2014-10-08

Lattice Isometries and Short Vector Enumeration

Meissen, Rebecca

UNIVERSITY OF CALGARY

Lattice Isometries and Short Vector Enumeration

by

Rebecca Meissen

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF MATHEMATICS AND STATISTICS

CALGARY, ALBERTA

September, 2014

Abstract

We present an implementation in Sage and overview of several algorithms for integer lattices. The first builds an isometry between lattices by considering partial maps. It also determines whether two lattices are nonisometric by exhaustively searching all possible maps. Using this algorithm, we give a method for computing the automorphism group of a lattice using strong generating sets. These both make use of the set of small vectors of a lattice, which can be enumerated using the last algorithm we present. We discuss the use of determining isometry classes as part of computing the Smith-Minkowski-Siegel mass formula.

Acknowledgements

I would like to thank my supervisor, Mark Bauer, for his guidance and support. Thanks to my committee members Clifton Cunningham, beating me to the punchline every time, and Mike Jacobson, a friendly perspective from computer science. Matthew Greenberg provided me with the original topic for my thesis, for which I am very grateful. I would also like to thank my many friends, both from the University of Calgary and from elsewhere, without whom I would surely have lost my mind.

Table of Contents

List of Tables

List of Algorithms

Chapter 1

Introduction

Lattices arose as a mathematical curiosity out of problems involving integer solutions to single and multivariate polynomials. These date back as early as the 18th century, when Joseph-Louis Lagrange considered lattice basis reductions in two dimensions [16]. Shortly after, Carl Friedrich Gauss made progress on what came to be known as Gauss' Circle Problem [11] which asks for the number of points inside a ball centered at the origin which satisfy a specific quadratic form. Although many other contributions to the field were made, it was not until 1910 that Minkowski published the seminal *Geometry of Numbers* [19], after which many subsequent texts followed suit to introduce the field in a more approachable manner.

Then in 1982, Arjen Lenstra, Hendrik Lenstra and László Lovász published a highly efficient algorithm [17] to reduce lattice bases, known as LLL reduction. This method found applications in many computational areas, notably in attacks on cryptographic systems. Now in the late 20th and early 21st centuries, lattices are being used to both design and attack cryptographic systems. Their structures also translate to problems in the physical sciences, such as crystallography and material science.

Many computational problems and algorithms call for the use of a collection of short vectors, including the Shortest Vector Problem. In 1985 mathematicians Fincke and Pohst published a method [10] of generating all lattice elements whose norm was less than a given constant. Independently, Kannan published a similar method [14], which is now known as Kannan-Fincke-Pohst enumeration (or KFP enumeration for short). Shortly after, Pohst worked with Plesken to develop methods [22] to construct lattices whose shortest vectors adhered to a given minimum. In 1997, Plesken and Souvignier published an algorithm [23]

for computing isometries between lattices which made use of these short vectors.

Being able to test whether or not two lattices are isometric allows one to construct isometry classes of lattices. With the help of some other tools, we can even verify that we are not missing any isometry classes, which we will discuss in Chapter 5. This is useful in the study of modular forms, which can be viewed as functions on these isometry classes.

Currently such algorithms are available in proprietary software such as Magma [18]. Sage [24] is an open-source software designed with algebraic and number theoretic problems in mind. The goal of this project is to add functionality to Sage by contributing algorithms which enumerate short vectors, find isometries, and compute automorphism groups of lattices.

Some background information on lattices is presented in Chapter 2. This information is used in Chapter 3, which covers the main algorithm for finding isometries between lattices. It also illustrates how to use the methods described for finding an isometry to also compute the automorphism group of a lattice. A necessary component of these is the enumeration of the short vectors within a lattice. An algorithm to compute these is presented in Chapter 4. Lastly, Chapter 5 shows how to apply these computations to the Smith-Minkowski-Siegel Mass Formula to verify the presence of all isometry classes within a collection of lattices.

Each algorithm is broken into pieces which may be run and tested independently. Pseudocode is given for each piece. All code is written in Sage and can be found in Appendix A.

Chapter 2

Background

A lattice can be thought of in several different ways, some of which will be more useful to us than others. One visualization might be the regularly spaced points on an n-dimensional grid. We might also describe a lattice as the integral span of a set of basis vectors. A formal definition is given below.

Definition 1 *A **lattice** $\mathcal{L}$ is a discrete subgroup of a vector space formed by the integer linear combination of basis vectors $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, which span the vector space.*

We will be using lattices specifically in $\mathbb{R}^n$ with integer coefficients.

$$\mathcal{L} = \left\{ \sum_{i=1}^{n} x_i \mathbf{b_i} \mid x_i \in \mathbb{Z} \right\}$$

Definition 2 *If we collect the basis vectors into the columns of a matrix B, we can express a lattice element $\mathbf{v}$ as*

$$\mathbf{v} = B\mathbf{x}$$

*for some $\mathbf{x} \in \mathbb{Z}^n$. We call $\mathbf{v}$ the **embedded vector** and $\mathbf{x}$ the **coordinate vector**.*

Example 3 *As an example, take the lattice formed by the basis $\{[2,1,0],[0,1,0],[0,3,3]\}$.*

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 1 & 3 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 3 \end{bmatrix}$$

Here $[1,0,1]$ is a coordinate vector, and $[2,4,3]$ is the associated embedded vector.

Definition 4 *A **bilinear form** on a lattice $\mathcal{L}$ is a function $\Phi\colon \mathcal{L} \times \mathcal{L} \to \mathbb{Z}$ satisfying linearity in each input variable.*

- $\Phi(\mathbf{u}, \mathbf{v} + \mathbf{w}) = \Phi(\mathbf{u}, \mathbf{v}) + \Phi(\mathbf{u}, \mathbf{w})$
- $\Phi(\mathbf{u} + \mathbf{v}, \mathbf{w}) = \Phi(\mathbf{u}, \mathbf{w}) + \Phi(\mathbf{v}, \mathbf{w})$
- $\Phi(a\mathbf{u}, \mathbf{w}) = a\Phi(\mathbf{u}, \mathbf{w})$
- $\Phi(\mathbf{u}, a\mathbf{w}) = a\Phi(\mathbf{u}, \mathbf{w})$

for all $\mathbf{u}, \mathbf{v}$*, and* $\mathbf{w}$ *in* $\mathcal{L}$ *and any scalar* a*.*

In particular, given a basis we can define a specific bilinear form Φ on our lattice $\mathcal{L}$ as part of its structure. In the case of integral lattices, we have $\Phi\colon \mathcal{L} \times \mathcal{L} \to \mathbb{Z}$. This form describes a kind of distance between elements $\mathbf{x}$ and $\mathbf{y}$ of the lattice defined by $\Phi(\mathbf{x}, \mathbf{y})$.

Definition 5 *A* ***quadratic form*** *is a homogeneous polynomial of degree 2. A form Q is called* ***positive definite*** *if $Q(\mathbf{x})$ is strictly positive for any nonzero $\mathbf{x}$.*

A lattice is called ***positive definite*** *if its quadratic form is positive definite. Similarly, a lattice is called* ***even*** *(or sometimes* ***type II****) if $Q(\mathbf{x})$ is always an even number. A lattice is called* ***odd*** *(or* ***type I****) if it is not even.*

The bilinear form has an associated quadratic form $Q\colon \mathcal{L} \to \mathbb{Z}$, which is simply defined by $Q(\mathbf{x}) = \Phi(\mathbf{x}, \mathbf{x})$. If the lattice $\mathcal{L}$ comes equipped with a bilinear form Φ, we will denote it as $(\mathcal{L}, \Phi)$.

The bilinear forms (and their associated quadratic forms) that we will be using will come from the usual inner product on vectors in $\mathbb{R}^n$, also known as the dot product, $\mathbf{u} \cdot \mathbf{v}$ for embedded vectors, and multiplication with the basis matrix for coordinate vectors. That is, if $\mathbf{u} = B\mathbf{x}$ and $\mathbf{v} = B\mathbf{y}$ for a basis B, we have $\Phi(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T B^T B\mathbf{y}$.

Example 6 *Continuing from Example 3, we can define a bilinear form on the elements of*

our lattice. Consider two coordinate vectors $\mathbf{x} = [x_1, x_2, x_3]$ *and* $\mathbf{y} = [y_1, y_2, y_3]$.

$$\varphi(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T \begin{bmatrix} 5 & 1 & 3 \\ 1 & 1 & 3 \\ 3 & 3 & 18 \end{bmatrix} \mathbf{y}$$

$$= y_1(5x_1 + x_2 + 3x_3) + y_2(x_1 + x_2 + 3x_3) + y_3(3x_1 + 3x_2 + 18x_3)$$

Note that this is determined from the basis matrix B*, so that the form is described by* $B^T B$*. The result is the inner product of the two embedded vectors corresponding to our coordinate vectors.*

The associated quadratic form is defined by the bilinear form on one element. For example, on $\mathbf{x}$ *we have:*

$$\varphi(\mathbf{x}) = 5x_1^2 + x_2^2 + 18x_3^2 + 2x_1x_2 + 6x_1x_3 + 6x_2x_3$$

This sends the coordinate vector $[1, 0, 1]$ *to* 29*, which is result of the inner product of* $[2, 4, 3]$ *with itself.*

This quadratic form specified by the basis is considered a norm on the lattice elements. If $\mathbf{v}$ is an embedded vector of the lattice $\mathcal{L}$, its norm is typically given by this quadratic form defined on $\mathcal{L}$.

Definition 7 *If* $\mathbf{v} = B\mathbf{x}$*, the* ***norm*** *of the embedded vector* $\mathbf{v}$ *is defined by the quadratic form. We will be using the inner product* $\mathbf{v} \cdot \mathbf{v}$*. The norm of the coordinate vector* $\mathbf{x}$ *is then* $\mathbf{x}^T B^T B\mathbf{x}$ *since*

$$\mathbf{v}^T\mathbf{v} = (B\mathbf{x})^T(B\mathbf{x}) = \mathbf{x}^T B^T B\mathbf{x}$$

Notice that this is also $\mathbf{x}^T A\mathbf{x}$, where $B^T B = A$. Here A is an example of the Gram matrix of the lattice.

Definition 8 *The* ***Gram matrix*** *of a lattice with basis* B *with respect to a bilinear form* Φ *is defined to be the matrix* A *with entries* $a_{ij} = \Phi(\mathbf{b}_i, \mathbf{b}_j)$.

Example 9 *Using the bilinear form from Example 6, our Gram matrix would be*

$$B^T B = \begin{bmatrix} 5 & 1 & 3 \\ 1 & 1 & 3 \\ 3 & 3 & 18 \end{bmatrix}$$

since our bilinear form was defined as the inner product of the embedded vectors. This matrix describes the form on the coordinate vectors.

We also saw that the associated quadratic form evaluated on the coordinate vector $[1, 0, 1]$ *is* 29*. This is the norm of the coordinate vector. It is also the norm of the embedded vector associated to the coordinate vector* $[1, 0, 1]$*, which is* $[2, 4, 3]$*.*

The bilinear form on $\mathcal{L}$ can be written with respect to either embedded or coordinate vectors. Using another basis to express the lattice elements is possible, and sometimes preferable. But the Gram matrix is specific to the bilinear form on the lattice, and should not change when operating on embedded vectors. If it is operating on coordinate vectors, the change of basis must be accounted for.

We can even define a lattice without specifying a basis at all, although a basis can be obtained once a description is given. In addition, although we use the usual inner product for our bilinear forms, there are many other kinds. One classification of such forms is the Hermitian form. Although our lattices are in $\mathbb{R}^n$, the general form of this example is in $\mathbb{C}^n$.

Definition 10 *A* ***Hermitian form*** *on a vector space over* $\mathbb{C}$ *is a map* $f \colon V \times V \to \mathbb{C}$ *satisfying:*

1. $f(a\mathbf{u} + b\mathbf{v}, \mathbf{w}) = af(\mathbf{u}, \mathbf{w}) + bf(\mathbf{v}, \mathbf{w})$
2. $f(\mathbf{u}, a\mathbf{v} + b\mathbf{w}) = \overline{a} f(\mathbf{u}, \mathbf{v}) + \overline{b} f(\mathbf{u}, \mathbf{w})$
3. $f(\mathbf{u}, \mathbf{w}) = \overline{f(\mathbf{w}, \mathbf{u})}$

for $\mathbf{u}, \mathbf{v}, \mathbf{w} \in V$ *and* $a, b \in \mathbb{C}$*.*

This definition yields an example of a bilinear form characterized by a matrix.

Definition 11 *A matrix in* ***Hermite Normal Form*** *is an upper triangular matrix with positive entries along the diagonal, and smaller nonnegative entries above each diagonal entry. Specifically it must satisfy the following properties:*

- *any rows of all zeros are at the bottom*
- *each pivot (leading nonzero) entry of a row is positive*
- *below each pivot entry are only zeros*
- *above each pivot entry are smaller, nonnegative values*

Example 12 *A Hermitian form on $\mathbb{C}^n$ can be given by a matrix H in Hermite Normal Form.*

$$f(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T H \overline{\mathbf{y}} = \sum_i \sum_j h_{ij} x_i \bar{y_j}$$

This defines a bilinear form with the vectors $\mathbf{x}$ and $\mathbf{y}$ as input. The coefficients of the form are determined by the entries h_{ij} in the matrix H when the expression above is expanded.

The matrix associated with the typical inner product $\mathbf{x} \cdot \mathbf{y}$ is simply the identity matrix.

A bilinear form on a lattice provides a sense of structure. Maps between lattices may or may not preserve this structure. That is, for a map $\varphi\colon (\mathcal{L}, \Phi) \to (\mathcal{J}, \Psi)$, it is not always the case that $\Phi(\mathbf{x}, \mathbf{y})$ will yield the same result as $\Psi(\varphi(\mathbf{x}), \varphi(\mathbf{y}))$.

Definition 13 *An* ***isometry*** *is a $\mathbb{Z}$-linear bijection φ between two lattices $(\mathcal{L}, \Phi)$ and $(\mathcal{J}, \Psi)$ that respects the bilinear form. That is, $\Phi(\mathbf{x}, \mathbf{y}) = \Psi(\varphi(\mathbf{x}), \varphi(\mathbf{y}))$ for all $\mathbf{x}$ and $\mathbf{y}$ in $\mathcal{L}$.*

If such a map exists, the two lattices are said to be ***isometric****.*

An automorphism of a lattice $(\mathcal{L}, \Phi)$ must preserve its bilinear form. We can try to understand the symmetry of a lattice through its automorphism group, which we denote $Aut(\mathcal{L})$. The automorphism group of a lattice is the set of all automorphisms $\varphi\colon (\mathcal{L}, \Phi) \to (\mathcal{L}, \Phi)$ with composition as the group action. This group is made up of isometries from the lattice to itself.

2.1 Matrix Decomposition

Many lattice methods deal with specific matrix decompositions or forms. These are useful because of the breadth of algorithms already available to work with matrices. In particular, we have already mentioned Hermite Normal Form in Example 12. This type of form is also used in the isometry finding algorithm in Chapter 3.

2.1.1 Hermite Normal Form

We have already seen the Hermite Normal Form from Definition 14. We will use it in our isometry finding algorithm to compare sets of vectors by comparing the Hermite Normal Form of matrices formed by these vectors.

For an integer-valued matrix A, the associated matrix H in Hermite Normal Form can be obtained by a series of row reductions represented by unimodular matrices. The matrix H is the unique matrix that can be obtained through this method. It can be helpful to keep the transformation matrix X after computing $H = XA$.

Since the Hermite Normal Form is calculated with respect to rows, if basis elements are represented as columns simply transpose the matrix A first to make use of algorithms which find the Hermite Normal Form. In particular, we will later be comparing the reduced form of matrices constructed from sums of vectors. Since the reduced matrix H is unique, we can expect the reduction to yield the same matrix H for suitably similar matrices which differ by only a unimodular matrix.

