^*$, that is the reason why $\tilde{\mathbf{a}}_2$ is said to be almost orthogonal to $\mathbf{a}_1$. Notice that we have applied this in Section 4.3.

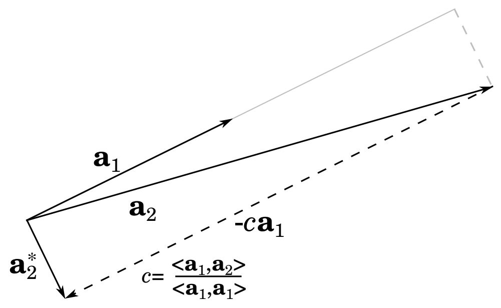

$\mathbf{a}_1$
$\mathbf{a}_2$
$-c\mathbf{a}_1$
$\mathbf{a}_2^*$
$c = \frac{<\mathbf{a}_1,\mathbf{a}_2>}{<\mathbf{a}_1,\mathbf{a}_1>}$

Figure 5.1: Find the shortest orthogonal vector $\mathbf{a}_2^*$, by subtracting $\mathbf{a}_2$ to the u times of $\mathbf{a}_1$. Also notice that the determinant of lattice, $\det \mathcal{L}(\mathbf{A}) = \|\mathbf{a}_1\| \, \|\mathbf{a}_2^*\|$.

The formal definition of primitive vector reduction is shown in Definition 3. By this definition, primitive vectors are sorted in such a way that the first vector is the shortest one. Moreover, the orthogonal projection coefficient is smaller than $\frac{1}{2}$.

Definition 3 *Reduced basis of rank 2*

A set of primitive vectors $\{\mathbf{a}_1, \mathbf{a}_2\}$ is reduced if and only if the norm of $\mathbf{a}_1$ is smaller or equal to the norm of $\mathbf{a}_2$ and the absolute value of orthogonal coefficient $c = \frac{\langle \mathbf{a}_1, \mathbf{a}_2 \rangle}{\langle \mathbf{a}_1, \mathbf{a}_1 \rangle}$ is less than or equal $\frac{1}{2}$.

$$\|\mathbf{a}_1\| \leq \|\mathbf{a}_2\| \quad \textit{and} \quad \frac{|\mathbf{a}_1 \cdot \mathbf{a}_2|}{\mathbf{a}_1 \cdot \mathbf{a}_1} \leq \frac{1}{2} \tag{5.1}$$

In order to have a better understanding of Definition 3 let us observe Figure 5.2. For an arbitrary $\mathbf{a}_1$, the set of primitive vectors is reduced if the projection of $\mathbf{a}_2$ to $\mathbf{a}_1$ is less than or equal to half of $\mathbf{a}_1$, thus $\mathbf{a}_2$ must lay within the grey area. As a result for a reduced primitive vector we have $\|\mathbf{a}_2^*\| \leq \frac{\sqrt{3}}{2} \|\mathbf{a}_1\|$. At this point we can easily come up with upper bound of the shortest vector.

A reduced set of primitive vectors $\{\mathbf{a}_1, \mathbf{a}_2\}$ generates lattice $\mathcal{L}$. By using orthogonal properties and definition of reduced primitive vectors we have:

$$\begin{aligned}
\mathbf{a}_2 &= \mathbf{a}_2^* + c\mathbf{a}_1 \\
\|\mathbf{a}_2\|^2 &= \|\mathbf{a}_2^*\| + c^2 \|\mathbf{a}_1\|^2 \\
\|\mathbf{a}_2^*\|^2 &= \|\mathbf{a}_2\|^2 - c^2 \|\mathbf{a}_1\|^2 \geq \|\mathbf{a}_1\|^2 - c^2 \|\mathbf{a}_1\|^2 \geq \|\mathbf{a}_1\|^2 - \frac{1}{4} \|\mathbf{a}_1\|^2 \\
\|\mathbf{a}_2^*\| &\geq \frac{\sqrt{3}}{2} \|\mathbf{a}_1\| \\
\|\mathbf{a}_2^*\| \, \|\mathbf{a}_1\| &\geq \frac{\sqrt{3}}{2} \|\mathbf{a}_1\|^2 \\
\sqrt{\frac{2}{\sqrt{3}} \det \mathcal{L}} &\geq \|\mathbf{a}_1\| \, .
\end{aligned} \tag{5.2}$$

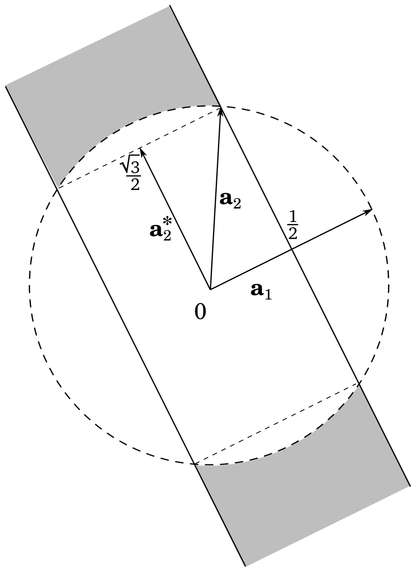


Figure 5.2: The set of primitive vectors is reduced if $\mathbf{a}_2$ lies in the grey area. Thence for a reduced primitive vectors we have $\|\mathbf{a}_2^*\| \le \frac{\sqrt{3}}{2}\|\mathbf{a}_1\|$.

It gives us an upper bound for 2-*d* lattice for the norm of the shortest vector $\|\mathbf{a}_1\|$ which is $\sqrt{\frac{2}{\sqrt{3}}\det\mathcal{L}}$. Now we can construct a code for 2-*d* primitive vector reduction. Firstly, we need to construct a module containing basic vector and matrix operations so it will be easy to re-use or update later. The module is called "`matvecop`" under filename `matvecop.py`, which is shown in Listing 5.1. In module `matvecop`, both functions `norm` and `swap` take an instance of `numpy.matrix` as input.

Listing 5.1: Basic vector matrix operations (`matvecop.py`)

```python
import numpy as np

# dot product of two vectors
def dot(v,u):
  assert(isinstance(u,np.matrix) and isinstance(v,np.matrix)),"both must matrix object"
  return sum([ev*eu for ev,eu in zip(u,v)])[0,0]

# return coefficient of projection v2 to v1
def projc(v1,v2):
  return dot(v2,v1)/dot(v1,v1)

# swap two columns in a matrix
def swap(M,i,j):
  idx = range(M.shape[1])
  idx[i], idx[j] = idx[j], idx[i]
  M[:] = M[:,idx]
```

The main function of 2-*d* primitive vector reduction (see Listing 5.2) is called `LLL2d`. Note that we need to import our previous module `matvecop`. The main idea of function `LLL2d` is to always put the shorter vector in the first column, reduce the second column and then swap. Keep repeating the same process until the second column can not be reduced, then we will have swapped result by the end.

Listing 5.2: Primitive vectors reduction for rank 2

```
import numpy as np
from matvecop import *

def LLL2d(A):      #Reduce 2x2 integer matrix
  assert(A.shape == (2,2) and np.linalg.det(A)), "Must be 2D non-singular matrix!"
  if np.linalg.norm(A[:,0]) > np.linalg.norm(A[:,1]): swap(A,0,1) #Shorter vector at first
  n, eps = 0, 1e-10
 #Keep reducing until fulfill the definition of LLL-reduced
  while np.linalg.norm(A[:,0])<np.linalg.norm(A[:,1]) or abs(projc(A[:,0],A[:,1]))>.5+eps:
    A[:,1] -= A[:,0]*np.around(projc(A[:,0],A[:,1]))  #Almost reduced
    swap(A,0,1)                                     #Swap after reduction
    n += 1
  swap(A,0,1)                       #The result is swapped
  print "Iteration %i times"%n      #Print how many reduction has performed
  return A
```

An example of the code execution is given below. Given in example matrix $\mathbf{A}$ has determinant 1, thence upper bound of shortest vector norm will be $\sqrt{\frac{2}{\sqrt{3}}\det\mathbf{M}} \approx 1.07$. The reduced vectors produce the correct result, since $\|\mathbf{a}_1\| = 1.0 \leq 1.07$.

```
>> A = np.matrix([[3., 8.],[2.,5.]])
>> LLL2d(A)
>> Iteration 3 times
   matrix([[1.00  0.00]
           [0.00 -1.00]])
```

5.3 n-d Primitive Vectors Reduction

We have been working on primitive vectors reduction on 2-d lattices previously, the same manner also applied to the n-d case. Roughly speaking, LLL performs successive orthogonal projections, if needed it also swaps two consecutive vectors in order to get a reduced or almost orthogonal primitive vectors.

In the same manner as the 2-d case, we find firstly the orthogonal projection for each successive vectors. For a given set of primitive vectors $\{\mathbf{a}_1, \ldots, \mathbf{a}_d\}$, that are linearly independent and span lattice $\mathcal{L}$, we find the orthogonal set of primitive vectors $\{\mathbf{a}_1^*, \ldots, \mathbf{a}_d^*\}$ that span the same space. This is the so called Gram-Schmidt orthogonalization process. The Gram-Schmidt orthogonalization process defines

$$\begin{aligned} \mathbf{a}_1^* &= \mathbf{a}_1 \\ \mathbf{a}_k^* &= \mathbf{a}_k - \sum_{i=1}^{k-1} \frac{\mathbf{a}_k \cdot \mathbf{a}_i^*}{\mathbf{a}_i^* \cdot \mathbf{a}_i^*} \cdot \mathbf{a}_i^* \quad \text{for} \quad k \geq 2 \end{aligned} \tag{5.3}$$

where the first vector remain the same and the rest are orthogonalized toward all preceding vectors. This orthogonalization process is basically a transformation of primitive vectors, that we can represent with matrix factorization form. Define $\mathbf{A} = \{\mathbf{a}_1, \cdots, \mathbf{a}_d\}$, $\mathbf{Q} = \{\mathbf{a}_1^*, \cdots, \mathbf{a}_d^*\}$, and introduce the transformation matrix $\mathbf{R}$, with $\mathbf{r}_{kk} = 1$ and $\mathbf{r}_{ik} = \frac{\mathbf{a}_k \cdot \mathbf{a}_i^*}{\mathbf{a}_i^* \cdot \mathbf{a}_i^*}$ so that we have

$$\mathbf{a}_k = \sum_{i=1}^{k} r_{ik} \mathbf{a}_i^*. \tag{5.4}$$

By means of the matrix factorization $\mathbf{A} = \mathbf{QR}$, where $\mathbf{R}$ has the form given below

$$\mathbf{R} = \begin{bmatrix} 1 & & \mathbf{r}_{ik} \\ & \ddots & \\ 0 & & \ddots \\ & & & 1 \end{bmatrix}.$$

Notice that for $\mathbf{A}$ that generate lattice $\mathcal{L}$, $\det \mathcal{L}$ equal to the volume of the parallelepiped spanned by $\{\mathbf{a}_1, \cdots, \mathbf{a}_d\}$ which is equal to the volume of another parallelepiped spanned by $\{\mathbf{a}_1^*, \cdots, \mathbf{a}_d^*\}$. Recall Figure 5.1, it is obvious that its volume is equal to the product of its edges. One concludes that $\det \mathcal{L} = \|\mathbf{a}_1^*\| \cdots \|\mathbf{a}_d^*\| = \prod_{i=1}^{d} \|\mathbf{a}_i^*\|$.

Now we have to make a general definition of reduced primitive vectors. We define *c-reduced* primitive vectors in Definition 4.

Definition 4 *A set of primitive vectors $\mathbf{A} = \{\mathbf{a}_1, \cdots, \mathbf{a}_d\}$ is said to be c-reduced, if and only if its orthogonal set of primitive vectors $\mathbf{Q} = \{\mathbf{a}_1^*, \cdots, \mathbf{a}_d^*\}$ that obtained by Gram-Schmidt orthogonalization process fulfills the following inequality*

$$\|\mathbf{a}_{i+1}^*\|^2 \geq \frac{\|\mathbf{a}_i^*\|^2}{c} \quad i = 1, \cdots, d-1. \tag{5.5}$$

Where a small value of c may be interpreted as good reduction. We may expect that at least $\|\mathbf{a}_{i+1}^*\| = \|\mathbf{a}_i^*\|$, however, not every set of primitive vectors are 1-reducible, but each of them are $\frac{4}{3}$-reducible. The number $\frac{4}{3}$ comes from property that is shown in Figure 5.1.

We want to transform primitive vectors $\mathbf{A}$ as closely as possible to the orthogonal primitive vectors $\mathbf{Q}$. We can also evaluate the notion of distance between orthogonal vectors $\mathbf{Q}$ and almost orthogonal vectors $\mathbf{A}$ in lattice $\mathcal{L}$ that is generated by c-reduced primitive vectors $\mathbf{A} = \{\mathbf{a}_1, \cdots, \mathbf{a}_d\}$, with $c \geq \frac{4}{3}$, and its orthogonal primitive vectors $\mathbf{Q} = \{\mathbf{a}_1^*, \cdots, \mathbf{a}_d^*\}$, we obtain that

$$\begin{aligned} \mathbf{a}_1^* &= \mathbf{a}_1 \\ \|\mathbf{a}_2^*\|^2 &\geq c^{-1} \|\mathbf{a}_1^*\|^2 = c^{-1} \|\mathbf{a}_1\|^2 \\ \|\mathbf{a}_3^*\|^2 &\geq c^{-2} \|\mathbf{a}_1\|^2 \\ &\vdots \\ \|\mathbf{a}_i^*\|^2 &\geq c^{-i+1} \|\mathbf{a}_1\|^2, \end{aligned}$$

since $\|\mathbf{a}_1\| \leq \|\mathbf{a}_2\| \leq \cdots \leq \|\mathbf{a}_d\|$ then

$$\|\mathbf{a}_i^*\|^2 \geq c^{-i+1}\|\mathbf{a}_i\|^2 \text{ or } \|\mathbf{a}_i\|^2 \leq c^{i-1}\|\mathbf{a}_i^*\|^2. \tag{5.6}$$

Then we take a product of both sides for $i = 1, \cdots, d$

$$\begin{aligned}\prod_{i=1}^{d}\|\mathbf{a}_i\|^2 &\leq c^{(\Sigma_{i=1}^{d-1} i)}\prod_{i=1}^{d}\|\mathbf{a}_i^*\|^2\\ \prod_{i=1}^{d}\|\mathbf{a}_i\|^2 &\leq c^{d(d-1)/2}\det\mathcal{L}^2\\ \prod_{i=1}^{d}\|\mathbf{a}_i\| &\leq c^{d(d-1)/4}\det\mathcal{L}.\end{aligned} \tag{5.7}$$

We can conclude that $c^{d(d-1)/4}$ measures the furthest of reduced primitive vectors from orthogonality. Now we can calculate upper bound of the norm of shortest vectors effortlessly. By using Equation 5.7 and inequality $\|\mathbf{a}_1\| \leq \|\mathbf{a}_2\| \leq \cdots \leq \|\mathbf{a}_d\|$ we get

$$\begin{aligned}\prod_{i=1}^{d}\|\mathbf{a}_1\| &\leq c^{d(d-1)/4}\det\mathcal{L}\\ \|\mathbf{a}_1\| &\leq c^{(d-1)/4}\det\mathcal{L}^{\frac{1}{d}}.\end{aligned} \tag{5.8}$$

In addition, we can demonstrate the bound of c-reduced set of primitive vectors by using Equation 5.6. Let λ is the shortest vector, $\mathbf{x} \in \mathcal{L} - \mathbf{0}$, and let i be the minimal number such that $\mathbf{x} \in \mathcal{L}_i$ where $\mathcal{L}_i \subset \mathcal{L}$ and $\mathcal{L}_i$ is generated by $\{\mathbf{a}_1, \cdots, \mathbf{a}_i\}$. Notice that $\|\mathbf{x}\| \leq \|\mathbf{a}_i\|$ then it follows

$$\lambda^2 \geq \|\mathbf{x}\|^2 \geq \|\mathbf{a}_i^*\|^2 \geq \frac{\|\mathbf{a}^*\|^2}{c^{i-1}} \geq \frac{\|\mathbf{a}_1\|^2}{c^{d-1}}.$$

We can conclude that

$$\|\mathbf{a}_1\| \leq c^{(d-1)/2}\lambda. \tag{5.9}$$

Notice, that this upper bound error $c^{(d-1)/2}$ increases exponentially with dimension as illustrated by Figure 5.3.

Now let us consider the LLL implementation. Firstly, it is useful to make a function to check whether a set of primitive vectors is reduced or not, then we can easily check our result. The implementation of this function is produced using Definition 4 and is given in Listing 5.3. The main function is called `is_LLL_reduced` with parameter reduction `c` with default value 4/3, and an auxiliary function `GramSchmidt` returns orthogonalized primitive vectors and its transformation matrix.

Listing 5.3: Check if a matrix is LLL reduced

```
import numpy as np
from matvecop import *

```

```
#Check if A is LLL reduced with c parameter
def is_LLL_reduced(A,c=4./3.):
  Q,R = GramSchmidt(A)
  for i in range(A.shape[1]-1):  #Compare all columns i to i+1
    if norm(Q[:,i])**2 > c*norm(Q[:,i+1])**2:  #Fail once one is fail
      return False
  return True

#Return orthogonalized matrix Q, with A=QR (Gram-Schmidt orthogonalization)
def GramSchmidt(A):
  Q = A.copy()     #Copy so it does not overwritten
  R = np.matrix(np.eye(Q.shape[0],Q.shape[1]))
  for j in range(1,Q.shape[1]):
    for i in range(j):
      R[i,j] = dot(Q[:,j],Q[:,i])/dot(Q[:,i],Q[:,i])
      Q[:,j] -= Q[:,i]*R[i,j]
  return (Q,R)
```

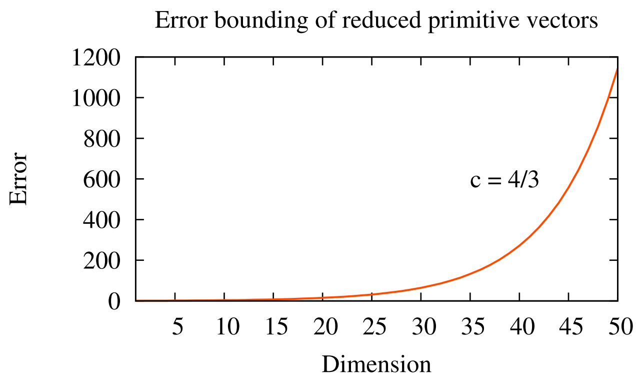


Figure 5.3: The value of bounding error $c^{(d-1)/2}$ increase exponentially to dimension

Now we have every instrument we need for constructing the LLL. The LLL reduction code is shown in Listing 5.4 with main function `LLL` and an auxiliary function `reduce`. It is clear that firstly we reduce the primitive vectors $\mathbf{A}$, then we check if the reduced $\tilde{\mathbf{A}}$ fulfils Definition 4, if not then swap the necessary columns and start over again with the new $\tilde{\mathbf{A}}$.

Listing 5.4: LLL reduction

```
#Reduce A into c-reduced with default value c = 4/3
def LLL(A, c=4./3.):
  Q,R = GramSchmidt(A)
  eps = 1e-10   #Numerical error tolerance
  for i in range(A.shape[1]): #Reduce all columns firstly
    reduce(i,A,R)
  for i in range(A.shape[1]-1): #Iterate until all columns are reduced
    if c*dot(Q[:,i+1],Q[:,i+1]) > dot(Q[:,i],Q[:,i]) :
      swap(A,i,i+1)
      return LLL(A)
  return A

#Reduce A and R with elementary column operations
def reduce(i, A, R):
```

```
for j in range(i-1,-1,-1):
  A[:,i] -= round(R[j,i])*A[:,j]
  R[:,i] -= round(R[j,i])*R[:,j]
```

Let us run some examples to make sure that our code produces the correct results. We assign two very well-known primitive cells namely BCC (body center cubic) and FCC (face center cubic) cells on variables `BCC` and `FCC` respectively. We initiate random shapes of primitive cells which are not the reduced ones. Then, after applying the `LLL` method to the matrices, the matrices are reduced to the correct result which are the most compact ones.

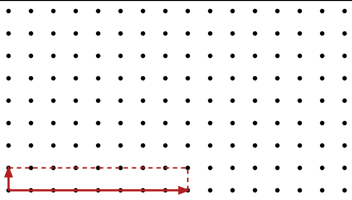
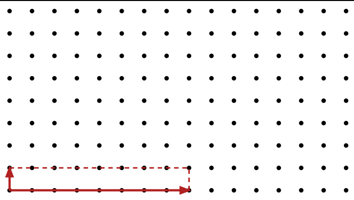
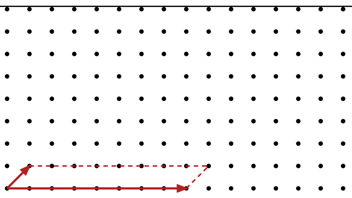
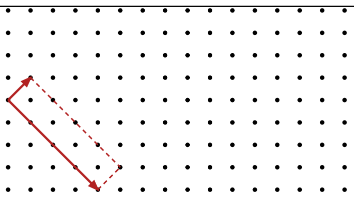
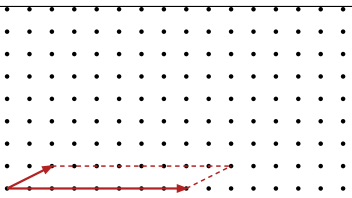
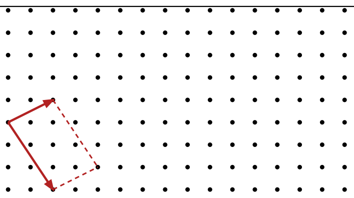
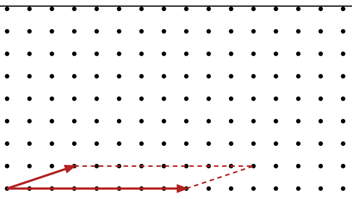
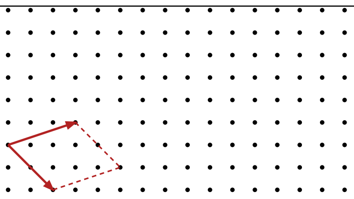
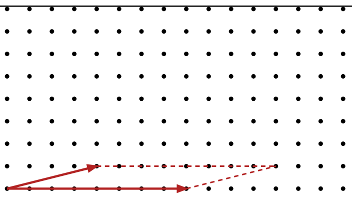
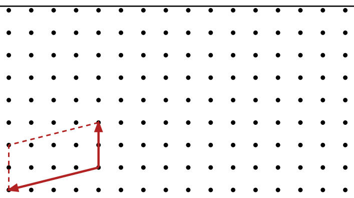
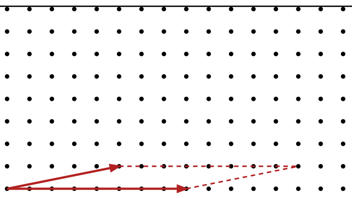
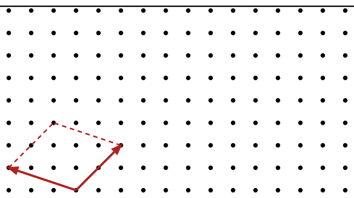
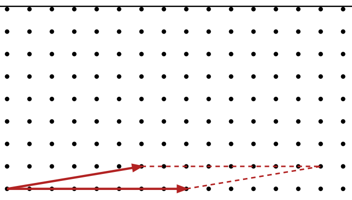
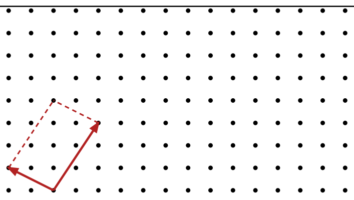
```
>> BCC = np.matrix([[0., 1., -2.5],[1., -6., 18.5],[2., -8., 26.5]])
>> FCC = np.matrix([[ 4., -39., -312.],[1., 10., 81.],[-5., -49., -391.]])
>> FCC
>> matrix([[  -4.,  -39., -312.],
          [   1.,   10.,   81.],
          [  -5.,  -49., -391.]])
>> BCC
>> matrix([[  0. ,   1. ,  -2.5],
          [  1. ,  -6. ,  18.5],
          [  2. ,  -8. ,  26.5]])
>> is_LLL_reduced(A), is_LLL_reduced(BCC)
>> (False, False)
>> LLL(BCC), LLL(FCC)
>> BCC
>> matrix([[ 0.5, -0.5,  0.5],
          [-0.5,  0.5,  0.5],
          [ 0.5,  0.5, -0.5]])
>> FCC
>> matrix([[ 1.,  1.,  0.],
          [ 0.,  1.,  1.],
          [ 1.,  0.,  1.]])
>> is_LLL_reduced(BCC), is_LLL_reduced(FCC)
>> (True, True)
```

By applying method for listing the possible sets of primitive vectors from Section 3.2, we list the possible HNF matrices then reduce each of them by LLL reduction. Table 5.1 shows all possible sublattices that contain 8 lattice points. There are 15 sublattices listed, the matrices are written on the left hand side and the illustration of matrices are drawn on the right side, with right-hand arrows pointing to the form after LLL reduction. We can easily notice from the plots that LLL reduction always transform vectors into more compact form.