2.1.2 Cholesky Decomposition

To enumerate the short vectors of a lattice, we make use of the Cholesky Decomposition to deconstruct the quadratic form.

Definition 14 *The **Cholesky Decomposition** of a symmetric, positive definite matrix A is a lower triangular matrix L with positive entries along the diagonal, and smaller nonneg-*

ative entries below each diagonal entry. Specifically it must satisfy the following properties:

- *any rows of all zeros are at the top*
- *each pivot (leading nonzero) entry of a row is positive*
- *above each pivot entry are only zeros*
- *below each pivot entry are smaller, nonnegative values*
- $A = LL^T$

Similarly, we can express this as $A = R^T R$ for an upper triangular matrix $R = L^T$, which we will use in Chapter 4.

In order to make use of the Cholesky Decomposition, we must restrict ourselves to certain types of lattices. We require that the matrix A representing our bilinear form be symmetric and positive definite, since the Cholesky Decomposition is only guaranteed to exist for these types of matrices.

It is possible to apply these methods to bilinear forms in general. However, since we only consider the inner product as our bilinear form, we can use the basis to express the form on coordinate vectors with the Gram matrix. Using the associated basis B for our lattice, the Gram matrix is $A = B^T B$, so it is indeed symmetric. To be positive definite, the associated quadratic form must be positive definite. That is, for a nonzero vector $\mathbf{x}$, we must have $\mathbf{x}^T A\mathbf{x}$ positive as well. Here L does not necessarily have to be equal to the original matrix B, but rather $B^T B = LL^T$.

The process for finding the upper Cholesky Decomposition matrix R with entries r_{ij} from a matrix A with entries a_{ij} is done one row at a time using the following formulas

$$r_{ii} = \sqrt{a_{ii} - \sum_{j=1}^{i-1} r_{ij}^2}$$
$$r_{ij} = \frac{1}{r_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} r_{ik} r_{jk} \right)$$

2.2 Hard Problems

There are several computationally hard problems associated with lattices. These mainly involve finding the smallest elements of the lattice. The symmetry of a lattice might imply that these problems are easy, whereas in reality they are quite difficult. Although a lattice might be easy to express in terms of its basis vectors, there are no known algorithms which solve the following problems in polynomial time. This refers to the number of operations, such as multiplications or comparisons, performed as a function of the size of the input, which is typically a basis of a lattice. We measure the size of the basis by its dimension. So if our input is a basis of dimension n, a polynomial time algorithm might perform $n^2 + n$ operations. An algorithm which runs in polynomial time will perform a number of operations that can be expressed as a polynomial of n. Current known algorithms which attempt to solve these problems, however, perform an exponential number of operations in this regard. An algorithm which runs in exponential time might perform 2^n operations.

Given a lattice $\mathcal{L}$ and an arbitrary point $\mathbf{x}$, both in $\mathbb{R}^n$, what is the closest element of the lattice to $\mathbf{x}$? That is, find

$$\mathbf{y}_{min} = \min_{\mathbf{y} \in \mathcal{L}} (||\mathbf{x} - \mathbf{y}||)$$

Note that here distance is typically given by the typical Euclidean norm in $\mathbb{R}^n$, as that is the original formulation of the problem. This holds true for the quadratic forms we will

evaluate, however. This problem is known as the *Closest Vector Problem*, often abbreviated CVP. Another problem, called the *Shortest Vector Problem*, is phrased similarly.

Given a lattice $\mathcal{L}$ in $\mathbb{R}^n$, find the shortest nonzero lattice element. That is, find

$$\mathbf{x}_{min} = \min_{\mathbf{x} \in \mathcal{L}}(||\mathbf{x} - 0||)$$

where $\mathbf{x}$ is not the zero vector.

This is really a variant on the Closest Vector Problem, with the origin standing in for the arbitrary point in $\mathbb{R}^n$. It stands to reason that any solution to the Closest Vector Problem will be a valid solution to the Shortest Vector Problem as well. A solution to the Shortest Vector Problem may also allow us to find a solution of the Closest Vector Problem by using our short vector to define a small space to search for close vectors. Though these problems were originally formulated for lattices in $\mathbb{R}^n$ under the Euclidean metric, they apply to other lattices as well. Later in Chapter 4 we will explore an algorithm which enumerates all of the shortest vectors of a lattice with respect to its inherent quadratic form.

2.2.1 LLL Reduction

These problems typically involve many computations involving the vectors of the lattice. Computationally speaking, it is usually better to have a basis comprised of small, orthogonal vectors. This keeps numbers from getting too large during routine computations, such as sums and products of the entries in our vectors, and lessens the need for lots of memory. The Lenstra-Lenstra-Lovász basis reduction algorithm (often abbreviated as LLL reduction) provides an orthogonal (but not orthonormal) basis whose vectors' norms are reasonably bounded. Furthermore, it does this in polynomial time.

Because of its efficiency, the LLL algorithm is often used to reduce otherwise large bases to smaller, more manageable bases. An outline and analysis can be found in [5].

Chapter 3

Isometry Finding

The definitions from Chapter 2 will guide the exploration in this chapter. Specifically we will be using integer lattices in $\mathbb{R}^n$ equipped with a bilinear form which is specified with a basis. Recall from Definition 4 that a bilinear form $\Phi\colon \mathcal{L} \times \mathcal{L} \to \mathbb{Z}$ on an integral lattice $\mathcal{L}$ also specifies a quadratic form $Q\colon \mathcal{L} \to \mathbb{Z}$ via $Q(\mathbf{x}) = \Phi(\mathbf{x}, \mathbf{x})$. As we search for an isometry, we must check that these forms are preserved.

What follows is a framework for computing an isometry between two lattices $(\mathcal{L}, \Phi)$ and $(\mathcal{J}, \Psi)$, each equipped with its own bilinear form. From there, we give a method for computing the automorphism group of a lattice $(\mathcal{L}, \Phi)$ using strong generating sets. These methods rely heavily on the use of a set of relatively small vectors, which we will denote S. Enumerating these short vectors is the primary topic of Chapter 4. This chapter explores the methods of [23] for computing lattice isometries and automorphisms. For future consideration, [1] is listed as a wishlist of future functionality for Sage.

Runtimes are given for the major algorithms. These times reflect computations on integer lattices using a Pentium dual-core 2.20GHz processor with 3GB of 667mhz DDR2 RAM.

3.0.2 Definitions

Given a set of basis vectors collected into a matrix $\mathbf{B} = (\mathbf{b}_1, \cdots, \mathbf{b}_n)$, we regard the lattice $\mathcal{L}$ as the set of all integer linear combinations of the basis vectors

$$\mathcal{L} = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i \ : \ x_i \in \mathbb{Z} \right\} = \{ \ B\mathbf{x} \ : \ \mathbf{x} \in \mathbb{Z}^n \ \}$$

equipped with a (positive definite) bilinear form $\Phi\colon \mathcal{L} \times \mathcal{L} \to \mathbb{Z}$, and a Gram matrix F with respect to Φ. That is, F is made up of the pairwise inner products of the basis vectors, $F_{ij} = \Phi(\mathbf{b}_i, \mathbf{b}_j)$. This inner product is given by the dot product of the two embedded lattice

elements. The norm of a lattice element $\mathbf{x}$ is defined to be the inner product $\Phi(\mathbf{x}, \mathbf{x})$. If we regard the basis elements as columns of a matrix, we can see that the Gram matrix F is $B^T B$. Alternatively, since the elements of the lattice can be represented by their coordinate vectors, we can define the inner product in terms of the basis. That is, if $\mathbf{u} = B\mathbf{x}$ and $\mathbf{v} = B\mathbf{y}$ are elements of the lattice, we can define their inner product by

$$\Phi(\mathbf{u}, \mathbf{v}) = \mathbf{u} \cdot \mathbf{v} = \mathbf{x}^T B^T B \mathbf{y}$$

For basis elements $\mathbf{b}_i$ and $\mathbf{b}_j$ we would then have

$$\Phi(\mathbf{b}_i, \mathbf{b}_j) = \mathbf{b}_i \cdot \mathbf{b}_j = \mathbf{e}_i^T B^T B \mathbf{e}_j = \mathbf{e}_i^T F \mathbf{e}_j = F_{ij}$$

Here $\mathbf{x}$ and $\mathbf{y}$ are known as the coordinate vectors of $\mathbf{u}$ and $\mathbf{v}$ with respect to the basis B. Vectors written will be coordinate vectors unless otherwise specified, and evaluated with respect to a chosen ordered basis. Although we can find multiple bases that yield the same lattice, we assume that one has been fixed initially. When searching for an isometry, we will attempt to construct alternative bases from a supply of candidate vectors. If found, our isometry will map the basis B to a new basis D which preserves the bilinear form.

3.1 Preprocessing

A vector is considered short if its norm is less than a prescribed maximum. Usually this maximum is given by the norm of the largest basis vector, which also appears in the Gram matrix as $\max_{1 \leq i \leq n}(F_{ii})$. We define the set of small vectors in $\mathcal{L}$ more formally as

$$S = \left\{ \mathbf{v} \in \mathcal{L} \ : \ \Phi(\mathbf{v}, \mathbf{v}) \leq \max_{1 \leq i \leq n}(F_{ii}) \right\}$$

This set of small vectors is built up in Chapter 4 by first deconstructing the bilinear form Φ and recursively searching for vectors with small inner products. Notice that this set will be finite, since our lattices are finite-dimensional and our coordinate vectors can take on only integer values.

To define an isometry between lattices, it suffices to define a good image of the basis vectors. From there, the image of an arbitrary element will be defined, as each element is a linear combination of the basis vectors. If we consider two lattice elements, $B\mathbf{x}$ and $B\mathbf{y}$, their inner product is defined by

$$\mathbf{x}^T B^T B\mathbf{y} = (B\mathbf{x}) \cdot (B\mathbf{y}) = \left(\sum_{i=1}^{n} x_i \mathbf{b}_i\right) \cdot \left(\sum_{j=1}^{n} y_j \mathbf{b}_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} x_i y_j \mathbf{b}_i \cdot \mathbf{b}_j$$

Say our map sends each basis vector $\mathbf{b}_i$ to a new vector $\mathbf{d}_i$, and that the inner product is preserved on these basis vectors. That is, $\mathbf{b}_i \cdot \mathbf{b}_j = \mathbf{d}_i \cdot \mathbf{d}_j$. Then the above can be equivalently written as

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_i y_j \mathbf{d}_i \cdot \mathbf{d}_j = \mathbf{x}^T D^T D\mathbf{y}$$

which is the inner product of $\mathbf{x}$ and $\mathbf{y}$ with respect to a second lattice. The columns of the matrix D are the basis vectors of this second lattice.

Since our goal is to build up a complete basis $(\mathbf{v}_1, \cdots, \mathbf{v}_n)$ from our candidate vectors in S, we choose a potential image for each basis vector under the isometry and test the partial image $(\mathbf{v}_1, \cdots, \mathbf{v}_k)$ using the bilinear form to see if the inner products have been preserved. Since we are building an isometry from one lattice to another, we will be taking our image vectors from the set of small vectors within the second lattice. If the inner products are preserved, we call this a k-partial automorphism, though it may be part of the construction of an isometry and not an automorphism.

Definition 15 *A* **k-*partial automorphism*** *is a partial map* $(\mathbf{v}_1, \cdots, \mathbf{v}_k)$ *which sends* $\mathbf{b}_i$ *to* $\mathbf{v}_i$ *for* $i \leq k$*, satisfying* $\Phi(\mathbf{v}_i, \mathbf{v}_j) = F_{ij}$ *for all* $i, j \leq k$*.*

The next step is to choose $\mathbf{v}_{k+1}$ from our small vectors within our second lattice and check the additional inner products.

It is important to note that not all k-partial automorphisms can be extended to $(k+1)$-partial automorphisms. However, [22] claims that we can use this fact to our advantage. The number of possible extensions will be preserved under automorphisms. We will store

the number of possible extensions in a matrix, called the fingerprint, and use it to rule out partial automorphisms that are not likely to extend to full automorphisms.

3.1.1 Fingerprint

The fingerprint stores information about extending k-partial automorphisms to $(k+1)$-partial automorphisms. To compute each entry f_{ki}, we count the number of vectors from S that have the same inner product as the i-th basis vector with a subset of the other basis vectors.

Definition 16 *The **fingerprint** is an upper-triangular matrix denoted by f, defined by*

$$f_{ki} = |\{\mathbf{v} \in S : \Phi(\mathbf{v}, \mathbf{v}) = \Phi(\mathbf{b}_i, \mathbf{b}_i) \text{ and } \Phi(\mathbf{v}, \mathbf{b}_j) = \Phi(\mathbf{b}_i, \mathbf{b}_j) \text{ for } j = 1, \dots k-1\}|$$

It will not be used for any matrix algebra. However, the matrix structure is convenient for storing values. An entry f_{ki} stores the number of candidate vectors that can act as the image of the basis vector $\mathbf{b}_i$ with respect to the first $k-1$ basis vectors. Each entry stores the calculated values for the number of extensions of a k-partial automorphism of $(\mathcal{L}, \Phi)$ to a $(k+1)$-partial automorphism. To test the number of these extensions, we let our k-partial automorphism simply be the first k basis vectors of $(\mathcal{L}, \Phi)$. Recall that the i-th vector in our k-partial automorphism represents the image of the i-th basis vector under our desired isometry.

For example, consider the partial map on a 4-dimensional lattice given by

$$(\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3) \to (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$$

This map sends $\mathbf{b}_1$ to $\mathbf{v}_1$, $\mathbf{b}_2$ to $\mathbf{v}_2$, and $\mathbf{b}_3$ to $\mathbf{v}_3$. However, it does not specify an image for the fourth basis vector $\mathbf{b}_4$. In fact, there are likely many choices as to how to complete this partial map by assigning an image to the last basis vector. If our partial map satisfies

the conditions of being a 3-partial automorphism, however, we cannot guarantee that the complete map (after choosing an image for $\mathbf{b}_4$) will be a 4-partial automorphism.

At each stage we check to see whether our partial map is a k-partial automorphism. For $(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k)$, we must check that $\Phi(\mathbf{v}_i, \mathbf{v}_j) = \Phi(\mathbf{b}_i, \mathbf{b}_j)$, which is the same as checking that $\Phi(\mathbf{v}_i, \mathbf{v}_j) = F_{ij}$ for values of i and j up to k. After verifying that a partial map is indeed a k-partial automorphism, we take a candidate vector $\mathbf{v}$ from S and test it as an extension to a $(k+1)$-partial automorphism. That is, we choose a candidate vector to be the image of the $(k+1)$-th basis vector and check its inner products with the other vectors. This means that we must verify that $\Phi(\mathbf{v}_i, \mathbf{v}) = \Phi(\mathbf{b}_i, \mathbf{b}_{k+1})$ for $i = 1, \ldots, k$.

If the inner products are preserved, then we say that there is at least one extension of a k-partial automorphism to a $(k+1)$-partial automorphism. Out of all the vectors in S, some may work as a viable candidate to extend the partial map, and some may not. We are interested in the number of vectors which will work as an image for $\mathbf{b_{k+1}}$ with the partial map $(\mathbf{b}_1, \mathbf{v}_2, \ldots, \mathbf{b}_k)$. By testing every candidate vector in S, we will know how many different extensions are possible. The number of such extensions is stored in the fingerprint.

For a lattice of rank 4, for example, the entry f_{23} would count small vectors $\mathbf{v}$ with the same norm as $\mathbf{b}_3$ and matching inner products with $\mathbf{b}_1$.

$$\Phi(\mathbf{v}, \mathbf{v}) = \Phi(\mathbf{b}_3, \mathbf{b}_3) = F_{33}$$

$$\Phi(\mathbf{v}, \mathbf{b}_1) = \Phi(\mathbf{b}_3, \mathbf{b}_1) = F_{31}$$

The entry f_{34} would count candidate vectors whose norm matched the basis vector $\mathbf{b}_4$ and whose inner products matched with $\mathbf{b}_1$, and $\mathbf{b}_2$. The entry f_{24} is similar to the entry f_{34}, except that it does not check for a correct inner product with $\mathbf{b}_2$. Since f is an upper-triangular matrix, entries below the diagonal will be 0. These entries correspond to basis vectors which we have already analyzed.