From the list of LLL reduced primitive vectors, notice that some of them seem to be equivalent up to the symmetry. We can try to reduce again the list by applying symmetry operations of square lattice from Table 4.1, then we eliminate the equivalent primitive vectors. Since we are working with four sides polygon, we apply symmetry transformations of point group C_{4v} to our list. The results are shown in Table 5.2.

Eventually, from 15 candidates of possible sets of primitive vectors, we obtain 7 unique sets by applying LLL reduction and eliminating the equivalent ones by symmetry operations. The final candidates are plotted in one picture (see Figure 5.4).

Table 5.1: The possible HNF matrices are listed for supercells that contain 8 lattice points. The right hand sides illustrates the left hand side matrices, the right hand arrows imply the form after LLL reduction

Matrix	Illustration		
$\begin{bmatrix} 8 & 0 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 8 \\ 1 & 0 \end{bmatrix}$		$\rightarrow$	
$\begin{bmatrix} 8 & 1 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 \\ 1 & -4 \end{bmatrix}$		$\rightarrow$	
$\begin{bmatrix} 8 & 2 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 2 \\ 1 & -3 \end{bmatrix}$		$\rightarrow$	
$\begin{bmatrix} 8 & 3 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 2 \\ 1 & -2 \end{bmatrix}$		$\rightarrow$	
$\begin{bmatrix} 8 & 4 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & -4 \\ 2 & -1 \end{bmatrix}$		$\rightarrow$	
$\begin{bmatrix} 8 & 5 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} -3 & 2 \\ 1 & 2 \end{bmatrix}$		$\rightarrow$	
$\begin{bmatrix} 8 & 6 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} -2 & 2 \\ 1 & 3 \end{bmatrix}$		$\rightarrow$	

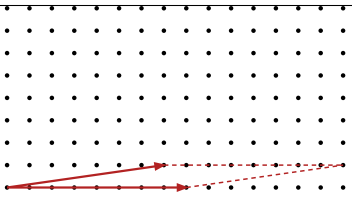
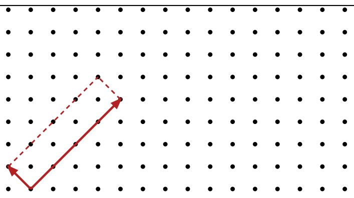
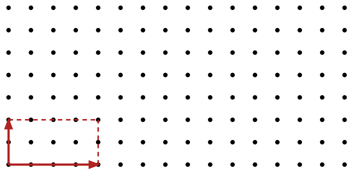
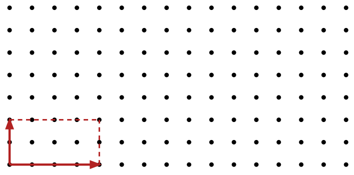
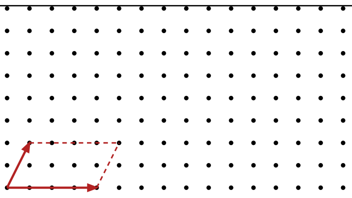
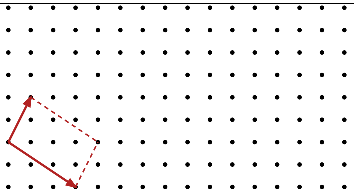
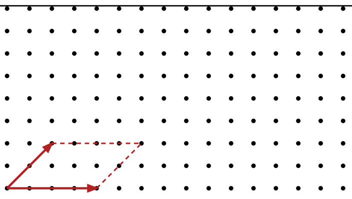
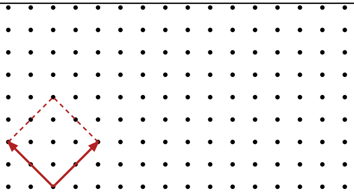
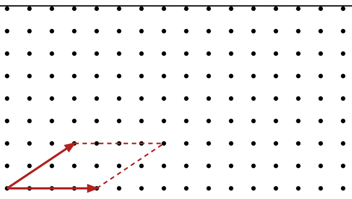
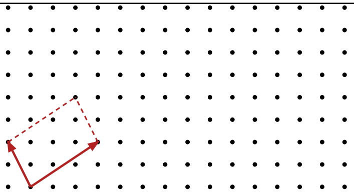
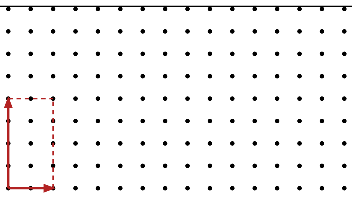
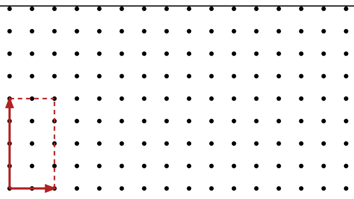
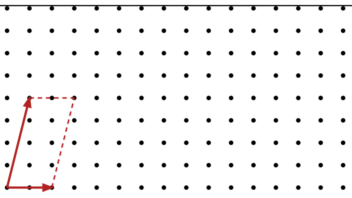
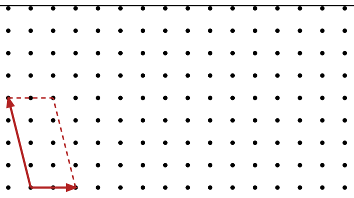
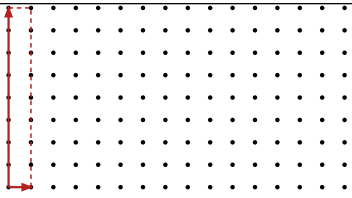
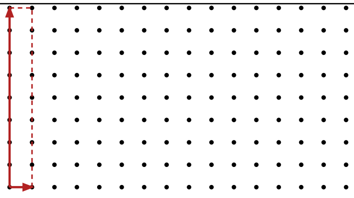
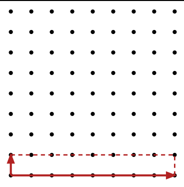
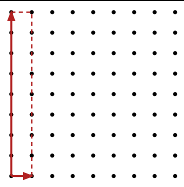
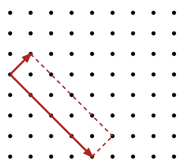
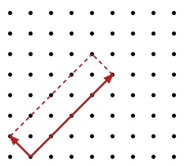
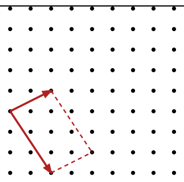
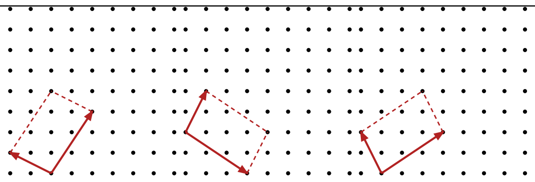
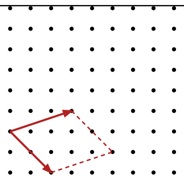
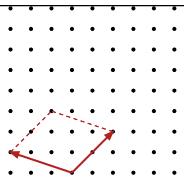
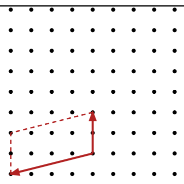
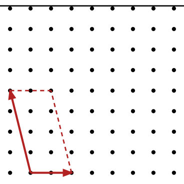
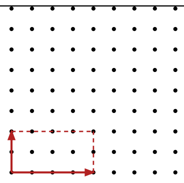
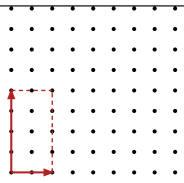
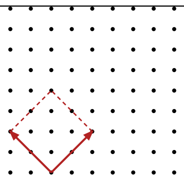
$\begin{bmatrix} 8 & 7 \\ 0 & 1 \end{bmatrix} \to \begin{bmatrix} -1 & 4 \\ 1 & 4 \end{bmatrix}$		$\to$	
$\begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix} \to \begin{bmatrix} 0 & 4 \\ 2 & 0 \end{bmatrix}$		$\to$	
$\begin{bmatrix} 4 & 1 \\ 0 & 2 \end{bmatrix} \to \begin{bmatrix} 1 & 3 \\ 2 & -2 \end{bmatrix}$		$\to$	
$\begin{bmatrix} 4 & 2 \\ 0 & 2 \end{bmatrix} \to \begin{bmatrix} -2 & 2 \\ 2 & 2 \end{bmatrix}$		$\to$	
$\begin{bmatrix} 4 & 3 \\ 0 & 2 \end{bmatrix} \to \begin{bmatrix} -1 & 3 \\ 2 & 2 \end{bmatrix}$		$\to$	
$\begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix} \to \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}$		$\to$	
$\begin{bmatrix} 2 & 1 \\ 0 & 4 \end{bmatrix} \to \begin{bmatrix} 2 & -1 \\ 0 & 4 \end{bmatrix}$		$\to$	
$\begin{bmatrix} 1 & 0 \\ 0 & 8 \end{bmatrix} \to \begin{bmatrix} 1 & 0 \\ 0 & 8 \end{bmatrix}$		$\to$	

Table 5.2: For each supercell that contains 8 lattice points, the LLL-reduced supercells are listed,its equivalent symmetries are shown on the right side

Primitive Vectors	Equivalent Primitive Vectors by Symmetry Operation	Symmetry Operation
		C_4
		C_4
		σ_y $C_4\sigma_y$ C_4
		C_4
		C_4
		C_4
		

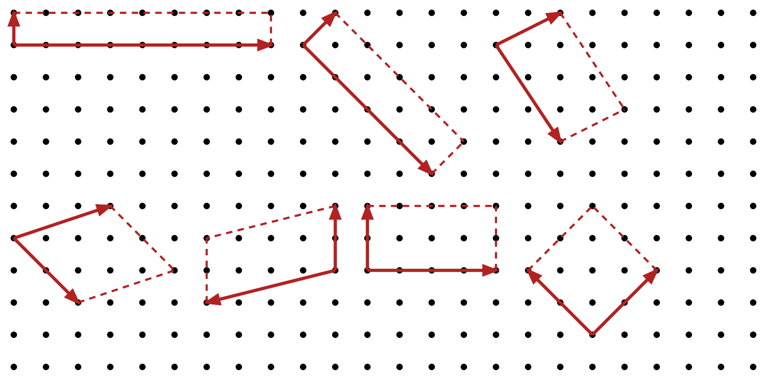


Figure 5.4: The unique primitive vectors of sublattices that have 8 lattice points

Chapter 6

Betts Criteria

6.1 Introduction

Since the discovery of high-temperature superconductivity [4], the calculation of quantum spin systems on a 2-dimensional lattice has been frequently discussed in literature as a topic of highly correlated electron systems. In these calculations, all supercells were based on square shape that tiled the infinite square lattice until [1, 2].

Haan *et al.* [1] performed exact-diagonalization for 2-dimensional spin-$\frac{1}{2}$ antiferromagnetic Heisenberg systems on various supercells (square and non-square) which have number of lattice points $N = 8 - 26$. They introduced an *asymmetry* parameter that defined by $(l_1 - l_2)/(l_1 + l_2)$ where l_1 and l_2 represent the length of vectors that construct supercell. The ground state energy per site and staggered magnetization were obtained by extrapolation of different results from various supercells. Then, they discovered apparent correlation between the asymmetry parameter, ground state energy, and staggered magnetization.

Correspondingly, Betts *et al.* [2, 3] initiated some works in order to find correlations between geometrical and topological properties of supercells and the spin systems calculation of 2-dimensional systems. They introduced some criteria to find the best supercells for performing the exact diagonalization of Hamiltonians of quantum spin models.

In the earlier paper [2], Betts *et al.* performed calculation of $S = \frac{1}{2}\,XY$-model for properties ground state energy per spin, spontaneous energy per spin, magnetization, and spin-spin correlations between first, second, and third nearest neighbor. Then each of the property is graded. These calculations were performed for bipartite and nonbipartite supercells with the number of lattice points $N \leq 26$. They considered three geometrical parameters for roughly guiding to the useful supercell namely point group symmetry (S), *squareness* (σ), and *geometrical imperfection* (J). In addition, they also introduce the numbering of lattice points and the labeling of supercells.

Latter, in the sequence of this first paper [3], Betts *et al.* considered the $S = \frac{1}{2}\,XY$ ferromagnetic model and spin-$\frac{1}{2}$ Heisenberg antiferromagnetic model for estimating the ground energy per site and staggered magnetization. For the case of spin-$\frac{1}{2}$ Heisenberg antiferromagnetic models, some bipartite supercells were considered in order to avoid imposing frustration. Then, they introduced topological imperfections namely *ferromagnetic imperfection* (I_F) for ferromagnetic models. For the antiferromagnetic models, they defined an imperfection of bipartite supercells (I_B) which is basically derived from the ferromagnetic imperfection. They considered supercells with $N = 8 - 32$ and calculated the energy per site and staggered magnetization for every supercells. They demonstrated fitting curves for the scaling of magnetization and ground state energy of different supercells, then showed most of outriders that encountered were mainly topologically imperfect supercells.

Correspondingly, we shall write several programs to classify supercells of two dimensional lattice system based on criteria that are defined by Betts *et al.* in their papers [2, 3] that have been introduced previously.

6.2 Representation of Supercell by Numbering

We have already represented supercells which contain a certain number of lattice points by a set of vectors. This supercell is tiled periodically to create infinite lattice points that is called superlattice. Betts, *et al.* [2] has an alternative way to represent supercells using a certain way of numbering the lattice points, which in turn produces a label for a supercell. An example is shown in Figure 6.1.

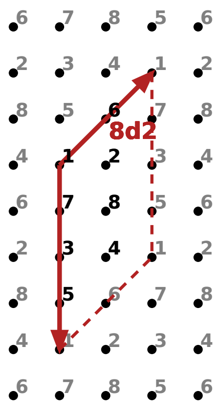


Figure 6.1: Periodically numbered lattice points for supercell $8d2$

The label consists of three components, $N\alpha k$, where N is the number of lattice points in the supercell, $\alpha = h, d, t, q, \ldots$ indicates that the finite lattice is formed of $1, 2, 3, 4, \ldots$ helices and k denotes the number of steps from a point in the first helix to the same point in another helix. Below we will explain the labeling in more details.

For example let us observe the supercell in Figure 6.1; it has $N = 8$, $\alpha = 2$, and $k = 2$. The numbers are periodically repeated from 1 to 8, notice that lattice points that lay on the solid edges are numbered uniquely, and the dotted edges have the same numbers as do the solid ones which is condition of periodic boundary.

On Figure 6.2 we start to number the lattice points from the origins of supercell vectors, we count from left to right under the same helix, then we move upward to reach the next helices. The first helix is denoted by blue color, with numbers 1, 2, 3 and 4 but then it returns to the point number 1 (the origin of the next supercell), thus we must jump to the next point by moving one step upward, then walk to the right again for the next numbers which are 5, 6, 7 and 8. We

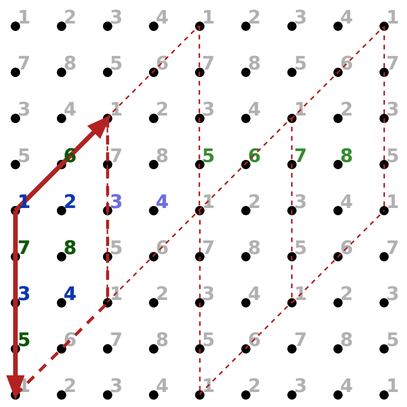


Figure 6.2: Numbering tile 8d2, contains two helices: the first helix denoted by blue with number 1, 2, 3, 4 and the second helix denoted by green with number 5, 6, 7, 8

repeat the same process until all lattice points inside the supercell are numbered. For this case two helices are enough to cover all lattice points. The notation for helix number are derived from tuples name *e.g*, *h* from single **h**elix, *d* from **d**ouble, *t* from **t**riple, and so forth. Summary of this second notation is formulated by Table 6.1.

Table 6.1: Notation of helix number

Helix number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Notation	*h*	*d*	*t*	*q*	*p*	*s*	*t*	*o*	*n*	*c*	*u*	*l*	*r*	*e*

At last, the third notation is the number of steps from one point in a helix to the same point in the helix above, for example see Figure 6.3 for the case of 8d2 sublattice. Since the number of helices is two and the number of lattice points in supercell is 8, then there are four sites per helix. Here we take the modulo value in order to create small number which is $2 = (2 \mod 4)$, then the possible values of k for 8dx are 0, 1, 2 and 3.

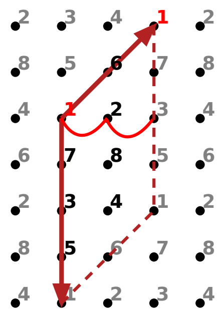


Figure 6.3: The last notation in labeling, there are two steps to move from point 1 in a helix to another point 1 from another helix. Then take the modulo value between number of steps and number of points per helix.

Based on our elaborations above, we present a program in Listing 6.1 which numbers lattice points and labels a supercell. The main function is called `number_latticepoints`, for which we need only the numpy library. For a given two dimensional integer supercell (Sc), the function returns an object with properties 'count' as the number of lattice points in the supercell, 'positions' as the list of positions of lattice points in the supercell (integers), 'numbered' as the number of each lattice points, 'helices' as the number of helices, and 'label' as the label of the supercell.

Basically, function number_latticepoints contains three main steps, firstly it generates all lattice points inside a supercell, secondly it walks through all lattice points in a certain way to number them, and lastly it moves to another helix and walk to the right while counting the steps. Correspondingly, it creates the label from these steps.

Listing 6.1: Numbering lattice points and labeling supercell

```
import numpy as np

eps          = 1e-10
helix_form   = ['h','d','t','q','p','s','t','o','n','c','u','l','r','e'] #helicity label

#Return list of points that numbered by betts
#Sc is 2-d supercell in basis A
def number_latticepoints(Sc) :
  Sc_inv = np.linalg.inv(Sc)
  res    = {} #Result of calculation, an object with some properties
  res['count']     = int(round(abs(np.linalg.det(Sc)))) #Count lattice points in supercell
  res['positions'] = [{}]*res['count']   #Positions [x,y] for each lattice points
  res['numbered']  = [None]*res['count'] #Number for each lattice points
  res['helices']   = 1                   #Number of helix
  res['label']     = None                #Label of supercell
  #Find box which is big enough for supercell
  xmax = int(np.ceil (max(0, Sc[0,0], Sc[0,1], Sc[0,0]+Sc[0,1])))+1
  ymax = int(np.ceil (max(0, Sc[1,0], Sc[1,1], Sc[1,0]+Sc[1,1])))+1
  xmin = int(np.floor(min(0, Sc[0,0], Sc[0,1], Sc[0,0]+Sc[0,1])))-1
  ymin = int(np.floor(min(0, Sc[1,0], Sc[1,1], Sc[1,0]+Sc[1,1])))-1
  ##This part is constructing positions for each lattice points
  idx = 0   #index of positions
  for y in range(ymin, ymax):
    for x in range(xmin, xmax):
      lp    = np.matrix([[float(x)],[float(y)]])   #Lattice point in basis A
      lp_Sc = Sc_inv*lp                            #Lattice point in basis Sc
      if np.all(lp_Sc >= -eps) and np.all(lp_Sc < 1.0-eps): #If lattice point inside Sc
        res['positions'][idx] = [x,y]
        idx += 1
 ##This part is mainly walk through lattice points from left to right, from bottom to top
  num = 0     #Number of lattice point
  pos = [0,0] #Start from the corner of Sc
  while(not np.all(res['numbered'])):  #Loop until all lattice points are numbered
    for dx in range(res['count']):     #Loop all over lattice points in Sc
      sc_pos  = Sc_inv*np.matrix([[pos[0]],[pos[1]]])#Generate a lattice point in basis Sc
      sc_pos -= np.floor(sc_pos)         #Make sure lattice point inside unit cell
      if abs(abs(sc_pos[0])-1.) <= eps : sc_pos[0] = 0.  #All corners are [0,0]
      if abs(abs(sc_pos[1])-1.) <= eps : sc_pos[1] = 0.  #All corners are [0,0]
      new_pos = Sc*sc_pos   #Get new position in basis A (integer points)
      npos    = [int(round(new_pos[0,0])), int(round(new_pos[1,0]))]
      index   = res['positions'].index(npos)   #Find index of position in list
      if not res['numbered'][index] :  #Number lattice points if not yet numbered
        num += 1
        res['numbered'][index] = num
```

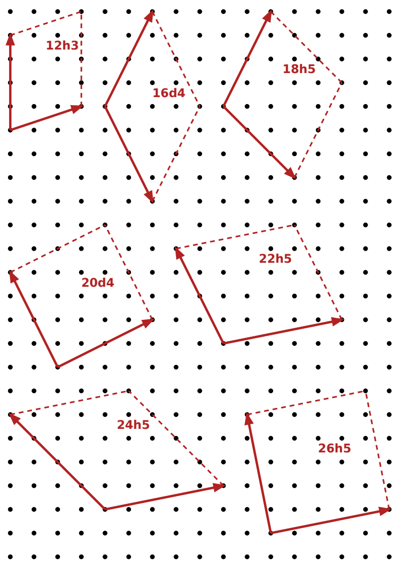
```
   pos[0] += 1 #Move to the next lattice point (walk to the right)
  else :  #If it is already numbered, move helix
   if not np.all(res['numbered']) : #prevent extra helix in the end
    res['helices'] += 1
    pos[1]        += 1  #Walking in upward direction, to the new helix
    break
#Calculate the third part of label, start from the next helix below
pos  = [res['positions'][0][0],res['positions'][0][1]-res['helices']]
dist = 0    #Distance from the same point between two different helix
index= 1000 #initial random integer
while(index!=0):  #Walk until we back to the corner
 pos[0] -= 1     #Keep walking to the right side, then find position and index
 dist   += 1     #Count every step
 sc_pos  = Sc_inv*np.matrix([[pos[0]],[pos[1]]])
 sc_pos -= np.floor(sc_pos)
 if abs(abs(sc_pos[0])-1.)<= eps : sc_pos[0] = 0.
 if abs(abs(sc_pos[1])-1.)<= eps : sc_pos[1] = 0.
 new_pos = Sc*sc_pos
 npos    = [int(round(new_pos[0,0])), int(round(new_pos[1,0]))]
 index   = res['positions'].index(npos)
dist = dist%(res['count']/res['helices']) #Make modulo to number of points/helix
if res['helices']-1 > len(helix_form) : #If number of helix is more than table say:x
 res['label'] = str(res['count'])+'x'+str(dist)
else :
 res['label'] = str(res['count'])+helix_form[res['helices']-1]+str(dist)
return res
```

As an example of program execution, let us run it for the example of the supercell from Figure 6.1.

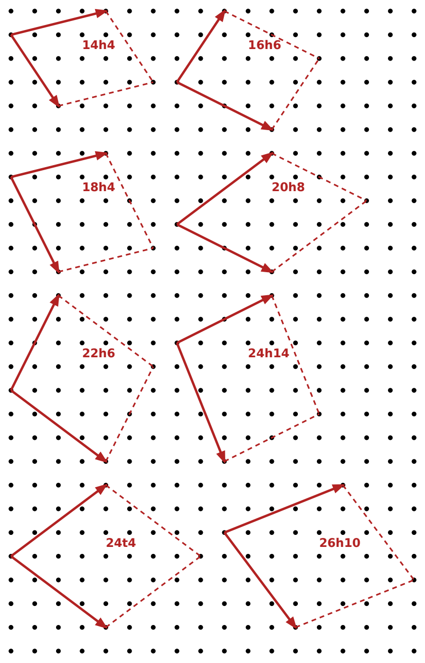
```
>> Sc = np.matrix([[2.,0],[2.,-4]])
>> number_latticepoints(Sc)
  {'count': 8,
   'helices': 2,
   'label': '8d2',
   'numbered': [5, 3, 4, 7, 8, 1, 2, 6],
   'positions': [[0, -3],
                 [0, -2],
                 [1, -2],
                 [0, -1],
                 [1, -1],
                 [0,  0],
                 [1,  0],
                 [1,  1]]}
```

In order to clarify the results from our code, we reproduce some examples of supercell with their labels from paperwork of Betts *et al.* [2], Figure 2, in our Figure 6.4. Initially, we encountered four discrepancies from our labels which are from Figure 6.4a ($16q2$), 6.4b ($24h14$ and $26d5$), and 6.4c ($19h5$). Thence, we apply a symmetry rotation C_4, and figure out that $16q2$ and $26d5$ are equivalent with $16d4$ and $26d5$ consecutively up to the C_4 symmetry. For the case of $24h14$, it is equivalent with $24h10$ up to the $C_4^2\sigma_y$ symmetry. We assume that the labels with the smallest numbers of α and k are considered. In the end we find one discrepancy between label $19h5$ and label $19h4$ from reference.