A different ordering of the basis vectors will yield a different fingerprint. Some orderings are better than others, in that they may lower computation time later when we can rule out

partial automorphisms faster. If there is a vector which only has a few possible images under any isometry, it behooves us to try to replace that vector first. A good ordering would check such vectors before other, less restrictive vectors. An optimal ordering cannot necessarily be known before computation, but we can try to optimize as we go. To optimize our ordering, we may reorder our basis vectors as we compute the fingerprint. After computing the k-th row of the fingerprint, check to see if f_{kk} is the minimal nonzero entry in the row. If it is, we do nothing. If it is not, we swap the k-th column of the fingerprint with whichever column contains the minimal nonzero entry in that row. We will swap the corresponding basis vectors, as well as the entries in any coordinate vectors in S.

Say we computed the first row of a fingerprint to be $[4, 5, 2]$. This tells us that there are four small vectors whose norm matches $\mathbf{b}_1$, five small vectors whose norm matches $\mathbf{b}_2$, and only two small vectors whose norm matches $\mathbf{b}_3$. Clearly then $\mathbf{b}_3$ will be the most restrictive choice. If we start our search with $\mathbf{b}_3$ instead of $\mathbf{b}_1$, there will be fewer possible partial maps to verify.

Alternatively, we can keep a running list of indices of the minimal nonzero entries in each row, and use that to compute the fingerprint. Instead of assigning zeros to columns in order from 1 to n, we assign them in order of these minimal entries. This results in a matrix which is not upper triangular, but keeps us from having to reorder our vectors.

In [23], f_{ki} is defined to be 0 when $i \in \mathcal{I}$, where $\mathcal{I}$ is a set of indices denoting the smallest entry in each row. Using this method, we would try to find an image for $\mathbf{b}_3$ first then, as it only has two possible replacements.

The smallest entry in our example row is 2, and so the index 3 would be added to $\mathcal{I}$. This tells us to find an image for $\mathbf{b}_3$ first, before proceeding to find images for the other basis vectors. The remaining entries of the third column of the fingerprint would be 0, since we will not be changing the image of $\mathbf{b}_3$ unless we exhaustively rule out all complete maps which use our current candidate. If we cannot complete our map to a full isometry, only

then will we backtrack and change our choice for the image of $\mathbf{b}_3$.

We use the values in the fingerprint to rule out k-partial automorphisms quickly. Testing a k-partial automorphism amounts to testing norms and inner products, as well as the number of continuations. Given a k-partial automorphism, we select small vectors from S and test them as continuations to a $(k+1)$-partial automorphism. If the number we find is different from the corresponding entry in the fingerprint, we know that our current k-partial automorphism will not work. We then throw out the last vector we had chosen from the previous step and try to extend a different k-partial automorphism.

Here we have an overview of how to compute the number of continuations for a k-partial automorphism, as well as an outline of how to compute the fingerprint. Below these is an example, which can be used to test the algorithms.

Algorithm 1 Number Of Continuations

1: **procedure** NUMCONT(B, F, S, k)

Input: basis B, Gram matrix F, small vectors S, index k

Output: number of continuations *count*

2: **for** $\mathbf{v} \in S$

3: **if** $\Phi(\mathbf{v}, \mathbf{v}) = F_{kk}$

4: **if** $\Phi(\mathbf{v}, \mathbf{b}_j) = F_{kj}\ \forall j = 1, \ldots, k-1$

5: count++

6: **return** count

Algorithm 2 Fingerprint

1: **procedure** FINGERPRINT(B, F, S)

Input: basis B, Gram matrix F, small vectors S

Output: fingerprint f

2: **for** $k = 1 \dots n$ ▷ Compute each row at a time

3: **for** $i = 1 \dots n$

4: **if** $i \geq k$

5: $f_{ki} = \text{NUMCONT}(B, F, S, k)$ ▷ Count replacements for $\mathbf{b}_k$

6: **else**

7: $f_{ki} = 0$

8: **return** f

Example 17 *Consider the basis* $B = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$. *The largest basis element is* $[0, 2]^T$, *whose norm is 4. The (coordinate) vectors in the lattice spanned by* B *with norm less than 4 are:*

$$S = \{[1, 0], [-1, 0], [2, 0], [-2, 0], [0, 1], [0, -1], [0, 0]\}$$

Note that these are the coordinate vectors associated to the lattice elements. To see the element as it would appear in $\mathbb{R}^2$, *multiply the basis by the coordinate vector. For example the coordinate vector* $[0, -1]$ *is associated with the vector* $B[0, -1]^T = [0, -2]^T$ *in* $\mathbb{R}^2$.

To calculate the fingerprint matrix f, *begin with* f_{11}. *We must first check the norm of all of our candidates to see that it matches the norm of the first basis element. The first basis element is* $[1, 0]^T$ *which has norm 1. The vectors with this norm are* $[1, 0]$ *and* $[-1, 0]$. *Since this is the first row of the fingerprint, we need not check the inner product with the other basis elements. The number of appropriate candidates is therefore 2, so* $f_{11} = 2$.

Next, to find f_{12}, *we must find the vectors whose norm matches that of the second basis element. There are 4 of these, so* $f_{12} = 4$.

We automatically set $f_{21} = 0$, *since* $k = 2$ *(where* k *is taken from the definition above),*

so we would be counting the number of vectors with the same norm as the first basis vector who also have the same inner product with the basis vectors numbered up to $k-1$. *But in this case* $k-1=1$, *so we would be counting the vectors whose norm is the same as the first basis vector, which we have already done. The fingerprint* f *is an upper triangular matrix for this very reason.*

To find f_{22}, *we must check for vectors with the same norm as the second basis vector, but who also match the inner product of the second basis vector taken with the first. We know that 4 vectors have the appropriate norm*

$$\{[2,0],[-2,0],[0,1],[0,-1]\}$$

but only $[0,1]$ *and* $[0,-1]$ *have the same inner product with the first basis vector. So* $f_{22}=2$.

The fingerprint f *is then* $f=\begin{bmatrix}2 & 4\\ 0 & 2\end{bmatrix}$.

Example 18 *Continuing with our basis from above, consider the map which sends the first basis vector to* $[-1,0]$. *Is this a* 1*-partial automorphism? We first check the norm of* $[-1,0]$ *to make sure it is the same as the norm of the first basis vector. Since they match, we move on to check the fingerprint.*

How many ways can we extend this map to a 2*-partial automorphism? We must consider all the candidates which have not only the same norm as the second basis vector, but also the same inner product with the first basis vector. The candidates with the same norm are*

$$\{[2,0],[-2,0],[0,1],[0,-1]\}$$

Of these, only $[0,1]$ *and* $[0,-1]$ *have an inner product of* 0 *with our first vector. Since either can be chosen as the image of the second basis vector, we can extend our* 1*-partial automorphism in two possible ways. This matches the entry in the fingerprint, so we may proceed to select an image for the second basis vector and check the vector sums which are explained below.*

Taking the product of the diagonal entries of each row in the fingerprint gives an upper bound on the size of the automorphism group of $\mathcal{L}$, since the diagonal entry in row i of the fingerprint bounds the size of the orbit $\mathbf{b}_i G_i$, where G_i is the pointwise stabilizer of $\{\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}\}$. That is, G_i is the set of maps which fix each basis vector up to $\mathbf{b}_i$. These are maps $\varphi\colon \mathcal{L} \to \mathcal{L}$ with $\varphi(\mathbf{b}_k) = \mathbf{b}_k$ for $k = 1, \ldots, i-1$. This will become useful to us when we determine $Aut(\mathcal{L})$ later on.

Example 19 *An interesting case is the E_8 root lattice, so named because it is formed as the integral span of the roots of the E_8 Lie algebra. These roots are vectors in $\mathbb{R}^8$, comprised of either whole- or half-integers which sum to an even number, with a norm of 2. There are 8 such vectors, arranged in columns as a basis for the E_8 root lattice.*

$$\begin{bmatrix}
2 & -1 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & \frac{1}{2} \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & \frac{1}{2} \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & \frac{1}{2} \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & \frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & \frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & \frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2}
\end{bmatrix}$$

We calculate the following fingerprint matrix:

$$\begin{bmatrix} 240 & 240 & 2160 & 240 & 240 & 240 & 240 & 240 \\ 0 & 56 & 126 & 126 & 126 & 126 & 126 & 126 \\ 0 & 0 & 27 & 27 & 72 & 72 & 72 & 72 \\ 0 & 0 & 0 & 10 & 40 & 16 & 40 & 40 \\ 0 & 0 & 0 & 0 & 8 & 8 & 24 & 24 \\ 0 & 0 & 0 & 0 & 0 & 4 & 6 & 12 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

In this example, reordering was used. After completing each row, the minimal nonzero entry was noted. The basis was reordered, simply by swapping $\mathbf{b}_k$ *with* $\mathbf{b}_j$ *whenever* f_{kj} *is the minimal nonzero entry in the* k*-th row. This also means we swap corresponding entries in the previous rows of the fingerprint, as well as permuting the values in the short vectors we enumerated. (If they are embedded vectors, no swapping is necessary.)*

Taking the product of all the diagonal nonzero entries in each row, we find a bound on the size of the automorphism group

$$240 \cdot 56 \cdot 27 \cdot 10 \cdot 8 \cdot 4 \cdot 3 \cdot 2 = (2^4 \cdot 3 \cdot 5)(2^3 \cdot 7)(3^3)(2 \cdot 5)(2^3)(2^2)(3)(2) = 2^{14} \cdot 3^5 \cdot 5^2 \cdot 7$$

This is indeed the order of the automorphism group of the E_8 *root lattice.*

While the number of arithmetic operations and comparisons needed in calculating the fingerprint grows polynomially with respect to the size of S, this set of small vectors can often be quite large and not necessarily polynomial in size with respect to the basis. The Gram matrix may give an indication as to how many small vectors there are. If the norm of the smallest basis vector is much smaller than the norm of the largest basis vector, this could indicate that there will be many small vectors. This is because the norm of the largest basis vector will determine the upper bound on norms of small vectors. There can be many

combinations of the smaller basis vectors whose norms would still be well below the upper bound. If, however, the basis vectors all have relatively similar norms, then there may not be very many small vectors.

The fingerprint was calculated for the standard integer lattice in varying dimensions to gauge the runtime using the `timeit()` command in Sage. Below are the results in milliseconds corresponding to each lattice in dimension 2 through 6. For example, in dimension 2 we used the lattice generated from the basis $([1,0],[0,1])$. These results are consistent with what we expect. Since the short vectors of the lattice are required to compute the fingerprint, these short vectors must be enumerated before any other computation can occur. The enumeration causes the runtime to grow exponentially as the dimension increases. These results are consistent with our predictions, as the time necessary to compute the fingerprint (and enumeration) increases at an exponential rate as the dimension increases.

dimension	2	3	4	5	6	7	8
runtime (ms)	12	43.2	116	267	538	1010	1760

Table 3.1: Fingerprint Runtime

3.1.2 Vector Sums

We continue to test our k-partial automorphism by taking inner products between our image vectors and our small vectors. Since an isometry must preserve these inner products, we can check them against the inner products of the basis vectors with the small vectors.

Here we check the inner product of each $\mathbf{v}_i$ from our k-partial automorphism $(\mathbf{v}_1, \ldots, \mathbf{v}_k)$ with small vectors from S. Since we are searching for an isometry, we must verify that inner products are preserved. To verify this completely, it would suffice to check the inner products $\Phi(\mathbf{v}_i, \mathbf{v}_j)$ of vectors from a full map $(\mathbf{v}_1, \ldots, \mathbf{v}_n)$ against the inner products of the original basis vectors. But it becomes potentially time intensive to only check full maps, since S can be quite large. It is helpful to be able to rule out partial maps as soon as possible. We

can instead check that the inner product of a basis vector with a small vector from S is preserved.

$$\Phi(\mathbf{v}_i, \varphi(\mathbf{u})) = \Phi(\mathbf{b}_i, \mathbf{u})$$

When checking this condition, we must be sure to take small vectors from the second lattice to check with the images of the basis vectors. In practice, this is not done by specifying the images of each small vector from the first lattice. Instead, we store the value of $\Phi(\mathbf{b}_i, \mathbf{u})$ in the form of a lookup table and check against it using the following methods.

Let $B_k = (\mathbf{b}_1, \cdots, \mathbf{b}_k)$ be the subset of the basis consisting of the first k basis vectors. For each candidate vector $\mathbf{u}$, we create a list of inner products of $\mathbf{u}$ with each member of this subset.

$$\Phi(\mathbf{u}, B_k) = (\Phi(\mathbf{u}, \mathbf{b}_1), \ldots, \Phi(\mathbf{u}, \mathbf{b}_k))$$

We then create a lookup table of pairs $(\Phi(\mathbf{u}, B_k), \mathbf{u})$. If two vectors $\mathbf{u}$ and $\mathbf{u}'$ exist with the same list of inner products,$\Phi(\mathbf{u}', B_k) = \Phi(\mathbf{u}, B_k)$, update the entry to be $(\Phi(\mathbf{u}, B_k), \mathbf{u} + \mathbf{u}')$. In this way, we have a table of pairs $(\Phi(\mathbf{u}, B_k), \mathbf{w})$, where $\mathbf{w}$ is the sum of all candidate vectors $\mathbf{u}$ from S that have the same list of inner products $\Phi(\mathbf{u}, B_k)$.

Definition 20 *A **vector sum** is the sum of two or more vectors from the lattice $\mathcal{L}$, say $\mathbf{u}$ and $\mathbf{u}'$, for which*

$$\Phi(\mathbf{u}', B_k) = \Phi(\mathbf{u}, B_k)$$

for a list of vectors B_k and a bilinear form $\Phi\colon \mathcal{L} \times \mathcal{L} \to \mathbb{Z}$.

Here we call each element $\mathbf{w}$ a vector sum. In practice, we separate the vectors in S by their inner products with B_k. We do this by creating a lookup table which uses the inner product with B_k as the key. Each entry then contains a list of vectors which have identical inner products with B_k. After the vectors have been separated, we then compute the vector sums by adding the vectors which had identical inner products with B_k.

Algorithm 3 Lookup Table Of Vector Sums

1: **procedure** LOOKUPTABLE(B, S)

Input: basis B, small vectors S

Output: lookup table of vector sums T

2: **for** $\mathbf{v} \in S$

3: compute $\Phi(\mathbf{v}, B_k) = [\Phi(\mathbf{v}, \mathbf{b}_i)]_{i=1}^{k}$

4: **if** $\Phi(\mathbf{v}, B_k)$ is in T

5: $(\Phi(\mathbf{v}, B_k), \mathbf{u}) = (\Phi(\mathbf{v}, B_k), \mathbf{u} + \mathbf{v})$ ▷ Update the entry

6: **else**

7: add $(\Phi(\mathbf{v}, B_k), \mathbf{v})$ to T ▷ Create a new entry

8: **return** T

Example 21 *Consider the lattice spanned by the columns of the matrix* $\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$. *It has 27 small vectors. We start with* B_1*, which is simply the first basis vector. We group the small vectors of the lattice by their inner product with* B_1*. Some of these are given below, grouped by the value of their inner product with* $[1, 0]$*.*

$$(0) : [11, -5], [-11, 5], [0, 0] \qquad (-6) : [-10, 4], [1, -1]$$

$$(-1) : [-9, 4], [2, -1] \qquad (7) : [8, -3], [-3, 2]$$

$$(-2) : [-7, 3], [4, -2] \qquad (8) : [-5, 3], [6, -2]$$

$$(3) : [-6, 3], [5, -2] \qquad (9) : [4, -1]$$

$$(-4) : [8, -4], [-3, 1] \qquad (-10) : [-2, 0]$$

$$(5) : [-10, 5], [12, -5], [1, 0] \qquad (11) : [0, 1]$$

In the table above, each entry is of the form $\Phi(\mathbf{u}_1, B_1) : \mathbf{u}_1, \ldots, \mathbf{u}_n$ *where each* $\mathbf{u}_i$ *has the same inner product with* B_1*. The first entry here is the set of vectors whose inner product with* B_1 *is (0). When we reach the list of vectors corresponding to* $\Phi(\mathbf{u}_1, B_1) = (0)$*, represented by*

the list (0), we see the coordinate vectors $[11, -5]$*,* $[-11, 5]$*, and* $[0, 0]$*. The vector sum for this entry is then* $[11, -5] + [-11, 5] + [0, 0] = [0, 0]$*.*

Each vector sum is itself a vector. These vector sums will become the columns of a matrix, W_k. That is, $W_k = (\mathbf{w}_1, \ldots, \mathbf{w}_h)$ for some $h \leq |S|$, where each $\mathbf{w}_i$ is a column of W_k. This matrix will help to determine if an ordered list of k vectors chosen from S is a k-partial automorphism by checking to see if it yields the same vector sums as the partial basis.

Algorithm 4 Vector Sums

1: **procedure** VECTORSUMS(B, S, k)

Input: basis B, small vectors S, index k

Output: vector sum matrix W_k, transformation matrix X_k, reduced matrix A_k, reverse transformation matrix Y_k

2: $T = \text{LOOKUPTABLE}(B, S)$

3: $W_k = [w]_{(w,(\Phi(w,b_i)))\in T}$ ▷ Vector sums become columns of W

4: $A_k, X_k = \text{HNF}(W_k)$ ▷ Compute Hermite Normal Form of W

▷ Save transformation matrix X

5: Solve $W_k = A_k Y_k$ for Y_k

6: **return** W_k, X_k, A_k, Y_k

We then create a lattice from our k-partial automorphism. Recall that at each step, our partial automorphism is given by the image of a partial basis (in this case, the first k basis elements). The potential k-partial automorphism is then itself a partial basis. We must check to see if the lattice it generates is the same as the lattice generated by the first k basis elements.

$$\mathcal{H}_k = \left\{ \sum_{i=1}^{h} x_i \mathbf{w}_i \ : \ x_i \in \mathbb{Z} \right\}$$

and find a basis $A_k = (\mathbf{a}_1 \ldots \mathbf{a}_\ell)$ with $\ell \leq k$ such that

$$\mathcal{H}_k = \{ A_k \mathbf{x} \ : \ \mathbf{x} \in \mathbb{Z}^\ell \}$$

In practice we do this by reducing W_k to its Hermite Normal Form and saving the transformation matrix as X_k. Since A_k is constructed from the vectors in W_k, we have two expressions:

$$A_k = W_k X_k$$

$$A_k Y_k = W_k$$

As one might expect, the matrix A_k will have at most as many columns as W_k. It is preferable here to pare A_k down to a full-rank matrix, but if necessary we may leave the unnecessary columns in, provided we do not consider them when we express the columns of W_k as linear combinations of the columns of A_k.

For each $1 \leq k \leq n$ we must remember W_k, A_k, X_k, and Y_k computed from S and B_k with respect to $\mathcal{L}$. In practice, we can store these in an array:

$$((W_1, A_1, X_1, Y_1), \ldots, (W_n, A_n, X_n, Y_n))$$

In the future we will use this information to rule out k-partial automorphisms by checking their vector sums against the vector sums of a partial basis consisting of k elements. This is done by feeding in the k-partial automorphism as B_k and computing W_k using the candidates from S. Since the completed automorphism must preserve inner products, the list of inner products should also be preserved. If the vector sums do not match, then the k-partial automorphism will not be extended to a full automorphism.

3.2 Finding an Isometry

To begin searching for an isometry between two lattices, $(\mathcal{L}, \Phi)$ and $(\mathcal{J}, \Psi)$ with bases B and B_2 respectively, we first enumerate the short vectors of $\mathcal{L}$, which we call S. We use S to then compute the fingerprint and the sublattice information of the vector sums for $\mathcal{L}$.

We must then enumerate the short vectors of $\mathcal{J}$, which we call S_2. Our isometry (if it exists) will be built up by assigning an image vector from S_2 to each basis vector from B.

At each step, check to see that the inner product is preserved by using the fingerprint and the vector sums. The isometry will be completely determined once we have found the image of the entire basis B using candidate vectors from S_2.

3.2.1 The Search

In building up an isometry between $(\mathcal{L}, \Phi)$ and $(\mathcal{J}, \Psi)$, we are really constructing a partial basis. At each step, we verify that the current partial basis $(\mathbf{v}_1, \cdots, \mathbf{v}_k)$ represents a k-partial automorphism by checking the number of possible extensions against the fingerprint of $\mathcal{L}$, as well as the vector sum information generated by using $(\mathbf{v}_1, \cdots, \mathbf{v}_k)$. Though we call it a k-partial automorphism, what we are building here is an isometry as $(\mathcal{L}, \Phi)$ and $(\mathcal{J}, \Psi)$ may be different lattices.

Since inner products will be preserved under an isometry φ, the vector sum information generated by B should be the same as the vector sum information generated by the image of B. To check a partial map $(\mathbf{v}_1, \cdots, \mathbf{v}_k)$, we compute the inner product of each candidate vector from S with $\mathbf{v}_i$. We store these in a lookup table W'_k, and use it as a matrix to calculate $A'_k = W'_k X_k$. Here X_k has already been calculated with respect to the first k basis vectors from B, which we denote by $B_k = (\mathbf{b}_1, \cdots, \mathbf{b}_k)$. Checking that the inner products of the vectors in A'_k match those of A_k will help us rule out possible k-partial automorphisms which will not extend to $(k+1)$-partial automorphisms. We also compute a third lookup table and matrix, $W''_k = A'_k Y_k$, where Y_k was computed with respect to B_k, and check that the inner products of the vectors from W''_k match those of W'_k.

If all of these conditions are met, we can continue to try to extend our partial automorphism. Choose a vector $\mathbf{v}$ from the set of candidates (in this case what is left of B_2 after removing the vectors already present in our partial automorphism) and check $(\mathbf{v}_1, \cdots, \mathbf{v}_k, \mathbf{v})$ using the above methods.

Algorithm 5 Test A Continuation

1: **procedure** MEETSCRITERIA($B_1, B_2, [\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}, \mathbf{v}], W_k, X_k, A_k, Y_k, S_1, S_2, f$)

Input: first basis B_1, second basis B_2, partial map $[\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}, \mathbf{v}]$, vector sum matrix W_k, transformation matrix X_k, reduced matrix A_k, reverse transformation matrix Y_k, first set of small vectors S_1, second set of small vectors S_2, fingerprint f

Output: True if continuation is valid, otherwise False

2: **if** $k < n$

3: $F_2 = B_2^T B_2$

4: count =NUMCONT(B_2, F_2, S_2, k)

5: **if** count $\neq f_{kk}$ ▷ First criteria

6: **return** False

7: $W'_k =$ VECTORSUMS($[\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}, \mathbf{v}], S, k$)

8: $A'_k = W'_k X$

9: **if** $\{\Phi(\mathbf{a}_i, \mathbf{a}_i)\}_A \neq \{\Phi(\mathbf{a}'_i, \mathbf{a}'_i)\}_{A'}$ ▷ Second criteria

10: **return** False

11: $W''_k = A'_k Y$

12: **if** $\{\mathbf{w}''_i\}_{W''} \neq \{\mathbf{w}'_i\}_{W'}$ ▷ Third criteria

13: **return** False

14: **return** True

If $(\mathbf{v}_1, \cdots, \mathbf{v}_k)$ fails one of these conditions, we remove it from the pool of candidates and choose another vector. If the pool of candidates is empty, we backtrack and throw out the most recently added vector, leaving us with $(\mathbf{v}_1, \cdots, \mathbf{v}_{k-1})$, which we must then extend using candidates other than $\mathbf{v}_k$. The search terminates either with a complete basis or an empty partial automorphism. In the case where no isometry is found, we will have exhaustively searched the set of candidates for a suitable image of the basis B. We can conclude with certainty that it will terminate since the pool of candidate vectors is finite. This limits the

search for each successive basis vector, and results in a finite search altogether. A simplified version of this function is given below.

Algorithm 6 Stabilizer Candidates

1: **procedure** STABCANDIDATES(B, F, S, k) ▷ Similar to numExt()

Input: basis B, Gram matrix F, small vectors S, index k

Output: candidates with correct inner products

2: **for** $\mathbf{v} \in S$

3: **if** $\Phi(\mathbf{v}, \mathbf{v}) = F_{kk}$

4: **if** $\Phi(\mathbf{v}, \mathbf{b}_j) = F_{kj}\ \forall j = 1, \ldots, k-1$

5: candidates = candidates + $\mathbf{v}$

6: **return** candidates

Algorithm 7 Backtrack Search

1: **procedure** SEARCH($B_1, B_2, [\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}], Candidates, W_k, X_k, A_k, Y_k, S_1, S_2, f$)

Input: first basis B_1, second basis B_2, partial map $[\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}]$, small vectors $Candidates$, vector sum matrix W_k, transformation matrix X_k, reduced matrix A_k, reverse transformation matrix Y_k, first set of small vectors S_1, second set of small vectors S_2, fingerprint f

Output: compete isometry *map* or -1 if none exists

2: $C_k = Candidates$

3: $C_{k+1} = Candidates$

4: **while** $C_k \neq \varnothing$

5: choose $\mathbf{v} \in C_k$

6: **if** MEETSCRITERIA($B_1, B_2, [\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}, \mathbf{v}], W_k, X_k, A_k, Y_k, S_1, S_2, f$)

7: $C_{k+1} = C_{k+1} \backslash \{\mathbf{v}\}$

8: map = SEARCH($B_1, B_2, [\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}], C_{k+1}, W_k, X_k, A_k, Y_k, S_1, S_2, f$)

9: **if** map = -1

10: $C_{k+1} = C_{k+1} + \{\mathbf{v}\}$

11: **else**

12: **return** map

13: **return** -1

The set of candidate vectors S is treated as an unordered list. In practice, however, it has an unspecified ordering which determines which candidates are chosen first. Because of this, the algorithm will search the space of candidate vectors in the same order, finding the same isometry between two lattices each time, even if many isometries exist.

It is possible to exhaustively find every isometry that exists between two lattices, either by reordering S to account for all permutations, or by keeping a list of isometries already found. When a candidate is selected to extend a partial map, those vectors which would

complete the map to an already found isometry should be removed from the candidate pool.

The randomized element of the backtrack search `random.shuffle(C)` allows the user to call the function many times and expect to get a few different maps. For lattices of small dimension this is one way of generating a few automorphisms. However, one drawback of this functionality is that we may not receive the particular isometry we expect to receive when more than one exists between two lattices. Because of the deterministic behavior of the algorithm otherwise, this problem still exists without randomization. The shuffle command at least allows the chance of finding different isometries, whereas without it we might expect to get the same isometry regardless of which is the most obvious map. An example of this is given in the Tests section.

Below is an overview of how to use the tools we have constructed so far to test two lattices for an isometry. Our search is conducted in such a way that if the two input lattices $(\mathcal{L}, \Phi)$ and $(\mathcal{J}, \Psi)$ are isometric, this algorithm will return an isometry $\varphi\colon (\mathcal{L}, \Phi) \to (\mathcal{J}, \Psi)$. If they are not isometric, the algorithm will return a value of -1 after exhaustively checking any partial maps that preserve the bilinear form on $\mathcal{L}$. If no partial map extends to a full isometry, only after ruling out all partial maps will our algorithm return -1.

Algorithm 8 Find An Isometry

1: **procedure** ISISOMETRIC(B_1, B_2)

Input: first basis B_1, second basis B_2

Output: compete isometry *map* or -1 if none exists

Preprocessing

2: $F_1 = (B_1)^T B_1$

3: $S_1 = \text{ENUMERATE}(B_1)$

4: $f = \text{FINGERPRINT}(B_1, F_1, S_1)$

5: **for** $k = 1, \ldots, n$

6: $W_k, X_k, A_k, Y_k = \text{VECTORSUMS}([\mathbf{b}_1, \ldots, \mathbf{b}_k], S_1, k)$

Search

7: $S_2 = \text{ENUMERATE}(B_2)$

8: map = $\text{SEARCH}(B_1, B_2, [\varnothing], S_2, W_k, X_k, A_k, Y_k, S_1, S_2, f)$

9: **return** map

The number of possible maps to check is exponential with respect to the size of S. We reduce this number by ruling out partial maps which will not extend to full maps. The number of arithmetic operations and comparisons in the search grows exponentially with respect to the size of the basis. The runtime will therefore also grow exponentially as the dimension increases. We tested this algorithm by asking it to find an isometry between the standard integer lattice in dimension n and an identical integer lattice in dimension n using the `timeit()` command in Sage. Below we list the average runtime for $n = 2, \ldots, 8$. The results are consistent with what we predicted. The runtime grows exponentially with the dimension.

dimension	2	3	4	5	6	7	8
runtime (ms)	101	295	720	1580	3240	6030	10700

Table 3.2: Isometry Finding Runtime

3.3 Automorphism Group

To compute the automorphism group $Aut(\mathcal{L})$, we rely on methods for computing strong generating sets. We make use of the pointwise stabilizers

$$G_i = \{\ \varphi \in Aut(\mathcal{L})\ :\ \varphi(\mathbf{b}_j) = \mathbf{b}_j \text{ for } j = 1, \ldots, i-1\ \}$$

and attempt to calculate a generating set $\mathcal{G}$ for $Aut(\mathcal{L})$ which contains the generators for each G_i. To do this, we iteratively build up $\mathcal{G}$ by building up each G_i. As we build it up, we call the intermediate group which we generate at each step H_i. We generate H_i from a set of generators in $\mathcal{G}$ which we know to be in G_i. As $\mathcal{G}$ grows, so does each H_i, until finally H_i is the full set of stabilizers G_i. At each step, we compute $H_i = \langle H_i \cap \mathcal{G} \rangle$ for $1 \leq i \leq n$.

Begin with $\mathcal{G}$ empty, and start by computing H_1. To do this, we simply search for isometries $\varphi\colon \mathcal{L} \to \mathcal{L}$. At each stage we continue this process and add these maps to $\mathcal{G}$. We then regenerate H_i and update the candidates. Over time the value f_{ii} taken from the fingerprint will decrease as vectors are ruled out. Similarly, the size of the orbit $H_i\mathbf{b}_i$ will increase as completed maps are added to H_i. Since the set of candidates C_k is finite, eventually $|H_i\mathbf{b}_i| \geq f_{ii}$, and we will move on to the next value of k. Once $k = n$, we cannot continue any further, and the algorithm will terminate. An outline of this method is given below.

Algorithm 9 Automorphism Group

1: **procedure** AUT(B)

Preprocessing

Input: basis B

Output: automorphism group Aut

2: Compute $F, S, f, W_k, X_k, A_k, Y_k$ as before

3: $\mathcal{G} = \varnothing$

4: **for** $k = 1, \ldots, n$

5: $H_k = \varnothing$

6: $C_k = \text{STABCANDIDATES}(B, F, S, k)$

7: **for** $k = 1, \ldots, n$

8: **while** $|H_k \mathbf{b}_k| > f_{kk}$ and $C_k \neq \varnothing$

9: choose $\mathbf{v} \in C_k$

10: map = $\text{SEARCH}(B_1, [\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}, v], C_k, W_k, X_k, A_k, Y_k, S_1, f)$

11: **if** map = -1

12: $f_{kk} = f_{kk} - |H_k \mathbf{v}|$

13: **else**

14: $\mathcal{G} = \mathcal{G} \cup \text{map}$

15: $H_k = \langle H_k, \text{map} \rangle$

16: $C_k = C_k \backslash H_k \mathbf{v}$

17: $Aut = \langle \mathcal{G} \rangle$

18: **return** Aut

We can use the automorphism group of a lattice to construct the set of all isometries between two lattices $\mathcal{L}$ and $\mathcal{J}$. First, search for an isometry $\varphi\colon \mathcal{L} \to \mathcal{J}$. Then compute the automorphism group $Aut(\mathcal{L})$. Compose each map $\gamma \in Aut(\mathcal{L})$ with φ to obtain all isometries

between $\mathcal{L}$ and $\mathcal{J}$. Searching for an isometry between the two lattices will also tell us if they are not isometric, in which case we will have nothing to compose with our maps in $Aut(\mathcal{L})$.