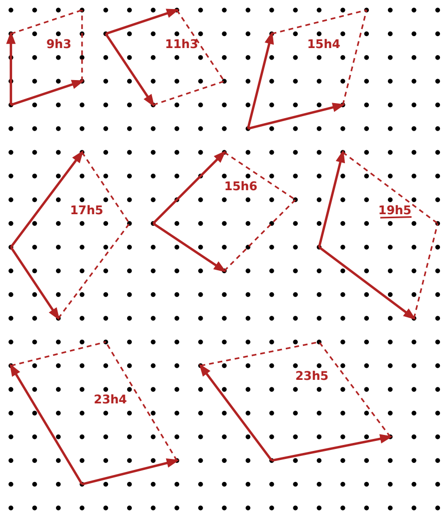
In fact, this labeling is independent from the choice of supercell, thus, for the equivalent supercells they own the same label. As an example, we create one supercell and generate other



(a)



(b)



(c)

Figure 6.4: Reproduced labels from paperwork Betts *et al.* [2] of Figure 2. Some supercells of the square lattice that serve as the good supercells: (a)bipartite lattice, (b) even N non-bipartite lattice (c) odd N lattices. The discrepancies to Figure 2 of [2] are underlined.

equivalent supercells by adding one vector to integer-multiple of another vector. Then each of them must show the same label. Furthermore, if we number the entire lattice points, it is even more obvious that the numbering is completely independent from the choice of the positive vectors of the supercell. In Figure 6.5 we create five equivalent supercells and properly place

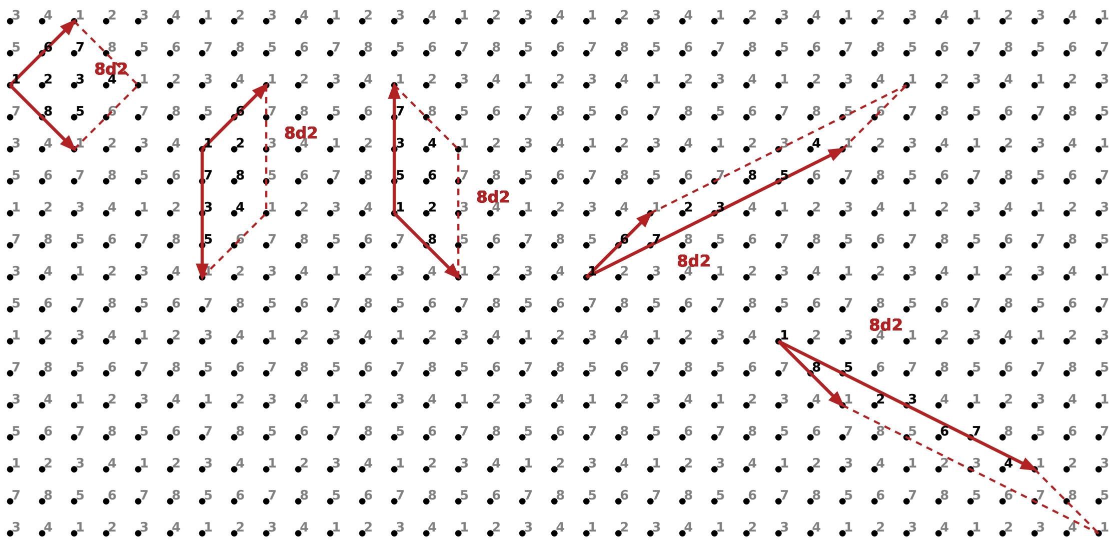


Figure 6.5: All equivalent supercells are properly placed. It is clear that the numbering is independent from the choice of supercell.

every corner of the supercell to the lattice points number 1, then all the numbers of lattice points inside supercells match the number of the whole lattice points and also they show the same label. This is because each supercell spans the same superlattice, hence each point in superlattice is translated to the same place by vectors of supercell hence numbers are preserved.

In addition, let us recall the previous section, where we have generated a list of unique supercells by listing possible HNF matrices, and then reducing them by LLL algorithm and eliminating their symmetries. If we apply numbering to this result (from Figure 5.4), then they must show unique labels, which is demonstrated in Figure 6.6.

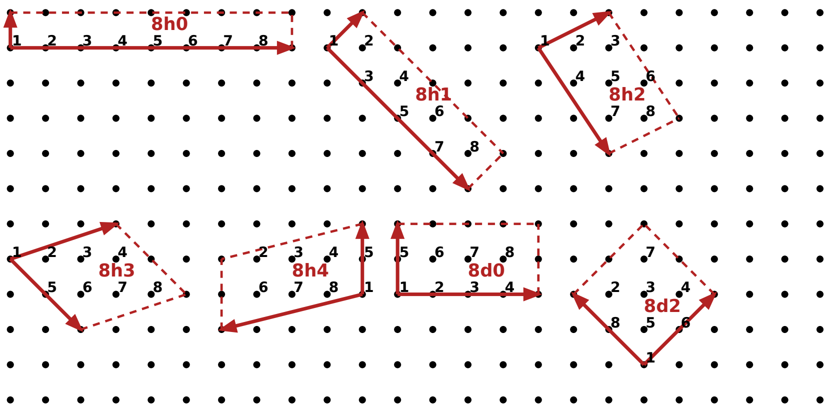


Figure 6.6: Numbered lattice points with its labels for all possible sublattice with 8 lattice points, each of the supercell shows unique label.

6.3 Criteria of Supercell

Squareness is defined as $\sigma = 2l_1l_2/d_1d_2$, where l_1 and l_2 are the lengths of the edges, d_1 and d_2 are the lengths of the diagonals, for instance, see Figure 6.7. This parameter might not be well defined because different supercells that span the same superlattice may have different squareness. For example, the squarenesses of the supercells from Figure 6.5 are tabulated in Table 6.2. The table shows how the value of squareness differs. The shapes of tiles are close to square when the square root of squareness close to 1.0. Note that we will always show the square root of squareness due to it is comparable to the tables that are listed in Tables 1 and 2 of [2].

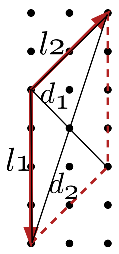


Figure 6.7: Define l_1 and l_2 as length of edges, and d_1 and d_2 are length of diagonals.

We provide a program that returns square root of squareness of a supercell (see Listing 6.2) which is called `ssquareness`. It requires library `numpy` and module `matvecop` from Listing 5.1. We make this program as a sequel from Listing 6.1, and we only need to import module `matvecop`.

Listing 6.2: Betts criteria: square root of squareness $\sqrt{\sigma}$

```
import matvecop as mvo

#Return square root of squareness of B
def ssquareness(B):
  l1 = mvo.norm(B[:,0])              #Length of first edge
  l2 = mvo.norm(B[:,1])              #Length of second edge
  d1 = mvo.norm(B[:,0] + B[:,1])     #Length of non-main diagonal
  d2 = mvo.norm(B[:,0] - B[:,1])     #Length of main diagonal
  return np.sqrt(2.*l1*l2/(d1*d2))
```

Table 6.2: Square root of squareness of supercells in Figure 6.5 are listed from left to right consecutively.

No	1	2	3	4	5
$\sqrt{\sigma}$	1.00	1.12	1.12	0.83	0.83

Notice, that in Table 6.2 only the first supercell has squareness 1.0, which means the supercell is perfectly square. If we reduce the other supercells, they will converge into the form of the first supercell which is the most compact one. It means LLL reduction produce supercells as square as possible. Thus, we define squareness of a supercell as squareness of the LLL-reduced supercell. List of squarenesses of the unique supercell for 8 lattice points in supercell is tabulated in Table 6.3. It shows that three of these supercells are close to square, since Betts *et al.* propose that the most satisfactory supercell are based on tiles for which $0.95 \leq \sigma \leq 1.05$. Notice for the Figure 6.6, that it shows that the supercells with $\sigma < 0.95$ or $\sigma > 1.05$ are the ones with the shape close to rectangular.

Table 6.3: Squareness of unique list of supercells that contain 8 lattice points

Label	$\sqrt{\sigma}$
$8h0$	0.50
$8h1$	0.69
$8h2$	0.95
$8h3$	1.05
$8h4$	0.89
$8d0$	0.89
$8d2$	1.00

Imperfection of a finite lattice is defined based on disparity of geometric nearest neighbor rings. A "perfect" supercells of N sites on a square lattice shall have complete rings of four first nearest neighbor, four second nearest neighbors, four third nearest neighbors, eight fourth nearest neighbors, $\ldots$, $n_i(N)$ ith nearest neighbor to each lattice points, where $n_i(N)$ equals the number of ith nearest neighbor in the infinite lattices $n_i(\infty)$ with one exception. Since we have a finite lattice points, $n_i(\infty)$ would not have a complete ring.

An example of a perfect supercell is $20h8$ (see Figure 6.8). Supercell $20h8$ has three closed rings and one open outermost ring. For a lattice point number 15 in $20h8$ has ring of neighborhood: the first ring contains 3, 7, 14, 16, the second ring contains 2, 4, 6, 8, the third ring contains 11, 13, 17, 19, and the fourth ring contains 1, 5, 9, 10, 12, 18, 20. Then each lattice point in the $20h8$ supercell has 4, 4, 4, 7 first to fourth geometric nearest neighbors respectively.

Thus we define *imperfection* J to quantify divergence from perfect neighborhood lattice points. We transfer each lattice points from the outermost ring inward in order to make perfect neighborhood, then we count how many steps are required in order to make perfect rings.

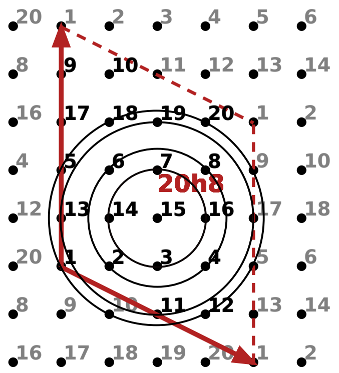


Figure 6.8: Supercell $20h8$ is a perfect supercell with three closed ring and one open ring which is the outermost ring. It has neighborhood 4, 4, 4, 7 from the first to the fourth nearest neighbor respectively.

For example, supercell $24h10$ is not a perfect supercell with neighborhood 4, 4, 4, 7, 2, 0, 2, the full seven rings neighbourhood would have 4, 4, 4, 8, 4, 4, 8. In order to make it perfect, we move a point from the outermost ring to the forth ring (three steps). At this point the neighborhood equals 4, 4, 4, 8, 2, 0, 1 and $J = 3$, then move another point from the outermost ring to the fifth ring (two steps). Hence the neighborhood now equals 4, 4, 4, 8, 3 and $J = 3 + 2 = 5$. Accordingly, we conclude that $24h10$ has five imperfection. The scheme of this counting is illustrated in Figure 6.9.

$$
\begin{array}{ccccccc|c}
4 & 4 & 4 & 8 & 4 & 4 & 8 & \\
\hline
4 & 4 & 4 & 7 & 2 & 0 & 2 & +3 \\
4 & 4 & 4 & 8 & 2 & 0 & 1 & +2 \\
4 & 4 & 4 & 8 & 3 & 0 & 0 &
\end{array}
$$

Figure 6.9: The first row numbers represent neighborhood of the seven full rings. The numbers in the following rows represent the neighborhood of $24h10$ at initial place and after moving the necessary points. The most right number represent the number of imperfection that is obtained from all the steps within the corresponding row.

Another example of supercell with nonzero imperfection is $8d2$ (see Figure 6.10). It has neighborhood 4, 2, 1, and it is quite straightforward that $8d2$ has $J = 1$.

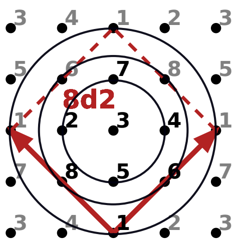


Figure 6.10: Supercell $8d2$ has nonzero imperfection with neighborhood 4, 2, 1. For every lattice point with number 3, first ring contains lattice points number 2, 4, 6, 7, second ring has 6, 8 and the outermost ring has 1, hence $J = 1$.

As a continuation of Listing 6.2, in Listing 6.3 is a program that counts geometrical imperfection J. The main function is called `imperfection_geometrical` which counts J of a supercell. Another important function is called `imperfection_count`, it returns imperfection by comparing the actual neighborhood and the perfect neighborhood. We can reuse this function later to calculate other imperfections such as ferromagnet and bipartite imperfection. Function `perfect_geometrical_nn` counts the neighborhood of a perfect square lattice, we use this as perfect neighborhood reference since we do not find any formula to count ith geometrical nearest neighbor neither to find the corresponding distance. An auxiliary function `find_index` is used to find index in an array of a list of lattice points.

Listing 6.3: Betts criteria: geometrical imperfection J

```
#Returns the geometrical imperfection of a square lattice with supercell Sc
def imperfection_geometrical(Sc):
  geo_nn = perfect_geometrical_nn(int(np.amax(np.abs(Sc)))) #Perfect neighborhood
  bettsn = number_latticepoints(Sc)              #Betts numbering (See Listing 6.1)
  #Find lattice point around the middle of supercell
  max_p  = np.amax(bettsn['positions'],axis=0)   #Maximum position
  min_p  = np.amin(bettsn['positions'],axis=0)   #Minimum position
  p_mid = np.ceil(0.5*(max_p + min_p)).astype(int)  #Middle position
  i_mid = find_index(p_mid[0],p_mid[1],Sc,bettsn['positions'])
  dist  = [mvo.norm(np.matrix([[p[0]-p_mid[0]],[p[1]-p_mid[1]]]))\
          for p in bettsn['positions']] #List of distance for each lattice points
  dist_max = max(dist)
  for i,d in enumerate(geo_nn['nearn']): #Only take the necessary neighborhood list
    if d > dist_max :
      perf_nearn       = geo_nn['nearn'][:i]
      perf_count_nearn = geo_nn['count_nearn'][:i]
      break
  count_nearn  = [0]*len(perf_nearn)
  half_wh= [int(np.ceil(0.5*(max_p[0]-min_p[0])))+1,int(np.ceil(0.5*(max_p[1]-min_p[1])))
      +1]
  for i in range(-half_wh[0],half_wh[0]+1):  #i,j positions relative to the middle
    for j in range(-half_wh[1],half_wh[1]+1):
      d   = np.sqrt(i*i+j*j)
      p   = [p_mid[0]+i,p_mid[1]+j]
      idx = find_index(p[0],p[1],Sc,bettsn['positions']) #Real position
      if dist[idx] > d : dist[idx] = d #Find the shortest distance among all equiv points
  for d in dist :
    for i,perf_d in enumerate(perf_nearn): #Add counting after find the distance
      if abs(d-perf_d)< eps :
        count_nearn[i] += 1
        break
  return imperfection_count(count_nearn,perf_count_nearn)  #Count imperfection

#Count imperfection by moving the points from the outermost shell inward
def imperfection_count(real, perfect): #real neighborhood, perfect neighborhood
  imperf = 0
  if len(real) > 1 :   #One shell is trivial, imperfection always 0
    for i in range(len(real)-1):
      diff = perfect[i] -  real[i]
      if(diff):
        idx = len(real)-1
        j   = diff
        while j > 0 :  #Keep moving points inward until empty
          if real[idx]:
            real[idx] -= 1
            real[i]   += 1
            imperf += (idx - i)
            j -= 1
          else :
            if(idx > i) :
              idx -= 1
          if sum(real[i+1:]) <= 0 : #Stop when only the outermost shell that open
            return imperf
  return imperf

#Count neighborhood of geometrical nearest neighbor for perfect lattice
#Returns an object with properties: 'nearn' and 'count_nearn'
def perfect_geometrical_nn(N):
  dist, geo_nn = [], {}
  #Create a square lattice (2N+1)x(2N+1), then calculate all distances to the center
  for i in range(-N,N+1):
    for j in range(-N,N+1):
      dist.append(np.sqrt(i*i+j*j))
  dist.sort()   #Sort the list of distance
  geo_nn['nearn'] =  [dist[1]]     #List of distance of nearest neighbor
  geo_nn['count_nearn'] = [0]      #List of the count of the neighborhood
  for d in dist[1:] : #Add to the counting once find the same distance
    if abs(d - geo_nn['nearn'][-1]) < eps :
      geo_nn['count_nearn'][-1] += 1
    else :
      geo_nn['nearn'].append(d)
      geo_nn['count_nearn'].append(1)
  return geo_nn

#Auxiliary function: find index of position in the list
def find_index(i,j,Sc,lpos): #x,y, supercell, list of positions
  sc_pos  = np.linalg.inv(Sc)*np.matrix([[i],[j]])  #Position with basis Sc
  sc_pos -= np.floor(sc_pos)   #Make sure it is inside supercell
  if abs(abs(sc_pos[0])-1.) <= eps : sc_pos[0] = 0. #All corners are the origin
  if abs(abs(sc_pos[1])-1.) <= eps : sc_pos[1] = 0.
  new_pos = Sc*sc_pos
  npos    = [int(round(new_pos[0,0])), int(round(new_pos[1,0]))]
  index   = lpos.index(npos)
  return index
```

We add some clarifications for all the corresponding geometrical properties by reproducing the geometrical properties part of Tables 1 and 2 from [2] (see Tables 6.4 and 6.5). We find some discrepancies of squareness from Table 6.4, $20h5$ ($\sqrt{\sigma} = 0.95$ from reference), and from Table 6.5 on sublattices $13h3$, $23h5$ ($\sqrt{\sigma} = 1.01, 1.03$ from reference consecutively). In addition, we also find two discrepancies on imperfection calculation namely on $8d2$ and $20h8$ from Table 6.5 with values $J = 0$ and $J = 5$ from reference consecutively. We have discussed previously that $8d2$ (see Figure 6.10) has nonzero imperfection $J = 1$ and $20h8$ (see Figure 6.8) is a perfect supercell with $J = 0$.

We also find eight discrepancies on assigning the point group of symmetries. It is not easy to compare, since the supercells were noticed only by labels in [2]. There is high probability that we do not apply the identical supercells.

Table 6.4: Geometrical properties of bipartite square lattices that are reproduced from Table 1 of [2]. The discrepacies to Table 1 of [2] are underlined.

$N\alpha k$	L_1	L_2	S	$\sqrt{\sigma}$	J
8d2	(-2, 2)	(2, 2)	O_h	1.000	$\underline{1}$
10h3	(3, 1)	(1,-3)	C_4	1.000	1
12h3	(3, 1)	(0,-4)	C_2	1.011	2
12h5	(2,-2)	(3, 3)	$\underline{D_4}$	0.961	3
14h3	(3, 1)	(2,-4)	C_2	0.975	2
16d4	(-4, 2)	(0, 4)	D_2	1.053	1

16q0	(4, 0)	(0, 4)	*Hypercubic*	1.000	3
18h5	(3,-3)	(2, 4)	C_2	1.026	0
18t3	(-3, 3)	(3, 3)	$\underline{O_h}$	1.000	2
20h5	(5, 1)	(0,-4)	C_2	$\underline{0.995}$	3
20d4	(4, 2)	(2,-4)	C_4	1.000	3
22h5	(5, 1)	(2,-4)	C_2	1.013	1
24h5	(5, 1)	(-1,-5)	$\underline{D_4}$	1.041	0
24h7	(3,-3)	(4, 4)	$\underline{D_4}$	0.980	8
24h9	(-3,-3)	(3,-5)	C_2	0.989	6
24d4	(4, 2)	(4,-4)	C_2	1.011	6
26h5	(5, 1)	(1,-5)	C_4	1.000	2

Table 6.5: Geometrical peroperties for nonbipartite square lattices for even and odd N that are reproduced from Table 2 of [2]. The discrepancies to Table 2 of [2] are underlined.

$N\alpha k$	L_1	L_2	S	$\sqrt{\sigma}$	J
			Even N		
12d3	(-3, 2)	(3, 2)	D_2	1.041	0
14h4	(4, 1)	(2,-3)	C_2	1.025	0
16h4	(4, 1)	(0,-4)	C_2	1.015	1
16h6	(-2,-3)	(4,-2)	C_2	0.992	0
18h4	(4, 1)	(2,-4)	C_2	1.010	0
20h4	(4, 1)	(0,-5)	C_2	1.005	1
20h8	(4,-2)	(0, 5)	C_2	1.053	$\underline{0}$
20d5	(0, 4)	(-5,-2)	D_2	1.012	3
22h4	(4, 1)	(2,-5)	C_2	0.987	4
22h6	(-2,-4)	(4,-3)	C_2	1.005	3
24h10	(4,-2)	(2, 5)	C_2	0.993	5
24t4	(-4, 3)	(4, 3)	D_2	1.021	3
26h10	(-4,-3)	(2,-5)	C_2	1.016	3
			Odd N		
9h3	(3, 1)	(0,-3)	C_2	1.026	0
9t0	(3, 0)	(0, 3)	$\underline{\textit{Hypercubic}}$	1.000	0

11h3	(3, 1)	(2,-3)	C_2	1.013	0
13h3	(3, 1)	(1,-4)	C_2	0.984	2
13h5	(-2,-3)	(3,-2)	C_4	1.000	0
15h4	(4, 1)	(-1,-4)	D_4	1.065	0
15h6	(3,-2)	(3, 3)	C_2	1.003	0
17h4	(4, 1)	(1,-4)	C_4	1.000	0
17h5	(2,-3)	(3, 4)	C_2	1.000	0
19h4	(4, 1)	(3,-4)	C_2	1.030	0
21h4	(4, 1)	(1,-5)	C_2	0.989	2
21h6	(3,-3)	(3, 4)	C_2	0.998	4
21h8	(-3,-3)	(2,-5)	D_4	1.026	2
23h4	(4, 1)	(3,-5)	C_2	0.990	4
23h5	(5, 1)	(3,-4)	C_2	1.053	0
25h7	(-3,-4)	(4,-3)	C_4	1.000	4
25p0	(5, 0)	(0, 5)	*Hypercubic*	1.000	0

Ferromagnetic imperfection, I_F, is defined as an indicator of the gooddness of the supercells in methods of exact diagonalization of ferromagnetic models. This indicator is based on disparity to the topologically perfect infinite lattice. On a perfect lattice, each lattice points has 4 first nearest neighbor, 8 second nearest neighbor, ..., $4n$ of nth nearest neighbor. Note that the term of "neighbor" here means the topological neighbor in the *Manhattan distance* between two points.

An example of a topologically imperfect supercell is $28h11$ (see Figure 6.11). This supercell contains four shells of neighborhood. The first shell contains the lattice points with number 4, 14, 16, 26, the second shell contains 3, 5, 9, 13, 17, 21, 25, 27, the third shell contains 2, 6, 8, 10, 12, 18, 20, 22, 24, 28, and the fourth shell contains 1, 7, 9, 11, 23. Hence the neighborhood from the first to the fourth nearest neighbor are 4, 8, 10, 5. In principle, we count the ferromagnetic imperfection in the same approach as we count the geometrical imperfection. The neighborhood of a lattice point with four full shells must have 4, 8, 12, 16 sites. Then, in order to make a topologically perfect lattice, we move two points from the fourth shell to the third shell, hence we obtain an imperfection of two. At this point the neighborhood are 4, 8, 12, 3, then, we conclude that $28h11$ has $I_F = 2$.

Bipartite imperfection, I_B, is defined as an indicator of the goodness of the supercells in method of exact diagonalization of antiferromagnetic models. These models such as the $\frac{1}{2}$ Heisenberg antiferromagnet require bipartite finite lattice. This means that for a supercell that is defined by vectors $\mathbf{L_1} = (l_{11}, l_{12})$ and $\mathbf{L_2} = (l_{21}, l_{22})$, the summs of $l_{11} + l_{12}$ and $l_{21} + l_{22}$ must be even numbers.

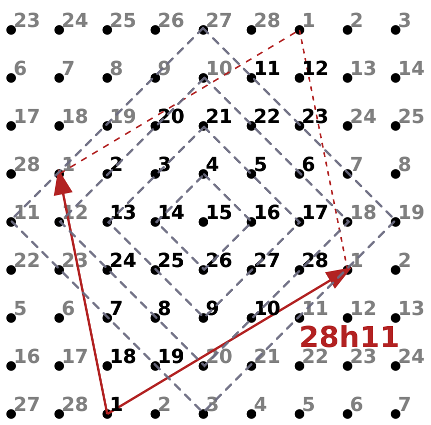


Figure 6.11: Supercell $28h11$ has nonzero topological imperfection. Each lattice point has four shells of neighborhood with the number of nearest neighbor from the first to the fourth shell 4, 8, 10, 5, hence $I_F = 2$.

Basically, counting the bipartite imperfection is similar to counting the ferromagnetic imperfection, but we separate the counting between the odd and the even number of shells *i.e.* treating two sublattices separately. For example, let us observe again supercell $28h11$. There are four shells of neighborhood which is 4, 8, 10, 5. The odd numbered nearest neighbor shells are the first and the third shell, hence its neighborhood is 4, 10. In the same manner, for the even numbered of nearest neighbors we get 8, 5. For the odd shells case, the full shell must have neighborhood 8, 16, and we have neighborhood 8, 5. Since the open shell is only the outermost, at this points $I_B = 0$. For the even shells case, the full shell must have neighborhood 4, 12, and we have neighborhood 4, 10. Again, only the outermost shell is open, hence we conclude that $28h11$ has $I_B = 0$.

As a continuation of Listing 6.3, in Listing 6.4 gives a program to count ferromagnetic imperfection I_F and bipartite imperfection I_B. Function `topological_nearn` returns the neighborhood of a supercell. Function `imperfection_ferromagnet` and `imperfection_bipartite` count the value of I_F and I_B respectively.