3.4 Tests

These methods were tested on integer lattices of varying dimensions, whose automorphism group is known to have order $2^n \cdot n!$ for dimension n. As an object of type `MatrixGroup`, the output of the above algorithm can be treated as a group in Sage, and the order can be obtained.

Because computing the automorphism group uses the same procedures as searching for isometries, it runs in exponential time with respect to the size of the basis. We tested this algorithm on integer lattices with the `timeit()` command in Sage. Listed below are the runtimes corresponding to $n = 2, \ldots, 5$. Higher dimensions require more memory to run. The lattices tested have relatively uniform basis vectors. Runtime is lower for these than it would be for lattices with basis vectors whose norms vary widely.

dimension	2	3	4	5
runtime (ms)	913	3250	14100	196000

Table 3.3: Automorphism Group Runtime

Lastly, obtaining different isometries by making use of the randomization allows us the possibility of finding a particular map. An example of this is given below for the integer lattice in 3 dimensions and a lattice given by a permutation of its basis vectors.

```
for i in range(100):
    V = is_isometric(L,J)
    if V not in poss_maps:
        poss_maps.append(V)
for v in poss_maps:
```

```
    print Matrix(v)
    if Matrix(v) == J.embedded_basis_matrix():
        print "We have a match!"
    print "\n"
```

...

```
[1  0  0]
[0  0  1]
[0  1  0]
We have a match!
```

...

Chapter 4

Enumeration

4.1 Generating Small Vectors

A popular method of finding the smallest vector of a lattice involves enumerating all the vectors of the lattice with relatively small norms and searching for the smallest member. Since the *Smallest Vector Problem* is considered computationally hard, we can expect this enumeration to have exponential complexity. Until 1985, this involved calculating all the suitable vectors inside a rectangular area. Michael Pohst and Ulrich Fincke improved on this method by reducing the search to a hyper-ellipsoid. At the same time, Kannan devised a similar method which involves searching a hyper-parallelepiped. This method is referred to as KFP enumeration, for its creators Kannan, Fincke, and Pohst.

Below we discuss the Fincke-Pohst algorithm. This algorithm is utilized in Chapter 3, where the set of small vectors belonging to a lattice helps us to find isometries and automorphisms. These methods are useful for determining isometry classes of lattices.

4.1.1 Overview

We are interested in finding all of the small vectors of a lattice $\mathcal{L}$.

Definition 22 *We call a vector* $\mathbf{v}$ ***small*** *if its norm* $\Phi(\mathbf{v}, \mathbf{v})$ *is less than a constant* C*. When* C *is not specified, it is assumed to be the norm of the largest basis vector of the lattice.*

This clearly depends on the basis which is given, and can vary depending on the choice of basis. If a particular basis is not specified, it is assumed to be the matrix B which defines the Gram matrix $A = B^T B$. This is equivalent to solving the inequality $\Phi(\mathbf{y}, \mathbf{y}) \leq C$, where $\Phi(\mathbf{y}, \mathbf{y})$ denotes the norm of the vector computed with respect to the lattice. Let B denote

the matrix whose columns are the basis vectors of the lattice $\mathcal{L}$. As an element of the lattice, $\mathbf{y} = B\mathbf{x}$ for some coordinate vector $\mathbf{x} \in \mathbb{Z}^n$. So our inequality becomes

$$\Phi(\mathbf{y}, \mathbf{y}) = \mathbf{y}^T\mathbf{y} = \mathbf{x}^T B^T B\mathbf{x} \leq C$$

We consider the quadratic form $Q(\mathbf{x}) = \mathbf{x}^T B^T B\mathbf{x}$ and solve $Q(\mathbf{x}) \leq C$.

4.2 Quadratic Completion

To solve our inequality, it helps to first rearrange the terms of our quadratic form. This reformulation is called the *quadratic completion* or *quadratic complementation*. Here we assume the lattice is positive definite. That is, every nonzero element has a positive norm. With this, we can find the Cholesky decomposition $A = LL^T$, where L is a lower triangular matrix. Equivalently, we can express this as $A = R^T R$, where R is an upper triangular matrix. Since [10] uses upper triangular matrices, this is what we will use. The formulas below will reflect this. We now express Q as:

$$Q(\mathbf{x}) = \sum_{i=1}^{m} q_{ii} \left(x_i + \sum_{j=i+1}^{m} q_{ij} x_j \right)^2$$

Our coefficients q_{ij} are obtained from R, and stored in a matrix for convenience.

$$q_{ij} = \begin{cases} \frac{r_{ij}}{r_{ii}}, & i < j \\ r_{ii}^2, & i = j \end{cases}$$

Since R is upper triangular, the matrix $Q = [q_{ij}]$ will be as well.

Algorithm 10 Generate Quadratic Completion

1: **procedure** GENQ(A)

Input: Gram matrix A

Output: coefficients q_{ij} stored in matrix Q

2: **for** $i = 1, \ldots, n$

3: **for** $j = i, \ldots, n$

4: $q_{ij} = a_{ij} - \sum_{k=1}^{i-1} q_{ki} q_{kj} q_{kk}$

5: **if** $i \neq j$ ▷ For non-diagonal entries

6: $q_{ij} = q_{ij}/q_{ii}$

7: **return** Q

Example 23 *Let* $A = \begin{pmatrix} 4 & -2 & -4 \\ -2 & 5 & 0 \\ -4 & 0 & 30 \end{pmatrix}$. *We will find* R *such that* $R^T R = A$. *Expanding this, note that*

$$\begin{bmatrix} r_{11} & 0 & 0 \\ r_{12} & r_{22} & 0 \\ r_{13} & r_{23} & r_{33} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{bmatrix} = \begin{bmatrix} r_{11}^2 & r_{11}r_{12} & r_{11}r_{13} \\ r_{11}r_{12} & r_{12}^2 + r_{22}^2 & r_{12}r_{13} + r_{22}r_{23} \\ r_{11}r_{13} & r_{12}r_{13} + r_{22}r_{23} & r_{13}^2 + r_{23}^2 + r_{33}^2 \end{bmatrix}$$

We proceed column by column, beginning with the first.

$$r_{11}^2 = 4$$

$$r_{11}r_{12} = -2$$

$$r_{11}r_{13} = -4$$

It is easy to see that $r_{11} = 2, r_{12} = -1, r_{13} = -2$.

$$r_{12}^2 + r_{22}^2 = 5$$

$$r_{12}r_{13} + r_{22}r_{23} = 0$$

Solving for the second column we find $r_{22} = 2$ *and* $r_{23} = -1$.

$$r_{13}^2 + r_{23}^2 + r_{33}^2 = 30$$

Lastly, $r_{33} = 5$. *Thus,* $R = \begin{bmatrix} 2 & -1 & -2 \\ 0 & 2 & -1 \\ 0 & 0 & 5 \end{bmatrix}$.

To obtain the upper triangular matrix R from our matrix A, we compute the diagonal and non-diagonal entries as follows:

$$r_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} r_{ki}^2}$$
$$r_{ij} = \frac{1}{r_{ii}} \left(a_{ij} - \sum_{k=1}^{j-1} r_{ki} r_{kj} \right)$$

Using these, we can reformulate the construction of the coefficients of Q to use values from A. We will soon see how it is possible to do away with using the Cholesky decomposition entirely.

$$q_{ii} = a_{ii} - \sum_{k=1}^{i-1} r_{ki}^2$$
$$q_{ij} = \frac{1}{r_{ii}^2} \left(a_{ij} - \sum_{k=1}^{j-1} r_{ki} r_{kj} \right)$$

Example 24 *Continuing with the example given above, we obtain* Q *(put in matrix form for simplicity) to be* $\begin{bmatrix} 4 & -1/2 & -1 \\ 0 & 4 & -1/2 \\ 0 & 0 & 25 \end{bmatrix}$. *The quadratic form is then given by:*

$$\begin{aligned} Q(\mathbf{x}) &= 4(x_1 - \frac{1}{2}x_2 - x_3)^2 + 4(x_2 - \frac{1}{2}x_3)^2 + 25(x_3)^2 \\ &= 4(x_1^2 - x_1x_2 - 2x_1x_3 + \frac{1}{4}x_2^2 + x_2x_3 + x_3^2) + 4(x_2^2 - x_2x_3 + \frac{1}{4}x_3^2) + 25(x_3^2) \\ &= 4x_1^2 - 4x_1x_2 - 8x_1x_3 + 5x_2^2 + 30x_3^2 \\ &= \mathbf{x}^T A \mathbf{x} \end{aligned}$$

By putting this construction in terms of the coefficients of Q only, we arrive at the following

$$q_{ii} = a_{ii} - \sum_{k=1}^{i-1} q_{ki}^2 q_{kk}$$
$$q_{ij} = \frac{1}{q_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} q_{ki} q_{kj} q_{kk} \right)$$

We can then calculate these coefficients, starting with q_{11} and calculating q_{1j} for $1 \leq j \leq m$. Then we continue by calculating q_{22} and q_{2j} for $2 \leq j \leq m$. We proceed by first always calculating the diagonal entry q_{ii} and then q_{ij} for $i \leq j \leq m$ until we reach q_{mm}.

In practice, this is how we compute the coefficients for our form. However, it is equally possible to first compute the Cholesky Decomposition using available methods, and then computing the entries of Q from this.

4.3 Bounding x_i

Since the sum $Q(\mathbf{x})$ is less than C, the individual term $q_{mm}x_m^2$ must also be less than C.

$$\sum_{i=1}^{m} q_{ii} \left(x_i + \sum_{j=i+1}^{m} q_{ij} x_j \right)^2 \leq C$$
$$q_{mm} x_m^2 \leq C$$
$$x_m^2 \leq C/q_{mm}$$

In fact, x_m is bounded above by $\sqrt{C/q_{mm}}$ and below by $-\sqrt{C/q_{mm}}$.

This illustrates the first step in establishing bounds on a specific entry x_i. Adding more

terms from the outer sum to this sequence, a pattern emerges.

$$q_{mm}x_m^2 \leq C$$

$$q_{m-1,m-1}\left(x_{m-1} + q_{m-1,m}x_m\right)^2 \leq C - q_{mm}x_m^2$$

$$q_{m-2,m-2}\left(x_{m-2} + \sum_{j=m-2}^{m} q_{m-2,j}x_j\right)^2 \leq C - q_{mm}x_m^2 - q_{m-1,m-1}\left(x_{m-1} + q_{m-1,m}x_m\right)^2$$

We denote this internal sum by U_k

$$U_k = \sum_{j=k+1}^{m} q_{kj}x_j$$

so that we can rewrite $Q(\mathbf{x})$ as

$$Q(\mathbf{x}) = \sum_{i=1}^{m} q_{ii}\left(x_i + U_i\right)^2$$

In general,

$$q_{kk}\left(x_k + U_k\right)^2 \leq C - \sum_{i=1}^{k-1} q_{ii}\left(x_i + U_i\right)^2$$

The bound on this summand is denoted T_k. So $T_m = C$, $T_{m-1} = C - q_{mm}x_m^2$, and

$$T_{m-2} = C - q_{mm}x_m^2 - q_{m-1,m-1}\left(x_{m-1} + q_{m-1,m}x_m\right)^2$$

It helps to simply set T_m as C and find each subsequent T_k by subtracting the next term from the outer summand.

$$T_k = C - \sum_{i=1}^{k-1} q_{ii}\left(x_i + U_i\right)^2$$

$$T_k = T_{k+1} - q_{k+1,k+1}\left(x_{k+1} + U_{k+1}\right)^2$$

Now we have an upper bound for each summand.

$$q_{kk}\left(x_k + U_k\right)^2 \leq T_k$$

Using this, we can estimate upper and lower bounds for each x_k in the coordinate vector $\mathbf{x}$. We start by computing the last entries of $\mathbf{x}$ and their bounds first. Assuming that the last several entries of $\mathbf{x}$ have been assigned, upper and lower bounds on x_k can be determined. Now that we have established a bound on a term in the outer sum, we can determine bounds on the specific entry x_k. Take the above equation, and solve for x_k.

$$\begin{aligned}(x_k + U_k)^2 &\leq T_k/q_{kk} \\ x_k + U_k &\leq \sqrt{T_k/q_{kk}} \\ x_k &\leq \sqrt{T_k/q_{kk}} - U_k\end{aligned}$$

Similarly we have a lower bound.

$$x_k \geq -\sqrt{T_k/q_{kk}} - U_k$$

Since x_k must be an integer, we can restrict our bounds further. Let $t_k = \sqrt{T_k/q_{kk}}$.

$$\begin{aligned}UB_k &= \lfloor t_k - U_k \rfloor \\ LB_k &= \lceil -t_k - U_k \rceil\end{aligned}$$

Here UB_k is the upper bound on x_k and LB_k is the lower bound on x_k.

$$LB_k \leq x_k \leq UB_k$$

4.3.1 Algorithm

To enumerate all of the vectors $\mathbf{x}$ such that $Q(\mathbf{x}) \leq C$, begin with the last entry x_m (letting all other $x_j = 0$). Determine the upper and lower bounds UB_m and LB_m by first calculating $t_m = \sqrt{T_m/q_{mm}}$. We define $U_m = 0$, and by definition remember that $T_m = C$.

For each entry x_i, starting with x_m and going down to x_1, we initialize the value to be $x_i = LB_i - 1$. After the value is initialized, we begin to increment the values of all the entries, adding 1 to each entry until we either reach the last index (in which case we have found a

solution) or we exceed the upper bound on a particular entry (we will need to readjust the previously assigned entries). If at any time the lower bound exceeds the upper bound for a given entry, it will become immediately apparent when the value for that entry is initialized. We must then backtrack to our previous entries (that is, entries with a higher index). If we reach x_1 without exceeding the upper bounds for any entry, then we have found a complete vector $\mathbf{x}$ which satisfies $Q(\mathbf{x}) \leq C$.

We will know we have found all the short vectors when we reach the zero vector. This is because we start by assigning each value x_i its lower bound, which is calculated with respect to the values $x_{i+1}, \ldots, x_n$. We increase x_i incrementally, until it exceeds the corresponding calculated upper bound. When this happens we revisit x_{i+1}, increasing its value. Since x_{i+1} was originally assigned its own lower bound, it starts off as a negative integer and increases steadily until it reaches 0. Likewise, the other values will start off negative at each iteration and slowly increase in value. It is only when all entries are 0 that the algorithm terminates. When we add each vector, we also add the vector with entries $-x_i$ for each i. In this we capture all the small vectors without having to check positive values for x_n.

Before beginning the search, first find the coefficients of the quadratic form expressed as above. Initialize T_k, U_k, UB_k and x_k to be 0 for all k. Begin with $i = m$ and $T_i = C$ as the value bounding our vectors.

Algorithm 11 Enumerate Small Vectors

1: **procedure** ENUMERATE(B)

Input: basis B

Output: small vectors *solutions*

2: Initialize Gram matrix A and start with $i = n$

3: $Q = \text{GEN}\mathrm{Q}(A)$

4: stillEnumerating = True

5: **while** stillEnumerating

6: $t_i = \sqrt{T_i / q_{ii}}$

7: $UB_i = \lfloor t_i - U_i \rfloor$ ▷ Set bounds for x_i

8: $x_i = \lceil -t_i - U_i \rceil - 1$

9: updatingEntries = True

10: **while** updatingEntries

11: $x_i = x_i + 1$ ▷ Increase x_i

12: **if** $x_i > UB_i$

13: $i = i + 1$ ▷ Increase i

14: **else**

15: **if** $i = 1$

16: add $\mathbf{x}$ to *solutions* ▷ Coordinate vector is a solution

17: **if** $\mathbf{x} = (0, \ldots, 0)$

18: **return** *solutions*

19: add $-x$ to *solutions*

20: **else**

21: $T_{i-1} = T_i - q_{ii}(x_i + U_i)^2$

22: $i = i - 1$ ▷ Decrease i

23: recompute U_i

24: updatingEntries = False

4.3.2 Improvements

It is noted in [10] that if we label the columns of R by $\mathbf{r}_i$ (from the Cholesky decomposition $\mathbf{x}^T R^T R\mathbf{x}$) and the rows of R^{-1} by $\mathbf{r}'_i$, then we see that

$$x_i^2 = \left(\mathbf{r}_i'^T \left(\sum_{k=1}^{m} x_k \mathbf{r}_k\right)\right)^2 \leq \mathbf{r}_i'^T \mathbf{r}_i \left(\mathbf{x}^T R^T R\mathbf{x}\right) \leq ||\mathbf{r}'_i||^2 C$$

So it may behoove us to reduce the rows of R^{-1} in order to reduce our search space. Furthermore, it helps to put the smallest basis vectors first, so reordering the columns may also be beneficial.

Express this reduction with a unimodular matrix V^{-1}, so that $R_1^{-1} = V^{-1}R^{-1}$. Then reorder the columns of R_1 with a permutation matrix P. Since $R_1 = RV$, we then have that $R_2 = (RV)P$.

Then $R_2^{-1} = P^{-1}V^{-1}R^{-1}$. If we find a solution to the inequality $\mathbf{y}^T R_2^T R_2 \mathbf{y} \leq C$, we can recover a solution to our original inequality by $\mathbf{x} = VP\mathbf{y}$. Since $R_2^{-1} = P^{-1}V^{-1}R^{-1}$, we know that $R_2 = RVP$.

$$\begin{aligned}
\mathbf{y}^T R_2^T R_2 \mathbf{y} &\leq C \\
\mathbf{y}^T (P^T V^T R^T)(RVP)\mathbf{y} &\leq C \\
(\mathbf{y}^T P^T V^T) R^T R (VP\mathbf{y}) &\leq C \\
(VP\mathbf{y})^T R^T R (VP\mathbf{y}) &\leq C \\
\mathbf{x}^T R^T R\mathbf{x} &\leq C
\end{aligned}$$

This improves the search time by giving us a nicer quadratic form to work with. Once we find solutions to the inequality given by $Q_2(\mathbf{y}) = \mathbf{y}^T R_2^T R_2 \mathbf{y} \leq C$, it is a simple matter of translating them into solutions of our original inequality.

4.3.3 Runtime

While reformulating the quadratic form runs in polynomial time with respect to the size of the basis, enumerating all of the small vectors runs in exponential time with respect to the

basis. There are an exponential number of possible vectors to check. This algorithm reduces that number slightly by estimating bounds on the values to rule out vectors which need not be checked. The algorithm was run on the standard integer lattice in varying dimensions, similar to the tests we ran in Chapter 3, using the `timeit()` command in Sage. The runtimes below are given in seconds.

dimension	2	3	4	5	6	7	8
runtime (s)	1.58	2.37	3.47	4.59	5.89	7.14	8.92

Table 4.1: Enumeration Runtime

Chapter 5

Smith-Minkowski-Siegel Mass Formula

One application of the methods that we have established so far is the computation of the *Smith-Minkowsky-Siegel mass formula*. The mass formula gives a weighted sum of the inequivalent lattices in a genus. To define the genus of a lattice, we first require the concept of localization.

Localizing a commutative ring at an element p allows us to include powers of p in our denominators. The resulting ring contains fractions $\frac{a}{p^k}$ where a is an element of the original ring and k is some integer. Localizing a lattice at p changes the types of coefficients that we can have in our linear combinations. Using our integer lattices, for example, if we localize at 3, then the element $\frac{2}{3}$ is now a valid coefficient that we can use in our coordinate vectors. A valid coordinate vector might be $[1, 0, \frac{2}{3}]$.

One way to view this is by examining the tensor product $\mathcal{L} \otimes_{\mathbb{Z}} \mathbb{Z}_p$. Here our coefficients come from the p-adic integers $\mathbb{Z}_p$, which are a subset of the p-adic completion $\mathbb{Q}_p$.

Definition 25 *Two lattices $\mathcal{L}$ and $\mathcal{J}$ are **locally isometric** at p if their localizations $\mathcal{L}_p$ and $\mathcal{J}_p$ are isometric.*

Lattices which are isometric are also locally isometric. However, lattices which are locally isometric are not necessarily isometric. Even if lattices are isometric at every prime p, they are still not necessarily isometric. In fact, this is what gives us the notion of the genus of a lattice.

Definition 26 *The **genus** of a lattice $\mathcal{L}$ is the set of all lattices whose tensor product $\mathcal{L} \otimes_{\mathbb{Z}} \mathbb{Z}_p$ and $\mathcal{J} \otimes_{\mathbb{Z}} \mathbb{Z}_p$ are isometric at every prime p.*

Since we know that not necessarily all lattices in a genus are isometric, a genus can be partitioned into isometry classes. Inside each isometry class are the lattices which are mutually isometric to each other. These form equivalence classes inside the genus.

Example 27 *Among lattices of rank 3, we have an example of two distinct isometry classes in the same genus. From the first, we take the standard integer lattice* $\mathcal{L}$ *with basis vectors* $[1,0,0],[0,1,0]$*, and* $[0,0,1]$ *to be the representative. From the second, take the lattice* $\mathcal{J}$ *with basis vectors* $[1,0,0],[0,3,0]$*, and* $[1/3,2/3,1/3]$*.*

We equip each lattice with the bilinear form given by the matrix

$$A = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 6 \end{bmatrix}$$

On coordinate vectors $\mathbf{x}$ *and* $\mathbf{y}$ *of* $\mathcal{L}$*, we have* $\mathbf{x}^T A\mathbf{y}$*. On coordinate vectors* $\mathbf{x}$ *and* $\mathbf{y}$ *of* $\mathcal{J}$ *we compute* $\mathbf{x}^T B^T AB\mathbf{y}$*, where* B *is the matrix whose columns are the basis vectors given for* $\mathcal{J}$*. These lattices, equipped with this bilinear form, are not isometric. That is, there is no map* $\varphi\colon \mathcal{L} \to \mathcal{J}$ *for which* $\varphi(\mathbf{x})B^T AB\varphi(\mathbf{y})$ *for all* $\mathbf{x},\mathbf{y}$ *in* $\mathcal{L}$*.*

Currently our methods use the inner product induced by the basis as the bilinear form on the elements of the lattice. Arbitrary bilinear forms can be included in the testing by modifying the functions which compute the Gram matrix and the inner product to use a new bilinear form. When tested on this example, the modified algorithm should exhaust all possible maps between the lattices before certifying that no isometry exists.

Example 28 *An example of two distinct genera of lattices of of the same rank occurs in dimension 8. One genus contains a single isometry class, characterized by the* E_8 *root lattice, which we see in Example 29. The other contains the standard integer lattice of dimension 8.*

From each isometry class, we choose one representative. The mass formula uses these

representatives to construct a weighted sum.

$$\sum_{i=1}^{h} \frac{1}{|Aut(\mathcal{L}^{(i)})|}$$

Here each $\mathcal{L}^{(i)}$ is a representative of an isometry class.

This formula is useful because it allows us to verify our enumeration of isometry classes. Say we have generated a multitude of lattices. We can use the isometry testing in Chapter 3 to separate the lattices into isometry classes, but we still would not know whether we have found every isometry class in the genus.

To test this, choose a representative from each known isometry class and compute the size of its automorphism group using the algorithm outlined in Chapter 3. Compute the mass of the genus using these values and the Smith-Minkowski-Siegel mass formula. Compare the result of this computation with known values of the mass formula, which are often calculated using other methods such as Bernoulli numbers. If the known value exceeds our computed value, we can conclude that we have not yet enumerated all of the isometry classes.

5.1 Lower Bound

To arrive at a lower bound for the mass formula, we begin with an upper bound on the size of the automorphism group of a lattice. If computing the full automorphism group of a class is too costly, a lower bound can be established.

In Chapter 3 we covered the construction of a matrix called the fingerprint of a lattice. When computing each row, we took note of the diagonal entry. Together these entries gave us an upper bound on the size of the automorphism group of the lattice.

We let G_i be the set of pointwise stabilizers for $\mathbf{b}_1$ up to $\mathbf{b}_{i-1}$. Each entry f_{ii} of the fingerprint then gives an upper bound on the size of $G_i\mathbf{b}_i$. We see that f_{ii} bounds the size of this particular orbit of $\mathbf{b}_i$, since f_{ii} tracks the number of extensions of the $(i-1)$-partial automorphism $(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})$ to an i-partial automorphism $(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{v})$.

For an n-dimensional lattice, we should obtain n of these bounds. Their product then yields an upper bound on the size of the automorphism group.

$$|Aut(clL)| \leq \prod_{i=1}^{n} f_{ii}$$

Example 29 *Recall from Chapter 3 that we estimated the size of the automorphism group of the E_8 root lattice by taking the product of the diagonal entries in the fingerprint. Below is the fingerprint. We take the product of the diagonal entries.*

$$\left[\begin{array}{cccccccc} 240 & 240 & 2160 & 240 & 240 & 240 & 240 & 240 \\ 0 & 56 & 126 & 126 & 126 & 126 & 126 & 126 \\ 0 & 0 & 27 & 27 & 72 & 72 & 72 & 72 \\ 0 & 0 & 0 & 10 & 40 & 16 & 40 & 40 \\ 0 & 0 & 0 & 0 & 8 & 8 & 24 & 24 \\ 0 & 0 & 0 & 0 & 0 & 4 & 6 & 12 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{array}\right]$$

$$240 \cdot 56 \cdot 27 \cdot 10 \cdot 8 \cdot 4 \cdot 3 \cdot 2 = 2^{14} \cdot 3^5 \cdot 5^2 \cdot 7 = 696729600$$

The mass of a genus can be calculated for even unimodular lattices whose rank is divisible by 8 by using Bernoulli numbers. The E_8 root lattice is in fact the only even unimodular matrix of rank 8, and so completely characterizes the mass of its genus. The mass of its genus is therefore $\frac{1}{696729600}$.

An upper bound on automorphism groups yields a lower bound on the mass of a genus, since in each case we can provide a lower bound for a given summand. Furthermore the upper bound on a single automorphism group can provide a somewhat looser bound on the mass of a genus.

$$\frac{1}{M_k} \leq \frac{1}{|Aut(\mathcal{L}^{(k)})|} \leq \sum_{i=1}^{h} \frac{1}{|Aut(\mathcal{L}^{(i)})|}$$

for a particular isometry class, represented by the lattice $\mathcal{L}^{(k)}$, whose automorphism group is bounded above $|Aut(\mathcal{L}^{(k)})| \leq M_k$.

5.2 Connections

The methods we have described so far are useful not only for finding isometries, but also in the context of algebraic modular forms. The work of Benedict Gross [13], M. Greenberg and J. Voight [12] tie together algebraic modular forms and methods on isometry classes of lattices.

Definition 30 *The **class set** of a lattice is the set of isometry classes within the genus.*

Using the mass formula, the algorithms described in Chapter 3 contribute to a larger endeavor to examine algebraic modular forms using lattice methods. In particular, Sarah Chisholm presents a method of enumerating all lattices in a genus [4]. Her methods involve enumerating lattices, which can be tested at each stage for isometries, until each isometry class is represented in the collection. We can test pairs of lattices to determine isometry classes within the collection. Termination requires all of the isometry classes to be present. At least one representative from each isometry class should be in the collection of constructed lattices.

To check this condition, we use the mass formula to ensure that we have a representative from each isometry class in the genus. This is done by selecting a representative from each isometry class and computing the size of its automorphism group. We then evaluate the mass formula using these automorphism groups. By comparing the result of the mass formula to known values, we can determine if we have the entire class set. If the result does not match the known value of the formula, then at least one isometry class is missing from our collection.

Once the enumeration of the class set is finished, we can use this to further examine certain types of algebraic modular forms using the work of Gross, Greenberg, and Voight. Since these class sets can be considered the domain of certain kinds of algebraic modular forms, combining our methods with the results of Chisholm [4] provides us with a necessary tool to futher the study of algebraic modular forms.

Chapter 6

Conclusion

These algorithms were coded and tested in Sage 5.10 using the standard packages. Although these algorithms have been adapted to work in existing software, Sage has not yet seen the addition of these methods. They are used as a framework in proprietary software such as Magma [18], where the algorithms cannot be read or improved upon by an outsider. By submitting this code to Sage, third parties will be able to learn from or improve upon these methods. In addition, they will expand functionality of Sage so that more complex ideas can be developed which make use of them. For example, having access to algorithms which can compute the order of an automorphism group of a lattice may enable development of algorithms which look at the Smith-Minkowski-Siegel mass formula.

Future work with these methods would involve reducing runtime and applying the concepts in a more abstract setting. Implementing a depth parameter or the use of Bacher polynomials may reduce computation time for finding isometries. Runtimes which include these parameters are given in [23]. In addition, perhaps specific pieces might be optimized regarding matrix operations. Lastly, these methods could also be extended to work over number fields with more interesting forms.

Bibliography

[1] Overview of Magma V2.17: Lattices and Quadratic Forms. 2014.

[2] J. W. S. Cassels. *An Introduction To The Geometry Of Numbers*. Springer-Verlag, 1959.

[3] J. W. S. Cassels. *Rational Quadratic Forms*. Academic Press, 1978.

[4] Sarah Chisholm. *Algorithmic enumeration of quaternionic lattices*. PhD thesis, University of Calgary.

[5] H. Cohen. *A Course in Computational Number Algebraic Number Theory*. Springer, 3rd edition edition, 1996.

[6] Henri Cohen. Hermite and smith normal form algorithms over dedekind domains. *Math. Comp*, 65:1681–1699, 1996.

[7] Shafi Goldwasser Daniele Micciancio. *Complexity of Lattice Problems*. Kluwer Academic Publishers, 2002.

[8] Fred Diamond and Jerry Shurman. *A First Course in Modular Forms*. Springer, 2005.

[9] Claus Fieker and Damien Stehl. Short bases of lattices over number fields. In *In Proc. of ANTS-IX, volume 6197 of LNCS*, pages 157–173. Springer, 2010.

[10] U. Fincke and M. Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 1985.

[11] C.F. Gauss. Disquisitiones arithmeticae. 1801.

[12] M. Greenberg and J. Voight. Lattice methods for algebraic modular forms on classical groups. 2012.

[13] Benedict Gross. Algebraic modular forms. *Israel Journal of Mathematics*, pages 61–93, 1999.

[14] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 193–206. ACM, 1983.

[15] L. J. P. Kilford. *Modular Forms: A Classical and Computational Introduction*. Imperial College Press, 2008.

[16] L. Lagrange. Recherehes d'arithmetique. *Nouv. Mdm. Acad.*, pages 265–312, 1773.

[17] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. 261(4):515–534, 1982.

[18] Magma. Magma computational algebra system for algebra, number theory, and geometry, July 2014.

[19] Hermann Minkowski. *Geometrie der Zahlen*. Leipzig : Teubner, 1910.

[20] O. T. O'Meara. *Introduction to Quadratic Form*. Springer-Verlag, 1963.

[21] Sachar Paulus. Lattice basis reduction in function fields. In *In ANTS-3 : Algorithmic*, pages 567–575. Springer-Verlag, 1998.