Listing 6.4: Betts criteria : topological imperfections I_F and I_B

```
#Count ferromagnetic imperfection of a supercell
def imperfection_ferromagnet(Sc):
  neigs_count   = topological_nearn(Sc)
  perfect_ferro = [(n+1)*4 for n in range(len(neigs_count))]
  return imperfection_count(neigs_count,perfect_ferro) #Function from Listing 6.3

#Count bipartite imperfection of a supercell
def imperfection_bipartite(Sc):
  if np.all((Sc[0,:]+Sc[1,:])%2.== 0.):  #Must be bipartite supercell
    neigs_count  = topological_nearn(Sc)
    perfect_odd  = [(2*n+1)*4 for n in range(int(np.ceil(0.5*len(neigs_count)))) ]
    perfect_even = [2*(n+1)*4 for n in range(int(np.floor(0.5*len(neigs_count))))]
    ncount_odd   = [neigs_count[2*n] for n in range(int(np.ceil(0.5*len(neigs_count))))]
    ncount_even  = [neigs_count[2*n+1] for n in range(int(np.floor(0.5*len(neigs_count))))
        ]
    return imperfection_count(ncount_odd, perfect_odd) + \
           imperfection_count(ncount_even, perfect_even)  #Function from Listing 6.3
  else :  print 'Supercell is not bipartite'  #If supercell is not bipartite

#List the topological neighborhood of supercell Sc
def topological_nearn(Sc):
  nearN     = 0         # N-th nearest neighbor
  bettsn    = number_latticepoints(Sc)  #Numbering lattice points (see Listing 6.1)
  neigs     = [False]*len(bettsn['positions'])  #Mark the counted neighborhood
  neigs_count = []       #The neighborhood list
  neigs[bettsn['positions'].index([0,0])] = True #Start from the corner
  while not np.all(neigs):  #Until all lattice points are marked
    nearN += 1
    neigs_count.append(0)
    for i in range(-nearN,nearN+1) :    #Start to walk from 1 to n-th nearest neighbor,
      for j in range(-nearN,nearN+1) :  #based on the number of Manhattan steps
        if abs(i)+abs(j) == nearN :
          idx = find_index(i,j,Sc,bettsn['positions'])
          if not neigs[idx] :
            neigs_count[nearN-1] += 1
            neigs[idx] = True
  return neigs_count
```

As a clarification, we reproduce Table 1A from [2] on Table 6.6, but we add more information about geometrical properties of the corresponding supercells. For a given supercell vectors l_1, l_2, we reduce them into L_1, L_2 with the LLL reduction. From the L_1, L_2, we calculate the geometrical properties such as $\sqrt{\sigma}$, J, S and the topological properties such as I_F and I_B. We find one disagreement to our reference on the value of I_F of $28h11$ (equivalent up to σ_y symmetry with $28h17$). We have already counted the topological imperfection of $28h11$ as an example case (see Figure 6.11), where $I_F = 2$, however, in our reference $I_F = 0$.

Table 6.6: Description of geometrical properties and topological properties of various supercells that are reproduced from Table 1A of [3]. The discrepancies to Table 1A of [3] are underlined.

$N\alpha k$	l_1	l_2	L_1	L_2	$\sqrt{\sigma}$	J	S	I_F	I_B
$8d2$	(2, 2)	(0, 4)	(2, 2)	(-2, 2)	1.000	1	O_h	0	0
$8h6$	(2, 3)	(0, 4)	(-2, 1)	(2, 3)	0.949	1	C_2	0	–
$9t0$	(3, 0)	(0, 3)	(3, 0)	(0, 3)	1.000	0	*Hypercubic*	0	–
$9t1$	(1, 3)	(0, 9)	(1, 3)	(-3, 0)	1.026	0	C_2	0	–
$10h4$	(2, 3)	(0, 5)	(2, 3)	(-2, 2)	0.995	1	C_2	0	–
$10h7$	(1, 3)	(0,10)	(1, 3)	(-3, 1)	1.000	1	C_4	1	0
$11h4$	(1, 3)	(0,11)	(1, 3)	(-3, 2)	1.013	0	C_2	0	–
$12d2$	(2, 2)	(0, 6)	(2, 2)	(-4, 2)	0.971	3	D_4	2	0
$12t2$	(2, 3)	(0, 6)	(2, 3)	(-2, 3)	1.041	0	D_2	0	–
$12t1$	(1, 3)	(0,12)	(1, 3)	(-4, 0)	1.011	2	C_2	2	0
$13h8$	(1, 5)	(0,13)	(-3,-2)	(-2, 3)	1.000	0	C_4	0	–

14*d*4	(1, 4)	(0,14)	(1, 4)	(-3, 2)	1.025	0	C_2	0	–
14*h*5	(1, 3)	(0,14)	(1, 3)	(-4, 2)	0.975	2	C_2	2	0
15*t*3	(1, 6)	(0,15)	(-2, 3)	(3, 3)	1.003	0	C_2	0	–
15*h*4	(1, 4)	(0,15)	(1, 4)	(-4,-1)	1.065	0	D_4	0	–
16*q*2	(2, 4)	(0, 8)	(2, 4)	(-4, 0)	1.053	1	D_2	1	0
16*q*0	(4, 0)	(0, 4)	(4, 0)	(0, 4)	1.000	3	*Hypercubic*	3	1
16*h*6	(2, 3)	(0, 8)	(2, 3)	(-4, 2)	0.992	0	C_2	0	–
16*h*4	(4, 1)	(0, 4)	(4, 1)	(0, 4)	1.015	1	C_2	1	–
16*h*13	(1, 5)	(0,16)	(-3, 1)	(1, 5)	0.949	3	C_2	3	1
17*h*13	(1, 4)	(0,17)	(1, 4)	(-4, 1)	1.000	0	C_4	0	–
17*h*7	(1, 5)	(0,17)	(-3, 2)	(4, 3)	1.000	0	C_2	0	–
18*t*3	(3, 3)	(0, 6)	(3, 3)	(-3, 3)	1.000	2	O_h	0	0
18*h*13	(1, 7)	(0,18)	(-3,-3)	(-2, 4)	1.026	0	C_2	0	0
18*d*5	(1, 4)	(0,18)	(1, 4)	(-4, 2)	1.010	0	C_2	0	–
18*h*3	(3, 1)	(0, 6)	(3, 1)	(-3, 5)	0.923	6	C_2	4	2
19*h*4	(1, 5)	(0,19)	(1, 5)	(-4,-1)	1.038	0	C_2	0	–
19*h*11	(1, 7)	(0,19)	(-3,-2)	(-2, 5)	0.970	2	C_2	0	–
20*d*6	(2, 4)	(0,10)	(2, 4)	(-4, 2)	1.000	3	C_4	1	0
20*q*3	(1, 8)	(0,20)	(-2, 4)	(5, 0)	1.053	0	C_2	0	–
20*q*1	(1, 4)	(0,20)	(1, 4)	(-5, 0)	1.005	1	C_2	1	–
20*h*5	(5, 1)	(0, 4)	(5, 1)	(0, 4)	0.995	3	C_2	3	1
21*h*6	(3, 4)	(0, 7)	(3, 4)	(-3, 3)	0.998	4	C_2	0	–
21*h*17	(1, 5)	(0,21)	(1, 5)	(-4, 1)	0.989	2	C_2	2	–
21*h*8	(1, 8)	(0,21)	(-3,-3)	(-5, 2)	1.026	2	D_4	0	–
22*h*9	(1, 5)	(0,22)	(1, 5)	(-4, 2)	1.013	1	C_2	2	0
22*d*4	(1, 6)	(0,22)	(-4,-2)	(-3, 4)	1.005	3	C_2	0	–
22*d*6	(1, 4)	(0,22)	(1, 4)	(-5, 2)	0.987	4	C_2	1	–
23*h*14	(1, 5)	(0,23)	(1, 5)	(-4, 3)	1.053	0	C_2	0	–
24*d*8	(4, 4)	(0, 6)	(4, 4)	(-4, 2)	1.011	6	C_2	3	0
24*s*2	(2, 6)	(0,12)	(-4, 0)	(-2, 6)	0.971	10	D_2	5	1
24*s*0	(4, 0)	(0, 6)	(4, 0)	(0, 6)	0.961	12	D_2	7	2
24*h*5	(1, 5)	(0,24)	(1, 5)	(-5,-1)	1.041	0	D_4	3	0
24*t*3	(1, 9)	(0,24)	(-3,-3)	(-5, 3)	0.989	6	C_2	3	0
24*h*7	(1, 7)	(0,24)	(-3, 3)	(4, 4)	0.980	8	D_4	3	0
24*t*4	(4, 3)	(0, 6)	(4, 3)	(-4, 3)	1.021	3	D_2	0	–

25*h*18	(1, 7)	(0,25)	(-4,-3)	(-3, 4)	1.000	4	C_4	0	–
25*h*15	(5, 2)	(0, 5)	(5, 2)	(0, 5)	1.036	0	C_2	2	–
26*h*15	(1, 7)	(0,26)	(-4,-2)	(-3, 5)	0.984	6	C_2	3	0
26*h*21	(1, 5)	(0,26)	(1, 5)	(-5, 1)	1.000	2	C_4	5	1
26*h*16	(2, 5)	(0,13)	(2, 5)	(-4, 3)	1.016	3	C_2	0	–
27*t*5	(1, 6)	(0,27)	(1, 6)	(-4, 3)	1.049	2	C_2	0	–
27*h*11	(1, 5)	(0,27)	(1, 5)	(-5, 2)	1.008	2	C_2	2	–
28*d*8	(2, 4)	(0,14)	(2, 4)	(-6, 2)	0.975	9	C_2	4	1
28*h*17	(1, 5)	(0,28)	(1, 5)	(-5, 3)	1.025	1	C_2	$\underline{2}$	0
28*h*6	(2, 5)	(0,14)	(2, 5)	(-4, 4)	1.042	3	C_2	0	–
28*h*8	(4, 4)	(0, 7)	(4, 4)	(-4, 3)	1.001	5	C_2	0	–
29*h*17	(1,12)	(0,29)	(-2, 5)	(5, 2)	1.000	4	C_4	0	–
29*h*8	(1,11)	(0,29)	(-3,-4)	(-5, 3)	0.997	4	C_2	0	–
30*t*5	(5, 3)	(0, 6)	(5, 3)	(-5, 3)	1.065	1	D_2	1	0
30*t*6	(2, 6)	(0,15)	(2, 6)	(-4, 3)	1.011	4	C_2	0	–
30*p*2	(2, 5)	(0,15)	(2, 5)	(-6, 0)	1.034	3	C_2	1	–
30*h*5	(5, 1)	(0, 6)	(5, 1)	(0, 6)	1.003	5	C_2	5	2
30*h*7	(1,13)	(0,30)	(-2, 4)	(7, 1)	0.971	11	C_2	5	2
31*h*12	(1,13)	(0,31)	(-2, 5)	(5, 3)	1.005	2	C_2	0	–
31*h*7	(1, 9)	(0,31)	(-3, 4)	(4, 5)	1.000	4	C_2	0	–
32*q*4	(4, 4)	(0, 8)	(4, 4)	(-4, 4)	1.000	6	O_h	0	0
32*d*6	(2, 6)	(0,16)	(2, 6)	(-4, 4)	1.053	2	D_4	0	0
32*h*25	(1, 9)	(0,32)	(-4,-4)	(-3, 5)	1.015	2	C_2	0	0
32*d*11	(1, 6)	(0,32)	(1, 6)	(-5, 2)	1.008	3	C_2	1	–
32*q*3	(1,12)	(0,32)	(-3,-4)	(-5, 4)	0.985	6	C_2	0	–

6.4 The Tight-Binding Model

At this point, we are able to estimate the good supercells that underlay two dimensional square lattice to use for the diagonalization of the Hamiltonians. In the following we shall perform an exact diagonalization of *tight-binding* model to estimate the ground state energy per site and observe apparent correlation between geometry and the ground state energy. In the following we shall construct the finite Hamiltonians from the corresponding supercells with consideration of several boundary conditions, namely *periodic boundary condition*, *anti-periodic boundary condition*, and *open boundary condition*, then we observe the band structures.

Here, we have a supercell as an external potential that constructs a lattice (crystal) by applying a boundary condition and electrons moving around the crystal. It is assumed that the crystal potential is strong, it means that when an electron is captured by an ion, it remains there for a long time before tunneling or leaking to the next ion. Within this capture interval, the electron move mainly only around the corresponding ion and hardly influenced by other electrons from the next sites. Hence, this model is primary suited to the low lying narrow band for which the radius of atoms in the site are much smaller than the lattice constant.