[22] W. Plesken and M. Pohst. Constructing integral lattices with prescribed minimum i. *Math. Comp.*, (45(171)):209–221, 1985.

[23] W. Plesken and B. Souvignier. Computing isometries of lattices. *Journal of Symbolic Computation*, (24):327–334, 1997.

[24] Sage. Sage is a free open-source mathematics software, July 2014.

[25] Jean-Pierre Serre. *A Course in Arithmetic*. Springer, 1973.

Appendix A

First Appendix

```
def gen_Q(A):
    """
    A - Gramm matrix of a lattice
    """
    ## numCols - Number of columns (rank)    ##
    ## Q - coefficients for quadratic completion ##
    ## Q_sum - summation, temporary variable ##

    numCols = len(A.rows())
    ## Initializing Q to a matrix of all 0s
    Q = [ [0 for s in range(numCols)] for t in range(numCols)]

    ## Constructing quadratic completion
    for i in range(numCols):
        for j in range(i,numCols):
            Q_sum = 0
            for k in range(i):
                Q_sum += QQ(Q[k][i])*QQ(Q[k][j])*QQ(Q[k][k])
            Q[i][j] = A[i][j] - Q_sum

            ## For non-diagonal entries
            if i != j:
```

```
                    Q[i][j] = QQ(Q[i][j])/QQ(Q[i][i])
        return Q


def gramMat(B):
        ## Computes Gram Matrix from column-based basis matrix
        return B.transpose()*B


def enumerate(B,maxVal=0,verbose=False):
        """
        B - list of columns
        maxVal - maximum value
        """
        ## GramMat - Gram Matrix ##
        ## dimA - dimension of A ##



        ## Construct basis and Gram matrix from list of columns
        B = matrix(B).transpose()
        GramMat = gramMat(B)
        if verbose:
                print B,"\n\n",GramMat
        if maxVal == 0:
                maxVal = max(GramMat.diagonal())
                if verbose: print "Maximum value:",maxVal


        ## Generate Q, the quadratic completion
```

```
A = GramMat
dimA = len(A.rows())-1
if verbose: print "Calculating Q..."
Q = gen_Q(A)
if verbose: print "Quadratic matrix:\n",matrix(Q)

## Initialize variables for enumeration
## i - index specifying entry in coordinate vector ##
## T - upper bound on partial sum ##
## U - bound on inner sum ##
## UB - upper bound on entries in coordinate vector ##
## x - candidate coordinate vector ##
## solutions - list of small vectors ##
i,T,U,UB = dimA,[0 for q in Q],[0 for q in Q],[0 for q in Q]
x = [0 for q in Q]
T[i] = maxVal
solutions = []

## Iterate through coordinate vectors recursively
outerLoop = 1
while outerLoop:
        Z = (QQ(T[i])/QQ(Q[i][i]))^(1/2)
        UB[i] = RR(Z-U[i]).floor()
        x[i] = RR(-Z-U[i]).ceiling()-1

        innerLoop = 1
```

```
        while innerLoop:
            x[i] += 1
            if x[i] > UB[i]:
                i += 1


            else:
                ## Found a small vector
                if i == 0:
                    solutions.append(vector(x)) # Aadds the
                        coordinate vector
                    if not vector(x): ## If x is the zero-vector
                        return solutions
                    solutions.append((-1)*vector(x))


                ## Updating bounds
                else:
                    if verbose:
                        print T[i], Q[i][i], x[i], U[i]
                    T[i-1] = T[i] - Q[i][i]*(x[i]+U[i])^2
                    i -= 1
                    U[i] = 0
                    for k in range(i+1,dimA+1):
                        if verbose:
                            print "adding",Q[i][k],x[k]
                        U[i] += Q[i][k]*x[k]
                    if verbose:
```

```
                print x,"\tT[%d] = "%(i),T[i],"U[%d] = %d"%(i,U[i])
                break
            innerLoop = 0

def test_enumerate():
    expectedVectors = [(10, -5), (-10, 5), (11, -5), (-11, 5), (12, -5), (-12, 5),
    (8, -4), (-8, 4), (9, -4), (-9, 4), (10, -4), (-10, 4),
    (5, -3), (-5, 3), (6, -3), (-6, 3), (7, -3), (-7, 3), (8, -3), (-8, 3),
    (3, -2), (-3, 2), (4, -2), (-4, 2), (5, -2), (-5, 2), (6, -2), (-6, 2),
    (0, -1), (0, 1), (1, -1), (-1, 1), (2, -1), (-2, 1), (3, -1), (-3, 1),
    (4, -1), (-4, 1), (-2, 0), (2, 0), (-1, 0), (1, 0), (0, 0)]
    actualVectors = enumerate ([[1,2],[3,4]])
    expectedVectors2 = [tuple(u) for u in expectedVectors]
    actualVectors2 = [tuple(w) for w in actualVectors]
    if set(expectedVectors2) != set(actualVectors2):
        print "Error in test_enumerate()"

def prod_vect(B,u,Bk,verbose=False):
    """
    Computes a list of dot products of u and bi in Bk.
    u - a vector
    Bk - a list of (column) vectors
    """
    inner = ()
    for bi in Bk:
```

```
        u_bi = dot(B,u,bi)
        if verbose: print "vector products:",u,bi,u_bi
        inner = inner + (u_bi,)

    if verbose: print "inner products:",inner
    return inner

def test1_prodVect():
        expectedList = (-10,10,-6)
        actualList = prod_vect    ([[1,2],[3,4]],[-1,1],[[-4,1],[-3,2],[-8,3]])
        if expectedList != actualList:
                print "Error in test1_prodVect"
                print expectedList,actualList

#--Dot Product
    ----------------------------------------
def dot(B,u,v,verbose=False):
        """
        B - list of basis columns
        u,v - coordinate vectors
        """
        if verbose:
                print "Inside dot product",u,v
                print "Inside dot product",vector(u)
                print "Inside dot product",matrix(B).transpose()
        u1 = matrix(B).transpose()*vector(u)
```

```
        if verbose: print "Inside dot product",u1
        v1 = matrix(B).transpose()*vector(v)
        if verbose: print "Inside dot product",v1
        return u1.dot_product(v1)


def test_dot():
        expectedDot = 7
        actualDot = dot   ([[1,0],[0,1]],[3,1],[1,4])
        if expectedDot != actualDot:
                print "Error in test_dot"



def swap(myList,i,j):
        temp = myList[i]
        myList[i] = myList[j]
        myList[j] = temp
        return myList


def numExt(k,S,inds,B,GramMat,k_partial=None,verbose=False):
        """
    Computes the number of ext/continuations from a k- to (k+1)-partial
        automorphism.
    k - length of the partial automorphism, number of vectors to check against
        S - set of candidates for continuation
        inds - set of indices
    B - list of basis vectors (columns)
```

```
    GramMat - Gram matrix corresponding to the basis B
        k_partial - set of vectors (k-partial automorphism)
    """
    ## If no k-partial automorphism is given, use the coordinate basis as a control
    if k_partial == None:
            k_partial = matrix.identity(len(B)).rows()

    if k == len(B):
            print "Last possible index."

    if verbose: print "Indices given:",inds
    ext = 0

    ## Test each candidate vector
    for u in S:
            if verbose: print "Trying to extend by",u
            ## Initialize decision variable
            toAdd = 0

            ## 1st Test: Does the norm (squared) match?
            if dot(B,u,u)== GramMat[k][k]:
                    if verbose: print "Inside numExt",k,k_partial,u,GramMat[k][k]
                    toAdd = 1
                    for j in inds:
                            if verbose: print "Inside numExt",k,inds,k_partial,u,dot
                                (B,u,k_partial[j]),GramMat[k][j]
```

```
                    ## 2nd Test: Do the inner products match?
                    if (j != k) and (dot(B,u,k_partial[j]) != GramMat[k][j
                        ]):
                            if verbose: print "Inside numExt dot product
                                (u,bk) does not match for",k,k_partial[j],
                                dot(B,u,k_partial[j]), GramMat[k][j]
                            toAdd = 0
                            break
                    if verbose: print "uk:",dot(B,u,k_partial[j]) ,GramMat[
                        k][j]

            else:
                    if verbose: print "dot (u,u) product did not match",dot(B,
                        u,u),GramMat[k][k]
            if toAdd:
                    ext += 1
                    if verbose: print "vector found:",u
        return ext

def test1_numExt(verbose=False):
        smallVectors = [(0, 0, -2), (0, 0, 2), (0, 1, -2), (0, -1, 2),
        (0, 2, -2), (0, -2, 2), (0, 3, -2), (0, -3, 2),
        (0, 4, -2), (0, -4, 2), (0, -1, -1), (0, 1, 1),
        (0, 0, -1), (0, 0, 1), (0, 1, -1), (0, -1, 1),
        (0, 2, -1), (0, -2, 1), (0, 3, -1), (0, -3, 1),
        (0, -2, 0), (0, 2, 0), (0, -1, 0), (0, 1, 0),
```

```
        (-1, 0, 0), (1, 0, 0), (0, 0, 0)]

        indicesToCompare = []
        basisCols =    [[4,0,0],[0,1,1],[0,0,2]]
        basisMatrix = matrix(basisCols).transpose()
        gramMatrix = basisMatrix.transpose()*basisMatrix

        correct = [6,4,4]

        for i in range(len(basisCols)):
            whichToReplace = i
            N = numExt(whichToReplace,smallVectors,indicesToCompare,basisCols,
                gramMatrix,None,verbose)
            if N != correct[i]:
                print "Error! test1_numExt",N,correct[i]


def nonzeroMin(fList):
        return min([ x for x in fList if x != 0 ])

def rowOfFingerprint(B,S,inds=[],verbose=False):
        """
        B - list of basis vectors (columns)
        inds - a list of indices [j1 ,..., j(i-1)], typically [1,..., i-1]
        S - set of candidates
        """
```

```
        ## Compute Gram matrix from column-basis
        F = gramMat(matrix(B).transpose())

        if verbose: print "Inside fingerprint",inds
        ## Initialize i-th row of fingerprint to all 0s
        f = [ 0 for i in B ]

        ## Compute the number of appropriate vectors
        for k in range(len(B)):
                if k not in inds:
                        if verbose:
                                print "fingerprint so far...",f
                        f[k] = numExt(k,S,inds,B,F,None,verbose)
                        if verbose: print "Entry for fingerprint:",f[k]

        ## Find the index of the smallest nonzero entry
        f_min = nonzeroMin(f)
        inds.append(f.index(f_min))
        return (f,inds)

def fingerprint (B,S,verbose=False):
        """
        B - list of basis vectors (columns)
        S - set of candidates
        """
        Inds = []
```

```
    f_rows = []

    for i in range(len(B)):
        fi ,Inds = rowOfFingerprint(B,S,Inds,verbose)
        f_rows.append(fi)
        if i != Inds[-1]:
            if verbose: print "finerprint:\n",matrix(f_rows)
            if verbose: print "Reordering basis...",i,Inds[-1]
            for row in f_rows:
                row = swap(row,i,Inds[-1])

            B = swap(B,i,Inds[-1])
            for v in S:
                if verbose: print v,i ,Inds[-1]
                v = swap(v,i,Inds[-1])
                if verbose: print v

            Inds[-1] = i

    finMat = matrix(f_rows)
    return B,S,Inds,finMat

def test_fingerprint ():
    expectedFin = ([4,6],[0])
    smallVectors = [(10, -5), (-10, 5), (11, -5), (-11, 5), (12, -5), (-12, 5),
    (8, -4), (-8, 4), (9, -4), (-9, 4), (10, -4), (-10, 4),
```

```
        (5, −3), (−5, 3), (6, −3), (−6, 3), (7, −3), (−7, 3), (8, −3), (−8, 3),
        (3, −2), (−3, 2), (4, −2), (−4, 2), (5, −2), (−5, 2), (6, −2), (−6, 2),
        (0, −1), (0, 1), (1, −1), (−1, 1), (2, −1), (−2, 1), (3, −1), (−3, 1),
        (4, −1), (−4, 1), (−2, 0), (2, 0), (−1, 0), (1, 0), (0, 0)]
    actualFin = fingerprint ([[1,2],[3,4]], smallVectors)
    if expectedFin[0] != list (actualFin [3][0]) :
            print "Error in test_fingerprint"


def innerProductTable(B,S,Bk,verbose=False):
    """
    B − list of basis vectors (columns)
    S − set of candidates
    Bk − partial basis or images of basis vectors
    """
  lookup = {}
  for u in S:
    inner = prod_vect(B,u,Bk)
    if inner in lookup.keys():          # If an entry is present ...
      if verbose: print "Found a duplicate:",u,inner,lookup[inner]
      lookup[inner].append(vector(u))   # update it!
    else:
      if verbose: print "New entry:",inner,u
      lookup[inner] = [vector(u)]       # Otherwise create a new entry
  return lookup
```

```
def test_innerProductTable():
        smallVectors = [(10, −5), (−10, 5), (11, −5), (−11, 5), (12, −5), (−12, 5),
        (8, −4), (−8, 4), (9, −4), (−9, 4), (10, −4), (−10, 4),
        (5, −3), (−5, 3), (6, −3), (−6, 3), (7, −3), (−7, 3), (8, −3), (−8, 3),
        (3, −2), (−3, 2), (4, −2), (−4, 2), (5, −2), (−5, 2), (6, −2), (−6, 2),
        (0, −1), (0, 1), (1, −1), (−1, 1), (2, −1), (−2, 1), (3, −1), (−3, 1),
        (4, −1), (−4, 1), (−2, 0), (2, 0), (−1, 0), (1, 0), (0, 0)]
        actualTable = innerProductTable ([[1,2],[3,4]], smallVectors ,[[1,0]])
        Table = {(−6,): [(−10, 4), (1, −1)], (−4,): [(8, −4), (−3, 1)],
        (−1,): [(−9, 4), (2, −1)], (9,): [(4, −1)], (11,): [(0, 1)],
        (−10,): [(−2, 0)], (1,): [(9, −4), (−2, 1)], (3,): [(−6, 3), (5, −2)],
        (5,): [(−10, 5), (12, −5), (1, 0)], (7,): [(8, −3), (−3, 2)],
        (−7,): [(−8, 3), (3, −2)], (−5,): [(10, −5), (−12, 5), (−1, 0)],
        (−3,): [(6, −3), (−5, 2)], (8,): [(−5, 3), (6, −2)], (−2,): [(−7, 3),
        (4, −2)], (10,): [(2, 0)], (−11,): [(0, −1)], (0,): [(11, −5), (−11, 5), (0, 0)
            ],
        (−9,): [(−4, 1)], (2,): [(7, −3), (−4, 2)], (4,): [(−8, 4), (3, −1)],
        (6,): [(10, −4), (−1, 1)], (−8,): [(5, −3), (−6, 2)]}
        expectedTable = {}
        for i in Table.keys():
                expectedTable[i] = []
                for j in Table[i]:
                        toAppend = vector(j)
                        expectedTable[i].append(toAppend)
        if expectedTable != actualTable:
                print "Error in test_innerProductTable"
```

```
            for i in actualTable:
                if expectedTable[i] != actualTable[i]:
                    print i,expectedTable[i],actualTable[i]


def vectorSums(B,S,k,exten=None,verbose=False,partial=False,depth=None):
    """
    B - a set of vectors (either the basis or automorphism)
    S - small vectors
    k - a maximum index
    depth - depth (previous steps to repeat)
    OUTPUT
    W - matrix of vector sums
    A - reduced basis for W
    X - coordinate vector WX = A
    Y - coordinate vector W = AY
    """
    ## lookupInner - lookup table of vectors and inner product lists ##
    ## inner - list of inner products of candidate vector with basis ##
    ## startIndex - which basis vector to start with for comparison ##

    if verbose: print "Vector sums, basis:\n",B
    if depth==None:
        startIndex = 0
    else:
            startIndex = max(0,k+1-depth)
```

```
if exten == None:
    Bk = matrix.identity(matrix(B).rank())[startIndex:k]
else:
    Bk = exten

if verbose: print "Inside vectorSums",startIndex,k,"\n",Bk
## Initilize lookup table of inner products
lookupInner = innerProductTable(B,S,Bk)
lookupkeys = lookupInner.keys()
lookupkeys.sort()

## Initialize list of vector sums
W = []

## Create vector sums as columns in a list
for matches in lookupkeys:
        if verbose: print "Vector Sums, before summing",lookupInner[
            matches]
        W.append(sum(lookupInner[matches]))
        if verbose: print "Vector Sums, after summing",sum(lookupInner[
            matches])