Correspondingly, we will apply boundary conditions to the Hamiltonian of a single atom

$$H_{\text{single}} = -\frac{1}{2}\nabla_{\mathbf{r}}^2 + V_{ext}(\mathbf{r}). \tag{6.1}$$

Let us start with an atomic orbital $\phi_i(\mathbf{r})$ of site i then examine the presence of other atoms in the lattice. Then, the matrix elements of the Hamiltonian

$$\begin{aligned}\langle\phi_i| H_{\text{single}} |\phi_i\rangle &= \varepsilon \\ \langle\phi_i| H_{\text{single}} |\phi_j\rangle &= -t_{ij}.\end{aligned} \tag{6.2}$$

Where ε is the ground state energy of a free electron and t_{ij} is the interaction between sites i and j. Since we assume that one electron is mainly orbiting in a local site, hence only the nearest neighbor interaction is considered. Then if we construct the Hamiltonian for a one dimensional infinite lattice, it shall form a tridiagonal matrix

$$H = \begin{pmatrix} \ddots & \ddots & \ddots & & & \\ & -t & \varepsilon & -t & & \\ & & -t & \varepsilon & -t & \\ & & & \ddots & \ddots & \ddots \end{pmatrix} \tag{6.3}$$

which has infinite dimension. In order to construct a finite Hamiltonian, one may impose periodicity to a finite lattice, *i.e.* the periodic potential of lattice. Hence, we shall find the suitable wave function to our system. Firstly, we introduce three of periodic boundary conditions for a given wave function ψ, where $\mathbf{r}$ as a local vector inside supercell and $\mathbf{R}$ as a vector that constructs supercell

$$\begin{aligned}\psi_{pbc}(\mathbf{r}+\mathbf{R}) &= \psi(\mathbf{r}) \\ \psi_{abc}(\mathbf{r}+\mathbf{R}) &= -\psi(\mathbf{r}) \\ \psi_{obc}(\mathbf{r}+\mathbf{R}) &= 0.\end{aligned} \tag{6.4}$$

Where *pbc*, *abc*, and *obc* denote periodic boundary condition, anti-periodic boundary condition, and open boundary condition respectively. One naturally consider the reciprocal lattice $\mathcal{R}_{\mathcal{L}}$ that associate with lattice $\mathcal{L}$ for approaching the periodic functions. The Fourier expansion of a general function $V(\mathbf{r})$ is given by

$$V(\mathbf{r}) = \int d^d k \hat{V}(\mathbf{k}) e^{i\mathbf{k}\cdot\mathbf{r}}. \tag{6.5}$$

By this property of infinite periodic potential lattice we construct the Hamiltonian dependent to k values. We set an ansatz $|\psi_{\mathbf{k};\mathbf{r}}\rangle$ as eigenfunction

$$|\psi_{\mathbf{k};\mathbf{r}}\rangle = \frac{1}{\sqrt{N}}\sum_{\mathbf{R}} e^{i\mathbf{k}\cdot\mathbf{R}} |\phi_{\mathbf{R}+\mathbf{r}}\rangle \tag{6.6}$$

which is summation of all over lattice by translation vector **R** as supercell vectors that consists of N lattice points. The value of **k** is now restricted to the primitive reciprocal cell. Under translations, vector $\psi_{\mathbf{k}}(\mathbf{r})$ transforms as

$$\psi_{\mathbf{k}}(\mathbf{r}+\mathbf{R}) = e^{i\mathbf{k}\cdot\mathbf{R}}\psi_{\mathbf{k}}(\mathbf{r}), \tag{6.7}$$

this is known as the Bloch theorem. The factor $e^{i\mathbf{k}\cdot\mathbf{R}}$ is the phase shift whose value depends on the boundary conditions 6.4. Now apply periodic potential over k-space

$$\langle\psi_{\mathbf{k};\mathbf{r}}|\, H\, |\psi_{\mathbf{k}';\mathbf{r}'}\rangle = \frac{1}{N}\sum_{\mathbf{R}}\sum_{\mathbf{R}'} e^{-i\mathbf{k}.\mathbf{R}} e^{i\mathbf{k}'.\mathbf{R}'} \underbrace{\langle\phi_{\mathbf{R}+\mathbf{r}}|\, H\, |\phi_{\mathbf{R}'+\mathbf{r}'}\rangle}_{H_{\mathbf{R}+\mathbf{r},\mathbf{R}'+\mathbf{r}'}}.$$

By imposing infinite periodic potential property of lattice $H_{\mathbf{R}+\mathbf{r}+\mathbf{R_0},\mathbf{R}'+\mathbf{r}'+\mathbf{R_0}} = H'_{\mathbf{R}+\mathbf{r},\mathbf{R}'+\mathbf{r}'}$

$$\begin{aligned}
\langle\psi_{\mathbf{k};\mathbf{r}}|\, H\, |\psi_{\mathbf{k}';\mathbf{r}'}\rangle &= \frac{1}{N}\sum_{\mathbf{R}}\sum_{\mathbf{R}'} e^{-i\mathbf{k}.\mathbf{R}} e^{i\mathbf{k}'.\mathbf{R}'} \underbrace{H_{\mathbf{r},\mathbf{R}'-\mathbf{R}+\mathbf{r}'}}_{H_{\mathbf{r},\mathbf{R}''+\mathbf{r}'}} \\
&= \frac{1}{N}\sum_{\mathbf{R}}\sum_{\mathbf{R}''+\mathbf{R}} e^{-i\mathbf{k}.\mathbf{R}} e^{i\mathbf{k}'(\mathbf{R}''+\mathbf{R})} H_{\mathbf{r},\mathbf{R}''+\mathbf{r}'} \\
&= \frac{1}{N}\underbrace{\sum_{\mathbf{R}} e^{i(\mathbf{k}-\mathbf{k}')\mathbf{R}}}_{N\delta_{\mathbf{k},\mathbf{k}'}} \sum_{\mathbf{R}+\mathbf{R}''} e^{i\mathbf{k}'\mathbf{R}''} H_{\mathbf{r},\mathbf{R}''+\mathbf{r}'} \\
&= \delta_{\mathbf{k}\mathbf{k}'}\sum_{\mathbf{R}''} e^{i\mathbf{k}'.\mathbf{R}''} H_{\mathbf{r},\mathbf{R}''+\mathbf{r}'} \\
&= \delta_{\mathbf{k},\mathbf{k}'} H_{\mathbf{r}\mathbf{r}'}(\mathbf{k}').
\end{aligned} \tag{6.8}$$

By function $\delta_{\mathbf{k},\mathbf{k}'}$, we see that the Hamiltonian only couples states whose wave-vectors differ by reciprocal lattice vectors. Hence, we can determine the eigenfunctions $\psi_{n,\mathbf{k}}(\mathbf{r})$ by solving the eigenvalue problem of H_{single}, where n is the band index

$$\left(-\frac{1}{2}\nabla^2_{\mathbf{r}} + V_{ext}(\mathbf{r})\right)\psi_{n,\mathbf{k}}(\mathbf{r}) = \varepsilon_{n,\mathbf{k}}\psi_{n,\mathbf{k}}(\mathbf{r}). \tag{6.9}$$

Now we reconstruct our Hamiltonian from 6.3 by imposing boundary conditions. By using the Bloch ansatz, from 6.7 we determine the phase $\varphi = e^{i\mathbf{k}\cdot\mathbf{R}}$. Then, we construct the finite Hamiltonians from 6.8, with matrix elements

$$H_{\mathbf{r},\mathbf{r}'}(\mathbf{k}) = \varepsilon + \sum_{\mathbf{R}} \varphi\, t. \tag{6.10}$$

For a given one dimensional lattice, with N size of supercell, the Hamiltonians for each boundary conditions are in the following.

For the case of periodic boundary condition, $\varphi = 0$. The Hamiltonian has the form

$$H = \begin{pmatrix} \varepsilon & -t & & & -t \\ -t & \varepsilon & -t & & \\ & \ddots & \ddots & \ddots & \\ & & & \ddots & -t \\ -t & & & -t & \varepsilon \end{pmatrix}_{N\times N} \tag{6.11}$$

with the k values $k_m = \frac{2\pi m}{R}$, where $m = 0, 1, \ldots, N-1$.

For the case of anti-periodic boundary condition, $\varphi = \pi$. The Hamiltonian shall has the form

$$H = \begin{pmatrix} \varepsilon & -t & & & t \\ -t & \varepsilon & -t & & \\ & \ddots & \ddots & \ddots & \\ & & & \ddots & -t \\ t & & & -t & \varepsilon \end{pmatrix}_{N\times N} \tag{6.12}$$

with the k values $k_m = \frac{2\pi m - \pi}{R}$, where $m = 1, 2, \ldots, N$. The wave function changing its phase while stepping over boundary.

At last, for the case of open boundary condition, there is no periodicity imposed the Hamiltonian is simply truncated. Then, the Hamiltonian has the form

$$H = \begin{pmatrix} \varepsilon & -t & & & \\ -t & \varepsilon & -t & & \\ & \ddots & \ddots & \ddots & \\ & & & \ddots & -t \\ & & & -t & \varepsilon \end{pmatrix}_{N\times N} . \tag{6.13}$$

Notice, that a boundary condition basically shift the values of $\mathbf{k}$.

Chapter 7

Summary

In this thesis, our main goal was giving a guidance for the usefulness of supercells underlaying two dimensional square lattice based on the criteria of Betts *et al.* [2, 3].

We started from the gentle explanations of the concept of lattice, primitive vectors, and primitive cell. Then we introduced the idea of constructing supercell (sublattice) and all the possibilities to construct equivalent lattices and supercells using unimodular integer matrices. By these notions, we provide the complete Phyton codes to construct classes `Lattice` and `Sublattice` that share methods since they have similar behaviors.

Correspondingly, we introduced the Hermite Normal Form (HNF) matrices and we could reduce any nonsingular square matrix into HNF form. Thus, we can tell if two integer matrices are equivalent by comparing their HNFs. Turnabout, we list the possible HNFs by a given volume. We applied this idea for listing supercells by its HNFs by given the number of lattice points inside supercells. Afterwards, we reduced every supercells by LLL reduction to obtain the most compact form of supercells. We introduced the point group symmetries of the underlaying the lattices. Then, we eliminated the supercells that are equivalent by applying the corresponding symmetry transformations. Hence, we could produce all unique supercells for a given volume. We also provided the complete Python code for listing the unique supercells that consists of N lattice points.

From the unique list of supercells, the Betts criteria are applied in order to guide to the useful ones for the exact diagonalization of quantum spin systems. We introduced the numbering of lattice points and the labeling the supercells. The geometrical properties and topological properties of supercells were rigorously discussed. The geometrical properties consists of squareness (σ), geometrical imperfection (J), and point symmetry group (S). The topological properties consist of ferromagnetic imperfection (I_F) and antiferromagnetic imperfection of bipartite supercells (I_B). Hence, we are able to list all the unique supercells underlaying two dimensional square lattice that consists of N lattice points with their classification from the criteria of Betts *et al.* [2, 3]. Finally, we discussed how to construct the Tight-binding models for a supercell using periodic, anti-periodic, and open boundary conditions.

Bibliography

[1] O. Haan, J.-U. Klaetke, and K.-H. Mütter, "Ground-state staggered magnetization of the antiferromagnetic heisenberg model," *Phys. Rev. B*, vol. 46, pp. 5723–5726, Sep 1992.

[2] D. D. Betts, S. Masui, N. Vats, and G. E. Stewart, "Improved finite-lattice method for estimating the zero-temperature properties of two-dimensional lattice models," *Canadian Journal of Physics*, vol. 74, no. 1-2, pp. 54–64, 1996.

[3] D. D. Betts, H. Q. Lin, and J. S. Flynn, "Improved finite-lattice estimates of the properties of two quantum spin models on the infinite square lattice," *Canadian Journal of Physics*, vol. 77, no. 5, pp. 353–369, 1999.

[4] J. Bednorz and K. Müller, "Possible high T_c superconductivity in the Ba-La-Cu-O system," *Zeitschrift für Physik B Condensed Matter*, vol. 64, no. 2, pp. 189–193, 1986.

[5] E. Koch, "Quantum cluster methods," in Pavarini *et al.* [6], pp. 8.1–8.37.

[6] E. Pavarini, E. Koch, D. Vollhardt, and A. Lichtenstein, eds., *DMFT at 25: Infinite Dimensions*, August 2014.

[7] D. Micciancio, "CSE206A: Lattices algorithms and applications (spring 2014)." http://cseweb.ucsd.edu/classes/sp14/cse206A-a. accessed: 2015-05-25.

[8] M. M. Bugeanu, "Simulations of strongly correlated materials: Clusters and DMFT using a Lanczos solver," January 2012. Record converted from JUWEL: 18.07.2013.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Prof. Dr. Erik Koch and my co-supervisor Prof. Dr. Eva Pavarini for the continuous support of my master thesis, for the patient, enthusiasm, immense knowledge, insightful ideas, and who always guide me to think and solve problems systematically.

Many thanks to my friends and family, who always supporting, cheering, and giving companies that also bring joys and memories into my thesis writing process. I would like to thank to my beloved parents Susanah and Sipan Andi Mulyono, for all the motivations and the prays. A special thanks to Evgenii Sovetkin for the English proofreading and for all the joys and supports in the last moments of writing. Also a special thanks for Qian Zhang and Thi Hoai for studying, working and having lunch together.

Lastly, I would like to thank to Evangelische Studierenden Gemeinde Aachen and Katholische Hochschulgemeinde Aachen for providing financial support during my thesis writing.

Declaration of authorship

I, Cica Gustiani, hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Aachen, 27 May 2015

Cica Gustiani