## Create a row-matrix of the list of vector sums
W = matrix(W)
if verbose: print "Index:",k,"\nVector Sums:\n",W
```

```
    ## If checking a k−partial automorphism, nothing further is necessary
    if partial: return W



    ## Prepare for LLL
    A, X = matrix(ZZ,W).hermite_form(transformation=True, include_zero_rows=False)

    if verbose: print "Reduced Basis:\n",A

    Y = A.solve_left(W)
    if verbose: print "Y",Y
    if verbose: print "X",X

    return (W,A,X,Y)

def condition1(num,f_k,verbose=False):
        """
        num − number of extensions
        f_k − entry in fingerprint
        """
    if num != f_k:
        if num < f_k:
            if verbose: print "Fewer extensions were found; backtrack."
        if verbose: print num,f_k,"The number of extensions did not match the
            fingerprint."
        return False
```

```
    if num == 0:
        if verbose: print "Could not extend to automorphism."
        return False
    return True


def test_cond1():
        expected = False
        actual = condition1(6,7)
        if expected!= actual:
                print "Error in test_cond1"


def condition2(B1st,B2nd,A,A2):
        """
        B1 - first list of basis vectors (columns)
        B2 - second list of basis vectors (columns)
        A - reduced vector sum matrix
        A2 - second reduced vector sum matrix
        """
        ## Check the scalar products of A2 against A
    prod_A2 = []
    for vect1 in A2.rows():
                for vect2 in A2.rows():
                        prod_A2.append(dot(B2nd,vect1,vect2))



    prod_A = []
```

```
    for vect1 in A.rows():
                for vect2 in A.rows():
                        prod_A.append(dot(B1st,vect1,vect2))


    if prod_A2 != prod_A:
        return False
    return True


def test_cond2():
        expected = True
        A1st = matrix ([[1,1],[2,-2]])
        A2nd = matrix([[1,1],[2,-2]])
        actual = condition2    ([[1,2],[3,4]],[[1,2],[3,4]],    A1st,A2nd)
        if expected!= actual:
                print "Error in test_cond2"


def condition3(A2,Y,W2):
        """
        A2 - second reduced vector sum matrix
        Y - transformation matrix YA = W
        W2 - second unreduced vector sum matrix
        """
    W3 = Y*A2
    W2 = set([tuple(w) for w in W2.rows()])
    W3 = set([tuple(w) for w in W3.rows()])
    if not W3.issubset(W2):
```

```
            return False
        return True


def test_cond3():
            expected = True
            Y1st = matrix ([[1,0],[0,1]])
            W2nd = matrix([[1,1],[2,-2]])
            A2nd = matrix([[1,1],[2,-2]])
            actual = condition3(A2nd,Y1st,W2nd)
            if expected!= actual:
                    print "Error in test_cond3"


def meetsCriteria(B1st,B2nd,k_partial,v,Vect_Sums,S,S2nd,f,inds,verbose=False):
            """
            B1 - first list of basis vectors (columns)
            B2 - second list of basis vectors (columns)
            k_partial - k-partial automorphism
            v - candidate for image of b(k+1)
            Vect_Sums - vector sum matrices
            S - first set of small vectors
            S2 - second set of small vectors
            f - fingerprint
            inds - set of indices (for fingerprint)
            """
            k = len(k_partial)
            v_ext = k_partial + [v]
```

```
GramMat1st = gramMat(matrix(B1st).transpose())
num = 0
if len(B1st)!=k+1:
        num = numExt(k+1,S2nd,inds[:k+1],B2nd,GramMat1st,v_ext)
        if verbose: print "Found",num,"extensions for", k_partial+[v]

#print "Inside meetsCriteria, numExt gives:",num,f[k+1][k+1],k
if dot(B2nd,v,v) != vector(B1st[k]).dot_product(vector(B1st[k])):
        if verbose: print "Inner product of candidate did not match
            basis."
        return False
if (len(B1st)==k+1) or condition1(num,f[k+1][k+1]):
        W2 = vectorSums(B2nd,S2nd,k+1,exten=v_ext,partial=True)
        try:
                X = Vect_Sums[k][2]
                A2 = X*W2
        except TypeError:
                if verbose: print "Vector Sums of candidate not
                    compatible with original vector."
                return False
        A = Vect_Sums[k][1]
        if condition2(B1st,B2nd,A,A2):
                Y = Vect_Sums[k][3]
                if condition3(A2,Y,W2):
                        if verbose: print "Extension approved",v
                        return True
```

```
                    else:
                        print "Failed condition 3",k_partial+[v]
                else:
                    print "Failed condition 2",k_partial+[v]
        else:
            print "Failed condition 1",num,f[k+1][k+1],k_partial+[v]
        return False


def Search(B1st,B2nd,k_partial,Candidates,Vect_Sums,S,S2nd,f,inds,randomized=False,
    verbose=False):
    """
        B1 - first list of basis vectors (columns)
        B2 - second list of basis vectors (columns)
        k_partial - k-partial automorphism
        Candidates - set of candidates to search for an extension
        Vect_Sums - vector sum matrices
        S - first set of small vectors
        S2 - second set of small vectors
        f - fingerprint
        inds - set of indices (for fingerprint)
    Note: Vect_Sums,S,f,inds - Needed to pass on to numExt and vect_sums
    OUTPUT
    Either a map [v1,..., vn] or False
    """
    if verbose: print "\n",matrix(k_partial)
    if len(k_partial) == len(B1st):
```

```
        print "Isometry found."
        return k_partial

## Initialize set of candidates, not "linked" to original set
C = []
C = C+Candidates

## Initialize set of candidates to pass along
C_next = []
C_next += Candidates

## Randomize to obtain a different isometry (if possible)
if randomized: random.shuffle(C)

## Exhaustively check combinatons until no candidates remain
while len(C) != 0:
    if verbose: print len(k_partial),"Candidates:",C
    v = C.pop()
    if meetsCriteria(B1st,B2nd,k_partial,v,Vect_Sums,S,S2nd,f,inds,verbose):
        if verbose: print "Candidate approved\t",k_partial+[v]
        C_next.remove(v)
        map = Search(B1st,B2nd,k_partial+[v],C_next,Vect_Sums,S,S2nd,f,inds,
            randomized)
        if verbose: print "backtracking...\n"
        if map == -1:
            C_next.append(v)
```

```
                continue
            return map
        else:
            if verbose: print "Candidate failed\t",k_partial+[v]
    return -1



def is_isometric (LBas,JBas,verbose=False,randomized=False):
        """
        LBas,JBas = two lists of (column) basis vectors
        NOTE: These cannot be from the same object, instantiate separately.
        OUTPUT
        Either an image of LBas in J [v1 ... vn] or -1 (no map found)
        """
        B = LBas
        B2nd = JBas
        F = gramMat(matrix(B).transpose())
        G = gramMat(matrix(B2nd).transpose())
        if randomized: import random
        ## Verify the inner products are the same
        if verbose: print "\nVerifying scalar products of J..."
        Inner_prod = set([])
        Inner_prod2 = set([])
        for bi in matrix.identity(len(B)):
                Inner_prod.add(dot(B,bi,bi))
                Inner_prod2.add(dot(B2nd,bi,bi))
```

```
if Inner_prod2 != Inner_prod:
        return -1


F_max = max(F.diagonal())
S = enumerate(B,F_max) ### Uses rows of B to reconstruct matrix
                                    ### Assumes matrix has column-
                                        basis
print "\n\nEnumerated short vectors of first lattice."
if verbose: print S
if verbose: print "Number of candidates:",len(S)


## Calculate fingerprint of L
if verbose: print "\nCalculating fingerprint of L..."
B,S,inds,fingerprintMat = fingerprint(B,S)
F = gramMat(matrix(B).transpose())
if verbose: print "\nFingerprint:\n",fingerprintMat,"\n",Inds
print "Fingerprint calculated."


aut_bound = prod([ fingerprintMat[i][inds[i]] for i in range(len(B)) ])
if verbose: print "We estimate Aut(L) <= %s"%(aut_bound)


## Calculate vector sums of L
if verbose: print "\nCalculating vector sums of L..."
Vect_Sums = []
for k in range(matrix(B).rank()):
        (W,A,X,Y) = vectorSums(B,S,k+1,exten=None,verbose=verbose)
```

```
                Vect_Sums.append((W,A,X,Y))
        print "Vector sums calculated."

        ## Search for an isometry using the basis of J as candidates,
        ## and S from L to measure extensions
        if verbose: print "Our second basis:\n",B2nd
        S2nd = enumerate(B2nd,F_max)
        #print S2nd
        print "Enumerated short vectors of second lattice."
        if len(S) != len(S2nd):
                print "Candidate sets have different sizes:",len(S),len(S2)
                return -1

        print "Searching for an isometry..."
        return Search(B,B2nd,[],S2nd,Vect_Sums,S,S2nd,fingerprintMat,inds,randomized,
            verbose)



def orbitCalc(vectors, group):
    """
        vectors - set of vectors to operate on
        group - group of maps to use
    """
        orbit = []
        for v in vectors:
                for h in group:
```

```
                orbit.append(h*vector(v))
        return list(set([tuple(w) for w in orbit])) ## Removes possible duplicates


def stabCandidates(k,S,B,GramMat,verbose=False):
    """
        k - current index
        S - set of small vectors
        B - list of (column) basis vectors
        GramMat - Gram matrix
    """
        k_partial = matrix.identity(len(B)).rows()
        candidates = []

        ## Test each candidate vector
        for u in S:
                if verbose: print "Trying to extend by",u
                ## Initialize decision variable
                toAdd = 0

                ## 1st Test: Does the norm (squared) match?
                if dot(B,u,u)== GramMat[k][k]:
                        if verbose: print "Inside numExt",k,k_partial,u,GramMat[k][k]
                        toAdd = 1
                        for j in range(k):
                                if verbose: print "Inside numExt",k,inds,k_partial,u,dot
                                    (B,u,k_partial[j]),GramMat[k][j]
```

```
                    ## 2nd Test: Do the inner products match?
                    if (j != k) and (dot(B,u,k_partial[j]) != GramMat[k][j
                        ]):
                            if verbose: print "Inside numExt dot product
                                (u,bk) does not match for",k,k_partial[j],
                                dot(B,u,k_partial[j]), GramMat[k][j]
                            toAdd = 0
                            break
                    if verbose: print "uk:",dot(B,u,k_partial[j]) ,GramMat[
                        k][j]

            else:
                    if verbose: print "dot (u,u) product did not match",dot(B,
                        u,u),GramMat[k][k]
            if toAdd:
                    candidates.append(u)
                    if verbose: print "vector found:",u
    return candidates


def Aut(Bas,verbose=False):
    """
    Bas - list of (column) basis vectors
    OUTPUT
    G - the automorphism group of the lattice
    """
```

```
if verbose: print "Inside Aut:\n",Bas
import random
B = Bas
r = len(B)
F = gramMat(matrix(B).transpose())
F_max = max(F.diagonal())
S = enumerate(B,F_max) ### Uses rows of B to reconstruct matrix
                                    ### Assumes matrix has column-
                                        basis
print "\n\nEnumerated short vectors of first lattice."
if verbose: print S
if verbose: print "Number of candidates:",len(S)

## Calculate fingerprint of L
if verbose: print "\nCalculating fingerprint of L..."
B,S,inds,fingerprintMat = fingerprint(B,S)
## Recomputing Gram Matrix after (possible) reconfiguration of basis
F = gramMat(matrix(B).transpose())
if verbose: print "\nFingerprint:\n",fingerprintMat,"\n",inds
print "Fingerprint calculated."

aut_bound = prod([ fingerprintMat[i][inds[i]] for i in range(r) ])
if verbose: print "We estimate Aut(L) <= %s"%(aut_bound)

## Calculate vector sums of L
if verbose: print "\nCalculating vector sums of L..."
```

```
Vect_Sums = []
for k in range(r):
        (W,A,X,Y) = vectorSums(B,S,k+1,exten=None,verbose=verbose)
        Vect_Sums.append((W,A,X,Y))
print "Vector sums calculated."


## Compute Strong Generating Sets
## Initialize candidates
gens,G = [matrix.identity(r)],[ [matrix.identity(r)] for i in range(r) ]
#H = [ MatrixGroup(G[i]+gens) for i in range(r) ]
H = [ MatrixGroup([ g for g in G[i] if g in gens ]) for i in range(r) ]
#C = [ S for i in range(r) ]
C = []
for i in range(r):
        C.append(stabCandidates(i,S,B,F))
orbits = [ orbitCalc(C[i],H[i]) for i in range(r) ]
orbitBounds = fingerprintMat.diagonal()
#orbitSizes = [ len(orbits[i]) for i in range(r) ]
# Update by: orbitSizes = [ len(orbitCalc(matrix.identity(r)[i],H[i])) for i in
    range(r) ]
orbitSizes = [ 1 for i in range(r) ]
if verbose:
        print "gens\n",gens
        print "G\n",G
        print "H\n",H
        print "orbitBounds\n",orbitBounds
```

```
        print "orbitSizes\n",orbitSizes
        print "C\n",C
        print "orbits\n",orbits


## Choose k
k=0
if verbose: print "\nSearching for generators..."
randomized=True
while k != None:
        if verbose:print "\n\nC[k]\t\t",k,C[k]
        if verbose: print "orbits[k]\t",k,orbits[k]
        v = C[k].pop()
        #v = orbits[k].pop()
        partialBasis = matrix.identity(r).rows()[:k]
        if verbose: print "Trying to complete...\t",partialBasis+[v]
        ## Does [b1,...,b(k-1),v] complete to a valid map?
        map = -1
        if meetsCriteria(B,B,partialBasis,v,Vect_Sums,S,S,fingerprintMat,inds,
            verbose):
                map = Search(B,B,partialBasis+[v],S,Vect_Sums,S,S,
                    fingerprintMat,inds,True,verbose)



        toRemove = orbitCalc([v],H[k])
        if verbose: print "toRemove\t",toRemove
        if map == -1:
```

```
            orbitBounds[k] -= len(toRemove)


    else:
            if verbose: print "map\t\t",map
            gens.append(matrix(map).transpose())
            G[k].append(matrix(map).transpose())
            H[k] = MatrixGroup([ g for g in G[k] if g in gens ])
            if verbose: print "H[k]\t",k,H[k]
            orbitSizes[k] -= len(orbitCalc([matrix.identity(r)[k]], H[k]))
            orbits[k] = orbitCalc(C[k],H[k])
            if verbose: print "Recalcuated orbits\t",orbits[k]


    for hv in toRemove:
            try:
                    C[k].remove(vector(hv))
                    if verbose: print "Removed\t\t",C[k]
            except ValueError:
                    if verbose:
                            print "Failed to remove: not a candidate."
k = None
for i in range(r):
        if orbitSizes[i] < orbitBounds[i]:
                if len(C[i])!=0:
                        k = i
                        continue
```

```
        print "Generators found."
        if verbose: print "Now for the generators..."
        M = MatrixGroup(gens)
        print "Automorphism group constructed."
        if verbose: print "Aut(L) has order", M.order()
        return M


def test_global():
        test_enumerate()
        test_dot()
        test1_prodVect()
        test1_numExt()
         test_fingerprint ()
        test_innerProductTable()
        test_cond1()
        test_cond2()
        test_cond3()
```